1 Aritmética entera usando GAP

1.1 Cociente y resto

Dados dos enteros a y b, el resto de dividir a entre b se puede calcular usando el comando `mod`.

```
gap> -3 mod 5;
2
```

Y el cociente se puede calcular de la siguiente forma.

```
gap> (-3-(-3 mod 5))/5;
-1
```

Si GAPescribimos $-3/5$, el resultado es un racional, y si usamos el comando `Int`, obtenemos la parte entera de dividir de ese racional, que no es precisamente lo que buscamos.

```
gap> Int((-3)/5);
0
```

1.2 Máximo común divisor y mínimo común múltiplo. Coeficientes de Bézout

El máximo común divisor de dos enteros (o más) puede ser calculado con el comando `Gcd`, y su mínimo común múltiplo con el comando `Lcm`.

```
gap> Gcd(3,-5);
1
gap> Lcm(3,-5);
15
```

Si queremos conseguir los coeficientes de Bézout, podemos usar el comando `GcdRepresentation` (entre las muchas posibilidades que da GAPpara esto).

```
gap> GcdRepresentation(3,-5);
[ 2, 1 ]
gap> GcdRepresentation(10,15,18);
[ 7, -7, 2 ]
```

1.3 Ecuaciones diofánticas

Como ya sabemos, una vez resuelto el problema de encontrar los coeficientes de Bézout de un máximo común divisor de dos enteros, tenemos también solución para resolver una ecuación diofántica. Lo único que tenemos que comprobar es si el término independiente es divisible por el máximo común divisor de los coeficientes, y luego multiplicar los coeficientes de Bézout por el factor apropiado para conseguir una solución particular.

Si queremos resolver $10x + 25y = 45$, hacemos lo siguiente.

```
gap> Gcd(10,25);
5
#vemos si 45 es divisible por 5
gap> 45 mod 5;
0
#calculamos 45 entre 5
gap> 45/last2;
9
#multiplicamos el resultado por los coeficientes de Bézout
gap> last*GcdRepresentation(10,25);
[ -18, 9 ]
#comprobamos el resultado
gap> last*[10,25];
45
```

1.4 Primos

Para factorizar un entero en producto de primos podemos usar el comando `Factors`.

```
gap> Factors(100);
[ 2, 2, 5, 5 ]
```

También podemos saber si un número es primo usando el comando `IsPrime`.

```
gap> IsPrime(10);
false
gap> IsPrime(7);
true
```

`Primes` es una lista que contiene los primos menores que mil.

```
#el tercer primo
gap> Primes[3];
5
#los primeros cuarenta primos
gap> Primes{[1..40]};
[ 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53,
  59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113,
  127, 131, 137, 139, 149, 151, 157, 163, 167, 173 ]
```

También podemos hacer un filtro de una lista para ver qué elementos en ella son primos.

```
gap> Filtered([1..300],IsPrime);
[ 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53,
  59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113,
  127, 131, 137, 139, 149, 151, 157, 163, 167, 173, 179, 181,
  191, 193, 197, 199, 211, 223, 227, 229, 233, 239, 241, 251,
  257, 263, 269, 271, 277, 281, 283, 293 ]
```

1.5 Congruencias

Gracias a los coeficientes de Bézout también podemos resolver cualquier congruencia del tipo $ax \equiv b \mod m$, con a, b y m enteros ($m \neq 0$).

Sea la congruencia $60x \equiv 90 \mod 105$.

```
gap> Gcd(60,105);
15
#dividimos todo por 15
gap> [60,90,105]/15;
[ 4, 6, 7 ]
#buscamos el inverso de 4 módulo 7
#(me quedo con el coeficiente de Bézout de 4)
gap> GcdRepresentation(4,7)[1];
2
```

```
#la solución particular es por tanto
gap> 2*6 mod 7;
5
```

Si lo que queremos es resolver un sistema de congruencias de la forma

$$\begin{aligned} x &\equiv a_1 \mod m_1, \\ \ldots x &\equiv a_n \mod m_n, \end{aligned}$$

entonces podemos usar el comando `ChineseRem`, cuyo primer argumento es la lista de módulos y el segundo una lista (con la misma longitud) de residuos.

Así, para resolver $x \equiv 3 \mod 14$, $x \equiv 7 \mod 16$ hacemos lo siguiente.

```
gap> ChineseRem([14,16],[3,7]);
87
```

1.6 Los anillos $\mathbb{Z}_n$

La función `ZmodnZ` nos permite definir en GAPel anillo de enteros módulo el argumento entero que le pasemos.

```
gap> A:=ZmodnZ(10);
(Integers mod 10)
```

El uno de un anillo (el elemento neutro del producto) se calcula con el comando `One`.

```
gap> uno:=One(A);
ZmodnZObj( 1, 10 )
```

Podemos hacer operaciones elementales en ese anillo, como calcular inversos, sumar elementos.

```
gap> 1/(3*uno);
ZmodnZObj( 7, 10 )
gap> Int(last);
7
gap> 2*uno+6/3*uno;
ZmodnZObj( 4, 10 )
```

Si un elemento no tiene inverso, devuelve `fail`.

```
gap> 1/(2*uno);
ZmodnZObj( fail, 10 )
gap> Int(last);
fail
```

También podemos usar la función `Inverse`.

```
gap> Inverse(2*uno);
fail
gap> Inverse(3*uno);
ZmodnZObj( 7, 10 )
```

Y extraer así el conjunto de unidades de $\mathbb{Z}_{10}$.

```
gap> Filtered([1..9],n->Inverse(n*uno)<>fail);
[ 1, 3, 7, 9 ]
```

Aunque esto lo podíamos haber hecho usando la función `PrimeResidues`.

```
gap> PrimeResidues(10);
[ 1, 3, 7, 9 ]
```

La función φ de Euler (función tociente) se expresa como `Phi` en GAP.

```
gap> Phi(10);
4
```

Para n positivo, en GAPexiste el comando `Z(n)` que nos proporciona el menor elemento primitivo de $\mathbb{Z}_n$ (el menor residuo módulo n que genera al grupo multiplicativo de $\mathbb{Z}_n$).

```
gap> uno:=One(Z(7));
Z(7)^0
gap> 2*uno+3*uno;
Z(7)^5
```

```
gap> Int(last);
5
gap> 2/3*uno;
Z(7)
gap> Int(last);
3
gap> 1/3*uno;
Z(7)^5
gap> last*2;
Z(7)
```

2 Anillos y extensiones cuadráticas usando GAP

2.1 Divisores de cero

Podemos utilizar el mismo procedimiento usado anteriormente para calcular las unidades en $\mathbb{Z}_{10}$, para determinar los divisores de cero de $\mathbb{Z}_{10}$.

```
gap> Filtered([1..9],n->ForAny([1..9],m->n*m mod 10=0));
[ 2, 4, 5, 6, 8 ]
```

Por tanto, como ya sabemos $\mathbb{Z}_{10}$, no es un dominio de integridad. En GAPpodemos usar la siguiente orden para comprobarlo directamente sin calcular sus divisores de cero.

```
gap> IsIntegralRing(ZmodnZ(10));
false
```

Veamos ahora cómo podemos calcular los divisores de cero del anillo

$$\mathbb{Z}_2[i] = \{a + bi | a, b \in \mathbb{Z}_2\}.$$

Primero calculamos los elementos de $\mathbb{Z}_2[i]$. Como i es la raíz cuarta de la unidad, usamos `E(4)` para representarlo.

```
gap> l:=Cartesian([0..1],[0..1]);
[ [ 0, 0 ], [ 0, 1 ], [ 1, 0 ], [ 1, 1 ] ]
gap> z2i:=Set(l,n->n[1]+n[2]*E(4));
[ 0, 1, E(4), 1+E(4) ]
```

Nos quedamos con los elementos no nulos.

```
gap> last{[2..4]};
[ 1, E(4), 1+E(4) ]
```

Seleccionamos (`Filtered`) ahora aquellos para los que exista (`ForAny`) un elemento no nulo que multiplicado por él de cero.

```
gap> Filtered(last,n->ForAny(last,m->EuclideanRemainder(n*m,2)=0));
[ 1+E(4) ]
```

Lo que indica que $1+i$ es el único divisor de cero no nulo de $\mathbb{Z}_2[i]$. Nótese que para hacer las cuentas módulo 2, hemos usado el comando `EuclideanRemainder`, ya que con los enteros de Gauss no podemos utilizar `mod`.

2.2 Unidades

GAPtiene un comando para determinar el grupo de unidades de un anillo. Usémoslo para ver las unidades de $\mathbb{Z}_{10}$.

```
gap> Units(ZmodnZ(10));
<group with 1 generators>
```

Como la salida es un grupo, para ver sus elementos lo pasamos a lista y luego cada elemento lo representamos como un entero.

```
gap> List(last,Int);
[ 1, 3, 7, 9 ]
```

También podemos optar por la fuerza bruta, y ver para qué enteros entre 1 y 9, existe otro de forma que su producto de 1 módulo 10.

```
gap> Filtered([1..9],n->ForAny([1..9],m->n*m mod 10=1));
[ 1, 3, 7, 9 ]
```

Además GAPtiene un comando para determinar si un anillo es o no un cuerpo.

```
gap> IsField(ZmodnZ(5));
true
```

2.3 Enteros de Gauss

Ya hemos visto en prácticas anteriores cómo factorizar enteros de Gauss. También vimos cómo calcular el cociente y resto de dos enteros cualesquiera, así como su máximo común divisor y los coeficientes de Bézout correspondientes. Por desgracia, como hemos visto anteriormente, la función `mod` no se puede utilizar con los enteros de Gauss. Podemos usar en su lugar, `EuclideanRemainder` y `EuclideanQuotient` para el cociente, o bien, `QuotientRemainder` si queremos obtener ambas cantidades a la vez.

```
gap> (9+7*E(4)) mod (1+E(4));
Error, no method found! For debugging hints type ?Recovery from NoMethodFound
Error, no 1st choice method found for `MOD' on 2 arguments 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

gap> (9+7*E(4))/(1+E(4));
8-E(4)

gap> QuotientRemainder(9+7*E(4),1+E(4));
[ 8-E(4), 0 ]

gap> QuotientRemainder(9+7*E(4),3+E(4));
[ 3+E(4), 1+E(4) ]
gap> (9+7*E(4))/(3+E(4));
17/5+6/5*E(4)
```

Para el máximo común divisor y coeficientes de Bézout, podemos usar las funciones que conocemos para enteros.

```
gap> Gcd(2*E(4),3-7*E(4));
1+E(4)
gap> GcdRepresentation(2*E(4),3-7*E(4));
[ 2-4*E(4), -E(4) ]
```

Para encontrar los enteros de Gauss de norma menor o igual que cinco que sean irreducibles, podemos usar la función `Norm`. Primero generamos los posibles candidatos, que tienen que tener parte real e imaginaria menor o igual que 2 en valor absoluto.

```
gap> elementos:=List(Cartesian([-2..2],[-2..2]),n->n[1]+E(4)*n[2]);
[ -2-2*E(4), -2-E(4), -2, -2+E(4), -2+2*E(4), -1-2*E(4), -1-E(4), -1,
  -1+E(4), -1+2*E(4), -2*E(4), -E(4), 0, E(4), 2*E(4), 1-2*E(4), 1-E(4), 1,
  1+E(4), 1+2*E(4), 2-2*E(4), 2-E(4), 2, 2+E(4), 2+2*E(4) ]
```

Seleccionamos ahora aquellos con norma menor o igual que cinco.

```
gap> Filtered(elementos,n->(Norm(n)<=5));
[ -2-E(4), -2, -2+E(4), -1-2*E(4), -1-E(4), -1, -1+E(4), -1+2*E(4), -2*E(4),
  -E(4), 0, E(4), 2*E(4), 1-2*E(4), 1-E(4), 1, 1+E(4), 1+2*E(4), 2-E(4), 2,
  2+E(4) ]
```

Si escribimos,

```
gap> Filtered(last,IsPrime);
[ -2-E(4), -2, -2+E(4), -1-2*E(4), -1-E(4), -1+E(4), -1+2*E(4), 1-2*E(4),
  1-E(4), 1+E(4), 1+2*E(4), 2-E(4), 2, 2+E(4) ]
```

la salida no es la correcta, ya que por ejemplo nos aparecen 2 y -2, que sabemos que no son irreducibles en $\mathbb{Z}[i]$. Esto se debe a que no hemos especificado el anillo en la orden `IsPrime`.

```
gap> Filtered(last,n->IsPrime(GaussianIntegers,n));
[ -2-E(4), -2+E(4), -1-2*E(4), -1-E(4), -1+E(4), -1+2*E(4), 1-2*E(4), 1-E(4),
  1+E(4), 1+2*E(4), 2-E(4), 2+E(4) ]
```

Si queremos saber cuántos tenemos salvo asociados, usamos la función `StandardAssociate` (que da un asociado estándar a cada elemento de $\mathbb{Z}[i]$) junto con la operación `Set` para eliminar repetidos.

```
gap> Set(last,StandardAssociate);
[ 1+E(4), 1+2*E(4), 2+E(4) ]
```

Obsérvese que la salida es la misma si hacemos lo siguiente (+por qué?).

```
gap> elementos:=List(Cartesian([0..2],[0..2]),n->n[1]+E(4)*n[2]);
[ 0, E(4), 2*E(4), 1, 1+E(4), 1+2*E(4), 2, 2+E(4), 2+2*E(4) ]
gap> Filtered(elementos,n->(Norm(n)<=5));
[ 0, E(4), 2*E(4), 1, 1+E(4), 1+2*E(4), 2, 2+E(4) ]
gap> Filtered(last,n->IsPrime(GaussianIntegers,n));
[ 1+E(4), 1+2*E(4), 2+E(4) ]
```

2.4 Operaciones en $\mathbb{Z}[\sqrt{(d)}]$, $d \in \{-1,2,-2,3\}$

Si introducimos la expresión

```
gap> (1+2*Sqrt(3))*(Sqrt(3));
-6*E(12)^4-E(12)^7-6*E(12)^8+E(12)^11
```

obtenemos una salida un poco difícil de tratar. Es por eso que vamos a definir nuestros propios productos, cociente y resto. Vamos representar un entero $a+b\sqrt{d}$ en $\mathbb{Z}[d]$ (con d libre de cuadrados) mediante una lista $[a,b]$, y pasaremos d como argumento extra en nuestras funciones. Así la función producto podría definirse como sigue.

```
por:=function(x,y,d)
    return [x[1]*y[1]+d*x[2]*y[2],x[1]*y[2]+x[2]*y[1]];
end;

gap> por([1,2],[0,1],3);
[ 6, 1 ]

gap> (1+2*Sqrt(3))*(Sqrt(3));
-6*E(12)^4-E(12)^7-6*E(12)^8+E(12)^11
gap> 6+Sqrt(3);
-6*E(12)^4-E(12)^7-6*E(12)^8+E(12)^11
```

Para hacer el cociente, necesitamos la norma. Vamos a definir una función para tal efecto, aunque como explicamos después, también se puede hacer definiendo el cuerpo $\mathbb{Q}(\sqrt{(d)})$.

```
norma:=function(x,d)
    return AbsInt(x[1]^2-d*x[2]^2);
end;

gap> norma([4,1],3);
13
gap> F:=Field(Sqrt(3));
NF(12,[ 1, 11 ])
gap> Norm(F,4+Sqrt(3));
13
```

Hay que tener cuidado con especificar en qué cuerpo estamos si usamos `Norm` para no obtener resultados no deseados.

```
gap> Norm(4+Sqrt(3));
169
```

Como hemos visto en teoría, para dividir, necesitamos aproximarnos a un racional lo mejor que podamos con un entero. Para ello introducimos una función de redondeo.

```
redondeo:=function(x)
    if ((x-Int(x))<(Int(x)+1-x)) then
        return Int(x);
    fi;
    return Int(x)+1;
end;

gap> redondeo(2/3);
1
gap> redondeo(1/3);
0
```

Usando la función auxiliar

```
conjugado:=function(x)
    return [x[1],-x[2]];
end;
```

podemos definir la función cociente de la siguiente forma.

```
cociente:=function ( x, y, d )
    return List( por( x, conjugado( y ), d ) / norma( y, d ), redondeo );
end;

gap> cociente([11,7],[1,1],-1);
[ 9, -2 ]
gap> (11+7*E(4))/(1+E(4));
9-2*E(4)
```

Por tanto, una función resto ya es bastante fácil de obtener.

```
resto:=function(x,y,d)
    return x-por(y,cociente(x,y,d),d);
end;

gap> resto([11,7],[1,1],-1);
[ 0, 0 ]
gap> EuclideanRemainder(11+7*E(4),1+E(4));
0
```

3 Polinomios usando GAP

3.1 Coeficientes

Como ya hemos visto con anterioridad, para empezar a trabajar con polinomios, tenemos que especificar las variables y qué anillo de coeficientes vamos a considerar. GAPpor defecto expande las expresiones que introducimos, a diferencia de MATHEMATICA.

```
gap> x:=Indeterminate(Rationals,"x");
x
gap> (x+1)*(x-1);
x^2-1
```

Si queremos obtener una lista de los coeficientes de un polinomio en una variable, podemos usar lo siguiente.

```
gap> CoefficientsOfUnivariatePolynomial(x^2+x-1);
[ -1, 1, 1 ]
```

Y el polinomio líder lo obtenemos con `LeadingCoefficient`.

```
gap> LeadingCoefficient(x^2+x-1);
1
```

Definamos una función para encontrar el término líder de un polinomio respecto de una variable. En ella usamos funciones que son alternativa a las que acabamos de ver para más de una variable.

```
terminolider:=function(p,x)
    local grado;
    grado:=DegreeIndeterminate(p,x);
    return PolynomialCoefficientsOfPolynomial(p,x)[grado+1]*x^grado;
end;

gap> terminolider(x^2+x-1,x);
x^2
gap> terminolider(3*x^2+x-1,x);
3*x^2
gap> y:=Indeterminate(Rationals,"y");
y
gap> terminolider(y*x^2+y^4*x-1,x);
x^2*y
```

3.2 División

Si el anillo de coeficientes que consideramos es un cuerpo, entonces sabemos que el anillo de polinomios sobre una sola variable es un dominio euclídeo. Por tanto, podemos usar las funciones que ya conocemos para calcular el cociente y resto de una división.

```
gap> x:=Indeterminate(Rationals,"x");
x
gap> QuotientRemainder(x^3-x+1,2*x^2-3);
[ 1/2*x, 1/2*x+1 ]
```

Si nuestro anillo de polinomios no es un dominio euclídeo, entonces no podemos usar estas funciones.

```
gap> y:=Indeterminate(Rationals,"y");
y

gap> QuotientRemainder((x^3-x+1)*(y-1),y-1);
Error, no method found! For debugging hints type ?Recovery from NoMethodFound
Error, no 2nd choice method found for ‘QuotientRemainder’ on 3 arguments calle\
d from
QuotientRemainder( DefaultRing( [ r, m ] ), r, m ) 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>
```

Ahora bien, si que podemos usar la función `Quotient` que nos da el cociente, en caso de que éste pertenezca a nuestro anillo de polinomio, y `fail` en caso contrario.

```
gap> Quotient((x^3-x+1)*(y-1),y-1);
x^3-x+1

gap> Quotient(2,3);
fail
```

(Esta última instrucción viene a decir que el cociente de dos entre tres no es entero, pues considera los argumentos de la función como enteros.)

3.3 Factorización

Si lo que queremos es factorizar polinomios, primero tenemos que definir la variable, e indicar cuál es el anillo de coeficientes para nuestros polinomios. Luego se usa `Factors` igual que antes.

```
gap> x:=Indeterminate(ZmodnZ(5),"x");
x
gap> Factors(x^2+1);
[ x+Z(5), x+Z(5)^3 ]
gap> Int(Z(5));
2
gap> Int(Z(5)^3);
3
```

Si cambiamos el anillo base, el resultado puede verse alterado.

```
gap> x:=Indeterminate(Rationals,"x");
x
gap> Factors(x^2+1);
[ x^2+1 ]
```

```
gap> x:=Indeterminate(Rationals,"x");
x
gap> Factors(x^3-1);
[ x-1, x^2+x+1 ]
gap> x:=Indeterminate(ZmodnZ(3),"x");
x
gap> Factors(x^3-1);
[ x-Z(3)^0, x-Z(3)^0, x-Z(3)^0 ]
```

Lo mismo ocurre con las raices y con el hecho de ser irreducible.

```
gap> x:=Indeterminate(ZmodnZ(3),"x");
x
gap> RootsOfUPol(x^3-1);
[ Z(3)^0, Z(3)^0, Z(3)^0 ]
gap> x:=Indeterminate(Rationals,"x");
x
gap> RootsOfUPol(x^3-1);
[ 1 ]

gap> x:=Indeterminate(ZmodnZ(3),"x");
x
gap> IsIrreducible(x^2+1);
true

gap> x:=Indeterminate(ZmodnZ(2),"x");
x
gap> IsIrreducible(x^2+1);
false
```

Veamos ahora a modo de ejemplo cómo calcular todos los polinomios irreducibles hasta un determinado grado en $\mathbb{Z}_m$. Empezamos definiendo una función que nos genere todos los polinomios hasta un determinado grado.

```
polshastagradomodm:=function(n,x,m)
    local ps;
```

```
    if (n=0) then
        return [0..(m-1)];
    fi;

    ps:=polshastagradomodm(n-1,x,m);
    return List(Cartesian(ps,List([0..(m-1)],i->i*x^n)),Sum);
end;
```

Así todos los polinomios en $\mathbb{Z}_3$ de grado menor o igual que dos son:

```
gap> polshastagradomodm(2,x,3);
[ 0*Z(3), x^2, -x^2, x, x^2+x, -x^2+x, -x, x^2-x, -x^2-x, Z(3)^0, x^2+Z(3)^0,
  -x^2+Z(3)^0, x+Z(3)^0, x^2+x+Z(3)^0, -x^2+x+Z(3)^0, -x+Z(3)^0,
  x^2-x+Z(3)^0, -x^2-x+Z(3)^0, -Z(3)^0, x^2-Z(3)^0, -x^2-Z(3)^0, x-Z(3)^0,
  x^2+x-Z(3)^0, -x^2+x-Z(3)^0, -x-Z(3)^0, x^2-x-Z(3)^0, -x^2-x-Z(3)^0 ]
```

De entre ellos podemos escoger los que son irreducibles.

```
gap> Filtered(last,IsIrreducible);
[ x, -x, x^2+Z(3)^0, x+Z(3)^0, -x^2+x+Z(3)^0, -x+Z(3)^0, -x^2-x+Z(3)^0,
  -x^2-Z(3)^0, x-Z(3)^0, x^2+x-Z(3)^0, -x-Z(3)^0, x^2-x-Z(3)^0 ]
```

Y si queremos quedarnos con un representante salvo asociados, podemos usar lo siguiente.

```
gap> Set(last,StandardAssociate);
[ x, x+Z(3)^0, x-Z(3)^0, x^2+Z(3)^0, x^2+x-Z(3)^0, x^2-x-Z(3)^0 ]
```

Para finalizar esta sección, implementamos una función que da los primos que se pueden aplicar en el criterio de Eisenstein para un polinomio en una variable.

```
eisenstein:=function(p)
    local lc,fp;
    lc:=CoefficientsOfUnivariatePolynomial(p);
    lc:=lc{[1..(Length(lc)-1)]};
    fp:=Factors(lc[1]);
    return Filtered(fp,f->(ForAll(lc,c->(c mod f=0)) and (lc[1] mod f^2=0)));
end;
```

```
gap> x:=Indeterminate(Rationals,"x");
x
gap> eisenstein(x^2+2*x-6);
[  ]
gap> eisenstein(x^2+2*x-4);
[ -2, 2 ]
```

3.4 Polinomios simétricos

Seguimos en esta sección la demostración dada en teoría para encontrar la expresión de un polinomio simétrico en función de los polinomios simétricos elementales.

Empezamos construyendo de forma recursiva el conjunto de polinomios simétricos elementales en un número determinado de variables (el argumento x contiene la lista de variables).

```
simetricoselementales:=function(x)
    local el;
    if (Length(x)=1) then
        return x;
    fi;
    el:=Concatenation([1],simetricoselementales(x{[2..Length(x)]}),[0]);
    return List([2..Length(el)],i->x[1]*el[i-1]+el[i]);
end;

gap> x:=Indeterminate(Rationals,"x");
x
gap> y:=Indeterminate(Rationals,"y");
y
gap> z:=Indeterminate(Rationals,"z");
z

gap> simetricoselementales([x,y,z]);
[ x+y+z, x*y+x*z+y*z, x*y*z ]
```

Vamos a identificar los polinomios simétricos elementales con las variables de entrada. Así si tenemos dos variables x e y, éstas vistas

como polinomios simétricos elementales denotan también respectivamente a $x+y$ y xy. Para traducir esta representación a notación estándar, usamos la siguiente función.

```
evaluasim:=function(f,x)
    if (IsRat(f)) then
        return f;
    fi;
    return Value(f,x,simetricoselementales(x));
end;
```

(La función `Value` sirve para evaluar un polinomio en varias variables. Si la entrada es un racional, no sabe hacer dicha evaluación. Es por eso que hemos puesto ese condicional al principio de la función.)

Ya tenemos pues los ingredientes necesarios para implementar el algoritmo.

```
sim:=function(f,x)
    local f0,f1,f2,g1,g2;

    if (Length(x)=1) or (IsRat(f)) then
        return f;
    fi;

    f0:=Value(f,[x[Length(x)]],[0]);
    if f0=0 then
        return 0;
    fi;
    g1:=sim(f0,x{[1..(Length(x)-1)]});
    f1:=f-evaluasim(g1,x);
    if f1=0 then
        return g1;
    fi;
    f2:=Quotient(f1,Product(x));
    g2:=sim(f2,x);
    return g1+x[Length(x)]*g2;

end;
```

```
gap> sim((x+y)*(y+z)*(z+x),[x,y,z]);
x*y-z
gap> evaluasim(last,[x,y,z]);
x^2*y+x^2*z+x*y^2+2*x*y*z+x*z^2+y^2*z+y*z^2
gap> (x+y)*(y+z)*(z+x);
x^2*y+x^2*z+x*y^2+2*x*y*z+x*z^2+y^2*z+y*z^2
```

3.5 Resultante y discriminante

Para calcular la resultante y el discriminante podemos usar las funciones `Resultant` y `Discriminant`, respectivamente.

```
gap> x:=Indeterminate(Rationals,"x");
x
gap> y:=Indeterminate(Rationals,"y");
y
gap> Resultant(x^2+y^2-1,x-y,y);
2*x^2-1

gap> Discriminant(x^3+1);
-27
gap> z:=Indeterminate(Rationals,"z");
z
gap> Discriminant(x^3+y*x^2+z,x);
-4*z*y^3-27*z^2
```

3.6 Cociente por un ideal

Intentemos calcular los divisores de cero y unidades del anillo cociente $R = \mathbb{Z}_2[x]/(x^2+1)$. Empezamos definiendo nuestra variable y el módulo.

```
gap> x:=Indeterminate(ZmodnZ(2),"x");
x

gap> modulo:=x^2+1;
```

```
x^2+Z(2)^0
```

Como cada elemento en R tiene un único representante de grado menor o igual que uno (el resto de dividir por x^2+1), podemos identificar R con el siguiente conjunto.

```
gap> elementos:=List(Cartesian([0..1],[0..1]),n->n[1]+x*n[2]);
[ 0*Z(2), x, Z(2)^0, x+Z(2)^0 ]
```

Que se lee como $\{0, x, 1, 1+x\}$. Seleccionamos aquellos elementos que son no nulos.

```
gap> elementosnonulos:=elementos{[2..4]};
[ x, Z(2)^0, x+Z(2)^0 ]
```

Así las unidades se pueden calcular de la siguiente forma.

```
gap> Filtered(elementosnonulos,n->
    ForAny(elementosnonulos,m->IsOne(EuclideanRemainder(n * m,modulo))));
[ x, Z(2)^0 ]
```

Obsérvese que hemos vuelto a utilizar `EuclideanRemainder`. La función `IsOne` sirve para determinar si un elemento en $\mathbb{Z}_2[x]$ es uno (no podemos en este caso escribir simplemente `EuclideanRemainder(n * m,modulo)=1`).

Los divisores de cero no nulos, se calculan de forma análoga.

```
gap> Filtered(elementosnonulos,
n->ForAny(elementosnonulos,
m->IsZero(EuclideanRemainder(n * m,modulo))))
[ x+Z(2)^0 ]
```

4 Grupos abelianos usando GAP

4.1 Grupos abelianos finitamente generados

Para crear un grupo libre en GAPcon un número determinado de generadores, usamos el comando `FreeGroup`. Así, `FreeGroup(n)` crea un grupo con `n` generadores, que son `f.1`, ..., `f.n` y que GAPpinta en pantalla como `f1`, ..., `fn`. Si queremos que en vez de `fi`, GAPimprima con otros nombres los generadores, podemos pasar como argumentos dichos nombres.

```
gap> f:=FreeGroup("a","b");
<free group on the generators [ a, b ]>
gap> a:=f.1;;
gap> b:=f.2;;
gap> IsAbelian(f);
false
```

Esto hace que `f` sea un grupo libre sobre {`a`,`b`} no conmutativo por tener al menos dos generadores. A partir de ahora podemos usar `a` como primer generador de `f` y `b` como segundo generador.

Si queremos definir un grupo abeliano generado por dos elementos, como por ejemplo $\mathbb{Z}_6 \times \mathbb{Z}_{12}$, podemos usar una de sus presentaciones

$$\{a,\ b \mid 6a = 0, 12b = 0\}$$

y hacer un cociente de `f` "modulo" los relatores. Como por defecto GAPtrabaja con grupos no abelianos, debemos usar notación multiplicativa. Además, tenemos que explicitar que `a` y `b` conmutan.

Con el comando `Elements` podemos imprimir los elementos del nuevo grupo creado (no se debe usar este comando con `f`, que es infinito).

```
gap> g:=f/[a*b*(b*a)^(-1),a^6,b^12];
<fp group on the generators [ a, b ]>
gap> u:=g.1;;
gap> v:=g.2;;
gap> Elements(g);
[ <identity ...>, a^3, b^9, a^2, b^4, b^6, a^3*b^9, a^5, a^3*b^4, a^3*b^6,
  a^2*b^9, b, b^3, a^4, a^2*b^4, a^2*b^6, b^8, b^10, a^5*b^9, a^3*b, a^3*b^3,
  a, a^5*b^4, a^5*b^6, a^3*b^8, a^3*b^10, a^4*b^9, a^2*b, a^2*b^3, b^5, b^7,
  a^4*b^4, a^4*b^6, a^2*b^8, a^2*b^10, b^2, a*b^9, a^5*b, a^5*b^3, a^3*b^5,
  a^3*b^7, a*b^4, a*b^6, a^5*b^8, a^5*b^10, a^3*b^2, a^4*b, a^4*b^3, a^2*b^5,
  a^2*b^7, b^11, a^4*b^8, a^4*b^10, a^2*b^2, a*b, a*b^3, a^5*b^5, a^5*b^7,
  a^3*b^11, a*b^8, a*b^10, a^5*b^2, a^4*b^5, a^4*b^7, a^2*b^11, a^4*b^2,
  a*b^5, a*b^7, a^5*b^11, a*b^2, a^4*b^11, a*b^11 ]
gap> a*b=b*a;
false
gap> u*v=v*u;
true
```

```
gap> a in g;
false
gap> u in g;
true
```

Si queremos ver los generadores de un grupo, podemos usar el comando `GeneratorsOfGroup`, para ver los relatores, usamos `RelatorsOfFpGroup`. Aunque para eso el argumento tiene que ser un grupo del cual GAPsepa que es finitamente presentado. Lo cual queda claro para `g` por haber sido definido mediante una presentación finita.

```
gap> GeneratorsOfGroup(g);
[ a, b ]
gap> RelatorsOfFpGroup(g);
[ a*b*a^-1*b^-1, a^6, b^12 ]
```

Una forma alternativa y fácil de crear grupos finitos es a través del comando `CyclicGroup`, que crea un grupo cíclico del orden del argumento que le pasemos, combinarlo con el comando `DirectProduct` que a partir de dos o más grupos crea su producto directo. El problema es que gap no entiende la salida de `CyclicGroup` como un grupo finitamente presentado. Para arreglar ese pequeño problema, se puede usar el siguiente truco.

```
gap> g:=CyclicGroup(12);
<pc group of size 12 with 3 generators>
gap> IsCyclic(g);
true
gap> IsomorphismFpGroup(g);
[ f1, f2, f3 ] -> [ F1, F2, F3 ]
gap> Image(IsomorphismFpGroup(g));
<fp group of size 12 on the generators [ F1, F2, F3 ]>
gap> IsomorphismFpGroup(g);
[ f1, f2, f3 ] -> [ F1, F2, F3 ]
gap> gfp:=Image(last);
<fp group of size 12 on the generators [ F1, F2, F3 ]>
gap> RelatorsOfFpGroup(gfp);
[ F1^2*F2^-1, F2^-1*F1^-1*F2*F1, F3^-1*F1^-1*F3*F1, F2^2*F3^-1,
  F3^-1*F2^-1*F3*F2, F3^3 ]
gap> gs:=SimplifiedFpGroup(gfp);
```

```
<fp group on the generators [ F1 ]>
gap> RelatorsOfFpGroup(gs);
[ F1^12 ]
```

Como se ve, esta última salida es más "natural" que la inicial.

```
gap> d:=DirectProduct(CyclicGroup(4),CyclicGroup(15));
<pc group of size 60 with 4 generators>
gap> dfp:=SimplifiedFpGroup(Image(IsomorphismFpGroup(d)));
<fp group on the generators [ F1, F3 ]>
gap> RelatorsOfFpGroup(dfp);
[ F1^4, F3^-1*F1^-1*F3*F1, F3^15 ]
```

Un subgrupo finitamente generado de un grupo finitamente generado se puede definir usando el comando `Subgroup`.

```
gap> f:=FreeGroup("a","b");
<free group on the generators [ a, b ]>
gap> a:=f.1;
a
gap> b:=f.2;
b
gap> g:=f/[a*b*(b*a)^(-1),a^6,b^12];
<fp group on the generators [ a, b ]>
gap> h:=Subgroup(g,[g.1*g.2,g.2^2]);
Group([ a*b, b^2 ])
gap> IsAbelian(h);
true
```

El cociente de `g` por `h` se calcula usando simplemente `/`.

```
gap> IsCyclic(g/h);
```

El orden de un grupo se puede calcular con el comando `Order`, y el mínimo m entero tal que $mx = 0$ para todo x en el grupo (éste es el factor invariante más grande del grupo, también conocido como anulador minimal) con `Exponent`.

```
gap> Order(DirectProduct(CyclicGroup(3),CyclicGroup(10)));
30
gap> Exponent(DirectProduct(CyclicGroup(3),CyclicGroup(10)));
30
gap> Order(DirectProduct(CyclicGroup(3),CyclicGroup(15)));
45
gap> Exponent(DirectProduct(CyclicGroup(3),CyclicGroup(15)));
15
```

A continuación mostramos cómo definir una nueva función para calcular el orden de un elemento de un grupo.

```
#Orden de un elemento
 orden:=function(g,a)
 return Order(Subgroup(g,[a]));
 end;

gap> f:=FreeGroup("a","b");
<free group on the generators [ a, b ]>
gap> a:=f.1;; b:=f.2;;
gap> g:=f/[a*b*(b*a)^(-1),a^6,b^12];
<fp group on the generators [ a, b ]>
gap> u:=g.1;; v:=g.2;;
gap> orden(g,u);
6
gap> orden(g,v);
12
gap> orden(g,u*v);
12
gap> orden(g,u*v^2);
6

#Elementos de orden dado
elementosorden:=function(n,g)
return Filtered(Elements(g),a->orden(g,a)=n);
end;
```

```
gap> elementosorden(1,g);
[ <identity ...> ]
gap> elementosorden(2,g);
[ a^3, b^6, a^3*b^6 ]
gap> elementosorden(3,g);
[ a^2, b^4, a^4, a^2*b^4, b^8, a^4*b^4, a^2*b^8, a^4*b^8 ]
gap> elementosorden(g,6);
[ a^5, a^3*b^4, a^2*b^6, b^10, a, a^5*b^4, a^5*b^6, a^3*b^8, a^3*b^10,
  a^4*b^6, a^2*b^10, b^2, a*b^4, a*b^6, a^5*b^8, a^5*b^10, a^3*b^2, a^4*b^10,
  a^2*b^2, a*b^8, a*b^10, a^5*b^2, a^4*b^2, a*b^2 ]
gap> Length(last);
24
```

La función `AbelianInvariant` calcula los divisores elementales de un grupo abeliano, por lo que podemos utilizarla para determinar si dos grupos son isomorfos.

```
sonisomorfos:=function(g,h)
return AbelianInvariants(g)=AbelianInvariants(h);
end;

gap> d:=DirectProduct(CyclicGroup(4),CyclicGroup(15));
<pc group of size 60 with 4 generators>
gap> dd:=AbelianGroup([4,5]);
<pc group of size 20 with 2 generators>
gap> dd:=AbelianGroup([4,15]);
<pc group of size 60 with 2 generators>
gap> AbelianInvariants(d);
[ 3, 4, 5 ]
gap> AbelianInvariants(dd);
[ 3, 4, 5 ]
gap> sonisomorfos(d,dd);
true
```

El comando `IndependentGeneratorsOfAbelianGroup` da un generador para cada una de las componentes de la descomposición cíclica primaria.

```
gap> f:=FreeGroup("a","b");
<free group on the generators [ a, b ]>
gap> a:=f.1;
a
gap> b:=f.2;
b
gap> g:=f/[a*b*(b*a)^(-1),a^6,b^12];
<fp group on the generators [ a, b ]>
gap> h:=Subgroup(g,[g.1*g.2,g.2^2]);
Group([ a*b, b^2 ])
gap> AbelianInvariants(h);
[ 3, 3, 4 ]
gap> IndependentGeneratorsOfAbelianGroup(h);
[ a*b*a*b*a*b, a*b*a*b*a*b*a*b, b^4 ]
gap> AbelianInvariants(g/h);
[ 2 ]
gap> k:=f/[a*b*a^-1*b^-1,b^2];
<fp group on the generators [ a, b ]>
gap> AbelianInvariants(k);
[ 0, 2 ]
gap> IndependentGeneratorsOfAbelianGroup(k);
Error, no method found! For debugging hints type ?Recovery from NoMethodFound
Error, no 3rd choice method found for `IndependentGeneratorsOfAbelianGroup' on\
 1 arguments called from
<compiled or corrupted call value>  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>
```

GAPtiene una base de datos con grupos de orden pequeño. Si queremos acceder a ella, podemos usar los comandos `SmallGroup` y `AllSmallGroups`.

```
gap> AllSmallGroups(360,IsAbelian);
```

```
[ <pc group of size 360 with 6 generators>,
  <pc group of size 360 with 6 generators>,
  <pc group of size 360 with 6 generators>,
  <pc group of size 360 with 6 generators>,
  <pc group of size 360 with 6 generators>,
  <pc group of size 360 with 6 generators> ]
gap> List(last,AbelianInvariants);
[ [ 5, 8, 9 ], [ 2, 4, 5, 9 ], [ 3, 3, 5, 8 ], [ 2, 2, 2, 5, 9 ],
  [ 2, 3, 3, 4, 5 ], [ 2, 2, 2, 3, 3, 5 ] ]
gap> AllSmallGroups(72,IsAbelian);
[ <pc group of size 72 with 5 generators>,
  <pc group of size 72 with 5 generators>,
  <pc group of size 72 with 5 generators>,
  <pc group of size 72 with 5 generators>,
  <pc group of size 72 with 5 generators>,
  <pc group of size 72 with 5 generators> ]
gap> List(last,AbelianInvariants);
[ [ 8, 9 ], [ 2, 4, 9 ], [ 3, 3, 8 ], [ 2, 2, 2, 9 ], [ 2, 3, 3, 4 ],
  [ 2, 2, 2, 3, 3 ] ]
```

4.2 Forma normal de Smith

En GAPlas listas se expresan separando sus elementos por comas dentro de un par de corchetes. Así $[1,2,3]$ es una lista con tres elementos, el 1, el 2 y el 3. Las matrices se representan como una lista de listas. Cada una de esas listas es una fila de la matriz. De esta forma la matriz

$$\begin{pmatrix} 9 & 4 & 5 \\ -4 & 0 & -3 \\ -6 & -4 & -3 \end{pmatrix}$$

se puede definir de la siguiente forma.

```
gap> m:=[[9,4,5],[-4,0,-3],[-6,-4,-3]];
[ [ 9, 4, 5 ], [ -4, 0, -3 ], [ -6, -4, -3 ] ]
```

Si queremos calcular su forma normal de Smith, usamos el comando `SmithNormalFormIntegerMat`.

```
gap> SmithNormalFormIntegerMat(m);
[ [ 1, 0, 0 ], [ 0, 1, 0 ], [ 0, 0, 4 ] ]
```

Si lo que buscamos son las matrices de cambio de base, entonces usamos el comando `SmithNormalFirmIntegerMatTransforms`. Para acceder a un campo de un registro se usa la sintaxis `registro.campo`.

```
gap> fnsm:=SmithNormalFormIntegerMatTransforms(m);
rec( normal := [ [ 1, 0, 0 ], [ 0, 1, 0 ], [ 0, 0, 4 ] ],
  rowC := [ [ 1, 0, 0 ], [ 0, 1, 0 ], [ 0, 0, 1 ] ],
  rowQ := [ [ 3, 2, 3 ], [ -4, -3, -4 ], [ -10, -9, -9 ] ],
  colC := [ [ 1, 0, 0 ], [ 0, 1, 0 ], [ 0, 1, 1 ] ],
  colQ := [ [ 1, 0, 0 ], [ 0, 1, -1 ], [ 0, 0, 1 ] ], rank := 3, signdet := -1,
  rowtrans := [ [ 3, 2, 3 ], [ -4, -3, -4 ], [ -10, -9, -9 ] ],
  coltrans := [ [ 1, 0, 0 ], [ 0, 1, -1 ], [ 0, 1, 0 ] ] )
gap> fnsm.rowtrans*m*fnsm.coltrans;
[ [ 1, 0, 0 ], [ 0, 1, 0 ], [ 0, 0, 4 ] ]
```

Si lo único que queremos son los factores invariantes, podemos usar lo siguiente.

```
gap> ElementaryDivisorsMat(m);
[ 1, 1, 4 ]
```

Para calcular los divisores elementales a partir de una lista de factores invariantes, usamos `AbelianInvariantsOfList` (nótese que los nombres están cambiados respecto a nuestras definiciones).

```
gap> AbelianInvariantsOfList(last);
[ 4 ]
gap> AbelianInvariantsOfList([6,12]);
[ 2, 3, 3, 4 ]
```

5 Formas canónicas usando GAP

5.1 Forma normal de Smith de la matriz característica de una matriz

Veamos cómo se puede calcular la forma normal de Smith sobre $\mathbb{Q}[x]$. Primero declaramos que x va a ser la variable con la que vamos a trabajar sobre los racionales.

```
gap> x:=Indeterminate(Rationals,"x");;
```

Empezamos definiendo una matriz.

```
gap> m:=[[7,-2,1],[-2,10,-2],[1,-2,7]];
[ [ 7, -2, 1 ], [ -2, 10, -2 ], [ 1, -2, 7 ] ]
```

Para convertir `m` a una matriz de entradas racionales, la multiplicamos por el elemento neutro de $\mathbb{Q}[x]$. Calculamos $x\mathrm{Id} - m$.

```
gap> mat:=x*m^0-m*One(x);
[ [ x-7, 2, -1 ], [ 2, x-10, 2 ], [ -1, 2, x-7 ] ]
```

Para diagonalizar esta matriz mediante operaciones elementales por filas y columnas, usamos el comando `DiagonalizeMat` indicando el dominio en el que vamos a efectuar las operaciones.

```
gap> DiagonalizeMat(PolynomialRing(Rationals,1),mat);
[ [ 1, 0, 0 ], [ 0, x-6, 0 ], [ 0, 0, x^2-18*x+72 ] ]
```

Podemos también calcular la matriz asociada a un polinomio.

```
gap> CompanionMat(x^2-18*x+72);
[ [ 0, -72 ], [ 1, 18 ] ]
```

O el polinomio característico de una matriz.

```
gap> CharacteristicPolynomial(m);
x^3-24*x^2+180*x-432
gap> Factors(last);
[ x-12, x-6, x-6 ]
```

Así como su polinomio mínimo (el £ltimo factor invariante).

```
gap> MinimalPolynomial(Rationals,m);
x^2-18*x+72
```

Si quisiésemos conocer los vectores propios usamos `Eigenvectors`.

```
gap> Eigenvectors(Rationals,m);
[ [ 1, -2, 1 ], [ 1, 0, -1 ], [ 0, 1, 2 ] ]
gap> Eigenvalues(Rationals,m);
[ 12, 6 ]
```

Y también podemos calcular los subespacios propios asociados a la matriz.

```
gap> lsp:=Eigenspaces(Rationals,m);
[ <vector space over Rationals, with 1 generators>, <vector space over Rationals, with
    2 generators> ]
gap> List(lsp,GeneratorsOfVectorSpace);
[ [ [ 1, -2, 1 ] ], [ [ 1, 0, -1 ], [ 0, 1, 2 ] ] ]
```

5.2 Cálculo de bases y formas canónicas

Como hemos visto, el comando `DiagonalizeMat` se puede usar para calcular los factores invariantes de una matriz. Por desgracia, en GAP no hay un comando que dé como salida las operaciones elementales por filas y columnas requeridas para llegar a la forma normal de Smith sobre un anillo de polinomios, tal y como ocurría con matrices de entradas enteras. Para salvar este obstáculo mostramos un sencillo procedimiento con un par de ejemplos.

Empezamos definiendo una variable

```
gap> x:=Indeterminate(Rationals,"x");;
```

Dada la matriz

```
gap> a:=[[-1,-1,-1],[-2,0,-1],[6,3,4]];
[ [ -1, -1, -1 ], [ -2, 0, -1 ], [ 6, 3, 4 ] ]
```

de la que queremos conocer su formas canónicas y en las bases en las que éstas se alcanzan, empezamos calculando los factores invariantes de su matriz caracter¡stica

```
gap> xima:=x*a^0-a*One(x);
[ [ x+1, 1, 1 ], [ 2, x, 1 ], [ -6, -3, x-4 ] ]
gap> DiagonalizeMat(PolynomialRing(Rationals,1),xima);
[ [ 1, 0, 0 ], [ 0, x-1, 0 ], [ 0, 0, x^2-2*x+1 ] ]
```

Sabemos por tanto qué aspecto tiene la forma canónica racional asociada a `a`. También obtenemos que el polinomio x^2-2x+1 es el polinomio mínimo de la matriz `a`, y en consecuencia es el anulador minimal, esto es, $(x^2-2x+1)v = 0$ para todo $v \in \mathbb{Q}^3$. El siguiente divisor elemental es $x-1$. Necesitamos un elemento que se anule por x^2-2x+1 y que no se anule con $x-1$. Ese elemento nos servirá para definir el trozo de base asociado a la matriz asociada de x^2-2x+1.

```
gap> NullspaceIntMat(TransposedMat(a^2-2*a+a^0));
[ [ 1, 0, 0 ], [ 0, 1, 0 ], [ 0, 0, 1 ] ]
gap> NullspaceIntMat(TransposedMat(a-a^0));
[ [ 1, 0, -2 ], [ 0, 1, -1 ] ]
gap> [1,0,0] in VectorSpace(Rationals,[[ 1, 0, -2 ], [ 0, 1, -1 ] ]
> );
false
```

Así un candidato bueno para empezar es $(0,0,1)$, y al tener x^2-2x+1 grado dos, utilizamos también la imagen de éste por `a`.

```
gap> u1:=[1,0,0];
[ 1, 0, 0 ]
gap> u2:=a*[1,0,0];
[ -1, -2, 6 ]
```

Para completar la base, es suficiente con encontrar un elemento en $\mathbb{Q}^3$ que se anule por $x-1$ y que no esté en el subespacio generado por `u1` y `u2`. Como el núcleo de `a`-Id está generado por $\{(1,0,-2),(0,1,-1)\}$, y sabemos que no puede ser igual al espacio generado por `u1` y `u2`, probamos con cada uno de los elementos de dicho conjunto.

```
gap> [1,0,-2] in VectorSpace(Rationals,[u1,u2]);
false
gap> v1:=[1,0,-2];
[ 1, 0, -2 ]
```

Así la matriz de cambio de base es

```
gap> s:=TransposedMat([v1,u1,u2]);
[ [ 1, 1, -1 ], [ 0, 0, -2 ], [ -2, 0, 6 ] ]
gap> s^(-1)*a*s;
[ [ 1, 0, 0 ], [ 0, 0, -1 ], [ 0, 1, 2 ] ]
gap> Display(last);
[ [   1,   0,   0 ],
  [   0,   0,  -1 ],
  [   0,   1,   2 ] ]
```

Obteniendo así la forma canónica racional, que en este caso también coincide con la racional primaria.

Para obtener la forma de Jordan y una base en la que se alcanze, modificamos ligeramente el proceso multiplicando $u1$ por `a` $-Id$ en vez de por `a`.

```
gap> w1:=u1;
[ 1, 0, 0 ]
gap> w2:=(a-a^0)*u1;
[ -2, -2, 6 ]
gap> [1,0,-2] in VectorSpace(Rationals,[w1,w2]);
false
gap> t:=TransposedMat([v1,w1,w2]);
[ [ 1, 1, -2 ], [ 0, 0, -2 ], [ -2, 0, 6 ] ]
gap> t^(-1)*a*t;
[ [ 1, 0, 0 ], [ 0, 1, 0 ], [ 0, 1, 1 ] ]
```

Veamos ahora otro ejemplo en el que las formas canonicas racional y racional primaria no coinciden.

```
gap> a:=[[4,2,-4],[2,3,-3],[3,2,-3]];
[ [ 4, 2, -4 ], [ 2, 3, -3 ], [ 3, 2, -3 ] ]
gap> xima:=x*a^0-a*One(x);
[ [ x-4, -2, 4 ], [ -2, x-3, 3 ], [ -3, -2, x+3 ] ]
gap> DiagonalizeMat(PolynomialRing(Rationals,1),xima);
[ [ 1, 0, 0 ], [ 0, 1, 0 ], [ 0, 0, x^3-4*x^2+5*x-2 ] ]
gap> Factors(x^3-4*x^2+5*x-2);
[ x-2, x-1, x-1 ]
gap> u1:=[1,0,0];
[ 1, 0, 0 ]
gap> u2:=a*[1,0,0];
[ 4, 2, 3 ]
gap> u3:=a*u2;
[ 8, 5, 7 ]
gap> s:=TransposedMat([u1,u2,u3]);
[ [ 1, 4, 8 ], [ 0, 2, 5 ], [ 0, 3, 7 ] ]
gap> Display(s^(-1)*a*s);
[ [   0,   0,   2 ],
  [   1,   0,  -5 ],
  [   0,   1,   4 ] ]
```

Como $x^3-4x^2+5x-2=(x-1)^2(x-2)$, para calcular la forma canónica racional primaria necesitamos elementos que se anulen por $(x-1)^2$ y $x-2$. +sos los construimos a partir de `u1`. El primero de ellos nos serve para construir el trozo de base que da lugar a la matriz compañera de $(x-1)^2$, y el segundo para la matriz asociada a $x-2$ (que es simplemente un 2).

```
gap> u11:=(a-2*a^0)*u1;
[ 2, 2, 3 ]
gap> u12:=(a-a^0)^2*u1;
[ 1, 1, 1 ]
gap> v1:=u11;
[ 2, 2, 3 ]
gap> v2:=a*u11;
[ 0, 1, 1 ]
gap> v3:=u12;
[ 1, 1, 1 ]
gap> t:=TransposedMat([v1,v2,v3]);
[ [ 2, 0, 1 ], [ 2, 1, 1 ], [ 3, 1, 1 ] ]
gap> Display(t^(-1)*a*t);
[ [   0,  -1,   0 ],
  [   1,   2,   0 ],
  [   0,   0,   2 ] ]
```

La forma de Jordan, y una base en que se consigue, se pueden calcular a partir de parte de la información ya obtenida.

```
gap> w1:=u11;
[ 2, 2, 3 ]
gap> w2:=(a-a^0)*w1;
[ -2, -1, -2 ]
gap> w3:=v3;
[ 1, 1, 1 ]
gap> p:=TransposedMat([w1,w2,w3]);
[ [ 2, -2, 1 ], [ 2, -1, 1 ], [ 3, -2, 1 ] ]
gap> Display(p^(-1)*a*p);
[ [  1,  0,  0 ],
  [  1,  1,  0 ],
  [  0,  0,  2 ] ]
```

5.3 Forma de Jordan a partir del polinomio característico y subespacios propios

Vemos ahora un método alternativo para calcular formas canónicas de Jordan con un ejemplo, sin pasar previamente por la forma canónica racional primaria.

```
gap> x:=Indeterminate(Rationals,"x");
x
gap> m75:=[[1,0,0,0],[0,1,0,0],[-2,-2,0,1],[-2,0,-1,-2]];
[ [ 1, 0, 0, 0 ], [ 0, 1, 0, 0 ], [ -2, -2, 0, 1 ], [ -2, 0, -1, -2 ] ]
gap> CharacteristicPolynomial(m75);
x^4-2*x^2+1
gap> Factors(last);
[ x-1, x-1, x+1, x+1 ]
```

Usamos las siguientes funciones para calcular los valores propios y los vectores propios.

```
gap> Eigenvectors(Rationals,TransposedMat(m75));
[ [ 1, 0, -2, 0 ], [ 0, 1, -3/2, 1/2 ], [ 0, 0, 1, -1 ] ]
gap> Eigenvalues(Rationals,TransposedMat(m75));
[ 1, -1 ]
```

O bien ...

```
gap> Eigenspaces(Rationals,TransposedMat(m75));
[ <vector space over Rationals, with 2 generators>,
  <vector space over Rationals, with 1 generators> ]
gap> List(last,GeneratorsOfVectorSpace);
[ [ [ 1, 0, -2, 0 ], [ 0, 1, -3/2, 1/2 ] ], [ [ 0, 0, 1, -1 ] ] ]
gap> m75*[1,0,-2,0];
[ 1, 0, -2, 0 ]
gap> m75*[0,0,1,-1];
[ 0, 0, -1, 1 ]
```

Como el núcleo de $\mathtt{m75}+\mathrm{Id}$ tiene dimensión uno frente a la doble multiplicidad de $(x-1)$ en el polinomio característico, tenemos que calcular el núcleo de $(\mathrm{Id}+\mathtt{m75})^2$, y escoger un elemento del núcleo de $(\mathrm{Id}+\mathtt{m75})^2$ que no esté en el núcleo de $\mathtt{m75}+\mathrm{Id}$.

```
gap> NullspaceMat(TransposedMat((m75^0+One(x)*m75)^2));
[ [ 0, 0, 1, 0 ], [ 0, 0, 0, 1 ] ]
gap> (m75^0+m75)*[0,0,0,1];
[ 0, 0, 1, -1 ]
```

Tenemos por tanto la base que buscamos en la que la matriz adopta la forma de Jordan.

```
gap> p:=[[ 1, 0, -2, 0 ], [ 0, 1, -3/2, 1/2 ] ,[0,0,0,1],[0,0,1,-1]];
[ [ 1, 0, -2, 0 ], [ 0, 1, -3/2, 1/2 ], [ 0, 0, 0, 1 ], [ 0, 0, 1, -1 ] ]
gap> p:=TransposedMat(p);
[ [ 1, 0, 0, 0 ], [ 0, 1, 0, 0 ], [ -2, -3/2, 0, 1 ], [ 0, 1/2, 1, -1 ] ]
gap> Display(p^-1*m75*p);
[ [   1,   0,   0,   0 ],
  [   0,   1,   0,   0 ],
  [   0,   0,  -1,   0 ],
  [   0,   0,   1,  -1 ] ]
```