数学特別講義 II

GAP 利用者ガイド

脇 克志
山形大学 理学部
waki@sci.kj.yamagata-u.ac.jp

2012 年 1 月 16 日

これは、2012 年 1 月 16 日から 1 月 18 日まで筑波大学で行なわれた集中講義の講義ノートです。内容は、代数構造計算システム GAP(version 4.4) を使った有限群論の計算方法を具体的な例を入れながら解説して行きます。いくつかの章は、GAP のマニュアルを翻訳して載せています。また、G. Butler の著書 [1] を基に書いている部分もあります。計算機による有限群の計算アルゴリズムについては、D. F. Holt, B. Eick, E. A. O'Brien [3] にまとめられています。K.Lux,H. Pahlings[4] には、表現に関連した GAP の例題がたくさん紹介されています。

計算機による置換群の計算については、Cameron [2] が参考になると思います。

目次

第 1 章

GAP の概説

この章は GAP の全体像を捕えてもらうことを目標にしています。GAP には、いろいろな代数構造の計算が可能ですが、ここでは、特に有限群と関係が深い内容を選びました。

1.1 GAP の概要

GAP は、ドイツで産まれた有限代数構造の計算を行なうフリーソフトです。UNIX(Linux), Windows, MacOS の 3 つ OS 上で動かすことが出来ますが、主に UNIX 上でもっとも良く動いているソフトです。この講義ノートでは、Linux 版の GAP を念頭に置いて話を進めて行きますが、ほとんどの計算はどの OS においても実行可能であるはずです。現在の最新版は Version 4.5β (4.4 が安定版) で、最新版の GAP は、`http://www.gap-system.org/` より入手できます。このサイトを参照すれば GAP についての必要にして十分な情報が得られます。

1.1.1 GAP の守備範囲 (出来ること出来ないこと)

GAP は、有限のメモリーを備えた計算機の中で動く代数構造の計算を得意とする計算システムです。(プログラム言語と呼ぶことも出来ます) よって、有限の代数構造の構成、分解、解析をその主なる目的としています。有理数体など一部有限でない代数構造も定義されていますが、これらの有限でない代数構造はあくまで有限の代数構造やその表現を実現するのに必要なため、用意されたものです。つまり、有理数体はそれ自身を解析するためではなく、これを土台としたべつの有限代数構造を定義したり、表現したりするために存在しています。GAP は、誰もが無償で利用できる代数構造解析ソフトウェアです。有限集合上で構成される代数構造なら利用者が自由にその演算を定義できる環境が調えられています。ただし、実際の計算能力の点で、有償のソフトウェアである MAGAMA に及ばない部分がいくつかあるのも事実です。しかし、中味がすべて公開されていて改良もできるは、GAP 大きな強みであるといえます。

1.1.2 GAP の基本操作

GAP がきちんとインストールされた状態では、端末画面で、“`gap`” とタイプすることで GAP が起動します。この場合デカデカと GAP のタイトルが現れますが、“`gap -b`” と打ち込んだ場合は、このタイトルを見ないで済ませる事が出来ます。また、GAP を終了

させるためには、quit; と打ち込みます。GAP に入力する命令は 1 つの命令語とセミコロン";"を続けて打ち込む必要があります。このセミコロンを入力しない場合、 GAP は、たとえ改行キーを押したとしてもまだ命令は完結していないと解釈して、行の頭に> を表示して命令の続きを待ちます。また命令の後にセミコロンを 2 回続けて打ち込んだ場合、実行された命令は計算結果を端末画面に表示しません。次の例では、長さ 3 のベクトル x と 3 次の正方行列 M を定義して、x と M の積を、y にしています。:= は、等式ではなく左辺の変数を、右辺の内容で定義する命令です。下の例では、変数 x に、長さ 3 のベクトルを定義して、最後にセミコロン";"を打っています。その次の行には、定義された内容が表示されています。行列 M の定義では、途中で入力を見易くするために改行キーを押していますので、次の行の頭に> が表示されています。x と M の積を計算して、その結果を y としていますが、命令の後にセミコロンを 2 回続けて";;"と打っていますので、その次の行に計算結果は表示されていません。計算結果が見たいときは、最後の命令のように"y;"と打ち込むことで変数"y;"の内容が表示されます。

```
gap> x:=[1,0,1];
[ 1, 0, 1 ]
gap> M:=[ [1,1,1],
>         [0,1,1],
>         [0,0,1]];
[ [ 1, 1, 1 ], [ 0, 1, 1 ], [ 0, 0, 1 ] ]
gap> y:=x*M;;
gap> y;
[ 1, 1, 2 ]
```

1.1.3 GAP の基本要素

GAP で計算をする上で、もっとも基本となる要素が、いくつかあります。ここでは、基本要素として、整数、真偽値、文字を紹介します。

整数

これは、説明するまでもありませんが、0, 1, 2, .. 100, ... 100000 など負の整数も含め数は自由に使えます。有理数は 2/3 のように表現できますが、小数表現は、使えません。これは、GAP が数値計算言語ではなく、記号処理言語であることを意味しています。このため、GAP では、計算誤差などを気にする必要は、ありません。これらの数については、足し算 (+)、引き算 (-)、かけ算 (*)、割算 (/)、剰余 (mod) などの演算が可能です。

べき乗根

有限群の指標を使うために、 1 の原始 n 乗根 E(n) や有理数のべき根 ER(n) を使うことができます。

```
gap> (-1+ER(-3))/2;
E(3)
gap> (-1-ER(-3))/2;
E(3)^2
```

真偽値

ある命題が与えられたときにその命題が「真」であるとき true、「偽」であるとき false が決まります。この 2 つの値が真偽値で、真偽値に対しては、論理和 or、 論理積 and、否定 not などの論理演算を利用することが可能です。また、真偽値では、ないのですが関数などが実行できなかった場合に、fail(失敗) という値を出すことがあります。fail に対しては、論理演算は使えません。

文字

シングル・クォーテーション' で挟まれた 1 文字を、**文字**と呼びます。特殊な文字として、改行を表す\n や、タブを表す\t があり、バックスラッシュ \ を出力したい場合は、\\ と打ち込みます。

また、ダブル・クォーテーション" で囲われたものを**文字列**と呼びます。文字列は、文字のリスト (1.1.4 を参照) なので、基本要素と呼ぶよりは、データ構造と呼ぶべきものになります。文字に関する関数としては、INT_CHAR や、CHAR_INT のように、文字と文字コード数 (例えば、CHAR_INT('A')=65, CHAR_INT('B')=66, CHAR_INT('a')=97 など) を対応させるものがあります。

1.1.4 GAP の基本データ構造 (リスト)

GAP には、いくつかの基本データ構造がありますが、ここではプログラムを作る上で特に重要なリスト (List) と呼ばれるデータ構造を紹介します。

リストの定義

リストとは、GAP のデータをカンマ , で区切りながら連ねて [と] で囲ったものです。例えば、1 から 5 までの数字のリストは、[1,2,3,4,5] と表せます。GAP の命令として、P:=[2,3,5,7,11,13,17,19]; と入力すれば、長さが 8 の 20 以下の素数のリストを定義したことになります。また、1 から 100 までの数のリストなら、 [1..100] と表すことができます。これは、$\{x \in \mathbb{Z} | 1 \leq x \leq 100\}$ を意味することなります。10 以下の正の偶数のリストなら、 [2,4..10] と表すことが出来ます。また、要素 0 が 100 個並んだリストを定義したいときは、ListWithIdenticalEntries(100,0) とすると作れます。

リストからのデータ抽出とリストの変更

先ほど定義した、リスト P から 3 番目の素数を取り出したい場合は、P[3]; と入力することで得られます。また、リスト P から複数の素数を取り出すには、取り出したい場所をリストにして指定します。例えば、リスト P の 4 番目と 5 番目と 7 番目の要素を取り出したい場合、リスト [4,5,7] を使って、P{[4,5,7]}; と入力すると、リスト [7,11,17] が得られます。定義したリストの一部を変更したいときは、その部分を指定して、代入命令:= を使って、リストを変更します。下の例では、リスト P を定義した後に、3 番目の要素を 9 に変更したり、4,5,7 番目の要素を全て 0 に変更したりしています。

```
gap> P:=[2,3,5,7,11,13,17,19];
[ 2, 3, 5, 7, 11, 13, 17, 19 ]
```

```
gap> P[3]:=9;;
gap> P;
[ 2, 3, 9, 7, 11, 13, 17, 19 ]
gap> P{[4,5,7]}:=[0,0,0];;
gap> P;
[ 2, 3, 9, 0, 0, 13, 0, 19 ]
```

文字列の定義

1.1.3 でも紹介しましたが、文字を要素とするリストを文字列と呼びます。ですから、`S:=['A',' ','b','o','o','k'];` と入力すれば、S には、文字列 `"A book"`が入ることになります。逆に、リストとして次のように文字列の内容を変更することも可能です。

```
gap> S:="A book ";
"A book "
gap> S{[1,7]}:=['2','s'];;
gap> S;
"2 books"
```

集合の定義

リストの特殊なものとして、各要素の間に決まった順序が存在し、その順序で小さい順に並んだ、重複を含まないリストを**集合 (Set)** と呼びます。リストが集合であるかどうかは、関数 `IsSet` で判断できます。また、関数 `Set` を使って、リストから集合を作ることも可能です。以下を参考にしてください。

```
gap> IsSet([1,2,3]);
true
gap> IsSet([1,1,3]);
false
gap> IsSet([2,1,3]);
false
gap> Set([1,1,3]);
[ 1, 3 ]
gap> Set([2,1,3]);
[ 1, 2, 3 ]
```

リストのリスト

1.1.2 で紹介した行列 M のように、リストの各要素がリストになっているものが、あります。このとき、リストの中のリストの成分を取り出したいときは、次の例のように [と] を 2 回使うことになります。{ } をうまく使うことで、行列の一部を取り出した小さい行列を作ることも出来ます。

```
gap> LL:=[[1,2,3],[4,5,6],[7,8,9]];;
```

```
gap> Display(LL);
[ [  1,  2,  3 ],
  [  4,  5,  6 ],
  [  7,  8,  9 ] ]
gap> LL[2][3];
6
gap> LL[3]{[1,2]};
[ 7, 8 ]
gap> LL{[2,3]}[2];
[ 5, 8 ]
gap> Display(LL{[1,3]}{[1,3]});
[ [  1,  3 ],
  [  7,  9 ] ]
```

リストに対する基本的な関数

ここでは、リストを扱う上で必要となる関数 in, Add, Concatenation, IsBound, Length, Position, List, Filtered を紹介します。

in　リスト L の中にある要素 a が含まれているかどうかを調べてたい時に、a in L; と命令すると、含まれていれば true、含まれて居なければ false が表示されます。

Add　リスト L とある要素 a があるとき、Add(L,a); と命令するとリスト L の最後に成分 a が追加されます。関数 Add では、入力に使ったリスト L が変化するだけで結果を出力しません。

Concatenation　2 つのリスト L, R があるとき、命令 A:=Concatenation(L,R); で、2 つのリストを結合させたリスト A が出来上がります。関数 Add の場合と違いリスト L と R の内容は変化しません。下の例で、R は、文字列 (文字のリスト) でしたが、リスト A には、文字以外に数字も要素として含まれているため、もはや文字列では無くなっています。ただし、A の 5,6,7 番目だけを取り出したリストは文字列になります。

```
gap> L:=[1,2,3];; R:="ABC";;
gap> 4 in L;
false
gap> Add(L,4);
gap> L;
[ 1, 2, 3, 4 ]
gap> 4 in L;
true
gap> A:=Concatenation(L,R);
[ 1, 2, 3, 4, 'A', 'B', 'C' ]
gap> A{[5,6,7]};
"ABC"
```

IsBound リストは、データをカンマ, で区切ったものとしましたが、カンマが連続してデータが無い部分があってもリストとして、成り立ちます。つまり、L:=[2,3,,5,,7]; といった、リストも許されます。しかし、プログラムの中で、L[3] を使おうとすると、エラーが発生して、プログラムが止まってしまいます。そこで、このようなリストを扱う場合は、前もって、L[3] にデータが割り当てられているか確かめる関数が必要になります。関数 IsBound を使うことで、次のようにリストにデータが割り当てられているかどうかを真偽値で確かめることが出来ます。

```
gap> L:=[2,3,,5,,7];
[ 2, 3,, 5,, 7 ]
gap> IsBound(L[3]);
false
gap> IsBound(L[4]);
true
gap> Length(L);
6
```

Length この関数はリストの長さを返します。これは、要素の数ではないことに注意しましょう。つまり上の例では、リスト L の要素の数は 4 ですが, リストの長さは、6 となります。

Position リストの中である要素がリストの何番目にあるかを調べるときは、この関数を使います。 リストの中にその要素が複数存在する場合は、場所を表す番号で最も小さい値が出力されます。また、もしその要素がリストの中に存在しない場合は、fail が与えられます。

```
gap> L:=[2,3,5,7,11,13,17,19];;
gap> Position(L,7);
4
gap> Position(L,23);
fail
```

PositionPropaty リストの中で、ある条件を満たす（ある関数に代入して true の値を得る）最初の要素がリストの何番目にあるかを調べます。次の例では、$2^{2^i}+1$ が最初に素数でなくなるのは、$i=5$ の時だと分かります。

```
gap> PositionProperty([1..10],i->not IsPrime(2^(2^i)+1));
5
```

List リストを扱うときもっとも頻繁に利用する関数が List です。関数 List は、リストからリストを作る関数です。入力されたリストの要素 1 つ 1 つに対して指定された計算を行いその結果をリストにしたものを出力します。
使い方は、List(リスト ,i-> i を使った計算)
と書いて、リストの各要素は変数 i に代入されて、計算結果がリストになり出力されます。例えば、1 から 5 までの数字のリストから、2 から 10 までの

偶数のリストを作るときは、List([1..5],i->2*i) とします。 また、次の例のように数 65 から 90 が関数 CHAR_INT で文字 'A' から'Z' になり、文字列 "ABCDEFGHIJKLMNOPQRSTUVWXYZ" が得られます。

```
gap> As:=List([1..26],i->CHAR_INT(i+64));;
gap> As;
"ABCDEFGHIJKLMNOPQRSTUVWXYZ"
```

Filtered この関数もリストからリストを作る関数ですが、これはリストの各要素から条件に合うものだけを抽出する関数です。次の例では、1 から 50 の数のリストから 1 引いた値が 4 で割り切れる素数の要素だけを選び出しています。関数 IsPrime(n) は、n が素数のとき 真 (true) 素数でないとき、偽 (false) を出す関数です。

```
gap> Filtered([1..50],i->(i-1) mod 4=0 and IsPrime(i));
[ 5, 13, 17, 29, 37, 41 ]
```

1.1.5 繰り返し・分岐命令

For 文の利用

For 文は、基本的な繰り返しの命令でです。 リスト L の要素を 1 回づつ変数 i に代入しながら繰り返す命令は、

for i in L do 繰り返したい実行命令 od;

となります。例は pp.8 の if の例といっしょに示します。

While 文の利用

これも、繰り返しの命令です。For 文では、ちょうどリストの長さ分だけ繰り返しが行われましたが、While 分では、与えられた条件を満たす間は何回でも繰り返します。

while 続ける条件 do 繰り返したい実行命令 od;

そのため、繰り返しを続けさせる条件をきちんと確認しておかないと**永遠に止まらない**プログラムを作る危険がありますので、注意しましょう。また、プログラムを書くときは、下の例のように段差を付けて書くことで、プログラムをより見やすく出来ます。

```
gap> i:=0;; s:=0;;
gap> while s<=200 do
>       i:=i+1;
>       s:=s+i^2;
>     od;
gap> s;
204
gap> i;
8
```

上のプログラムでは、i と s を 0 にしてプログラムをスタートして、s の値が、200 以下 の間は、i の値を 1 つづつ増やしながら、s に i^2 を加え続けます。よって、$1^2 + 2^2 + \cdots + 8^2 = 204$ は、 2 乗数の和が始めて 200 を越えた時の数となります。

繰り返しを止める命令

繰り返しを強制的に止める命令として break; という命令が用意されています。この命令を使うことで、For 文でも While 文でも繰り返しをを停止して、繰り返しの後の命令 (つまり、od; の後の命令) へ移ります。単純な繰り返しでは、まず使いません。例えば、While 文での繰り返し実行するのための条件が複雑な場合に、使われます。

IF 文の利用

IF 文は、条件分岐に使われます。使い方は、

if 評価 then 評価が真の時の実行命令 else 評価が偽の時の実行命令 fi;

です。 次の例では、変数 i が 1 から 5 まで、変化します。その中で、リスト ListOfWords の中の 4 つの文字列の i 番目の文字を次々に CanYouSee に加えて行きます。ただし、文字列の長さが i より短い場合は、代わりに空白文字 ’ ’ を CanYouSee に加えて行きます。こうして、長さ 20 の暗号文字列 CanYouSee が出来ます。この文字を解読するには、文字を 4 つごとに集めてまとめれば良いので、関数 List を使って、 4 つごとの文字を集めると暗号が解読されています。

```
gap> ListOfWords:=["Have","a","great","time"];
[ "Have", "a", "great", "time" ]
gap> CanYouSee:=[];;
gap> for i in [1..5] do
      for word in ListOfWords do
        if i<=Length(word) then
          Add(CanYouSee,word[i]);
         else
          Add(CanYouSee,' ');
        fi;
      od;
    od;
gap> CanYouSee;
"Hagta riv eme ae  t "
gap> List([1..4],i->CanYouSee{[i,i+4..i+16]});
[ "Have ", "a    ", "great", "time " ]
```

1.1.6 関数の定義と出力

関数 function による関数の定義

関数名 :=function(入力) 関数の内容 end;

で、関数を定義できます。プログラムを作るとは、まさに関数を定義することです。では、2 つの入力 x, y を与えて、$x^2 - y^2$ を計算する関数 f を作ってみます。関数の計算結果は return を使って出力します。次の例では、リスト [1..10] を使って、$2^2 - 1^2$, $3^2 - 2^2$,... $11^2 - 10^2$ を順に計算してリストにしています。

```
gap> f:=function(x,y)
> return x^2-y^2;
> end;
```

```
function( x, y ) ... end
gap> List([1..10],i->f(i+1,i));
[ 3, 5, 7, 9, 11, 13, 15, 17, 19, 21 ]
```

今までに自分で定義した変数や関数の名前は、NamesUserGVars(); を使って表示することが出来ます。

```
gap> NamesUserGVars();
[ "f" ]
```

局所変数

関数の定義の途中で、**その関数の中だけで**必要となる変数を局所変数と呼びます。GAP では、局所変数を関数定義の最初のところで local を使って定義する必要があります。次の例では、局所変数 W と i を使って 1 から N までの数の和を計算する関数 Wa を定義しています。

```
gap>   Wa:=function(N)
>     local W, i;
>     W:=0;
>     for i in [1..N] do
>       W:=W+i;
>     od;
>     return W;
>   end;
function( N ) ... end
gap> Wa(10);
55
gap> Wa(30000);
450015000
```

変数に関連してもう 1 つお話します。関数の入力値は、その種類により関数内の計算に影響を**受ける場合**と**受けない場合**があるようです。

```
gap> f:=function(I)
>   I:=I+1;
> end;
function( I ) ... end
gap> g:=function(L)
>   L[1]:=L[1]+1;
> end;
function( L ) ... end
gap> I:=1;; L:=[1];;
gap> f(I); g(L);
gap> I; L;
1
[ 2 ]
```

上の例では、数値 I の値は関数 f を適用した後も変化していませんが、リスト L の値は、関数 g を適用した後で増えています。この辺の違いは C 言語などでプログラムを書いたことのある人は、なんとなく理解できるのではないかと思います。

1.1.7 変数複製関数

1.1.6 の後半の話と関連して、値を複製する関数 ShallowCopy を紹介します。次の例では、リストを別の名前で定義するとき起こる注意すべき現象を示しています。T:=L; と単純に T を L と定義すると、L の要素が変化したとき、名前が違うだけで実質同一の T の要素も変わります。

```
gap> L:=[1,2];;
gap> T:=L;
[ 1, 2 ]
gap> L[2]:=1;
1
gap> T;
[ 1, 1 ]
```

関数 ShallowCopy (または、StructuralCopy) を使うことで、T は、L と同じ内容を持つ別のリストになります。よって、L の内容が変化しても T は、変化しません。

```
gap> L:=[1,2];;
gap> T:=ShallowCopy(L);
[ 1, 2 ]
gap> L[2]:=1;;
gap> T;
[ 1, 2 ]
```

1.1.8 計算結果の出力方法

1.1.2 の最後でも紹介したように、変数 x の中身を見たいときは、"x;"と打ち込めば、良いのですが、出力結果を見やすくするために、いくつかの出力命令が用意されています。

Print　この命令は、まさに内容を表示する命令ですが、変数や文字列や特殊記号をカンマ , で区切って並べることで、これらを組み合わせてより分かりやすく表示することも来ます。例えば、次の例では、先ほど定義した変数 x と y を表示していますが、文字列" x="や改行記号\n などを組み合わせて綺麗に表示しています。

```
gap> x:=[1,0,1];; y:=[1,1,2];;
gap> Print(" x=",x,"\n y=",y,"\n");
 x=[ 1, 0, 1 ]
 y=[ 1, 1, 2 ]
```

Display　この命令では、GAP のデータの中に埋め込まれたデータ独自の表示方法にしたがって、データを表示します。これは、それぞれの GAP データに合わせて出来るだけ見やすい形で表示しようとしますが、いつでも綺麗に表示できるとは、限りません。次の例では、正方行列 M を、Display で表示しています。
注意：f や M が定義されてないと表示しません。
また、Print 命令を使って、定義した関数を表示することも可能です。

```
gap> Print(f,"\n");
function ( x, y )
```

```
        return x ^ 2 - y ^ 2;
    end

    gap> Display(M);
    [ [  1,  1,  1 ],
      [  0,  1,  1 ],
      [  0,  0,  1 ] ]
```

PrintTo　Print 命令とほぼ同じですが、画面ではなく指定されたファイルに表示内容を出力します。このとき、このファイル名のファイルが存在した場合にそのファイルに**上書き**されてしまうので、注意が必要です。次の例では、先ほど表示した内容をファイル"sample.txt"に出力します。

```
gap> PrintTo("sample.txt"," x=",x,"\n y=",y,"\n");
```

AppnedTo　PrintTo と同じ働きをしますが、指定したファイルが存在するとき**上書きをしないで**ファイルの最後に表示内容を**追加します。**よって、計算結果をファイルに出力するには、最初に 1 回 PrintTo 命令を実行し、後は AppendTo 命令で追加の出力を行うことになります。

1.1.9　データやプログラムの一括入力

入力したいデータやプログラムを、毎回キーボードから入力するのは、面倒ですし打ち間違いも心配です。そこで、入力したいデータをテキストファイルにしておいて、Read 命令で一括して入力してしまうと便利です。これにより、同じ様な入力を毎回する必要もなくなり、プログラムの誤り訂正も楽になります。例えば、テキストファイル prg.txt に、

```
i:=2*;
j:=3;
k:=7;
Print("i*j*k=",i*j*k,"\n");
```

と言う内容を書き込んで、Read 命令を使ってこの内容を読み込みます。すると、

```
gap> Read("prg.txt");
Syntax error: expression expected in prg.txt line 1
i:=2*;
     ^
Variable: 'i' must have a value
```

と出力されます。これは、ファイル prg.txt の 1 行目の^のところに GAP の文法上のエラーがあることを示しています。これでは、変数 i は、きちんと定義できません。prg.txt の 4 行目では、この変数 i を使った Print 命令がありますが、i が定義されていないので、実行できません。このことを、示しているのが、最後の文

```
Variable: 'i' must have a value
```

です。そこで prg.txt を次のように訂正します。

```
i:=2*5;
j:=3;
k:=7;
Print("i*j*k=",i*j*k,"\n");
```

すると、こんどはキチンと結果が帰って来ます。Read 命令では、Print や Display などの表示命令の結果は表示されますが、それ以外の命令は表示されないことに注意してください。

```
gap> Read("prg.txt");
i*j*k=210
```

1.1.10 計算時間の測定

実際に、プログラムを動かしたときにその計算のためにパソコンが使った時間を知りたいとき、time; と打ち込んで、直前の命令の計算時間を知ることが出来ます。時間の単位は、1/1000 秒です。例えば、次の例では 1 から 10,000,000 までの和を計算するのに、2.61 秒かかったことになります。

```
gap> Wa(10000000);
50000005000000
gap> time;
2610
```

1.1.11 プログラムがうまく動かないとき

プログラムを作っている過程で、実行エラーに遭遇した場合、GAP は、プロンプト brk> を出して停止します。そんなときは、Where(); を使ってプログラムがどんな関数を実行している途中で、エラーを出してしまったのかを確認知ることが出来ます。ただし、プログラムが複雑になると、関数の中で別の関数が使われる入れ子の状態が発生し、この階層構造のどのレベルでエラーが発生したのか見えにくくなります。DownEnv(); を使うことで、1 つ外側の関数に移動して、各レベルでの変数の状態を確認することが出来ます。DownEnv(-1); と打ち込むと逆に 1 つ内側の関数に入り込みます。。次の例では、関数 Error を使って意図的にエラーを発生させています。関数を移動することで、変数 n の値が変化していることが分かります。

```
gap> fc:=function( n )
>     if n > 2 then
>         Error("!");
>     else
>         return 0;
>     fi;
> end;
function( n ) ... end
gap> fb:=function( n )
>     if n > 3 then
>         Error(‘‘!’’);
>     else
>         return fc(n+1);
>     fi;
> end;
function( n ) ... end
gap> fa:=function( n )
>     return fb(n+1);
> end;
gap> fa(1);
Error, ! called from
fc( n + 1 ) called from
fb( n + 1 ) called from
<function>( <arguments> ) called from read-eval-loop
```

```
Entering break read-eval-print loop ...
you can 'quit;' to quit to outer loop, or
you can 'return;' to continue
brk> n;
3
brk> Where();
 called from
fc( n + 1 ) called from
fb( n + 1 ) called from
<function>( <arguments> ) called from read-eval-loop
brk> DownEnv();
brk> n;
2
brk> Where();
 called from
fb( n + 1 ) called from
<function>( <arguments> ) called from read-eval-loop
brk> DownEnv();
brk> n;
1
brk> Where();
 called from
<function>( <arguments> ) called from read-eval-loop
```

1.1.12 コメント文

これは GAP でのプログラムに限った話ではありませんが、プログラムには、出来るだけ沢山のコメント文 (プログラムを説明するための文章で計算機からは無視される部分) を付けるようにしましょう。コメント文は、他人にプログラムを使ってもらったり、自分が過去に作ったプログラムを再活用するときに、とても役に立ちます。GAP では、記号 # の後に書かれた文字がコメント文と見なされ計算機からは、すべて無視されます。

1.1.13 GAP のヘルプ機能と補完・編集機能

GAP は、オンラインヘルプ機能を持っています。ある**命令**についての詳しい説明を知りたい場合は、**?命令**と打ち込みます。そして、見ているオンラインヘルプの次の頁を見たい場合は ?>、前の頁が見たい場合は ?<、 を入力するすることで、前後の関連したオンラインヘルプを読むことが出来ます。

GAP の入力を助けてくれる機能として、命令の補完機能があります。これは、命令を途中まで打ち込んだ段階で、tab キーを押すことにより GAP に定義されている命令の中から、今まで入力した文字で始まるものを探して当てはまるものが 1 つの場合は、入力した文字を補完してくれます。もし、複数の侯補がある場合は、その侯補が共通に持つ文字列のところまで、補完します。さらに、複数の侯補がある場合は、 tab キーを続けて 2 度打ち込むことで、複数侯補の一覧を表示します。

入力を助けるもう 1 つの機能として、いままで入力した命令を上矢印を押すことで、もう一度表示できる機能です。似た命令を複数行なうときは、この機能により、以前打ち込んだ命令を表示して、左右矢印と Delete キーを使って命令を編集して使うことが出来ます。上矢印を打つ前に、少し文字列を打ち込んでおくと以前同じように文字列を入力した命令を導き出して表示してくれますので、さらに便利です。

1.2 GAP を使った代数構造の定義

この節では、GAP を使ってどのように代数構造を定義するかを紹介していきます。

1.2.1 有限体の定義

有限体の定義は、GF(p^n) の形で行ないます。(p はある素数）例えば、2^3 個の元からなる有限体を k としたい場合は、k:=GF(8); と打ち込めば良いわけです。また、0 以外の有限体 GF(p^n) の元は、巡回乗法群の生成元 Z(p^n) の巾として作ることが出来ます。また、1.1.3 で見たように E(n) は、1 の原始 n 乗根を表します。このため、GAP では、文字 Z と E を利用者が作るプログラムの中で変数として利用することは出来ません。また、有限体を加法群としてみたいときは、 AsVectorSpace を使って有限体を素体上のベクトル空間として定義し直して、各元をこのベクトル空間のベクトルとして表すことも出来ます。有限体 k の零元や乗法の単位元は、Zero(k) や One(k) で定義できます。

```
gap> k:=GF(8);
GF(2^3)
gap> a:=Z(8);
Z(2^3)
gap> List([1..7],i->a^i);
[ Z(2^3), Z(2^3)^2, Z(2^3)^3, Z(2^3)^4, Z(2^3)^5, Z(2^3)^6, Z(2)^0 ]
gap> V:=AsVectorSpace(GF(2),k);
GF(2^3)
gap> Dimension(V);
3
gap> BasisVectors(Basis(V));
[ Z(2)^0, Z(2^3), Z(2^3)^2 ]
gap> Coefficients(Basis(V),a^5);
[ Z(2)^0, Z(2)^0, Z(2)^0 ]
gap> a^5=a^0+a+a^2;
true
```

1.2.2 ベクトル空間の定義

ベクトル空間を作るためには、まず体上のベクトルを作る必要があります。 ベクトルとは、各成分が同じ体の要素で構成されたリストのことです。ベクトル空間は生成ベクトルとなるベクトルのリストを使って、関数 VectorSpace で定義します。次の例では、有理数体を表す Rationals を使って、有理数体上のベクトル空間を構成しています。有限体 GF(q) 上の n 次元ベクトル空間は、GF(q)^n で定義できます。

```
gap> V:= VectorSpace( Rationals, [ [ 1, 2, 3 ], [ 1, 1, 1 ] ] );;
gap> W:= Subspace( V, [ [ 0, 1, 2 ] ] );
<vector space over Rationals, with 1 generators>
gap> U:=GF(3)^4;
( GF(3)^4 )
gap> BasisVectors(Basis(U));
[ [ Z(3)^0, 0*Z(3), 0*Z(3), 0*Z(3) ], [ 0*Z(3), Z(3)^0, 0*Z(3), 0*Z(3) ],
  [ 0*Z(3), 0*Z(3), Z(3)^0, 0*Z(3) ], [ 0*Z(3), 0*Z(3), 0*Z(3), Z(3)^0 ] ]
```

1.2.3 置換群の定義

置換群は、交代群や対象の部分群として存在しています。やはり置換群の生成元となる置換のリストから、関数 Group を使って構成できます。置換群の計算については、第 2 章で詳しく紹介します。

1.2.4 行列群の定義

行列群は、一般線形群の部分群として、行列群の生成元となる行列のリストから、やはり関数 Group を使って構成できます。行列とは、長さの揃った同じ体の要素で構成されたベクトルのリストです。もちろん行列群の生成元となるものは、正方行列である必要があります。

```
gap> m1 := [ [ Z(3)^0, Z(3)^0,   Z(3) ],
>            [   Z(3), 0*Z(3),   Z(3) ],
>            [ 0*Z(3),   Z(3), 0*Z(3) ] ];;
gap> m2 := [ [   Z(3),   Z(3), Z(3)^0 ],
>            [   Z(3), 0*Z(3),   Z(3) ],
>            [ Z(3)^0, 0*Z(3),   Z(3) ] ];;
gap> m := Group( [m1, m2] );
Group(
[
 [ [ Z(3)^0,Z(3)^0,Z(3) ], [ Z(3),0*Z(3),Z(3) ], [ 0*Z(3),Z(3),0*Z(3) ] ],
 [ [ Z(3),Z(3),Z(3)^0 ],   [ Z(3),0*Z(3),Z(3) ], [ Z(3)^0,0*Z(3),Z(3) ] ]
])
gap> Size(m);
864
```

1.2.5 生成元と関係式での群 (Fp-群) の定義

生成元に対する関係式から、群を定義する場合は、まず自由群 f を関数 FreeGroup を使って定義して、その生成元を使って関係式を表します。一般に群 f について、その n 番目の生成元は、f.n と表すことが出来ます。最後に自由群をその関係式で割ることで、新しい群が構成できます。以下の例では、有限群 $2.A_8$ を定義してその位数を求めています。

```
gap>   f := FreeGroup( "m1","m2","m3","m4", "m5","m6");
<free group on the generators [ m1, m2, m3, m4, m5, m6 ]>
gap>   m1:=f.1;; m2:=f.2;; m3:=f.3;; m4:=f.4;;
gap>   m5:=f.5;; m6:=f.6;;
gap>   rel:=[
>   m1^6, m2^2*m1^3, m3^2*m1^3, m4^2*m1^3, m5^2*m1^3, m6^2*m1^3,
>   (m1*m2)^3*m1^3,
>   (m1*m3)^2*m1^3, (m1*m4)^2*m1^3, (m1*m5)^2*m1^3, (m1*m6)^2*m1^3,
>   (m2*m3)^3*m1^3, (m2*m4)^2*m1^3, (m2*m5)^2*m1^3, (m2*m6)^2*m1^3,
>                   (m3*m4)^3*m1^3, (m3*m5)^2*m1^3, (m3*m6)^2*m1^3,
>                                   (m4*m5)^3*m1^3, (m4*m6)^2*m1^3,
>                                                   (m5*m6)^3*m1^3 ];;
gap> g:=f/rel;
```

```
<fp group on the generators [ m1, m2, m3, m4, m5, m6 ]>
gap> Size(g);
40320
```

1.2.6 Power-Commutator 表現による群 (Pc-群) の定義

可解群を生成元と関係式で定義する場合、Power - Commutator 表現を使うと、可解群に関連した計算結果を素早く得られます。Power - Commutator 表現とは、関係式の表し方で、左辺が必ずある 1 つの生成元 a_i の冂か 2 つの生成元 a_i, a_j $(i \leqq j)$ の交換子になっているもので、そのとき対応する右辺は、$a_{i+1}^{\alpha_{i+1}} a_{i+2}^{\alpha_{i+2}} \cdots a_n^{\alpha_n}$ の形で与えられるものです。次の例では、4 つの生成元 a, b, c, d で、関係式が $a^2 = b^2 = c^2 = d^2 = 1$, $[b, a] = b$, $[c, a] = d$, $[c, b] = c * d$ $[d, a] = 1$, $[d, b] = c$, $[d, c] = 1$ となっている可解群を、Fp-群として定義した上で、関数 PcGroupFpGroup で、Pc-群に変換しています。

```
gap> F := FreeGroup("a","b","c","d");;
gap> a := F.1;; b := F.2;; c := F.3;; d := F.4;;
gap> rels := [a^2, b^3, c^2, d^2, Comm(b,a)/b, Comm(c,a)/d, Comm(c,b)/(c*d),
>              Comm(d,a), Comm(d,b)/c, Comm(d,c)];;
gap> G := F / rels;
<fp group on the generators [a,b,c,d]>
gap> H := PcGroupFpGroup( G );
<pc group of size 24 with 4 generators>
```

1.2.7 GAP の群ライブラリから群を呼び出す

群ライブラリから群を呼び出して定義する場合、Filter と呼ばれる属性を指定することが出来ます。主な Filter は、IsPermGroup, IsMatrixGroup, IsPcGroup などです。Filter を指定しないで、群ライブラリからの呼び出しを行なった場合は、それぞれの関数に設定されているデフォルトの Filter が適用されます。ただし、ライブラリの種類によって、使えない Filter もあります。

代表的な群

代表的な群を定義する関数としては、TrivialGroup, CyclicGroup, AbelianGroup, ElementaryAbelianGroup, DihedralGroup, ExtraspecialGroup, AlternatingGroup, SymmetricGroup, MathieuGroup, SuzukiGroup (Sz) などが上げられます。Filter として IsMatrixGroup を指定した場合は、さらに行列の成分が含まれる有限体も指定することが出来ます。

```
gap> CyclicGroup(12);
<pc group of size 12 with 3 generators>
gap> CyclicGroup(IsPermGroup,12);
Group( [ ( 1, 2, 3, 4, 5, 6, 7, 8, 9,10,11,12) ] )
gap> matgrp1:= CyclicGroup( IsMatrixGroup, 12 );
<matrix group of size 12 with 1 generators>
gap> FieldOfMatrixGroup( matgrp1 );
Rationals
gap> matgrp2:= CyclicGroup( IsMatrixGroup, GF(2), 12 );
<matrix group of size 12 with 1 generators>
gap> FieldOfMatrixGroup( matgrp2 );
```

```
GF(2)
```

古典群

次のようにして、一般線形群、シンプレスティク群そしてユニタリ群を定義出来ます。また関数 Action と作用に関するオプション OnLines を利用することで、線形群から射影線形群を作ることも出来ます。

```
gap> Gm:=GL(4,2);
SL(4,2)
gap> DisplayCompositionSeries(Gm);
G (size 20160)
 | A(8) ~ A(3,2) = L(4,2) ~ D(3,2) = O+(6,2)
1 (size 1)
gap> g:=GL(4,3);;Size(g);
24261120
gap> pgl:=Action(g,Orbit(g,Z(3)^0*[1,0,0,0],OnLines),OnLines);;
gap> Size(pgl);
12130560
```

選択関数

選択関数を使うことで、GAP の持つ群のライブラリから与えられた条件を満たすものだけを、選択して抽出することが出来ます。ここで与える条件とは、関数とその出力値（または出力値の範囲）の組です。最初の例では、関数 Size とリスト [4,8..60] (1.1.4 参照) で、位数が 60 以下の 4 の倍数となるものを条件として設定しています。更に、関数 f を使って、位数 2 の元の中心化群が位数 4 の基本可換群となる群だけを選ぶように設定しています。２番目の例は、次数が 4 から 10 までの原始的な置換群について、関数 f の条件を満たすものを選び出しています。計算結果に５次の交代群が５次、６次そして１０次の原始的な置換群として、登場しているのが分かります。

```
gap> f:=function(G)
> local cc,cc2,x,C;
> cc:=List(ConjugacyClasses(G),Representative);
> cc2:=Filtered(cc,x->Order(x)=2);
> for x in cc2 do
>   C:=Centralizer(G,x);
>   if Order(C)=4 and IsElementaryAbelian(C) then
>     return true;
>   fi;
> od;
> return false;
> end;
gap> ag:=AllGroups(Size,[4,8..60],f,true);;
gap> List(ag,Size);
[  4,  8, 12, 12, 16, 16, 20, 24, 24, 24, 28, 32, 32, 36, 36,
  36, 40, 40, 44, 48,  48, 48, 48, 48, 52, 56, 56, 60, 60, 60 ]
gap> Filtered(ag,IsSimple);
[ Alt( [ 1 .. 5 ] ) ]
gap> at:=AllPrimitiveGroups(NrMovedPoints,[4..10],f,true);
[ A(4), S(4), A(5), PSL(2,5), A(5) ]
gap> List(at,NrMovedPoints);
```

```
[ 4, 4, 5, 6, 10 ]
```

第 2 章

置換群の計算

2.1 置換の定義

GAP では、置換をそのまま表現できます。a:=(1,2); と定義するだけで a は、1 と 2 を入れ換える互換となります。 b:=(1,2,3); で、b は、1 を 2 に、2 を 3 に、3 を 1 に写す置換となります。実際に、b を 3 に作用させた結果を計算したい場合は、3^b; と入力すれば、GAP は、3 に b を作用させた結果を表示します。n 次の置換は、集合 $\Omega = \{1, 2 \cdots n\}$ から Ω への全単射写像なので、$1, 2, \cdots, n$ の行き先 $i_1,\ i_2,\ \cdots i_n$ が決まれば置換が決まります。そこで、リスト $[i_1, i_2, \cdots, i_n]$ から置換を作る関数 PermList が用意されています。また、置換から対応するリストを作りたい場合は ListPerm を使います。次の例では、リスト l から置換 a を作っています。また、置換 b から対応するリストを表示させています。最後に置換 a, b の型を関数 CycleStructurePerm を使って求めています。置換 a は、長さ 2 の巡回置換が **1** と長さ 3 の巡回置換が **1** なので、[1,1] と出力されます。置換 b は、長さ 2 の巡回置換が**無し**と長さ 3 の巡回置換が **2** なので、[,2] と出力されます。(もともと長さが 1 の巡回置換は存在しませんので、長さ 2 から始まっています。)

```
gap> l:=[2,3,1,5,4];;
gap> a:=PermList(l);
(1,2,3)(4,5)
gap> b:=(1,3,5)(2,4,6);;
gap> ListPerm(b);
[ 3, 4, 5, 6, 1, 2 ]
gap> CycleStructurePerm( a );
[ 1, 1 ]
gap> CycleStructurePerm( b );
[ , 2 ]
```

さて、b*a*b の型はどうなるでしょうか？ ノートに b*a*b を計算してから予想してみましょう！

恒等置換は () で表されます。置換 a の n 乗 a^n を求めたいときは、a^n を使います。また、置換の位数は、関数 Order を使って求めることが出来ます。下の例では、恒等置換を e として、置換 a, b の位数を求めて、位数分だけ巾乗すると恒等置換になることを確かめています。また、最後に置換 a, b の逆元も求めています。

```
gap> e:=();;
gap> Order(a);
6
```

```
gap> Order(b);
3
gap> a^6=e;
true
gap> b^3=e;
true
gap> a^-1;
(1,3,2)(4,5)
gap> b^-1;
(1,5,3)(2,6,4)
```

2.2　置換群の構成

この節では、GAP での群の計算を実感してもらうために、GAP のマニュアルにある Tutorial:Group and Homomorphisms の内容の 1 部を紹介して行きます。

2.2.1　置換群を定義する

ここでは、置換群を使っていくつかの簡単な計算をしてみましょう。

最初の例では、 2 つの置換 (1,2) と (1,2,3,4,5,6,7,8) で 生成される置換群 s8 が定義されています。(もちろんこれは、 8 次の対称群ですね)

```
gap> s8:= Group( (1,2), (1,2,3,4,5,6,7,8) );
Group( (1,2), (1,2,3,4,5,6,7,8) )
```

この s8 から例えば、s8 のすべての偶置換の群 (8 個の点の交代群) は s8 の交換子群を計算することで求めることが出来ます。

```
gap> a8:= CommutatorSubgroup( s8, s8 );
Group([ (1,2,3), (2,3,4), (3,4,5), (4,5,6), (5,6,7), (6,7,8) ])
```

交代群 a8 は 6 つの置換で生成される部分群として定義されました。余談ですが、このように生成元を沢山持つ群は、一般にその後のこの群に対する計算を遅くしてしまう可能性があります。関数 SmallGeneratingSet を使うことで、生成元の個数を減らすことが出来ます。

```
gap> SmallGeneratingSet(a8);
[ (1,3)(4,6,5,8), (1,5,6,2,3)(4,8,7) ]
```

2.2.2　置換群の性質を調べる

それでは、定義した置換群の基本的な性質を調べてみましょう。a8 の位数、可換群であるかどうか、完全群かどうかなどが次のようにして計算できます。

```
gap> Size( a8 ); IsAbelian( a8 ); IsPerfect( a8 );
20160
false
true
```

また、群の位数の約数となる素数 p に対するシロー p-部分群 は、関数 SylowSubgroup を呼ぶことで求めることが出来ます。ここでは、FactorsInt を使って位数を素因数分解したときの、素因数をまず計算しています。

```
gap> Set( FactorsInt( Size( a8 ) ) );
[ 2, 3, 5, 7 ]
gap> syl2:=SylowSubgroup( a8, 2 );
Group([ (1,8)(6,7), (2,3)(6,7), (4,5)(6,7), (2,4)(3,5), (1,6)(7,8),
  (1,2)(3,8)(4,6)(5,7) ])
```

上の例ではシロー 2-部分群が計算され変数 syl2 に入りました。次の計算はこの syl2 に対してその位数、正規化群、中心、その中心の元の中心化群 cent、更に この cent の交換子群列と降中心列が計算されています。変数 last は、1 つ前の計算結果を表しています。

```
gap> Size( syl2 );
64
gap> Normalizer( a8, syl2 );
Group([ (1,8)(6,7), (4,5)(6,7), (2,3)(6,7), (2,4)(3,5), (1,6)(7,8),
  (1,5)(2,7)(3,6)(4,8) ])
gap> last = syl2;
true
gap> Centre( syl2 );
Group([ ( 1, 8)( 2, 3)( 4, 5)( 6, 7) ])
gap> cent:= Centralizer( a8, last );
Group([ (4,5)(6,7), (4,6)(5,7), (2,3)(6,7), (2,4)(3,5), (1,2)(3,8) ])
gap> Size( cent );
192
gap> DerivedSeries( cent );
[ Group([ (4,6)(5,7), (2,6,4)(3,7,5), (1,4)(2,6)(3,7)(5,8),
      (1,2)(3,8)(4,6)(5,7), (4,5)(6,7), (2,3)(4,5), (1,8)(2,3)(4,5)(6,7)]),
  Group([ (2,6,4)(3,7,5), (1,4)(2,6)(3,7)(5,8), (1,2)(3,8)(4,6)(5,7),
      (4,5)(6,7), (2,3)(4,5), (1,8)(2,3)(4,5)(6,7) ]),
  Group([ (1,4)(2,6)(3,7)(5,8), (1,2)(3,8)(4,6)(5,7), (4,5)(6,7),
(2,3)(4,5), (1,8)(2,3)(4,5)(6,7) ]), Group([ (1,8)(2,3)(4,5)(6,7) ]),
Group(()) ]
gap> List( last, Size );
[ 192, 96, 32, 2, 1 ]
gap> low:= LowerCentralSeries( cent );
[ Group([ (4,5)(6,7), (4,6)(5,7), (2,3)(6,7), (2,4)(3,5), (1,2)(3,8) ]),
  Group([ (2,6,4)(3,7,5), (1,4)(2,6)(3,7)(5,8), (1,2)(3,8)(4,6)(5,7),
      (4,5)(6,7), (2,3)(4,5), (1,8)(2,3)(4,5)(6,7) ]) ]
```

ある一点を固定させる置換の集合で与えられる部分群 (固定部分群) も計算できます。

```
gap> stab:= Stabilizer( a8, 1 );
Group([ (2,3,4), (3,4,5), (4,5,6), (5,6,7), (6,7,8) ])
gap> Size( stab );
2520
gap> Index( a8, stab );
8
```

2.2.3　置換群の元に対する計算

例えば、群の任意の元をランダム取ってくることが出来ます。 また、ある元の中心化群を計算したり、 部分群の共役を取ったり、 2 つの部分群の共通部分を求めることも出来ます。

```
gap> Random( a8 );
(1,6,3,2,7)(4,5,8)
gap> Random( a8 );
(1,3,2,4,7,5,6)
gap> cent:= Centralizer( a8, (1,2)(3,4)(5,8)(6,7) );
Group([ (5,6)(7,8), (5,7)(6,8), (3,4)(6,7), (3,5)(4,8), (1,2)(6,7),
  (1,3)(2,4) ])
gap> Size( cent );
192
gap> conj:= ConjugateSubgroup( cent, (2,3,4) );
Group([ (5,6)(7,8), (5,7)(6,8), (2,4)(6,7), (2,8)(4,5), (1,3)(6,7),
  (1,4)(2,3) ])
gap> inter:= Intersection( cent, conj );
Group([ (5,7)(6,8), (5,8)(6,7), (1,2)(3,4)(5,7)(6,8),(1,3)(2,4)(5,7)(6,8) ])
gap> Size( inter );
16
gap> IsElementaryAbelian( inter );
true
gap> norm:= Normalizer( a8, inter );
Group([ (5,7)(6,8), (5,8)(6,7), (1,2)(3,4)(5,7)(6,8),
(1,3)(2,4)(5,7)(6,8), (5,7,8), (3,4)(5,7,8,6), (2,3)(5,7,8,6),
(1,5,4,8,2,6)(3,7) ])
gap> Size( norm );
576
```

2.2.4　部分群の構成

例えば、a8 における $2^3 : L_3(2)$ のような構造を持つ部分群を構成したいといった場合を考えてみましょう。固定点を持たない 3 つの involution(位数が 2 の元) から生成される基本可換群 elab を作り、その a8 の正規化群 norm を計算するのが 1 つの方法です。elab は、2^3 の部分になります。もし、norm が、$2^3 : L_3(2)$ であれば、 $L_3(2)$ の位数が 168 より norm の位数は、$168 \times 8 = 1344$ となるはずです。

```
gap> elab:= Group( (1,2)(3,4)(5,6)(7,8), (1,3)(2,4)(5,7)(6,8),
>                  (1,5)(2,6)(3,7)(4,8) );;
gap> Size( elab );
8
gap> IsElementaryAbelian( elab );
true
gap> SetName( elab, "2^3" ); elab;
2^3
gap> norm:= Normalizer( a8, elab );;
```

```
gap> Size( norm );
1344
```

次に剰余群を計算してみましょう。norm という群と、それの剰余群 f は、f への自然な準同形写像 (核が elab となる準同形写像) hom で結ばれています。GAP は、その後の剰余群の計算を楽にするため出来るだけ剰余群 f の次数が小さくなるようにしています。そのため、f の作用域と norm との間には何の係わり合いもありません。剰余群 f の元に対する準同形写像 hom の逆像は norm の右剰余類となります。

```
gap> hom:=NaturalHomomorphismByNormalSubgroup( norm , elab );
<action homomorphism>
gap> f := Image( hom );
Group([(),(),(),(1,2)(3,4),(1,3)(2,4),(1,2)(5,6),(3,5)(4,6),(2,3)(6,7)])
gap> Size( f );
168
gap> Kernel( hom ) = elab;
true
gap> x:= Random( norm );
(1,7,4,3,6,8,5)
gap> Image( hom, x );
(1,2,3,5,4,7,6)
gap> coset:= PreImages( hom, last );
RightCoset(Group( [ (1,2)(3,4)(5,6)(7,8), (1,3)(2,4)(5,7)(6,8),
  (1,5)(2,6)(3,7)(4,8) ] ),(2,8,3,4,5,7,6))
gap> IsRightCoset( coset );
true
```

出力が示すように、でき上がった剰余群は、また置換群になります。norm の 8 つの生成元が剰余群でどの元に対応しているのかが分かります。最初の 3 つの元は、elab に含まれていたことになります。得られた群 f が、ほんとうに $L_3(2)$ であるか確認するため、関数 IsSimple で単純群であることをそして関数 IsomorphismTypeInfoFiniteSimpleGroup で具体的に単純群の種類を求めています。

```
gap> IsSimple( f ); IsomorphismTypeInfoFiniteSimpleGroup( f );
true
rec( series := "L", parameter := [ 2, 7 ],
  name := "A(1,7) = L(2,7) ~ B(1,7) = O(3,7) ~ C(1,7) = S(2,7) ~
  2A(1,7) = U(2\
,7) ~ A(2,2) = L(3,2)" )
gap> SetName( f, "L_3(2)" );
```

今回の剰余群 f は、8 次の置換群として定義されていますが、f の元全体の集合を作用域として右から乗法で作用を定義することで、f の正則置換表現が得られます。

```
gap> op:= Operation( f, f , OnRight );;
gap> IsPermGroup( op );
true
gap> IsSimpleGroup( op );
true
```

norm の 7 つの自明でない元の正規部分群 elab への共役による作用は $L_3(2)$ の 7 つの

点への表現を与えます。この置換群を norm に忠実に埋め込むことで、 norm が基本的可換群 2^3 の $L_3(2)$ と同型な norm の部分群の半直積となっていることが分かります。

```
gap> op := Action( norm, elab );
Group( [ (), (), (), (5,6)(7,8), (5,7)(6,8), (3,4)(7,8), (3,5)(4,6),
  (2,3)(6,7) ] )
gap> IsSubgroup( a8, op ); IsSubgroup( norm, op );
true
true
gap> IsTrivial( Intersection( elab, op ) );
true
gap> SetName( norm, "2^3:L_3(2)" );
```

もう 1 つ別の見方をすると、正規部分群 elab は、正則な (つまり $\{1, 2 \cdots 8\}$ に可移で忠実に作用している）位数が 8 の部分群なので、norm は、elab と 1 点固定部分群との半直積となります。

2.2.5　群の作用

置換群 a8 の別の置換表現を得るために、a8 の 1 つの共役類に対する a8 の作用を考えてみましょう。

```
gap> ccl := ConjugacyClasses( a8 );; Length( ccl );
14
gap> List( ccl, c -> Order( Representative( c ) ) );
[ 1, 2, 2, 3, 6, 3, 4, 4, 5, 15, 15, 6, 7, 7 ]
gap> List( ccl, Size );
[ 1,210,105,112,1680,1120,2520,1260,1344,1344,1344,3360,2880,2880 ]
```

共役類の集合 ccl は、共役類のリストになってますが、ここでは、その中で、元の個数が 112 の位数が 3 の元の共役類に注目してみましょう。

```
gap> class := First( ccl, c -> Size(c) = 112 );;
gap> op := Action( a8, AsList( class ) );;
```

関数 First は、共役類のリスト ccl の要素 (つまり共役類）のうち元の個数が 112 となる最初の要素 (つまり共役類）を出します。共役類 class と AsList(class) は、本質的には何等変わりない位数 3 の共役類ですが、実質的には class の方は基になっている群、代表元、中心化群の位数、この共役類に含まれる元の個数、などの情報を集めたものであるのに対して、AsList(class) は、まさに、この共役類に含まれる元をリストとして列挙したものとなってます。この計算によりこの 112 個の元に作用する置換群 op が作られました。まずこの置換群が 原始的であるか確認してみましょう。置換群が原始的で無い場合は、関数 Blocks を使って最小ブロックシステムを計算することが出来ます。

```
gap> IsPrimitive( op, [ 1 .. 112 ] );
false
gap> blocks := Blocks( op, [ 1 .. 112 ] );
[[1,7],[2,13],[3,19],[8,14],[9,20],[16,27],[23,34],[15,21],[10,26],
[43,48],[4,25],[5,31],[44,53],[74,81],[22,28],[17,33],[18,39],
[79,86],[11,32],[73,77],[45,58],[6,37],[50,59],[56,65],[12,38],
```

```
[49,54],[55,60],[51,64],[52,69],[78,82],[30,41],[75,85],[29,35],
[76,89],[24,40],[36,42],[46,63],[47,68],[93,96],[94,99],[84,91],
[97,100],[105,107],[80,90],[98,103],[108,110],[62,71],[61,66],
[83,87],[57,70],[67,72],[88,92],[95,102],[106,109],[101,104],[111,112]]
```

上の計算で、作用域 [1..112] を指定していることに注意して下さい。作用域は、置換群から簡単に決定できると思われるかも知れませんが、例えば元 $(2, 3, 4)$ で生成された置換群の作用域は [1..4] かも知れないし、[2..5] かも知れません。上の例で作用域を [1..113] とすれば、群 op は、 原始的どころか、可移にすらなりません。変数 blocks には、ブロックのリストが入ります。次の計算では、個々のブロックの大きさとブロックの総数を求めて、更にこの blocks に対する op の作用によって出来る置換群 op2 を求めています。今回の作用は、個々のブロックを作用域の 1 つ元のと見て、そのブロックに op を作用させ、別のどのブロックに、移るのかを見ています。ブロックの総数が 56 なので、op2 の作用域は [1..56] となります。

```
gap> Length( blocks[1] );  Length( blocks );
2
56
gap> op2 := Action( op, blocks, OnSets );;
gap> IsPrimitive( op2, [ 1 .. 56 ] );
true
```

以上の方法で次数が 56 の 原始的な置換群が得られました。この primitivity より、この群 op2 の 1 点固定部分群は極大部分群となります。そこで、a8 から op2 までを結ぶ準同形写像を計算し、この 1 点固定部分群の逆像を求める事で a8 の別の極大部分群を求めてみましょう。

```
gap> ophom := ActionHomomorphism( a8, op );;
gap> ophom2 := ActionHomomorphism( op, op2 );;
gap> composition := ophom * ophom2;;
gap> stab := Stabilizer( op2, 2 );;
gap> preim := PreImages( composition, stab );
Group([(2,5,7),(1,4)(2,7),(2,6,7),(1,3)(5,7),(6,8,7)])
```

このように新しい作用を構成することで、置換群の別の姿を見いだすことが出来ます。そしてそれらの橋渡しをするのが、準同形写像と言うことが出来ます。実は、2.2.4 で作った自然な準同形写像 hom も、ある作用域に対する作用からでき上がっています。それが何かを見るためには、次のようなコマンドを入力する必要があります。

```
gap> t := UnderlyingExternalSet( hom );
<xset:RightTransversal(2^3:L_3(2), Group([(5,6)(7,8),
  (5,7)(6,8),(3,4)(7,8),(3,5)(4,6),(1,2)(7,8),(1,3)(2,4)]))>
```

ここに現れた t は、外部集合 (external sets) と呼ばれるもので、作用域と作用している群さらに作用の方法を規定している関数 (つまり作用している群と作用域の直積から作用域への写像) の 3 つを 1 つにまとめたものとなります。今回の例では、位数が 192 の norm の部分群

```
Group([(5,6)(7,8),(5,7)(6,8),(3,4)(7,8),(3,5)(4,6),
                                (1,2)(7,8),(1,3)(2,4)])
```

の右剰余類が作用域であることが分かります。

2.2.6　固定部分群

利用者が独自に群の作用域と作用を定義することももちろん出来ます。ここでは、a8 の部分群を、ある作用域の 1 点固定部分群として構成してみましょう。まず簡単な 8 点の集合を作用域としたときの 1 点固定部分群を見て行きましょう。

```
gap> u8 := Stabilizer( a8, 1 );
Group([(2,3,4),(2,4)(3,5),(2,6,4),(2,4)(5,7),(2,8,6,4)(3,5)])
gap> Index( a8, u8 );
8
gap> Orbit( a8, 1 ); Length( last );
[ 1, 3, 2, 4, 5, 6, 7, 8 ]
8
```

この計算例は、さらに a8 の部分群たちを見つけるためのヒントを与えています。というのも、すべての部分群は、ある作用域 (具体的にはその部分群の剰余類の集合) に対する 1 点固定部分群として与えられるいるからです。ですから、部分群を見つけることは、対応する作用域 (作用) を見つけることにほかなりません。 では、べつの作用を作るため、まず 1 から 8 までの数字の組を作ることから始めましょう。

```
gap> pairs := Tuples( [1..8], 2 );;
```

この数字の組を作用域として a8 の作用を考えてみましょう。ここでは、作用の方法を明示するため、OnPairs というオプションを付け加えて、数字の組 pairs に対する a8 の作用が可移であるか如何か聞いています。

```
gap> IsTransitive( a8, pairs, OnPairs );
false
```

作用は当然可移でないので、その軌跡を計算します。

```
gap> orbs := Orbits( a8, pairs, OnPairs );; Length( orbs );
2
gap> List( orbs, Length );
[ 8, 56 ]
gap> List( orbs, o -> o[1] );
[ [ 1, 1 ], [ 1, 2 ] ]
```

この計算により、a8 の pair 上での作用は長さ 8 と 56 の 2 つの軌跡を持ち、それぞれの軌跡の代表が [1,1] と [1,2] であることが分かりました。それでは、2 番目の軌跡についてその代表 [1,2] を固定する 1 点固定部分群を計算しましょう。

```
gap> u56 := Stabilizer( a8, orbs[2][1], OnPairs );; Index( a8, u56 );
56
```

さあこれで、2 つめの部分群が得られました。次に以後の計算をやりやすくするため、2 番目の軌跡である 56 個の数字の組に対する a8 の作用を、関数 ActionHomomorphism 使って、56 個の点に対する作用に変換してみましょう。でき上がった群 a8_56 は、56 個の点に作用する置換群となります。

```
gap> h56 := ActionHomomorphism( a8, orbs[2], OnPairs );;
gap> a8_56 := Image( h56 );;
```

もし、この置換群 a8_56 が原始的であれば、その 1 点固定部分群に対応する u56 は、a8 の極大部分群となりますが...

```
gap> IsPrimitive( a8_56, [1..56] );
false
```

残念ながら、u56 は、極大部分群では無いようです。では、当然 u56 を含むような a8 の部分群 (super groups of u56 in a8) を見つけたくなりますね。a8_56 が原始的でないので、関数 Blocks を使って、a8_56 の作用に対するブロックシステムを求めることが出来ます。

```
gap> blocks := Blocks( a8_56, [1..56], [1,14] );
[ [ 1, 11, 13, 14, 39, 44, 45 ], [ 2, 7, 15, 16, 30, 31, 46 ],
  [ 3, 8, 17, 25, 28, 41, 42 ], [ 4, 6, 24, 27, 29, 43, 52 ],
  [ 5, 12, 18, 26, 34, 40, 54 ], [ 9, 19, 21, 32, 37, 49, 53 ],
  [ 10, 20, 23, 33, 36, 48, 55 ], [ 22, 35, 38, 47, 50, 51, 56 ] ]
```

上の計算では、作用域 [1..56] をブロックに分解していますが、ただしその中の 1 つのブロックには 1 と 14 が含まれるように注文を付けています。この条件で、作用域 [1..56] を 8 つのブロックに分解することが出来ました。では、この 8 つのブロックを作用域としてそのうち 1 つめのブロックを固定する固定部分群を計算してみましょう。それぞれのブロックは、1 つの集合と考えて個々の集合単位で作用を行なうため、オプション OnSets を付け加えておきます。

```
gap> u8_56 := Stabilizer( a8_56, blocks[1], OnSets );;
gap> Index( a8_56, u8_56 );
8
gap> u8b := PreImages( h56, u8_56 );; Index( a8, u8b );
8
gap> IsConjugate( a8, u8, u8b );
true
```

上の計算では、この 1 点固定部分群 u8_56 から、逆像として a8 の部分群 u8b を得て、以前計算した u8 と u8b が共役である事も示しています。(実は、これはそれほどビックリすることではありませんというのも、u56 は、もともと a8 の 2 点固定部分群と思えるので、それを含む部分群として 1 点固定部分群が出てきたわけです。) 実は、次のようにすると別のブロックシステムを得ることが出来ます。

```
gap> blocks := Blocks( a8_56, [1..56], [1,7] );;
gap> u28_56 := Stabilizer( a8_56, [1,7], OnSets );;
gap> u28 := PreImages( h56, u28_56 );;
gap> Index( a8, u28 );
28
```

この指数が 28 の部分群が、極大部分群となることは a8 が 2,4,7 を指数とする部分群を持たないと云う事からあきらかですが、具体的に a8_56 の blocks 上での作用が原始的であることを確認することで確かめることが出来ます。

```
gap> IsPrimitive( a8_56, blocks, OnSets );
true
```

固定部分群を計算する関数 Stabilizer は、部分群にも適用できます。次の計算では、部分群 u56 の部分群で点 3 を固定するものを計算しています。(u56 は、点 1,2 を固定しますので、u336 は、3 つの点 1,2,3 を固定します。)

```
gap> u336 := Stabilizer( u56, 3 );;
gap> Index( a8, u336 );
336
```

他の関数もまた部分群に適用可能です。以下の計算では、u336 が、点 4〜点 8 の 5 つの点の 3 ツ組の集合 (元の個数は $5 \times 4 \times 3 = 60$ 個) に正則に作用していることが分かります。

```
gap> IsRegular( u336, Orbit( u336, [4,5,6], OnTuples ), OnTuples );
true
```

以前数字の組でやったように、次の計算で u336 の右剰余類の集合に対する作用を通じて、作用域 [1..336] に作用する新しい置換群を作ることが出来ます。

```
gap> t := RightTransversal( a8, u336 );;
gap> a8_336 := Action( a8, t, OnRight );;
```

この u336 を含むような部分群を得るため、別の自明でないブロックシステムを見つけます。

```
gap> blocks := Blocks( a8_336, [1..336] );; blocks[1];
[ 1, 43, 85 ]
```

つまり、u336 と 43 番目と 85 番目の右剰余類の和集合は指数 112 の部分群となることが分かります。そこで、次のように u336 の 43 番目の右剰余類の代表元を加えて生成される部分群を作れば指数 112 のこの部分群が得られます。

```
gap> u112 := ClosureGroup( u336, t[43] );;
gap> Index( a8, u112 );
112
```

同様に u112 を含む部分群 u56b を得ることが出来ますが、こちらは u56 とは、共役にならないばかりか、極大部分群となってしまうことが分かります。

```
gap> blocks := Blocks( a8_336, [1..336], [1,7,43] );;
gap> Length( blocks );
56
gap> u56b := ClosureGroup( u112, t[7] );; Index( a8, u56b );
56
gap> IsPrimitive( a8_336, blocks, OnSets );
true
```

2.2.5 で紹介したように、共役も 1 つの群の作用となります。作用を指定しないで関数 OrbitLength を使った場合は、置換群の元に対するデフォルトの作用である共役が採用されます。

```
gap> OrbitLength( a8, (1,2)(3,4)(5,6)(7,8) );
105
gap> orb := Orbit( a8, (1,2)(3,4)(5,6)(7,8) );;
gap> u105 := Stabilizer( a8, (1,2)(3,4)(5,6)(7,8) );; Index( a8, u105 );
105
```

注意：群 G の元 elm を代表元とする共役類の大きさは、OrbitLength(G,elm) で計算できますが、Size(ConjugacyClass(G, elm)) を使った方が計算効率が良いようです。

```
gap> Size( ConjugacyClass( a8, (1,2)(3,4)(5,6)(7,8) ) );
105
```

もちろん、u105 は、元 (1,2)(3,4)(5,6)(7,8) の中心化群です。では、いままでと同様に u105 を含むような部分群を見つめましょう。

```
gap> blocks := Blocks( a8, orb );; Length( blocks );
15
gap> blocks[1];
[ (1,2)(3,4)(5,6)(7,8), (1,3)(2,4)(5,8)(6,7), (1,8)(2,7)(3,5)(4,6),
  (1,7)(2,8)(3,6)(4,5), (1,5)(2,6)(3,8)(4,7), (1,6)(2,5)(3,7)(4,8),
  (1,4)(2,3)(5,7)(6,8)]
```

u105 を含む指数が 15 の部分群を作るために、ふたたび閉包を使います。ここからは、誤解や混乱を招かないように慎重に話を進めて行きたいと思いますが、もう少しこちらの話につきあってください。

部分群 u105 は、点 (1,2)(3,4)(5,6)(7,8) の一点固定部分群として定義されました。このとき、点 (1,2)(3,4)(5,6)(7,8) の軌跡の元と u105 の剰余類の間に一対一の対応ができ上がっています。点 (1,2)(3,4)(5,6)(7,8) には、u105 が対応しています。u105 を含むような指数 15 の a8 の部分群を得るためには、最初のブロック blocks[1] に含まれる (1,2)(3,4)(5,6)(7,8) 以外の点に対応する、剰余類の元を 1 つ u105 に加えるて閉包を取ることによって得られます。仮に、(1,2)(3,4)(5,6)(7,8) 以外の点として、点 (1,3)(2,4)(5,8)(6,7) を採用すると、a8 の元で共役の作用で (1,2)(3,4)(5,6)(7,8) を (1,3)(2,4)(5,8)(6,7) に移す元を見つける必要があります。そして、関数 RepresentativeAction は、まさにそれをしてくれるものです。この関数は、2 つの点と 1 つの群を与えて群の元の中で、1 つ目の元を 2 つめの元に移す元を見つけます。実は、4 番目の引数として作用の方法も指定できますが、今回の例では、この引数を必要としません。もし、2 つの点を共役で結ぶような a8 の元が見つからないとき、関数 RepresentativeAction は、False を結果として出力します。

```
gap> rep := RepresentativeAction( a8, (1,2)(3,4)(5,6)(7,8),
>                                      (1,3)(2,4)(5,8)(6,7) );
(2,3)(6,8)
gap> u15 := ClosureGroup( u105, rep );; Index( a8, u15 );
15
```

群 a8 は、指数 5,3 の部分群を持たないので、ここでで き上がった指数 15 の部分群 u15 は、極大部分群となります。また、元 (2,3)(6,7) を、u105 に加えることで、別の指数 15 の部分群 u15b を得ることが出来ます。

```
gap> u15b := ClosureGroup( u105, (2,3)(6,7) );; Index( a8, u15b );
15
gap> RepresentativeAction( a8, u15, u15b );
fail
```

関数 RepresentativeAction は、u15 と u15b は、非共役であることを示しています。

第 3 章

有限群の既約表現構成

この章では、有限群の既約表現を指標の情報を元に構成していきたいと思います。

3.1 有限群の表現

3.1.1 表現の定義

有限群 G のある体 k 上の n 次表現とは、G から $GL(n,k)$ への準同型写像のことです。G の n 次表現 R と k の n 次正則行列 P が与えられたとき、元 $x \in G$ に対して、$PR(x)P^{-1}$ を対応させる写像 R' も、n 次表現とな、R と R' は、同型な表現と呼ばれます。もし、G の各元 x に対して、表現 R の像 $R(x)$ が、2 つの正方行列 $S(x)$ と $T(x)$ を使って、

$$R(x) = \begin{pmatrix} S(x) & 0 \\ * & T(x) \end{pmatrix}$$

と表されたとき、写像 S と T はやはり表現となります。このとき、この表現 R と同型な表現を可約な表現と呼びます。また、可約でない表現を既約な表現と呼びます。正則行列 P を上手に選ぶことで、表現 R は、既約な表現 S_i $(i = 1, \ldots, r)$ を使って、

$$PR(x)P^{-1} = \begin{pmatrix} S_1(x) & 0 & \cdots & 0 \\ * & S_2(x) & \ddots & \vdots \\ * & * & \ddots & 0 \\ * & * & * & S_r(x) \end{pmatrix}$$

と表されたとき、この表現 S_i を表現 R の既約因子と呼びます。同型な表現は、同じ既約因子を持ちます。

体 k の標数が 0 （例えば k が複素数体の部分体）の場合、有限群 G の既約な表現の同型類の個数は、共役類の個数と一致することが知られています。

よって、有限群の表現を解析することは、既約な表現を調べることから始めるのが、

とても自然です。それでは、GAP を使って表現を作ってみましょう。表現 R が準同型写像であることから、まずは有限群 G の生成元 x に対応する行列 $R(x)$ をきちんと定義できれば、十分です。次の例では、3 次の対称群 `G` の生成元 `a=(1,2,3)`, `b=(1,2)` に、関数 `PermutationMat` を使ってその置換行列 `ma`, `mb` を対応させることで、表現を作っています。関数 `GroupHomomorphismByImages` を使うことで、群の準同型写像として、表現 `R` を定義できました。更に、適当な正則行列 `P` を与えて、この表現を 2 つの既約な既約表現に分解することが出来ます。

```
gap> G:=SymmetricGroup(3);
Sym( [ 1 .. 3 ] )
gap> a:=G.1;; b:=G.2;;
gap> ma:=PermutationMat(a,3,Rationals);;
gap> mb:=PermutationMat(b,3,Rationals);;
gap> Display(ma);
[ [  0,  1,  0 ],
  [  0,  0,  1 ],
  [  1,  0,  0 ] ]
gap> Display(mb);
[ [  0,  1,  0 ],
  [  1,  0,  0 ],
  [  0,  0,  1 ] ]
gap> R:=GroupHomomorphismByImages(G,Group([ma,mb]),[a,b],[ma,mb]);
[ (1,2,3), (1,2) ] -> [ [ [ 0, 1, 0 ], [ 0, 0, 1 ], [ 1, 0, 0 ] ],
  [ [ 0, 1, 0 ], [ 1, 0, 0 ], [ 0, 0, 1 ] ] ]

gap> P:=[[1,1,1],[1,0,-1],[0,1,-1]];
[ [ 1, 1, 1 ], [ 1, 0, -1 ], [ 0, 1, -1 ] ]
gap> Display(P*ma*P^-1);
[ [   1,   0,   0 ],
  [   0,  -1,   1 ],
  [   0,  -1,   0 ] ]
gap> Display(P*mb*P^-1);
[ [  1,  0,  0 ],
  [  0,  0,  1 ],
  [  0,  1,  0 ] ]
```

3.1.2　表現と指標の関係

G の n 次表現 R が与えられたとき、各元 x に対して、$\mathrm{Trace}(R(x))$ を対応させる写像を、表現 R に対応する G の指標と呼び χ_R と表すことにします。元 $x, y \in G$ に対して、Trace の性質から

$$\mathrm{Trace}(R(y^{-1}xy)) = \mathrm{Trace}(R(y)^{-1}R(x)R(y))) = \mathrm{Trace}(R(x))$$

となるので、指標の値は、G の共役類毎に値をとります。また、同型な表現の指標は一致し、表現 R の既約因子が、S_i $(i = 1, \ldots, r)$ なら、元 $x \in G$ に対して、

$$\chi_R(x) = \sum_{i=1}^{r} \chi_{S_i}(x)$$

が成り立ちます。つまり、指標の値から、表現 R が既約因子を持っているのかを予想することが、可能となります。特に、体 k の標数が 0 の場合は、指標から既約因子を確定することが可能です。

GAP では、関数 `CharactarTable` を使って既約な指標の各共役類での値を表にまとめた指標表を計算することが出来ます。先ほどの 3 次の対称群 `G` の指標表を計算すると次のように表示されます。この表から `G` には、3 つの既約な表現に対応する指標 `X.1`, `X.2`, `X.3` が存在することや 3 の共役類 `1a`（単位元）, `2a`,（`(1,2)` を含む共役類）, `3a`（`(1,2,3)` を含む共役類） でのそれぞれの指標の値がなんなのかが分かります。例えば、`G` の生成元 `a`, `b` について、それぞれの指標の値は、

`X.1(b)=-1`,`X.1(a)=1`,
`X.2(b)=0`, `X.2(a)=-1`,
`X.3(b)=1`, `X.2(a)=1`,
となっています。

3.1.1 の表現 R の分解を見ると、この表現が、X.2 と X.3 の指標に対応する既約表現を、既約因子として持つことが見て取れます。

```
gap> Display(CharacterTable(G));
CT1

     2  1  1  .
     3  1  .  1

       1a 2a 3a
    2P 1a 1a 3a
    3P 1a 2a 1a

X.1     1 -1  1
X.2     2  . -1
X.3     1  1  1
```

3.1.3 部分群を使った表現構成

有限群 G とその部分群 H、さらに H の表現 X が与えられたとき、G の H による右剰余類 Hg_i (i=1,...h) の代表系 g_i を用いて、次のように G の表現を構成することが出来ます。この表現を、H の表現 X による G の誘導表現と呼び、X^G と書くことにします。

$$X^G(x) := \begin{pmatrix} \tilde{X}(g_1 x g_1^{-1}) & \cdots & \tilde{X}(g_1 x g_h^{-1}) \\ \vdots & \ddots & \vdots \\ \tilde{X}(g_h x g_1^{-1}) & \cdots & \tilde{X}(g_h x g_h^{-1}) \end{pmatrix}$$

ただし、ここで写像 $\tilde{X}$ は、

$$\tilde{X}(x) = \begin{cases} X(x) & x \in H \\ 0 & x \notin H \end{cases}$$

で定義します。

GAP では、有限群 G と部分群 H の表現 r が与えられたとき、関数 InducedRepresentation で、誘導表現 r^G を作ることが出来ます。次の例では、(1,2) で生成される G の部分群の表現 r を定義して、その誘導表現 R を作っています。表現の Trace を計算すると、この表現 R が、X.1 と X.2 の指標に対応する既約表現を、既約因子として持つことが見て取れます。

```
gap> r:=GroupHomomorphismByImages(Group([b]),Group([[[-1]]]),[b],[[[-1]]]);
[ (1,2) ] -> [ [ [ -1 ] ] ]
gap> R:=InducedRepresentation(r,G);
[ (1,2,3), (1,2) ] -> [ <block matrix of dimensions (3*1)x(3*1)>,
  <block matrix of dimensions (3*1)x(3*1)> ]
gap> Display(Image(R,a));
[ [   0,  -1,   0 ],
  [   0,   0,  -1 ],
  [   1,   0,   0 ] ]
gap> Display(Image(R,b));
[ [  -1,   0,   0 ],
  [   0,   0,   1 ],
  [   0,   1,   0 ] ]
```

3.2　5 次の交代群の既約表現

それでは、GAP を使って 5 次交代群 G の既約表現を、すべて求めてみましょう。まずは、指標表を計算して、どんな既約指標がいくつあるかを見ていきましょう。表現が準同型写像であることから、単位元の表現は、単位行列になります。よって指標表の第 1 列目（単位元 1a での値）は、表現の次数を表してみます。次の計算で、5 次交代群には、5 つの既約表現がありそれぞれの次数は、1,3,3,4,5 であることが分かります。また 3 次既約指標の位数が 5 の元での値は、1 の原始 5 乗根 E(5) を使って表される 2 つの実数 A = -E(5)-E(5)^4 = $\frac{1-\sqrt{5}}{2}$, *A = -E(5)^2-E(5)^3 = $\frac{1+\sqrt{5}}{2}$ であることが分かります。

```
gap> G:=AlternatingGroup(5);;
gap> ctG:=CharacterTable(G);;
gap> Display(ctG);
CT5

     2  2  2  .  .  .
     3  1  .  1  .  .
     5  1  .  .  1  1

       1a 2a 3a 5a 5b
    2P 1a 1a 3a 5b 5a
    3P 1a 2a 1a 5b 5a
    5P 1a 2a 3a 1a 1a

X.1     1  1  1  1  1
X.2     3 -1  .  A *A
X.3     3 -1  . *A  A
X.4     4  .  1 -1 -1
X.5     5  1 -1  .  .

A = -E(5)-E(5)^4
  = (1-ER(5))/2 = -b5
```

3.2.1　置換表現から 4 次既約表現を作る

交代群 G を生成する置換 a,b,c から 3.1.1 のように置換行列を使って表現を作ると、5 次の表現が出来上がります。正則行列 P を使って、4 次の既約表現 S4 を抽出しています。

```
gap> a:=(1,2,3,4,5);; b:=(2,5)(3,4);; c:=(1,2,3);;
gap> R5:=List([a,b,c],x->PermutationMat(x,5,Rationals));;
gap> P:=[[ 1, 1, 1, 1, 1],
>        [ 1,-1, 0, 0, 0],
>        [ 1, 0,-1, 0, 0],
>        [ 1, 0, 0,-1, 0],
>        [ 1, 0, 0, 0,-1]];;
gap> R14:=List(R5,x->P*x*P^-1);;
gap> S4:=List(R14,x->x{[2..5]}{[2..5]});;
```

確認のため、指標を計算してみましょう。表現 S4 の指標の値は、次のようにして求めることが出来ます。

```
gap> cc:=[(),b,c,a,(1,2,3,5,4)];;
gap> R:=GroupHomomorphismByImages(G,Group(S4),[a,b,c],S4);;
gap> List(cc,x->Trace(Image(R,x)));
[ 4, 0, 1, -1, -1 ]
```

3.2.2 誘導表現から残りの既約表現を作る

最後に、元 a, b で生成される部分群 H の表現から残りの既約表現を求めます。まず、H の指標を計算して、それぞれの指標を関数 InducedClassFunctions を使って誘導表現に対応する指標（誘導指標）を計算します。（部分群との間の共役類の関係が分かると誘導指標は簡単に計算できます）次に既約指標の間にある直交関係を利用して、誘導指標の中に含まれる既約指標（既約因子）を関数 MatScalarProducts を使って求めます。この計算で、H の 1 次の表現 2 つを G に誘導して得られる表現から、3 次と 5 次の既約表現が得られることが分かりました。

```
gap> H:=Normalizer(G,Group([(1,2,3,4,5)]));
Group([ (1,2,3,4,5), (2,5)(3,4) ])
gap> ctH:=CharacterTable(H);;
gap> Display(ctH);
CT6

     2  1  1  .  .
     5  1  .  1  1

       1a 2a 5a 5b
    2P 1a 1a 5b 5a
    3P 1a 2a 5b 5a
    5P 1a 2a 1a 1a

X.1     1  1  1  1
X.2     1 -1  1  1
X.3     2  .  A *A
X.4     2  . *A  A

A = E(5)^2+E(5)^3
  = (-1-ER(5))/2 = -1-b5
ms:=MatScalarProducts(Irr(ctG),InducedClassFunctions(Irr(ctH),G));;
gap> Display(ms);
[ [  1,  0,  0,  0,  1 ],
  [  0,  1,  1,  0,  0 ],
  [  0,  1,  0,  1,  1 ],
  [  0,  0,  1,  1,  1 ] ]
```

2 つの 1 次表現 r1, r2 を次のように構成し、G の 6 次誘導表現 R1, R2 を作ります。

```
gap> ma:=[[1]];; mb:=[[-1]];;
gap> r1:=GroupHomomorphismByImages(H,Group([[[-1]]]),[a,b],[ma,ma]);
[ (1,2,3,4,5), (2,5)(3,4) ] -> [ [ [ 1 ] ], [ [ 1 ] ] ]
gap> r2:=GroupHomomorphismByImages(H,Group([[[-1]]]),[a,b],[ma,mb]);
[ (1,2,3,4,5), (2,5)(3,4) ] -> [ [ [ 1 ] ], [ [ -1 ] ] ]
gap> R1:=InducedRepresentation(r1,G);;
gap> R2:=InducedRepresentation(r2,G);;
```

これまでと同じような正則行列 P を使って 6 次表現 R1 から 5 次既約表現 S5 を抽出します。また、行列 R2x[1] の固有ベクトルから 3 次元ベクトル空間を 2 つ作り、この空間の基底となるベクトルから正則行列 P を作って 6 次表現 R2 を 2 つの 3 次既約表現に分解するします。

```
gap> R1x:=List([a,b,c],x->Image(R1,x));;
gap> R2x:=List([a,b,c],x->Image(R2,x));;
gap> P:=[[ 1, 1, 1, 1, 1, 1],
>        [ 1,-1, 0, 0, 0, 0],
>        [ 1, 0,-1, 0, 0, 0],
>        [ 1, 0, 0,-1, 0, 0],
>        [ 1, 0, 0, 0,-1, 0],
>        [ 1, 0, 0, 0, 0,-1]];;
gap> R15:=List(R1x,x->P*x*P^-1);;
gap> S1:=List(R15,x->x{[1]}{[1]});;
gap> S5:=List(R15,x->x{[2..6]}{[2..6]});;
gap> F:=CyclotomicField(5);;
gap> ev:=Eigenvectors(F,R2x[1]);;
gap> Display(ev);
[ [        1,        0,        0,        0,        0,        0 ],
  [        0,        1,        1,       -1,        1,        1 ],
  [        0,        1,   E(5)^3,  -E(5)^2,     E(5),   E(5)^4 ],
  [        0,        1,     E(5),  -E(5)^4,   E(5)^2,   E(5)^3 ],
  [        0,        1,   E(5)^4,    -E(5),   E(5)^3,   E(5)^2 ],
  [        0,        1,   E(5)^2,  -E(5)^3,   E(5)^4,     E(5) ] ]
gap> wa:=ev[3]+ev[6];;
gap> wb:=ev[4]+ev[5];;
gap> Dimension(VectorSpace(F,[wa,wa*R2x[1],wa*R2x[3]]));
3
gap> Dimension(VectorSpace(F,[wb,wb*R2x[1],wb*R2x[3]]));
3
gap> P:=[wa,wa*R2x[1],wa*R2x[3],wb,wb*R2x[1],wb*R2x[3]];;
gap> R33:=List(R2x,x->P*x*P^-1);;
gap> S3a:=List(R33,x->x{[1..3]}{[1..3]});;
gap> S3b:=List(R33,x->x{[4..6]}{[4..6]});;
```

最後に、得られた既約表現を IRR にまとめて、それぞれの指標を計算して、その値が指標表の値と一致することを確認します。

```
gap> IRR:=[S1,S3a,S3b,S4,S5];;
gap> Rs:=List(IRR,R->GroupHomomorphismByImages(G,Group(R),[a,b,c],R));;
gap> List(Rs,R->List(cc,x->Trace(Image(R,x))));
[ [ 1,  1,  1,             1,             1 ],
  [ 3, -1,  0, -E(5)^2-E(5)^3,   -E(5)-E(5)^4 ],
  [ 3, -1,  0,   -E(5)-E(5)^4, -E(5)^2-E(5)^3 ],
  [ 4,  0,  1,            -1,            -1 ],
  [ 5,  1, -1,             0,             0 ] ]
```

参考文献

[1] G. Butler, *Fundamental Algorithms for Permutation Groups*, Lecture Notes in Computer Science 559, Springer-Verlag.

[2] Peter J. Cameron, *Permutation Groups*, London Mathematical Society Student Texts 45.

[3] Derek F. Holt, Bettina Eick, Eamonn A. O'Brien, *Handbook of Computational Group Theory*, Descreate Mathematics and its Applications (Boca Raton, FL:Chapman & Hall/CRC,2005).

[4] Klaus Lux, Herbert Pahlings, *Representations of Groups*, Cambridge studies in advanced mathematics I24.