Encoder_demo

```
from sage.coding.Gabidulin_code import *
F.<t> = GF(4)
Fm.<tt> = GF(4^3)
n=3
k=2
q = F.order()
p = F.characteristic()
Frob = Fm.frobenius_endomorphism(log(q,p))
L.<x>=Fm['x',Frob]
C=GabidulinCode(F,Fm,L,n,k)
E1=C.encoder("GeneratorEncoder")
E2=C.encoder("EvaluationEncoder")
D=C.decoder("GabidulinGao")
```

```
C?
```

File: /home/musab/SageMath/local/lib/python2.7/site-packages/sage/coding/Gabidulin_code.py

Type: <class 'sage.coding.Gabidulin_code.GabidulinCode_with_category'>

Definition: C(m)

Docstring:

Class for Gabidulin codes $Gab[n, k]$.

INPUT:

- `ground_field` – A finite field $\mathbf{F}_q$ of a prime power order q.
- `extension_field` – A finite field $\mathbf{F}_{q^m}$ which is an extension field of degree m of $\mathbf{F}_q$.
- `length` – The length of the Gabidulin Code, i.e., length (n) should be less than or equal to (m).
- `dimension` – The dimesnion of the Gabidulin Code, i.e., dimension (k) should be less than or equal to the length (n).

EXAMPLES:

```
sage: load("/home/maeahmed/Gabidulin sage/Gabidulin_code.py")
sage: F.<t> = GF(4)
sage: Fm.<tt> = GF(4^3)
sage: n=3
sage: k=2
sage: q = F.order()
sage: p = F.characteristic()
sage: Frob = Fm.frobenius_endomorphism(log(q,p))
sage: L.<x>=Fm['x',Frob]
sage: C=GabidulinCode(F,Fm,L,n,k)
sage: C
Gabidulin Code Gab[3,2] over Finite Field in tt of size 2^6
```

```
C.length?
```

File: /home/musab/SageMath/local/lib/python2.7/site-packages/sage/coding/Gabidulin_code.py

Type: <type 'instancemethod'>

Definition: C.length()

Docstring:

Returns the length of `self`.

EXAMPLES:

```
sage: load("/home/maeahmed/Gabidulin sage/Gabidulin_code.py")
sage: F.<t> = GF(4)
sage: Fm.<tt> = GF(4^3)
sage: n=3
sage: k=2
sage: q = F.order()
sage: p = F.characteristic()
sage: Frob = Fm.frobenius_endomorphism(log(q,p))
sage: L.<x>=Fm['x',Frob]
sage: C=GabidulinCode(F,Fm,L,n,k)
sage: C.length()

3
```

```
C.dimension?
```

File: /home/musab/SageMath/local/lib/python2.7/site-packages/sage/coding/Gabidulin_code.py

Type: <type 'instancemethod'>

Definition: C.dimension()

Docstring:

Returns the dimension of `self`.

EXAMPLES:

```
sage: load("/home/maeahmed/Gabidulin sage/Gabidulin_code.py")
sage: F.<t> = GF(4)
sage: Fm.<tt> = GF(4^3)
sage: n=3
sage: k=2
sage: q = F.order()
sage: p = F.characteristic()
sage: Frob = Fm.frobenius_endomorphism(log(q,p))
sage: L.<x>=Fm['x',Frob]
sage: C=GabidulinCode(F,Fm,L,n,k)
sage: C.dimension()

2
```

```
C.minimum_distance?
```

File: /home/musab/SageMath/local/lib/python2.7/site-packages/sage/coding/Gabidulin_code.py

Type: <type 'instancemethod'>

Definition: C.minimum_distance()

Docstring:

Returns the minimum distance of `self`.

Minimum distance,

$$d = n - k + 1$$

EXAMPLES:

```
sage: load("/home/maeahmed/Gabidulin sage/Gabidulin_code.py")
sage: F.<t> = GF(4)
sage: Fm.<tt> = GF(4^3)
sage: n=3
sage: k=2
sage: q = F.order()
sage: p = F.characteristic()
sage: Frob = Fm.frobenius_endomorphism(log(q,p))
sage: L.<x>=Fm['x',Frob]
sage: C=GabidulinCode(F,Fm,L,n,k)
```

```
sage: C.minimum_distance()
2
```

```
C.linearized_poly_ring?
```

File: /home/musab/SageMath/local/lib/python2.7/site-packages/sage/coding/Gabidulin_code.py

Type: <type 'instancemethod'>

Definition: C.linearized_poly_ring()

Docstring:

Returns the linearized polynomials ring of `self` over $\mathbf{F}_{q^m}$ denoted by $\mathcal{L}_{q^m}[x]$.

EXAMPLES:

```
sage: load("/home/maeahmed/Gabidulin sage/Gabidulin_code.py")
sage: F.<t> = GF(4)
sage: Fm.<tt> = GF(4^3)
sage: n=3
sage: k=2
sage: q = F.order()
sage: p = F.characteristic()
sage: Frob = Fm.frobenius_endomorphism(log(q,p))
sage: L.<x>=Fm['x',Frob]
sage: C=GabidulinCode(F,Fm,L,n,k)
sage: C.linearized_poly_ring()
Skew Polynomial Ring in x over Finite Field in tt of size 2^6 twisted by
tt |--> tt^(2^2)
```

```
C.cardinality?
```

File: /home/musab/SageMath/local/lib/python2.7/site-packages/sage/coding/Gabidulin_code.py

Type: <type 'instancemethod'>

Definition: C.cardinality()

Docstring:

Returns the cardinality of `self`.

The code cardinality is the number of codewords in the code.

EXAMPLES:

```
sage: load("/home/maeahmed/Gabidulin sage/Gabidulin_code.py")
sage: F.<t> = GF(4)
sage: Fm.<tt> = GF(4^3)
sage: n=3
sage: k=2
sage: q = F.order()
sage: p = F.characteristic()
sage: Frob = Fm.frobenius_endomorphism(log(q,p))
sage: L.<x>=Fm['x',Frob]
sage: C=GabidulinCode(F,Fm,L,n,k)
sage: C.cardinality()
4096
```

```
C.ground_field?
```

File: /home/musab/SageMath/local/lib/python2.7/site-packages/sage/coding/Gabidulin_code.py

Type: <type 'instancemethod'>

Definition: C.ground_field()

Docstring:

Returns the ground field $\mathbf{F}_q$ of `self`.

EXAMPLES:

```
sage: load("/home/maeahmed/Gabidulin sage/Gabidulin_code.py")
sage: F.<t> = GF(4)
sage: Fm.<tt> = GF(4^3)
sage: n=3
sage: k=2
sage: q = F.order()
sage: p = F.characteristic()
sage: Frob = Fm.frobenius_endomorphism(log(q,p))
sage: L.<x>=Fm['x',Frob]
sage: C=GabidulinCode(F,Fm,L,n,k)
sage: C.ground_field()
Finite Field in t of size 2^2
```

```
C.extension_field?
```

File: /home/musab/SageMath/local/lib/python2.7/site-packages/sage/coding/Gabidulin_code.py

Type: <type ‘instancemethod’>

Definition: C.extension_field()

Docstring:

Returns the extension field $\mathbf{F}_{q^m}$ of `self`.

EXAMPLES:

```
sage: load("/home/maeahmed/Gabidulin sage/Gabidulin_code.py")
sage: F.<t> = GF(4)
sage: Fm.<tt> = GF(4^3)
sage: n=3
sage: k=2
sage: q = F.order()
sage: p = F.characteristic()
sage: Frob = Fm.frobenius_endomorphism(log(q,p))
sage: L.<x>=Fm['x',Frob]
sage: C=GabidulinCode(F,Fm,L,n,k)
sage: C.extension_field()
Finite Field in tt of size 2^6
```

```
C.extension_degree?
```

File: /home/musab/SageMath/local/lib/python2.7/site-packages/sage/coding/Gabidulin_code.py

Type: <type ‘instancemethod’>

Definition: C.extension_degree()

Docstring:

Returns the extension degree (m) of an extension field $\mathbf{F}_{q^m}$ over $\mathbf{F}_q$ of `self`.

EXAMPLES:

```
sage: load("/home/maeahmed/Gabidulin sage/Gabidulin_code.py")
sage: F.<t> = GF(4)
sage: Fm.<tt> = GF(4^3)
sage: n=3
sage: k=2
sage: q = F.order()
sage: p = F.characteristic()
sage: Frob = Fm.frobenius_endomorphism(log(q,p))
sage: L.<x>=Fm['x',Frob]
sage: C=GabidulinCode(F,Fm,L,n,k)
```

```
sage: C.extension_degree()
3
```

```
C.rank_weight?
```

File: /home/musab/SageMath/local/lib/python2.7/site-packages/sage/coding/Gabidulin_code.py

Type: <type 'instancemethod'>

Definition: C.rank_weight(codeword_in_vector_repr)

Docstring:

Returns the rank weight of a vector over $\mathbf{F}_{q^m}$ of `self`.

Let $x \in \mathbf{F}_{q^m}^n$ a word of length n which could be spanned to a matrix $A \in \mathbf{F}_q^{m \times n}$. Then, the Rank Weight denoted by $wt_R(x)$ is the rank of the matrix A, that is:

$$wt_R(x) = Rk(x) = Rk(A)$$

EXAMPLES:

```
sage: load("/home/maeahmed/Gabidulin sage/Gabidulin_code.py")
sage: F.<t> = GF(4)
sage: Fm.<tt> = GF(4^3)
sage: n=3
sage: k=2
sage: q = F.order()
sage: p = F.characteristic()
sage: Frob = Fm.frobenius_endomorphism(log(q,p))
sage: L.<x>=Fm['x',Frob]
sage: C=GabidulinCode(F,Fm,L,n,k)
sage: eval = C.evaluation_points()
sage: C.rank_weight(eval)

3
```

```
C.rank_distance?
```

File: /home/musab/SageMath/local/lib/python2.7/site-packages/sage/coding/Gabidulin_code.py

Type: <type 'instancemethod'>

Definition: C.rank_distance(codeword_in_vector_repr1, codeword_in_vector_repr2)

Docstring:

Returns the rank distance between two vectors over $\mathbf{F}_{q^m}$ of `self`.

Let $x1, x2 \in \mathbf{F}_{q^m}^n$ be two words of length n and $A1, A2 \in \mathbf{F}_q^{m \times n}$ be the matrix representations respectively. Then, the rank distance denoted by $d_R(x1, x2)$ is the rank of the difference between these two matrices, that is:

$$d_R(x1, x2) = Rk(x1 - x2) = Rk(A1 - A2)$$

EXAMPLES:

```
sage: load("/home/maeahmed/Gabidulin sage/Gabidulin_code.py")
sage: F.<t> = GF(4)
sage: Fm.<tt> = GF(4^3)
sage: n=3
sage: k=2
sage: q = F.order()
sage: p = F.characteristic()
sage: Frob = Fm.frobenius_endomorphism(log(q,p))
sage: L.<x>=Fm['x',Frob]
sage: C=GabidulinCode(F,Fm,L,n,k)
sage: eval = C.evaluation_points()
sage: C.rank_distance(eval,[0*tt,0*tt,0*tt])
```

```
3
```

```
C.polynomial_basis?
```

File: /home/musab/SageMath/local/lib/python2.7/site-packages/sage/coding/Gabidulin_code.py

Type: <type 'instancemethod'>

Definition: C.polynomial_basis()

Docstring:

Returns the polynomial(power) basis of an extension_field $\mathbf{F}_{q^m}$ over $\mathbf{F}_q$ of `self`.

A polynomial basis is a basis of the form $(1, \alpha^1, \alpha^2, \cdots, \alpha^{m-1})$

EXAMPLES:

```
sage: load("/home/maeahmed/Gabidulin sage/Gabidulin_code.py")
sage: F.<t> = GF(4)
sage: Fm.<tt> = GF(4^3)
sage: n=3
sage: k=2
sage: q = F.order()
sage: p = F.characteristic()
sage: Frob = Fm.frobenius_endomorphism(log(q,p))
sage: L.<x>=Fm['x',Frob]
sage: C=GabidulinCode(F,Fm,L,n,k)
sage: C.polynomial_basis()
(1, tt, tt^2)
```

```
C.normal_basis?
```

File: sage/coding/Gabidulin_code.py

Type: <type 'sage.misc.cachefunc.CachedMethodCallerNoArgs'>

Definition: C.normal_basis()

Docstring:

Returns a normal basis of an extension_field $\mathbf{F}_{q^m}$ over $\mathbf{F}_q$ of `self`.

A normal basis is a basis of the form $(\alpha, \alpha^q, \alpha^{q^2}, \cdots, \alpha^{q^{m-1}})$, where (α) is a normal element in $\mathbf{F}_{q^m}$.

EXAMPLES:

```
sage: load("/home/maeahmed/Gabidulin sage/Gabidulin_code.py")
sage: F.<t> = GF(4)
sage: Fm.<tt> = GF(4^3)
sage: n=3
sage: k=2
sage: q = F.order()
sage: p = F.characteristic()
sage: Frob = Fm.frobenius_endomorphism(log(q,p))
sage: L.<x>=Fm['x',Frob]
sage: C=GabidulinCode(F,Fm,L,n,k)
sage: C.normal_basis()
(tt^5 + tt^4 + 1, tt^4 + tt^2 + 1, tt^5 + tt^2 + 1)
```

```
C._is_normal_basis?
```

File: /home/musab/SageMath/local/lib/python2.7/site-packages/sage/coding/Gabidulin_code.py

Type: <type 'instancemethod'>

Definition: C._is_normal_basis(normal_basis)

Docstring:

Checks if a given basis is a normal basis of an extension_field $\mathbf{F}_{q^m}$ over $\mathbf{F}_q$ of `self`.

A normal basis is a basis of the form $(\alpha, \alpha^q, \alpha^{q^2}, \cdots, \alpha^{q^{m-1}})$, where (α) is a normal element in $\mathbf{F}_{q^m}$.

EXAMPLES:

```
sage: load("/home/maeahmed/Gabidulin sage/Gabidulin_code.py")
sage: F.<t> = GF(4)
sage: Fm.<tt> = GF(4^3)
sage: n=3
sage: k=2
sage: q = F.order()
sage: p = F.characteristic()
sage: Frob = Fm.frobenius_endomorphism(log(q,p))
sage: L.<x>=Fm['x',Frob]
sage: C=GabidulinCode(F,Fm,L,n,k)
sage: poly_basis = C.polynomial_basis()
sage: C._is_normal_basis(poly_basis)
Traceback (click to the left of this block for traceback)
...
ValueError: value of 'basis' keyword is not a normal basis
```

```
C._trace?
```

File: /home/musab/SageMath/local/lib/python2.7/site-packages/sage/coding/Gabidulin_code.py

Type: <type 'instancemethod'>

Definition: C._trace(alpha)

Docstring:

Calculates the trace of an element $\alpha \in \mathbf{F}_{q^m}$ of `self`.

EXAMPLES:

```
sage: load("/home/maeahmed/Gabidulin sage/Gabidulin_code.py")
sage: F.<t> = GF(4)
sage: Fm.<tt> = GF(4^3)
sage: n=3
sage: k=2
sage: q = F.order()
sage: p = F.characteristic()
sage: Frob = Fm.frobenius_endomorphism(log(q,p))
sage: L.<x>=Fm['x',Frob]
sage: C=GabidulinCode(F,Fm,L,n,k)
sage: C._trace(tt)
1
```

```
C.dual_basis?
```

File: sage/coding/Gabidulin_code.py

Type: <type 'sage.misc.cachefunc.CachedMethodCaller'>

Definition: C.dual_basis(basis)

Docstring:

Returns a dual-basis β of a given basis α of an extension field $\mathbf{F}_{q^m}$ over $\mathbf{F}_q$ of `self`.

A basis β is a dual of a basis α if and only if:

$$trace(\alpha_i \beta_j) = \begin{cases} 1, & for\ i = j, \\ 0, & else. \end{cases}$$

INPUT:

- `basis` – basis of $\mathbf{F}_{q^m}$ over $\mathbf{F}_q$

OUTPUT:

- `dual_basis` – the dual basis of basis.

EXAMPLES:

```
sage: load("/home/maeahmed/Gabidulin sage/Gabidulin_code.py")
sage: F.<t> = GF(4)
sage: Fm.<tt> = GF(4^3)
sage: n=3
sage: k=2
sage: q = F.order()
sage: p = F.characteristic()
sage: Frob = Fm.frobenius_endomorphism(log(q,p))
sage: L.<x>=Fm['x',Frob]
sage: C=GabidulinCode(F,Fm,L,n,k)
sage: basis=C.polynomial_basis()
sage: dual=C.dual_basis(basis);dual
(tt^5 + tt^2 + 1, tt^5 + tt^4 + tt^2 + tt, tt^4 + tt)
sage: C.dual_basis(dual)==basis
True
```

```
C._is_dual_basis?
```

File: /home/musab/SageMath/local/lib/python2.7/site-packages/sage/coding/Gabidulin_code.py

Type: <type ‘instancemethod’>

Definition: C._is_dual_basis(basis, dual_basis)

Docstring:

Checks if the given bases are dual of an extension_field $\mathbf{F}_{q^m}$ over $\mathbf{F}_q$ of `self`.

INPUT:

- `basis` – basis of $\mathbf{F}_{q^m}$ over $\mathbf{F}_q$.
- `dual_basis` – the dual basis of basis.

OUTPUT:

- raise an error if the given bases are not dual

EXAMPLES:

```
sage: load("/home/maeahmed/Gabidulin sage/Gabidulin_code.py")
sage: F.<t> = GF(4)
sage: Fm.<tt> = GF(4^3)
sage: n=3
sage: k=2
sage: q = F.order()
sage: p = F.characteristic()
sage: Frob = Fm.frobenius_endomorphism(log(q,p))
sage: L.<x>=Fm['x',Frob]
sage: C=GabidulinCode(F,Fm,L,n,k)
sage: poly_basis = C.polynomial_basis()
sage: dual_basis = C.dual_basis(poly_basis)
sage: C._is_dual_basis(poly_basis,dual_basis)
Traceback (click to the left of this block for traceback)
...
ValueError: value of 'basis' keyword and 'dual_basis' keyword are not
dual
```

```
C._normal_dual_basis_matrix?
```

File: sage/coding/Gabidulin_code.py

Type: <type 'sage.misc.cachefunc.CachedMethodCaller'>

Definition: C._normal_dual_basis_matrix(normal_basis)

Docstring:

Construct the matrices $\mathcal{B}$ and $\mathcal{B}^{\mathcal{T}}$ which are used to construct the generetor and the parity-check matrices using a given normal basis.

EXAMPLES:

```
sage: load("/home/maeahmed/Gabidulin sage/Gabidulin_code.py")
sage: F.<t> = GF(4)
sage: Fm.<tt> = GF(4^3)
sage: n=3
sage: k=2
sage: q = F.order()
sage: p = F.characteristic()
sage: Frob = Fm.frobenius_endomorphism(log(q,p))
sage: L.<x>=Fm['x',Frob]
sage: C=GabidulinCode(F,Fm,L,n,k)
sage: normal_basis = C.normal_basis()
sage: B,B_dual=C._normal_dual_basis_matrix(normal_basis)
```

```
C.evaluation_points?
```

File: /home/musab/SageMath/local/lib/python2.7/site-packages/sage/coding/Gabidulin_code.py

Type: <type 'instancemethod'>

Definition: C.evaluation_points(basis=None)

Docstring:

Returns the points of $\mathbf{F}_{q^m}$ in which the polynomials are evaluated. A basis of $\mathbf{F}_{q^m}$ over $\mathbf{F}_q$ could be given optionally in the input 'points'.

The evaluation points are fixed elements $g_0, g_1, \cdots, g_{n-1} \in \mathbf{F}_{q^m}$ that are linearly independent over $\mathbf{F}_q$, where n is the code length.

INPUT:

- `basis` – basis of $\mathbf{F}_{q^m}$ over $\mathbf{F}_q$

OUTPUT:

- `evaluation_points` – the first n points in basis.

EXAMPLES:

```
sage: load("/home/maeahmed/Gabidulin sage/Gabidulin_code.py")
sage: F.<t> = GF(4)
sage: Fm.<tt> = GF(4^3)
sage: n=3
sage: k=2
sage: q = F.order()
sage: p = F.characteristic()
sage: Frob = Fm.frobenius_endomorphism(log(q,p))
sage: L.<x>=Fm['x',Frob]
sage: C=GabidulinCode(F,Fm,L,n,k)
sage: C.evaluation_points()
[1, tt, tt^2]
sage: normal_basis = C.normal_basis();normal_basis
(tt^4 + tt^2 + tt, tt^5 + tt^4 + tt^2, tt^5 + tt + 1)
sage: C.evaluation_points(normal_basis)
[tt^4 + tt^2 + tt, tt^5 + tt^4 + tt^2, tt^5 + tt + 1]
```

```
C.map_into_ground_field?
```

File: /home/musab/SageMath/local/lib/python2.7/site-packages/sage/coding/Gabidulin_code.py

Type: <type ‘instancemethod’>

Definition: C.map_into_ground_field(vector)

Docstring:

Maps a vector $v \in \mathbf{F}_{q^m}^n$ into a matrix $A \in \mathbf{F}_q^{m \times n}$ where any element $vi \in \mathbf{F}_{q^m}$ is constituting a row in the matrix.

EXAMPLES:

```
sage: load("/home/maeahmed/Gabidulin sage/Gabidulin_code.py")
sage: F.<t> = GF(4)
sage: Fm.<tt> = GF(4^3)
sage: n=3
sage: k=2
sage: q = F.order()
sage: p = F.characteristic()
sage: Frob = Fm.frobenius_endomorphism(log(q,p))
sage: L.<x>=Fm['x',Frob]
sage: C=GabidulinCode(F,Fm,L,n,k)
sage: poly_basis = C.evaluation_points()
sage: C.map_into_ground_field(poly_basis)

[1 0 0]
[0 1 0]
[0 0 1]
```

```
C.map_into_extension_field?
```

File: /home/musab/SageMath/local/lib/python2.7/site-packages/sage/coding/Gabidulin_code.py

Type: <type ‘instancemethod’>

Definition: C.map_into_extension_field(matrix)

Docstring:

Maps a matrix $A \in \mathbf{F}_q^{m \times n}$ into a vector $v \in \mathbf{F}_{q^m}^n$.

EXAMPLES:

```
sage: load("/home/maeahmed/Gabidulin sage/Gabidulin_code.py")
sage: F.<t> = GF(4)
sage: Fm.<tt> = GF(4^3)
sage: n=3
sage: k=2
sage: q = F.order()
sage: p = F.characteristic()
sage: Frob = Fm.frobenius_endomorphism(log(q,p))
sage: L.<x>=Fm['x',Frob]
sage: C=GabidulinCode(F,Fm,L,n,k)
sage: poly_basis = C.evaluation_points()
sage: poly_basis_matrix = C.map_into_ground_field(poly_basis)
sage: C.map_into_extension_field(poly_basis_matrix) == poly_basis

True
```

```
C.linear_independency_over_ground_field?
```

File: /home/musab/SageMath/local/lib/python2.7/site-packages/sage/coding/Gabidulin_code.py

Type: <type ‘instancemethod’>

Definition: C.linear_independency_over_ground_field(basis)

Docstring:

Validates that a basis of $\mathbf{F}_{q^m}$ over $\mathbf{F}_q$ is linearly independent over $\mathbf{F}_q$.

EXAMPLES:

```
sage: load("/home/maeahmed/Gabidulin sage/Gabidulin_code.py")
sage: F.<t> = GF(4)
sage: Fm.<tt> = GF(4^3)
sage: n=3
sage: k=2
sage: q = F.order()
sage: p = F.characteristic()
sage: Frob = Fm.frobenius_endomorphism(log(q,p))
sage: L.<x>=Fm['x',Frob]
sage: C=GabidulinCode(F,Fm,L,n,k)
sage: basis = [tt,tt,tt]
sage: C.linear_independency_over_ground_field(basis)

Traceback (click to the left of this block for traceback)
...
ValueError: The elements provided are not linearly independent over
Finite Field in t of size 2^2
```

```
C.frobenius_automorphism?
```

File: /home/musab/SageMath/local/lib/python2.7/site-packages/sage/coding/Gabidulin_code.py

Type: <type ‘instancemethod’>

Definition: C.frobenius_automorphism()

Docstring:

Defines the mapping $\Phi : \mathbf{F}_{q^m} \to \mathbf{F}_{q^m}$, where $\Phi(x) = x^q$ which maps an element $x \in \mathbf{F}_{q^m}$ over $\mathbf{F}_q$ into x^q.

EXAMPLES:

```
sage: load("/home/maeahmed/Gabidulin sage/Gabidulin_code.py")
sage: F.<t> = GF(4)
sage: Fm.<tt> = GF(4^3)
sage: n=3
sage: k=2
sage: q = F.order()
sage: p = F.characteristic()
sage: Frob = Fm.frobenius_endomorphism(log(q,p))
sage: L.<x>=Fm['x',Frob]
sage: C=GabidulinCode(F,Fm,L,n,k)
sage: Frob = C.frobenius_automorphism()
sage: Frob(tt)

tt^4
```

```
C.random_linearized_poly?
```

File: /home/musab/SageMath/local/lib/python2.7/site-packages/sage/coding/Gabidulin_code.py

Type: <type ‘instancemethod’>

Definition: C.random_linearized_poly(dual_code=None)

Docstring:

Choose a random linearized polynomial with degree less than the dimension (k) from The set of all linearized polynomials over $\mathbf{F}_{q^m}$ denoted by $\mathcal{L}_{q^m}[x]$.

A *linearized polynomial* over $\mathbf{F}_{q^m}$ is a polynomial of the form:

$$f(x) = \sum_{i=0}^{d} \alpha_i x^{[i]}, \quad \alpha_i \in \mathbf{F}_{q^m}, \alpha_d \neq 0$$

In the case we want a message polynomial for the dual Gabidulin code, give the optional parameter dual_code = True

INPUT:

- `dual_code` – an optional input if the random polynomial is from the dual code of C.

OUTPUT:

- `linearized_poly` – a linearized polynomial $\in \mathcal{L}_{q^m}[x]$ of degree less than the dimension.

EXAMPLES:

```
sage: load("/home/maeahmed/Gabidulin sage/Gabidulin_code.py")
sage: F.<t> = GF(4)
sage: Fm.<tt> = GF(4^3)
sage: n=3
sage: k=2
sage: q = F.order()
sage: p = F.characteristic()
sage: Frob = Fm.frobenius_endomorphism(log(q,p))
sage: L.<x>=Fm['x',Frob]
sage: C=GabidulinCode(F,Fm,L,n,k)
sage: Frob = C.frobenius_automorphism()
sage: L.<x>=Fm['x',Frob]
sage: C.random_linearized_poly()
(tt^4 + tt^3 + tt^2 + 1)*x + 1
sage: C.random_linearized_poly(dual_code=True)
tt^3 + 1
```

```
C._right_LEEA?
```

File: /home/musab/SageMath/local/lib/python2.7/site-packages/sage/coding/Gabidulin_code.py

Type: <type ‘instancemethod’>

Definition: C._right_LEEA(a, b, d_stop)

Docstring:

Performs the right linearized extended Euclidean algorithm on the given polynomials $a(x)$ and $b(x)$ where $deg_q(a(x)) \geq deg_q(b(x))$. The algorithm have $a(x)$, $b(x)$ and the stop degree d_{stop} as inputs and $r_{out}(x)$, $v_{out}(x)$ and $u_{out}(x)$ as outputs, i.e.,

$$r_{out}(x) = v_{out}(x) \otimes a(x) + u_{out}(x) \otimes b(x),$$

where $deg_q(r_{out}) < d_{stop}$.

INPUT:

- a – a linearized polynomial $\in \mathcal{L}_{q^m}[x]$
- b – a linearized polynomial $\in \mathcal{L}_{q^m}[x]$
- `d_stop` – the stopping degree

OUTPUT:

- r – a linearized polynomial $\in \mathcal{L}_{q^m}[x]$
- u – a linearized polynomial $\in \mathcal{L}_{q^m}[x]$
- v – a linearized polynomial $\in \mathcal{L}_{q^m}[x]$

EXAMPLES:

```
sage: load("/home/maeahmed/Gabidulin sage/Gabidulin_code.py")
sage: F.<t> = GF(4)
sage: Fm.<tt> = GF(4^3)
sage: n=3
sage: k=2
sage: q = F.order()
sage: p = F.characteristic()
```

```
sage: Frob = Fm.frobenius_endomorphism(log(q,p))
sage: L.<x>=Fm['x',Frob]
sage: C=GabidulinCode(F,Fm,L,n,k)
sage: a=x^2+x; b=x^2; C._right_LEEA(a,b,2)
(x, 1, 1)
```

```
C._lagrange_interpolating_polynomial?
```

File: sage/coding/Gabidulin_code.py

Type: <type 'sage.misc.cachefunc.CachedMethodCaller'>

Definition: C._lagrange_interpolating_polynomial(polynomial_coefficients, basis=None)

Docstring:

Let $(f_0, f_1, \cdots, f_{n-1})$ be the coefficients of a linearized polynomial $f(x) \in \mathcal{L}_{q^m}[x]$ where $deg_q(f(x)) < n$. A linearized Lagrange interpolating polynomial denoted by $\hat{f}(x)$ is the polynomial that pass through n points $\{(u_0, f_0), (u_1, f_1), \cdots, (u_{n-1}, f_{n-1})\}$, such that the following holds,

$$\hat{f}(u_i) = f_i.$$

INPUT:

- `polynomial_coefficients` – coefficients of a linearized polynomial $\in \mathcal{L}_{q^m}[x]$
- `basis` – basis of $\mathbf{F}_{q^m}$ over $\mathbf{F}_q$

OUTPUT:

- `lagrange_interpolating_polynomial` – a lineairized polynomial that evaluates with the basis into polynomial_coefficients.

EXAMPLES:

```
sage: load("/home/maeahmed/Gabidulin sage/Gabidulin_code.py")
sage: F.<t> = GF(4)
sage: Fm.<tt> = GF(4^3)
sage: n=3
sage: k=2
sage: q = F.order()
sage: p = F.characteristic()
sage: Frob = Fm.frobenius_endomorphism(log(q,p))
sage: L.<x>=Fm['x',Frob]
sage: C=GabidulinCode(F,Fm,L,n,k)
sage: c=vector([1, tt, tt^2]);c
(1, tt, tt^2)
sage: basis=C.polynomial_basis(); evaluation=C.evaluation_points(basis)
sage: c_hat=C._lagrange_interpolating_polynomial(c,basis)
sage: C.evaluate_linearized_poly(c_hat,evaluation)
(1, tt, tt^2)
```

```
C.evaluate_linearized_poly?
```

File: /home/musab/SageMath/local/lib/python2.7/site-packages/sage/coding/Gabidulin_code.py

Type: <type 'instancemethod'>

Definition: C.evaluate_linearized_poly(linearized_poly=None, evaluation_points=None)

Docstring:

Given the ring of linearized polynomials $\mathcal{L}_{q^m}[x]$, evaluation points(use C.evaluation_poits() if not given) and a linearized polynomial(optional) this method evaluates the given polynomial at these points.

INPUT:

- `linearized_poly` – linearized polynomial $\in \mathcal{L}_{q^m}[x]$
- `evaluation points` – basis of $\mathbf{F}_{q^m}$ over $\mathbf{F}_q$ of length n.

OUTPUT:

- `evaluation` – the evaluation vector

EXAMPLES:

```
sage: load("/home/maeahmed/Gabidulin sage/Gabidulin_code.py")
sage: F.<t> = GF(4)
sage: Fm.<tt> = GF(4^3)
sage: n=3
sage: k=2
sage: q = F.order()
sage: p = F.characteristic()
sage: Frob = Fm.frobenius_endomorphism(log(q,p))
sage: L.<x>=Fm['x',Frob]
sage: C=GabidulinCode(F,Fm,L,n,k)
sage: c=vector([1, tt, tt^2]);c
(1, tt, tt^2)
sage: basis=C.polynomial_basis(); evaluation=C.evaluation_points(basis)
sage: c_hat=C._lagrange_interpolating_polynomial(c,basis)
sage: C.evaluate_linearized_poly(c_hat,evaluation)
(1, tt, tt^2)
```

```
C.q_power?
```

File: /home/musab/SageMath/local/lib/python2.7/site-packages/sage/coding/Gabidulin_code.py

Type: <type 'instancemethod'>

Definition: C.q_power(field_element, exponent)

Docstring:

Given an element $\alpha \in \mathbf{F}_{q^m}$ and an exponent i, it outputs the q-power $\alpha^{[i]} = \alpha^{q^i}$.

INPUT:

- `field_element` – an element $\alpha \in \mathbf{F}_{q^m}$
- `exponet` – an integer number

OUTPUT:

- `result` – the i^{th} q-power of field element, where i is equal to the exponent value.

EXAMPLES:

```
sage: load("/home/maeahmed/Gabidulin sage/Gabidulin_code.py")
sage: F.<t> = GF(4)
sage: Fm.<tt> = GF(4^3)
sage: n=3
sage: k=2
sage: q = F.order()
sage: p = F.characteristic()
sage: Frob = Fm.frobenius_endomorphism(log(q,p))
sage: L.<x>=Fm['x',Frob]
sage: C=GabidulinCode(F,Fm,L,n,k)
sage: C.q_power(tt,0)

tt
```

```
C.code_space?
```

File: /home/musab/SageMath/local/lib/python2.7/site-packages/sage/coding/Gabidulin_code.py

Type: <type 'instancemethod'>

Definition: C.code_space()

Docstring:

Returns the Code vector space of *self*.
EXAMPLES:

```
sage: load("/home/maeahmed/Gabidulin sage/Gabidulin_code.py")
sage: F.<t> = GF(4)
sage: Fm.<tt> = GF(4^3)
sage: n=3
sage: k=2
sage: q = F.order()
sage: p = F.characteristic()
sage: Frob = Fm.frobenius_endomorphism(log(q,p))
sage: L.<x>=Fm['x',Frob]
sage: C=GabidulinCode(F,Fm,L,n,k)
sage: C.code_space()

Vector space of dimension 3 over Finite Field in tt of size 2^6
```

```
C.generator_matrix?
```

File: sage/coding/Gabidulin_code.py

Type: <type 'sage.misc.cachefunc.CachedMethodCaller'>

Definition: C.generator_matrix(basis=None)

Docstring:

Defines the Generator matrix of a gabidulin code $Gab[n, k]$.

A *Generator matrix* of a $Gab[n, k]$ code is the $n \times k$ matrix:

$$\mathbf{G} = \begin{pmatrix} g_0^{[0]} & g_1^{[0]} & \cdots & g_{n-1}^{[0]} \\ g_0^{[1]} & g_1^{[1]} & \cdots & g_{n-1}^{[1]} \\ \vdots & \vdots & \ddots & \vdots \\ g_0^{[k-1]} & g_1^{[k-1]} & \cdots & g_{n-1}^{[k-1]} \end{pmatrix}.$$

INPUT:

- basis – basis of $\mathbf{F}_{q^m}$ over $\mathbf{F}_q$

OUTPUT:

- G – Generator matrix of $Gab[n, k]$

EXAMPLES:

```
sage: load("/home/maeahmed/Gabidulin sage/Gabidulin_code.py")
sage: F.<t> = GF(4)
sage: Fm.<tt> = GF(4^3)
sage: n=3
sage: k=2
sage: q = F.order()
sage: p = F.characteristic()
sage: Frob = Fm.frobenius_endomorphism(log(q,p))
sage: L.<x>=Fm['x',Frob]
sage: C=GabidulinCode(F,Fm,L,n,k)
sage: C.generator_matrix()

[                          1                          tt                        tt^2]
[                          1                        tt^4 tt^5 + tt^4 + tt^2 + tt + 1]
sage: normal_basis = C.normal_basis()
sage: C.generator_matrix(normal_basis)
[tt^5 + tt^2 + tt             tt^5    tt^2 + tt + 1]
[            tt^5    tt^2 + tt + 1 tt^5 + tt^2 + tt]
```

```
C.parity_check_matrix?
```

File: sage/coding/Gabidulin_code.py

Type: <type ‘sage.misc.cachefunc.CachedMethodCaller’>

Definition: C.parity_check_matrix(basis=None)

Docstring:

Defines the Parity-Check matrix of a gabidulin code $Gab[n, k]$.

Let $\mathbf{G}$ be a generator matrix of a $Gab[n, k]$ and $g_0, g_1, \cdots, g_{n-1} \in \mathbf{F}_{q^m}$ are linearly independent over $\mathbf{F}_q$. A *Parity-Check matrix* denoted by $\mathbf{H}$ where $\mathbf{G} \cdot \mathbf{H}^T = \mathbf{0}$ is the $n \times k$ matrix:

$$\mathbf{H} = \begin{pmatrix} h_0^{[0]} & h_1^{[0]} & \cdots & h_{n-1}^{[0]} \\ h_0^{[1]} & h_1^{[1]} & \cdots & h_{n-1}^{[1]} \\ \vdots & \vdots & \ddots & \vdots \\ h_0^{[n-k-1]} & h_1^{[n-k-1]} & \cdots & h_{n-1}^{[n-k-1]} \end{pmatrix},$$

where $h_0, h_1, \cdots, h_{n-1}$ are non-zero solution of the equations:

$$\sum_{i=0}^{n-1} g_i^{[j]} h_i \qquad \forall j \in [-n+k+1, k-1].$$

INPUT:

- `basis` – basis of $\mathbf{F}_{q^m}$ over $\mathbf{F}_q$

OUTPUT:

- H – parity check matrix of $Gab[n, k]$

EXAMPLES:

```
sage: load("/home/maeahmed/Gabidulin sage/Gabidulin_code.py")
sage: F.<t> = GF(4)
sage: Fm.<tt> = GF(4^3)
sage: n=3
sage: k=2
sage: q = F.order()
sage: p = F.characteristic()
sage: Frob = Fm.frobenius_endomorphism(log(q,p))
sage: L.<x>=Fm['x',Frob]
sage: C=GabidulinCode(F,Fm,L,n,k)
sage: C.parity_check_matrix()

[                     1 tt^5 + tt^4 + tt^2 + 1            tt^3 + tt^2]
sage: normal_basis = C.normal_basis()
sage: C.parity_check_matrix(normal_basis)
[tt^5 + tt^2 + tt             tt^5    tt^2 + tt + 1]
```

```
C.dual_code?
```

File: sage/coding/Gabidulin_code.py

Type: <type ‘sage.misc.cachefunc.CachedMethodCallerNoArgs’>

Definition: C.dual_code()

Docstring:

Defines the dual code of the given Gabidulin code of `self`.

Let $Gab[n, k]$ be a Gabidulin code and $\mathrm{c} \in Gab[n, k]$ be any codeword in the code. The dual code $Gab[n, k]^\perp$ is the Gabidulin code $Gab[n, n-k]$ that is defined by,

$$Gab[n,k]^{\perp} = \{c^{\perp} \in \mathbf{F}_{q^m}^n | \langle c^{\perp}, c\rangle = 0, \quad \forall c \in Gab[n,k]\}.$$

OUTPUT:

- `C_dual` – the dual code $Gab[n, n-k]$

EXAMPLES:

```
sage: load("/home/maeahmed/Gabidulin sage/Gabidulin_code.py")
sage: F.<t> = GF(4)
sage: Fm.<tt> = GF(4^3)
sage: n=3
sage: k=2
sage: q = F.order()
sage: p = F.characteristic()
sage: Frob = Fm.frobenius_endomorphism(log(q,p))
sage: L.<x>=Fm['x',Frob]
sage: C=GabidulinCode(F,Fm,L,n,k)
sage: C
Gabidulin Code Gab[3,2] over Finite Field in tt of size 2^6
sage: C.dual_code()
Gabidulin Code Gab[3,1] over Finite Field in tt of size 2^6
```

```
E1
```

```
Generator matrix based encoder for Gabidulin Code Gab[3,2] over
Finite Field in tt of size 2^6
```

```
E1?
```

File: /home/musab/SageMath/local/lib/python2.7/site-packages/sage/coding/Gabidulin_code.py

Type: <class ‘sage.coding.Gabidulin_code.GabidulinCodeGeneratorMatrixEncoder’>

Definition: E1(m)

Docstring:

Defines the encoding of a Gabidulin code $Gab[n,k]$ using the generator matrix and an information vector.

Let $f(x)$ be a linearized polynomial with degree less than the dimension (k) from the set of all linearized polynomials over $\mathbf{F}_{q^m}$ denoted by $\mathcal{L}_{q^m}[x]$ and f be a vector represents the coefficients of $f(x)$. The encoding of a $Gab[n,k]$ using a generator matrix $\mathbf{G}$:

$$Gab[n,k] = \{\mathrm{c} \in \mathbf{F}_{q^m}^n | \quad \mathrm{c} = \mathrm{f} \cdot \mathrm{G}, \quad \forall \mathrm{f} \in \mathbf{F}_{q^m}^k\}.$$

Such that, an information vector f is mapped into a codeword $\mathrm{c} = \mathrm{f} \cdot \mathrm{G}$ using the following mapping:

$$\mathrm{f} \mapsto \mathrm{f} \cdot \mathrm{G}.$$

EXAMPLES:

```
sage: load("/home/maeahmed/Gabidulin sage/Gabidulin_code.py")
sage: F.<t> = GF(4)
sage: Fm.<tt> = GF(4^3)
sage: n=3
sage: k=2
sage: q = F.order()
sage: p = F.characteristic()
sage: Frob = Fm.frobenius_endomorphism(log(q,p))
sage: L.<x>=Fm['x',Frob]
sage: C=GabidulinCode(F,Fm,L,n,k)
sage: p=C.random_linearized_poly()
sage: G = C.generator_matrix()
sage: E1=C.encoder("GeneratorEncoder")
sage: c1=E1.encode(G,p);c1

(tt^5 + tt^3 + tt + 1, tt^5 + tt^3 + tt^2 + tt + 1, tt^4)
```

```
E2
```

```
Polynomial evaluation based encoder for Gabidulin Code Gab[3,2] over
Finite Field in tt of size 2^6
```

```
E2?
```

File: /home/musab/SageMath/local/lib/python2.7/site-packages/sage/coding/Gabidulin_code.py

Type: <class ‘sage.coding.Gabidulin_code.GabidulinCodePolynomialEvaluationEncoder’>

Definition: E2(m)

Docstring:

Defines the encoding of a Gabidulin code $Gab[n, k]$ using evaluation of a linearized polynomial at fixed points.

An $[n, k]$ Gabidulin $Gab[n, k]$ is a linear MRD code that consists of all words (vectors) of the form $(f(g_0), f(g_1), \cdots, f(g_{n-1}))$, where the fixed elements $g_0, g_1, \cdots, g_{n-1} \in \mathbf{F}_{q^m}$ are linearly independent over $\mathbf{F}_q$ (referred to as the evaluation points) and $f(x)$ include all linearized polynomials over $\mathbf{F}_{q^m}$ of degree less than k:

$$Gab[n, k] = \{(f(g_0), f(g_1), \cdots, f(g_{n-1})) \mid \quad deg_q(f(x)) < k\}.$$

EXAMPLES:

```
sage: load("/home/maeahmed/Gabidulin sage/Gabidulin_code.py")
sage: F.<t> = GF(4)
sage: Fm.<tt> = GF(4^3)
sage: n=3
sage: k=2
sage: q = F.order()
sage: p = F.characteristic()
sage: Frob = Fm.frobenius_endomorphism(log(q,p))
sage: L.<x>=Fm['x',Frob]
sage: C=GabidulinCode(F,Fm,L,n,k)
sage: p=C.random_linearized_poly()
sage: basis=C.polynomial_basis()
sage: E2=C.encoder("EvaluationEncoder")
sage: c2=E2.encode(basis,p);c2

(tt^5 + tt^3 + tt + 1, tt^5 + tt^3 + tt^2 + tt + 1, tt^4)
```

```
E2._is_codeword?
```

File: /home/musab/SageMath/local/lib/python2.7/site-packages/sage/coding/Gabidulin_code.py

Type: <type ‘instancemethod’>

Definition: E2._is_codeword(codeword, basis=None)

Docstring:

Return True if the given codeword is a valid codeword of `self` code.

INPUT:

- `codeword` – codeword vector $\in Gab[n, k]$
- `basis` – basis of $\mathbf{F}_{q^m}$ over $\mathbf{F}_q$

OUTPUT:

- `True` – if the given codeword $\in Gab[n, k]$ is a valid codeword.

EXAMPLES:

```
sage: load("/home/maeahmed/Gabidulin sage/Gabidulin_code.py")
sage: F.<t> = GF(4)
sage: Fm.<tt> = GF(4^3)
sage: n=3
sage: k=2
sage: q = F.order()
sage: p = F.characteristic()
sage: Frob = Fm.frobenius_endomorphism(log(q,p))
```

```
sage: L.<x>=Fm['x',Frob]
sage: C=GabidulinCode(F,Fm,L,n,k)
sage: p=C.random_linearized_poly()
sage: basis=C.polynomial_basis()
sage: E2=C.encoder("EvaluationEncoder")
sage: c2=E2.encode(basis,p);c2
sage: E2._is_codeword(c2)
True
```

```
E2.encode?
```

File: /home/musab/SageMath/local/lib/python2.7/site-packages/sage/coding/Gabidulin_code.py

Type: <type 'instancemethod'>

Definition: E2.encode(linearized_poly, basis=None)

Docstring:

Return a codeword of `self` code.

INPUT:

- `linearized_poly` – linearized polynomial $\in \mathcal{L}_{q^m}[x]$
- `basis` – basis of $\mathbf{F}_{q^m}$ over $\mathbf{F}_q$

OUTPUT:

- `codeword` – encoded codeword $\in Gab[n, k]$

EXAMPLES:

```
sage: load("/home/maeahmed/Gabidulin sage/Gabidulin_code.py")
sage: F.<t> = GF(4)
sage: Fm.<tt> = GF(4^3)
sage: n=3
sage: k=2
sage: q = F.order()
sage: p = F.characteristic()
sage: Frob = Fm.frobenius_endomorphism(log(q,p))
sage: L.<x>=Fm['x',Frob]
sage: C=GabidulinCode(F,Fm,L,n,k)
sage: p=C.random_linearized_poly()
sage: basis=C.polynomial_basis()
sage: E2=C.encoder("EvaluationEncoder")
sage: c2=E2.encode(basis,p);c2
(tt^5 + tt^3 + tt + 1, tt^5 + tt^3 + tt^2 + tt + 1, tt^4)
```

```
E2.unencode_nocheck?
```

File: /home/musab/SageMath/local/lib/python2.7/site-packages/sage/coding/Gabidulin_code.py

Type: <type 'instancemethod'>

Definition: E2.unencode_nocheck(codeword, basis=None)

Docstring:

Return the message polynomial of the given codeword.

This method does not check if the given codeword is a valid codeword.

INPUT:

- `codeword` – codeword vector $\in \mathbf{F}_{q^m}^n$
- `basis` – basis of $\mathbf{F}_{q^m}$ over $\mathbf{F}_q$

OUTPUT:

- p – linearized polynomial $p \in \mathcal{L}_{q^m}[x]$

EXAMPLES:

```
sage: load("/home/maeahmed/Gabidulin sage/Gabidulin_code.py")
sage: F.<t> = GF(4)
sage: Fm.<tt> = GF(4^3)
sage: n=3
sage: k=2
sage: q = F.order()
sage: p = F.characteristic()
sage: Frob = Fm.frobenius_endomorphism(log(q,p))
sage: L.<x>=Fm['x',Frob]
sage: C=GabidulinCode(F,Fm,L,n,k)
sage: p=C.random_linearized_poly()
sage: basis=C.polynomial_basis()
sage: E2=C.encoder("EvaluationEncoder")
sage: c2=E2.encode(basis,p);
sage: p_estimated=E2.unencode_nocheck(c2,basis)
sage: p_estimated==p
True
```

```
E2.message_space?
```

File: /home/musab/SageMath/local/lib/python2.7/site-packages/sage/coding/Gabidulin_code.py

Type: <type ‘instancemethod’>

Definition: E2.message_space()

Docstring:

Return the message space of `self`

OUTPUT:

- L – the linearized polynomial ring $\mathcal{L}_{q^m}[x]$

EXAMPLES:

```
sage: load("/home/maeahmed/Gabidulin sage/Gabidulin_code.py")
sage: F.<t> = GF(4)
sage: Fm.<tt> = GF(4^3)
sage: n=3
sage: k=2
sage: q = F.order()
sage: p = F.characteristic()
sage: Frob = Fm.frobenius_endomorphism(log(q,p))
sage: L.<x>=Fm['x',Frob]
sage: C=GabidulinCode(F,Fm,L,n,k)
sage: E2=C.encoder("EvaluationEncoder")
sage: E2.message_space()
Skew Polynomial Ring in x over Finite Field in tt of size 2^6 twisted by
tt |--> tt^(2^2)
```

```
D
```

```
Gao-like decoder for Gabidulin Code Gab[3,2] over Finite Field in tt
of size 2^6
```

```
D?
```

File: /home/musab/SageMath/local/lib/python2.7/site-packages/sage/coding/Gabidulin_code.py

Type: <class ‘sage.coding.Gabidulin_code.GabidulinCodeGaoDecoder’>

Definition: D([noargspec])

Docstring:

The Gao-like Gabidulin decoder is a (transformed) key-equation-based algorithm that directly gives the decoding result in one step, as analogous to Gao’s decoder for Reed-Solomon codes.

In order to decode a codeword the Gao-like algorithm uses the following key equation,

$$\Lambda(x) \otimes f(x) = \Omega(x) \otimes M_{\mathcal{G}}(x) + \Lambda(x) \otimes \hat{r}(x).$$

where $\mathcal{G} = \{g_0, g_1, \cdots, g_{n-1}\}$ is a basis over $\mathbf{F}_q$ that is used as evaluation points of $Gab[n, k]$, $r(x)$ is the received word polynmial such that $\hat{r}(x)$ is its transformed polynomial, $M_{\mathcal{G}}(x)$ is the minimal subspace polynomial, $\Omega(x) \in \mathcal{L}_{q^m}[x]$ is a linearized polynomial over $\mathbf{F}_{q^m}$, $\Lambda(x)$ is the error span polynomial and $f(x)$ is the message polynmial. The degree constraints is as follows, $deg_q(M_{\mathcal{G}}(x)) = n$, $deg_q(\hat{r}(x)) < n$, $deg_q(\Lambda(x)) = t \leq \lfloor (d-1)/2 \rfloor = \lfloor (n-k)/2 \rfloor$ and $deg_q(f(x)) < k$.

INPUT:

- `code` – the Gabidulin code that will be decoded.

EXAMPLES:

```
sage: load("/home/maeahmed/Gabidulin sage/Gabidulin_code.py")
sage: F.<t> = GF(4)
sage: Fm.<tt> = GF(4^3)
sage: n=3
sage: k=2
sage: q = F.order()
sage: p = F.characteristic()
sage: Frob = Fm.frobenius_endomorphism(log(q,p))
sage: L.<x>=Fm['x',Frob]
sage: C=GabidulinCode(F,Fm,L,n,k)
sage: D=C.decoder("GabidulinGao")
sage: D
Gao-like decoder for the Gabidulin Code Gab[3,2] over Finite Field
in tt of size 2^6
```

```
D._minimal_subspace_polynomial?
```

File: sage/coding/Gabidulin_code.py

Type: <type 'sage.misc.cachefunc.CachedMethodCaller'>

Definition: D._minimal_subspace_polynomial(basis=None)

Docstring:

Return the minimal subspace polynomial $M_{\mathcal{U}}(x)$ of the given basis $\mathcal{U} = (u_0, u_1, \cdots, u_{n-1}) \in \mathbf{F}_{q^m}$.

A minimal subspace polynomial is the linearized polynomial with least degree such that it evaluates to zero at all basis elements, i.e.,

$$M_{\mathcal{U}}(x) = \prod_{i=1}^{n-1} (x - u_i).$$

INPUT:

- `basis` – basis of $\mathbf{F}_{q^m}$ over $\mathbf{F}_q$

OUTPUT:

- `M` – Minimal subspace polynomial of the given basis.

EXAMPLES:

```
sage: load("/home/maeahmed/Gabidulin sage/Gabidulin_code.py")
sage: F.<t> = GF(4)
sage: Fm.<tt> = GF(4^3)
sage: n=3
sage: k=2
sage: q = F.order()
sage: p = F.characteristic()
sage: Frob = Fm.frobenius_endomorphism(log(q,p))
sage: L.<x>=Fm['x',Frob]
sage: C=GabidulinCode(F,Fm,L,n,k)
sage: D=C.decoder("GabidulinGao")
sage: D._minimal_subspace_polynomial()
x^3 + 1
```

```
D.decode_to_code?
```

File: sage/coding/Gabidulin_code.py

Type: <type 'sage.misc.cachefunc.CachedMethodCaller'>

Definition: D.decode_to_code(received_word, basis=None)

Docstring:

Decode a received word into a codeword.

This decoder find the unique error word $\mathrm{e} = \mathrm{r} - \mathrm{c}$ such that $wt_{rk}(\mathrm{e}) = t \leq \frac{d-1}{2}$ where r is the received word and c is the transmitted codeword.

INPUT:

- `received_word` – the received word $\in \mathbf{F}_{q^m}^n$
- `basis` – basis of $\mathbf{F}_{q^m}$ over $\mathbf{F}_q$

OUTPUT:

- `estimated_codeword` – estimeted codeword $\in Gab[n, k]$

EXAMPLES:

```
sage: load("/home/maeahmed/Gabidulin sage/Gabidulin_code.py")
sage: F.<t> = GF(4)
sage: Fm.<tt> = GF(4^4)
sage: n=4
sage: k=2
```

```
sage: q = F.order()
sage: p = F.characteristic()
sage: Frob = Fm.frobenius_endomorphism(log(q,p))
sage: L.<x>=Fm['x',Frob]
sage: C=GabidulinCode(F,Fm,L,n,k)
sage: E=C.encoder("EvaluationEncoder")
sage: D=C.decoder("GabidulinGao")
sage: t = (1,D.decoding_radius())
sage: V = C.code_space();
sage: Chan = channels.StaticErrorRateChannel(V, t)
sage: message_polynomial = x^1+tt;message_polynomial
x + tt
sage: basis = C.polynomial_basis()
sage: codeword=E.encode(message_polynomial,basis)
sage: received_word=Chan(codeword)
sage: estimated_codeword = D.decode_to_code(received_word,basis
sage: estimated_codeword == codeword
True
```

```
D.decode_to_message?
```

File: /home/musab/SageMath/local/lib/python2.7/site-packages/sage/coding/Gabidulin_code.py

Type: <type ‘instancemethod’>

Definition: D.decode_to_message(received_word, basis=None)

Docstring:

Decode a received word into a message polynomial

INPUT:

- `received_word` – the received word $\in \mathbf{F}_{q^m}^n$

OUTPUT:

- p – estimeted message polynomial $\in \mathcal{L}_{q^m}[x]$

EXAMPLES:

```
sage: load("/home/maeahmed/Gabidulin sage/Gabidulin_code.py")
sage: F.<t> = GF(4)
sage: Fm.<tt> = GF(4^4)
sage: n=4
sage: k=2
sage: q = F.order()
sage: p = F.characteristic()
sage: Frob = Fm.frobenius_endomorphism(log(q,p))
sage: L.<x>=Fm['x',Frob]
sage: C=GabidulinCode(F,Fm,L,n,k)
sage: E=C.encoder("EvaluationEncoder")
sage: D=C.decoder("GabidulinGao")
sage: t = (1,D.decoding_radius())
sage: V = C.code_space();
sage: Chan = channels.StaticErrorRateChannel(V, t)
sage: message_polynomial = x^1+tt;message_polynomial
x + tt
sage: basis = C.polynomial_basis()
sage: codeword=E.encode(message_polynomial,basis)
sage: received_word=Chan(codeword)
sage: estimated_message = D.decode_to_message(received_word,basis)
sage: estimated_message == message_polynomial
True
```

```
D.decoding_radius?
```

File: /home/musab/SageMath/local/lib/python2.7/site-packages/sage/coding/Gabidulin_code.py

Type: <type 'instancemethod'>

Definition: D.decoding_radius()

Docstring:

Return the decoding radius of the decoder,

OUTPUT:

- `t_max` – maximum number of guranteed decodable errors $= \lfloor (n-k)/2 \rfloor$

EXAMPLES:

```
sage: load("/home/maeahmed/Gabidulin sage/Gabidulin_code.py")
sage: F.<t> = GF(4)
sage: Fm.<tt> = GF(4^4)
sage: n=4
sage: k=2
sage: q = F.order()
sage: p = F.characteristic()
sage: Frob = Fm.frobenius_endomorphism(log(q,p))
sage: L.<x>=Fm['x',Frob]
sage: C=GabidulinCode(F,Fm,L,n,k)
sage: E=C.encoder("EvaluationEncoder")
sage: D=C.decoder("GabidulinGao")
sage: D.decoding_radius()
1
```