

FSR

...

Version 1.0.4

17 January 2017

Nusa Zidaric

Nusa Zidaric Email: [email](#)

Homepage: [http://](#)

Abstract

The GAP package FSR ...

Copyright

© 2017-2017 by Nusa Zidaric, Mark Aagaard, Guang Gong

FSR is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version. For details, see the FSF's own site <http://www.gnu.org/licenses/gpl.html>.

If you obtained FSR, we would be grateful for a short notification sent to one of the authors.

If you publish a result which was partially obtained with the usage of FSR, please cite it in the following form:

N. Zidaric. ...

Acknowledgements

...

Contents

1	Preface	4
2	Output formatting functions and TEX drawing functions	5
2.1	Output formatting functions	5
2.2	TEX drawing functions	6
3	FSR (Feedback Shift Register)	7
3.1	Common functionality	7
3.2	LFSR specific functionality	9
3.3	NLFSR specific functionality	11
4	misc - helper functions	14
4.1	misc - helper functions	14
	References	15
	Index	16

Chapter 1

Preface

The *GAP* package FSR implements Feedback Shift Registers

Chapter 2

Output formatting functions and TEX drawing functions

2.1 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)

2.1.1 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.

2.1.2 VecToString

- ▷ VecToString($[B,]vec$) (method)

Writes a FFE vector or matrix as string or list of strings using the given basis B or canonical basis representation of \mathbb{F}_p if no basis is provided. This method calls methods `IntFFExt`, `IntVecFFExt` and `IntMatFFExt` from section LINK. The list of strings is more practically useful: we wish to have the components as strings, therefore the human friendly version of a matrix is not an actual string.

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

2.1.3 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.

NOTE: the basis MUST be provided.

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

2.1.4 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.

2.1.5 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 !!!

2.2 TEX drawing functions

Chapter 3

FSR (Feedback Shift Register)

3.1 Common functionality

We define an object FSR (Feedback Shift Register), which can come in two flavours: with linear feedback LFSR (3.2.1) and nonlinear feedback NLFSR (3.3.2). 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.

3.1.1 IsFSR

▷ IsFSR (filter)

This is the category of FSR objects. Objects in this category are created using functions LFSR (3.2.1) or NLFSR (3.3.2).

3.1.2 FieldPoly (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 store 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.

3.1.3 Length (for an FSR)

- ▷ `Length(fsr)` (attribute)
- ▷ `InternalStateSize(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 * width)

3.1.4 LoadFSR (for an FSR)

- ▷ `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.

3.1.5 StepFSR (for an FSR)

- ▷ `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*. Element *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 optional nonzero *elm* as *nonlinear step*. Similarly, the NLFBSR can also have an extra element added to the (already nonlinear) feedback.

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

3.1.6 RunFSR (for an FSR)

- ▷ `RunFSR(fsr[, ist][, num][, pr])` (method)

The *fsr* will be run for a certain (*num* or *threshold*) number of steps: there is a threshold value, currently set to $2^{\text{Length}(\text{fsr})} + \text{Length}(\text{fsr})$, which is used by all versions without explicit *num* and enforced when *num* exceeds *threshold*. 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*).

- `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])` - load *fsr* with *ist*, then 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[, num, 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: for the load and run versions, element *seq*₀ is a part of the output sequence The output of `RunFSR` is:

- sequence of FFEs : *seq*₀, *seq*₁, *seq*₂, ..., for $Length(OutputTap)=1$
- sequence of vectors, each of them with *t* FFEs : *seq*₀, *seq*₁, *seq*₂, ..., where *seq*_{*i*} = (*seq*_{*i*1}, ..., *seq*_{*i**t*}) for $Length(OutputTap)=t$

3.2 LFSR specific functionality

3.2.1 LFSR

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

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

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

Inputs:

- *F* - the underlying finite field (either an extension field or a prime field)
- *charpol* - LFSR defining polynomial
- *fieldpol* - defining polynomial of the extension field (must be irreducible)
- *p* - characteristic
- *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.

Components:

- *init* - FFE vector of length $n=\deg(charpol)$, storing the initial state of the LFSR, with indices from *n*-1, ..., 0
- *state* - FFE vector of length $n=\deg(charpol)$, storing the current state of the LFSR, 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` (??) and incremented by 1 with each step (using `StepFSR` (??)))

Attributes `FieldPoly` (??), `UnderlyingField` (??), `CharPoly`, `FeedbackVec` (??), `Length` (??) and `OutputTap` (??) 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 polynomial), an error message appears and the function returns *fail*.

3.2.2 IsLinearFeedback (for an LFSR)

- ▷ IsLinearFeedback(*lfsr*) (property)
- ▷ IsLFSR(*lfsr*) (filter)

If we were to represent the *lfsr* 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.

3.2.3 CharPoly (for an LFSR)

- ▷ CharPoly(*lfsr*) (attribute)

Attribute holding the characteristic polynomial (the feedback polynomial).

3.2.4 IsPeriodic (for an LFSR)

- ▷ 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 *lfsr*.

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:

3.2.5 ViewObj (for an LFSR)

- ▷ ViewObj(*[B,]lfsr*) (method)
- ▷ PrintObj(*[B,]lfsr*) (method)
- ▷ PrintAll(*[B,]lfsr*) (method)

Different detail on the *lfsr* created by LFSR (3.2.1):

- **Display/View:** show the CharPoly and wheter or not the *lsfr* is empty
- **Print:** same as Display/View if *lsfr* is empty, otherwise it also shows the values of the three components *init*, *state* and *numsteps*
- **PrintAll:** same as Print if *lsfr* is empty, otherwise it also shows the values of the three components *init*, *state* and *numsteps* with additional information about the underlying field and the tap positions

Can be used with optional parameter basis *B* for desiered output format.

3.3 NLFSR specific functionality

3.3.1 ChooseField (for a given field)

▷ ChooseField(*F*) (function)

Workaround for the UNKNOWNEntity(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.

3.3.2 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 object, main difference is in creation of the underlying finite field.

Inputs:

- *F* - the underlying finite field (either an extension field or a prime field)
- *fieldpol* - defining polynomial of the extension field (must be irreducible) TO DO
- *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 $n=\deg(\text{charpol})$, storing the initial state of the NLFSR, with indeces from $n-1, \dots, 0$
- `state` - FFE vector of length $n=\deg(\text{charpol})$, storing the current state of the NLFSR, with indeces from $n-1, \dots, 0$
- `numsteps` - the number of steps performed thus far (initialized to -1 when created, set to 0 when loaded using `LoadFSR(??)` and incremented by 1 with each step (using `StepFSR(??)`))

Attributes `FieldPoly(??)`, `UnderlyingFied(??)`, `MultivarPoly`, `FeedbackVec(??)`, `IndetList(??)`, `Length(??)` and `OutputTap(??)` 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 polynomial), an error message appears and the function returns `fail`.

3.3.3 IsNonLinearFeedback (for an NLFSR)

- ▷ `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).

3.3.4 MultivarPoly (for an NLFSR)

- ▷ `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`.

3.3.5 ViewObj (for an NLFSR)

- ▷ `ViewObj([B,]nlfsr)` (method)
- ▷ `PrintObj([B,]nlfsr)` (method)
- ▷ `PrintAll([B,]nlfsr)` (method)

Different detail on `nlfsr` created by NLFSR (3.3.2):

- `Display/View`: show the `MultivarPoly` and wheter or not the `nlfsr` is empty
- `Print`: same as `Display/View` if `nlfsr` is empty, otherwise it also shows the values of the three components `init`, `state` and `numsteps`

- `PrintAll`: same as `Print` if `nlfsr` is empty, otherwise it also shows the values of the three components `init`, `state` and `numsteps` with additional information about the underlying field and the tap positions

Can be used with optional parameter basis B for desired output format.

Chapter 4

misc - helper functions

4.1 misc - helper functions

4.1.1 MonomialsOverField (for an NLFSR)

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

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

4.1.2 DegreeOfPolynomial (DegreeOfPolynomial)

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

`DegreeOfPolynomial` as follows for both monomial of form $p = \prod x_i^{e_i}$ and polynomial of form $P = \sum c_j \cdot x^j$ where $p_j = \prod x_i^{e_i}$ `DegreeOfPolynomial` for a monomial: $= \sum e_i$, where i runs through all indeterminates present in this monomial `DegreeOfPolynomial` for a polynomial: $= \max(\text{DegreeOfPolynomial}(p_j))$, where \max runs through all monomials p_j present in this polynomial so an actual extra function called `DegreeOfMonomial` is not needed

References

Index

- IntFFExt, [5](#)
- IntMatFFExt, [5](#)
- IntVecFFExt, [5](#)
- CharPoly
 - for an LFSR, [10](#)
- ChooseField
 - for a given field, [11](#)
- DegreeOfPolynomial
 - DegreeOfPolynomial, [14](#)
- FeedbackVec
 - FeedbackVec, [7](#)
- FieldPoly
 - FieldPoly, [7](#)
- fsr, [7](#)
- FSR package, [2](#)
- IndetList
 - for an NLFSR, [12](#)
- InternalStateSize
 - for an FSR, [8](#)
- IsFSR, [7](#)
- IsLFSR
 - for an LFSR, [10](#)
- IsLinearFeedback
 - for an LFSR, [10](#)
- IsMaxSeqLFSR
 - for an LFSR, [10](#)
- IsNLFSR
 - for an NLFSR, [12](#)
- IsNonLinearFeedback
 - for an NLFSR, [12](#)
- IsPeriodic
 - for an LFSR, [10](#)
- IsUltPeriodic
 - for an LFSR, [10](#)
- Length
 - for an FSR, [8](#)
- LFSR, [9](#)
- LoadFSR
 - for an FSR, [8](#)
- MonomialsOverField
 - for an NLFSR, [14](#)
- MultivarPoly
 - for an NLFSR, [12](#)
- NLFSR, [11](#)
- outputs, [5](#)
- OutputTap
 - OutputTap, [7](#)
- Period
 - for an LFSR, [10](#)
- PeriodIrreducible
 - for an LFSR, [10](#)
- PeriodReducible
 - for an LFSR, [10](#)
- PrintAll
 - for an LFSR, [10](#)
 - for an NLFSR, [12](#)
- PrintObj
 - for an LFSR, [10](#)
 - for an NLFSR, [12](#)
- RunFSR
 - for an FSR, [8](#)
- StepFSR
 - for an FSR, [8](#)
- UnderlyingField
 - UnderlyingField, [7](#)
- VecToString, [5](#)
- ViewObj
 - for an LFSR, [10](#)

for an NLFSR, [12](#)

`WriteMatrix`

for a matrix of FFE and given basis, [6](#)

`WriteMatrixTEX`, [6](#)

`WriteVector`

for a FFE and given basis, [6](#)