FSR

•••

Version 1.1.0

22 March 2017

Nusa Zidaric
Mark Aagaard
Guang Gong

Nusa Zidaric Email: nzidaric@uwaterloo.ca
Homepage: http://

Mark Aagaard Email: maagaard@uwaterloo.ca
Homepage: http://

Guang Gong Email: ggong@uwaterloo.ca
Homepage: http://

Abstract

The GAP package FSR ...

Copyright

Acknowledgements

...

Contents

Chapter 1

Preface

The GAP package FSR implements Feedback Shift Registers

Chapter 2

FSR (Feedback Shift Register)

2.1 Common functionality

We define an object FSR (Feedback Shift Register), which can come in two flavours: with linear feedback `LFSR` (2.2.1) and nonlinear feedback `NLFSR` (2.3.2). Because of many similarities between the two, the basic common functionality can be found here, while specialized functions (such as `LFSR` and `NLFSR` object creation) in corresponding sections.

2.1.1 IsFSR

▷ `IsFSR` (filter)

This is the category of `FSR` objects. Objects in this category are created using functions `LFSR` (2.2.1) or `NLFSR` (2.3.2).

2.1.2 FieldPoly

▷ `FieldPoly(`*fsr*`)` (attribute)
▷ `UnderlyingField(`*fsr*`)` (attribute)
▷ `FeedbackVec(`*fsr*`)` (attribute)
▷ `OutputTap(`*fsr*`)` (attribute)

`FieldPoly` of the *fsr* stores the irreducible polynomial used to construct the extension field or 1 in case of a prime field.

`UnderlyingField` of the *fsr* is the finite field over which the *fsr* is defined (all indeterminates and constants are from this field).

NOTE: it may seem redundant to sore both `FieldPoly` and `UnderlyingField`, however, they are used by other functions in the package.

`FeedbackVec` of the *fsr* stores the coefficients of the `CharPoly` without its leading term in case of `LFSR`, and coefficients of the nonzero monomials present in the multivariate function defining the feedback in case of `NLFSR`.

`OutputTap` holds the output tap position(s): the sequence elements are taken from the stage(s) listed in `OutputTap`.

2.1.3 Length

▷ `Length(fsr)` (attribute)
▷ `InternalStateSize(fsr)` (attribute)
▷ `Threshold(fsr)` (attribute)

`Length` of the *fsr* is the number of its stages.

`InternalStateSize` of the *fsr* is size in bits needed to store the state $length \cdot width$, where $width = DegreeOverPrimeField(UnderlyingField(\texttt{fsr}))$.

`Threshold` of the *fsr* is currently set to $Characteristic(\texttt{fsr})^t + \ell$, where $t = InternalStateSize(\texttt{fsr})$ and $\ell = Length(\texttt{fsr})$.

2.1.4 ChangeBasis

▷ `ChangeBasis(fsr, B)` (method)
▷ `WhichBasis(fsr)` (method)

`ChangeBasis` allows changing the basis of the *fsr* to basis *B*. Basis B must be given for `UnderlyingField(fsr)` over its prime subfield.

`WhichBasis` returns the basis currently set for the *fsr*. Elements in the *fsr* state are still represented in GAP native representation, but the functions with basis switch turned on will print the elements w.r.t to currently set basis.

2.1.5 LoadFSR

▷ `LoadFSR(fsr, ist)` (method)

Loading the *fsr* with the initial state *ist*, which is a *FFE* vector of same length as *fsr* and with elements from its underlying finite field. If either of those two requirements is violated, loading fails and error message appears. At the time of loading the initial sequence elements (ie zeroth elements) are obtained and `numsteps` is set to 0.

2.1.6 StepFSR

▷ `StepFSR(fsr[, elm])` (method)

Perform one step the *fsr*, ie. compute the new `state` and update the `numsteps`, then output the elements denoted by `OutputTap`. If the optional parameter *elm* is used then the new element is computed as a sum of computed feedback and *elm*. Elemen *elm* must be an element of the underlying finite field.

As this is a way to destroy the linearity of an `LFSR`, we refer to `StepFSR` with the optiomal nonzero *elm* as `nonlinear step`. Similarly, the `NLFSR` can also have an extra element added to the (already nonlinear) feedback.

Returns an error if the *fsr* is not loaded!

2.1.7 RunFSR

▷ `RunFSR(fsr[, ist, num, pr])` (method)
▷ `RunFSR(fsr, elm[, num, pr])` (method)
▷ `RunFSR(fsr, ist, elmvec[, pr])` (method)
▷ `RunFSR(fsr, z, elmvec[, pr])` (method)

Returns: A sequence of elements generated by FSR.

The `fsr` will be run for $min(num, Threshold(fsr))$ number of steps: value Threshold(`fsr`) is used by all versions without explicit `num` and enforced when `num` exceeds Threshold(`fsr`). There is an optional printing switch `pr`, with default set to *false*; if *true* then the state and the output sequence element(s) are printed in GAP shell on every step of the `fsr` (we call this output for `RunFSR`), and the currently set basis `B` is used for representation of elements.

- `RunFSR(fsr[, num, pr] )` - run `fsr` for `num/threshold` steps with/without output
- `RunFSR(fsr, ist[, num, pr] )` - load `fsr` with `ist`, then run `fsr` for `num/threshold` steps with/without output (ie. *linear* version)
- `RunFSR(fsr, elm[, num, pr] )` - run `fsr` for `num/threshold` steps, whereby the SAME element `elm` is added to the feedback at each step, with/without output (ie. *non-linear* version)
- `RunFSR(fsr, ist, elmvec[, pr] )` - load `fsr` with `ist`, then run `fsr` for *Length(*`elmvec`*)* steps, whereby one element of `elmvec` is added to the feedback at each step (starting with `elmvec[1]`), with/without output (ie. *non-linear* version). NOTE: the sequence returned has length *Length(elmvec)+1*, because the zeroth sequence element is returned at the time of loading the FSR.
- `RunFSR(fsr, z, elmvec[, pr] )` - input `z` must be set to 0 to indicate we want to continue a run with new `elmvec`: run `fsr` for *Length(*`elmvec`*)* steps, whereby one element of `elmvec` is added to the feedback at each step (starting with `elmvec[1]`), with/without output (ie. *non-linear* version). NOTE: the sequence returned has length *Length(elmvec)*.

For the load and run versions, element seq_0 is a part of the output sequence, hence the output sequence has the length `num+1/threshold+1/Length(elmvec)+1`.

For versions without the loading of `ist`, calling `RunFSR` returns an error if the `fsr` is not loaded!

The ouput of `RunFSR` is:

- sequence of *FFE*s : $\mathrm{seq}_0, \mathrm{seq}_1, \mathrm{seq}_2, \ldots$, for *Length*(*OutputTap*)=1
- sequence of vectors, each of them with *t FFE*s : $\mathrm{seq}_0, \mathrm{seq}_1, \mathrm{seq}_2, \ldots$, where $\mathrm{seq}_i = (\mathrm{seq}_{i1}, \ldots, \mathrm{seq}_{it})$ for *Length*(*OutputTap*)=t

Example of `RunFSR` called for an lfsr *test* over F_{2^4}, with initial state *ist*, print switch *true* for basis *B*, with run length 5:

Example

```
gap> K := GF(2);; x := X(K, "x");;
gap> f := x^4 + x^3 + 1;; F := FieldExtension(K, f);; B := Basis(F);;
gap> y := X(F, "y");; l := y^4+ y+ Z(2^4);;
gap> test := LFSR(K, f, l);;
< empty LFSR given by CharPoly = y^4+y+Z(2^4)>
```

```
gap> ist :=[0*Z(2), Z(2^4), Z(2^4)^5, Z(2)^0 ];;
gap> RunFSR(test, ist, 5, true);
using basis B := [ Z(2)^0, Z(2^4)^7, Z(2^4)^14, Z(2^4)^6 ]
elm                       [ 3,         ...         ...,0 ]  with taps  [ 0 ]
  [ [ 0, 0, 0, 0 ], [ 0, 1, 1, 0 ], [ 1, 1, 0, 1 ], [ 1, 0, 0, 0 ] ]
                                                       [ 1, 0, 0, 0 ]
        [ [ 1, 0, 1, 1 ], [ 0, 0, 0, 0 ], [ 0, 1, 1, 0 ], [ 1, 1, 0, 1 ] ]
                                                       [ 1, 1, 0, 1 ]
        [ [ 0, 1, 1, 1 ], [ 1, 0, 1, 1 ], [ 0, 0, 0, 0 ], [ 0, 1, 1, 0 ] ]
                                                       [ 0, 1, 1, 0 ]
        [ [ 1, 0, 1, 1 ], [ 0, 1, 1, 1 ], [ 1, 0, 1, 1 ], [ 0, 0, 0, 0 ] ]
                                                       [ 0, 0, 0, 0 ]
        [ [ 1, 0, 1, 1 ], [ 1, 0, 1, 1 ], [ 0, 1, 1, 1 ], [ 1, 0, 1, 1 ] ]
                                                       [ 1, 0, 1, 1 ]
        [ [ 1, 1, 0, 1 ], [ 1, 0, 1, 1 ], [ 1, 0, 1, 1 ], [ 0, 1, 1, 1 ] ]
                                                       [ 0, 1, 1, 1 ]
[ Z(2)^0, Z(2^2), Z(2^4), 0*Z(2), Z(2^4)^2, Z(2^4)^11 ]
```

Example of RunFSR called for an lfsr *test* over F_{2^4}, with initial state *ist*, print switch *true* for basis *B*, with 5 nonlinear inputs :

Example

```
gap> elmvec := [Z(2^4)^2, Z(2^4)^2, Z(2^2), Z(2^4)^7, Z(2^4)^6];;
gap> RunFSR(test, ist, elmvec, true);
using basis B := [ Z(2)^0, Z(2^4)^7, Z(2^4)^14, Z(2^4)^6 ]
elm                [ 3,         ...         ...,0 ]  with taps  [ 0 ]
[ 0, 0, 0, 0 ]               [ [ 0, 0, 0, 0 ], [ 0, 1, 1, 0 ], [ 1, 1, 0, 1 ],
 [ 1, 0, 0, 0 ] ]               [ 1, 0, 0, 0 ]
[ 1, 0, 1, 1 ]               [ [ 0, 0, 0, 0 ], [ 0, 0, 0, 0 ], [ 0, 1, 1, 0 ],
 [ 1, 1, 0, 1 ] ]               [ 1, 1, 0, 1 ]
[ 1, 0, 1, 1 ]               [ [ 1, 1, 0, 0 ], [ 0, 0, 0, 0 ], [ 0, 0, 0, 0 ],
 [ 0, 1, 1, 0 ] ]               [ 0, 1, 1, 0 ]
[ 1, 1, 0, 1 ]               [ [ 0, 1, 1, 0 ], [ 1, 1, 0, 0 ], [ 0, 0, 0, 0 ],
 [ 0, 0, 0, 0 ] ]               [ 0, 0, 0, 0 ]
[ 0, 1, 0, 0 ]               [ [ 0, 1, 0, 0 ], [ 0, 1, 1, 0 ], [ 1, 1, 0, 0 ],
 [ 0, 0, 0, 0 ] ]               [ 0, 0, 0, 0 ]
[ 0, 0, 0, 1 ]               [ [ 1, 1, 0, 1 ], [ 0, 1, 0, 0 ], [ 0, 1, 1, 0 ],
 [ 1, 1, 0, 0 ] ]               [ 1, 1, 0, 0 ]
[ Z(2^4)^9, Z(2^2), Z(2^4), 0*Z(2), 0*Z(2), Z(2^4)^9 ]
```

In both examples above the there is a column *elm*, which is in first case empty, because we are not adding nonlinear inputs to the feedback, while in the second example, this column shows the element being added at each step (empty in first row - the loading step). Also note that the two examples above use the call LoadFSR, which adds the elm seq_0 to the sequence, so both sequences above are of length *num+1/Length(elmvec)+1*, ie 6. The last row in both examples is the actual sequence obtained from this run, and is kept in Zechs logarithm representation.

Example

```
gap> RunFSR(test,  ist); Length(last);
[ Z(2)^0, Z(2^2), Z(2^4), 0*Z(2), Z(2^4)^2, Z(2^4)^11, Z(2^4)^2, Z(2^4)^2,
```

```
Z(2^2), Z(2^4)^7, Z(2^4)^6, Z(2^4)^11, Z(2^2)^2, Z(2^4)^14, Z(2^4)^8,
 Z(2^4)^3, Z(2^2)^2, Z(2^4)^2, Z(2^4), Z(2^4)^2, Z(2^4)^9 ]
21
```

Last example above shows a sequence of length 21, ie *threshold+1*, getting first sequence element from LoadFSR followed by *threshold* iterations of StepFSR.

2.2 LFSR specific funcionality

2.2.1 LFSR

▷ `LFSR(F, charpol[, B, tap])` (function)
▷ `LFSR(K, fieldpol, charpol[, B, tap])` (function)
▷ `LFSR(F, charpol[, B, tap])` (function)
▷ `LFSR(p, m, n[, tap])` (function)

Returns: An empty `LFSR` with components `init`, `state` , `numsteps` and `basis`

Different ways to create an `LFSR` obljcet, main difference is in creation of the underlying finite field.

Inputs:

- *F* - the underlying finite field (either an extension field or a prime field)
- *B* - basis of F over its prime subfield
- *charpol* - `LFSR` dfining polynomial
- *fieldpol* - defifning polynomial of the extension field (must be irreducible)
- *p* - characteeristic
- *m* - degree of extension (degree of *fieldpol*)
- *n* - length of `LFSR` (degree of *charpoly*)
- *tap* - optional parameter: the output tap (must be a positive integer or a list of positive integers) and will be changed to the default S_0 if the specified integer is out of `LFSR` range.

Compoents:

- `init` - *FFE* vector of length n=deg(charpol), storing the initial state of the `LFSR`, with indeces from n-1, ..., 0
- `state` - *FFE* vector of length n=deg(charpol), storing the current state of the `LFSR`, with indeces from n-1, ..., 0
- `numsteps` - the number of steps performed thus far (initialized to -1 when created, set to 0 when loaded using `LoadFSR` (2.1.5) and incremented by 1 with each step (using `StepFSR` (2.1.6)))
- `basis` - basis of F over its prime subfield (if no basis is given this field is set to canonical basis of F over its prime subfield)

Attributes `FieldPoly` (2.1.2), `UnderlyingFied` (**??**), `CharPoly`, `FeedbackVec` (2.1.2), `Length` (2.1.3) and `OutputTap` (2.1.2) and the property `IsLinearFeedback` are set during the construction of an`LFSR`.

If there is something wrong with the arguments (e.g. attempting to create an extension field using a reducible poynomial), an error message appears and the function returns `fail`.

Example below shows how to create an empty `LFSR` over F_{2^4} created as extension of F_2, called *test*, firstly without a specified basis, and then with basis *B*:

Example

```
gap> K := GF(2);; x := X(K, "x");;
gap> f := x^4 + x^3 + 1;; F := FieldExtension(K, f);;
gap> y := X(F, "y");; l := y^4+ y+ Z(2^4);;
gap> test := LFSR(K, f, l);
< empty LFSR given by CharPoly = y^4+y+Z(2^4)>
gap> WhichBasis(test);
CanonicalBasis( GF(2^4) )
gap> B := Basis(F, Conjugates(Z(2^4)^3));;
gap> test := LFSR(K, f, l, B);
< empty LFSR given by CharPoly = y^4+y+Z(2^4)>
gap> WhichBasis(test);
Basis( GF(2^4), [ Z(2^4)^3, Z(2^4)^6, Z(2^4)^12, Z(2^4)^9 ] )
```

2.2.2 IsLinearFeedback

▷ `IsLinearFeedback(lfsr)` (property)
▷ `IsLFSR(lfsr)` (filter)

If we were to represent the `lsfr` with a multivariate polynomial, DegreeOfPolynomial would return 1 - the feedback polynomial is linear and `IsLinearFeedback` is set to *true*. (ie. only linear terms are present: monomials with only one variable)

Filter `IsLFSR` is defined as and-filter of `IsFSR` and `IsLinearFeedback`.

2.2.3 CharPoly

▷ `CharPoly(lfsr)` (attribute)

Attribute holding the characteristic polynomial (the feedback polynomial).

2.2.4 IsPeriodic

▷ `IsPeriodic(lfsr)` (property)
▷ `IsUltPeriodic(lfsr)` (property)
▷ `IsMaxSeqLFSR(lfsr)` (property)
▷ `Period(lfsr)` (attribute)
▷ `PeriodIrreducible(lfsr)` (method)
▷ `PeriodReducible(lfsr)` (method)

Properties, attributes and methods concerning the periodicity of the output sequence(s), generated by the *lsfr*.

Properties:

- `IsPeriodic`: true if constant term of `CharPoly` != 0 (8.11 lidl, niederreiter)
- `IsUltPeriodic`: true if `IsLFSR` is true (8.7 lidl, niederreiter)
- `IsMaxSeqLFSR`: true if `CharPoly` is primitive (ref???)

Attributes:

- `Period`: holds the period of the UNKNOWNEntity(LFSR)

Methods to compute the period:

- `PeriodIrreducible`:
- `PeriodReducible`:

2.3 NLFSR specific funcionality

2.3.1 ChooseField

▷ `ChooseField(`*F*`)` (function)

Workaround for the `NLFSR` object definition: we need to fix the chosen underlying finite field and prepare indeterminates in the chosen field. The indeterminates will be used for the multivariable polynomial, which will define the `NLFSR` feedback. Current threshold is set by global `MaxNLFSRLen` = 100.

2.3.2 NLFSR

▷ `NLFSR(`*K, clist, mlist, len[, tap]*`)` (function)

▷ `NLFSR(`*K, fieldpol, clist, mlist, len[, tap]*`)` (function)

Returns: An empty `NLFSR` with components `init`, `state` and `numsteps`

Different ways to create an `NLFSR` obljject, main difference is in creation of the underlying finite field.

NOTE: before creating the `NLFSR`, we must always create the indeterminates to be used for the feedback using `ChooseField` function call!!! please see example below

Inputs:

- *F* - the underlying finite field (either an extension field or a prime field)
- *fieldpol* - defifning polynomial of the extension field (must be irreducible)
- *clist* - list of coefficients for the monomials in *mlist*
- *mlist* - list of monomials
- *len* - length of `NLFSR`

- *tap* - optional parameter: the output tap (must be a positive integer or a list of positive integers) and will be changed to the default S_0 if the specified integer is out of NLFSRrange.

NOTE: *clist* and *mlist* must be of same length, all elements in *clist* must belong to the underlying field. Monomials in *mlist* must not include any indeterminates that are out of range specified by *len*: stages of NLFSR are represented by indeterminants and the feedback is not allowed to use a stage that doesnt exist. A second constraint on *mlist* requires that it must contain at least one monomial of degree > 1, otherwise we must create an LFSR.

Compoents:

- init - *FFE* vector of length n=deg(charpol), storing the initial state of the NLFSR, with indeces from n-1, ..., 0
- state - *FFE* vector of length n=deg(charpol), storing the current state of the NLFSR, with indices from n-1, ..., 0
- numsteps - the number of steps performed thus far (initialized to -1 when created, set to 0 when loaded using LoadFSR (2.1.5) and incremented by 1 with each step (using StepFSR (2.1.6)))

Attributes FieldPoly (2.1.2), UnderlyingFied (**??**), MultivarPoly, FeedbackVec (2.1.2), IndetList (2.3.4), Length (2.1.3) and OutputTap (2.1.2) and the property IsNonLinearFeedback are set during the construction of an NLFSR.

If there is something wrong with the arguments (e.g. attempting to create an extension field using a reducible poynomial), an error message appears and the function returns fail.

Example

```
gap>  F := GF(2);;  clist := [One(F), One(F)];; mlist := [x_0*x_1, x_2];;
Error, Variable: 'x_0' must have a value
not in any function at line 2 of *stdin*
gap> test := NLFSR(F, clist, mlist, 3);
Error, Variable: 'mlist' must have a value
not in any function at line 3 of *stdin*
gap> ChooseField(F);
You can now create an NLFSR with up to 100 stages
with up to  100 nonzero terms
gap> mlist := [x_0*x_1, x_2];;
gap> test := NLFSR(F, clist, mlist, 3);
< empty NLFSR of length 3,
 given by MultivarPoly = x_0*x_1+x_2>
```

2.3.3 IsNonLinearFeedback

▷ IsNonLinearFeedback(*nlfsr*) (property)
▷ IsNLFSR(*nlfsr*) (filter)

For the multivariate polynomial given by *clist* and *mlist*, DegreeOfPolynomial greter than 1 sets IsNonLinearFeedback to *true*. otherwise it prints out a warning that you need to use the LFSR constructor instead.

Filter IsNLFSR is defined as and-filter of IsFSR and IsNonLinearFeedback.

NOTE: at the same time IsLinearFeedback is set to *false* (for coding purposes).

2.3.4 MultivarPoly

▷ `MultivarPoly(nlfsr)` (attribute)
▷ `IndetList(nlfsr)` (attribute)

`MultivarPoly` holds the multivariate function defining the feedback of the `NLFSR`. `IndetList` holds all the indeterminates that are present in `MultivarPoly` and `FeedbackVec` holds only the nonzero coefficients (as opposed to the `LFSR`, where this field holds coefficients for all stages of the `FSR`). The feedback element is computed from `MultivarPoly`, `IndetList` and `state`, and not from `FeedbackVec`.

Example

```
gap> MultivarPoly(test); IndetList(test);
x_0*x_1+x_2
[ 0, 1, 2 ]
```

Chapter 3

Output formatting functions and TEX drawing functions

3.1 View/Display/Print/PrintAll

3.1.1 ViewObj

▷ `ViewObj(lfsr)` (method)
▷ `PrintObj(lfsr[, b])` (method)
▷ `PrintAll(lfsr[, b])` (method)

Different detail on *nsr* created either by `LFSR` (2.2.1) or `NLFSR` (2.3.2):

- `Display/View`:
  - for LFSR: show the `CharPoly` and wheter or not the *fsr* is empty
  - for NLFSR: show the `MultivarPoly` and wheter or not the *fsr* is empty
- `Print`: same as `Display/View` if *fsr* is empty, otherwise it also shows the values of components `state` , `numsteps` and `basis`
- `PrintAll`: same as `Print` if *fsr* is empty, otherwise it also shows the values of all four components `init`, `state` , `numsteps` and `basis` with additional information about the underlying field and the tap positions

Both `Print` and `PrintAll` can be used with optional parameter *b* for desiered output format: when `true` the output will used the currently chosen basis.

Examples below show different outputs for an LFSR:

Example
```
gap> K := GF(2);; x := X(K, "x");;
gap> f := x^4 + x^3 + 1;; F := FieldExtension(K, f);;
gap> y := X(F, "y");; l := y^4+ y+ Z(2^4);;
gap> test := LFSR(K, f, l);;
gap> Print(test);
empty LFSR over GF(2^4) given by CharPoly = y^4+y+Z(2^4)
gap> ist := [ 0*Z(2), Z(2^4), Z(2^2), Z(2)^0 ];; LoadFSR(test, ist);
```

```
Z(2)^0
gap> Print(test);
LFSR over GF(2^4)  given by CharPoly = y^4+y+Z(2^4)
with basis =[ Z(2)^0, Z(2^4)^7, Z(2^4)^14, Z(2^4)^6 ]
with current state =[ 0*Z(2), Z(2^4), Z(2^2), Z(2)^0 ]
after  0 steps
gap> RunFSR(test,5);
[ Z(2^2), Z(2^4), 0*Z(2), Z(2^4)^2, Z(2^4)^11 ]
gap> Print(test);
LFSR over GF(2^4)  given by CharPoly = y^4+y+Z(2^4)
with basis =[ Z(2)^0, Z(2^4)^7, Z(2^4)^14, Z(2^4)^6 ]
with current state =[ Z(2^2), Z(2^4)^2, Z(2^4)^2, Z(2^4)^11 ]
after  5 steps
gap> PrintAll(test);
LFSR over GF(2^4)  given by CharPoly = y^4+y+Z(2^4)
with basis =[ Z(2)^0, Z(2^4)^7, Z(2^4)^14, Z(2^4)^6 ]
with feedback coeff =[ 0*Z(2), 0*Z(2), Z(2)^0, Z(2^4) ]
with initial state  =[ 0*Z(2), Z(2^4), Z(2^2), Z(2)^0 ]
with current state  =[ Z(2^2), Z(2^4)^2, Z(2^4)^2, Z(2^4)^11 ]
after 5 steps
with output from stage S_0
gap>  PrintAll(test, true);
LFSR over GF(2^4) defined by FieldPoly=x^4+x^3+Z(2)^0  given by CharPoly = y^4\
+y+Z(2^4)
with basis =[ Z(2)^0, Z(2^4)^7, Z(2^4)^14, Z(2^4)^6 ]
with feedback coeff =[ [ 0, 0, 0, 0 ], [ 0, 0, 0, 0 ], [ 1, 0, 0, 0 ],
  [ 0, 1, 1, 0 ] ]
with initial state  =[ [ 0, 0, 0, 0 ], [ 0, 1, 1, 0 ], [ 1, 1, 0, 1 ],
  [ 1, 0, 0, 0 ] ]
with current state  =[ [ 1, 1, 0, 1 ], [ 1, 0, 1, 1 ], [ 1, 0, 1, 1 ],
  [ 0, 1, 1, 1 ] ]
after 5 steps
with output from stage S_0
```

Examples below show different outputs for an NLFSR:

Example

```
gap>  F := GF(2);; ChooseField(F);
You can now create an NLFSR with up to 100 stages
with up to  100 nonzero terms
gap>  clist := [One(F), One(F)];; mlist := [x_0*x_1, x_2];;
gap>  test := NLFSR(F, clist, mlist, 3);
< empty NLFSR of length 3 over GF(2),
  given by MultivarPoly = x_0*x_1+x_2>
gap> Display(test);
< empty NLFSR of length 3 over GF(2),
 given by MultivarPoly = x_0*x_1+x_2>
gap> PrintAll(test,true);
empty NLFSR of length 3 over GF(2),
  given by MultivarPoly = x_0*x_1+x_2
with basis =[ Z(2)^0 ]
with initial state  =[ [ 0 ], [ 0 ], [ 0 ] ]
```

```
with current state  =[ [ 0 ], [ 0 ], [ 0 ] ]
after initialization
with output from stage S_0
```

3.2 Writing to *.txt or *.tex

3.2.1 WriteAllFSR

▷ `WriteAllFSR(output, fsr)` (function)

Equivalent to PrintAll , but it writes to an output stream. NOTE: The basis switch must be present and if *true*, the currently set basis of the `fsr` is used.

3.2.2 WriteSequenceFSR

▷ `WriteSequenceFSR(output, fsr, sequence)` (function)
▷ `WriteTBSequenceFSR(output, fsr, sequence)` (function)
▷ `WriteTEXSequenceByGenerator(output, fsr, sequence, strGen, gen)` (function)

`WriteSequenceFSR` writes the sequence generated by some version of RunFSR(lfsr) to *.txt file, with addition of separating sequences from different taps and writing them in currently set basis of the `fsr`.

`WriteTBSequenceFSR` is a version of `WriteSequenceFSR` intended for testbenching purposes: the generated sequence is written to *.txt file, with sequences from different taps separated into *columns* separated by "\t". Again the currently set basis of the `fsr` is used. The order of columns is determined by `OutputTap(fsr)`.

`WriteTEXSequenceByGenerator` is a *.tex version of `WriteSequenceFSR` but allows to write the sequence elements as powers of a chosen generator `gen`. Generator `gen` is used to get the exponents of the elements, and the elements themselfs are printed as $\`strGen`^{exponent}$, where strGen must be a string representing a greek letter in *.tex, for example `strGen` "alpha" will give α.

3.2.3 WriteRunFSR

▷ `WriteRunFSR(output, lfsr, ist, numsteps)` (function)
▷ `WriteNonlinRunFSR(output, lfsr, ist, numsteps)` (function)

`WriteRunFSR` is an output to *.txt version of RunFSR(`fsr`, `ist`, `num`), with addition of separating sequences from different taps and writing them in currently set basis of the `fsr`. Before the run begins and after loading, the `WriteAllFSR(output, x, true)` is called to record the FSR being used. When the run is finished, `WriteSequenceFSR` is called to record the output sequence in compact version. `WriteRunFSR` returns the sequence generated by this run.

`WriteNonlinRunFSR` is an output to *.txt version of RunFSR(`fsr`, `ist`, `elmvec`). `WriteNonlinRunFSR` returns a sequence generated by this run, however, the length of returned sequence is Length(`elmvec`)+1, because the first element of the output sequence is the element that was loaded with `ist`.

3.2.4 WriteTEXRunFSR

▷ `WriteTEXRunFSR(output, fsr, ist, numsteps)` (function)
▷ `WriteTEXNonlinRunFSR(output, fsr, ist, numsteps)` (function)
▷ `WriteTEXRunFSRByGenerator(output, fsr, ist, numsteps, strGen, gen)` (function)

`WriteTEXRunFSR` is an output to *.tex version of RunFSR(*fsr*, *ist*, *num*), which writes a table that can be included dircetly (except for the label). Rows of the table represent the steps of the FSR and include the state of the FSR and the elements from stages specfied by outputTap, that is the sequence outputs at this step. The table entries (FFEs) are printed using currently set basis of the *fsr*. When the run is finished, `WriteTEXSequenceByGenerator` is called to record the output sequence in compact version. `WriteTEXRunFSR` returns the sequence generated by this run.

`WriteTEXNonlinRunFSR` TO DO

`WriteTEXRunFSRByGenerator` is a *.tex version of `WriteTEXRunFSR` but instead of using the currently set basis of the *fsr*, the table entries are printed as powers of a chosen generator *gen*. Generator *gen* is used to get the exponents of the elements, and the elements themselfs are printed as $*strGen*^{exponent}$, where *strGen* must be a string representing a greek letter in *.tex, for example *strGen* "alpha" will give α.

3.2.5 WriteTEXElementTableByGenerator

▷ `WriteTEXElementTableByGenerator(output, F, B, strGen, gen)` (function)

`WriteTEXElementTableByGenerator` provides the context information for `WriteTEXSequenceByGenerator` and `WriteTEXRunFSRByGenerator`. Its output is a *.tex file with a table containing the elements of *F* represented in basis *B* and their representation as powers of a chosen generator *gen*, printed as $*strGen*^{exponent}$, where by the greek letter passed to the function as a string *strGen*. There is an extra table column containing the order of each element.

The output file contains additional information: defining polynomial of *F*, basis elements of *B* as powers of generator *gen*, and information whether or not *gen* is a root of the defining polynomial.

3.3 TEX drawing functions

Chapter 4

misc - helper functions

4.1 Output formatting functions

There are two types of functions: ones that return the input in a human friendly version (as strings or list of strings), and ones that write the human friendly version of the input into a file (txt or tex)

4.1.1 IntFFExt

▷ `IntFFExt([B, ]ffe)` (method)
▷ `IntVecFFExt([B, ]vec)` (method)
▷ `IntMatFFExt([B, ]M)` (method)

`IntFFExt` takes the *ffe* and writes it as an integer of the prime field if *ffe* is an element of the prime field (same as Int(ffe)), or writes it as a vector of integers from the prime subfield if *ffe* is an element of an extension field, using the given basis *B* or canonical basis representation of *ffe* if no basis is provided.

`IntVecFFExt` takes the vector *vec* of FFEs and writes it in a human friendly version: as a vector of integers from the prime field if all components of *vec* belong to a prime field, or as a vector of vectors of integers from the prime subfield, if the components belong to an extension field, using the given basis *B* or canonical basis representation of *ffe*, if no basis is provided. (note: all components are treated as elements of the largest field).

`IntMatFFExt` takes a matrix *M* and returns its human friendly version: a matrix of vectors of integers from the prime field if all components of *M* belong to a prime field, or a vector of row vectors, whose elements are vectors of integers from the prime subfield, if the components belong to an extension field, using the given basis *B* or canonical basis representation of components of *M*.

NOTE: the non-basis versions return a representation in the smallest field that contains the element. For representation in a specific field, use the basis version with desired basis.

4.1.2 VecToString

▷ `VecToString([B, ]vec)` (method)

Writes a FFE verctor or matrix as string or list of strings using the given basis *B* or canonical basis representation of *ffe* if no basis is provided. This mathod calls methods `IntFFExt`, `IntVecFFExt` and `IntMatFFExt` from section LINK. The list of strings is more practically useful: we wish to have the components as srings, therefore the human friendly version of a matrix is not an actual string.

NOTE: the non-basis versions return a representation in the cononical basis of the smallest field that contains the element. For representation in a specific field, use the basis version with desired basis.

4.1.3 WriteVector (for a FFE and given basis)

▷ `WriteVector(output, B, vec)` (function)

Writes the human friendly version of vector *vec* represented in basis *B*, to the output file *output*. Also works if *vec* is an integer or FFE. can be used to write the sequence produced by the FSR to a file, make sure that the sequence does not contain any subsequences (ie if merging two runs of the FSR, must use Append(seq,seq1), if adding new step to a run must use Add(seq, elm1))

NOTE: the basis MUST be provided.

Also works for writing matrices, but writes them as a row vector, not as a rectangle.

4.1.4 WriteMatrix (for a matrix of FFE and given basis)

▷ `WriteMatrix(output, B, M)` (function)

Writes the human friendly version of matrix *M* represented in basis *B* to the output file *output* nicely formatted (rectangular, each row in a new line).

NOTE: the basis MUST be provided.

4.1.5 WriteMatrixTEX

▷ `WriteMatrixTEX(output, M)` (function)

Writes the TEX code for matrix *M* over a prime field to the output file *output*.

NOTE: Only works for matrices over a prime field !!!

4.2 misc - helper functions

4.2.1 MonomialsOverField (for an NLFSR)

▷ `MonomialsOverField(F, poly)` (method)

MonomialsOverField reduces takes a monomial or a list of monomials, and reduces all the exponents modulo (Size(F)-1) for all extension fields and prime fields except for F=$(F)_2$. For $(F)_2$ all the exponents are set to 1.

4.2.2 DegreeOfPolynomial (DegreeOfPolynomial)

▷ `DegreeOfPolynomial(F, poly)` (method)

`DegreeOfPolynomial` as follows for both monomial of form $p = \prod x_i^{e_i}$ and polynomial of form $P = \sum c_j \cdot_j$ where $p_j = \prod_i x_i^{e_i}$ DegreeOfPolynomial for a monomial: $= \sum e_i$, where i runs through all indeterminates present in this monomial

DegreeOfPolynomial for a polynomial:$=$ $\max(DegreeOfPolynomial(p_j))$, where max runs through all monomials p_j present in this polynomial so an actual extra funstion called DegreeOf-Monomial is not needed

4.2.3 GeneratorOfUnderlyingField

▷ `GeneratorOfUnderlyingField(`F`)` (method)

`GeneratorOfUnderlyingField` returns the first element $\ni$: $order(x) = Size(F) - 1$

References

Index