

StandardFF

(Version 1.0)

September 2023

Frank Lübeck

Frank Lübeck Email: Frank.Luebeck@Math.RWTH-Aachen.De

Homepage: <https://www.math.rwth-aachen.de/~Frank.Luebeck>

Copyright

© 2020– by Frank Lübeck

This package may be distributed under the terms and conditions of the GNU Public License Version 3 or later, see <https://www.gnu.org/licenses>.

Colophon

This package implements the constructions in the paper [Lüb23], that is it provides relatively easy to reproduce generators of finite fields and compatible generators of their multiplicative cyclic subgroups.

Contents

1	Introduction to StandardFF package	4
1.1	Aim	4
2	Standard finite fields	5
2.1	Definition of standard finite fields	5
2.2	Creating standard finite fields	5
2.3	Elements in standard finite fields	7
2.4	Embeddings of standard finite fields	8
3	Standard generators of cyclic groups	12
3.1	Generators of multiplicative groups	12
4	Utilities from the StandardFF package	14
4.1	A simple bijection on a range	14
4.2	Finding linear combinations	14
4.3	Irreducibility over finite fields	15
4.4	Connection to Conway polynomials	15
4.5	Discrete logarithms	17
4.6	Minimal polynomials of sequences	17
4.7	Brauer characters with respect to different lifts	18
4.8	Known factorizations of multiplicative group orders	20
4.9	Some loops for StandardFF	21
4.10	Undocumented features	22
	References	23
	Index	24

Chapter 1

Introduction to StandardFF package

1.1 Aim

This GAP-package provides a reference implementation for the standardized constructions of finite fields and generators of cyclic subgroups defined in the article [Lüb23].

The main functions are `FF` (2.2.1) to construct the finite field of order p^n and `StandardCyclicGenerator` (3.1.1) to construct a standardized generator of the multiplicative subgroup of a given order m in such a finite field. The condition on m is that it divides $p^n - 1$ and that GAP can factorize this number. (The factorization of the multiplicative group order $p^n - 1$ is not needed.)

Each field of order p^n comes with a natural \mathbb{F}_p -basis which is a subset of the natural basis of each extension field of order p^{nm} . The union of these bases is a basis of the algebraic closure of \mathbb{F}_p . Each element of the algebraic closure can be identified by its degree d over its prime field and a number $0 \leq k \leq p^d - 1$ (see `SteinitzPair` (2.4.1)) or, equivalently, by a certain multivariate polynomial (see `AsPolynomial` (2.3.1)). This can be useful for transferring finite field elements between programs which use the same construction of finite fields.

The standardized generators of multiplicative cyclic groups have a nice compatibility property: There is a unique group isomorphism from the multiplicative group $\bar{\mathbb{F}}_p^\times$ of the algebraic closure of the finite field with p elements into the group of complex roots of unity whose order is not divisible by p which maps a standard generator of order m to $\exp(2\pi i/m)$. In particular, the minimal polynomials of standard generators of order $p^n - 1$ for all n fulfill the same compatibility conditions as Conway polynomials (see `ConwayPolynomial` (**Reference: ConwayPolynomial**)). This can provide an alternative for the lifts used by `BrauerCharacterValue` (**Reference: BrauerCharacterValue**) which works for a much wider set of finite field elements where Conway polynomials are very difficult or impossible to compute.

A translation of existing Brauer character tables relative to the lift defined by Conway polynomials to the lift defined by our `StandardCyclicGenerator` (3.1.1) can be computed with `StandardValuesBrauerCharacter` (4.7.1), provided the relevant Conway polynomials are known.

The article [Lüb23] also defines a standardized embedding of GAPs finite fields constructed with `GF` (**Reference: GF for field size**) into the algebraic closure of the prime field \mathbb{F}_p constructed here. This is available with `StandardIsomorphismGF` (2.4.5).

Chapter 2

Standard finite fields

2.1 Definition of standard finite fields

In [Lüb23] we define for each prime p and positive integer n a standardized model for the finite field with p^n elements. This is done by defining for each prime r polynomials of degree r which define recursively r -power extensions of the prime field $GF(p)$ and by combining these for all $r|n$ in a unique tower of extensions of finite fields where the successive degrees are non-decreasing primes.

Relative to this tower of prime degree extensions the resulting field comes with a natural basis over the prime field which we call the *tower basis*. This construction has the nice property that whenever $n|m$ then the tower basis of the field with p^n elements is a subset of the tower basis of the field with p^m elements. (See [Lüb23] for more details.)

Expressing elements as linear combination of the tower basis we define a bijection from the elements in the field of order p^n to the range $[0 \dots p^n - 1]$; we call the number assigned to an element its *Steinitz number*.

Via this construction each element in the algebraic closure of $GF(p)$ can be identified by its degree d over the prime field and its Steinitz number in the field with p^d elements (we call this a *Steinitz pair*).

Since arithmetic in simple algebraic extensions is more efficient than in iterated extensions we construct the fields recursively as simple extensions, and including information about the base change between the tower basis and the basis given by the powers of the generator.

2.2 Creating standard finite fields

2.2.1 Constructing standard finite fields

▷ `StandardFiniteField(p, n)` (function)

▷ `FF(p, n)` (function)

Returns: a finite field

▷ `StandardPrimeDegreePolynomial(p, r, k)` (function)

Returns: a polynomial of degree r

The arguments are a prime p and a positive integer n . The function `FF` (or its synonym `StandardFiniteField`) is one of the main functions of this package. It returns a standardized field F of order p^n . It is implemented as a simple extension over the prime field $GF(p)$ using `AlgebraicExtension` (**Reference:** `AlgebraicExtension`)

The polynomials used for the prime degree extensions are accessible with `StandardPrimeDegreePolynomial`. For arguments p , r , k it returns the irreducible polynomial of degree r for the k -th iterated extension of degree r over the prime field. The polynomial is in the variable xr_k and the coefficients can contain variables xr_l with $l < k$.

Example

[illegible]

2.2.2 Filters for standard fields

- ▷ `IsStandardPrimeField(F)` (property)
- ▷ `IsStandardFiniteField(F)` (property)
- ▷ `IsStandardFiniteFieldElement(x)` (Category)

Returns: true or false

These properties identify the finite fields constructed with `FF` (2.2.1). Prime fields constructed as `FF(p, 1)` have the property `IsStandardPrimeField`. They are identical with `GF(p)`, but calling them via `FF` (2.2.1) we store some additional information in these objects.

The fields constructed by `FF(p,k)` with $k > 1$ have the property `IsStandardFiniteField`. Elements x in such a field are in `IsStandardFiniteFieldElement`.

Example

```
gap> F := FF(19,1);
GF(19)
gap> IsStandardFiniteField(F);
false
gap> IsStandardPrimeField(F);
true
gap> F := FF(23,48);
FF(23, 48)
gap> IsStandardFiniteField(F);
true
gap> IsStandardFiniteFieldElement(Random(F));
true
```

2.3 Elements in standard finite fields

For fields in `IsStandardFiniteField` (2.2.2) we provide functions to map elements to their linear combination of the tower basis, to their Steinitz number and Steinitz pair, or to their representing multivariate polynomial with respect to all prime degree extensions, and vice versa.

2.3.1 Maps for elements of standard finite fields

- ▷ `AsVector(a)` (method)
Returns: a vector over prime field of F
- ▷ `ElementVector(F, v)` (method)
Returns: an element in F
- ▷ `AsPolynomial(a)` (method)
Returns: a polynomial in variables of the tower of F
- ▷ `ElementPolynomial(F, pol)` (method)
Returns: an element in F
- ▷ `SteinitzNumber(a)` (method)
Returns: an integer
- ▷ `ElementSteinitzNumber(F, nr)` (method)
Returns: an element in F

Here, F is always a standard finite field (`IsStandardFiniteField` (2.2.2)) and a is an element of F .

`AsVector` (2.3.1) returns the coefficient vector of a with respect to the tower basis of F . And vice versa `ElementVector` returns the element of F with the given coefficient vector.

Similarly, `AsPolynomial` (2.3.1) returns the (reduced) polynomial in the indeterminates defining the tower of F . Here, for each prime r dividing the degree of the field the polynomial defining the k -th extension of degree r over the prime field is written in the variable xr_k . And `ElementPolynomial` returns the element of F represented by the given polynomial (which does not need to be reduced).

Finally, `SteinitzNumber` returns the Steinitz number of a . And `ElementSteinitzNumber` returns the element with given Steinitz number.

Example

```
gap> F := FF(17, 12);
FF(17, 12)
gap> a := PrimitiveElement(F);; a := a^11-3*a^5+a;
ZZ(17,12,[0,1,0,0,0,14,0,0,0,0,0,1])
gap> v := AsVector(a);
< immutable compressed vector length 12 over GF(17) >
gap> a = ElementVector(F, v);
true
gap> ExtRepOfObj(a) = v * TowerBasis(F);
true
gap> pol := AsPolynomial(a);;
gap> ElementPolynomial(F, pol^10) = a^10;
true
gap> nr := SteinitzNumber(a);
506020624175737
gap> a = ElementSteinitzNumber(F, nr);
true
gap> ## primitive element of FF(17, 6)
```

```

gap> y := ElementSteinitzNumber(F, 17^5);
ZZ(17,12,[0,0,1,0,0,0,12,0,0,0,5,0])
gap> y = ValuePol([0,0,1,0,0,0,12,0,0,0,5,0], PrimitiveElement(F));
true
gap> x6 := Indeterminate(FF(17,1), "x6");;
gap> MinimalPolynomial(FF(17,1), y, x6) = DefiningPolynomial(FF(17,6));
true

```

2.4 Embeddings of standard finite fields

The tower basis of a standard finite field F contains the tower basis of any subfield. This yields a construction of canonical embeddings of all subfields of F into F . And one can easily read off the smallest subfield containing an element in F from its coefficient vector with respect to the tower basis. Each element of the algebraic closure of $\text{FF}(p, 1)$ is uniquely determined by its degree d and its Steinitz number in $\text{FF}(p, d)$.

2.4.1 SteinitzPair

- ▷ `SteinitzPair(a)` (operation)
Returns: a pair of integers
- ▷ `SteinitzPair(K, snr)` (method)
Returns: a pair of integers
- ▷ `SteinitzNumber(K, pair)` (method)
Returns: an integer

The argument a must be an element in `IsStandardFiniteFieldElement` (2.2.2). Then `SteinitzPair` returns a pair $[d, nr]$ where d is the degree of a over the prime field $\text{FF}(p, 1)$ and nr is the Steinitz number of a considered as element of $\text{FF}(p, d)$.

In the second variant a standard finite field K is given and the Steinitz number of an element in K and the result is the Steinitz pair of the corresponding element.

The inverse map is provided by a method for `SteinitzNumber` (2.4.1) which gets a standard finite field and a Steinitz pair.

Example

```

gap> F := FF(7, 360);
FF(7, 360)
gap> t := ElementSteinitzNumber(F, 7^10);; # prim. elt of FF(7,12)
gap> sp := SteinitzPair(t);
[ 12, 117649 ]
gap> H := FF(7, 12);
FF(7, 12)
gap> b := ElementSteinitzNumber(H, 117649);
ZZ(7,12,[0,1,0,0,0,0,0,0,0,0,0,0])
gap> Value(MinimalPolynomial(FF(7,1), t), b);
ZZ(7,12,[0])
gap> nr := SteinitzNumber(t);
282475249
gap> nr = SteinitzNumber(F, sp);
true
gap> sp = SteinitzPair(F, nr);
true

```



```
gap> One(FF(19,5)) = ZZ(19,5,[1]);
true
gap> ZZ(19,5,Z(19^5)); # zero of ConwayPolynomial(19,5)
ZZ(19,5,[12,5,3,4,5])
```

2.4.4 MoveToSmallestStandardField

- ▷ MoveToSmallestStandardField(x) (function)
- ▷ \+(x, y) (method)
- ▷ *(x, y) (method)
- ▷ \-(x, y) (method)
- ▷ \/(x, y) (method)

Returns: a field element

Here x and y must be elements in standard finite fields (of the same characteristic).

Then MoveToSmallestStandardField returns the element x as element of the smallest possible degree extension over the prime field.

The arithmetic operations are even possible when x and y are not represented as elements in the same field. In this case the elements are first mapped to the smallest field containing both.

Example

```
gap> F := FF(1009,4);
FF(1009, 4)
gap> G := FF(1009,6);
FF(1009, 6)
gap> x := (PrimitiveElement(F)+One(F))^13;
ZZ(1009,4,[556,124,281,122])
gap> y := (PrimitiveElement(G)+One(G))^5;
ZZ(1009,6,[1,5,10,10,5,1])
gap> x+y;
ZZ(1009,12,[557,0,936,713,332,0,462,0,843,191,797,0])
gap> x-y;
ZZ(1009,12,[555,0,73,713,677,0,97,0,166,191,212,0])
gap> x*y;
ZZ(1009,12,[253,289,700,311,109,851,345,408,813,657,147,887])
gap> x/y;
ZZ(1009,12,[690,599,714,648,184,217,563,130,251,675,73,782])
gap> z := -y + (x+y);
ZZ(1009,12,[556,0,0,713,0,0,784,0,0,191,0,0])
gap> SteinitzPair(z);
[ 4, 125450261067 ]
gap> x=z;
true
gap> MoveToSmallestStandardField(z);
ZZ(1009,4,[556,124,281,122])
```

2.4.5 StandardIsomorphismGF

- ▷ StandardIsomorphismGF(F) (function)
- Returns:** a field isomorphism

The argument F must be a standard finite field, say $\text{FF}(p, n)$ such that **GAP** can generate $\text{GF}(p, n)$. This function returns the field isomorphism from $\text{GF}(p, n)$ to F , which sends $Z(p, n)$ to the element with Steinitz pair computed by `SteinitzPairConwayGenerator` (4.4.3).

Example

```
gap> F := FF(13,21);
FF(13, 21)
gap> iso := StandardIsomorphismGF(F);
MappingByFunction( GF(13^21), FF(13, 21), function( x ) ... end )
gap> K := GF(13,21);
GF(13^21)
gap> x := Random(K);;
gap> l := [1,2,3,4,5];;
gap> ValuePol(l, x)^iso = ValuePol(l, x^iso);
true
gap> y := ElementSteinitzNumber(F, SteinitzPairConwayGenerator(F)[2]);;
gap> PreImageElm(iso, y);
z
```

Chapter 3

Standard generators of cyclic groups

3.1 Generators of multiplicative groups

The multiplicative group of each finite field is cyclic and so for each divisor m of its order there is a unique subgroup of order m .

In [Lüb23] we define standardized generators x_m of these cyclic groups in the standard finite fields described in chapter 2 which fulfill the following compatibility condition: If $k|m$ then $x_m^{m/k} = x_k$.

The condition that x_m can be computed is that m can be factorized. (If we do not know the prime divisors of m then we cannot show that a given element has order m .) Note that this means that we can compute x_m in $\text{FF}(p, n)$ when $m|(p^n - 1)$ and we know the prime divisors of m , even when the factorization of $(p^n - 1)$ is not known.

In the case that the factorization of $m = p^n - 1$ is known the corresponding x_m is a standardized primitive root of $\text{FF}(p, n)$ that can be computed.

Let $l|n$ and set $m = p^n - 1$ and $k = p^l - 1$. Then x_m and x_k are the standard primitive roots of $\text{FF}(p, n)$ and $\text{FF}(p, l)$ (considered as subfield of $\text{FF}(p, n)$), respectively. The compatibility condition says that $x_m^{m/k} = x_k$. This shows that the minimal polynomials of x_m and x_k over the prime field fulfill the same compatibility conditions as Conway polynomials (see `ConwayPolynomial` (**Reference: ConwayPolynomial**)).

3.1.1 StandardCyclicGenerator

▷ `StandardCyclicGenerator(F[, m])` (operation)

▷ `StandardPrimitiveRoot(F)` (attribute)

Returns: an element of F or fail

The argument F must be a standard finite field (see `FF (2.2.1)`) and m a positive integer. If m does not divide $|F| - 1$ the function returns fail. Otherwise a standardized element x_m of order m is returned, as described above.

The argument m is optional, if not given its default value is $|F| - 1$. In this case x_m can also be computed with the attribute `StandardPrimitiveRoot`.

Example

```
gap> F := FF(67, 18); # Conway polynomial was hard to compute
FF(67, 18)
gap> x := PrimitiveElement(F);
ZZ(67,18,[0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0])
gap> xprim := StandardPrimitiveRoot(F);
```

```

gap> k := (Size(F)-1) / Order(x);
6853662165340556076084083497526
gap> xm := StandardCyclicGenerator(F, Order(x));
gap> xm = xprim^k;
true
gap> F := FF(23, 201); # factorization of (|F| - 1) not known
FF(23, 201)
gap> m:=79*269*67939;
1443771689
gap> (Size(F)-1) mod m;
0
gap> OrderMod(23, m);
201
gap> xm := StandardCyclicGenerator(F, m);
gap> IsOne(xm^m);
true
gap> ForAll(Factors(m), r-> not IsOne(xm^(m/r)));
true
gap> F := FF(7,48);
FF(7, 48)
gap> K := FF(7,12);
FF(7, 12)
gap> emb := Embedding(K, F);
gap> x := StandardPrimitiveRoot(F);
gap> y := StandardPrimitiveRoot(K);
gap> y^emb = x^((Size(F)-1)/(Size(K)-1));
true
gap> v := Indeterminate(FF(7,1), "v");
v
gap> px := MinimalPolynomial(FF(7,1), x, v);
gap> py := MinimalPolynomial(FF(7,1), y, v);
gap> Value(py, PowerMod(v, (Size(F)-1)/(Size(K)-1), px)) mod px;
0*Z(7)

```

Chapter 4

Utilities from the StandardFF package

4.1 A simple bijection on a range

4.1.1 StandardAffineShift

▷ `StandardAffineShift(q, i)` (function)

Returns: an integer in range $[0..q-1]$

This function returns $(mi + a) \bmod q$, where m is the largest integer prime to q and $\leq 4q/5$, and a is the largest integer $\leq 2q/3$.

For fixed q this function provides a bijection on the range $[0..q-1]$.

Example

```
gap> List([0..10], i-> StandardAffineShift(11, i));  
[ 7, 4, 1, 9, 6, 3, 0, 8, 5, 2, 10 ]
```

4.2 Finding linear combinations

4.2.1 FindLinearCombination

▷ `FindLinearCombination(v, start)` (function)

Returns: a pair `[serec, lk]` of a record and vector or fail

Repeated calls of this function build up a semiechelon basis from the given arguments v which must be row vectors. To initialize a computation the function is called with a start vector v and false as second argument. The return value is a pair `[serec, lk]` where `serec` is a record which collects data from the previous calls of the function and `lk` is a row vector which expresses v as linear combination of the vectors from previous calls, or fail if there is no such linear combination. In the latter case the data in the record is extended with the linearly independent vector v .

In the following example we show how to compute a divisor of the minimal polynomial of a matrix.

Example

```
gap> mat := Product(GeneratorsOfGroup(Sp(30,5)));;  
gap> x := Indeterminate(GF(5), "x");;  
gap> v := (mat^0)[1];;  
gap> b := FindLinearCombination(v, false);;  
gap> repeat  
>   v := v*mat;  
>   l := FindLinearCombination(v, b[1]);
```

```

> until IsList(l[2]);
gap> mp := Value(UnivariatePolynomial(GF(5),
>      Concatenation(-l[2], [One(GF(5))])), x);
x^30+Z(5)^3*x^29+Z(5)^3*x+Z(5)^0
gap> # equal to minimal polynomial because of degree
gap> mp = Value(MinimalPolynomial(GF(5), mat), x);
true

```

4.3 Irreducibility over finite fields

4.3.1 IsIrreducibleCoeffList

▷ IsIrreducibleCoeffList(*coeffs*, *q*) (function)

Returns: true or false

The argument *coeffs* must be a list of elements in a finite field with *q* elements (or some subfield of it).

The function checks if the univariate polynomial f with coefficient list *coeffs* (ending with the leading coefficient) is irreducible over the field with *q* elements.

The algorithm computes the greatest common divisor of f with $X^{q^i} - X$ for $i = 1, 2, \dots$ up to half of the degree of f .

Example

```

gap> cs := Z(3)^0 * ConwayPol(3,8);
[ Z(3), Z(3), Z(3), 0*Z(3), Z(3)^0, Z(3), 0*Z(3), 0*Z(3), Z(3)^0 ]
gap> IsIrreducibleCoeffList(cs, 3);
true
gap> F := FF(17,4);; x := PrimitiveElement(F);;
gap> cs := [x, x+x^0, 0*x, x^0];
[ ZZ(17,4,[0,1,0,0]), ZZ(17,4,[1,1,0,0]), ZZ(17,4,[0]), ZZ(17,4,[1]) ]
gap> while not IsIrreducibleCoeffList(cs, 17^4) do
>   cs[1] := cs[1] + One(F);
> od;
gap> cs;
[ ZZ(17,4,[8,1,0,0]), ZZ(17,4,[1,1,0,0]), ZZ(17,4,[0]), ZZ(17,4,[1]) ]

```

4.4 Connection to Conway polynomials

4.4.1 FindConjugateZeroes

▷ FindConjugateZeroes(*K*, *cpol*, *qq*) (function)

Returns: a list of field elements

The arguments must be a finite field *K*, a polynomial *cpol* over *K* (or its coefficient list) and the order *qq* of a subfield of *K*. The polynomial must have coefficients in the subfield with *qq* elements, must be irreducible over this subfield and split into linear factors over *K*. The function FindConjugateZeroes returns the list of zeroes of *cpol* in *K*.

Example

```

gap> K := GF(67,18);
GF(67^18)
gap> F := FF(67,18);
FF(67, 18)

```

```
gap> p1 := DefiningPolynomial(K);;
gap> p2 := DefiningPolynomial(F);;
gap> lK := FindConjugateZeroes(K, p2, 67);;
gap> lF := FindConjugateZeroes(F, p1, 67);;
gap> Minimum(List(lF, SteinitzNumber));
12274789318154414216760893584069
```

4.4.2 ZeroesConway

▷ ZeroesConway(F) (function)

Returns: a list of field elements

Here, F must be a standard finite field, say of degree n over the prime field with p elements. This function returns the same as `FindConjugateZeroes(F, One(F)*ConwayPol(p, n), p)` (using a specific implementation).

Example

```
gap> F := FF(23,29);
FF(23, 29)
gap> l := Set(FindConjugateZeroes(F, One(F)*ConwayPol(23,29), 23));;
gap> l = Set(ZeroesConway(F));
true
```

4.4.3 SteinitzPairConwayGenerator

▷ SteinitzPairConwayGenerator(F) (function)

Returns: a pair of integers

For a standard finite field F of order q for which a Conway polynomial (see `ConwayPolynomial` (**Reference: ConwayPolynomial**)) is known this function returns the `SteinitzPair` (2.4.1) for the element of F corresponding to $Z(q)$ (which is by definition the zero of the Conway polynomial in F with the smallest Steinitz number which is compatible with the choice in all proper subfields).

This is used to construct the `StandardIsomorphismGF` (2.4.5) for F .

Example

```
gap> F := FF(23,18);
FF(23, 18)
gap> st := SteinitzPairConwayGenerator(F);
[ 18, 1362020736983803830549380 ]
gap> st9 := SteinitzPairConwayGenerator(FF(23,9));
[ 9, 206098743447 ]
gap> st6 := SteinitzPairConwayGenerator(FF(23,6));
[ 6, 45400540 ]
gap> z := ElementSteinitzNumber(F, st[2]);;
gap> z9 := ElementSteinitzNumber(F, SteinitzNumber(F, st9));;
gap> z6 := ElementSteinitzNumber(F, SteinitzNumber(F, st6));;
gap> e9 := (Size(F)-1)/(23^9-1);
1801152661464
gap> e6 := (Size(F)-1)/(23^6-1);
21914624580056211
gap> z9 = z^e9;
true
gap> z6 = z^e6;
true
```



```
gap> l := Filtered(ZeroesConway(F), x-> x^e9 = z9 and x^e6 = z6);;
gap> List(l, SteinitzNumber);
[ 1362020736983803830549380 ]
```

4.5 Discrete logarithms

4.5.1 DLog

▷ `DLog(base, x[, m])` (function)

Returns: an integer

The argument *base* must be a multiplicative element and *x* must lie in the cyclic group generated by *base*. The third argument *m* must be the order of *base* or its factorization. If *m* is not given, it is computed first. This function returns the discrete logarithm, that is an integer *e* such that $\text{base}^e = x$.

If *m* is prime then Shanks' algorithm is used (which needs $O(\sqrt{m})$ space and time). Otherwise let $m = rl$ and $e = a + br$ with $0 \leq a < r$. Then $a = \text{DLog}(\text{base}^l, x^l, r)$ and $b = \text{DLog}(\text{base}^r, x/\text{base}^a, l)$.

This function is used for a method of LogFFE (**Reference: LogFFE**).

Example

```
gap> F := FF(67, 12);
FF(67, 12)
gap> st := SteinitzPairConwayGenerator(F);
[ 12, 5118698034368952035290 ]
gap> z := ElementSteinitzNumber(F, st[2]);;
gap> x := StandardPrimitiveRoot(F);;
gap> DLog(z, x, Size(F)-1);
231901568073107448223
gap> K := GF(67,12);
GF(67^12)
gap> zz := Z(67^12);
Z
gap> LogFFE(zz^2+1, zz);
1667375214152688471247
```

4.6 Minimal polynomials of sequences

4.6.1 InvModCoeffs

▷ `InvModCoeffs(fcoeffs, gcoeffs)` (operation)

Returns: a list of fail

The arguments *fcoeffs* and *gcoeffs* are coefficient lists of two polynomials *f* and *g*. This operation returns the coefficient list of the inverse f^{-1} modulo *g*, if *f* and *g* are coprime, and fail otherwise.

The default method computes the inverse by the extended Euclidean algorithm.

Example

```
gap> f := Z(13)^0*[ 1, 10, 1, 11, 0, 1 ];;
gap> g := Z(13)^0*[ 5, 12, 5, 12, 2, 0, 2 ];;
gap> InvModCoeffs(f, g);
fail
gap> GcdCoeffs(f, g);
[ Z(13)^0, 0*Z(13), Z(13)^0 ]
```

```

gap> f[1]:=f[1]+1;;
gap> finv := InvModCoeffs(f, g);
[ Z(13)^9, Z(13)^10, Z(13)^10, Z(13)^8, Z(13)^5, Z(13)^6 ]
gap> pr := ProductCoeffs(finv, f);;
gap> ReduceCoeffs(pr, g);; ShrinkRowVector(pr);; pr;
[ Z(13)^0 ]

```

4.6.2 BerlekampMassey

▷ BerlekampMassey(*u*) (function)

Returns: a list of field elements

The argument *u* is a list of elements in a field *F*. This function implements the Berlekamp–Massey algorithm which returns the shortest sequence *c* of elements in *F* such that for each $i > l$, the length of *c*, we have $u[i] = \sum_{j=1}^l u[i-j]c[j]$.

Example

```

gap> x := Indeterminate(GF(23), "x");;
gap> f := x^5 + Z(23)^16*x + Z(23)^12;;
gap> u := List([1..50], i-> Value(x^i mod f, 0));;
gap> c := BerlekampMassey(u);;
gap> ForAll([6..50], i-> u[i] = Sum([1..5], j-> u[i-j]*c[j]));
true
gap> -c;
[ 0*Z(23), 0*Z(23), 0*Z(23), Z(23)^16, Z(23)^12 ]

```

4.6.3 MinimalPolynomialByBerlekampMassey

▷ MinimalPolynomialByBerlekampMassey(*x*) (function)

▷ MinimalPolynomialByBerlekampMasseyShoup(*x*) (function)

Returns: the minimal polynomial of *x*

Here *x* must be an element of an algebraic extension field F/K . (*K* must be the LeftActingDomain (**Reference:** LeftActingDomain) of *F*). This function computes the minimal polynomial of *x* over *K* by applying the Berlekamp–Massey algorithm to the list of traces of x^i .

The second variant uses the algorithm by Shoup in [Sho99].

Example

```

gap> x := Indeterminate(GF(23), "x");;
gap> f := x^5 + Z(23)^16*x + Z(23)^12;;
gap> F := AlgebraicExtension(GF(23), f);;
gap> mp := MinimalPolynomialByBerlekampMassey(PrimitiveElement(F));;
gap> Value(mp, x) = f;
true
gap> mp = MinimalPolynomialByBerlekampMasseyShoup(PrimitiveElement(F));
true

```

4.7 Brauer characters with respect to different lifts

Let *G* be a finite group, $g \in G$, and $\rho : G \rightarrow GL(d, p^n)$ be a representation over a finite field. The Brauer character value $\chi(g)$ of ρ at *g* is defined as the sum of the eigenvalues of $\rho(g)$ in the algebraic closure of \mathbb{F}_p lifted to complex roots of unity.

The lift used by `BrauerCharacterValue` (**Reference: `BrauerCharacterValue`**) and in the computation of many Brauer character tables (available through the `CTblLib` package) is defined by Conway polynomials (see `ConwayPolynomial` (**Reference: `ConwayPolynomial`**)): They define the primitive root $Z(q)$ in $GF(q)$ which is mapped to $\exp(2\pi i/(q-1))$ (that is $E(q-1)$ in `GAP`).

Another lift is defined by the function `StandardCyclicGenerator` (3.1.1) provided by this package. Here, `StandardCyclicGenerator(F, m)` is mapped to $\exp(2\pi i/m)$ (that is $E(m)$ in `GAP`).

The following function translates between these two lifts.

4.7.1 StandardValuesBrauerCharacter

▷ `StandardValuesBrauerCharacter(tab, bch)` (function)

Returns: a Brauer character

▷ `IsGaloisInvariant(tab, bch)` (function)

Returns: true or false

The argument `tab` must be a Brauer character table for which the Brauer characters are defined with respect to the lift given by Conway polynomials. And `bch` must be an irreducible Brauer character of this table.

The function `StandardValuesBrauerCharacter` recomputes the values corresponding to the lift given by `StandardCyclicGenerator` (3.1.1), provided that the Conway polynomials for computing the Frobenius character values of `bch` are available. If Conway polynomials are missing the corresponding character values are substituted by `fail`. If the result does not contain `fail` it is a class function which is Galois conjugate to `bch` (see `GaloisCyc` (**Reference: `GaloisCyc` for a class function**)).

The utility `IsGaloisInvariant` returns true if all Galois conjugates of `bch` are Brauer characters in `tab`. If this is the case then different lifts will permute the Galois conjugates and all of them are Brauer characters with respect to any lift.

WARNING: The result of this function may not be a valid Brauer character for the table `tab` (that is an integer linear combination of irreducible Brauer characters in `tab`). For a proper handling of several lifts the data structure of Brauer character tables needs to be extended (it must refer to the lift), and then the result of this function should return a Brauer character of another table that refers to another lift.

Example

```
gap> tab := BrauerTable("M", 19);
BrauerTable( "M", 19 )
gap> # cannot translate some values to different lift
gap> fail in AsList(StandardValuesBrauerCharacter(tab, Irr(tab)[16]));
true
gap> # but table contains the irreducible Brauer characters for any lift
gap> ForAll(Irr(tab), bch-> IsGaloisInvariant(tab, bch));
true
gap> tab := BrauerTable("A18", 3);
BrauerTable( "A18", 3 )
gap> # here different lifts lead to different Brauer character tables
gap> bch := Irr(tab)[38];
gap> IsGaloisInvariant(tab, bch);
false
gap> new := StandardValuesBrauerCharacter(tab, bch);
gap> fail in AsList(new);
false
```

```
gap> Position(Irr(tab), new);
fail
```

The inverse of a lift is used to reduce character values in characteristic 0 modulo a prime p . Choosing a lift is equivalent to choosing a p -modular system. GAP has the function `FrobeniusCharacterValue` (**Reference: FrobeniusCharacterValue**) which computes this reduction with respect to the lift defined by Conway polynomials.

Here is the corresponding function with respect to the lift constructed in this package.

4.7.2 Frobenius character values

▷ `SmallestDegreeFrobeniusCharacterValue(cyc, p)` (function)

Returns: a positive integer or fail

▷ `StandardFrobeniusCharacterValue(cyc, F)` (function)

Returns: an element of F or fail

The argument `cyc` must be a cyclotomic whose conductor and denominator are not divisible by the prime integer p or the characteristic of the standard finite field F .

The order of the multiplicative group of F must be divisible by the conductor of `cyc`.

Then `StandardFrobeniusCharacterValue` returns the image of `cyc` in F under the homomorphism which maps the root of unity $E(n)$ to the `StandardCyclicGenerator` (3.1.1) of order n in F . If the conditions are not fulfilled the function returns fail.

The function `SmallestDegreeFrobeniusCharacterValue` returns the smallest degree of a field over the prime field of order p containing the image of `cyc`.

Example

```
gap> SmallestDegreeFrobeniusCharacterValue(E(13), 19);
12
gap> F := FF(19,12);
FF(19, 12)
gap> x := StandardFrobeniusCharacterValue(E(13),F);;
gap> x^13;
ZZ(19,12,[1])
gap> x = StandardCyclicGenerator(F, 13);
true
gap> cc := (E(13)+1/3)^4;;
gap> xx := StandardFrobeniusCharacterValue(cc, F);;
gap> xx = StandardFrobeniusCharacterValue(E(13)+1/3, F)^4;
true
```

4.8 Known factorizations of multiplicative group orders

4.8.1 CANFACT

▷ `CANFACT` (global variable)

This variable contains a list where for each prime $p < 10000$ the entry `CANFACT[p]` holds a list of integers i such that the number $p^i - 1$ (the order of the multiplicative group of the finite field $\text{FF}(p, i)$) can be factored by GAP in a short time. This is based on the enormous efforts to find factors of numbers of this form, see [Cro].

For $p < 10$ the range of considered exponents is $2 \leq i \leq 2000$, for $10 < p < 100$ it is $2 \leq i \leq 500$, and for $100 < p < 10000$ it is $2 \leq i \leq 100$.

These data describe (in May 2022) 112968 pairs p, i such that `StandardPrimitiveRoot(FF(p,i))` can be computed in reasonable time. Only for 10858 of these cases **GAP** knows or can easily compute the corresponding Conway polynomial (see `ConwayPolynomial` (**Reference: ConwayPolynomial**)).

The current content of **CANFACT** was generated after updating the data in the **FactInt** package concerning factors of numbers of the form $a^n \pm 1$. If you want to use that list you should also update your **GAP** installation with:

Example

```
FetchMoreFactors(
  "https://maths-people.anu.edu.au/~brent/ftp/factors/factors.gz",
  false);
FetchMoreFactors(
  "http://myfactorcollection.mo00.com:8090/brentdata/May31_2022/factors.gz",
  true);
```

4.9 Some loops for StandardFF

4.9.1 Computing all fields in various ranges

- ▷ `AllPrimeDegreePolynomials(p, bound)` (function)
- ▷ `AllFF(p, bound)` (function)
- ▷ `AllPrimitiveRoots(p, bound)` (function)
- ▷ `AllPrimitiveRootsCANFACT()` (function)
- ▷ `AllFieldsWithConwayPolynomial(["ConwayGen"] [,] ["MiPo"])` (function)

These function compute all fields in some range, sometimes with further data. All functions return a list with some timings and print a log-file in the current directory.

`AllPrimeDegreePolynomials` computes all irreducible polynomials of prime degree needed for the construction of all finite fields of order p^i , $1 \leq i \leq bound$. This is the most time consuming part in the construction of the fields.

`AllFF` computes all `FF(p,i)` for $1 \leq i \leq bound$. When the previous function was called before for the same range, this function spends most of its time by computing the minimal polynomials of the standardized primitive elements of `FF(p,i)`.

`AllPrimitiveRoots` computes the standardized primitive roots in `FF(p,i)` for $1 \leq i \leq bound$. The most time consuming cases are when a large prime divisor r of $p^i - 1$ already divides $p^j - 1$ for some $j < i$ (but then r divides i/j). Cases where **GAP** cannot factorize $p^i - 1$ (that is i is not contained in `CANFACT[p]`) are skipped.

`AllPrimitiveRootsCANFACT` does the same as the previous function for all pairs p,i stored in `CANFACT` (4.8.1).

`AllFieldsWithConwayPolynomial` computes all `FF(p,i)` for the cases where **GAP** knows the precomputed `ConwayPolynomial(p,i)`. With the optional argument "ConwayGen" the function computes for all fields the `SteinitzPairConwayGenerator` (4.4.3) and writes it into a file `SteinitzPairConway`. With the optional argument "MiPo" the function also computes the minimal polynomials of the `StandardPrimitiveRoot` (3.1.1) and writes it to a file `MiPoPrimitiveRoots` (these polynomials have the same compatibility properties as Conway polynomials).

4.10 Undocumented features

We mention some features of this package which may be temporary, vanish or changed.

A directory `ntl` contains some simple standalone programs which use the library NTL [\[Sho\]](#). There is a function `StandardIrreducibleCoeffListNTL(K, d, a)` which can be used instead of `StandardIrreducibleCoeffListNTL(K, d, a)` when K is a prime field. This gives a good speedup for not too small d , say $d > 500$.

References

- [Cro] J. Crombie. Factor collection $a^n \pm 1$. <http://myfactorcollection.mooo.com:8090/>. 20
- [Lüb23] F. Lübeck. Standard generators of finite fields and their cyclic subgroups. *J. Symbolic Comput.*, 117:51--67, 2023. Similar to arXiv 2107.02257. 2, 4, 5, 12
- [Sho] V. Shoup. Ntl: A library for doing number theory. <https://libntl.org/>. 22
- [Sho99] V. Shoup. Efficient computation of minimal polynomials in algebraic extensions of finite fields. In *Proceedings of the 1999 International Symposium on Symbolic and Algebraic Computation (Vancouver, BC)*, page 53--58. ACM, New York, 1999. 18

Index

- `*`
 - for standard finite field elements, [10](#)
- `\+`
 - for standard finite field elements, [10](#)
- `\/`
 - for standard finite field elements, [10](#)
- `\-`
 - for standard finite field elements, [10](#)
- `AllFF`, [21](#)
- `AllFieldsWithConwayPolynomial`, [21](#)
- `AllPrimeDegreePolynomials`, [21](#)
- `AllPrimitiveRoots`, [21](#)
- `AllPrimitiveRootsCANFACT`, [21](#)
- `AsPolynomial`
 - for elements in standard finite fields, [7](#)
- `AsVector`
 - for elements in standard finite fields, [7](#)
- `BerlekampMassey`, [18](#)
- `CANFACT`, [20](#)
- `DLog`, [17](#)
- `ElementPolynomial`, [7](#)
- `ElementSteinitzNumber`, [7](#)
- `ElementVector`, [7](#)
- `Embedding`
 - for standard finite fields, [9](#)
- `FF`, [5](#)
- `FindConjugateZeroes`, [15](#)
- `FindLinearCombination`, [14](#)
- `InvModCoeffs`, [17](#)
- `IsGaloisInvariant`, [19](#)
- `IsIrreducibleCoeffList`, [15](#)
- `IsStandardFiniteField`, [6](#)
- `IsStandardFiniteFieldElement`, [6](#)
- `IsStandardPrimeField`, [6](#)
- `MinimalPolynomialByBerlekampMassey`, [18](#)
- `MinimalPolynomialByBerlekampMassey-Shoup`, [18](#)
- `MoveToSmallestStandardField`, [10](#)
- `SmallestDegreeFrobeniusCharacterValue`, [20](#)
- `StandardAffineShift`, [14](#)
- `StandardCyclicGenerator`, [12](#)
- `StandardFiniteField`, [5](#)
- `StandardFrobeniusCharacterValue`, [20](#)
- `StandardIsomorphismGF`, [10](#)
- `StandardPrimeDegreePolynomial`, [5](#)
- `StandardPrimitiveRoot`, [12](#)
- `StandardValuesBrauerCharacter`, [19](#)
- `SteinitzNumber`, [7](#)
 - for Steinitz pair, [8](#)
- `SteinitzPair`, [8](#)
 - for Steinitz number, [8](#)
- `SteinitzPairConwayGenerator`, [16](#)
- `ZeroesConway`, [16](#)
- `ZZ`, [9](#)
 - for IsFFE, [9](#)