GAP

Todd Gaugler

January 30, 2012

Contents

1 Introductory Material **5**
1.0.1 Arrays . . . 5
1.0.2 While and For Loops . . . 6
1.0.3 If/Else/Etc . . . 7
1.0.4 Functions . . . 8
1.0.5 Links . . . 8

2 Finite Fields **11**
2.0.6 Links . . . 17

3 Coxeter Groups **19**
3.0.7 Links . . . 23

4 Tableaus **25**
4.0.8 A [very] Informal Description of the Problem . . . 25
4.1 Result . . . 27
4.1.1 PolyDerive . . . 27
4.1.2 PolyCoeffs . . . 28
4.1.3 Final Code . . . 28
4.1.4 Matrices Corresponding to The Vector-Space Homomorphisms Induced by elements of S_n . . . 39
4.1.5 Links . . . 42

Chapter 1

Introductory Material

The following is the fairly complete summary of the Gap language found on its website, `www.gap-system.org` : GAP is a system for computational discrete algebra, with particular emphasis on Computational Group Theory. GAP provides a programming language, a library of thousands of functions implementing algebraic algorithms written in the GAP language as well as large data libraries of algebraic objects. See also the overview and the description of the mathematical capabilities. GAP is used in research and teaching for studying groups and their representations, rings, vector spaces, algebras, combinatorial structures, and more. The system, including source, is distributed freely. You can study and easily modify or extend it for your special use.

The rest of this chapter will be dedicated to simple introductory material.

1.0.1 Arrays

It turns out that GAP seems to simplify arrays more than a traditional programming language like c++ would: where in c++ one would have to first specify the types of things one is putting into an array, GAP doesn't seem to need that at all. One traditionally would create a 1-dimensional array of integers in the following way:

```
integerArray:= [1,2,3,4,5] ;
[ 1, 2, 3, 4, 5 ]
integerArray[1];
1
integerArray[5];
5
```

Notice that when accessing the first member of the array, one uses [1] as opposed to [0], like in c++. Also notice that ':=' is used equivalently to '=' in c++. It also turns out that GAP allows you to store 'objects' of different types in the same array: for example, we store an indeterminate object named 'x' alongside an integer, '2' in the array 'a'.

```
x:=Indeterminate(Rationals, "x");
x
a:=[x, 2];
[ x, 2 ]
```

While this one works, it's pretty obvious that this might not ALWAYS work- however, I haven't found any documentation to the contrary, and it's worked with quite a few experiments (e.g., a matrix, a polynomial, and integer, and a variable all seem to be allowed to be put into one array).

One can also use '..' to avoid having to write out all the members of particular arrays. For example,

```
a:=[1..10];  # This is the same as saying
[ 1 .. 10 ]     #a:= [ 1,2,3,4,5,6,7,8,9,10];
a[4];
4
```

1.0.2 While and For Loops

As we all know, being able to run through set of data is something one would like to be able to do. Here we define the syntax for while and for loops, and examples are provided in later sections.

While loops are fairly simple to implement–they look like the following:

```
while [boolean-expression] do
[statements]
od;
```

For example,

```
a:= 2;
counter:=1;
while a<5 do
counter:= counter+1;
a:=a+1;
od;
```

Similarly, for loops look as follows:

```
for [simple-variable in some list]   do
[statements]
 od;
```

For example, this was taken directly from the GAP documentation:

```
gap> s := 0;;
gap> for i in [1..100] do
>    s := s + i;
> od;
gap> s;
5050
```

1.0.3 If/Else/Etc

The GAP documentation defines the if statement in the following way:

```
if [boolean expression 1] then
[statements1]
elif [boolean expression 2] then
[statements2 ]
 else
[statements3 ]
fi;
```

Which consists of an 'if' statement, an 'else if' statement by way of 'elif', and an 'else' statement. A concrete example is below:

```
gap> i := 10;;
gap> if 0 < i then
>    s := 1;
>  elif i < 0 then
>    s := -1;
>  else
>    s := 0;
>  fi;
```

1.0.4 Functions

In defining functions, it is easier to give a simple example in trying to explain their syntax. Let us look at the following function that prints 'hello world':

```
sayhello:= function(); #This says that 'hello world'
Print("Hello world. \n");   #is a function that takes no input
end;
function(  ) ... end
sayhello(); #<-- this is the call of the function after we defined it
Hello world.
```

Functions have the following syntax:

```
function( arguments )
[statements]
end;
```

The following example code calculates the sign of a number n, which the function takes as input:

```
gap> sign:= function(n)
>        if n < 0 then
>           return -1;
>        elif n = 0 then
>           return 0;
>        else
>           return 1;
>        fi;
>    end;
function( n ) ... end
gap> sign(0); sign(-99); sign(11);
0
-1
1
```

1.0.5 Links

- `http://www.gap-system.org/Manuals/doc/htm/ref/CHAP004.htm` is a good overview of most of the basics in Gap. If I left something basic out, it's almost certainly in here.

- `http://www.gap-system.org/Manuals/doc/htm/tut/CHAP004.htm` has a nice discussion of functions in Gap

Chapter 2

Finite Fields

Before really getting into a discussion of how Gap handles Finite Fields, it's worth mentioning that `http://www.math.colostate.edu/~hulpke/CGT/education.html` has a great installer for Windows that includes GGAP, a really convenient graphical interface that cooperates with Gap. There are natrually other ways to get GGAP, but this seemed to be by far the most simple and convenient.

We would like to talk about some of the simplest applications of GAP to finite fields. For the most part, I'll skip talking about the general structure of coding in Gap, since it's mostly familiar to anyone thats ever used a programming language, and I'll leave a link to a page discussing some of the basics.

The first natural step to take when talking about finite fields is to tell Gap to construct a field of some order p^n where p is prime and n is a natural number. This is done through the assignment operator, ':=', and the function $Z(p, n)$ or $Z(p^n)$ constructs a field of order p^n.

```
gap> K:= Z(8);
Z(2^3)
```

Before continuing, we have to make the following observations about Gap:

1. $Z(p^n)$ returns a generator of the finite field. So, one could cycle through all the elements of a particular field by taking $Z(p^n)^i$ for different values of i.
2. The additive neutral (identity) element is $0 * Z(p^n)$.
3. The multiplicative identity element is $Z(p^n)^0$

So, in locating values of $x \in K$ such that $x^3 + x + 1 = 0$, we can cycle through the elements of K, put them in the form of this polynomial, and test to see if the resulting element is equal to $0 * Z(p^n)$. This can be done with the following syntax:

```
gap> K:= Z(8);
Z(2^3)
```

```
gap> c:=0;
>0
gap> while c<9 do
gap> if (K^c)^3 + K^c +1 = 0*Z(8) then
gap> Print("K^"); Print(c); Print("\n");
gap> fi;
gap> c:=c+1;
gap> od;
>K^1
>K^2
>K^4
>K^8
```

Notice that the above code gives us a printout of the elements of K that satisfy our polynomial, and are all in terms of K raised to some power i. Taking the elements that satisfy our polynomial, for each polynomial $ax^2 + bx + c$ where a, b and c are all nonzero and are chosen from $\{0, 1\}$, we would like to find the smallest power of y that is '1'.

Given a general element k in K, the function $Order(k)$ returns the smallest power i such that $k^i = Z(p^n)^0$. So, writing an element $k \in K$ as a polynomial function of the generator of K, the same concept applies. In other words, to find the smallest powers of y that is 1, we can tell gap to print out the orders of all such polynomial functions of the elements of K that satisfy our first polynomial. While there is probably a way to do with using nested for loops, the structure of local variables in Gap makes writing nested for loops tricky, and its easier to simply manually input all 7 functions. Notice that we have the following polynomials:

$$\begin{aligned} x^2 + x + 1 \\ x^2 + x \\ x^2 + 1 \\ x + 1 \\ x^2 \\ x \\ 1 \end{aligned}$$

So, in finding the orders of the elements Gaps previously printed out for us under these polynomial functions, we can do the following:

```
gap> Order( (K^1)^2 + (K^1) + 1 ) ; Order ( (K^1)^2 + (K^1) );
Order((K^1)^2+ 1);  Order((K^1)^2) ; Order( K^1 +1 );
Order(K^1); Order(1);
>7
>7
>7
```

```
>7
>7
>7
>1
```

And similar things can be done for the other elements that were printed out by Gap. Completing these exercises for fields of order 16, 32, and 64 is the same idea, except the original polynomials have to be changed. I computed that already, and you have a copy of that code in an email.

Just as a nice example of Gap's finite field capabilities, I ran some calculations with the finite field of order 7. We have the following:

$$\mathbb{Z}/7\mathbb{Z} = \{\overline{0}, \overline{1}, \overline{2}, \overline{3}, \overline{4}, \overline{5}, \overline{6}\}$$

Element	Order
$\overline{0}$	0
$\overline{1}$	1
$\overline{2}$	3
$\overline{3}$	6
$\overline{4}$	3
$\overline{5}$	6
$\overline{6}$	2

These numbers can be verified with the following code:

```
gap> A:= Z(7);
Z(7)
gap> Order(A^0);
1
gap>Order(A);
6
gap>Order(A^2);
3
gap>Order(A^3);
2
gap>Order(A^4);
3
gap>Order(A^5);
6
gap>Order(A^6);
1
```

Which makes sense, since under mod 7, $A = 3$, and thus: $A^1 = 3$, $A^2 = 2$, $A^3 = 6$, $A^4 = 4$, $A^5 = 5$, $A^6 = 1$.

We can use the following code to run through all degree 5 polynomials with coefficients in $\{0, 1\}$:

```
c:=1;; K:=Z(64);;
while c<65 do
if (K^c)^9=Z(2)^0 then k:=(K^c); break; fi;
c:=c+1;
od;
Print("c="); k; Print("\n");

a:=0;;
while a<2 do
   b:=0;;
     while b<2 do
        c:=0;;
        while c<2 do
          d:=0;;
             while d<2 do
                 e:=0;;
                    while e<2 do
                        f:=0;;
                         while f<2 do
   if a+b+c+d+e+f=0 then f:=f+1;;
   else
Print("P(x)= ");
Print(a);Print("x^5+");
Print(" ");Print(b);Print("x^4+");
Print(" "); Print(c);Print("x^3+");
Print(" "); Print(d);Print("x^2+");
Print(" ");Print(e);Print("x+");
Print(" "); Print(f); Print("  Order(P(c)= ");
Print(Order(a*k^5+ b*k^4+c*k^3+d*k^2+e*k + f));
Print("\n");
f:=f+1;;
   fi;
od;
e:=e+1;;
od;
d:=d+1;;
od;
c:=c+1;;
od;
b:=b+1;;
od;
a:=a+1;;
od;
#This code has the following output in GGAP:
c=Z(2^6)^7
```

```
P(x)= 0x^5+ 0x^4+ 0x^3+ 0x^2+ 0x+ 1  Order(P(c)= 1
P(x)= 0x^5+ 0x^4+ 0x^3+ 0x^2+ 1x+ 0  Order(P(c)= 9
P(x)= 0x^5+ 0x^4+ 0x^3+ 0x^2+ 1x+ 1  Order(P(c)= 63
P(x)= 0x^5+ 0x^4+ 0x^3+ 1x^2+ 0x+ 0  Order(P(c)= 9
P(x)= 0x^5+ 0x^4+ 0x^3+ 1x^2+ 0x+ 1  Order(P(c)= 63
P(x)= 0x^5+ 0x^4+ 0x^3+ 1x^2+ 1x+ 0  Order(P(c)= 21
P(x)= 0x^5+ 0x^4+ 0x^3+ 1x^2+ 1x+ 1  Order(P(c)= 63
P(x)= 0x^5+ 0x^4+ 1x^3+ 0x^2+ 0x+ 0  Order(P(c)= 3
P(x)= 0x^5+ 0x^4+ 1x^3+ 0x^2+ 0x+ 1  Order(P(c)= 3
P(x)= 0x^5+ 0x^4+ 1x^3+ 0x^2+ 1x+ 0  Order(P(c)= 63
P(x)= 0x^5+ 0x^4+ 1x^3+ 0x^2+ 1x+ 1  Order(P(c)= 63
P(x)= 0x^5+ 0x^4+ 1x^3+ 1x^2+ 0x+ 0  Order(P(c)= 63
P(x)= 0x^5+ 0x^4+ 1x^3+ 1x^2+ 0x+ 1  Order(P(c)= 63
P(x)= 0x^5+ 0x^4+ 1x^3+ 1x^2+ 1x+ 0  Order(P(c)= 63
P(x)= 0x^5+ 0x^4+ 1x^3+ 1x^2+ 1x+ 1  Order(P(c)= 21
P(x)= 0x^5+ 1x^4+ 0x^3+ 0x^2+ 0x+ 0  Order(P(c)= 9
P(x)= 0x^5+ 1x^4+ 0x^3+ 0x^2+ 0x+ 1  Order(P(c)= 63
P(x)= 0x^5+ 1x^4+ 0x^3+ 0x^2+ 1x+ 0  Order(P(c)= 9
P(x)= 0x^5+ 1x^4+ 0x^3+ 0x^2+ 1x+ 1  Order(P(c)= 63
P(x)= 0x^5+ 1x^4+ 0x^3+ 1x^2+ 0x+ 0  Order(P(c)= 21
P(x)= 0x^5+ 1x^4+ 0x^3+ 1x^2+ 0x+ 1  Order(P(c)= 63
P(x)= 0x^5+ 1x^4+ 0x^3+ 1x^2+ 1x+ 0  Order(P(c)= 7
P(x)= 0x^5+ 1x^4+ 0x^3+ 1x^2+ 1x+ 1  Order(P(c)= 7
P(x)= 0x^5+ 1x^4+ 1x^3+ 0x^2+ 0x+ 0  Order(P(c)= 63
P(x)= 0x^5+ 1x^4+ 1x^3+ 0x^2+ 0x+ 1  Order(P(c)= 63
P(x)= 0x^5+ 1x^4+ 1x^3+ 0x^2+ 1x+ 0  Order(P(c)= 63
P(x)= 0x^5+ 1x^4+ 1x^3+ 0x^2+ 1x+ 1  Order(P(c)= 63
P(x)= 0x^5+ 1x^4+ 1x^3+ 1x^2+ 0x+ 0  Order(P(c)= 21
P(x)= 0x^5+ 1x^4+ 1x^3+ 1x^2+ 0x+ 1  Order(P(c)= 63
P(x)= 0x^5+ 1x^4+ 1x^3+ 1x^2+ 1x+ 0  Order(P(c)= 63
P(x)= 0x^5+ 1x^4+ 1x^3+ 1x^2+ 1x+ 1  Order(P(c)= 63
P(x)= 1x^5+ 0x^4+ 0x^3+ 0x^2+ 0x+ 0  Order(P(c)= 9
P(x)= 1x^5+ 0x^4+ 0x^3+ 0x^2+ 0x+ 1  Order(P(c)= 63
P(x)= 1x^5+ 0x^4+ 0x^3+ 0x^2+ 1x+ 0  Order(P(c)= 21
P(x)= 1x^5+ 0x^4+ 0x^3+ 0x^2+ 1x+ 1  Order(P(c)= 63
P(x)= 1x^5+ 0x^4+ 0x^3+ 1x^2+ 0x+ 0  Order(P(c)= 9
P(x)= 1x^5+ 0x^4+ 0x^3+ 1x^2+ 0x+ 1  Order(P(c)= 63
P(x)= 1x^5+ 0x^4+ 0x^3+ 1x^2+ 1x+ 0  Order(P(c)= 7
P(x)= 1x^5+ 0x^4+ 0x^3+ 1x^2+ 1x+ 1  Order(P(c)= 7
P(x)= 1x^5+ 0x^4+ 1x^3+ 0x^2+ 0x+ 0  Order(P(c)= 63
P(x)= 1x^5+ 0x^4+ 1x^3+ 0x^2+ 0x+ 1  Order(P(c)= 63
P(x)= 1x^5+ 0x^4+ 1x^3+ 0x^2+ 1x+ 0  Order(P(c)= 21
P(x)= 1x^5+ 0x^4+ 1x^3+ 0x^2+ 1x+ 1  Order(P(c)= 63
P(x)= 1x^5+ 0x^4+ 1x^3+ 1x^2+ 0x+ 0  Order(P(c)= 63
P(x)= 1x^5+ 0x^4+ 1x^3+ 1x^2+ 0x+ 1  Order(P(c)= 63
```

```
P(x)= 1x^5+ 0x^4+ 1x^3+ 1x^2+ 1x+ 0  Order(P(c)= 63
P(x)= 1x^5+ 0x^4+ 1x^3+ 1x^2+ 1x+ 1  Order(P(c)= 63
P(x)= 1x^5+ 1x^4+ 0x^3+ 0x^2+ 0x+ 0  Order(P(c)= 7
P(x)= 1x^5+ 1x^4+ 0x^3+ 0x^2+ 0x+ 1  Order(P(c)= 7
P(x)= 1x^5+ 1x^4+ 0x^3+ 0x^2+ 1x+ 0  Order(P(c)= 21
P(x)= 1x^5+ 1x^4+ 0x^3+ 0x^2+ 1x+ 1  Order(P(c)= 63
P(x)= 1x^5+ 1x^4+ 0x^3+ 1x^2+ 0x+ 0  Order(P(c)= 21
P(x)= 1x^5+ 1x^4+ 0x^3+ 1x^2+ 0x+ 1  Order(P(c)= 63
P(x)= 1x^5+ 1x^4+ 0x^3+ 1x^2+ 1x+ 0  Order(P(c)= 21
P(x)= 1x^5+ 1x^4+ 0x^3+ 1x^2+ 1x+ 1  Order(P(c)= 63
P(x)= 1x^5+ 1x^4+ 1x^3+ 0x^2+ 0x+ 0  Order(P(c)= 63
P(x)= 1x^5+ 1x^4+ 1x^3+ 0x^2+ 0x+ 1  Order(P(c)= 63
P(x)= 1x^5+ 1x^4+ 1x^3+ 0x^2+ 1x+ 0  Order(P(c)= 63
P(x)= 1x^5+ 1x^4+ 1x^3+ 0x^2+ 1x+ 1  Order(P(c)= 21
P(x)= 1x^5+ 1x^4+ 1x^3+ 1x^2+ 0x+ 0  Order(P(c)= 63
P(x)= 1x^5+ 1x^4+ 1x^3+ 1x^2+ 0x+ 1  Order(P(c)= 21
P(x)= 1x^5+ 1x^4+ 1x^3+ 1x^2+ 1x+ 0  Order(P(c)= 21
P(x)= 1x^5+ 1x^4+ 1x^3+ 1x^2+ 1x+ 1  Order(P(c)= 63
```

Gap also offers the ability to implement user-defined functions. The syntax is as follows:

```
FunctionName:= function(a, b, ..... ,n)
.
.
.
return n;
end;
```

One example from the Gap manual is as follows:

```
gap> sign:= function(n)
  >        if n < 0 then
  >           return -1;
  >        elif n = 0 then
  >           return 0;
  >        else
  >           return 1;
  >        fi;
  >    end;
```

Which returns a value based on the sign of the input n.

I was asked to write a function that takes in a single polynomial object, and outputs its sequence of coefficients viewed as a vector. It turns out that Gap actually offers a nice environment for polynomials, and we can manipulate it as follows to get some nice returns:

```
gap> x:=UnivariatePolynomial(Rationals, [0,1], 1);
gap> P:=4*x^2 + 2*x + 1;
>4*x_1^2+2*x_1+1
gap> CoefficientMatrix:=CoefficientsOfUnivariatePolynomial(P);
>[ 1, 2, 4 ]
```

Simply, we allowed x to be equal to UnivariatePolynomial(Domain, Coefficient Matrix, Invariant). Through manipulations of the object x, we could construct new polynomials that still allowed us to call the function CoefficientsOfUnivariatePolynomial on them. This function happens to return an array-like structure the contains the coefficients of the polynomial. Here, P is our polynomial.

2.0.6 Links

- `http://schmidt.nuigalway.ie/gap/CHAP018.htm` Is the page in the Gap manual regarding finite fields
- `http://www.gap-system.org/search.html` Is the search function on the Gap website
- `http://www.gap-system.org/Manuals/doc/htm/ref/CHAP004.htm` Is a nice overview of some of the syntax behind Gap.

Chapter 3

Coxeter Groups

A Coxeter group is a group generated by a finite set of generators such that each generator squared is the identity, and that there exists some $n \in \mathbb{N}$ such that for any pair of generators, say $a_i a_j$, $(a_i a_j)^n = 1$.

The nicest and easiest way to implement a Coxeter group in Gap is to take a free group with n generators, then to modulo out by a satisfactory number of relations to arrive at a Coxeter group. For example, suppose we took the group $G1$ generated by a, b, c such that $a^2 = b^2 = c^2$, and $(bc)^5 = (cd)^3 = (abc)^5 = 1$. This can be implemented in Gap as follows:

```
gap> G1:=FreeGroup(3);
<free group on the generators [ f1, f2, f3 ]>
gap> a:= G1.1;; b:= G1.2;; c:= G1.3;; #G1.(i) refers to the
gap> relations:= [                              #i'th generator of G1
(a*a), (a*b)^3, (a*c)^2,
       (b*b),   (b*c)^5,
                (c*c),
(a*b*c)^5 ] ;;
gap> C:= G1/relations;
<fp group on the generators [ f1, f2, f3 ]>
gap> Size(C);
> 60
#So, G1 contains 60 unique elements
```

Following this format, we can do the same thing for the following two Coxeter Groups:

- G2: Generated by a, b, c, d with the following presentation:

$$a^2 = b^2 = c^2 = d^2 = (ab)^3 = (bc)^5 = (cd)^3 = (abc)^5 = (bcd)^5 = 1$$

  (all other unmentioned pairs will have order 2)

- G3: Generated by a, b, c, d, e with the following presentation:

$$a^2 = b^2 = c^2 = d^2 = e^2 = (ab)^3 = (bc)^5 = (cd)^3 = (de)^5 = (abc)^5 = 1, \quad (cde)^5 = a$$

(again, all other unmentioned pairs will have order 2)

In finding the size of $G2$ and $G3$, we have the following:

```
gap> G2:= FreeGroup(4);
<free group on the generators [ f1, f2, f3, f4 ]>
gap> a:= G2.1 ;; b:= G2.2;; c:= G2.3;; d:= G2.4;;
gap> relations:= [
a*a, (a*b)^3, (a*c)^2, (a*d)^2,
     (b*b),   (b*c)^5, (b*d)^2,
              (c*c),   (c*d)^3,
                       (d*d),
       (a*b*c)^5, (b*c*d)^5 ] ;;
gap> CoxeterGroup:= G2/relations;
<fp group on the generators [ f1, f2, f3, f4 ]>
gap> Size(CoxeterGroup);
> 660


gap> G3:= FreeGroup(5);
<free group on the generators [ f1, f2, f3, f4, f5 ]>
gap> a:= G3.1;; b:= G3.2;; c:= G3.3;; d:= G3.4;; e:=G3.5;
gap> relations:= [
a^2, (a*b)^3, (a*c)^2, (a*d)^2, (a*e)^2,
      b^2,    (b*c)^5, (b*d)^2, (b*e)^2,
               c^2,    (c*d)^3, (c*e)^2,
                        d^2,    (d*e)^5,
                                  e^2,
(a*b*c)^5, a*(c*d*e)^5] ;
gap> Coxeter:= G3/relations;
<fp group on the generators [ f1, f2, f3, f4, f5 ]>
gap> Size(Coxeter);
> 175560
```

According to the second link in the link-subsection, calling the function Size() uses the Todd-Coxeter algorithm when it tries to figure out the size of the group. The same can be said of the Index() function:

In Group theory, the **index** of a subgroup H in a group G is the "relative size" of H in G: equivalently, the number of 'copies', or cosets of H that fill up G. Gap has a built-in function that uses the Todd-Coxeter algorithm to calculate the index of a subgroup. The syntax is as follows, for a subgroup H of G:

```
gap> Index( G, H );
```

And in looking to create a subgroup U of G, we have the following commands:

```
U:= Subgroup(G, [generators]) ;
IsSubgroup(G,U) #Returns true if U is a subgroup of G
```

We can find the index of G1 in G2 in the following way:

```
gap> G2:= FreeGroup(4);
<free group on the generators [ f1, f2, f3, f4 ]>
gap>  a:= G2.1 ;; b:= G2.2;; c:= G2.3;; d:= G2.4;;
gap> relations:= [
a*a, (a*b)^3, (a*c)^2, (a*d)^2,
     (b*b),   (b*c)^5, (b*d)^2,
              (c*c),   (c*d)^3,
                       (d*d),
       (a*b*c)^5, (b*c*d)^5 ] ;;
gap>  C2:= G2/relations;
<fp group on the generators [ f1, f2, f3, f4 ]>
gap>  C1:= Subgroup(C2, [C2.1, C2.2, C2.3]);
>Group([ f1, f2, f3 ])
gap> IsSubgroup(C2, C1);
>true
gap> Size(C2);
660
Size(C1);
60  #These two Size() Calls were to ensure that we had the right groups
gap> Index(C2, C1);
>11
```

The idea was to look at our Coxeter Group (originally called $G2$) and to compare it to its subgroup $G1$ by calling it the subgroup of G2 whose generators are only the first three generators of $G2$. Similarly, to find the index of $G2$ in $G3$, we have:

```
gap> G3:= FreeGroup(5);
<free group on the generators [ f1, f2, f3, f4, f5 ]>
gap> a:= G3.1;; b:= G3.2;; c:= G3.3;; d:= G3.4;; e:=G3.5;;
gap> relations:= [
a^2, (a*b)^3, (a*c)^2, (a*d)^2, (a*e)^2,
      b^2,    (b*c)^5, (b*d)^2, (b*e)^2,
               c^2,    (c*d)^3, (c*e)^2,
```

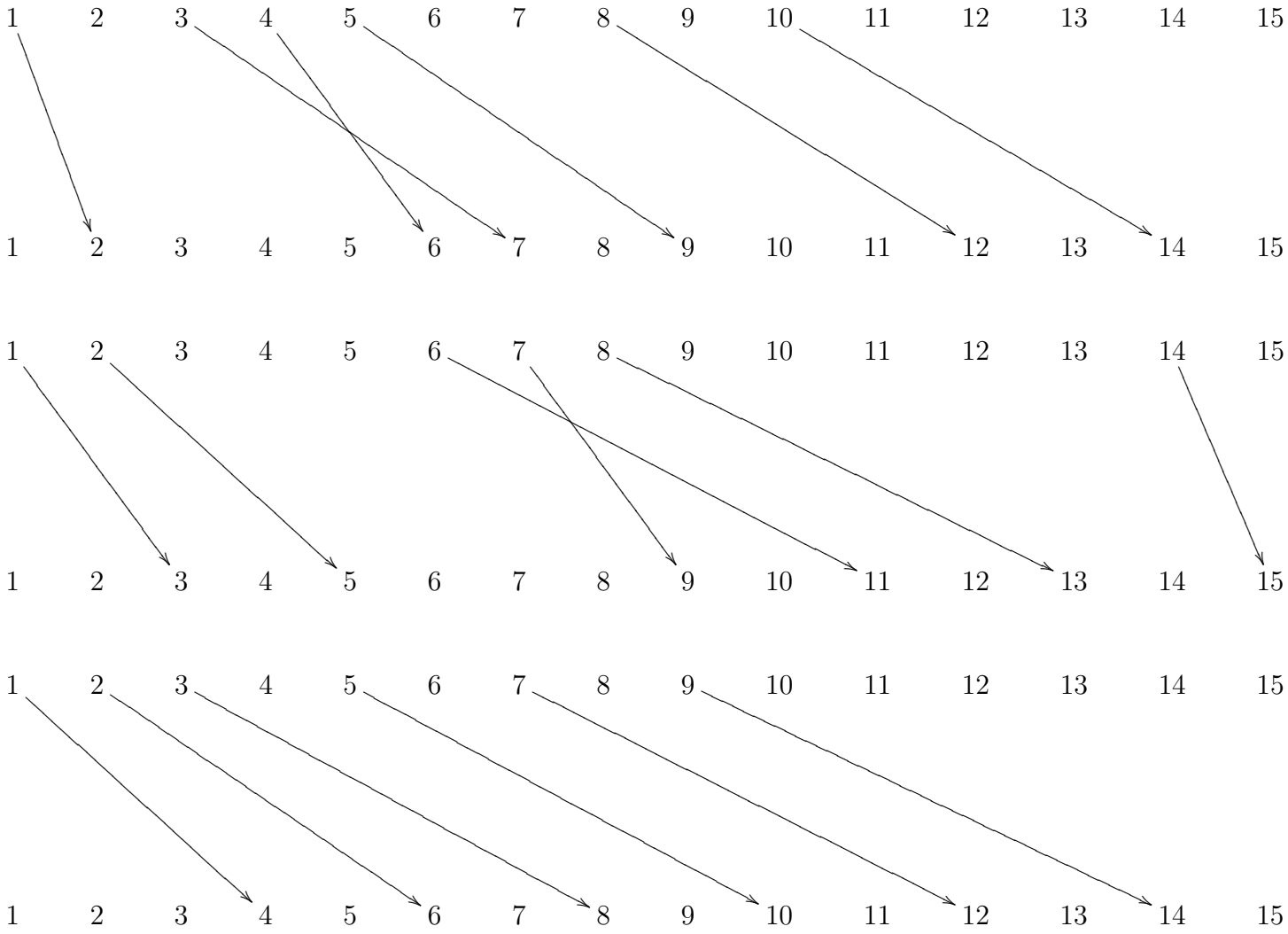
```
                        d^2,    (d*e)^5,
                                  e^2,
(a*b*c)^5, a*(c*d*e)^5] ;;
gap> C3:= G3/relations;
<fp group on the generators [ f1, f2, f3, f4, f5 ]>
gap>C2:= Subgroup(C3, [ C3.1, C3.2, C3.3, C3.4] );
>Group([ f1, f2, f3, f4 ])
gap>Size(C2);
>660
gap>Index(C3, C2);
>266
```

Where again, we take $G3$, take the subgroup of $G3$ spanned only by the first four generators of $G3$ (which is $G2$), then run our Index function which utilizes the Todd-Coxeter algorithm.

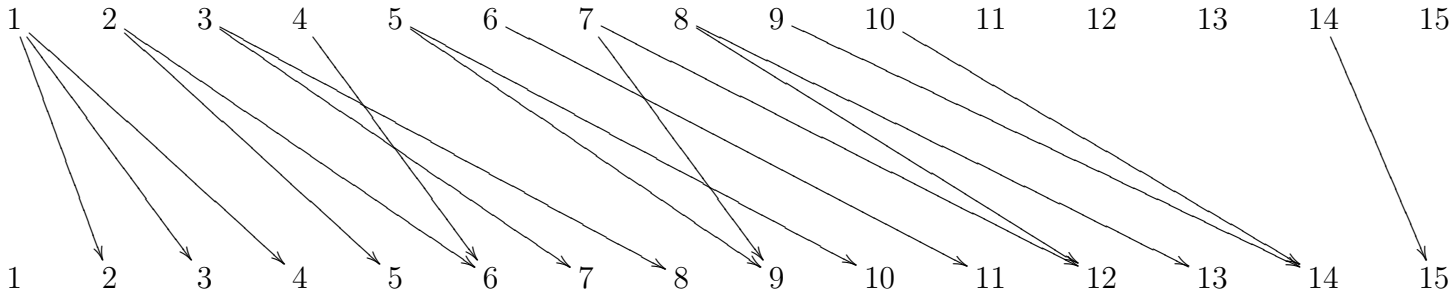
Through the use of the function $FactorCosetAction(G, U)$, one can get permutation representations of our group G based on the representation of some subset U of G. This function operates more successfully based on the size of the group U. While we could try to factor G out by a few relations to find a nice subgroup, we can us the function *SylowSubgroup* to get us a proper subgroup of a large size (We have the following definition of a Sylow Subgroup: a Sylow p-subgroup of a group G is a maximal p-subgroup of G). Letting $G = G1$, we have the following example:

```
gap> G1:=FreeGroup(3);;
gap> a:= G1.1;; b:= G1.2;; c:= G1.3;; #G1.(i) refers to the
gap> relations:= [                              #i'th generator of G1
(a*a), (a*b)^3, (a*c)^2,
       (b*b),   (b*c)^5,
                (c*c),
(a*b*c)^5 ] ;;
gap> C:= G1/relations;
gap> Size(C);
> 60
gap> U:=SylowSubgroup(C,2);
>Group(<fp, no generators known>)
gap> FactorCosetAction(C,U);
>[ f1, f2, f3 ] -> [ (1,2)(3,7)(4,6)(5,9)(8,12)(10,14),
  (1,3)(2,5)(6,11)(7,9)(8,13)(14,15), (1,4)(2,6)(3,8)(5,10)(7,12)(9,14) ]
```

In math 333, we were advised to draw diagrams of permutation groups in the following way:



Overlapping these diagrams, we have the following:



Which unfortunately isn't particularly enlightening to me; I can't pinpoint exactly what group this might be.

3.0.7 Links

- `http://www.gap-system.org/Manuals/doc/htm/ref/CHAP045.htm` Is the best section in the Gap manual about finitely presented groups

- `http://www.gap-system.org/Gap3/Manual3/C001S020.htm` Offers an alternative way to represent Coxeter groups.
- `http://en.wikipedia.org/wiki/Index_of_a_subgroup` Is the wikipedia page I looked at to get the idea of the index of a subgroup.
- `http://en.wikipedia.org/wiki/Sylow_theorems` Is a wikipedia page with information on Sylow Subgroups

Chapter 4

Tableaus

4.0.8 A [very] Informal Description of the Problem

A **Tableau** is a figure that looks roughly like the following: This particular Tableau has 9 cells,

3	6	2	7
9	5	9	
1			
8			

and thus corresponds to a partition of 9, and a specific polynomial:

$$(x_3 - x_9)(x_3 - x_1)(x_3 - x_8)(x_9 - x_1)(x_9 - x_8)(x_1 - x_8)(x_6 - x_5)(x_2 - x_9)(x_7)$$

Which can be derived from the tableau by looking at the integer entries as indices for 9-variables (labeled $x_1, x_2, ..., x_9$), then working from the first column, one can get the above polynomial by following a simple pattern. Tableaus that we wish to talk about have to have the following properties:

- Indices may be repeated across rows
- Indices may **not** be repeated down columns

Since I'm not writing this for a general audience, I'll leave this description as it is for now.

One can now use the symmetric group S_9 and have it act on the indices in the above tableau to rearrange the integers, thus changing the polynomial associated with it. For example, having (1 2)(3 4) act on our tableau above, we get figure 4.1, which has the new polynomial associated with it:

$$(x_4 - x_9)(x_4 - x_2)(x_4 - x_9)(x_9 - x_2)(x_9 - x_8)(x_2 - x_8)(x_6 - x_5)(x_1 - x_9)(x_7)$$

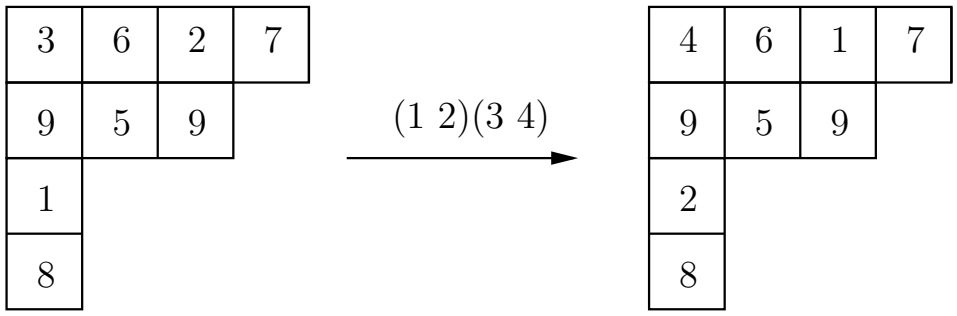


Figure 4.1:

One can now look at the vector space spanned by the polynomials associated with a specific tableau and the actions of a symmetric group on that tableau. The hope is that GAP can help us find the matrices associated with the linear transformations that S_n performs on the n-cell tableau as it takes polynomials to other polynomials, help us find the trace of those matrices, and help us find the dimension of the vector space associated with a specific tableau.

The first order of business that we need to talk about is representing the n-variable polynomials as a vector. Unfortunately GAP4 doesn't have fantastic support for multivariate polynomials (GAP3 did, but still didn't really have what we were looking for). However, it appears we can continue the following line of thought in GAP4: take a one variable polynomial, say of degree 2 just for simplicity's sake. It would look like this:

$$f(x_1) = ax_1^2 + bx_1 + c$$

We can represent it's coefficients a, b, c as a matrix in terms of f in the following way:

$$[a, b, c] = [f^{(2)}(0)/2!, f^{(2)}(0)/1!, f(0)]$$

Even more generally, for a polynomial of degree n, we would have:

$$[\text{... coefficient behind } x_1^n ...] = [....f^{(n)}(0)/n!...]$$

It turns out that adding another variable won't really affect hurt this process, it just makes the matrix larger; we can use the other variable to 'index' the other:

$$f(x_2, x_1) = (ax_1^2 + bx_1 + c) + x_2(dx_1^2 + ex_1 + f) + x_2^2(gx_1^2 + hx_1 + i)$$

Which is really just 3 new and separate cases of what we had above, only now we need to be more intelligent about how we get each case (it's really almost the same thing):

$$[a, b, c, d, e, f, g, h, i] = [f(0,0)/(\partial x_1)^2 2!, f(0,0)/(\partial x_1)1!, f(0,0)/0!, ..., f(0,0)/(\partial x_2)^2 2!]$$

And naturally, this idea then continues to adding a third variable, where now this third variable indexes what is already indexed by variable x_2 (while not indicated, it is worth mentioning that getting 'g' could be done by taking $f(0,0)/(\partial x_1)^2(\partial x_2)^2 \cdot 2!^2$). The issue here would be to implement this idea in Gap- one would need to able to do the following:

- Take derivatives of multi-variable polynomials

- Evaluate values of these polynomials for specific values of x_i.

Both of these should be do-able, see the links section as to where I found documentation on both.

The next issue would be to figure out a way the symmetric group could act on a Tableau. One way to do this would be to use the OnIndeterminates function that GAP offers, that can do things like the following:

```
OnIndeterminates(x^7*y+x*y^4,(1,17)(2,28));
x_17^7*x_28+x_17*x_28^4
```

Which takes a multivariate polynomial, and permutes the variables by using an element of a symmetric group. So, one idea would be to take the polynomial associated with a specific Tableau, run through the elements of the symmetric group S_n, and have OnIndeterminates run through all the polynomials associated with that shape of tableau, and to print out all the vectors of coefficients.

So, now in summary, we should be able to use GAP to find all the coefficients associated with the polynomials coming from some tableau and a symmetric group's action in it. The biggest issue now is calculating things regarding the vector space these polynomials form- for example, in finding the dimension of the vector space, it is difficult to find in the GAP documentation the proper commands to use with the data we collected.

4.1 Result

After quite a bit of trial and error along with digging deeply into GAP's manual, I put something together that seems to do what we want it to do. Unfortunately I need to have a specifically shaped Tableau in mind before I can run anything, but first I'd like to see if my result matches up with what is expected. The following two functions, PlolyDerive and PolyCoeff are vitally important to what I wrote, so let's go through them before I try to explain what else I wrote:

4.1.1 PolyDerive

This code creates a function called PolyDerive that allows to me to input a polynomial, specify an integer *integer* and a variable, then returns the $integer^{th}$ derivative with respect to the variable we specified, and divides everything by *integer*!, which helps us get the coefficients in front of specific terms, as discussed previously.

```
PolyDerive:=function(polynomial, integer, variable);
if(integer = 0 ) then return polynomial; fi;
counter:=1;
while counter<= integer do
```

```
polynomial:= Derivative(polynomial, variable);
counter:=counter+1;
od;
return polynomial/Factorial(integer);
end;
return polynomial/Factorial(integer);
end;
```

4.1.2 PolyCoeffs

PolyCoeffs is a function that takes in a polynomial and returns a (very large vector, mind you) vector that corresponds exactly to the coefficients of some inputted polynomial.

```
PolyCoeffs:= function(polynomial);
matrix:= [];
for i in [0 .. 3] do
     for j in [0 .. 3] do
        for k in [0 .. 3] do
            for m in [ 0 .. 3] do
matrix[m*(4^3)+k*16+i*4+ j + 1] := Value( PolyDerive
....( PolyDerive(   PolyDerive(  PolyDerive(polynomial, j, y),
.... i, x) , k, z), m, w)  ,
....[x,y,z,w], [0,0,0,0]); #The line Breaks ('...') make this
od; od; od; od;                #Easier to Read
return matrix;
end;
```

4.1.3 Final Code

So suppose we take the following Tableau:

1	2	3
4		

This corresponds to the following polynomial:

$$(x_1 - x_4)(x_2)(x_3)$$

Our plan is to take all the elements of the symmetric group $S(4)$ and have them permute the indices of this tableau. Then we can generate vectors that represent the coefficients of these polynomials, put them into another array (thus giving us a matrix) and have us call $Rank()$ on that matrix, giving us the degree of the space generated by our row vectors–our polynomials. The code looks like the following:

```
PolyDerive:=function(polynomial, integer, variable);
if(integer = 0 ) then return polynomial; fi;
counter:=1;
while counter<= integer do
polynomial:= Derivative(polynomial, variable);
counter:=counter+1;
od;
return polynomial/Factorial(integer);
end;;


PolyCoeffs:= function(polynomial);
matrix:= [];
for i in [0 .. 3] do
     for j in [0 .. 3] do
        for k in [0 .. 3] do
            for m in [ 0 .. 3] do
matrix[m*(4^3)+k*16+i*4+ j + 1] := Value( PolyDerive( PolyDerive(
.....PolyDerive(  PolyDerive(polynomial, j, y),
.... i, x) , k, z), m, w)  ,[x,y,z,w], [0,0,0,0]);
od; od; od; od;
return matrix;
end;


mx:=Indeterminate(Rationals, 1);
y:=Indeterminate(Rationals, 2);
z:= Indeterminate(Rationals, 3);
w:=Indeterminate(Rationals, 4);
x_1
x_2
x_3
x_4
OnIndeterminates(x^1 + y^2 + w^3 + z^4, (1, 2)( 3, 4) );
x_4^4+x_3^3+x_1^2+x_2
polynomialStart:= (x-w)*(y)*(z);
x_1*x_2*x_3-x_2*x_3*x_4
PolyDerive(polynomialStart, 1, x);
x_2*x_3 x_2*x_3
PolyDerive(polynomialStart, 2, y);
```

```
0
PolyDerive(x^2*y*z, 2, x);
x_2*x_3
PolyCoeffs(polynomialStart);
Variable: 'matrix' must have an assigned value
PolyCoeffs(15*x^2*y^2*w*z);
[ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 15, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0 ]
m:=[PolyCoeffs(15*x^2*y^2*w*z),PolyCoeffs(12*x^2*y^2*w*z)] ;
[ [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
      0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
      0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
      0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 15, 0, 0, 0, 0,
      0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
      0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
      0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
      0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
      0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
      0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
      0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ],
  [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
      0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
      0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
      0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 12, 0, 0, 0, 0,
      0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
      0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
      0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
      0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
      0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
      0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
      0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ] ]
IsMatrix(m);
true
Rank(m);
1
elementmatrix:= Elements(SymmetricGroup(4) );
```

```
[ (), (3,4), (2,3), (2,3,4), (2,4,3), (2,4), (1,2), (1,2)(3,4), (1,2,3),
  (1,2,3,4), (1,2,4,3), (1,2,4), (1,3,2), (1,3,4,2), (1,3), (1,3,4),
  (1,3)(2,4), (1,3,2,4), (1,4,3,2), (1,4,2), (1,4,3), (1,4), (1,4,2,3),
  (1,4)(2,3) ]
i:=1;
1
elementmatrix[i];
()
MATRIX:= [];

[  ]
MATRIX[i]:=PolyCoeffs(OnIndeterminates(polynomialStart,elementmatrix[2]) );

[ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0 ]
for i in [1..Size(SymmetricGroup(4))] do
Print("Polynomial: ");
Print(OnIndeterminates(polynomialStart,elementmatrix[i]));
Print("\n \n ");
MATRIX[i]:=PolyCoeffs(OnIndeterminates(polynomialStart,elementmatrix[i]) );
i:= i+1;
od;
Polynomial: x_1*x_2*x_3-x_2*x_3*x_4

 Polynomial: x_1*x_2*x_4-x_2*x_3*x_4

 Polynomial: x_1*x_2*x_3-x_2*x_3*x_4

 Polynomial: x_1*x_3*x_4-x_2*x_3*x_4

 Polynomial: x_1*x_2*x_4-x_2*x_3*x_4

 Polynomial: x_1*x_3*x_4-x_2*x_3*x_4

 Polynomial: x_1*x_2*x_3-x_1*x_3*x_4
```

```
 Polynomial: x_1*x_2*x_4-x_1*x_3*x_4

 Polynomial: x_1*x_2*x_3-x_1*x_3*x_4

 Polynomial: -x_1*x_3*x_4+x_2*x_3*x_4

 Polynomial: x_1*x_2*x_4-x_1*x_3*x_4

 Polynomial: -x_1*x_3*x_4+x_2*x_3*x_4

 Polynomial: x_1*x_2*x_3-x_1*x_2*x_4

 Polynomial: -x_1*x_2*x_4+x_1*x_3*x_4

 Polynomial: x_1*x_2*x_3-x_1*x_2*x_4

 Polynomial: -x_1*x_2*x_4+x_2*x_3*x_4

 Polynomial: -x_1*x_2*x_4+x_1*x_3*x_4

 Polynomial: -x_1*x_2*x_4+x_2*x_3*x_4

 Polynomial: -x_1*x_2*x_3+x_1*x_2*x_4

 Polynomial: -x_1*x_2*x_3+x_1*x_3*x_4

 Polynomial: -x_1*x_2*x_3+x_1*x_2*x_4

 Polynomial: -x_1*x_2*x_3+x_2*x_3*x_4

 Polynomial: -x_1*x_2*x_3+x_1*x_3*x_4

 Polynomial: -x_1*x_2*x_3+x_2*x_3*x_4


a:=1; for a in [1.. Size(SymmetricGroup(4) )]do
Print("Matrix[", a ,"] :" ); Print(MATRIX[a], "\n \n ");
od;
1
Matrix[1] :[ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ]

Matrix[2] :[ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ]

Matrix[3] :[ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ]

Matrix[4] :[ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, -1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ]

Matrix[5] :[ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0,
```

```
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ]

Matrix[6] :[ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, -1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ]

Matrix[7] :[ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ]

Matrix[8] :[ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ]
```

```
Matrix[9] :[ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ]

Matrix[10] :[ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ]

Matrix[11] :[ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ]

Matrix[12] :[ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ]

Matrix[13] :[ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, -1, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ]

Matrix[14] :[ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, -1, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ]

Matrix[15] :[ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, -1, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ]

Matrix[16] :[ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, -1, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ]

Matrix[17] :[ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, -1, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ]

Matrix[18] :[ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, -1, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ]

Matrix[19] :[ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ]

Matrix[20] :[ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ]

Matrix[21] :[ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ]

Matrix[22] :[ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ]

Matrix[23] :[ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ]
```

```
 Matrix[24] :[ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ]



Rank(MATRIX);
3
```

So according to GAP, if this code is correct (I think the idea is correct, I wouldn't be surprised to hear that the code is buggy somewhere though) the dimension of the vector space corresponding to our Tableau from a few pages ago is 3.

4.1.4 Matrices Corresponding to The Vector-Space Homomorphisms Induced by elements of S_n

One can run the command $TriangulizeMat()$ on our matrix $MATRIX$ to get three basis vectors. One can now link the non-zero entries in our basis vectors to polynomials: We end up with the following (this is paraphrased, I have the code saved somewhere, I'd just rather explain what I'm doing for now):

```
newMatrix:=TriangulizeMat(MATRIX);;
NonzeroEntriesFor(newMatrix[1]);
22 (entry=1)  85(entry = -1)
NonzeroEntriesFor(newMatrix[2]);
70 (entry=1)  85(entry = -1)
NonzeroEntriesFor(newMatrix[3]);
82 (entry=1)  85(entry = -1)
```

The entries in the vectors from $MATRIX$ that are non-zero refer to specific polynomials. They are the following:

```
22 --> x_1 * x_2 * x_3
```

```
70 --> x_1 * x_2 * x_4
82 --> x_2 * x_3 * x_4
85 --> x_1 * x_3 * x_4
```

Notice that we then have the following polynomials associated with our basis:

```
e_1 = 22-85 -->  x_1 * x_2 * x_3 -  x_1 * x_3 * x_4
e_2 = 70-85 -->  x_1 * x_2 * x_4 -  x_1 * x_3 * x_4
e_3 = 82-85 -->  x_2 * x_3 * x_4 -  x_1 * x_3 * x_4
```

So, we'll give them each the following vector representation:

```
e_1 = [ 1 0 0 ]
e_2 = [ 0 1 0 ]
e_3 = [ 0 0 1 ]
```

Using this notation, we can find matrices A_i corresponding to the linear transforms induced by elements of the Symmetric group. We can look at the way in which some element $s \in S_4$ acts on our basis vectors to get an idea of what A_i should look like: We know that A should be a 3×3 matrix such that

$$T(x) = Ax$$

Where x is in our vector space V, and $T : V \to V$ is a linear transformation that corresponds to the action of some element in S_4. This all being said, we can find A by letting its columns i be $T(e_i)$. Starting off with a simple example, let $s \in S_4$ be the Identity. then:

$$s(e_1) = e_2, \quad s(e_2) = e_2, \quad s(e_3) = e_3$$

So our resulting matrix A is:

$$A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \qquad Trace(A) = 3$$

Now for a better example, let $s = (1\ 2)$. In this case,

$$\begin{aligned} s(e_1) &= (1\ 2)(x_1 \cdot x_2 \cdot x_3 - x_1 \cdot x_3 \cdot x_4) = x_1 \cdot x_2 \cdot x_3 - x_2 \cdot x_3 \cdot x_4 = \text{`}22 - 82' \\ s(e_2) &= (1\ 2)(x_1 \cdot x_2 \cdot x_4 - x_1 \cdot x_3 \cdot x_4) = x_1 \cdot x_2 \cdot x_4 - x_1 \cdot x_3 \cdot x_4 = \text{`}70 - 82' \\ s(e_3) &= (1\ 2)(x_2 \cdot x_3 \cdot x_4 - x_1 \cdot x_3 \cdot x_4) = x_1 \cdot x_3 \cdot x_4 - x_2 \cdot x_3 \cdot x_4 = \text{`}85 - 82' \end{aligned}$$

We represent the our results as vectors in the following way:

$$\text{`}22 - 82' = [1\ 0\ -1] \qquad \text{`}70 - 82' = [0\ 1\ -1] \qquad 85 - 82 = [0\ 0\ -1]$$

And so,

$$A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -1 & -1 & -1 \end{bmatrix} \qquad Trace(A) = 1$$

Now letting $s = (1\ 2\ 3)$, we have the following:

$$\begin{aligned} s(e_1) &= (1\ 2\ 3)(x_1 \cdot x_2 \cdot x_3 - x_1 \cdot x_3 \cdot x_4) = x_1 \cdot x_2 \cdot x_3 - x_2 \cdot x_1 \cdot x_4 = \text{`}22 - 70' \\ s(e_2) &= (1\ 2\ 3)(x_1 \cdot x_2 \cdot x_4 - x_1 \cdot x_3 \cdot x_4) = x_2 \cdot x_3 \cdot x_4 - x_2 \cdot x_1 \cdot x_4 = \text{`}22 - 70' \\ s(e_3) &= (1\ 2\ 3)(x_2 \cdot x_3 \cdot x_4 - x_1 \cdot x_3 \cdot x_4) = x_3 \cdot x_1 \cdot x_4 - x_2 \cdot x_1 \cdot x_4 = \text{`}85 - 70' \end{aligned}$$

This corresponds to the following matrix:

$$A = \begin{bmatrix} 1 & 0 & 0 \\ -1 & -1 & -1 \\ 0 & 1 & 0 \end{bmatrix} \qquad Trace(A) = 0$$

When $s = (1\ 2\ 3\ 4)$ we have:

$$\begin{aligned} s(e_1) &= (1\ 2\ 3\ 4)(x_1 \cdot x_2 \cdot x_3 - x_1 \cdot x_3 \cdot x_4) = x_2 \cdot x_3 \cdot x_4 - x_2 \cdot x_4 \cdot x_1 = \text{`}82 - 70' \\ s(e_2) &= (1\ 2\ 3\ 4)(x_1 \cdot x_2 \cdot x_4 - x_1 \cdot x_3 \cdot x_4) = x_2 \cdot x_3 \cdot x_1 - x_2 \cdot x_4 \cdot x_1 = \text{`}22 - 70' \\ s(e_3) &= (1\ 2\ 3\ 4)(x_2 \cdot x_3 \cdot x_4 - x_1 \cdot x_3 \cdot x_4) = x_3 \cdot x_4 \cdot x_1 - x_2 \cdot x_4 \cdot x_1 = \text{`}85 - 70' \end{aligned}$$

This corresponds to the following matrix:

$$A = \begin{bmatrix} 0 & 1 & 0 \\ -1 & -1 & -1 \\ 1 & 0 & 0 \end{bmatrix} \qquad Trace(A) = -1$$

When $s = (1\ 2)(3\ 4)$ we have

$$\begin{aligned} s(e_1) &= (1\ 2\ 3\ 4)(x_1 \cdot x_2 \cdot x_3 - x_1 \cdot x_3 \cdot x_4) = x_2 \cdot x_1 \cdot x_4 - x_2 \cdot x_4 \cdot x_3 = \text{`}70 - 82' \\ s(e_2) &= (1\ 2\ 3\ 4)(x_1 \cdot x_2 \cdot x_4 - x_1 \cdot x_3 \cdot x_4) = x_2 \cdot x_1 \cdot x_3 - x_2 \cdot x_4 \cdot x_3 = \text{`}22 - 82' \\ s(e_3) &= (1\ 2\ 3\ 4)(x_2 \cdot x_3 \cdot x_4 - x_1 \cdot x_3 \cdot x_4) = x_1 \cdot x_4 \cdot x_3 - x_2 \cdot x_4 \cdot x_3 = \text{`}85 - 82' \end{aligned}$$

This corresponds to the following matrix:

$$A = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix} \qquad Trace(A) = -1$$

This was all done by hand, without pen and paper- so bear in mind there may be quite a few errors. If some of these match up to what they should be, I might just doing something wrong.

While I did these by hand as opposed to having GAP do them, the results I used were from the code I wrote, so if that code is correct these matrices and calculations should be correct.

4.1.5 Links

- http://www.gap-system.org/Manuals/doc/htm/ref/CHAP064.htm#SECT007 uses the function "OnIndeterminates" to permute indices of a polynomial.
- http://www.gap-system.org/Manuals/doc/htm/ref/CHAP064.htm#SECT007 has a function called 'Value' that seems to be able to evaluate multi-variable polynomials
- http://www.gap-system.org/Manuals/doc/htm/ref/CHAP064.htm#SECT006 seems to have some support for derivatives in multi-variables
- http://www.gap-system.org/Manuals/doc/htm/ref/CHAP059.htm#SECT002 has some interesting ideas for bases of vector spaces
- http://www.math.tamu.edu/~yvorobet/MATH304-504/Lect2-12web.pdf just some quick notes regarding matrices and linear transforms