Sage Reference Manual: Coding Theory

Release 7.4

The Sage Development Team

Oct 20, 2016

CHAPTER

ONE

ABSTRACT CLASSES, CATALOGS AND DATABASES

1.1 Decoder

Representation of an error-correction algorithm for a code.

AUTHORS:

- David Joyner (2009-02-01): initial version
- David Lucas (2015-06-29): abstract class version

class sage.coding.decoder. **Decoder** (*code*, *input_space*, *connected_encoder_name*)
Bases: sage.structure.sage_object.SageObject

Abstract top-class for *Decoder* objects.

Every decoder class should inherit from this abstract class.

To implement an decoder, you need to:

•inherit from *Decoder*

•call Decoder.__init__ in the subclass constructor. Example: super(SubclassName,self).__init__(code,input_space,connected_encoder_name) . By doing that, your subclass will have all the parameters described above initialized.

•Then, you need to override one of decoding methods, either *decode_to_code()* or *decode_to_message()* . You can also override the optional method *decoding_radius()* .

•By default, comparison of *Decoder* (using methods __eq__ and __ne__) are by memory reference: if you build the same decoder twice, they will be different. If you need something more clever, override __eq__ and __ne__ in your subclass.

•As *Decoder* is not designed to be instantiated, it does not have any representation methods. You should implement _repr_ and _latex_ methods in the subclass.

code ( )
Returns the code for this *Decoder* .

EXAMPLES:

```
sage: G = Matrix(GF(2),␣
↪[[1,1,1,0,0,0,0],[1,0,0,1,1,0,0],[0,1,0,1,0,1,0],[1,1,0,1,0,0,1]])
sage: C = LinearCode(G)
sage: D = C.decoder()
sage: D.code()
Linear code of length 7, dimension 4 over Finite Field of size 2
```

connected_encoder ( )

Returns the connected encoder of `self` .

EXAMPLES:

```
sage: G = Matrix(GF(2),␣
↪[[1,1,1,0,0,0,0],[1,0,0,1,1,0,0],[0,1,0,1,0,1,0],[1,1,0,1,0,0,1]])
sage: C = LinearCode(G)
sage: D = C.decoder()
sage: D.connected_encoder()
Generator matrix-based encoder for Linear code of length 7, dimension 4 over␣
↪Finite Field of size 2
```

decode_to_code (*r*)

Corrects the errors in `r` and returns a codeword.

This is a default implementation which assumes that the method *decode_to_message()* has been implemented, else it returns an exception.

INPUT:

- `r` – a element of the input space of `self` .

OUTPUT:

- a vector of *code()* .

EXAMPLES:

```
sage: G = Matrix(GF(2),␣
↪[[1,1,1,0,0,0,0],[1,0,0,1,1,0,0],[0,1,0,1,0,1,0],[1,1,0,1,0,0,1]])
sage: C = LinearCode(G)
sage: word = vector(GF(2), (1, 1, 0, 0, 1, 1, 0))
sage: word in C
True
sage: w_err = word + vector(GF(2), (1, 0, 0, 0, 0, 0, 0))
sage: w_err in C
False
sage: D = C.decoder()
sage: D.decode_to_code(w_err)
(1, 1, 0, 0, 1, 1, 0)
```

decode_to_message (*r*)

Decodes `r` to the message space of meth:$connected_encoder$.

This is a default implementation, which assumes that the method *decode_to_code()* has been implemented, else it returns an exception.

INPUT:

- `r` – a element of the input space of `self` .

OUTPUT:

- a vector of *message_space()* .

EXAMPLES:

```
sage: G = Matrix(GF(2),␣
↪[[1,1,1,0,0,0,0],[1,0,0,1,1,0,0],[0,1,0,1,0,1,0],[1,1,0,1,0,0,1]])
sage: C = LinearCode(G)
sage: word = vector(GF(2), (1, 1, 0, 0, 1, 1, 0))
sage: w_err = word + vector(GF(2), (1, 0, 0, 0, 0, 0, 0))
```

```
sage: D = C.decoder()
sage: D.decode_to_message(w_err)
(0, 1, 1, 0)
```

classmethod decoder_type ( )

Returns the set of types of `self` . These types describe the nature of `self` and its decoding algorithm.

This method can be called on both an uninstantiated decoder class, or on an instance of a decoder class.

EXAMPLES:

We call it on a class:

```
sage: codes.decoders.LinearCodeSyndromeDecoder.decoder_type()
{'dynamic', 'hard-decision', 'unique'}
```

We can also call it on a instance of a Decoder class:

```
sage: G = Matrix(GF(2), [[1, 0, 0, 1], [0, 1, 1, 1]])
sage: C = LinearCode(G)
sage: D = C.decoder()
sage: D.decoder_type()
{'complete', 'hard-decision', 'might-error', 'unique'}
```

decoding_radius (***kwargs*)

Returns the maximal number of errors that `self` is able to correct.

This is an abstract method and it should be implemented in subclasses.

EXAMPLES:

```
sage: G = Matrix(GF(2),␣
↪[[1,1,1,0,0,0,0],[1,0,0,1,1,0,0],[0,1,0,1,0,1,0],[1,1,0,1,0,0,1]])
sage: C = LinearCode(G)
sage: D = codes.decoders.LinearCodeSyndromeDecoder(C)
sage: D.decoding_radius()
1
```

input_space ( )

Returns the input space of `self` .

EXAMPLES:

```
sage: G = Matrix(GF(2),␣
↪[[1,1,1,0,0,0,0],[1,0,0,1,1,0,0],[0,1,0,1,0,1,0],[1,1,0,1,0,0,1]])
sage: C = LinearCode(G)
sage: D = C.decoder()
sage: D.input_space()
Vector space of dimension 7 over Finite Field of size 2
```

message_space ( )

Returns the message space of `self` 's *connected_encoder()* .

EXAMPLES:

```
sage: G = Matrix(GF(2),␣
↪[[1,1,1,0,0,0,0],[1,0,0,1,1,0,0],[0,1,0,1,0,1,0],[1,1,0,1,0,0,1]])
sage: C = LinearCode(G)
sage: D = C.decoder()
```

```
sage: D.message_space()
Vector space of dimension 4 over Finite Field of size 2
```

exception sage.coding.decoder. **DecodingError**
Bases: exceptions.Exception

Special exception class to indicate an error during decoding.

1.2 Encoder

Representation of a bijection between a message space and a code.

class sage.coding.encoder. **Encoder** (*code*)
Bases: sage.structure.sage_object.SageObject

Abstract top-class for *Encoder* objects.

Every encoder class should inherit from this abstract class.

To implement an encoder, you need to:

- inherit from *Encoder* ,
- call Encoder.__init__ in the subclass constructor. Example: super(SubclassName,self).__init__(code) . By doing that, your subclass will have its code parameter initialized.
- Then, if the message space is a vector space, default implementations of *encode()* and *unencode_nocheck()* methods are provided. These implementations rely on *generator_matrix()* which you need to override to use the default implementations.
- If the message space is not of the form F^k, where F is a finite field, you cannot have a generator matrix. In that case, you need to override *encode()* , *unencode_nocheck()* and *message_space()* .
- By default, comparison of *Encoder* (using methods __eq__ and __ne__) are by memory reference: if you build the same encoder twice, they will be different. If you need something more clever, override __eq__ and __ne__ in your subclass.
- As *Encoder* is not designed to be instantiated, it does not have any representation methods. You should implement _repr_ and _latex_ methods in the subclass.

REFERENCES:

code ( )
Returns the code for this *Encoder* .

EXAMPLES:

```
sage: G = Matrix(GF(2),␣
↪[[1,1,1,0,0,0,0],[1,0,0,1,1,0,0],[0,1,0,1,0,1,0],[1,1,0,1,0,0,1]])
sage: C = LinearCode(G)
sage: E = C.encoder()
sage: E.code() == C
True
```

encode (*word*)
Transforms an element of the message space into a codeword.

This is a default implementation which assumes that the message space of the encoder is F^k, where F is *sage.coding.linear_code.AbstractLinearCode.base_field()* and k is

sage.coding.linear_code.AbstractLinearCode.dimension() . If this is not the case, this method should be overwritten by the subclass.

Note: *encode()* might be a partial function over `self` 's *message_space()* . One should use the exception *EncodingError* to catch attempts to encode words that are outside of the message space.

INPUT:

- `word` – a vector of the message space of the `self` .

OUTPUT:

- a vector of *code()* .

EXAMPLES:

```
sage: G = Matrix(GF(2),␣
↪[[1,1,1,0,0,0,0],[1,0,0,1,1,0,0],[0,1,0,1,0,1,0],[1,1,0,1,0,0,1]])
sage: C = LinearCode(G)
sage: word = vector(GF(2), (0, 1, 1, 0))
sage: E = codes.encoders.LinearCodeGeneratorMatrixEncoder(C)
sage: E.encode(word)
(1, 1, 0, 0, 1, 1, 0)
```

If `word` is not in the message space of `self` , it will return an exception:

```
sage: word = random_vector(GF(7), 4)
sage: E.encode(word)
Traceback (most recent call last):
...
ValueError: The value to encode must be in Vector space of dimension 4 over␣
↪Finite Field of size 2
```

generator_matrix ( )

Returns a generator matrix of the associated code of `self` .

This is an abstract method and it should be implemented separately. Reimplementing this for each subclass of *Encoder* is not mandatory (as a generator matrix only makes sense when the message space is of the F^k, where F is the base field of *code()* .)

EXAMPLES:

```
sage: G = Matrix(GF(2),␣
↪[[1,1,1,0,0,0,0],[1,0,0,1,1,0,0],[0,1,0,1,0,1,0],[1,1,0,1,0,0,1]])
sage: C = LinearCode(G)
sage: E = C.encoder()
sage: E.generator_matrix()
[1 1 1 0 0 0 0]
[1 0 0 1 1 0 0]
[0 1 0 1 0 1 0]
[1 1 0 1 0 0 1]
```

message_space ( )

Returns the ambient space of allowed input to *encode()* . Note that *encode()* is possibly a partial function over the ambient space.

EXAMPLES:

```
sage: G = Matrix(GF(2),␣
↪[[1,1,1,0,0,0,0],[1,0,0,1,1,0,0],[0,1,0,1,0,1,0],[1,1,0,1,0,0,1]])
sage: C = LinearCode(G)
sage: E = C.encoder()
sage: E.message_space()
Vector space of dimension 4 over Finite Field of size 2
```

unencode (*c*, *nocheck=False*)

Returns the message corresponding to the codeword `c` .

This is the inverse of *encode()* .

INPUT:

- `c` – a codeword of *code()* .
- `nocheck` – (default: `False`) checks if `c` is in *code()* . You might set this to `True` to disable the check for saving computation. Note that if `c` is not in `self()` and `nocheck = True` , then the output of *unencode()* is not defined (except that it will be in the message space of `self`).

OUTPUT:

- an element of the message space of `self`

EXAMPLES:

```
sage: G = Matrix(GF(2),␣
↪[[1,1,1,0,0,0,0],[1,0,0,1,1,0,0],[0,1,0,1,0,1,0],[1,1,0,1,0,0,1]])
sage: C = LinearCode(G)
sage: c = vector(GF(2), (1, 1, 0, 0, 1, 1, 0))
sage: c in C
True
sage: E = codes.encoders.LinearCodeGeneratorMatrixEncoder(C)
sage: E.unencode(c)
(0, 1, 1, 0)
```

TESTS:

If `nocheck` is set to `False` , and one provides a word which is not in *code()* , *unencode()* will return an error:

```
sage: c = vector(GF(2), (0, 1, 0, 0, 1, 1, 0))
sage: c in C
False
sage: E.unencode(c, False)
Traceback (most recent call last):
...
EncodingError: Given word is not in the code
```

Note that since ticket :trac: 21326, codes cannot be of length zero:

```
sage: G = Matrix(GF(17), [])
sage: C = LinearCode(G)
Traceback (most recent call last):
...
ValueError: length must be a non-zero positive integer
```

unencode_nocheck (*c*)

Returns the message corresponding to `c` .

When `c` is not a codeword, the output is unspecified.

AUTHORS:

This function is taken from codinglib *[Nielsen]*

INPUT:

- `c` – a codeword of `code()`.

OUTPUT:

- an element of the message space of `self`.

EXAMPLES:

```
sage: G = Matrix(GF(2),␣
↪[[1,1,1,0,0,0,0],[1,0,0,1,1,0,0],[0,1,0,1,0,1,0],[1,1,0,1,0,0,1]])
sage: C = LinearCode(G)
sage: c = vector(GF(2), (1, 1, 0, 0, 1, 1, 0))
sage: c in C
True
sage: E = codes.encoders.LinearCodeGeneratorMatrixEncoder(C)
sage: E.unencode_nocheck(c)
(0, 1, 1, 0)
```

Taking a vector that does not belong to `C` will not raise an error but probably just give a non-sensical result:

```
sage: c = vector(GF(2), (1, 1, 0, 0, 1, 1, 1))
sage: c in C
False
sage: E = codes.encoders.LinearCodeGeneratorMatrixEncoder(C)
sage: E.unencode_nocheck(c)
(0, 1, 1, 0)
sage: m = vector(GF(2), (0, 1, 1, 0))
sage: c1 = E.encode(m)
sage: c == c1
False
```

exception `sage.coding.encoder.` **`EncodingError`**

Bases: `exceptions.Exception`

Special exception class to indicate an error during encoding or unencoding.

1.3 Index of bounds

The `codes.bounds` object may be used to access the bounds that Sage can compute.

codesize_upper_bound()	This computes the minimum value of the upper bound using the methods of Singleton, Hamming, Plotkin, and Elias.
dimension_upper_bound()	Returns an upper bound $B(n, d) = B_q(n, d)$ for the dimension of a linear code of length n, minimum distance d over a field of size q. Parameter "algorithm" has the same meaning as in codesize_upper_bound()
elias_bound_asymp()	Computes the asymptotic Elias bound for the information rate, provided $0 < \delta < 1 - 1/q$.
elias_upper_bound()	Returns the Elias upper bound for number of elements in the largest code of minimum distance d in $\mathbf{F}_q^n$. Wraps GAP's UpperBoundElias.
entropy()	Computes the entropy at x on the q-ary symmetric channel.
gilbert_lower_bound()	Returns lower bound for number of elements in the largest code of minimum distance d in $\mathbf{F}_q^n$.
griesmer_upper_bound()	Returns the Griesmer upper bound for number of elements in the largest code of minimum distance d in $\mathbf{F}_q^n$. Wraps GAP's UpperBoundGriesmer.
gv_bound_asymp()	Computes the asymptotic GV bound for the information rate, R.
gv_info_rate()	GV lower bound for information rate of a q-ary code of length n minimum distance delta*n
hamming_bound_asymp()	Computes the asymptotic Hamming bound for the information rate.
hamming_upper_bound()	Returns the Hamming upper bound for number of elements in the largest code of minimum distance d in $\mathbf{F}_q^n$. Wraps GAP's UpperBoundHamming.
mrrw1_bound_asymp()	Computes the first asymptotic McEliese-Rumsey-Rodemich-Welsh bound for the information rate, provided $0 < \delta < 1 - 1/q$.
plotkin_bound_asymp()	Computes the asymptotic Plotkin bound for the information rate, provided $0 < \delta < 1 - 1/q$.
plotkin_upper_bound()	Returns Plotkin upper bound for number of elements in the largest code of minimum distance d in $\mathbf{F}_q^n$.
singleton_bound_asymp()	Computes the asymptotic Singleton bound for the information rate.
singleton_upper_bound()	Returns the Singleton upper bound for number of elements in the largest code of minimum distance d in $\mathbf{F}_q^n$. Wraps GAP's UpperBoundSingleton.
volume_hamming()	Returns number of elements in a Hamming ball of radius r in $\mathbf{F}_q^n$. Agrees with Guava's SphereContent(n,r,GF(q)).

Note: To import these names into the global namespace, use:

sage: from sage.coding.bounds_catalog import *

1.4 Index of channels

The `channels` object may be used to access the codes that Sage can build.

- channel_constructions.ErrorErasureChannel
- channel_constructions.StaticErrorRateChannel
- channel_constructions.QarySymmetricChannel

Note: To import these names into the global namespace, use:

sage: from sage.coding.channels_catalog import *

1.5 Index of codes

The `codes` object may be used to access the codes that Sage can build.

BCHCode()	A 'Bose-Chaudhuri-Hockenghem code' (or BCH code for short) is the largest possible cyclic code of length n over field F=GF(q), whose generator polynomial has zeros (which contain the set) $Z = \{a^b, a^{b+1}, ..., a^{b+delta-2}\}$, where a is a primitive n^{th} root of unity in the splitting field $GF(q^m)$, b is an integer $0 \leq b \leq n - delta + 1$ and m is the multiplicative order of q modulo n. (The integers $b, ..., b + delta - 2$ typically lie in the range $1, ..., n - 1$.) The integer $delta \geq 1$ is called the "designed distance". The length n of the code and the size q of the base field must be relatively prime. The generator polynomial is equal to the least common multiple of the minimal polynomials of the elements of the set Z above.
BinaryGolayCode()	BinaryGolayCode() returns a binary Golay code. This is a perfect [23,12,7] code. It is also (equivalent to) a cyclic code, with generator polynomial $g(x) = 1 + x^2 + x^4 + x^5 + x^6 + x^{10} + x^{11}$. Extending it yields the extended Golay code (see ExtendedBinaryGolayCode).
CyclicCode()	If g is a polynomial over GF(q) which divides $x^n - 1$ then this constructs the code "generated by g" (ie, the code associated with the principle ideal gR in the ring $R = GF(q)[x]/(x^n - 1)$ in the usual way).
CyclicCodeFromCheckPolynomial()	If h is a polynomial over GF(q) which divides $x^n - 1$ then this constructs the code "generated by $g = (x^n - 1)/h$" (ie, the code associated with the principle ideal gR in the ring $R = GF(q)[x]/(x^n - 1)$ in the usual way). The option "ignore" says to ignore the condition that the characteristic of the base field does not divide the length (the usual assumption in the theory of cyclic codes).
DuadicCodeEvenPair()	Constructs the "even pair" of duadic codes associated to the "splitting" (see the docstring for `_is_a_splitting` for the definition) S1, S2 of n.
DuadicCodeOddPair()	Constructs the "odd pair" of duadic codes associated to the "splitting" S1, S2 of n.
ExtendedBinaryGolayCode()	ExtendedBinaryGolayCode() returns the extended binary Golay code. This is a perfect [24,12,8] code. This code is self-dual.
ExtendedQuadraticResidueCode()	The extended quadratic residue code (or XQR code) is obtained from a QR code by adding a check bit to the last coordinate. (These codes have very remarkable properties such as large automorphism groups and duality properties - see *[HP]*, Section 6.6.3-6.6.4.)
ExtendedTernaryGolayCode()	ExtendedTernaryGolayCode returns a ternary Golay code. This is a self-dual perfect [12,6,6] code.
QuadraticResidueCode(q)	A quadratic residue code (or QR code) is a cyclic code whose generator polynomial is the product of the polynomials $x - \alpha^i$ (α is a primitive n^{th} root of unity; i ranges over the set of quadratic residues modulo n).
QuadraticResidueCodeEvenPair(n, q)	Quadratic residue codes of a given odd prime length and base ring either don't exist at all or occur as 4-tuples - a pair of "odd-like" codes and a pair of "even-like" codes. If $n > 2$ is prime then (Theorem 6.6.2 in *[HP]*) a QR code exists over $GF(q)$ iff q is a quadratic residue mod n.
QuadraticResidueCodeOddPair(n, q)	Quadratic residue codes of a given odd prime length and base ring either don't exist at all or occur as 4-tuples - a pair of "odd-like" codes and a pair of "even-like" codes. If n 2 is prime then (Theorem 6.6.2 in *[HP]*) a QR code exists over GF(q) iff q is a quadratic residue mod n.
QuasiQuadraticResidueCode(p)	A (binary) quasi-quadratic residue code (or QQR code), as defined by Proposition 2.2 in *[BM]*, has a generator matrix in the block form $G = (Q, N)$. Here Q is a $p \times p$ circulant matrix whose top row is $(0, x_1, ..., x_{p-1})$, where $x_i = 1$ if and only if i is a quadratic residue $\mod p$, and N is a $p \times p$ circulant matrix whose top row is $(0, y_1, ..., y_{p-1})$, where $x_i + y_i = 1$ for all i.
RandomLinearCode()	Deprecated alias of `random_linear_code()` .
RandomLinearCodeGuruswami()	The method used is to first construct a $k \times n$ matrix of the block form (I, A), where I is a $k \times k$ identity matrix and A is a $k \times (n - k)$ matrix constructed using random elements of F. Then the columns are permuted using a randomly selected element of the symmetric group S_n.
ReedMullerCode()	Returns a Reed-Muller code.
ReedSolomonCode()	NO DOCSTRING
TernaryGolayCode()	TernaryGolayCode returns a ternary Golay code. This is a perfect [11,6,5] code. It is

Note: To import these names into the global namespace, use:

sage: from sage.coding.codes_catalog import *

1.6 Index of decoders

The `codes.decoders` object may be used to access the decoders that Sage can build.

Generic decoders

- *linear_code.LinearCodeSyndromeDecoder*
- *linear_code.LinearCodeNearestNeighborDecoder*

Subfield subcode decoder

- *subfield_subcode.SubfieldSubcodeOriginalCodeDecoder*

Generalized Reed-Solomon code decoders

- *grs.GRSBerlekampWelchDecoder*
- *grs.GRSErrorErasureDecoder*
- *grs.GRSGaoDecoder*
- *grs.GRSKeyEquationSyndromeDecoder*
- *guruswami_sudan.gs_decoder.GRSGuruswamiSudanDecoder*

Extended code decoders

- *extended_code.ExtendedCodeOriginalCodeDecoder*

Punctured codes decoders

- *punctured_code.PuncturedCodeOriginalCodeDecoder*

Note: To import these names into the global namespace, use:

sage: from sage.coding.decoders_catalog import *

1.7 Index of encoders

The `codes.encoders` object may be used to access the encoders that Sage can build.

Generic encoders

- *linear_code.LinearCodeGeneratorMatrixEncoder*
- *linear_code.LinearCodeParityCheckEncoder*
- *linear_code.LinearCodeSystematicEncoder*

Generalized Reed-Solomon code encoders

- *grs.GRSEvaluationVectorEncoder*
- *grs.GRSEvaluationPolynomialEncoder*

Extended code encoders

- *extended_code.ExtendedCodeExtendedMatrixEncoder*

Note: To import these names into the global namespace, use:

sage: from sage.coding.encoders_catalog import *

1.8 Databases and accessors of online databases for coding theory

sage.coding.databases. **best_linear_code_in_codetables_dot_de** (*n*, *k*, *F*, *verbose=False*)

Return the best linear code and its construction as per the web database http://codetables.de.

INPUT:

- `n` - Integer, the length of the code
- `k` - Integer, the dimension of the code
- `F` - Finite field, of order 2, 3, 4, 5, 7, 8, or 9
- `verbose` - Bool (default: `False`)

OUTPUT:

- An unparsed text explaining the construction of the code.

EXAMPLES:

```
sage: L = codes.databases.best_linear_code_in_codetables_dot_de(72, 36, GF(2))
↪# optional - internet
sage: print(L)
↪# optional - internet
Construction of a linear code
[72,36,15] over GF(2):
[1]:  [73, 36, 16] Cyclic Linear Code over GF(2)
     CyclicCode of length 73 with generating polynomial x^37 + x^36 + x^34 +
x^33 + x^32 + x^27 + x^25 + x^24 + x^22 + x^21 + x^19 + x^18 + x^15 + x^11 +
x^10 + x^8 + x^7 + x^5 + x^3 + 1
[2]:  [72, 36, 15] Linear Code over GF(2)
     Puncturing of [1] at 1

last modified: 2002-03-20
```

This function raises an `IOError` if an error occurs downloading data or parsing it. It raises a `ValueError` if the `q` input is invalid.

AUTHORS:

- Steven Sivek (2005-11-14)
- David Joyner (2008-03)

sage.coding.databases. **best_linear_code_in_guava** (*n*, *k*, *F*)

Returns the linear code of length `n` , dimension `k` over field `F` with the maximal minimum distance which is known to the GAP package GUAVA.

The function uses the tables described in `bounds_on_minimum_distance_in_guava` to construct this code. This requires the optional GAP package GUAVA.

INPUT:

- `n` – the length of the code to look up
- `k` – the dimension of the code to look up
- `F` – the base field of the code to look up

OUTPUT:

- A *LinearCode* which is a best linear code of the given parameters known to GUAVA.

EXAMPLES:

```
sage: codes.databases.best_linear_code_in_guava(10,5,GF(2))     # long time;␣
↪optional - gap_packages (Guava package)
Linear code of length 10, dimension 5 over Finite Field of size 2
sage: gap.eval("C:=BestKnownLinearCode(10,5,GF(2))")            # long time;␣
↪optional - gap_packages (Guava package)
'a linear [10,5,4]2..4 shortened code'
```

This means that the best possible binary linear code of length 10 and dimension 5 is a code with minimum distance 4 and covering radius s somewhere between 2 and 4. Use `bounds_on_minimum_distance_in_guava(10,5,GF(2))` for further details.

`sage.coding.databases.` **bounds_on_minimum_distance_in_guava** (n, k, F)

Computes a lower and upper bound on the greatest minimum distance of a $[n, k]$ linear code over the field `F` .

This function requires the optional GAP package GUAVA.

The function returns a GAP record with the two bounds and an explanation for each bound. The function Display can be used to show the explanations.

The values for the lower and upper bound are obtained from a table constructed by Cen Tjhai for GUAVA, derived from the table of Brouwer. See http://www.codetables.de/ for the most recent data. These tables contain lower and upper bounds for $q = 2$ (when `n <= 257`), $q = 3$ (when `n <= 243`), $q = 4$ (`n <= 256`). (Current as of 11 May 2006.) For codes over other fields and for larger word lengths, trivial bounds are used.

INPUT:

- `n` – the length of the code to look up
- `k` – the dimension of the code to look up
- `F` – the base field of the code to look up

OUTPUT:

- A GAP record object. See below for an example.

EXAMPLES:

```
sage: gap_rec = codes.databases.bounds_on_minimum_distance_in_guava(10,5,GF(2))
↪# optional - gap_packages (Guava package)
sage: print(gap_rec)
↪# optional - gap_packages (Guava package)
rec(
  construction :=
   [ <Operation "ShortenedCode">,
      [
          [ <Operation "UUVCode">,
              [
                  [ <Operation "DualCode">,
                      [ [ <Operation "RepetitionCode">, [ 8, 2 ] ] ] ],
```

```
                  [ <Operation "UUVCode">,
                      [
                          [ <Operation "DualCode">,
                              [ [ <Operation "RepetitionCode">, [ 4, 2 ] ] ] ]
                            , [ <Operation "RepetitionCode">, [ 4, 2 ] ] ] ]
                ] ], [ 1, 2, 3, 4, 5, 6 ] ] ],
  k := 5,
  lowerBound := 4,
  lowerBoundExplanation := ...
  n := 10,
  q := 2,
  references := rec(
       ),
  upperBound := 4,
  upperBoundExplanation := ... )
```

sage.coding.databases. **self_orthogonal_binary_codes** (*n*, *k*, *b=2*, *parent=None*, *BC=None*, *equal=False*, *in_test=None*)

Returns a Python iterator which generates a complete set of representatives of all permutation equivalence classes of self-orthogonal binary linear codes of length in `[1..n]` and dimension in `[1..k]`.

INPUT:

- `n` - Integer, maximal length
- `k` - Integer, maximal dimension
- `b` - Integer, requires that the generators all have weight divisible by `b` (if `b=2`, all self-orthogonal codes are generated, and if `b=4`, all doubly even codes are generated). Must be an even positive integer.
- `parent` - Used in recursion (default: `None`)
- `BC` - Used in recursion (default: `None`)
- `equal` - If `True` generates only [n, k] codes (default: `False`)
- `in_test` - Used in recursion (default: `None`)

EXAMPLES:

Generate all self-orthogonal codes of length up to 7 and dimension up to 3:

```
sage: for B in codes.databases.self_orthogonal_binary_codes(7,3):
....:     print(B)
Linear code of length 2, dimension 1 over Finite Field of size 2
Linear code of length 4, dimension 2 over Finite Field of size 2
Linear code of length 6, dimension 3 over Finite Field of size 2
Linear code of length 4, dimension 1 over Finite Field of size 2
Linear code of length 6, dimension 2 over Finite Field of size 2
Linear code of length 6, dimension 2 over Finite Field of size 2
Linear code of length 7, dimension 3 over Finite Field of size 2
Linear code of length 6, dimension 1 over Finite Field of size 2
```

Generate all doubly-even codes of length up to 7 and dimension up to 3:

```
sage: for B in codes.databases.self_orthogonal_binary_codes(7,3,4):
....:     print(B); print(B.generator_matrix())
Linear code of length 4, dimension 1 over Finite Field of size 2
[1 1 1 1]
Linear code of length 6, dimension 2 over Finite Field of size 2
```

```
[1 1 1 1 0 0]
[0 1 0 1 1 1]
Linear code of length 7, dimension 3 over Finite Field of size 2
[1 0 1 1 0 1 0]
[0 1 0 1 1 1 0]
[0 0 1 0 1 1 1]
```

Generate all doubly-even codes of length up to 7 and dimension up to 2:

```
sage: for B in codes.databases.self_orthogonal_binary_codes(7,2,4):
....:     print(B); print(B.generator_matrix())
Linear code of length 4, dimension 1 over Finite Field of size 2
[1 1 1 1]
Linear code of length 6, dimension 2 over Finite Field of size 2
[1 1 1 1 0 0]
[0 1 0 1 1 1]
```

Generate all self-orthogonal codes of length equal to 8 and dimension equal to 4:

```
sage: for B in codes.databases.self_orthogonal_binary_codes(8, 4, equal=True):
....:     print(B); print(B.generator_matrix())
Linear code of length 8, dimension 4 over Finite Field of size 2
[1 0 0 1 0 0 0 0]
[0 1 0 0 1 0 0 0]
[0 0 1 0 0 1 0 0]
[0 0 0 0 0 0 1 1]
Linear code of length 8, dimension 4 over Finite Field of size 2
[1 0 0 1 1 0 1 0]
[0 1 0 1 1 1 0 0]
[0 0 1 0 1 1 1 0]
[0 0 0 1 0 1 1 1]
```

Since all the codes will be self-orthogonal, b must be divisible by 2:

```
sage: list(self_orthogonal_binary_codes(8, 4, 1, equal=True))
Traceback (most recent call last):
...
ValueError: b (1) must be a positive even integer.
```

1.9 Database of two-weight codes

This module stores a database of two-weight codes.

$q = 2$	$n = 68$	$k = 8$	$w_1 = 32$	$w_2 = 40$	Shared by Eric Chen *[ChenDB]*.
$q = 2$	$n = 85$	$k = 8$	$w_1 = 40$	$w_2 = 48$	Shared by Eric Chen *[ChenDB]*.
$q = 2$	$n = 70$	$k = 9$	$w_1 = 32$	$w_2 = 40$	Found by Axel Kohnert *[Kohnert07]* and shared by Alfred Wassermann.
$q = 2$	$n = 73$	$k = 9$	$w_1 = 32$	$w_2 = 40$	Shared by Eric Chen *[ChenDB]*.
$q = 2$	$n = 219$	$k = 9$	$w_1 = 96$	$w_2 = 112$	Shared by Eric Chen *[ChenDB]*.
$q = 2$	$n = 198$	$k = 10$	$w_1 = 96$	$w_2 = 112$	Shared by Eric Chen *[ChenDB]*.
$q = 3$	$n = 15$	$k = 4$	$w_1 = 9$	$w_2 = 12$	Shared by Eric Chen *[ChenDB]*.
$q = 3$	$n = 55$	$k = 5$	$w_1 = 36$	$w_2 = 45$	Shared by Eric Chen *[ChenDB]*.
$q = 3$	$n = 56$	$k = 6$	$w_1 = 36$	$w_2 = 45$	Shared by Eric Chen *[ChenDB]*.
$q = 3$	$n = 84$	$k = 6$	$w_1 = 54$	$w_2 = 63$	Shared by Eric Chen *[ChenDB]*.
$q = 3$	$n = 98$	$k = 6$	$w_1 = 63$	$w_2 = 72$	Shared by Eric Chen *[ChenDB]*.
$q = 3$	$n = 126$	$k = 6$	$w_1 = 81$	$w_2 = 90$	Shared by Eric Chen *[ChenDB]*.
$q = 3$	$n = 140$	$k = 6$	$w_1 = 90$	$w_2 = 99$	Found by Axel Kohnert *[Kohnert07]* and shared by Alfred Wassermann.
$q = 3$	$n = 154$	$k = 6$	$w_1 = 99$	$w_2 = 108$	Shared by Eric Chen *[ChenDB]*.
$q = 3$	$n = 168$	$k = 6$	$w_1 = 108$	$w_2 = 117$	From *[Disset00]*
$q = 4$	$n = 34$	$k = 4$	$w_1 = 24$	$w_2 = 28$	Shared by Eric Chen *[ChenDB]*.
$q = 4$	$n = 121$	$k = 5$	$w_1 = 88$	$w_2 = 96$	From *[Disset00]*
$q = 4$	$n = 132$	$k = 5$	$w_1 = 96$	$w_2 = 104$	From *[Disset00]*
$q = 4$	$n = 143$	$k = 5$	$w_1 = 104$	$w_2 = 112$	From *[Disset00]*
$q = 5$	$n = 39$	$k = 4$	$w_1 = 30$	$w_2 = 35$	From Bouyukliev and Simonis (*[BS03]*, Theorem 4.1)
$q = 5$	$n = 52$	$k = 4$	$w_1 = 40$	$w_2 = 45$	Shared by Eric Chen *[ChenDB]*.
$q = 5$	$n = 65$	$k = 4$	$w_1 = 50$	$w_2 = 55$	Shared by Eric Chen *[ChenDB]*.

REFERENCE:

TESTS:

Check the data's consistency:

```
sage: from sage.coding.two_weight_db import data
sage: for code in data:
....:     M = code['M']
....:     assert code['n'] == M.ncols()
```

```
....:     assert code['k'] == M.nrows()
....:     w1,w2 = [w for w,f in enumerate(LinearCode(M).weight_distribution()) if w␣
↪and f]
....:     assert (code['w1'], code['w2']) == (w1, w2)
```

CHAPTER

TWO

LINEAR CODES AND RELATED CONSTRUCTIONS

2.1 Generic structures for linear codes

2.1.1 Linear Codes

Let $F = \mathbf{F}_q$ be a finite field. A rank k linear subspace of the vector space F^n is called an $[n, k]$-linear code, n being the length of the code and k its dimension. Elements of a code C are called codewords.

A linear map from F^k to an $[n, k]$ code C is called an "encoding", and it can be represented as a $k \times n$ matrix, called a generator matrix. Alternatively, C can be represented by its orthogonal complement in F^n, i.e. the $n - k$-dimensional vector space $C^\perp$ such that the inner product of any element from C and any element from $C^\perp$ is zero. $C^\perp$ is called the dual code of C, and any generator matrix for $C^\perp$ is called a parity check matrix for C.

We commonly endow F^n with the Hamming metric, i.e. the weight of a vector is the number of non-zero elements in it. The central operation of a linear code is then "decoding": given a linear code $C \subset F^n$ and a "received word" $r \in F^n$, retrieve the codeword $c \in C$ such that the Hamming distance between r and c is minimal.

2.1.2 Families or Generic codes

Linear codes are either studied as generic vector spaces without any known structure, or as particular sub-families with special properties.

The class *sage.coding.linear_code.LinearCode* is used to represent the former.

For the latter, these will be represented by specialised classes; for instance, the family of Hamming codes are represented by the class *sage.coding.hamming_code.HammingCode* . Type `codes.<tab>` for a list of all code families known to Sage. Such code family classes should inherit from the abstract base class *sage.coding.linear_code.AbstractLinearCode* .

AbstractLinearCode

This is a base class designed to contain methods, features and parameters shared by every linear code. For instance, generic algorithms for computing the minimum distance, the covering radius, etc. Many of these algorithms are slow, e.g. exponential in the code length. For specific subfamilies, better algorithms or even closed formulas might be known, in which case the respective method should be overridden.

`AbstractLinearCode` is an abstract class for linear codes, so any linear code class should inherit from this class. Also `AbstractLinearCode` should never itself be instantiated.

See *sage.coding.linear_code.AbstractLinearCode* for details and examples.

LinearCode

This class is used to represent arbitrary and unstructured linear codes. It mostly rely directly on generic methods provided by `AbstractLinearCode` , which means that basic operations on the code (e.g. computation of the minimum distance) will use slow algorithms.

A `LinearCode` is instantiated by providing a generator matrix:

```
sage: M = matrix(GF(2), [[1, 0, 0, 1, 0],\
                         [0, 1, 0, 1, 1],\
                         [0, 0, 1, 1, 1]])
sage: C = codes.LinearCode(M)
sage: C
Linear code of length 5, dimension 3 over Finite Field of size 2
sage: C.generator_matrix()
[1 0 0 1 0]
[0 1 0 1 1]
[0 0 1 1 1]

sage: MS = MatrixSpace(GF(2),4,7)
sage: G = MS([[1,1,1,0,0,0,0], [1,0,0,1,1,0,0], [0,1,0,1,0,1,0], [1,1,0,1,0,0,1]])
sage: C = LinearCode(G)
sage: C.basis()
[
(1, 1, 1, 0, 0, 0, 0),
(1, 0, 0, 1, 1, 0, 0),
(0, 1, 0, 1, 0, 1, 0),
(1, 1, 0, 1, 0, 0, 1)
]
sage: c = C.basis()[1]
sage: c in C
True
sage: c.nonzero_positions()
[0, 3, 4]
sage: c.support()
[0, 3, 4]
sage: c.parent()
Vector space of dimension 7 over Finite Field of size 2
```

Further references

If you want to get started on Sage's linear codes library, see http://doc.sagemath.org/html/en/thematic_tutorials/coding_theory.html

If you want to learn more on the design of this library, see http://doc.sagemath.org/html/en/thematic_tutorials/structures_in_coding_theory.html

REFERENCES:

AUTHORS:

- David Joyner (2005-11-22, 2006-12-03): initial version
- William Stein (2006-01-23): Inclusion in Sage
- David Joyner (2006-01-30, 2006-04): small fixes
- David Joyner (2006-07): added documentation, group-theoretical methods, ToricCode

- David Joyner (2006-08): hopeful latex fixes to documentation, added list and __iter__ methods to LinearCode and examples, added hamming_weight function, fixed random method to return a vector, TrivialCode, fixed subtle bug in dual_code, added galois_closure method, fixed mysterious bug in permutation_automorphism_group (GAP was over-using "G" somehow?)
- David Joyner (2006-08): hopeful latex fixes to documentation, added CyclicCode, best_known_linear_code, bounds_minimum_distance, assmus_mattson_designs (implementing Assmus-Mattson Theorem).
- David Joyner (2006-09): modified decode syntax, fixed bug in is_galois_closed, added LinearCode_from_vectorspace, extended_code, zeta_function
- Nick Alexander (2006-12-10): factor GUAVA code to guava.py
- David Joyner (2007-05): added methods punctured, shortened, divisor, characteristic_polynomial, binomial_moment, support for LinearCode. Completely rewritten zeta_function (old version is now zeta_function2) and a new function, LinearCodeFromVectorSpace.
- David Joyner (2007-11): added zeta_polynomial, weight_enumerator, chinen_polynomial; improved best_known_code; made some pythonic revisions; added is_equivalent (for binary codes)
- David Joyner (2008-01): fixed bug in decode reported by Harald Schilly, (with Mike Hansen) added some doctests.
- David Joyner (2008-02): translated standard_form, dual_code to Python.
- David Joyner (2008-03): translated punctured, shortened, extended_code, random (and renamed random to random_element), deleted zeta_function2, zeta_function3, added wrapper automorphism_group_binary_code to Robert Miller's code), added direct_sum_code, is_subcode, is_self_dual, is_self_orthogonal, redundancy_matrix, did some alphabetical reorganizing to make the file more readable. Fixed a bug in permutation_automorphism_group which caused it to crash.
- David Joyner (2008-03): fixed bugs in spectrum and zeta_polynomial, which misbehaved over non-prime base rings.
- David Joyner (2008-10): use CJ Tjhal's MinimumWeight if char = 2 or 3 for min_dist; add is_permutation_equivalent and improve permutation_automorphism_group using an interface with Robert Miller's code; added interface with Leon's code for the spectrum method.
- David Joyner (2009-02): added native decoding methods (see module_decoder.py)
- David Joyner (2009-05): removed dependence on Guava, allowing it to be an option. Fixed errors in some docstrings.
- Kwankyu Lee (2010-01): added methods generator_matrix_systematic, information_set, and magma interface for linear codes.
- Niles Johnson (2010-08): trac ticket ##3893: `random_element()` should pass on `*args` and `**kwds`.
- Thomas Feulner (2012-11): trac ticket #13723: deprecation of `hamming_weight()`
- Thomas Feulner (2013-10): added methods to compute a canonical representative and the automorphism group

TESTS:

```
sage: MS = MatrixSpace(GF(2),4,7)
sage: G  = MS([[1,1,1,0,0,0,0], [1,0,0,1,1,0,0], [0,1,0,1,0,1,0], [1,1,0,1,0,0,1]])
sage: C  = LinearCode(G)
sage: C == loads(dumps(C))
True
```

class sage.coding.linear_code. **AbstractLinearCode** (*base_field*, *length*, *default_encoder_name*, *default_decoder_name*)

Bases: sage.modules.module.Module

Abstract class for linear codes.

This class contains all methods that can be used on Linear Codes and on Linear Codes families. So, every Linear Code-related class should inherit from this abstract class.

To implement a linear code, you need to:

- inherit from AbstractLinearCode
- call AbstractLinearCode __init__ method in the subclass constructor. Example: super(SubclassName,self).__init__(base_field,length,"EncoderName","DecoderName") . By doing that, your subclass will have its length parameter initialized and will be properly set as a member of the category framework. You need of course to complete the constructor by adding any additional parameter needed to describe properly the code defined in the subclass.
- fill the dictionary of its encoders in sage.coding.__init__.py file. Example: I want to link the encoder MyEncoderClass to MyNewCodeClass under the name MyEncoderName . All I need to do is to write this line in the __init__.py file: MyNewCodeClass._registered_encoders["NameOfMyEncoder"] = MyEncoderClass and all instances of MyNewCodeClass will be able to use instances of MyEncoderClass .
- fill the dictionary of its decoders in sage.coding.__init__ file. Example: I want to link the encoder MyDecoderClass to MyNewCodeClass under the name MyDecoderName . All I need to do is to write this line in the __init__.py file: MyNewCodeClass._registered_decoders["NameOfMyDecoder"] = MyDecoderClass and all instances of MyNewCodeClass will be able to use instances of MyDecoderClass .

As AbstractLinearCode is not designed to be implemented, it does not have any representation methods. You should implement _repr_ and _latex_ methods in the subclass.

Note: *AbstractLinearCode* has generic implementations of the comparison methods __cmp__ and __eq__ which use the generator matrix and are quite slow. In subclasses you are encouraged to override these functions.

Warning: The default encoder should always have F^k as message space, with k the dimension of the code and F is the base ring of the code.

A lot of methods of the abstract class rely on the knowledge of a generator matrix. It is thus strongly recommended to set an encoder with a generator matrix implemented as a default encoder.

add_decoder (*name*, *decoder*)

Adds an decoder to the list of registered decoders of self .

Note: This method only adds decoder to self , and not to any member of the class of self . To know how to add an *sage.coding.decoder.Decoder* , please refer to the documentation of *AbstractLinearCode* .

INPUT:

- `name` – the string name for the decoder
- `decoder` – the class name of the decoder

EXAMPLES:

First of all, we create a (very basic) new decoder:

```
sage: class MyDecoder(sage.coding.decoder.Decoder):
....:     def __init__(self, code):
....:         super(MyDecoder, self).__init__(code)
....:     def _repr_(self):
....:         return "MyDecoder decoder with associated code %s" % self.code()
```

We now create a new code:

```
sage: C = codes.HammingCode(GF(2), 3)
```

We can add our new decoder to the list of available decoders of C:

```
sage: C.add_decoder("MyDecoder", MyDecoder)
sage: C.decoders_available()
['MyDecoder', 'Syndrome', 'NearestNeighbor']
```

We can verify that any new code will not know MyDecoder:

```
sage: C2 = codes.HammingCode(GF(2), 3)
sage: C2.decoders_available()
['Syndrome', 'NearestNeighbor']
```

TESTS:

It is impossible to use a name which is in the dictionary of available decoders:

```
sage: C.add_decoder("Syndrome", MyDecoder)
Traceback (most recent call last):
...
ValueError: There is already a registered decoder with this name
```

add_encoder (*name*, *encoder*)

Adds an encoder to the list of registered encoders of `self`.

Note: This method only adds `encoder` to `self`, and not to any member of the class of `self`. To know how to add an *sage.coding.encoder.Encoder*, please refer to the documentation of *AbstractLinearCode*.

INPUT:

- `name` – the string name for the encoder
- `encoder` – the class name of the encoder

EXAMPLES:

First of all, we create a (very basic) new encoder:

```
sage: class MyEncoder(sage.coding.encoder.Encoder):
....:     def __init__(self, code):
....:         super(MyEncoder, self).__init__(code)
```

```
....:     def _repr_(self):
....:         return "MyEncoder encoder with associated code %s" % self.code()
```

We now create a new code:

```
sage: C = codes.HammingCode(GF(2), 3)
```

We can add our new encoder to the list of available encoders of C:

```
sage: C.add_encoder("MyEncoder", MyEncoder)
sage: C.encoders_available()
['MyEncoder', 'ParityCheck', 'Systematic']
```

We can verify that any new code will not know MyEncoder:

```
sage: C2 = codes.HammingCode(GF(2), 3)
sage: C2.encoders_available()
['Systematic', 'ParityCheck']
```

TESTS:

It is impossible to use a name which is in the dictionary of available encoders:

```
sage: C.add_encoder("ParityCheck", MyEncoder)
Traceback (most recent call last):
...
ValueError: There is already a registered encoder with this name
```

ambient_space ( )

Returns the ambient vector space of $self$.

EXAMPLES:

```
sage: C = codes.HammingCode(GF(2), 3)
sage: C.ambient_space()
Vector space of dimension 7 over Finite Field of size 2
```

assmus_mattson_designs (*t*, *mode=None*)

Assmus and Mattson Theorem (section 8.4, page 303 of *[HP]*): Let $A_0, A_1, ..., A_n$ be the weights of the codewords in a binary linear $[n, k, d]$ code C, and let $A_0^*, A_1^*, ..., A_n^*$ be the weights of the codewords in its dual $[n, n-k, d^*]$ code C^*. Fix a t, $0 < t < d$, and let

$$s = |\{i \mid A_i^* \neq 0, 0 < i \leq n - t\}|.$$

Assume $s \leq d - t$.

1.If $A_i \neq 0$ and $d \leq i \leq n$ then $C_i = \{c \in C \mid wt(c) = i\}$ holds a simple t-design.

2.If $A_i^* \neq 0$ and $d* \leq i \leq n - t$ then $C_i^* = \{c \in C^* \mid wt(c) = i\}$ holds a simple t-design.

A block design is a pair (X, B), where X is a non-empty finite set of $v > 0$ elements called points, and B is a non-empty finite multiset of size b whose elements are called blocks, such that each block is a non-empty finite multiset of k points. A design without repeated blocks is called a simple block design. If every subset of points of size t is contained in exactly λ blocks the block design is called a $t - (v, k, \lambda)$ design (or simply a t-design when the parameters are not specified). When $\lambda = 1$ then the block design is called a $S(t, k, v)$ Steiner system.

In the Assmus and Mattson Theorem (1), X is the set $\{1, 2, ..., n\}$ of coordinate locations and $B = \{supp(c) \mid c \in C_i\}$ is the set of supports of the codewords of C of weight i. Therefore, the parameters of the t-design for C_i are

```
t =       given
v =       n
k =       i    (k not to be confused with dim(C))
b =       Ai
lambda = b*binomial(k,t)/binomial(v,t)  (by Theorem 8.1.6,
                                         p 294, in [HP]_)
```

Setting the `mode="verbose"` option prints out the values of the parameters.

The first example below means that the binary [24,12,8]-code C has the property that the (support of the) codewords of weight 8 (resp., 12, 16) form a 5-design. Similarly for its dual code C^* (of course $C = C^*$ in this case, so this info is extraneous). The test fails to produce 6-designs (ie, the hypotheses of the theorem fail to hold, not that the 6-designs definitely don't exist). The command assmus_mattson_designs(C,5,mode="verbose") returns the same value but prints out more detailed information.

The second example below illustrates the blocks of the 5-(24, 8, 1) design (i.e., the S(5,8,24) Steiner system).

EXAMPLES:

```
sage: C = codes.ExtendedBinaryGolayCode()              #  example 1
sage: C.assmus_mattson_designs(5)
['weights from C: ',
[8, 12, 16, 24],
'designs from C: ',
[[5, (24, 8, 1)], [5, (24, 12, 48)], [5, (24, 16, 78)], [5, (24, 24, 1)]],
'weights from C*: ',
[8, 12, 16],
'designs from C*: ',
[[5, (24, 8, 1)], [5, (24, 12, 48)], [5, (24, 16, 78)]]]
sage: C.assmus_mattson_designs(6)
0
sage: X = range(24)                                #  example 2
sage: blocks = [c.support() for c in C if c.hamming_weight()==8]; len(blocks)␣
↪ # long time computation
759
```

automorphism_group_gens (*equivalence='semilinear'*)

Return generators of the automorphism group of `self` .

INPUT:

- `equivalence` (optional) – which defines the acting group, either
  - `permutational`
  - `linear`
  - `semilinear`

OUTPUT:

- generators of the automorphism group of `self`
- the order of the automorphism group of `self`

EXAMPLES:

```
sage: C = codes.HammingCode(GF(4, 'z'), 3)
sage: C.automorphism_group_gens()
([((1, 1, 1, z, z + 1, z + 1, z + 1, z, z, 1, 1, 1, z, z, z + 1, z, z, z + 1,␣
↪z + 1, z + 1, 1); (1,6,12,17)(2,16,4,5,11,8,14,13)(3,21,19,10,20,18,15,9),␣
↪Ring endomorphism of Finite Field in z of size 2^2
```

```
      Defn: z |--> z + 1), ((1, 1, 1, z, z + 1, 1, 1, z, z, z + 1, z, z, z +␣
↪1, z + 1, z + 1, 1, z + 1, z, z, 1, 1);␣
↪(1,6,9,13,15,18)(2,21)(3,16,7)(4,5,11,10,12,14)(17,19), Ring endomorphism␣
↪of Finite Field in z of size 2^2
      Defn: z |--> z), ((z, z, z, z, z, z, z, z, z, z, z, z, z, z, z, z, z,␣
↪z, z, z, z); (), Ring endomorphism of Finite Field in z of size 2^2
      Defn: z |--> z)], 362880)
sage: C.automorphism_group_gens(equivalence="linear")
([((z, z, 1, 1, z + 1, z, z + 1, z, z, z + 1, 1, 1, 1, z + 1, z, z, z + 1, z␣
↪+ 1, 1, 1, z); (1,5,10,9,4,14,11,16,18,20,6,19,12,15,3,8,2,17,7,13,21),␣
↪Ring endomorphism of Finite Field in z of size 2^2
      Defn: z |--> z), ((z + 1, 1, z, 1, 1, z + 1, z + 1, z, 1, z, z + 1, z,␣
↪z + 1, z + 1, z, 1, 1, z + 1, z + 1, z + 1, z);␣
↪(1,17,10)(2,15,13)(4,11,21)(5,18,12)(6,14,19)(7,8,16), Ring endomorphism of␣
↪Finite Field in z of size 2^2
      Defn: z |--> z), ((z + 1, z + 1, z + 1, z + 1, z + 1, z + 1, z + 1, z +␣
↪1, z + 1, z + 1, z + 1, z + 1, z + 1, z + 1, z + 1, z + 1, z + 1, z + 1, z␣
↪+ 1, z + 1, z + 1); (), Ring endomorphism of Finite Field in z of size 2^2
      Defn: z |--> z)], 181440)
sage: C.automorphism_group_gens(equivalence="permutational")
([((1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1);␣
↪(1,11)(3,10)(4,9)(5,7)(12,21)(14,20)(15,19)(16,17), Ring endomorphism of␣
↪Finite Field in z of size 2^2
      Defn: z |--> z), ((1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,␣
↪1, 1, 1, 1); (2,18)(3,19)(4,10)(5,16)(8,13)(9,14)(11,21)(15,20), Ring␣
↪endomorphism of Finite Field in z of size 2^2
      Defn: z |--> z), ((1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,␣
↪1, 1, 1, 1); (1,19)(3,17)(4,21)(5,20)(7,14)(9,12)(10,16)(11,15), Ring␣
↪endomorphism of Finite Field in z of size 2^2
      Defn: z |--> z), ((1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,␣
↪1, 1, 1, 1); (2,13)(3,14)(4,20)(5,11)(8,18)(9,19)(10,15)(16,21), Ring␣
↪endomorphism of Finite Field in z of size 2^2
      Defn: z |--> z)], 64)
```

base_field ( )

Return the base field of `self` .

EXAMPLES:

```
sage: G  = Matrix(GF(2), [[1,1,1,0,0,0,0], [1,0,0,1,1,0,0], [0,1,0,1,0,1,0],␣
↪[1,1,0,1,0,0,1]])
sage: C  = LinearCode(G)
sage: C.base_field()
Finite Field of size 2
```

basis ( )

Returns a basis of $self$.

OUTPUT:

- `Sequence` - an immutable sequence whose universe is ambient space of $self$.

EXAMPLES:

```
sage: C = codes.HammingCode(GF(2), 3)
sage: C.basis()
[
(1, 0, 0, 0, 0, 1, 1),
(0, 1, 0, 0, 1, 0, 1),
```

```
(0, 0, 1, 0, 1, 1, 0),
(0, 0, 0, 1, 1, 1, 1)
]
sage: C.basis().universe()
Vector space of dimension 7 over Finite Field of size 2
```

binomial_moment (*i*)

Returns the i-th binomial moment of the $[n, k, d]_q$-code C:

$$B_i(C) = \sum_{S,|S|=i} \frac{q^{k_S} - 1}{q - 1}$$

where k_S is the dimension of the shortened code C_{J-S}, $J = [1, 2, ..., n]$. (The normalized binomial moment is $b_i(C) = \binom{n}{}, d + i)^{-1} B_{d+i}(C)$.) In other words, C_{J-S} is isomorphic to the subcode of C of codewords supported on S.

EXAMPLES:

```
sage: C = codes.HammingCode(GF(2), 3)
sage: C.binomial_moment(2)
0
sage: C.binomial_moment(4)     # long time
35
```

Warning: This is slow.

REFERENCE:

canonical_representative (*equivalence='semilinear'*)

Compute a canonical orbit representative under the action of the semimonomial transformation group.

See *sage.coding.codecan.autgroup_can_label* for more details, for example if you would like to compute a canonical form under some more restrictive notion of equivalence, i.e. if you would like to restrict the permutation group to a Young subgroup.

INPUT:

- `equivalence` (optional) – which defines the acting group, either
  - `permutational`
  - `linear`
  - `semilinear`

OUTPUT:

- a canonical representative of `self`
- a semimonomial transformation mapping `self` onto its representative

EXAMPLES:

```
sage: F.<z> = GF(4)
sage: C = codes.HammingCode(F, 3)
sage: CanRep, transp = C.canonical_representative()
```

Check that the transporter element is correct:

```
sage: LinearCode(transp*C.generator_matrix()) == CanRep
True
```

Check if an equivalent code has the same canonical representative:

```
sage: f = F.hom([z**2])
sage: C_iso = LinearCode(C.generator_matrix().apply_map(f))
sage: CanRep_iso, _ = C_iso.canonical_representative()
sage: CanRep_iso == CanRep
True
```

Since applying the Frobenius automorphism could be extended to an automorphism of C, the following must also yield `True` :

```
sage: CanRep1, _ = C.canonical_representative("linear")
sage: CanRep2, _ = C_iso.canonical_representative("linear")
sage: CanRep2 == CanRep1
True
```

cardinality ( )

Return the size of this code.

EXAMPLES:

```
sage: C = codes.HammingCode(GF(2), 3)
sage: C.cardinality()
16
sage: len(C)
16
```

characteristic ( )

Returns the characteristic of the base ring of $self$.

EXAMPLES:

```
sage: C = codes.HammingCode(GF(2), 3)
sage: C.characteristic()
2
```

characteristic_polynomial ( )

Returns the characteristic polynomial of a linear code, as defined in van Lint's text *[vL]*.

EXAMPLES:

```
sage: C = codes.ExtendedBinaryGolayCode()
sage: C.characteristic_polynomial()
-4/3*x^3 + 64*x^2 - 2816/3*x + 4096
```

REFERENCES:

chinen_polynomial ( )

Returns the Chinen zeta polynomial of the code.

EXAMPLES:

```
sage: C = codes.HammingCode(GF(2), 3)
sage: C.chinen_polynomial()        # long time
1/5*(2*sqrt(2)*t^3 + 2*sqrt(2)*t^2 + 2*t^2 + sqrt(2)*t + 2*t + 1)/(sqrt(2) +␣
↪1)
```

```
sage: C = codes.TernaryGolayCode()
sage: C.chinen_polynomial()        # long time
1/7*(3*sqrt(3)*t^3 + 3*sqrt(3)*t^2 + 3*t^2 + sqrt(3)*t + 3*t + 1)/(sqrt(3) +␣
↪1)
```

This last output agrees with the corresponding example given in Chinen's paper below.

REFERENCES:

- Chinen, K. "An abundance of invariant polynomials satisfying the Riemann hypothesis", April 2007 preprint.

covering_radius ( )

Return the minimimal integer r such that any element in the ambient space of `self` has distance at most r to a codeword of `self` .

This method requires the optional GAP package Guava.

If the covering radius a code equals its minimum distance, then the code is called perfect.

EXAMPLES:

```
sage: C = codes.HammingCode(GF(2), 5)
sage: C.covering_radius()  # optional - gap_packages (Guava package)
1
```

decode (*right*, *algorithm='syndrome'*)

Corrects the errors in `right` and returns a codeword.

INPUT:

- `right` – a vector of the same length as `self` over the base field of `self`
- `algorithm` – (default: `'syndrome'`) Name of the decoding algorithm which will be used to decode `right` . Can be `'syndrome'` or `'nearest_neighbor'` .

Note: This is a deprecated method which will soon be removed from Sage. Please use *decode_to_code()* instead.

---

decode_to_code (*word*, *decoder_name=None*, ***kwargs*)

Corrects the errors in `word` and returns a codeword.

INPUT:

- `word` – a vector of the same length as `self` over the base field of `self`
- `decoder_name` – (default: `None`) Name of the decoder which will be used to decode `word` . The default decoder of `self` will be used if default value is kept.
- `kwargs` – all additional arguments are forwarded to *decoder()*

OUTPUT:

- A vector of `self` .

EXAMPLES:

```
sage: G = Matrix(GF(2),␣
↪[[1,1,1,0,0,0,0],[1,0,0,1,1,0,0],[0,1,0,1,0,1,0],[1,1,0,1,0,0,1]])
sage: C = LinearCode(G)
sage: word = vector(GF(2), (1, 1, 0, 0, 1, 1, 0))
```

```
sage: w_err = word + vector(GF(2), (1, 0, 0, 0, 0, 0, 0))
sage: C.decode_to_code(w_err)
(1, 1, 0, 0, 1, 1, 0)
```

It is possible to manually choose the decoder amongst the list of the available ones:

```
sage: C.decoders_available()
['Syndrome', 'NearestNeighbor']
sage: C.decode_to_code(w_err, 'NearestNeighbor')
(1, 1, 0, 0, 1, 1, 0)
```

decode_to_message (*word*, *decoder_name=None*, ***kwargs*)

Correct the errors in word and decodes it to the message space.

INPUT:

- `word` – a vector of the same length as `self` over the base field of `self`
- `decoder_name` – (default: `None`) Name of the decoder which will be used to decode `word`. The default decoder of `self` will be used if default value is kept.
- `kwargs` – all additional arguments are forwarded to *decoder()*

OUTPUT:

- A vector of the message space of `self`.

EXAMPLES:

```
sage: G = Matrix(GF(2),␣
↪[[1,1,1,0,0,0,0],[1,0,0,1,1,0,0],[0,1,0,1,0,1,0],[1,1,0,1,0,0,1]])
sage: C = LinearCode(G)
sage: word = vector(GF(2), (1, 1, 0, 0, 1, 1, 0))
sage: C.decode_to_message(word)
(0, 1, 1, 0)
```

It is possible to manually choose the decoder amongst the list of the available ones:

```
sage: C.decoders_available()
['Syndrome', 'NearestNeighbor']
sage: C.decode_to_message(word, 'NearestNeighbor')
(0, 1, 1, 0)
```

decoder (*decoder_name=None*, ***kwargs*)

Return a decoder of `self`.

INPUT:

- `decoder_name` – (default: `None`) name of the decoder which will be returned. The default decoder of `self` will be used if default value is kept.
- `kwargs` – all additional arguments will be forwarded to the constructor of the decoder that will be returned by this method

OUTPUT:

- a decoder object

Besides creating the decoder and returning it, this method also stores the decoder in a cache. With this behaviour, each decoder will be created at most one time for `self`.

EXAMPLES:

```
sage: G = Matrix(GF(2),␣
↪[[1,1,1,0,0,0,0],[1,0,0,1,1,0,0],[0,1,0,1,0,1,0],[1,1,0,1,0,0,1]])
sage: C = LinearCode(G)
sage: C.decoder()
Syndrome decoder for Linear code of length 7, dimension 4 over Finite Field␣
↪of size 2 handling errors of weight up to 1
```

If the name of a decoder which is not known by `self` is passed, an exception will be raised:

```
sage: C.decoders_available()
['Syndrome', 'NearestNeighbor']
sage: C.decoder('Try')
Traceback (most recent call last):
...
ValueError: Passed Decoder name not known
```

decoders_available (*classes=False*)

Returns a list of the available decoders' names for `self` .

INPUT:

- `classes` – (default: `False`) if `classes` is set to `True` , it also returns the decoders' classes associated with the decoders' names.

EXAMPLES:

```
sage: G = Matrix(GF(2),␣
↪[[1,1,1,0,0,0,0],[1,0,0,1,1,0,0],[0,1,0,1,0,1,0],[1,1,0,1,0,0,1]])
sage: C = LinearCode(G)
sage: C.decoders_available()
['Syndrome', 'NearestNeighbor']

sage: C.decoders_available(True)
{'NearestNeighbor': <class 'sage.coding.linear_code.
↪LinearCodeNearestNeighborDecoder'>,
 'Syndrome': <class 'sage.coding.linear_code.LinearCodeSyndromeDecoder'>}
```

dimension ( )

Returns the dimension of this code.

EXAMPLES:

```
sage: G = matrix(GF(2),[[1,0,0],[1,1,0]])
sage: C = LinearCode(G)
sage: C.dimension()
2
```

direct_sum (*other*)

Returns the code given by the direct sum of the codes `self` and `other` , which must be linear codes defined over the same base ring.

EXAMPLES:

```
sage: C1 = codes.HammingCode(GF(2), 3)
sage: C2 = C1.direct_sum(C1); C2
Linear code of length 14, dimension 8 over Finite Field of size 2
sage: C3 = C1.direct_sum(C2); C3
Linear code of length 21, dimension 12 over Finite Field of size 2
```

divisor ( )
Returns the greatest common divisor of the weights of the nonzero codewords.

EXAMPLES:

```
sage: C = codes.ExtendedBinaryGolayCode()
sage: C.divisor()   # Type II self-dual
4
sage: C = codes.QuadraticResidueCodeEvenPair(17,GF(2))[0]
sage: C.divisor()
2
```

dual_code ( )
Returns the dual code $C^\perp$ of the code C,

$$C^\perp = \{v \in V \mid v \cdot c = 0, \ \forall c \in C\}.$$

EXAMPLES:

```
sage: C = codes.HammingCode(GF(2), 3)
sage: C.dual_code()
Linear code of length 7, dimension 3 over Finite Field of size 2
sage: C = codes.HammingCode(GF(4, 'a'), 3)
sage: C.dual_code()
Linear code of length 21, dimension 3 over Finite Field in a of size 2^2
```

encode (*word*, *encoder_name=None*, ***kwargs*)
Transforms an element of a message space into a codeword.

INPUT:

- `word` – a vector of a message space of the code.
- `encoder_name` – (default: `None`) Name of the encoder which will be used to encode `word` . The default encoder of `self` will be used if default value is kept.
- `kwargs` – all additional arguments are forwarded to the construction of the encoder that is used.

Note: The default encoder always has F^k as message space, with k the dimension of `self` and F the base ring of `self` .

OUTPUT:

- a vector of `self` .

EXAMPLES:

```
sage: G = Matrix(GF(2),␣
↪[[1,1,1,0,0,0,0],[1,0,0,1,1,0,0],[0,1,0,1,0,1,0],[1,1,0,1,0,0,1]])
sage: C = LinearCode(G)
sage: word = vector((0, 1, 1, 0))
sage: C.encode(word)
(1, 1, 0, 0, 1, 1, 0)
```

It is possible to manually choose the encoder amongst the list of the available ones:

```
sage: C.encoders_available()
['GeneratorMatrix', 'Systematic']
sage: word = vector((0, 1, 1, 0))
```

```
sage: C.encode(word, 'GeneratorMatrix')
(1, 1, 0, 0, 1, 1, 0)
```

encoder (*encoder_name=None*, ***kwargs*)

Returns an encoder of `self` .

The returned encoder provided by this method is cached.

This methods creates a new instance of the encoder subclass designated by `encoder_name` . While it is also possible to do the same by directly calling the subclass' constructor, it is strongly advised to use this method to take advantage of the caching mechanism.

INPUT:

- `encoder_name` – (default: `None`) name of the encoder which will be returned. The default encoder of `self` will be used if default value is kept.
- `kwargs` – all additional arguments are forwarded to the constructor of the encoder this method will return.

OUTPUT:

- an Encoder object.

Note: The default encoder always has F^k as message space, with k the dimension of `self` and F the base ring of `self` .

EXAMPLES:

```
sage: G = Matrix(GF(2),␣
↪[[1,1,1,0,0,0,0],[1,0,0,1,1,0,0],[0,1,0,1,0,1,0],[1,1,0,1,0,0,1]])
sage: C = LinearCode(G)
sage: C.encoder()
Generator matrix-based encoder for Linear code of length 7, dimension 4 over␣
↪Finite Field of size 2
```

We check that the returned encoder is cached:

```
sage: C.encoder.is_in_cache()
True
```

If the name of an encoder which is not known by `self` is passed, an exception will be raised:

```
sage: C.encoders_available()
['GeneratorMatrix', 'Systematic']
sage: C.encoder('NonExistingEncoder')
Traceback (most recent call last):
...
ValueError: Passed Encoder name not known
```

encoders_available (*classes=False*)

Returns a list of the available encoders' names for `self` .

INPUT:

- `classes` – (default: `False`) if `classes` is set to `True` , it also returns the encoders' classes associated with the encoders' names.

EXAMPLES:

```
sage: G = Matrix(GF(2),␣
↪[[1,1,1,0,0,0,0],[1,0,0,1,1,0,0],[0,1,0,1,0,1,0],[1,1,0,1,0,0,1]])
sage: C = LinearCode(G)
sage: C.encoders_available()
['GeneratorMatrix', 'Systematic']
sage: C.encoders_available(True)
{'GeneratorMatrix': <class 'sage.coding.linear_code.
↪LinearCodeGeneratorMatrixEncoder'>,
 'Systematic': <class 'sage.coding.linear_code.LinearCodeSystematicEncoder'>}
```

extended_code ( )

Returns $self$ as an extended code.

See documentation of *sage.coding.extended_code.ExtendedCode* for details. EXAMPLES:

```
sage: C = codes.HammingCode(GF(4,'a'), 3)
sage: C
[21, 18] Hamming Code over Finite Field in a of size 2^2
sage: Cx = C.extended_code()
sage: Cx
Extended code coming from [21, 18] Hamming Code over Finite Field in a of␣
↪size 2^2
```

galois_closure (*F0*)

If `self` is a linear code defined over F and F_0 is a subfield with Galois group $G = Gal(F/F_0)$ then this returns the G-module C^- containing C.

EXAMPLES:

```
sage: C = codes.HammingCode(GF(4,'a'), 3)
sage: Cc = C.galois_closure(GF(2))
sage: C; Cc
[21, 18] Hamming Code over Finite Field in a of size 2^2
Linear code of length 21, dimension 20 over Finite Field in a of size 2^2
sage: c = C.basis()[2]
sage: V = VectorSpace(GF(4,'a'),21)
sage: c2 = V([x^2 for x in c.list()])
sage: c2 in C
False
sage: c2 in Cc
True
```

generator_matrix (*encoder_name=None*, ***kwargs*)

Returns a generator matrix of `self` .

INPUT:

- `encoder_name` – (default: `None`) name of the encoder which will be used to compute the generator matrix. The default encoder of `self` will be used if default value is kept.
- `kwargs` – all additional arguments are forwarded to the construction of the encoder that is used.

EXAMPLES:

```
sage: G = matrix(GF(3),2,[1,-1,1,-1,1,1])
sage: code = LinearCode(G)
sage: code.generator_matrix()
[1 2 1]
[2 1 1]
```

generator_matrix_systematic (**args*, ***kwds*)
Deprecated: Use *systematic_generator_matrix()* instead. See trac ticket #20835 for details.

gens ( )
Returns the generators of this code as a list of vectors.

EXAMPLES:

```
sage: C = codes.HammingCode(GF(2), 3)
sage: C.gens()
 [(1, 0, 0, 0, 0, 1, 1), (0, 1, 0, 0, 1, 0, 1), (0, 0, 1, 0, 1, 1, 0), (0, 0,␣
↪0, 1, 1, 1, 1)]
```

genus ( )
Returns the "Duursma genus" of the code, $\gamma_C = n + 1 - k - d$.

EXAMPLES:

```
sage: C1 = codes.HammingCode(GF(2), 3); C1
[7, 4] Hamming Code over Finite Field of size 2
sage: C1.genus()
1
sage: C2 = codes.HammingCode(GF(4,"a"), 2); C2
[5, 3] Hamming Code over Finite Field in a of size 2^2
sage: C2.genus()
0
```

Since all Hamming codes have minimum distance 3, these computations agree with the definition, $n + 1 - k - d$.

information_set ( )
Return an information set of the code.

Return value of this method is cached.

A set of column positions of a generator matrix of a code is called an information set if the corresponding columns form a square matrix of full rank.

OUTPUT:

•Information set of a systematic generator matrix of the code.

EXAMPLES:

```
sage: G = matrix(GF(3),2,[1,2,0,                                        2,1,1])
sage: code = LinearCode(G)
sage: code.systematic_generator_matrix()
[1 2 0]
[0 0 1]
sage: code.information_set()
(0, 2)
```

is_galois_closed ( )
Checks if `self` is equal to its Galois closure.

EXAMPLES:

```
sage: C = codes.HammingCode(GF(4,"a"), 3)
sage: C.is_galois_closed()
False
```

is_information_set (*positions*)
Return whether the given positions form an information set.

INPUT:

•A list of positions, i.e. integers in the range 0 to $n-1$ where n is the length of $self$.

OUTPUT:

•A boolean indicating whether the positions form an information set.

EXAMPLES:

```
sage: G = matrix(GF(3),2,[1,2,0,                                        2,1,1])
sage: code = LinearCode(G)
sage: code.is_information_set([0,1])
False
sage: code.is_information_set([0,2])
True
```

is_permutation_automorphism (*g*)
Returns 1 if g is an element of S_n (n = length of self) and if g is an automorphism of self.

EXAMPLES:

```
sage: C = codes.HammingCode(GF(3), 3)
sage: g = SymmetricGroup(13).random_element()
sage: C.is_permutation_automorphism(g)
0
sage: MS = MatrixSpace(GF(2),4,8)
sage: G  =␣
↪MS([[1,0,0,0,1,1,1,0],[0,1,1,1,0,0,0,0],[0,0,0,0,0,0,0,1],[0,0,0,0,0,1,0,0]])
sage: C  = LinearCode(G)
sage: S8 = SymmetricGroup(8)
sage: g = S8("(2,3)")
sage: C.is_permutation_automorphism(g)
1
sage: g = S8("(1,2,3,4)")
sage: C.is_permutation_automorphism(g)
0
```

is_permutation_equivalent (*other*, *algorithm=None*)
Returns `True` if `self` and `other` are permutation equivalent codes and `False` otherwise.

The `algorithm="verbose"` option also returns a permutation (if `True`) sending `self` to `other` .

Uses Robert Miller's double coset partition refinement work.

EXAMPLES:

```
sage: P.<x> = PolynomialRing(GF(2),"x")
sage: g = x^3+x+1
sage: C1 = codes.CyclicCodeFromGeneratingPolynomial(7,g); C1
Linear code of length 7, dimension 4 over Finite Field of size 2
sage: C2 = codes.HammingCode(GF(2), 3); C2
[7, 4] Hamming Code over Finite Field of size 2
sage: C1.is_permutation_equivalent(C2)
True
sage: C1.is_permutation_equivalent(C2,algorithm="verbose")
(True, (3,4)(5,7,6))
sage: C1 = codes.random_linear_code(GF(2), 10, 5)
sage: C2 = codes.random_linear_code(GF(3), 10, 5)
```

```
sage: C1.is_permutation_equivalent(C2)
False
```

is_projective ( )

Test whether the code is projective.

A linear code C over a field is called *projective* when its dual Cd has minimum weight ≥ 3, i.e. when no two coordinate positions of C are linearly independent (cf. definition 3 from *[BS11]* or 9.8.1 from [BH12]).

EXAMPLE:

```
sage: C = codes.BinaryGolayCode()
sage: C.is_projective()
True
sage: C.dual_code().minimum_distance()
8
```

A non-projective code:

```
sage: C = codes.LinearCode(matrix(GF(2),[[1,0,1],[1,1,1]]))
sage: C.is_projective()
False
```

REFERENCE:

is_self_dual ( )

Returns `True` if the code is self-dual (in the usual Hamming inner product) and `False` otherwise.

EXAMPLES:

```
sage: C = codes.ExtendedBinaryGolayCode()
sage: C.is_self_dual()
True
sage: C = codes.HammingCode(GF(2), 3)
sage: C.is_self_dual()
False
```

is_self_orthogonal ( )

Returns `True` if this code is self-orthogonal and `False` otherwise.

A code is self-orthogonal if it is a subcode of its dual.

EXAMPLES:

```
sage: C = codes.ExtendedBinaryGolayCode()
sage: C.is_self_orthogonal()
True
sage: C = codes.HammingCode(GF(2), 3)
sage: C.is_self_orthogonal()
False
sage: C = codes.QuasiQuadraticResidueCode(11)  # optional - gap_packages␣
↪(Guava package)
sage: C.is_self_orthogonal()                   # optional - gap_packages (Guava␣
↪package)
True
```

is_subcode (*other*)

Returns `True` if `self` is a subcode of `other` .

EXAMPLES:

```
sage: C1 = codes.HammingCode(GF(2), 3)
sage: G1 = C1.generator_matrix()
sage: G2 = G1.matrix_from_rows([0,1,2])
sage: C2 = LinearCode(G2)
sage: C2.is_subcode(C1)
True
sage: C1.is_subcode(C2)
False
sage: C3 = C1.extended_code()
sage: C1.is_subcode(C3)
False
sage: C4 = C1.punctured([1])
sage: C4.is_subcode(C1)
False
sage: C5 = C1.shortened([1])
sage: C5.is_subcode(C1)
False
sage: C1 = codes.HammingCode(GF(9,"z"), 3)
sage: G1 = C1.generator_matrix()
sage: G2 = G1.matrix_from_rows([0,1,2])
sage: C2 = LinearCode(G2)
sage: C2.is_subcode(C1)
True
```

length ( )

Returns the length of this code.

EXAMPLES:

```
sage: C = codes.HammingCode(GF(2), 3)
sage: C.length()
7
```

list ( )

Return a list of all elements of this linear code.

EXAMPLES:

```
sage: C = codes.HammingCode(GF(2), 3)
sage: Clist = C.list()
sage: Clist[5]; Clist[5] in C
(1, 0, 1, 0, 1, 0, 1)
True
```

minimum_distance (*algorithm=None*)

Returns the minimum distance of this linear code.

By default, this uses a GAP kernel function (in C and not part of Guava) written by Steve Linton. If `algorithm="guava"` is set and q is 2 or 3 then this uses a very fast program written in C written by CJ Tjhal. (This is much faster, except in some small examples.)

Raises a `ValueError` in case there is no non-zero vector in this linear code.

The minimum distance of the code is stored once it has been computed or provided during the initialization of *LinearCode*. If `algorithm` is `None` and the stored value of minimum distance is found, then the stored value will be returned without recomputing the minimum distance again.

Note: When using GAP, this raises a `NotImplementedError` if the base field of the code has size greater than 256 due to limitations in GAP.

INPUT:

- `algorithm` - Method to be used, `None`, `"gap"`, or `"guava"` (default: `None`).

OUTPUT:

- Integer, minimum distance of this code

EXAMPLES:

```
sage: MS = MatrixSpace(GF(3),4,7)
sage: G = MS([[1,1,1,0,0,0,0], [1,0,0,1,1,0,0], [0,1,0,1,0,1,0],␣
↪[1,1,0,1,0,0,1]])
sage: C = LinearCode(G)
sage: C.minimum_distance()
3
```

Once the minimum distance has been computed, it's value is stored. Hence the following command will return the value instantly, without further computations.:

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

If `algorithm` is provided, then the minimum distance will be recomputed even if there is a stored value from a previous run.:

```
sage: C.minimum_distance(algorithm="gap")
3
sage: C.minimum_distance(algorithm="guava")  # optional - gap_packages (Guava␣
↪package)
3
```

Another example.:

```
sage: C = codes.HammingCode(GF(4,"a"), 2); C
[5, 3] Hamming Code over Finite Field in a of size 2^2
sage: C.minimum_distance()
3
```

TESTS:

```
sage: C = codes.random_linear_code(GF(4,"a"), 5, 2)
sage: C.minimum_distance(algorithm='something')
Traceback (most recent call last):
...
ValueError: The algorithm argument must be one of None, 'gap' or 'guava'; got
↪'something'
```

The field must be size at most 256:

```
sage: C = codes.random_linear_code(GF(257,"a"), 5, 2)
sage: C.minimum_distance()
Traceback (most recent call last):
...
```

```
NotImplementedError: the GAP algorithm that Sage is using
 is limited to computing with fields of size at most 256
```

module_composition_factors (*gp*)

Prints the GAP record of the Meataxe composition factors module in Meataxe notation. This uses GAP but not Guava.

EXAMPLES:

```
sage: MS = MatrixSpace(GF(2),4,8)
sage: G  =␣
↪MS([[1,0,0,0,1,1,1,0],[0,1,1,1,0,0,0,0],[0,0,0,0,0,0,0,1],[0,0,0,0,0,1,0,0]])
sage: C  = LinearCode(G)
sage: gp = C.permutation_automorphism_group()
```

Now type "C.module_composition_factors(gp)" to get the record printed.

parity_check_matrix ( )

Returns the parity check matrix of `self` .

The parity check matrix of a linear code C corresponds to the generator matrix of the dual code of C.

EXAMPLES:

```
sage: C = codes.HammingCode(GF(2), 3)
sage: Cperp = C.dual_code()
sage: C; Cperp
[7, 4] Hamming Code over Finite Field of size 2
Linear code of length 7, dimension 3 over Finite Field of size 2
sage: C.generator_matrix()
 [1 0 0 0 0 1 1]
 [0 1 0 0 1 0 1]
 [0 0 1 0 1 1 0]
 [0 0 0 1 1 1 1]
sage: C.parity_check_matrix()
 [1 0 1 0 1 0 1]
 [0 1 1 0 0 1 1]
 [0 0 0 1 1 1 1]
sage: Cperp.parity_check_matrix()
 [1 0 0 0 0 1 1]
 [0 1 0 0 1 0 1]
 [0 0 1 0 1 1 0]
 [0 0 0 1 1 1 1]
sage: Cperp.generator_matrix()
 [1 0 1 0 1 0 1]
 [0 1 1 0 0 1 1]
 [0 0 0 1 1 1 1]
```

permutation_automorphism_group (*algorithm='partition'*)

If C is an $[n, k, d]$ code over F, this function computes the subgroup $Aut(C) \subset S_n$ of all permutation automorphisms of C. The binary case always uses the (default) partition refinement algorithm of Robert Miller.

Note that if the base ring of C is $GF(2)$ then this is the full automorphism group. Otherwise, you could use `automorphism_group_gens()` to compute generators of the full automorphism group.

INPUT:

- `algorithm` - If `"gap"` then GAP's MatrixAutomorphism function (written by Thomas Breuer) is used. The implementation combines an idea of mine with an improvement suggested by Cary

Huffman. If "gap+verbose" then code-theoretic data is printed out at several stages of the computation. If "partition" then the (default) partition refinement algorithm of Robert Miller is used. Finally, if "codecan" then the partition refinement algorithm of Thomas Feulner is used, which also computes a canonical representative of self (call canonical_representative() to access it).

OUTPUT:

•Permutation automorphism group

EXAMPLES:

```
sage: MS = MatrixSpace(GF(2),4,8)
sage: G  =␣
↪MS([[1,0,0,0,1,1,1,0],[0,1,1,1,0,0,0,0],[0,0,0,0,0,0,0,1],[0,0,0,0,0,1,0,0]])
sage: C  = LinearCode(G)
sage: C
Linear code of length 8, dimension 4 over Finite Field of size 2
sage: G = C.permutation_automorphism_group()
sage: G.order()
144
sage: GG = C.permutation_automorphism_group("codecan")
sage: GG == G
True
```

A less easy example involves showing that the permutation automorphism group of the extended ternary Golay code is the Mathieu group M_{11}.

```
sage: C = codes.ExtendedTernaryGolayCode()
sage: M11 = MathieuGroup(11)
sage: M11.order()
7920
sage: G = C.permutation_automorphism_group()  # long time (6s on sage.math,␣
↪2011)
sage: G.is_isomorphic(M11)                    # long time
True
sage: GG = C.permutation_automorphism_group("codecan") # long time
sage: GG == G # long time
True
```

Other examples:

```
sage: C = codes.ExtendedBinaryGolayCode()
sage: G = C.permutation_automorphism_group()
sage: G.order()
244823040
sage: C = codes.HammingCode(GF(2), 5)
sage: G = C.permutation_automorphism_group()
sage: G.order()
9999360
sage: C = codes.HammingCode(GF(3), 2); C
[4, 2] Hamming Code over Finite Field of size 3
sage: C.permutation_automorphism_group(algorithm="partition")
Permutation Group with generators [(1,3,4)]
sage: C = codes.HammingCode(GF(4,"z"), 2); C
[5, 3] Hamming Code over Finite Field in z of size 2^2
sage: G = C.permutation_automorphism_group(algorithm="partition"); G
Permutation Group with generators [(1,3)(4,5), (1,4)(3,5)]
sage: GG = C.permutation_automorphism_group(algorithm="codecan") # long time
```

```
sage: GG == G # long time
True
sage: C.permutation_automorphism_group(algorithm="gap")  # optional - gap_
↪packages (Guava package)
Permutation Group with generators [(1,3)(4,5), (1,4)(3,5)]
sage: C = codes.TernaryGolayCode()
sage: C.permutation_automorphism_group(algorithm="gap")  # optional - gap_
↪packages (Guava package)
Permutation Group with generators [(3,4)(5,7)(6,9)(8,11),␣
↪(3,5,8)(4,11,7)(6,9,10), (2,3)(4,6)(5,8)(7,10), (1,2)(4,11)(5,8)(9,10)]
```

However, the option `algorithm="gap+verbose"` , will print out:

```
Minimum distance: 5 Weight distribution: [1, 0, 0, 0, 0, 132, 132,
0, 330, 110, 0, 24]

Using the 132 codewords of weight 5 Supergroup size: 39916800
```

in addition to the output of `C.permutation_automorphism_group(algorithm="gap")` .

permuted_code (*p*)

Returns the permuted code, which is equivalent to `self` via the column permutation `p` .

EXAMPLES:

```
sage: C = codes.HammingCode(GF(2), 3)
sage: G = C.permutation_automorphism_group(); G
Permutation Group with generators [(4,5)(6,7), (4,6)(5,7), (2,3)(6,7),␣
↪(2,4)(3,5), (1,2)(5,6)]
sage: g = G("(2,3)(6,7)")
sage: Cg = C.permuted_code(g)
sage: Cg
Linear code of length 7, dimension 4 over Finite Field of size 2
sage: C.generator_matrix() == Cg.systematic_generator_matrix()
True
```

punctured (*L*)

Returns a *sage.coding.punctured_code* object from `L` .

INPUT:

- `L` - List of positions to puncture

OUTPUT:

- an instance of *sage.coding.punctured_code*

EXAMPLES:

```
sage: C = codes.HammingCode(GF(2), 3)
sage: C.punctured([1,2])
Punctured code coming from [7, 4] Hamming Code over Finite Field of size 2␣
↪punctured on position(s) [1, 2]
```

random_element (**args*, ***kwds*)

Returns a random codeword; passes other positional and keyword arguments to `random_element()` method of vector space.

OUTPUT:

- Random element of the vector space of this code

EXAMPLES:

```
sage: C = codes.HammingCode(GF(4,'a'), 3)
sage: C.random_element() # random test
(1, 0, 0, a + 1, 1, a, a, a + 1, a + 1, 1, 1, 0, a + 1, a, 0, a, a, 0, a, a,␣
↪1)
```

Passes extra positional or keyword arguments through:

```
sage: C.random_element(prob=.5, distribution='1/n') # random test
(1, 0, a, 0, 0, 0, 0, a + 1, 0, 0, 0, 0, 0, 0, 0, 0, a + 1, a + 1, 1, 0, 0)
```

TESTS:

Test that the codeword returned is immutable (see trac ticket #16469):

```
sage: c = C.random_element()
sage: c.is_immutable()
True
```

Test that codeword returned has the same parent as any non-random codeword (see trac ticket #19653):

```
sage: C = codes.random_linear_code(GF(16, 'a'), 10, 4)
sage: c1 = C.random_element()
sage: c2 = C[1]
sage: c1.parent() == c2.parent()
True
```

rate ( )

Return the ratio of the number of information symbols to the code length.

EXAMPLES:

```
sage: C = codes.HammingCode(GF(2), 3)
sage: C.rate()
4/7
```

redundancy_matrix ( )

Returns the non-identity columns of a systematic generator matrix for `self` .

A systematic generator matrix is a generator matrix such that a subset of its columns forms the identity matrix. This method returns the remaining part of the matrix.

For any given code, there can be many systematic generator matrices (depending on which positions should form the identity). This method will use the matrix returned by *AbstractLinearCode.systematic_generator_matrix()* .

OUTPUT:

- An $k \times (n - k)$ matrix.

EXAMPLES:

```
sage: C = codes.HammingCode(GF(2), 3)
sage: C.generator_matrix()
 [1 0 0 0 0 1 1]
 [0 1 0 0 1 0 1]
 [0 0 1 0 1 1 0]
 [0 0 0 1 1 1 1]
sage: C.redundancy_matrix()
```

```
 [0 1 1]
 [1 0 1]
 [1 1 0]
 [1 1 1]
sage: C = LinearCode(matrix(GF(3),2,[1,2,0,\
                                     2,1,1]))
sage: C.systematic_generator_matrix()
[1 2 0]
[0 0 1]
sage: C.redundancy_matrix()
[2]
[0]
```

relative_distance ( )

Return the ratio of the minimum distance to the code length.

EXAMPLES:

```
sage: C = codes.HammingCode(GF(2),3)
sage: C.relative_distance()
3/7
```

sd_duursma_data (*i*, *warn=True*)

Compute two integers pertaining to the computation of the self-dual Duursma zeta function for `self` , if `self` is a self-dual code.

INPUT:

- `i` - Type number

OUTPUT:

- Pair `(v,m)` as in Duursma *[D]*

REFERENCES:

EXAMPLES:

```
sage: MS = MatrixSpace(GF(2),2,4)
sage: G = MS([1,1,0,0,0,0,1,1])
sage: C = LinearCode(G)
sage: C == C.dual_code()  # checks that C is self dual
True
sage: for i in [1,2,3,4]: print(C.sd_duursma_data(i))
doctest:...: DeprecationWarning: AbstractLinearCode.sd_duursma_data will be␣
↪removed in a future release of Sage. Please use AbstractLinearCode.zeta_
↪polynomial() to compute the Duursma zeta polynomial
See http://trac.sagemath.org/21165 for details.
(2, -1)
(2, -3)
(2, -2)
(2, -1)
```

sd_duursma_q (*i*, *d0*, *warn=True*)

Compute a polynomial pertaining to the computation of the self-dual Duursma zeta function for `self` , if `self` is a self-dual code.

INPUT:

- `i` - Type number, one of 1,2,3,4

•`d0` - Divisor, the smallest integer such that each $A_i > 0$ iff i is divisible by $d0$

OUTPUT:

•The polynomial $Q(T)$ as in Duursma *[D]*

EXAMPLES:

```
sage: C1 = codes.HammingCode(GF(2), 3)
sage: C2 = C1.extended_code(); C2
Extended code coming from [7, 4] Hamming Code over Finite Field of size 2
sage: C2.sd_duursma_q(1,1)
doctest:...: DeprecationWarning: AbstractLinearCode.sd_duursma_q will be␣
↪removed in a future release of Sage. Please use AbstractLinearCode.zeta_
↪polynomial() to compute the Duursma zeta polynomial
See http://trac.sagemath.org/21165 for details.
2/5*T^2 + 2/5*T + 1/5
sage: C2.sd_duursma_q(3,1)
3/5*T^4 + 1/5*T^3 + 1/15*T^2 + 1/15*T + 1/15
```

sd_zeta_polynomial (*typ=1*)

Return the Duursma zeta polynomial, computed in a fashion that only works if `self` is self-dual.

Warning: This function does not check that `self` is self-dual. Indeed, it is not even clear which notion of self-dual is supported ([D] seems to indicate formal self-dual, but the example below is a hexacode which is Hermitian self-dual).

INPUT:

•`typ` - Integer, type of this s.d. code; one of 1,2,3, or 4 (default: 1)

OUTPUT:

•Polynomial in a variable "T": the Duursma zeta function as in [D]

EXAMPLES:

```
sage: C1 = codes.HammingCode(GF(2), 3)
sage: C2 = C1.extended_code(); C2
Extended code coming from [7, 4] Hamming Code over Finite Field of size 2
sage: P = C2.sd_zeta_polynomial(); P
doctest:...: DeprecationWarning: AbstractLinearCode.sd_zeta_polynomial() will␣
↪be removed in a future release of Sage. Please use AbstractLinearCode.zeta_
↪polynomial() instead
See http://trac.sagemath.org/21165 for details.
2/5*T^2 + 2/5*T + 1/5
sage: P(1)
1
sage: F.<z> = GF(4,"z")
sage: MS = MatrixSpace(F, 3, 6)
sage: G = MS([[1,0,0,1,z,z],[0,1,0,z,1,z],[0,0,1,z,z,1]])
sage: C = LinearCode(G)  # the "hexacode"
sage: C.sd_zeta_polynomial(4)
1
```

shortened (L)

Returns the code shortened at the positions `L` , where $L \subset \{1, 2, ..., n\}$.

Consider the subcode $C(L)$ consisting of all codewords $c \in C$ which satisfy $c_i = 0$ for all $i \in L$. The punctured code $C(L)^L$ is called the shortened code on L and is denoted C_L. The code constructed is actually only isomorphic to the shortened code defined in this way.

By Theorem 1.5.7 in *[HP]*, C_L is $((C^\perp)^L)^\perp$. This is used in the construction below.

INPUT:

- `L` - Subset of $\{1, ..., n\}$, where n is the length of this code

OUTPUT:

- Linear code, the shortened code described above

EXAMPLES:

```
sage: C = codes.HammingCode(GF(2), 3)
sage: C.shortened([1,2])
Linear code of length 5, dimension 2 over Finite Field of size 2
```

spectrum (*algorithm=None*)

Returns the weight distribution, or spectrum, of `self` as a list.

The weight distribution a code of length n is the sequence $A_0, A_1, ..., A_n$ where A_i is the number of codewords of weight i.

INPUT:

- `algorithm` - (optional, default: `None`) If set to `"gap"` , call GAP. If set to *"leon"*, call the option GAP package GUAVA and call a function therein by Jeffrey Leon (see warning below). If set to `"binary"` , use an algorithm optimized for binary codes. The default is to use `"binary"` for binary codes and `"gap"` otherwise.

OUTPUT:

- A list of non-negative integers: the weight distribution.

Warning: Specifying `algorithm = "leon"` sometimes prints a traceback related to a stack smashing error in the C library. The result appears to be computed correctly, however. It appears to run much faster than the GAP algorithm in small examples and much slower than the GAP algorithm in larger examples.

EXAMPLES:

```
sage: MS = MatrixSpace(GF(2),4,7)
sage: G =␣
↪MS([[1,1,1,0,0,0,0],[1,0,0,1,1,0,0],[0,1,0,1,0,1,0],[1,1,0,1,0,0,1]])
sage: C = LinearCode(G)
sage: C.weight_distribution()
[1, 0, 0, 7, 7, 0, 0, 1]
sage: F.<z> = GF(2^2,"z")
sage: C = codes.HammingCode(F, 2); C
[5, 3] Hamming Code over Finite Field in z of size 2^2
sage: C.weight_distribution()
[1, 0, 0, 30, 15, 18]
sage: C = codes.HammingCode(GF(2), 3); C
[7, 4] Hamming Code over Finite Field of size 2
sage: C.weight_distribution(algorithm="leon")    # optional - gap_packages␣
↪(Guava package)
[1, 0, 0, 7, 7, 0, 0, 1]
```

```
sage: C.weight_distribution(algorithm="gap")
[1, 0, 0, 7, 7, 0, 0, 1]
sage: C.weight_distribution(algorithm="binary")
[1, 0, 0, 7, 7, 0, 0, 1]
sage: C = codes.HammingCode(GF(3), 3); C
[13, 10] Hamming Code over Finite Field of size 3
sage: C.weight_distribution() == C.weight_distribution(algorithm="leon")    #␣
↪optional - gap_packages (Guava package)
True
sage: C = codes.HammingCode(GF(5), 2); C
[6, 4] Hamming Code over Finite Field of size 5
sage: C.weight_distribution() == C.weight_distribution(algorithm="leon")    #␣
↪optional - gap_packages (Guava package)
True
sage: C = codes.HammingCode(GF(7), 2); C
[8, 6] Hamming Code over Finite Field of size 7
sage: C.weight_distribution() == C.weight_distribution(algorithm="leon")    #␣
↪optional - gap_packages (Guava package)
True
```

standard_form (*return_permutation=True*)

Returns a linear code which is permutation-equivalent to `self` and admits a generator matrix in standard form.

A generator matrix is in standard form if it is of the form $[I|A]$, where I is the $k \times k$ identity matrix. Any code admits a generator matrix in systematic form, i.e. where a subset of the columns form the identity matrix, but one might need to permute columns to allow the identity matrix to be leading.

INPUT:

- `return_permutation` – (default: `True`) if `True` , the column permutation which brings `self` into the returned code is also returned.

OUTPUT:

- A *LinearCode* whose *systematic_generator_matrix()* is guaranteed to be of the form $[I|A]$.

EXAMPLES:

```
sage: C = codes.HammingCode(GF(2), 3)
sage: C.generator_matrix()
[1 0 0 0 0 1 1]
[0 1 0 0 1 0 1]
[0 0 1 0 1 1 0]
[0 0 0 1 1 1 1]
sage: Cs,p = C.standard_form()
sage: p
[]
sage: Cs is C
True
sage: C = LinearCode(matrix(GF(2), [[1,0,0,0,1,1,0],\
                                    [0,1,0,1,0,1,0],\
                                    [0,0,0,0,0,0,1]]))
sage: Cs, p = C.standard_form()
sage: p
[1, 2, 7, 3, 4, 5, 6]
sage: Cs.generator_matrix()
[1 0 0 0 0 1 1]
```

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

support ( )

Returns the set of indices j where A_j is nonzero, where A_j is the number of codewords in $self$ of Hamming weight j.

OUTPUT:

•List of integers

EXAMPLES:

```
sage: C = codes.HammingCode(GF(2), 3)
sage: C.weight_distribution()
[1, 0, 0, 7, 7, 0, 0, 1]
sage: C.support()
[0, 3, 4, 7]
```

syndrome (*r*)

Returns the syndrome of `r` .

The syndrome of `r` is the result of $H \times r$ where H is the parity check matrix of `self` . If `r` belongs to `self` , its syndrome equals to the zero vector.

INPUT:

•`r` – a vector of the same length as `self`

OUTPUT:

•a column vector

EXAMPLES:

```
sage: MS = MatrixSpace(GF(2),4,7)
sage: G  = MS([[1,1,1,0,0,0,0], [1,0,0,1,1,0,0], [0,1,0,1,0,1,0],␣
↪[1,1,0,1,0,0,1]])
sage: C  = LinearCode(G)
sage: r = vector(GF(2), (1,0,1,0,1,0,1))
sage: r in C
True
sage: C.syndrome(r)
(0, 0, 0)
```

If `r` is not a codeword, its syndrome is not equal to zero:

```
sage: r = vector(GF(2), (1,0,1,0,1,1,1))
sage: r in C
False
sage: C.syndrome(r)
(0, 1, 1)
```

Syndrome computation works fine on bigger fields:

```
sage: C = codes.random_linear_code(GF(59), 12, 4)
sage: c = C.random_element()
sage: C.syndrome(c)
(0, 0, 0, 0, 0, 0, 0, 0)
```

systematic_generator_matrix (*systematic_positions=None*)
Return a systematic generator matrix of the code.

A generator matrix of a code is called systematic if it contains a set of columns forming an identity matrix.

INPUT:

- `systematic_positions` – (default: `None`) if supplied, the set of systematic positions in the systematic generator matrix. See the documentation for *LinearCodeSystematicEncoder* details.

EXAMPLES:

```
sage: G = matrix(GF(3), [[ 1, 2, 1, 0],                                [␣
↪2, 1, 1, 1]])
sage: C = LinearCode(G)
sage: C.generator_matrix()
[1 2 1 0]
[2 1 1 1]
sage: C.systematic_generator_matrix()
[1 2 0 1]
[0 0 1 2]
```

Specific systematic positions can also be requested:

sage: C.systematic_generator_matrix(systematic_positions=[3,2]) [1 2 0 1] [1 2 1 0]

unencode (*c*, *encoder_name=None*, *nocheck=False*, ***kwargs*)
Returns the message corresponding to `c` .

This is the inverse of *encode()* .

INPUT:

- `c` – a codeword of `self` .
- `encoder_name` – (default: `None`) name of the decoder which will be used to decode `word` . The default decoder of `self` will be used if default value is kept.
- `nocheck` – (default: `False`) checks if `c` is in `self` . You might set this to `True` to disable the check for saving computation. Note that if `c` is not in `self` and `nocheck = True` , then the output of *unencode()* is not defined (except that it will be in the message space of `self`).
- `kwargs` – all additional arguments are forwarded to the construction of the encoder that is used.

OUTPUT:

- an element of the message space of `encoder_name` of `self` .

EXAMPLES:

```
sage: G = Matrix(GF(2),␣
↪[[1,1,1,0,0,0,0],[1,0,0,1,1,0,0],[0,1,0,1,0,1,0],[1,1,0,1,0,0,1]])
sage: C = LinearCode(G)
sage: c = vector(GF(2), (1, 1, 0, 0, 1, 1, 0))
sage: C.unencode(c)
(0, 1, 1, 0)
```

weight_distribution (*algorithm=None*)
Returns the weight distribution, or spectrum, of `self` as a list.

The weight distribution a code of length n is the sequence $A_0, A_1, ..., A_n$ where A_i is the number of codewords of weight i.

INPUT:

- `algorithm` - (optional, default: `None`) If set to `"gap"`, call GAP. If set to "*leon*", call the option GAP package GUAVA and call a function therein by Jeffrey Leon (see warning below). If set to `"binary"`, use an algorithm optimized for binary codes. The default is to use `"binary"` for binary codes and `"gap"` otherwise.

OUTPUT:

- A list of non-negative integers: the weight distribution.

Warning: Specifying `algorithm = "leon"` sometimes prints a traceback related to a stack smashing error in the C library. The result appears to be computed correctly, however. It appears to run much faster than the GAP algorithm in small examples and much slower than the GAP algorithm in larger examples.

EXAMPLES:

```
sage: MS = MatrixSpace(GF(2),4,7)
sage: G =␣
↪MS([[1,1,1,0,0,0,0],[1,0,0,1,1,0,0],[0,1,0,1,0,1,0],[1,1,0,1,0,0,1]])
sage: C = LinearCode(G)
sage: C.weight_distribution()
[1, 0, 0, 7, 7, 0, 0, 1]
sage: F.<z> = GF(2^2,"z")
sage: C = codes.HammingCode(F, 2); C
[5, 3] Hamming Code over Finite Field in z of size 2^2
sage: C.weight_distribution()
[1, 0, 0, 30, 15, 18]
sage: C = codes.HammingCode(GF(2), 3); C
[7, 4] Hamming Code over Finite Field of size 2
sage: C.weight_distribution(algorithm="leon")   # optional - gap_packages␣
↪(Guava package)
[1, 0, 0, 7, 7, 0, 0, 1]
sage: C.weight_distribution(algorithm="gap")
[1, 0, 0, 7, 7, 0, 0, 1]
sage: C.weight_distribution(algorithm="binary")
[1, 0, 0, 7, 7, 0, 0, 1]
sage: C = codes.HammingCode(GF(3), 3); C
[13, 10] Hamming Code over Finite Field of size 3
sage: C.weight_distribution() == C.weight_distribution(algorithm="leon")   #␣
↪optional - gap_packages (Guava package)
True
sage: C = codes.HammingCode(GF(5), 2); C
[6, 4] Hamming Code over Finite Field of size 5
sage: C.weight_distribution() == C.weight_distribution(algorithm="leon")   #␣
↪optional - gap_packages (Guava package)
True
sage: C = codes.HammingCode(GF(7), 2); C
[8, 6] Hamming Code over Finite Field of size 7
sage: C.weight_distribution() == C.weight_distribution(algorithm="leon")   #␣
↪optional - gap_packages (Guava package)
True
```

weight_enumerator (*names=None*, *name2=None*, *bivariate=True*)

Return the weight enumerator polynomial of `self`.

This is the bivariate, homogeneous polynomial in x and y whose coefficient to $x^i y^{n-i}$ is the number of

codewords of $self$ of Hamming weight i. Here, n is the length of $self$.

INPUT:

- `names` - (default: `"xy"`) The names of the variables in the homogeneous polynomial. Can be given as a single string of length 2, or a single string with a comma, or as a tuple or list of two strings.
- `name2` - Deprecated, (default: `None`) The string name of the second variable.
- `bivariate` - (default: $True$) Whether to return a bivariate, homogeneous polynomial or just a univariate polynomial. If set to `False`, then `names` will be interpreted as a single variable name and default to `"x"`.

OUTPUT:

- The weight enumerator polynomial over $\mathbf{Z}$.

EXAMPLES:

```
sage: C = codes.HammingCode(GF(2), 3)
sage: C.weight_enumerator()
x^7 + 7*x^4*y^3 + 7*x^3*y^4 + y^7
sage: C.weight_enumerator(names="st")
s^7 + 7*s^4*t^3 + 7*s^3*t^4 + t^7
sage: C.weight_enumerator(names="var1, var2")
var1^7 + 7*var1^4*var2^3 + 7*var1^3*var2^4 + var2^7
sage: (var1, var2) = var('var1, var2')
sage: C.weight_enumerator(names=(var1, var2))
var1^7 + 7*var1^4*var2^3 + 7*var1^3*var2^4 + var2^7
sage: C.weight_enumerator(bivariate=False)
x^7 + 7*x^4 + 7*x^3 + 1
```

zero ( )

Returns the zero vector of `self`.

EXAMPLES:

```
sage: C = codes.HammingCode(GF(2), 3)
sage: C.zero()
(0, 0, 0, 0, 0, 0, 0)
sage: C.sum(()) # indirect doctest
(0, 0, 0, 0, 0, 0, 0)
sage: C.sum((C.gens())) # indirect doctest
(1, 1, 1, 1, 1, 1, 1)
```

zeta_function (*name='T'*)

Returns the Duursma zeta function of the code.

INPUT:

- `name` - String, variable name (default: `"T"`)

OUTPUT:

- Element of $\mathbf{Q}(T)$

EXAMPLES:

```
sage: C = codes.HammingCode(GF(2), 3)
sage: C.zeta_function()
(2/5*T^2 + 2/5*T + 1/5)/(2*T^2 - 3*T + 1)
```

zeta_polynomial (*name='T'*)

Returns the Duursma zeta polynomial of this code.

Assumes that the minimum distances of this code and its dual are greater than 1. Prints a warning to `stdout` otherwise.

INPUT:

- `name` - String, variable name (default: `"T"`)

OUTPUT:

- Polynomial over $\mathbf{Q}$

EXAMPLES:

```
sage: C = codes.HammingCode(GF(2), 3)
sage: C.zeta_polynomial()
2/5*T^2 + 2/5*T + 1/5
sage: C = codes.databases.best_linear_code_in_guava(6,3,GF(2))  # optional -␣
↪gap_packages (Guava package)
sage: C.minimum_distance()                    # optional - gap_packages (Guava␣
↪package)
3
sage: C.zeta_polynomial()                     # optional - gap_packages (Guava␣
↪package)
2/5*T^2 + 2/5*T + 1/5
sage: C = codes.HammingCode(GF(2), 4)
sage: C.zeta_polynomial()
16/429*T^6 + 16/143*T^5 + 80/429*T^4 + 32/143*T^3 + 30/143*T^2 + 2/13*T + 1/13
sage: F.<z> = GF(4,"z")
sage: MS = MatrixSpace(F, 3, 6)
sage: G = MS([[1,0,0,1,z,z],[0,1,0,z,1,z],[0,0,1,z,z,1]])
sage: C = LinearCode(G)  # the "hexacode"
sage: C.zeta_polynomial()
1
```

REFERENCES:

class sage.coding.linear_code. **LinearCode** (*generator*, *d=None*)

Bases: *sage.coding.linear_code.AbstractLinearCode*

Linear codes over a finite field or finite ring, represented using a generator matrix.

This class should be used for arbitrary and unstructured linear codes. This means that basic operations on the code, such as the computation of the minimum distance, will use generic, slow algorithms.

If you are looking for constructing a code from a more specific family, see if the family has been implemented by investigating *codes.* $<$ *tab* $>$. These more specific classes use properties particular to that family to allow faster algorithms, and could also have family-specific methods.

See Wikipedia article Linear_code for more information on unstructured linear codes.

INPUT:

- `generator` – a generator matrix over a finite field (`G` can be defined over a finite ring but the matrices over that ring must have certain attributes, such as `rank`); or a code over a finite field
- `d` – (optional, default: `None`) the minimum distance of the code

Note: The veracity of the minimum distance `d` , if provided, is not checked.

EXAMPLES:

```
sage: MS = MatrixSpace(GF(2),4,7)
sage: G  = MS([[1,1,1,0,0,0,0], [1,0,0,1,1,0,0], [0,1,0,1,0,1,0],␣
↪[1,1,0,1,0,0,1]])
sage: C  = LinearCode(G)
sage: C
Linear code of length 7, dimension 4 over Finite Field of size 2
sage: C.base_ring()
Finite Field of size 2
sage: C.dimension()
4
sage: C.length()
7
sage: C.minimum_distance()
3
sage: C.spectrum()
[1, 0, 0, 7, 7, 0, 0, 1]
sage: C.weight_distribution()
[1, 0, 0, 7, 7, 0, 0, 1]
```

The minimum distance of the code, if known, can be provided as an optional parameter.:

```
sage: C  = LinearCode(G, d=3)
sage: C.minimum_distance()
3
```

Another example.:

```
sage: MS = MatrixSpace(GF(5),4,7)
sage: G  = MS([[1,1,1,0,0,0,0], [1,0,0,1,1,0,0], [0,1,0,1,0,1,0],␣
↪[1,1,0,1,0,0,1]])
sage: C  = LinearCode(G)
sage: C
Linear code of length 7, dimension 4 over Finite Field of size 5
```

Providing a code as the parameter in order to "forget" its structure (see trac ticket #20198):

```
sage: C = codes.GeneralizedReedSolomonCode(GF(23).list(), 12)
sage: LinearCode(C)
Linear code of length 23, dimension 12 over Finite Field of size 23
```

Another example:

```
sage: C = codes.HammingCode(GF(7), 3)
sage: C
[57, 54] Hamming Code over Finite Field of size 7
sage: LinearCode(C)
Linear code of length 57, dimension 54 over Finite Field of size 7
```

AUTHORS:

- David Joyner (11-2005)
- Charles Prior (03-2016): trac ticket #20198, LinearCode from a code

generator_matrix (*encoder_name=None*, ***kwargs*)

Returns a generator matrix of `self` .

INPUT:

- encoder_name – (default: None) name of the encoder which will be used to compute the generator matrix. self._generator_matrix will be returned if default value is kept.
- kwargs – all additional arguments are forwarded to the construction of the encoder that is used.

EXAMPLES:

```
sage: G = matrix(GF(3),2,[1,-1,1,-1,1,1])
sage: code = LinearCode(G)
sage: code.generator_matrix()
[1 2 1]
[2 1 1]
```

sage.coding.linear_code. **LinearCodeFromVectorSpace** (*V*, *d=None*)

Simply converts a vector subspace V of $GF(q)^n$ into a $LinearCode$.

INPUT:

- V – The vector space
- d – (Optional, default: None) the minimum distance of the code, if known. This is an optional parameter.

Note: The veracity of the minimum distance d, if provided, is not checked.

EXAMPLES:

```
sage: V = VectorSpace(GF(2), 8)
sage: L = V.subspace([[1,1,1,1,0,0,0,0],[0,0,0,0,1,1,1,1]])
sage: C = LinearCodeFromVectorSpace(L)
doctest:...: DeprecationWarning: LinearCodeFromVectorSpace is deprecated. Simply␣
↪call LinearCode with your vector space instead.
See http://trac.sagemath.org/21165 for details.
sage: C.generator_matrix()
[1 1 1 1 0 0 0 0]
[0 0 0 0 1 1 1 1]
sage: C.minimum_distance()
4
```

Here, we provide the minimum distance of the code.:

```
sage: C = LinearCodeFromVectorSpace(L, d=4)
sage: C.minimum_distance()
4
```

class sage.coding.linear_code. **LinearCodeGeneratorMatrixEncoder** (*code*)

Bases: *sage.coding.encoder.Encoder*

Encoder based on generator_matrix for Linear codes.

This is the default encoder of a generic linear code, and should never be used for other codes than *LinearCode*.

INPUT:

- code – The associated *LinearCode* of this encoder.

generator_matrix ( )

Returns a generator matrix of the associated code of self.

EXAMPLES:

```
sage: G = Matrix(GF(2),␣
↪[[1,1,1,0,0,0,0],[1,0,0,1,1,0,0],[0,1,0,1,0,1,0],[1,1,0,1,0,0,1]])
sage: C = LinearCode(G)
sage: E = codes.encoders.LinearCodeGeneratorMatrixEncoder(C)
sage: E.generator_matrix()
[1 1 1 0 0 0 0]
[1 0 0 1 1 0 0]
[0 1 0 1 0 1 0]
[1 1 0 1 0 0 1]
```

class sage.coding.linear_code. **LinearCodeNearestNeighborDecoder** (*code*)

Bases: *sage.coding.decoder.Decoder*

Construct a decoder for Linear Codes. This decoder will decode to the nearest codeword found.

INPUT:

- code – A code associated to this decoder

decode_to_code (*r*)

Corrects the errors in word and returns a codeword.

INPUT:

- r – a codeword of self

OUTPUT:

- a vector of self 's message space

EXAMPLES:

```
sage: G = Matrix(GF(2),␣
↪[[1,1,1,0,0,0,0],[1,0,0,1,1,0,0],[0,1,0,1,0,1,0],[1,1,0,1,0,0,1]])
sage: C = LinearCode(G)
sage: D = codes.decoders.LinearCodeNearestNeighborDecoder(C)
sage: word = vector(GF(2), (1, 1, 0, 0, 1, 1, 0))
sage: w_err = word + vector(GF(2), (1, 0, 0, 0, 0, 0, 0))
sage: D.decode_to_code(w_err)
(1, 1, 0, 0, 1, 1, 0)
```

decoding_radius ( )

Return maximal number of errors self can decode.

EXAMPLES:

```
sage: G = Matrix(GF(2),␣
↪[[1,1,1,0,0,0,0],[1,0,0,1,1,0,0],[0,1,0,1,0,1,0],[1,1,0,1,0,0,1]])
sage: C = LinearCode(G)
sage: D = codes.decoders.LinearCodeNearestNeighborDecoder(C)
sage: D.decoding_radius()
1
```

class sage.coding.linear_code. **LinearCodeParityCheckEncoder** (*code*)

Bases: *sage.coding.encoder.Encoder*

Encoder based on parity_check_matrix() for Linear codes.

It constructs the generator matrix through the parity check matrix.

INPUT:

- code – The associated code of this encoder.

generator_matrix ( )

Returns a generator matrix of the associated code of `self` .

EXAMPLES:

```
sage: G = Matrix(GF(2),␣
↪[[1,1,1,0,0,0,0],[1,0,0,1,1,0,0],[0,1,0,1,0,1,0],[1,1,0,1,0,0,1]])
sage: C = LinearCode(G)
sage: E = codes.encoders.LinearCodeParityCheckEncoder(C)
doctest:...: DeprecationWarning: LinearCodeParityCheckEncoder is now␣
↪deprecated. Please use LinearCodeSystematicEncoder instead.
See http://trac.sagemath.org/20835 for details.
sage: E.generator_matrix()
[1 0 0 0 0 1 1]
[0 1 0 0 1 0 1]
[0 0 1 0 1 1 0]
[0 0 0 1 1 1 1]
```

class sage.coding.linear_code. **LinearCodeSyndromeDecoder** (*code*, *maximum_error_weight=None*)

Bases: *sage.coding.decoder.Decoder*

Constructs a decoder for Linear Codes based on syndrome lookup table.

The decoding algorithm works as follows:

- First, a lookup table is built by computing the syndrome of every error pattern of weight up to `maximum_error_weight` .
- Then, whenever one tries to decode a word `r` , the syndrome of `r` is computed. The corresponding error pattern is recovered from the pre-computed lookup table.
- Finally, the recovered error pattern is subtracted from `r` to recover the original word.

`maximum_error_weight` need never exceed the covering radius of the code, since there are then always lower-weight errors with the same syndrome. If one sets `maximum_error_weight` to a value greater than the covering radius, then the covering radius will be determined while building the lookup-table. This lower value is then returned if you query `decoding_radius` after construction.

If `maximum_error_weight` is left unspecified or set to a number at least the covering radius of the code, this decoder is complete, i.e. it decodes every vector in the ambient space.

Constructing the lookup table takes time exponential in the length of the code and the size of the code's base field. Afterwards, the individual decodings are fast.

INPUT:

- `code` – A code associated to this decoder
- `maximum_error_weight` – (default: `None`) the maximum number of errors to look for when building the table. An error is raised if it is set greater than $n - k$, since this is an upper bound on the covering radius on any linear code. If `maximum_error_weight` is kept unspecified, it will be set to $n - k$, where n is the length of `code` and k its dimension.

EXAMPLES:

```
sage: G = Matrix(GF(3),␣
↪[[1,0,0,1,0,1,0,1,2],[0,1,0,2,2,0,1,1,0],[0,0,1,0,2,2,2,1,2]])
sage: C = LinearCode(G)
sage: D = codes.decoders.LinearCodeSyndromeDecoder(C)
sage: D
Syndrome decoder for Linear code of length 9, dimension 3 over Finite Field of␣
↪size 3 handling errors of weight up to 4
```

If one wants to correct up to a lower number of errors, one can do as follows:

```
sage: D = codes.decoders.LinearCodeSyndromeDecoder(C, maximum_error_weight=2)
sage: D
Syndrome decoder for Linear code of length 9, dimension 3 over Finite Field of␣
↪size 3 handling errors of weight up to 2
```

If one checks the list of types of this decoder before constructing it, one will notice it contains the keyword `dynamic` . Indeed, the behaviour of the syndrome decoder depends on the maximum error weight one wants to handle, and how it compares to the minimum distance and the covering radius of `code` . In the following examples, we illustrate this property by computing different instances of syndrome decoder for the same code.

We choose the following linear code, whose covering radius equals to 4 and minimum distance to 5 (half the minimum distance is 2):

```
sage: G = matrix(GF(5), [[1, 0, 0, 0, 0, 4, 3, 0, 3, 1, 0],
....:                    [0, 1, 0, 0, 0, 3, 2, 2, 3, 2, 1],
....:                    [0, 0, 1, 0, 0, 1, 3, 0, 1, 4, 1],
....:                    [0, 0, 0, 1, 0, 3, 4, 2, 2, 3, 3],
....:                    [0, 0, 0, 0, 1, 4, 2, 3, 2, 2, 1]])
sage: C = LinearCode(G)
```

In the following examples, we illustrate how the choice of `maximum_error_weight` influences the types of the instance of syndrome decoder, alongside with its decoding radius.

We build a first syndrome decoder, and pick a `maximum_error_weight` smaller than both the covering radius and half the minimum distance:

```
sage: D = C.decoder("Syndrome", maximum_error_weight = 1)
sage: D.decoder_type()
{'always-succeed', 'bounded_distance', 'hard-decision', 'unique'}
sage: D.decoding_radius()
1
```

In that case, we are sure the decoder will always succeed. It is also a bounded distance decoder.

We now build another syndrome decoder, and this time, `maximum_error_weight` is chosen to be bigger than half the minimum distance, but lower than the covering radius:

```
sage: D = C.decoder("Syndrome", maximum_error_weight = 3)
sage: D.decoder_type()
{'bounded_distance', 'hard-decision', 'might-error', 'unique'}
sage: D.decoding_radius()
3
```

Here, we still get a bounded distance decoder. But because we have a maximum error weight bigger than half the minimum distance, we know it might return a codeword which was not the original codeword.

And now, we build a third syndrome decoder, whose `maximum_error_weight` is bigger than both the covering radius and half the minimum distance:

```
sage: D = C.decoder("Syndrome", maximum_error_weight = 5)
sage: D.decoder_type()
{'complete', 'hard-decision', 'might-error', 'unique'}
sage: D.decoding_radius()
4
```

In that case, the decoder might still return an unexpected codeword, but it is now complete. Note the decoding radius is equal to 4: it was determined while building the syndrome lookup table that any error with weight more than 4 will be decoded incorrectly. That is because the covering radius for the code is 4.

The minimum distance and the covering radius are both determined while computing the syndrome lookup table. They user did not explicitly ask to compute these on the code `C` . The dynamic typing of the syndrome decoder might therefore seem slightly surprising, but in the end is quite informative.

decode_to_code (*r*)

Corrects the errors in `word` and returns a codeword.

INPUT:

- `r` – a codeword of `self`

OUTPUT:

- a vector of `self` 's message space

EXAMPLES:

```
sage: G = Matrix(GF(3),[
....:    [1, 0, 0, 0, 2, 2, 1, 1],
....:    [0, 1, 0, 0, 0, 0, 1, 1],
....:    [0, 0, 1, 0, 2, 0, 0, 2],
....:    [0, 0, 0, 1, 0, 2, 0, 1]])
sage: C = LinearCode(G)
sage: D = codes.decoders.LinearCodeSyndromeDecoder(C, maximum_error_weight =␣
↪2)
sage: Chan = channels.StaticErrorRateChannel(C.ambient_space(), 2)
sage: c = C.random_element()
sage: r = Chan(c)
sage: c == D.decode_to_code(r)
True
```

decoding_radius ( )

Returns the maximal number of errors a received word can have and for which `self` is guaranteed to return a most likely codeword.

EXAMPLES:

```
sage: G = Matrix(GF(3),␣
↪[[1,0,0,1,0,1,0,1,2],[0,1,0,2,2,0,1,1,0],[0,0,1,0,2,2,2,1,2]])
sage: C = LinearCode(G)
sage: D = codes.decoders.LinearCodeSyndromeDecoder(C)
sage: D.decoding_radius()
4
```

maximum_error_weight ( )

Returns the maximal number of errors a received word can have and for which `self` is guaranteed to return a most likely codeword.

Same as `self.decoding_radius` .

EXAMPLES:

```
sage: G = Matrix(GF(3),␣
↪[[1,0,0,1,0,1,0,1,2],[0,1,0,2,2,0,1,1,0],[0,0,1,0,2,2,2,1,2]])
sage: C = LinearCode(G)
sage: D = codes.decoders.LinearCodeSyndromeDecoder(C)
```

```
sage: D.maximum_error_weight()
4
```

syndrome_table ( )

Returns the syndrome lookup table of `self` .

EXAMPLES:

```
sage: G = Matrix(GF(2),␣
↪[[1,1,1,0,0,0,0],[1,0,0,1,1,0,0],[0,1,0,1,0,1,0],[1,1,0,1,0,0,1]])
sage: C = LinearCode(G)
sage: D = codes.decoders.LinearCodeSyndromeDecoder(C)
sage: D.syndrome_table()
{(0, 0, 0): (0, 0, 0, 0, 0, 0, 0),
 (1, 0, 0): (1, 0, 0, 0, 0, 0, 0),
 (0, 1, 0): (0, 1, 0, 0, 0, 0, 0),
 (1, 1, 0): (0, 0, 1, 0, 0, 0, 0),
 (0, 0, 1): (0, 0, 0, 1, 0, 0, 0),
 (1, 0, 1): (0, 0, 0, 0, 1, 0, 0),
 (0, 1, 1): (0, 0, 0, 0, 0, 1, 0),
 (1, 1, 1): (0, 0, 0, 0, 0, 0, 1)}
```

class sage.coding.linear_code. **LinearCodeSystematicEncoder** (*code*, *systematic_positions=None*)

Bases: *sage.coding.encoder.Encoder*

Encoder based on a generator matrix in systematic form for Linear codes.

To encode an element of its message space, this encoder first builds a generator matrix in systematic form. What is called systematic form here is the reduced row echelon form of a matrix, which is not necessarily $[I|H]$, where I is the identity block and H the parity block. One can refer to *LinearCodeSystematicEncoder.generator_matrix()* for a concrete example. Once such a matrix has been computed, it is used to encode any message into a codeword.

This encoder can also serve as the default encoder of a code defined by a parity check matrix: if the *LinearCodeSystematicEncoder* detects that it is the default encoder, it computes a generator matrix as the reduced row echelon form of the right kernel of the parity check matrix.

INPUT:

- `code` – The associated code of this encoder.
- `systematic_positions` – (default: `None`) the positions in codewords that should correspond to the message symbols. A list of k distinct integers in the range 0 to $n-1$ where n is the length of the code and k its dimension. The 0th symbol of a message will then be at position `systematic_positions[0]` , the 1st index at position `systematic_positions[1]` , etc. A `ValueError` is raised at construction time if the supplied indices do not form an information set.

EXAMPLES:

The following demonstrates the basic usage of *LinearCodeSystematicEncoder* :

```
sage: G = Matrix(GF(2), [[1,1,1,0,0,0,0,0],\
                         [1,0,0,1,1,0,0,0],\
                         [0,1,0,1,0,1,0,0],\
                         [1,1,0,1,0,0,1,1]])
sage: C = LinearCode(G)
sage: E = codes.encoders.LinearCodeSystematicEncoder(C)
sage: E.generator_matrix()
[1 0 0 0 0 1 1 1]
```

```
[0 1 0 0 1 0 1 1]
[0 0 1 0 1 1 0 0]
[0 0 0 1 1 1 1 1]
sage: E2 = codes.encoders.LinearCodeSystematicEncoder(C, systematic_
↪positions=[5,4,3,2])
sage: E2.generator_matrix()
[1 0 0 0 0 1 1 1]
[0 1 0 0 1 0 1 1]
[1 1 0 1 0 0 1 1]
[1 1 1 0 0 0 0 0]
```

An error is raised if one specifies systematic positions which do not form an information set:

```
sage: E3 = codes.encoders.LinearCodeSystematicEncoder(C, systematic_
↪positions=[0,1,6,7])
Traceback (most recent call last):
...
ValueError: systematic_positions are not an information set
```

We exemplify how to use *LinearCodeSystematicEncoder* as the default encoder. The following class is the dual of the repetition code:

```
sage: class DualRepetitionCode(sage.coding.linear_code.AbstractLinearCode):
....:     def __init__(self, field, length):
....:         sage.coding.linear_code.AbstractLinearCode.__init__(self,field,␣
↪length, "Systematic", "Syndrome")
....:
....:     def parity_check_matrix(self):
....:         return Matrix(self.base_field(), [1]*self.length())
....:
....:     def _repr_(self):
....:         return "Dual of the Repetition Code of length %d over %s" % (self.
↪length(), self.base_field())
....:
sage: DualRepetitionCode(GF(3), 5).generator_matrix()
[1 0 0 0 2]
[0 1 0 0 2]
[0 0 1 0 2]
[0 0 0 1 2]
```

An exception is thrown if *LinearCodeSystematicEncoder* is the default encoder but no parity check matrix has been specified for the code:

```
sage: class BadCodeExample(sage.coding.linear_code.AbstractLinearCode):
....:     def __init__(self, field, length):
....:         sage.coding.linear_code.AbstractLinearCode.__init__(self, field,␣
↪length, "Systematic", "Syndrome")
....:
....:     def _repr_(self):
....:         return "I am a badly defined code"
....:
sage: BadCodeExample(GF(3), 5).generator_matrix()
Traceback (most recent call last):
...
ValueError: a parity check matrix must be specified if␣
↪LinearCodeSystematicEncoder is the default encoder
```

generator_matrix ( )

Returns a generator matrix in systematic form of the associated code of `self` .

Systematic form here means that a subsets of the columns of the matrix forms the identity matrix.

Note: The matrix returned by this method will not necessarily be $[I|H]$, where I is the identity block and H the parity block. If one wants to know which columns create the identity block, one can call *systematic_positions()*

EXAMPLES:

```
sage: G = Matrix(GF(2), [[1,1,1,0,0,0,0],\
                         [1,0,0,1,1,0,0],\
                         [0,1,0,1,0,1,0],\
                         [1,1,0,1,0,0,1]])
sage: C = LinearCode(G)
sage: E = codes.encoders.LinearCodeSystematicEncoder(C)
sage: E.generator_matrix()
[1 0 0 0 0 1 1]
[0 1 0 0 1 0 1]
[0 0 1 0 1 1 0]
[0 0 0 1 1 1 1]
```

We can ask for different systematic positions:

```
sage: E2 = codes.encoders.LinearCodeSystematicEncoder(C, systematic_
↪positions=[5,4,3,2])
sage: E2.generator_matrix()
[1 0 0 0 0 1 1]
[0 1 0 0 1 0 1]
[1 1 0 1 0 0 1]
[1 1 1 0 0 0 0]
```

Another example where there is no generator matrix of the form $[I|H]$:

```
sage: G = Matrix(GF(2), [[1,1,0,0,1,0,1],\
                         [1,1,0,0,1,0,0],\
                         [0,0,1,0,0,1,0],\
                         [0,0,1,0,1,0,1]])
sage: C = LinearCode(G)
sage: E = codes.encoders.LinearCodeSystematicEncoder(C)
sage: E.generator_matrix()
[1 1 0 0 0 1 0]
[0 0 1 0 0 1 0]
[0 0 0 0 1 1 0]
[0 0 0 0 0 0 1]
```

systematic_permutation ( )

Returns a permutation which would take the systematic positions into [0,..,k-1]

EXAMPLES:

```
sage: C = LinearCode(matrix(GF(2), [[1,0,0,0,1,1,0],\
                                    [0,1,0,1,0,1,0],\
                                    [0,0,0,0,0,0,1]]))
sage: E = codes.encoders.LinearCodeSystematicEncoder(C)
sage: E.systematic_positions()
(0, 1, 6)
```

```
sage: E.systematic_permutation()
[1, 2, 7, 3, 4, 5, 6]
```

systematic_positions ( )

Returns a tuple containing the indices of the columns which form an identity matrix when the generator matrix is in systematic form.

EXAMPLES:

```
sage: G = Matrix(GF(2), [[1,1,1,0,0,0,0],\
                         [1,0,0,1,1,0,0],\
                         [0,1,0,1,0,1,0],\
                         [1,1,0,1,0,0,1]])
sage: C = LinearCode(G)
sage: E = codes.encoders.LinearCodeSystematicEncoder(C)
sage: E.systematic_positions()
(0, 1, 2, 3)
```

We take another matrix with a less nice shape:

```
sage: G = Matrix(GF(2), [[1,1,0,0,1,0,1],\
                         [1,1,0,0,1,0,0],\
                         [0,0,1,0,0,1,0],\
                         [0,0,1,0,1,0,1]])
sage: C = LinearCode(G)
sage: E = codes.encoders.LinearCodeSystematicEncoder(C)
sage: E.systematic_positions()
(0, 2, 4, 6)
```

The systematic positions correspond to the positions which carry information in a codeword:

```
sage: MS = E.message_space()
sage: m = MS.random_element()
sage: c = m * E.generator_matrix()
sage: pos = E.systematic_positions()
sage: info = MS([c[i] for i in pos])
sage: m == info
True
```

When constructing a systematic encoder with specific systematic positions, then it is guaranteed that this method returns exactly those positions (even if another choice might also be systematic):

```
sage: G = Matrix(GF(2), [[1,0,0,0],\
                         [0,1,0,0],\
                         [0,0,1,1]])
sage: C = LinearCode(G)
sage: E = codes.encoders.LinearCodeSystematicEncoder(C, systematic_
↪positions=[0,1,3])
sage: E.systematic_positions()
(0, 1, 3)
```

sage.coding.linear_code. **wtdist_gap** (*Gmat*, *n*, *F*)

2.2 Generalized Reed-Solomon code

Given n different evaluation points $\alpha_1, \ldots, \alpha_n$ from some finite field F, and n column multipliers $\beta_1, \ldots, \beta_n$, the corresponding GRS code of dimension k is the set:

$$\{(\beta_1 f(\alpha_1), \ldots, \beta_n f(\alpha_n) \mid f \in F[x], \deg f < k\}$$

This file contains the following elements:

- *GeneralizedReedSolomonCode* , the class for GRS codes
- *GRSEvaluationVectorEncoder* , an encoder with a vectorial message space
- *GRSEvaluationPolynomialEncoder* , an encoder with a polynomial message space
- *GRSBerlekampWelchDecoder* , a decoder which corrects errors using Berlekamp-Welch algorithm
- *GRSGaoDecoder* , a decoder which corrects errors using Gao algorithm
- *GRSErrorErasureDecoder* , a decoder which corrects both errors and erasures
- *GRSKeyEquationSyndromeDecoder* , a decoder which corrects errors using the key equation on syndrome polynomials

class sage.coding.grs. **GRSBerlekampWelchDecoder** (*code*)
Bases: *sage.coding.decoder.Decoder*

Decoder for Generalized Reed-Solomon codes which uses Berlekamp-Welch decoding algorithm to correct errors in codewords.

This algorithm recovers the error locator polynomial by solving a linear system. See *[HJ04]* pp. 51-52 for details.

REFERENCES:

INPUT:

•`code` – A code associated to this decoder

EXAMPLES:

```
sage: F = GF(59)
sage: n, k = 40, 12
sage: C = codes.GeneralizedReedSolomonCode(F.list()[:n], k)
sage: D = codes.decoders.GRSBerlekampWelchDecoder(C)
sage: D
Berlekamp-Welch decoder for [40, 12, 29] Generalized Reed-Solomon Code over␣
↪Finite Field of size 59
```

Actually, we can construct the decoder from `C` directly:

```
sage: D = C.decoder("BerlekampWelch")
sage: D
Berlekamp-Welch decoder for [40, 12, 29] Generalized Reed-Solomon Code over␣
↪Finite Field of size 59
```

decode_to_code (*r*)
Corrects the errors in `r` and returns a codeword.

Note: If the code associated to `self` has the same length as its dimension, `r` will be returned as is.

INPUT:

- `r` – a vector of the ambient space of `self.code()`

OUTPUT:

- a vector of `self.code()`

EXAMPLES:

```
sage: F = GF(59)
sage: n, k = 40, 12
sage: C = codes.GeneralizedReedSolomonCode(F.list()[:n], k)
sage: D = codes.decoders.GRSBerlekampWelchDecoder(C)
sage: c = C.random_element()
sage: Chan = channels.StaticErrorRateChannel(C.ambient_space(), D.decoding_
↪radius())
sage: y = Chan(c)
sage: c == D.decode_to_code(y)
True
```

TESTS:

If one tries to decode a word which is too far from any codeword, an exception is raised:

```
sage: e = vector(F,[0, 0, 54, 23, 1, 0, 0, 0, 53, 21, 0, 0, 0, 34, 6, 11, 0,␣
↪0, 16, 0, 0, 0, 9, 0, 10, 27, 35, 0, 0, 0, 0, 46, 0, 0, 0, 0, 0, 0, 44, 0]);
↪ e.hamming_weight()
15
sage: D.decode_to_code(c + e)
Traceback (most recent call last):
...
DecodingError: Decoding failed because the number of errors exceeded the␣
↪decoding radius
```

If one tries to decode something which is not in the ambient space of the code, an exception is raised:

```
sage: D.decode_to_code(42)
Traceback (most recent call last):
...
ValueError: The word to decode has to be in the ambient space of the code
```

The bug detailed in trac ticket #20340 has been fixed:

```
sage: C = codes.GeneralizedReedSolomonCode(GF(59).list()[:40], 12)
sage: c = C.random_element()
sage: D = C.decoder("BerlekampWelch")
sage: D.decode_to_code(c) == c
True
```

decode_to_message (*r*)

Decodes `r` to an element in message space of `self`.

Note: If the code associated to `self` has the same length as its dimension, `r` will be unencoded as is. In that case, if `r` is not a codeword, the output is unspecified.

INPUT:

- `r` – a codeword of `self`

OUTPUT:

- a vector of `self` message space

EXAMPLES:

```
sage: F = GF(59)
sage: n, k = 40, 12
sage: C = codes.GeneralizedReedSolomonCode(F.list()[:n], k)
sage: D = codes.decoders.GRSBerlekampWelchDecoder(C)
sage: c = C.random_element()
sage: Chan = channels.StaticErrorRateChannel(C.ambient_space(), D.decoding_
↪radius())
sage: y = Chan(c)
sage: D.connected_encoder().unencode(c) == D.decode_to_message(y)
True
```

TESTS:

If one tries to decode a word which is too far from any codeword, an exception is raised:

```
sage: e = vector(F,[0, 0, 54, 23, 1, 0, 0, 0, 53, 21, 0, 0, 0, 34, 6, 11, 0,␣
↪0, 16, 0, 0, 0, 9, 0, 10, 27, 35, 0, 0, 0, 0, 46, 0, 0, 0, 0, 0, 0, 44, 0]);
↪ e.hamming_weight()
15
sage: D.decode_to_message(c + e)
Traceback (most recent call last):
...
DecodingError: Decoding failed because the number of errors exceeded the␣
↪decoding radius
```

If one tries to decode something which is not in the ambient space of the code, an exception is raised:

```
sage: D.decode_to_message(42)
Traceback (most recent call last):
...
ValueError: The word to decode has to be in the ambient space of the code
```

The bug detailed in trac ticket #20340 has been fixed:

```
sage: C = codes.GeneralizedReedSolomonCode(GF(59).list()[:40], 12)
sage: c = C.random_element()
sage: D = C.decoder("BerlekampWelch")
sage: E = D.connected_encoder()
sage: m = E.message_space().random_element()
sage: c = E.encode(m)
sage: D.decode_to_message(c) == m
True
```

decoding_radius ( )

Returns maximal number of errors that `self` can decode.

OUTPUT:

- the number of errors as an integer

EXAMPLES:

```
sage: F = GF(59)
sage: n, k = 40, 12
sage: C = codes.GeneralizedReedSolomonCode(F.list()[:n], k)
```

```
sage: D = codes.decoders.GRSBerlekampWelchDecoder(C)
sage: D.decoding_radius()
14
```

class sage.coding.grs. **GRSErrorErasureDecoder** (*code*)

Bases: *sage.coding.decoder.Decoder*

Decoder for Generalized Reed-Solomon codes which is able to correct both errors and erasures in codewords.

INPUT:

- `code` – The associated code of this decoder.

EXAMPLES:

```
sage: F = GF(59)
sage: n, k = 40, 12
sage: C = codes.GeneralizedReedSolomonCode(F.list()[:n], k)
sage: D = codes.decoders.GRSErrorErasureDecoder(C)
sage: D
Error-Erasure decoder for [40, 12, 29] Generalized Reed-Solomon Code over Finite␣
↪Field of size 59
```

Actually, we can construct the decoder from `C` directly:

```
sage: D = C.decoder("ErrorErasure")
sage: D
Error-Erasure decoder for [40, 12, 29] Generalized Reed-Solomon Code over Finite␣
↪Field of size 59
```

decode_to_message (*word_and_erasure_vector*)

Decode `word_and_erasure_vector` to an element in message space of `self`

INPUT:

- word_and_erasure_vector – a tuple whose: - first element is an element of the ambient space of the code - second element is a vector over GF(2) whose length is the same as the code's

---

Note: If the code associated to `self` has the same length as its dimension, `r` will be unencoded as is. If the number of erasures is exactly $n - k$, where n is the length of the code associated to `self` and k its dimension, `r` will be returned as is. In either case, if `r` is not a codeword, the output is unspecified.

INPUT:

- `word_and_erasure_vector` – a pair of vectors, where first element is a codeword of `self` and second element is a vector of GF(2) containing erasure positions

OUTPUT:

- a vector of `self` message space

EXAMPLES:

```
sage: F = GF(59)
sage: n, k = 40, 12
sage: C = codes.GeneralizedReedSolomonCode(F.list()[:n], k)
sage: D = codes.decoders.GRSErrorErasureDecoder(C)
sage: c = C.random_element()
sage: n_era = randint(0, C.minimum_distance() - 2)
```

```
sage: Chan = channels.ErrorErasureChannel(C.ambient_space(), D.decoding_
↪radius(n_era), n_era)
sage: y = Chan(c)
sage: D.connected_encoder().unencode(c) == D.decode_to_message(y)
True
```

TESTS:

If one tries to decode a word with too many erasures, it returns an exception:

```
sage: Chan = channels.ErrorErasureChannel(C.ambient_space(), 0, C.minimum_
↪distance() + 1)
sage: y = Chan(c)
sage: D.decode_to_message(y)
Traceback (most recent call last):
...
DecodingError: Too many erasures in the received word
```

If one tries to decode something which is not in the ambient space of the code, an exception is raised:

```
sage: D.decode_to_message((42, random_vector(GF(2), C.length())))
Traceback (most recent call last):
...
ValueError: The word to decode has to be in the ambient space of the code
```

If one tries to pass an erasure_vector which is not a vector over GF(2) of the same length as code's, an exception is raised:

```
sage: D.decode_to_message((C.random_element(), 42))
Traceback (most recent call last):
...
ValueError: The erasure vector has to be a vector over GF(2) of the same␣
↪length as the code
```

decoding_radius (*number_erasures*)

Return maximal number of errors that `self` can decode according to how many erasures it receives

INPUT:

- `number_erasures` – the number of erasures when we try to decode

OUTPUT:

- the number of errors as an integer

EXAMPLES:

```
sage: F = GF(59)
sage: n, k = 40, 12
sage: C = codes.GeneralizedReedSolomonCode(F.list()[:n], k)
sage: D = codes.decoders.GRSErrorErasureDecoder(C)
sage: D.decoding_radius(5)
11
```

If we receive too many erasures, it returns an exception as codeword will be impossible to decode:

```
sage: D.decoding_radius(30)
Traceback (most recent call last):
...
ValueError: The number of erasures exceed decoding capability
```

class sage.coding.grs. **GRSEvaluationPolynomialEncoder** (*code*, *polynomial_ring=None*)
Bases: *sage.coding.encoder.Encoder*

Encoder for Generalized Reed-Solomon codes which uses evaluation of polynomials to obtain codewords.

INPUT:

- code – The associated code of this encoder.
- polynomial_ring – (default: None) A polynomial ring to specify the message space of self , if needed. It is set to $F[x]$ (where F is the base field of code) if default value is kept.

EXAMPLES:

```
sage: F = GF(59)
sage: n, k = 40, 12
sage: C = codes.GeneralizedReedSolomonCode(F.list()[:n], k)
sage: E = codes.encoders.GRSEvaluationPolynomialEncoder(C)
sage: E
Evaluation polynomial-style encoder for [40, 12, 29] Generalized Reed-Solomon␣
↪Code over Finite Field of size 59
sage: E.message_space()
Univariate Polynomial Ring in x over Finite Field of size 59
```

Actually, we can construct the encoder from C directly:

```
sage: E = C.encoder("EvaluationPolynomial")
sage: E
Evaluation polynomial-style encoder for [40, 12, 29] Generalized Reed-Solomon␣
↪Code over Finite Field of size 59
```

We can also specify another polynomial ring:

```
sage: R = PolynomialRing(F, 'y')
sage: E = C.encoder("EvaluationPolynomial", polynomial_ring=R)
sage: E.message_space()
Univariate Polynomial Ring in y over Finite Field of size 59
```

encode (*p*)
Transforms the polynomial p into a codeword of code() .

INPUT:

- p – A polynomial from the message space of self of degree less than self.code().dimension() .

OUTPUT:

- A codeword in associated code of self

EXAMPLES:

```
sage: F = GF(11)
sage: Fx.<x> = F[]
sage: n, k = 10 , 5
sage: C = codes.GeneralizedReedSolomonCode(F.list()[:n], k)
sage: E = C.encoder("EvaluationPolynomial")
sage: p = x^2 + 3*x + 10
sage: c = E.encode(p); c
(10, 3, 9, 6, 5, 6, 9, 3, 10, 8)
sage: c in C
True
```

If a polynomial of too high degree is given, an error is raised:

```
sage: p = x^10
sage: E.encode(p)
Traceback (most recent call last):
...
ValueError: The polynomial to encode must have degree at most 4
```

If `p` is not an element of the proper polynomial ring, an error is raised:

```
sage: Qy.<y> = QQ[]
sage: p = y^2 + 1
sage: E.encode(p)
Traceback (most recent call last):
...
ValueError: The value to encode must be in Univariate Polynomial Ring in x␣
↪over Finite Field of size 11
```

TESTS:

The bug described in trac ticket #20744 is now fixed:

```
sage: F = GF(11)
sage: Fm.<my_variable> = F[]
sage: n, k = 10 , 5
sage: C = codes.GeneralizedReedSolomonCode(F.list()[:n], k)
sage: E = C.encoder("EvaluationPolynomial", polynomial_ring = Fm)
sage: p = my_variable^2 + 3*my_variable + 10
sage: c = E.encode(p)
sage: c in C
True
```

message_space ( )

Returns the message space of `self`

EXAMPLES:

```
sage: F = GF(11)
sage: n, k = 10 , 5
sage: C = codes.GeneralizedReedSolomonCode(F.list()[:n], k)
sage: E = C.encoder("EvaluationPolynomial")
sage: E.message_space()
Univariate Polynomial Ring in x over Finite Field of size 11
```

polynomial_ring ( )

Returns the message space of `self`

EXAMPLES:

```
sage: F = GF(11)
sage: n, k = 10 , 5
sage: C = codes.GeneralizedReedSolomonCode(F.list()[:n], k)
sage: E = C.encoder("EvaluationPolynomial")
sage: E.message_space()
Univariate Polynomial Ring in x over Finite Field of size 11
```

unencode_nocheck (*c*)

Returns the message corresponding to the codeword `c` .

Use this method with caution: it does not check if `c` belongs to the code, and if this is not the case, the output is unspecified. Instead, use `unencode()`.

INPUT:

- `c` – A codeword of `code()`.

OUTPUT:

- An polynomial of degree less than `self.code().dimension()`.

EXAMPLES:

```
sage: F = GF(11)
sage: n, k = 10 , 5
sage: C = codes.GeneralizedReedSolomonCode(F.list()[:n], k)
sage: E = C.encoder("EvaluationPolynomial")
sage: c = vector(F, (10, 3, 9, 6, 5, 6, 9, 3, 10, 8))
sage: c in C
True
sage: p = E.unencode_nocheck(c); p
x^2 + 3*x + 10
sage: E.encode(p) == c
True
```

Note that no error is thrown if `c` is not a codeword, and that the result is undefined:

```
sage: c = vector(F, (11, 3, 9, 6, 5, 6, 9, 3, 10, 8))
sage: c in C
False
sage: p = E.unencode_nocheck(c); p
6*x^4 + 6*x^3 + 2*x^2
sage: E.encode(p) == c
False
```

class sage.coding.grs. **GRSEvaluationVectorEncoder** (*code*)

Bases: *sage.coding.encoder.Encoder*

Encoder for Generalized Reed-Solomon codes which encodes vectors into codewords.

INPUT:

- `code` – The associated code of this encoder.

EXAMPLES:

```
sage: F = GF(59)
sage: n, k = 40, 12
sage: C = codes.GeneralizedReedSolomonCode(F.list()[:n], k)
sage: E = codes.encoders.GRSEvaluationVectorEncoder(C)
sage: E
Evaluation vector-style encoder for [40, 12, 29] Generalized Reed-Solomon Code␣
↪over Finite Field of size 59
```

Actually, we can construct the encoder from `C` directly:

```
sage: E = C.encoder("EvaluationVector")
sage: E
Evaluation vector-style encoder for [40, 12, 29] Generalized Reed-Solomon Code␣
↪over Finite Field of size 59
```

generator_matrix ( )

Returns a generator matrix of `self`

EXAMPLES:

```
sage: F = GF(11)
sage: n, k = 10, 5
sage: C = codes.GeneralizedReedSolomonCode(F.list()[:n], k)
sage: E = codes.encoders.GRSEvaluationVectorEncoder(C)
sage: E.generator_matrix()
[1 1 1 1 1 1 1 1 1 1]
[0 1 2 3 4 5 6 7 8 9]
[0 1 4 9 5 3 3 5 9 4]
[0 1 8 5 9 4 7 2 6 3]
[0 1 5 4 3 9 9 3 4 5]
```

class sage.coding.grs. **GRSGaoDecoder** (*code*)

Bases: *sage.coding.decoder.Decoder*

Decoder for Generalized Reed-Solomon codes which uses Gao decoding algorithm to correct errors in codewords.

Gao decoding algorithm uses early terminated extended Euclidean algorithm to find the error locator polynomial. See *[G02]* for details.

REFERENCES:

INPUT:

- `code` – The associated code of this decoder.

EXAMPLES:

```
sage: F = GF(59)
sage: n, k = 40, 12
sage: C = codes.GeneralizedReedSolomonCode(F.list()[:n], k)
sage: D = codes.decoders.GRSGaoDecoder(C)
sage: D
Gao decoder for [40, 12, 29] Generalized Reed-Solomon Code over Finite Field of␣
↪size 59
```

Actually, we can construct the decoder from `C` directly:

```
sage: D = C.decoder("Gao")
sage: D
Gao decoder for [40, 12, 29] Generalized Reed-Solomon Code over Finite Field of␣
↪size 59
```

decode_to_code (*r*)

Corrects the errors in `r` and returns a codeword.

Note: If the code associated to `self` has the same length as its dimension, `r` will be returned as is.

INPUT:

- `r` – a vector of the ambient space of `self.code()`

OUTPUT:

- a vector of `self.code()`

EXAMPLES:

```
sage: F = GF(59)
sage: n, k = 40, 12
sage: C = codes.GeneralizedReedSolomonCode(F.list()[:n], k)
sage: D = codes.decoders.GRSGaoDecoder(C)
sage: c = C.random_element()
sage: Chan = channels.StaticErrorRateChannel(C.ambient_space(), D.decoding_
↪radius())
sage: y = Chan(c)
sage: c == D.decode_to_code(y)
True
```

TESTS:

If one tries to decode a word which is too far from any codeword, an exception is raised:

```
sage: e = vector(F,[0, 0, 54, 23, 1, 0, 0, 0, 53, 21, 0, 0, 0, 34, 6, 11, 0,␣
↪0, 16, 0, 0, 0, 9, 0, 10, 27, 35, 0, 0, 0, 0, 46, 0, 0, 0, 0, 0, 0, 44, 0]);
↪ e.hamming_weight()
15
sage: D.decode_to_code(c + e)
Traceback (most recent call last):
...
DecodingError: Decoding failed because the number of errors exceeded the␣
↪decoding radius
```

If one tries to decode something which is not in the ambient space of the code, an exception is raised:

```
sage: D.decode_to_code(42)
Traceback (most recent call last):
...
ValueError: The word to decode has to be in the ambient space of the code
```

The bug detailed in trac ticket #20340 has been fixed:

```
sage: C = codes.GeneralizedReedSolomonCode(GF(59).list()[:40], 12)
sage: c = C.random_element()
sage: D = C.decoder("Gao")
sage: c = C.random_element()
sage: D.decode_to_code(c) == c
True
```

decode_to_message (*r*)

Decodes `r` to an element in message space of `self` .

Note: If the code associated to `self` has the same length as its dimension, `r` will be unencoded as is. In that case, if `r` is not a codeword, the output is unspecified.

INPUT:

- `r` – a codeword of `self`

OUTPUT:

- a vector of `self` message space

EXAMPLES:

```
sage: F = GF(59)
sage: n, k = 40, 12
sage: C = codes.GeneralizedReedSolomonCode(F.list()[:n], k)
sage: D = codes.decoders.GRSGaoDecoder(C)
sage: c = C.random_element()
sage: Chan = channels.StaticErrorRateChannel(C.ambient_space(), D.decoding_
↪radius())
sage: y = Chan(c)
sage: D.connected_encoder().unencode(c) == D.decode_to_message(y)
True
```

TESTS:

If one tries to decode a word which is too far from any codeword, an exception is raised:

```
sage: e = vector(F,[0, 0, 54, 23, 1, 0, 0, 0, 53, 21, 0, 0, 0, 34, 6, 11, 0,␣
↪0, 16, 0, 0, 0, 9, 0, 10, 27, 35, 0, 0, 0, 0, 46, 0, 0, 0, 0, 0, 0, 44, 0]);
↪ e.hamming_weight()
15
sage: D.decode_to_message(c + e)
Traceback (most recent call last):
...
DecodingError: Decoding failed because the number of errors exceeded the␣
↪decoding radius
```

If one tries to decode something which is not in the ambient space of the code, an exception is raised:

```
sage: D.decode_to_message(42)
Traceback (most recent call last):
...
ValueError: The word to decode has to be in the ambient space of the code
```

The bug detailed in trac ticket #20340 has been fixed:

```
sage: C = codes.GeneralizedReedSolomonCode(GF(59).list()[:40], 12)
sage: c = C.random_element()
sage: D = C.decoder("Gao")
sage: E = D.connected_encoder()
sage: m = E.message_space().random_element()
sage: c = E.encode(m)
sage: D.decode_to_message(c) == m
True
```

decoding_radius ( )

Return maximal number of errors that `self` can decode

OUTPUT:

- the number of errors as an integer

EXAMPLES:

```
sage: F = GF(59)
sage: n, k = 40, 12
sage: C = codes.GeneralizedReedSolomonCode(F.list()[:n], k)
sage: D = codes.decoders.GRSGaoDecoder(C)
sage: D.decoding_radius()
14
```

class sage.coding.grs. **GRSKeyEquationSyndromeDecoder** (*code*)

Bases: *sage.coding.decoder.Decoder*

Decoder for Generalized Reed-Solomon codes which uses a Key equation decoding based on the syndrome polynomial to correct errors in codewords.

This algorithm uses early terminated extended euclidean algorithm to solve the key equations, as described in *[R06]*, pp. 183-195.

INPUT:

- `code` – The associated code of this decoder.

EXAMPLES:

```
sage: F = GF(59)
sage: n, k = 40, 12
sage: C = codes.GeneralizedReedSolomonCode(F.list()[1:n+1], k)
sage: D = codes.decoders.GRSKeyEquationSyndromeDecoder(C)
sage: D
Key equation decoder for [40, 12, 29] Generalized Reed-Solomon Code over Finite␣
↪Field of size 59
```

Actually, we can construct the decoder from `C` directly:

```
sage: D = C.decoder("KeyEquationSyndrome")
sage: D
Key equation decoder for [40, 12, 29] Generalized Reed-Solomon Code over Finite␣
↪Field of size 59
```

decode_to_code (*r*)

Corrects the errors in `r` and returns a codeword.

Note: If the code associated to `self` has the same length as its dimension, `r` will be returned as is.

INPUT:

- `r` – a vector of the ambient space of `self.code()`

OUTPUT:

- a vector of `self.code()`

EXAMPLES:

```
sage: F = GF(59)
sage: n, k = 40, 12
sage: C = codes.GeneralizedReedSolomonCode(F.list()[1:n+1], k)
sage: D = codes.decoders.GRSKeyEquationSyndromeDecoder(C)
sage: c = C.random_element()
sage: Chan = channels.StaticErrorRateChannel(C.ambient_space(), D.decoding_
↪radius())
sage: y = Chan(c)
sage: c == D.decode_to_code(y)
True
```

TESTS:

If one tries to decode a word with too many errors, it returns an exception:

```
sage: Chan = channels.StaticErrorRateChannel(C.ambient_space(), D.decoding_
↪radius()+1)
sage: y = Chan(c)
sage: D.decode_to_message(y)
Traceback (most recent call last):
...
DecodingError: Decoding failed because the number of errors exceeded the␣
↪decoding radius
```

If one tries to decode something which is not in the ambient space of the code, an exception is raised:

```
sage: D.decode_to_code(42)
Traceback (most recent call last):
...
ValueError: The word to decode has to be in the ambient space of the code
```

decode_to_message (*r*)

Decodes‘‘r‘‘ to an element in message space of `self`

Note: If the code associated to `self` has the same length as its dimension, `r` will be unencoded as is. In that case, if `r` is not a codeword, the output is unspecified.

INPUT:

- `r` – a codeword of `self`

OUTPUT:

- a vector of `self` message space

EXAMPLES:

```
sage: F = GF(59)
sage: n, k = 40, 12
sage: C = codes.GeneralizedReedSolomonCode(F.list()[1:n+1], k)
sage: D = codes.decoders.GRSKeyEquationSyndromeDecoder(C)
sage: c = C.random_element()
sage: Chan = channels.StaticErrorRateChannel(C.ambient_space(), D.decoding_
↪radius())
sage: y = Chan(c)
sage: D.connected_encoder().unencode(c) == D.decode_to_message(y)
True
```

decoding_radius ( )

Return maximal number of errors that `self` can decode

OUTPUT:

- the number of errors as an integer

EXAMPLES:

```
sage: F = GF(59)
sage: n, k = 40, 12
sage: C = codes.GeneralizedReedSolomonCode(F.list()[1:n+1], k)
sage: D = codes.decoders.GRSKeyEquationSyndromeDecoder(C)
sage: D.decoding_radius()
14
```

class sage.coding.grs. **GeneralizedReedSolomonCode** (*evaluation_points*, *dimension*, *column_multipliers=None*)

Bases: *sage.coding.linear_code.AbstractLinearCode*

Representation of a Generalized Reed-Solomon code.

INPUT:

- `evaluation_points` – A list of distinct elements of some finite field F.
- `dimension` – The dimension of the resulting code.
- `column_multipliers` – (default: `None`) List of non-zero elements of F. All column multipliers are set to 1 if default value is kept.

EXAMPLES:

A Reed-Solomon code can be constructed by taking all non-zero elements of the field as evaluation points, and specifying no column multipliers:

```
sage: F = GF(7)
sage: evalpts = [F(i) for i in range(1,7)]
sage: C = codes.GeneralizedReedSolomonCode(evalpts,3)
sage: C
[6, 3, 4] Generalized Reed-Solomon Code over Finite Field of size 7
```

More generally, the following is a GRS code where the evaluation points are a subset of the field and includes zero:

```
sage: F = GF(59)
sage: n, k = 40, 12
sage: C = codes.GeneralizedReedSolomonCode(F.list()[:n], k)
sage: C
[40, 12, 29] Generalized Reed-Solomon Code over Finite Field of size 59
```

It is also possible to specify the column multipliers:

```
sage: F = GF(59)
sage: n, k = 40, 12
sage: colmults = F.list()[1:n+1]
sage: C = codes.GeneralizedReedSolomonCode(F.list()[:n], k, colmults)
sage: C
[40, 12, 29] Generalized Reed-Solomon Code over Finite Field of size 59
```

column_multipliers ( )

Returns the column multipliers of `self` as a vector.

EXAMPLES:

```
sage: F = GF(11)
sage: n, k = 10, 5
sage: C = codes.GeneralizedReedSolomonCode(F.list()[:n], k)
sage: C.column_multipliers()
(1, 1, 1, 1, 1, 1, 1, 1, 1, 1)
```

covering_radius ( )

Returns the covering radius of `self` .

The covering radius of a linear code C is the smallest number r s.t. any element of the ambient space of C is at most at distance r to C.

As GRS codes are Maximum Distance Separable codes (MDS), their covering radius is always $d - 1$, where d is the minimum distance. This is opposed to random linear codes where the covering radius is computationally hard to determine.

EXAMPLES:

```
sage: F = GF(2^8, 'a')
sage: n, k = 256, 100
sage: C = codes.GeneralizedReedSolomonCode(F.list()[:n], k)
sage: C.covering_radius()
156
```

decode_to_message (*r*)

Decodes‘‘r‘‘ to an element in message space of `self`

Note: If the code associated to `self` has the same length as its dimension, `r` will be unencoded as is. In that case, if `r` is not a codeword, the output is unspecified.

INPUT:

- `r` – a codeword of `self`

OUTPUT:

- a vector of `self` message space

EXAMPLES:

```
sage: F = GF(11)
sage: n, k = 10, 5
sage: C = codes.GeneralizedReedSolomonCode(F.list()[1:n+1], k)
sage: r = vector(F, (8, 2, 6, 10, 6, 10, 7, 6, 7, 2))
sage: C.decode_to_message(r)
(3, 6, 6, 3, 1)
```

dual_code ( )

Returns the dual code of `self` , which is also a GRS code.

EXAMPLES:

```
sage: F =  GF(59)
sage: colmults = [ F.random_element() for i in range(40) ]
sage: C = codes.GeneralizedReedSolomonCode(F.list()[:40], 12, colmults)
sage: Cd = C.dual_code(); Cd
[40, 28, 13] Generalized Reed-Solomon Code over Finite Field of size 59
```

The dual code of the dual code is the original code:

```
sage: C == Cd.dual_code()
True
```

evaluation_points ( )

Returns the evaluation points of `self` as a vector.

EXAMPLES:

```
sage: F = GF(11)
sage: n, k = 10, 5
sage: C = codes.GeneralizedReedSolomonCode(F.list()[:n], k)
```

```
sage: C.evaluation_points()
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
```

minimum_distance ( )

Returns the minimum distance of `self` . Since a GRS code is always Maximum-Distance-Separable (MDS), this returns `C.length() -C.dimension() + 1` .

EXAMPLES:

```
sage: F = GF(59)
sage: n, k = 40, 12
sage: C = codes.GeneralizedReedSolomonCode(F.list()[:n], k)
sage: C.minimum_distance()
29
```

multipliers_product ( )

Returns the component-wise product of the column multipliers of `self` with the column multipliers of the dual GRS code.

This is a simple Cramer's rule-like expression on the evaluation points of `self` . Recall that the column multipliers of the dual GRS code is also the column multipliers of the parity check matrix of `self` .

EXAMPLES:

```
sage: F = GF(11)
sage: n, k = 10, 5
sage: C = codes.GeneralizedReedSolomonCode(F.list()[:n], k)
sage: C.multipliers_product()
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

parity_check_matrix ( )

Returns the parity check matrix of `self` .

EXAMPLES:

```
sage: F = GF(11)
sage: n, k = 10, 5
sage: C = codes.GeneralizedReedSolomonCode(F.list()[:n], k)
sage: C.parity_check_matrix()
[10  9  8  7  6  5  4  3  2  1]
[ 0  9  5 10  2  3  2 10  5  9]
[ 0  9 10  8  8  4  1  4  7  4]
[ 0  9  9  2 10  9  6  6  1  3]
[ 0  9  7  6  7  1  3  9  8  5]
```

parity_column_multipliers ( )

Returns the list of column multipliers of `self` 's parity check matrix.

EXAMPLES:

```
sage: F = GF(11)
sage: n, k = 10, 5
sage: C = codes.GeneralizedReedSolomonCode(F.list()[:n], k)
sage: C.parity_column_multipliers()
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

weight_distribution ( )

Returns the list whose i'th entry is the number of words of weight i in `self` .

Computing the weight distribution for a GRS code is very fast. Note that for random linear codes, it is computationally hard.

EXAMPLES:

```
sage: F = GF(11)
sage: n, k = 10, 5
sage: C = codes.GeneralizedReedSolomonCode(F.list()[:n], k)
sage: C.weight_distribution()
[1, 0, 0, 0, 0, 0, 2100, 6000, 29250, 61500, 62200]
```

2.3 Hamming Code

Given an integer r and a field F, such that $F = GF(q)$, the $[n, k, d]$ code with length $n = \frac{q^r-1}{q-1}$, dimension $k = \frac{q^r-1}{q-1} - r$ and minimum distance $d = 3$ is called the Hamming Code of order r.

REFERENCES:

class sage.coding.hamming_code. **HammingCode** (*base_field*, *order*)

Bases: *sage.coding.linear_code.AbstractLinearCode*

Representation of a Hamming code.

INPUT:

- `base_field` – the base field over which `self` is defined.
- `order` – the order of `self` .

EXAMPLES:

```
sage: C = codes.HammingCode(GF(7), 3)
sage: C
[57, 54] Hamming Code over Finite Field of size 7
```

minimum_distance ( )

Returns the minimum distance of `self` . It is always 3 as `self` is a Hamming Code.

EXAMPLES:

```
sage: C = codes.HammingCode(GF(7), 3)
sage: C.minimum_distance()
3
```

parity_check_matrix ( )

Returns a parity check matrix of `self` .

The construction of the parity check matrix in case `self` is not a binary code is not really well documented. Regarding the choice of projective geometry, one might check:

- the note over section 2.3 in *[R06]*, pages 47-48
- the dedicated paragraph in *[HP]*, page 30

EXAMPLES:

```
sage: C = codes.HammingCode(GF(3), 3)
sage: C.parity_check_matrix()
[1 0 1 1 0 1 0 1 1 1 0 1 1]
```

```
[0 1 1 2 0 0 1 1 2 0 1 1 2]
[0 0 0 0 1 1 1 1 1 2 2 2 2]
```

2.4 Guruswami-Sudan decoder for Generalized Reed-Solomon codes

REFERENCES:

AUTHORS:

- Johan S. R. Nielsen, original implementation (see *[Nielsen]* for details)
- David Lucas, ported the original implementation in Sage

class sage.coding.guruswami_sudan.gs_decoder. **GRSGuruswamiSudanDecoder** (*code*, *tau=None*, *parameters=None*, *interpolation_alg=None*, *root_finder=None*)

Bases: *sage.coding.decoder.Decoder*

The Guruswami-Sudan list-decoding algorithm for decoding Generalized Reed-Solomon codes.

The Guruswami-Sudan algorithm is a polynomial time algorithm to decode beyond half the minimum distance of the code. It can decode up to the Johnson radius which is $n - \sqrt{(n(n-d))}$, where n, d is the length, respectively minimum distance of the RS code. See *[GS99]* for more details. It is a list-decoder meaning that it returns a list of all closest codewords or their corresponding message polynomials. Note that the output of the `decode_to_code` and `decode_to_message` methods are therefore lists.

The algorithm has two free parameters, the list size and the multiplicity, and these determine how many errors the method will correct: generally, higher decoding radius requires larger values of these parameters. To decode all the way to the Johnson radius, one generally needs values in the order of $O(n^2)$, while decoding just one error less requires just $O(n)$.

This class has static methods for computing choices of parameters given the decoding radius or vice versa.

The Guruswami-Sudan consists of two computationally intensive steps: Interpolation and Root finding, either of which can be completed in multiple ways. This implementation allows choosing the sub-algorithms among currently implemented possibilities, or supplying your own.

INPUT:

- `code` – A code associated to this decoder.
- `tau` – (default: `None`) an integer, the number of errors one wants the Guruswami-Sudan algorithm to correct.
- **`parameters` – (default: `None`) a pair of integers, where:**
  - the first integer is the multiplicity parameter, and
  - the second integer is the list size parameter.
- `interpolation_alg` – (default: `None`) the interpolation algorithm that will be used. The following possibilities are currently available:
  - `"LinearAlgebra"` – uses a linear system solver.

– "LeeOSullivan" – uses Lee O'Sullivan method based on row reduction of a matrix

– None – one of the above will be chosen based on the size of the code and the parameters.

You can also supply your own function to perform the interpolation. See NOTE section for details on the signature of this function.

- root_finder – (default: None) the rootfinding algorithm that will be used. The following possibilities are currently available:

  – "RothRuckenstein" – uses Roth-Ruckenstein algorithm.

  – None – one of the above will be chosen based on the size of the code and the parameters.

  You can also supply your own function to perform the interpolation. See NOTE section for details on the signature of this function.

Note: One has to provide either tau or parameters . If neither are given, an exception will be raised.

If one provides a function as root_finder , its signature has to be: my_rootfinder(Q,maxd=default_value,precision=default_value) . Q will be given as an element of $F[x][y]$. The function must return the roots as a list of polynomials over a univariate polynomial ring. See *roth_ruckenstein_root_finder()* for an example.

If one provides a function as interpolation_alg , its signature has to be: my_inter(interpolation_points,tau,s_and_l,wy) . See *sage.coding.guruswami_sudan.interpolation.gs_interpolation_linalg()* for an example.

EXAMPLES:

```
sage: C = codes.GeneralizedReedSolomonCode(GF(251).list()[:250], 70)
sage: D = codes.decoders.GRSGuruswamiSudanDecoder(C, tau = 97)
sage: D
Guruswami-Sudan decoder for [250, 70, 181] Generalized Reed-Solomon Code over␣
↪Finite Field of size 251 decoding 97 errors with parameters (1, 2)
```

One can specify multiplicity and list size instead of tau :

```
sage: D = codes.decoders.GRSGuruswamiSudanDecoder(C, parameters = (1,2))
sage: D
Guruswami-Sudan decoder for [250, 70, 181] Generalized Reed-Solomon Code over␣
↪Finite Field of size 251 decoding 97 errors with parameters (1, 2)
```

One can pass a method as root_finder (works also for interpolation_alg):

```
sage: from sage.coding.guruswami_sudan.gs_decoder import roth_ruckenstein_root_
↪finder
sage: rf = roth_ruckenstein_root_finder
sage: D = codes.decoders.GRSGuruswamiSudanDecoder(C, parameters = (1,2), root_
↪finder = rf)
sage: D
Guruswami-Sudan decoder for [250, 70, 181] Generalized Reed-Solomon Code over␣
↪Finite Field of size 251 decoding 97 errors with parameters (1, 2)
```

If one wants to use the native Sage algorithms for the root finding step, one can directly pass the string given in the Input block of this class. This works for interpolation_alg as well:

```
sage: D = codes.decoders.GRSGuruswamiSudanDecoder(C, parameters = (1,2), root_
↪finder="RothRuckenstein")
sage: D
Guruswami-Sudan decoder for [250, 70, 181] Generalized Reed-Solomon Code over␣
↪Finite Field of size 251 decoding 97 errors with parameters (1, 2)
```

Actually, we can construct the decoder from `C` directly:

```
sage: D = C.decoder("GuruswamiSudan", tau = 97)
sage: D
Guruswami-Sudan decoder for [250, 70, 181] Generalized Reed-Solomon Code over␣
↪Finite Field of size 251 decoding 97 errors with parameters (1, 2)
```

decode_to_code (*r*)

Return the list of all codeword within radius `self.decoding_radius()` of the received word r.

INPUT:

- `r` – a received word, i.e. a vector in F^n where F and n are the base field respectively length of `self.code()`.

EXAMPLES:

```
sage: C = codes.GeneralizedReedSolomonCode(GF(17).list()[:15], 6)
sage: D = codes.decoders.GRSGuruswamiSudanDecoder(C, tau=5)
sage: c = vector(GF(17), [3,13,12,0,0,7,5,1,8,11,1,9,4,12,14])
sage: c in C
True
sage: r = vector(GF(17), [3,13,12,0,0,7,5,1,8,11,15,12,14,7,10])
sage: r in C
False
sage: codewords = D.decode_to_code(r)
sage: len(codewords)
2
sage: c in codewords
True
```

TESTS:

Check that trac ticket #21347 is fixed:

```
sage: C = codes.GeneralizedReedSolomonCode(GF(13).list()[:10], 3)
sage: D = codes.decoders.GRSGuruswamiSudanDecoder(C, tau = 4)
sage: c = vector(GF(13), [6, 8, 2, 1, 5, 1, 2, 8, 6, 9])
sage: e = vector(GF(13), [1, 0, 0, 1, 1, 0, 0, 1, 0, 1])
sage: D.decode_to_code(c+e)
[]
```

decode_to_message (*r*)

Decodes `r` to the list of polynomials whose encoding by `self.code()` is within Hamming distance `self.decoding_radius()` of `r`.

INPUT:

- `r` – a received word, i.e. a vector in F^n where F and n are the base field respectively length of `self.code()`.

EXAMPLES:

```
sage: C = codes.GeneralizedReedSolomonCode(GF(17).list()[:15], 6)
sage: D = codes.decoders.GRSGuruswamiSudanDecoder(C, tau=5)
sage: F.<x> = GF(17)[]
sage: m = 13*x^4 + 7*x^3 + 10*x^2 + 14*x + 3
sage: c = D.connected_encoder().encode(m)
sage: r = vector(GF(17), [3,13,12,0,0,7,5,1,8,11,15,12,14,7,10])
sage: (c-r).hamming_weight()
5
sage: messages = D.decode_to_message(r)
sage: len(messages)
2
sage: m in messages
True
```

TESTS:

If one has provided a method as a `root_finder` or a `interpolation_alg` which does not fit the allowed signature, an exception will be raised:

```
sage: C = codes.GeneralizedReedSolomonCode(GF(17).list()[:15], 6)
sage: D = codes.decoders.GRSGuruswamiSudanDecoder(C, tau=5, root_finder=next_
↪prime)
sage: F.<x> = GF(17)[]
sage: m = 9*x^5 + 10*x^4 + 9*x^3 + 7*x^2 + 15*x + 2
sage: c = D.connected_encoder().encode(m)
sage: r = vector(GF(17), [3,1,4,2,14,1,0,4,13,12,1,16,1,13,15])
sage: m in D.decode_to_message(r)
Traceback (most recent call last):
...
ValueError: The provided root-finding algorithm has a wrong signature. See␣
↪the documentation of `codes.decoders.GRSGuruswamiSudanDecoder.rootfinding_
↪algorithm()` for details
```

decoding_radius ( )

Returns the maximal number of errors that `self` is able to correct.

EXAMPLES:

```
sage: C = codes.GeneralizedReedSolomonCode(GF(251).list()[:250], 70)
sage: D = C.decoder("GuruswamiSudan", tau = 97)
sage: D.decoding_radius()
97
```

An example where tau is not one of the inputs to the constructor:

```
sage: C = codes.GeneralizedReedSolomonCode(GF(251).list()[:250], 70)
sage: D = C.decoder("GuruswamiSudan", parameters = (2,4))
sage: D.decoding_radius()
105
```

static gs_satisfactory (*tau*, *s*, *l*, *C=None*, *n_k=None*)

Returns whether input parameters satisfy the governing equation of Guruswami-Sudan.

See *[N13]* page 49, definition 3.3 and proposition 3.4 for details.

INPUT:

- `tau` – an integer, number of errrors one expects Guruswami-Sudan algorithm to correct
- `s` – an integer, multiplicity parameter of Guruswami-Sudan algorithm

- `l` – an integer, list size parameter
- `C` – (default: `None`) a `GeneralizedReedSolomonCode`
- `n_k` – (default: `None`) a tuple of integers, respectively the length and the dimension of the `GeneralizedReedSolomonCode`

..NOTE:

```
One has to provide either ``C`` or ``(n, k)``. If none or both are
given, an exception will be raised.
```

EXAMPLES:

```
sage: tau, s, l = 97, 1, 2
sage: n, k = 250, 70
sage: codes.decoders.GRSGuruswamiSudanDecoder.gs_satisfactory(tau, s, l, n_k␣
↪= (n, k))
True
```

One can also pass a GRS code:

```
sage: C = codes.GeneralizedReedSolomonCode(GF(251).list()[:250], 70)
sage: codes.decoders.GRSGuruswamiSudanDecoder.gs_satisfactory(tau, s, l, C =␣
↪C)
True
```

Another example where `s` and `l` does not satisfy the equation:

```
sage: tau, s, l = 118, 47, 80
sage: codes.decoders.GRSGuruswamiSudanDecoder.gs_satisfactory(tau, s, l, n_k␣
↪= (n, k))
False
```

If one provides both `C` and `n_k` an exception is returned:

```
sage: tau, s, l = 97, 1, 2
sage: n, k = 250, 70
sage: C = codes.GeneralizedReedSolomonCode(GF(251).list()[:250], 70)
sage: codes.decoders.GRSGuruswamiSudanDecoder.gs_satisfactory(tau, s, l, C =␣
↪C, n_k = (n, k))
Traceback (most recent call last):
...
ValueError: Please provide only the code or its length and dimension
```

Same if one provides none of these:

```
sage: codes.decoders.GRSGuruswamiSudanDecoder.gs_satisfactory(tau, s, l)
Traceback (most recent call last):
...
ValueError: Please provide either the code or its length and dimension
```

static guruswami_sudan_decoding_radius (*C=None*, *n_k=None*, *l=None*, *s=None*)

Returns the maximal decoding radius of the Guruswami-Sudan decoder and the parameter choices needed for this.

If `s` is set but `l` is not it will return the best decoding radius using this `s` alongside with the required `l`. Vice versa for `l`. If both are set, it returns the decoding radius given this parameter choice.

INPUT:

- C – (default: None) a GeneralizedReedSolomonCode
- n_k – (default: None) a pair of integers, respectively the length and the dimension of the GeneralizedReedSolomonCode
- s – (default: None) an integer, the multiplicity parameter of Guruswami-Sudan algorithm
- l – (default: None) an integer, the list size parameter

Note: One has to provide either C or n_k . If none or both are given, an exception will be raised.

OUTPUT:

- **(tau, (s,l)) – where**
  - tau is the obtained decoding radius, and
  - s,ell are the multiplicity parameter, respectively list size parameter giving this radius.

EXAMPLES:

```
sage: n, k = 250, 70
sage: codes.decoders.GRSGuruswamiSudanDecoder.guruswami_sudan_decoding_
↪radius(n_k = (n, k))
(118, (47, 89))
```

One parameter can be restricted at a time:

```
sage: n, k = 250, 70
sage: codes.decoders.GRSGuruswamiSudanDecoder.guruswami_sudan_decoding_
↪radius(n_k = (n, k), s=3)
(109, (3, 5))
sage: codes.decoders.GRSGuruswamiSudanDecoder.guruswami_sudan_decoding_
↪radius(n_k = (n, k), l=7)
(111, (4, 7))
```

The function can also just compute the decoding radius given the parameters:

```
sage: codes.decoders.GRSGuruswamiSudanDecoder.guruswami_sudan_decoding_
↪radius(n_k = (n, k), s=2, l=6)
(92, (2, 6))
```

interpolation_algorithm ( )

Returns the interpolation algorithm that will be used.

Remember that its signature has to be: my_inter(interpolation_points,tau,s_and_l,wy) . See *sage.coding.guruswami_sudan.interpolation.gs_interpolation_linalg()* for an example.

EXAMPLES:

```
sage: C = codes.GeneralizedReedSolomonCode(GF(251).list()[:250], 70)
sage: D = C.decoder("GuruswamiSudan", tau = 97)
sage: D.interpolation_algorithm()
<function gs_interpolation_lee_osullivan at 0x...>
```

list_size ( )

Returns the list size parameter of self .

EXAMPLES:

```
sage: C = codes.GeneralizedReedSolomonCode(GF(251).list()[:250], 70)
sage: D = C.decoder("GuruswamiSudan", tau = 97)
sage: D.list_size()
2
```

multiplicity ( )

Returns the multiplicity parameter of `self` .

EXAMPLES:

```
sage: C = codes.GeneralizedReedSolomonCode(GF(251).list()[:250], 70)
sage: D = C.decoder("GuruswamiSudan", tau = 97)
sage: D.multiplicity()
1
```

parameters ( )

Returns the multiplicity and list size parameters of `self` .

EXAMPLES:

```
sage: C = codes.GeneralizedReedSolomonCode(GF(251).list()[:250], 70)
sage: D = C.decoder("GuruswamiSudan", tau = 97)
sage: D.parameters()
(1, 2)
```

static **parameters_given_tau** (*tau*, *C=None*, *n_k=None*)

Returns the smallest possible multiplicity and list size given the given parameters of the code and decoding radius.

INPUT:

- `tau` – an integer, number of errors one wants the Guruswami-Sudan algorithm to correct
- `C` – (default: `None`) a `GeneralizedReedSolomonCode`
- `n_k` – (default: `None`) a pair of integers, respectively the length and the dimension of the `GeneralizedReedSolomonCode`

OUTPUT:

- **`(s,l)` – a pair of integers, where:**
  - `s` is the multiplicity parameter, and
  - `l` is the list size parameter.

Note: One should to provide either `C` or `(n,k)` . If neither or both are given, an exception will be raised.

EXAMPLES:

```
sage: tau, n, k = 97, 250, 70
sage: codes.decoders.GRSGuruswamiSudanDecoder.parameters_given_tau(tau, n_k =␣
↪(n, k))
(1, 2)
```

Another example with a bigger decoding radius:

```
sage: tau, n, k = 118, 250, 70
sage: codes.decoders.GRSGuruswamiSudanDecoder.parameters_given_tau(tau, n_k =␣
↪(n, k))
(47, 89)
```

Choosing a decoding radius which is too large results in an errors:

```
sage: tau = 200
sage: codes.decoders.GRSGuruswamiSudanDecoder.parameters_given_tau(tau, n_k =␣
↪(n, k))
Traceback (most recent call last):
...
ValueError: The decoding radius must be less than the Johnson radius (which␣
↪is 118.66)
```

rootfinding_algorithm ( )

Returns the rootfinding algorithm that will be used.

Remember that its signature has to be: `my_rootfinder(Q,maxd=default_value,precision=default_valu` . See *roth_ruckenstein_root_finder()* for an example.

EXAMPLES:

```
sage: from sage.coding.guruswami_sudan.gs_decoder import roth_ruckenstein_
↪root_finder
sage: C = codes.GeneralizedReedSolomonCode(GF(251).list()[:250], 70)
sage: D = C.decoder("GuruswamiSudan", tau = 97)
sage: D.rootfinding_algorithm()
<function roth_ruckenstein_root_finder at 0x...>
```

sage.coding.guruswami_sudan.gs_decoder. **n_k_params** (C, n_k)

Internal helper function for the GRSGuruswamiSudanDecoder class for allowing to specify either a GRS code C or the length and dimensions n, k directly, in all the static functions.

If neither C or n, k were specified to those functions, an appropriate error should be raised. Otherwise, n, k of the code or the supplied tuple directly is returned.

INPUT:

- C – A GRS code or $None$
- n_k – A tuple (n, k) being length and dimension of a GRS code, or $None$.

OUTPUT:

- n_k – A tuple (n, k) being length and dimension of a GRS code.

EXAMPLES:

```
sage: from sage.coding.guruswami_sudan.gs_decoder import n_k_params
sage: n_k_params(None, (10, 5))
(10, 5)
sage: C = codes.GeneralizedReedSolomonCode(GF(11).list()[:10], 5)
sage: n_k_params(C,None)
(10, 5)
sage: n_k_params(None,None)
Traceback (most recent call last):
...
ValueError: Please provide either the code or its length and dimension
sage: n_k_params(C,(12, 2))
Traceback (most recent call last):
```

```
...
ValueError: Please provide only the code or its length and dimension
```

sage.coding.guruswami_sudan.gs_decoder. **roth_ruckenstein_root_finder** (*p*, *maxd=None*, *precision=None*)

Wrapper for Roth-Ruckenstein algorithm to compute the roots of a polynomial with coefficients in `F[x]` .

TESTS:

```
sage: from sage.coding.guruswami_sudan.gs_decoder import roth_ruckenstein_root_
↪finder
sage: R.<x> = GF(13)[]
sage: S.<y> = R[]
sage: p = (y - x^2 - x - 1) * (y + x + 1)
sage: roth_ruckenstein_root_finder(p, maxd = 2)
[12*x + 12, x^2 + x + 1]
```

2.5 Interpolation algorithms for the Guruswami-Sudan decoder

AUTHORS:

- Johan S. R. Nielsen, original implementation (see *[Nielsen]* for details)
- David Lucas, ported the original implementation in Sage

sage.coding.guruswami_sudan.interpolation. **gs_interpolation_lee_osullivan** (*points*, *tau*, *parameters*, *wy*)

Returns an interpolation polynomial Q(x,y) for the given input using the module-based algorithm of Lee and O'Sullivan.

This algorithm constructs an explicit $(\ell+1) \times (\ell+1)$ polynomial matrix whose rows span the $\mathbf{F}_q[x]$ module of all interpolation polynomials. It then runs a row reduction algorithm to find a low-shifted degree vector in this row space, corresponding to a low weighted-degree interpolation polynomial.

INPUT:

- `points` – a list of tuples `(xi,yi)` such that we seek `Q` with `(xi,yi)` being a root of `Q` with multiplicity `s` .
- `tau` – an integer, the number of errors one wants to decode.
- **`parameters` – (default: `None`) a pair of integers, where:**
  - the first integer is the multiplicity parameter of Guruswami-Sudan algorithm and
  - the second integer is the list size parameter.
- `wy` – an integer, the y-weight, where we seek `Q` of low `(1,wy)` weighted degree.

EXAMPLES:

```
sage: from sage.coding.guruswami_sudan.interpolation import gs_interpolation_lee_
↪osullivan
sage: F = GF(11)
sage: points = [(F(0), F(2)), (F(1), F(5)), (F(2), F(0)), (F(3), F(4)), (F(4),␣
↪F(9))\
        , (F(5), F(1)), (F(6), F(9)), (F(7), F(10))]
sage: tau = 1
sage: params = (1, 1)
sage: wy = 1
sage: gs_interpolation_lee_osullivan(points, tau, params, wy)
x^3*y + 2*x^3 - x^2*y + 5*x^2 + 5*x*y - 5*x + 2*y - 4
```

sage.coding.guruswami_sudan.interpolation. **gs_interpolation_linalg** (*points*, *tau*, *parameters*, *wy*)

Compute an interpolation polynomial Q(x,y) for the Guruswami-Sudan algorithm by solving a linear system of equations.

`Q` is a bivariate polynomial over the field of the points, such that the polynomial has a zero of multiplicity at least s at each of the points, where s is the multiplicity parameter. Furthermore, its `(1,wy)` -weighted degree should be less than `_interpolation_max_weighted_deg(n,tau,wy)` , where `n` is the number of points

INPUT:

- `points` – a list of tuples `(xi,yi)` such that we seek `Q` with `(xi,yi)` being a root of `Q` with multiplicity `s` .
- `tau` – an integer, the number of errors one wants to decode.
- **`parameters` – (default: `None`) a pair of integers, where:**
  - the first integer is the multiplicity parameter of Guruswami-Sudan algorithm and
  - the second integer is the list size parameter.
- `wy` – an integer, the y-weight, where we seek `Q` of low `(1,wy)` weighted degree.

EXAMPLES:

The following parameters arise from Guruswami-Sudan decoding of an [6,2,5] GRS code over F(11) with multiplicity 2 and list size 4:

```
sage: from sage.coding.guruswami_sudan.interpolation import gs_interpolation_
↪linalg
sage: F = GF(11)
sage: points = [ (F(x),F(y)) for (x,y) in (0, 5), (1, 1), (2, 4), (3, 6), (4, 3),␣
↪(5, 3)]
sage: tau = 3
sage: params = (2, 4)
sage: wy = 1
sage: Q = gs_interpolation_linalg(points, tau, params, wy); Q
4*x^5 - 4*x^4*y - 2*x^2*y^3 - x*y^4 + 3*x^4 - 4*x^2*y^2 + 5*y^4 - x^3 + x^2*y +␣
↪5*x*y^2 - 5*y^3 + 3*x*y - 2*y^2 + x - 4*y + 1
```

We verify that the interpolation polynomial has a zero of multiplicity at least 2 in each point:

```
sage: all( Q(x=a, y=b).is_zero() for (a,b) in points )
True
sage: x,y = Q.parent().gens()
```

```
sage: dQdx = Q.derivative(x)
sage: all( dQdx(x=a, y=b).is_zero() for (a,b) in points )
True
sage: dQdy = Q.derivative(y)
sage: all( dQdy(x=a, y=b).is_zero() for (a,b) in points )
True
```

sage.coding.guruswami_sudan.interpolation. **lee_osullivan_module** (*points*, *parameters*, *wy*)

Returns the analytically straight-forward basis for the $\mathbf{F}_q[x]$ module containing all interpolation polynomials, as according to Lee and O'Sullivan.

The module is constructed in the following way: Let $R(x)$ be the Lagrange interpolation polynomial through the sought interpolation points (x_i, y_i), i.e. $R(x_i) = y_i$. Let $G(x) = \prod_{i=1}^{n}(x - x_i)$. Then the i'th row of the basis matrix of the module is the coefficient-vector of the following polynomial in $\mathbf{F}_q[x][y]$:

$$P_i(x, y) = G(x)^{[i-s]}(y - R(x))^{i-[i-s]}y^{[i-s]} ,$$

where $[a]$ for real a is a when $a > 0$ and 0 otherwise. It is easily seen that $P_i(x, y)$ is an interpolation polynomial, i.e. it is zero with multiplicity at least s on each of the points (x_i, y_i).

INPUT:

- `points` – a list of tuples `(xi,yi)` such that we seek `Q` with `(xi,yi)` being a root of `Q` with multiplicity `s` .
- **`parameters` – (default: `None`) a pair of integers, where:**
  - the first integer is the multiplicity parameter s of Guruswami-Sudan algorithm and
  - the second integer is the list size parameter.
- `wy` – an integer, the y-weight, where we seek `Q` of low `(1,wy)` weighted degree.

EXAMPLES:

```
sage: from sage.coding.guruswami_sudan.interpolation import lee_osullivan_module
sage: F = GF(11)
sage: points = [(F(0), F(2)), (F(1), F(5)), (F(2), F(0)), (F(3), F(4)), (F(4),␣
↪F(9))\
        , (F(5), F(1)), (F(6), F(9)), (F(7), F(10))]
sage: params = (1, 1)
sage: wy = 1
sage: lee_osullivan_module(points, params, wy)
[x^8 + 5*x^7 + 3*x^6 + 9*x^5 + 4*x^4 + 2*x^3 + 9*x   0]
[ 10*x^7 + 4*x^6 + 9*x^4 + 7*x^3 + 2*x^2 + 9*x + 9   1]
```

2.6 Guruswami-Sudan utility methods

AUTHORS:

- Johan S. R. Nielsen, original implementation (see *[Nielsen]* for details)
- David Lucas, ported the original implementation in Sage

sage.coding.guruswami_sudan.utils. **apply_shifts** (*M*, *shifts*)

Applies column shifts inplace to the polynomial matrix M.

This is equivalent to multiplying the n'$thcolumnof$'M with $x^{shifts[n]}$.

INPUT:

- `M` – a polynomial matrix
- `shifts` – a list of non-negative integer shifts

EXAMPLES:

```
sage: from sage.coding.guruswami_sudan.utils import apply_shifts
sage: F.<x> = GF(7)[]
sage: M = matrix(F, [[2*x^2 + x, 5*x^2 + 2*x + 1, 4*x^2 + x],\
                     [x^2 + 3*x + 3, 5*x^2 + 5*x + 1, 6*x^2 + 5*x + 4],\
                     [5*x^2 + 2*x + 4, 4*x^2 + 2*x, 5*x^2 + x + 2]])
sage: shifts = [1, 2, 3]
sage: apply_shifts(M, shifts)
sage: M
[          2*x^3 + x^2   5*x^4 + 2*x^3 + x^2           4*x^5 + x^4]
[    x^3 + 3*x^2 + 3*x   5*x^4 + 5*x^3 + x^2 6*x^5 + 5*x^4 + 4*x^3]
[  5*x^3 + 2*x^2 + 4*x         4*x^4 + 2*x^3   5*x^5 + x^4 + 2*x^3]
```

sage.coding.guruswami_sudan.utils. **gilt** (*x*)

Returns the greatest integer smaller than `x` .

EXAMPLES:

```
sage: from sage.coding.guruswami_sudan.utils import gilt
sage: gilt(43)
42
```

It works with any type of numbers (not only integers):

```
sage: gilt(43.041)
43
```

sage.coding.guruswami_sudan.utils. **johnson_radius** (*n*, *d*)

Returns the Johnson-radius for the code length n and the minimum distance d.

The Johnson radius is defined as $n - \sqrt{(n(n-d))}$.

INPUT:

- `n` – an integer, the length of the code
- `d` – an integer, the minimum distance of the code

EXAMPLES:

```
sage: sage.coding.guruswami_sudan.utils.johnson_radius(250, 181)
-5*sqrt(690) + 250
```

sage.coding.guruswami_sudan.utils. **leading_term** (*v*, *shifts=None*)

Returns the term of `v` with the highest degree.

This methods can manage shifted degree, by providing `shift` to it.

In case of several positions having the same, highest degree, the term with the right-most position is returned.

INPUT:

- `v` – a vector of polynomials
- `shifts` – (default: `None`) a list of integer shifts to consider `v` under. If `None` , all shifts are considered as `0` .

EXAMPLES:

```
sage: from sage.coding.guruswami_sudan.utils import leading_term
sage: F.<x> = GF(7)[]
sage: v = vector(F, [3*x^2 + 3*x + 4, 4*x + 3, 4*x^2 + 4*x + 5, x^2 + 2*x + 5,␣
↪3*x^2 + 5*x])
sage: leading_term(v)
3*x^2 + 5*x
```

sage.coding.guruswami_sudan.utils. **ligt** (*x*)

Returns the least integer greater than `x` .

EXAMPLES:

```
sage: from sage.coding.guruswami_sudan.utils import ligt
sage: ligt(41)
42
```

It works with any type of numbers (not only integers):

```
sage: ligt(41.041)
42
```

sage.coding.guruswami_sudan.utils. **polynomial_to_list** (*p*, *len*)

Returns `p` as a list of its coefficients of length `len` .

INPUT:

- `p` – a polynomial
- `len` – an integer. If `len` is smaller than the degree of `p` , the returned list will be of size degree of `p` , else it will be of size `len` .

EXAMPLES:

```
sage: from sage.coding.guruswami_sudan.utils import polynomial_to_list
sage: F.<x> = GF(41)[]
sage: p = 9*x^2 + 8*x + 37
sage: polynomial_to_list(p, 4)
[37, 8, 9, 0]
```

sage.coding.guruswami_sudan.utils. **remove_shifts** (*M*, *shifts*)

Removes the shifts inplace to the matrix M as they were introduced by *apply_shifts()* .

If M was not earlier called with *apply_shifts()* using the same shifts, then the least significant coefficients of the entries of M, corresponding to how much we are shifting down, will be lost.

INPUT:

- `M` – a polynomial matrix
- `shifts` – a list of non-negative integer shifts

EXAMPLES:

```
sage: from sage.coding.guruswami_sudan.utils import remove_shifts
sage: F.<x> = GF(7)[]
sage: M = matrix(F, [[2*x^3 + x^2, 5*x^4 + 2*x^3 + x^2, 4*x^5 + x^4],\
                     [x^3 + 3*x^2 + 3*x, 5*x^4 + 5*x^3 + x^2, 6*x^5 + 5*x^4 + 4*x^
↪3],\
                     [5*x^3 + 2*x^2 + 4*x, 4*x^4 + 2*x^3, 5*x^5 + x^4 + 2*x^3]])
```

```
sage: shifts = [1, 2, 3]
sage: remove_shifts(M, shifts)
sage: M
[      2*x^2 + x 5*x^2 + 2*x + 1       4*x^2 + x]
[  x^2 + 3*x + 3 5*x^2 + 5*x + 1 6*x^2 + 5*x + 4]
[5*x^2 + 2*x + 4     4*x^2 + 2*x   5*x^2 + x + 2]
```

sage.coding.guruswami_sudan.utils. **solve_degree2_to_integer_range** (*a*, *b*, *c*)

Returns the greatest integer range $[i_1, i_2]$ such that $i_1 > x_1$ and $i_2 < x_2$ where x_1, x_2 are the two zeroes of the equation in x: $ax^2 + bx + c = 0$.

If there is no real solution to the equation, it returns an empty range with negative coefficients.

INPUT:

- `a` , `b` and `c` – coefficients of a second degree equation, `a` being the coefficient of the higher degree term.

EXAMPLES:

```
sage: from sage.coding.guruswami_sudan.utils import solve_degree2_to_integer_range
sage: solve_degree2_to_integer_range(1, -5, 1)
(1, 4)
```

If there is no real solution:

```
sage: solve_degree2_to_integer_range(50, 5, 42)
(-2, -1)
```

2.7 Subfield subcode

Let C be a $[n, k]$ code over $\mathbf{F}_(q^t)$. Let $Cs = \{c \in C | \forall i, c_i \in \mathbf{F}_(q)\}$, c_i being the i-th coordinate of c.

Cs is called the subfield subcode of C over $\mathbf{F}_(q)$

class sage.coding.subfield_subcode. **SubfieldSubcode** (*original_code*, *subfield*, *embedding=None*)

Bases: *sage.coding.linear_code.AbstractLinearCode*

Representation of a subfield subcode.

INPUT:

- `original_code` – the code `self` comes from.
- `subfield` – the base field of `self` .
- `embedding` – (default: `None`) an homomorphism from `subfield` to `original_code` 's base field. If `None` is provided, it will default to the first homomorphism of the list of homomorphisms Sage can build.

EXAMPLES:

```
sage: C = codes.random_linear_code(GF(16, 'aa'), 7, 3)
sage: codes.SubfieldSubcode(C, GF(4, 'a'))
doctest:...: FutureWarning: This class/method/function is marked as experimental.␣
↪It, its functionality or its interface might change without a formal␣
↪deprecation.
See http://trac.sagemath.org/20284 for details.
Subfield subcode over Finite Field in a of size 2^2 coming from Linear code of␣
↪length 7, dimension 3 over Finite Field in aa of size 2^4
```

dimension ( )
Returns the dimension of `self` .

dimension_lower_bound ( )
Returns a lower bound for the dimension of `self` .

EXAMPLES:

```
sage: C = codes.random_linear_code(GF(16, 'aa'), 7, 3)
sage: Cs = codes.SubfieldSubcode(C, GF(4, 'a'))
sage: Cs.dimension_lower_bound()
-1
```

dimension_upper_bound ( )
Returns an upper bound for the dimension of `self` .

EXAMPLES:

```
sage: C = codes.random_linear_code(GF(16, 'aa'), 7, 3)
sage: Cs = codes.SubfieldSubcode(C, GF(4, 'a'))
sage: Cs.dimension_upper_bound()
3
```

embedding ( )
Returns the field embedding between the base field of `self` and the base field of its original code.

EXAMPLES:

```
sage: C = codes.random_linear_code(GF(16, 'aa'), 7, 3)
sage: Cs = codes.SubfieldSubcode(C, GF(4, 'a'))
sage: Cs.embedding()
Relative field extension between Finite Field in aa of size 2^4 and Finite␣
↪Field in a of size 2^2
```

original_code ( )
Returns the original code of `self` .

EXAMPLES:

```
sage: C = codes.random_linear_code(GF(16, 'aa'), 7, 3)
sage: Cs = codes.SubfieldSubcode(C, GF(4, 'a'))
sage: Cs.original_code()
Linear code of length 7, dimension 3 over Finite Field in aa of size 2^4
```

parity_check_matrix ( )
Returns a parity check matrix of `self` .

EXAMPLES:

```
sage: C = codes.GeneralizedReedSolomonCode(GF(16, 'aa').list()[:13], 5)
sage: Cs = codes.SubfieldSubcode(C, GF(4, 'a'))
sage: Cs.parity_check_matrix()
[    1     0     0     0     0     0     0     0     0     0     1 a + 1 a +␣
↪1]
[    0     1     0     0     0     0     0     0     0     0 a + 1     0     ␣
↪a]
[    0     0     1     0     0     0     0     0     0     0 a + 1     a     ␣
↪0]
```

```
[    0     0     0     1     0     0     0     0     0     0     0 a + 1
↪a]
[    0     0     0     0     1     0     0     0     0     0 a + 1     1 a +
↪1]
[    0     0     0     0     0     1     0     0     0     0     1     1
↪1]
[    0     0     0     0     0     0     1     0     0     0     a     a
↪1]
[    0     0     0     0     0     0     0     1     0     0     a     1
↪a]
[    0     0     0     0     0     0     0     0     1     0 a + 1 a + 1
↪1]
[    0     0     0     0     0     0     0     0     0     1     a     0 a +
↪1]
```

class sage.coding.subfield_subcode. **SubfieldSubcodeOriginalCodeDecoder**(*code*, *original_decoder=None*, ***kwargs*)

Bases: *sage.coding.decoder.Decoder*

Decoder decoding through a decoder over the original code of `code` .

INPUT:

- `code` – The associated code of this decoder
- `original_decoder` – (default: `None`) The decoder that will be used over the original code. It has to be a decoder object over the original code. If it is set to `None` , the default decoder over the original code will be used.
- `**kwargs` – All extra arguments are forwarded to original code's decoder

EXAMPLES:

```
sage: C = codes.GeneralizedReedSolomonCode(GF(16, 'aa').list()[:13], 5)
sage: Cs = codes.SubfieldSubcode(C, GF(4, 'a'))
sage: codes.decoders.SubfieldSubcodeOriginalCodeDecoder(Cs)
Decoder of Subfield subcode over Finite Field in a of size 2^2 coming from [13,
↪5, 9] Generalized Reed-Solomon Code over Finite Field in aa of size 2^4 through
↪Gao decoder for [13, 5, 9] Generalized Reed-Solomon Code over Finite Field in
↪aa of size 2^4
```

decode_to_code(*y*)

Corrects the errors in `word` and returns a codeword.

EXAMPLES:

```
sage: C = codes.GeneralizedReedSolomonCode(GF(16, 'aa').list()[:13], 5)
sage: Cs = codes.SubfieldSubcode(C, GF(4, 'a'))
sage: D = codes.decoders.SubfieldSubcodeOriginalCodeDecoder(Cs)
sage: Chan = channels.StaticErrorRateChannel(Cs.ambient_space(), D.decoding_
↪radius())
sage: c = Cs.random_element()
sage: y = Chan(c)
sage: c == D.decode_to_code(y)
True
```

decoding_radius(***kwargs*)

Returns maximal number of errors `self` can decode.

INPUT:

- `kwargs` – Optional arguments are forwarded to original decoder's *sage.coding.decoder.Decoder.decoding_radius()* method.

EXAMPLES:

```
sage: C = codes.GeneralizedReedSolomonCode(GF(16, 'aa').list()[:13], 5)
sage: Cs = codes.SubfieldSubcode(C, GF(4, 'a'))
sage: D = codes.decoders.SubfieldSubcodeOriginalCodeDecoder(Cs)
sage: D.decoding_radius()
4
```

original_decoder ( )

Returns the decoder over the original code that will be used to decode words of *sage.coding.decoder.Decoder.code()* .

EXAMPLES:

```
sage: C = codes.GeneralizedReedSolomonCode(GF(16, 'aa').list()[:13], 5)
sage: Cs = codes.SubfieldSubcode(C, GF(4, 'a'))
sage: D = codes.decoders.SubfieldSubcodeOriginalCodeDecoder(Cs)
sage: D.original_decoder()
Gao decoder for [13, 5, 9] Generalized Reed-Solomon Code over Finite Field in␣
↪aa of size 2^4
```

2.8 Linear code constructors that do not preserve the structural information.

This file contains a variety of constructions which builds the generator matrix of special (or random) linear codes and wraps them in a *sage.coding.linear_code.LinearCode* object. These constructions are therefore not rich objects such as `sage.coding.grs.GeneralizedReedSolomonCodes` .

For deprecation reasons, this file also contains some constructions for which Sage now does have rich representations.

All codes available here can be accessed through the `codes` object:

```
sage: codes.BinaryGolayCode()
Linear code of length 23, dimension 12 over Finite Field of size 2
```

AUTHOR:

- David Joyner (2007-05): initial version
- " (2008-02): added cyclic codes, Hamming codes
- " (2008-03): added BCH code, LinearCodeFromCheckmatrix, ReedSolomonCode, WalshCode, DuadicCodeEvenPair, DuadicCodeOddPair, QR codes (even and odd)
- " (2008-09) fix for bug in BCHCode reported by F. Voloch
- " (2008-10) small docstring changes to WalshCode and walsh_matrix

`sage.coding.code_constructions.` **BCHCode** (*n*, *delta*, *F*, *b=0*)

A 'Bose-Chaudhuri-Hockenghem code' (or BCH code for short) is the largest possible cyclic code of length n over field F=GF(q), whose generator polynomial has zeros (which contain the set) $Z = \{a^b, a^{b+1}, ..., a^{b+delta-2}\}$, where a is a primitive n^{th} root of unity in the splitting field $GF(q^m)$, b is an integer $0 \leq b \leq n - delta + 1$ and m is the multiplicative order of q modulo n. (The integers $b, ..., b + delta - 2$

typically lie in the range $1, ..., n-1$.) The integer $delta \geq 1$ is called the "designed distance". The length n of the code and the size q of the base field must be relatively prime. The generator polynomial is equal to the least common multiple of the minimal polynomials of the elements of the set Z above.

Special cases are b=1 (resulting codes are called 'narrow-sense' BCH codes), and $n = q^m - 1$ (known as 'primitive' BCH codes).

It may happen that several values of delta give rise to the same BCH code. The largest one is called the Bose distance of the code. The true minimum distance, d, of the code is greater than or equal to the Bose distance, so $d \geq delta$.

EXAMPLES:

```
sage: FF.<a> = GF(3^2,"a")
sage: x = PolynomialRing(FF,"x").gen()
sage: L = [b.minpoly() for b in [a,a^2,a^3]]; g = LCM(L)
sage: f = x^(8)-1
sage: g.divides(f)
True
sage: C = codes.CyclicCode(8,g); C
Linear code of length 8, dimension 4 over Finite Field of size 3
sage: C.minimum_distance()
4
sage: C = codes.BCHCode(8,3,GF(3),1); C
Linear code of length 8, dimension 4 over Finite Field of size 3
sage: C.minimum_distance()
4
sage: C = codes.BCHCode(8,3,GF(3)); C
Linear code of length 8, dimension 5 over Finite Field of size 3
sage: C.minimum_distance()
3
sage: C = codes.BCHCode(26, 5, GF(5), b=1); C
Linear code of length 26, dimension 10 over Finite Field of size 5
```

sage.coding.code_constructions. **BinaryGolayCode** ( )

BinaryGolayCode() returns a binary Golay code. This is a perfect [23,12,7] code. It is also (equivalent to) a cyclic code, with generator polynomial $g(x) = 1 + x^2 + x^4 + x^5 + x^6 + x^{10} + x^{11}$. Extending it yields the extended Golay code (see ExtendedBinaryGolayCode).

EXAMPLE:

```
sage: C = codes.BinaryGolayCode()
sage: C
Linear code of length 23, dimension 12 over Finite Field of size 2
sage: C.minimum_distance()
7
sage: C.minimum_distance(algorithm='gap') # long time, check d=7
7
```

AUTHORS:

- David Joyner (2007-05)

sage.coding.code_constructions. **CyclicCode** (*n*, *g*, *ignore=True*)

If g is a polynomial over GF(q) which divides $x^n - 1$ then this constructs the code "generated by g" (ie, the code associated with the principle ideal gR in the ring $R = GF(q)[x]/(x^n - 1)$ in the usual way).

The option "ignore" says to ignore the condition that (a) the characteristic of the base field does not divide the length (the usual assumption in the theory of cyclic codes), and (b) g must divide $x^n - 1$. If ignore=True, instead of returning an error, a code generated by $gcd(x^n - 1, g)$ is created.

EXAMPLES:

```
sage: P.<x> = PolynomialRing(GF(3),"x")
sage: g = x-1
sage: C = codes.CyclicCodeFromGeneratingPolynomial(4,g); C
Linear code of length 4, dimension 3 over Finite Field of size 3
sage: P.<x> = PolynomialRing(GF(4,"a"),"x")
sage: g = x^3+1
sage: C = codes.CyclicCodeFromGeneratingPolynomial(9,g); C
Linear code of length 9, dimension 6 over Finite Field in a of size 2^2
sage: P.<x> = PolynomialRing(GF(2),"x")
sage: g = x^3+x+1
sage: C = codes.CyclicCodeFromGeneratingPolynomial(7,g); C
Linear code of length 7, dimension 4 over Finite Field of size 2
sage: C.generator_matrix()
[1 1 0 1 0 0 0]
[0 1 1 0 1 0 0]
[0 0 1 1 0 1 0]
[0 0 0 1 1 0 1]
sage: g = x+1
sage: C = codes.CyclicCodeFromGeneratingPolynomial(4,g); C
Linear code of length 4, dimension 3 over Finite Field of size 2
sage: C.generator_matrix()
[1 1 0 0]
[0 1 1 0]
[0 0 1 1]
```

On the other hand, CyclicCodeFromPolynomial(4,x) will produce a ValueError including a traceback error message: "x must divide $x^4 - 1$". You will also get a ValueError if you type

```
sage: P.<x> = PolynomialRing(GF(4,"a"),"x")
sage: g = x^2+1
```

followed by CyclicCodeFromGeneratingPolynomial(6,g). You will also get a ValueError if you type

```
sage: P.<x> = PolynomialRing(GF(3),"x")
sage: g = x^2-1
sage: C = codes.CyclicCodeFromGeneratingPolynomial(5,g); C
Linear code of length 5, dimension 4 over Finite Field of size 3
```

followed by C = CyclicCodeFromGeneratingPolynomial(5,g,False), with a traceback message including "x^2+2 must divide $x^5 - 1$".

sage.coding.code_constructions. **CyclicCodeFromCheckPolynomial** (*n*, *h*, *ignore=True*)

If h is a polynomial over GF(q) which divides $x^n - 1$ then this constructs the code "generated by $g = (x^n - 1)/h$" (ie, the code associated with the principle ideal gR in the ring $R = GF(q)[x]/(x^n - 1)$ in the usual way). The option "ignore" says to ignore the condition that the characteristic of the base field does not divide the length (the usual assumption in the theory of cyclic codes).

EXAMPLES:

```
sage: P.<x> = PolynomialRing(GF(3),"x")
sage: C = codes.CyclicCodeFromCheckPolynomial(4,x + 1); C
Linear code of length 4, dimension 1 over Finite Field of size 3
sage: C = codes.CyclicCodeFromCheckPolynomial(4,x^3 + x^2 + x + 1); C
Linear code of length 4, dimension 3 over Finite Field of size 3
sage: C.generator_matrix()
[2 1 0 0]
```

```
[0 2 1 0]
[0 0 2 1]
```

sage.coding.code_constructions. **CyclicCodeFromGeneratingPolynomial** (*n*, *g*, *ignore=True*)

If g is a polynomial over GF(q) which divides $x^n - 1$ then this constructs the code "generated by g" (ie, the code associated with the principle ideal gR in the ring $R = GF(q)[x]/(x^n - 1)$ in the usual way).

The option "ignore" says to ignore the condition that (a) the characteristic of the base field does not divide the length (the usual assumption in the theory of cyclic codes), and (b) g must divide $x^n - 1$. If ignore=True, instead of returning an error, a code generated by $gcd(x^n - 1, g)$ is created.

EXAMPLES:

```
sage: P.<x> = PolynomialRing(GF(3),"x")
sage: g = x-1
sage: C = codes.CyclicCodeFromGeneratingPolynomial(4,g); C
Linear code of length 4, dimension 3 over Finite Field of size 3
sage: P.<x> = PolynomialRing(GF(4,"a"),"x")
sage: g = x^3+1
sage: C = codes.CyclicCodeFromGeneratingPolynomial(9,g); C
Linear code of length 9, dimension 6 over Finite Field in a of size 2^2
sage: P.<x> = PolynomialRing(GF(2),"x")
sage: g = x^3+x+1
sage: C = codes.CyclicCodeFromGeneratingPolynomial(7,g); C
Linear code of length 7, dimension 4 over Finite Field of size 2
sage: C.generator_matrix()
[1 1 0 1 0 0 0]
[0 1 1 0 1 0 0]
[0 0 1 1 0 1 0]
[0 0 0 1 1 0 1]
sage: g = x+1
sage: C = codes.CyclicCodeFromGeneratingPolynomial(4,g); C
Linear code of length 4, dimension 3 over Finite Field of size 2
sage: C.generator_matrix()
[1 1 0 0]
[0 1 1 0]
[0 0 1 1]
```

On the other hand, CyclicCodeFromPolynomial(4,x) will produce a ValueError including a traceback error message: "x must divide $x^4 - 1$". You will also get a ValueError if you type

```
sage: P.<x> = PolynomialRing(GF(4,"a"),"x")
sage: g = x^2+1
```

followed by CyclicCodeFromGeneratingPolynomial(6,g). You will also get a ValueError if you type

```
sage: P.<x> = PolynomialRing(GF(3),"x")
sage: g = x^2-1
sage: C = codes.CyclicCodeFromGeneratingPolynomial(5,g); C
Linear code of length 5, dimension 4 over Finite Field of size 3
```

followed by C = CyclicCodeFromGeneratingPolynomial(5,g,False), with a traceback message including "$x^2 + 2$ must divide $x^5 - 1$".

sage.coding.code_constructions. **DuadicCodeEvenPair** (*F*, *S1*, *S2*)

Constructs the "even pair" of duadic codes associated to the "splitting" (see the docstring for `_is_a_splitting` for the definition) S1, S2 of n.

Warning: Maybe the splitting should be associated to a sum of q-cyclotomic cosets mod n, where q is a *prime*.

EXAMPLES:

```
sage: from sage.coding.code_constructions import _is_a_splitting
sage: n = 11; q = 3
sage: C = Zmod(n).cyclotomic_cosets(q); C
[[0], [1, 3, 4, 5, 9], [2, 6, 7, 8, 10]]
sage: S1 = C[1]
sage: S2 = C[2]
sage: _is_a_splitting(S1,S2,11)
True
sage: codes.DuadicCodeEvenPair(GF(q),S1,S2)
(Linear code of length 11, dimension 5 over Finite Field of size 3,
 Linear code of length 11, dimension 5 over Finite Field of size 3)
```

sage.coding.code_constructions. **DuadicCodeOddPair** (*F*, *S1*, *S2*)

Constructs the "odd pair" of duadic codes associated to the "splitting" S1, S2 of n.

Warning: Maybe the splitting should be associated to a sum of q-cyclotomic cosets mod n, where q is a *prime*.

EXAMPLES:

```
sage: from sage.coding.code_constructions import _is_a_splitting
sage: n = 11; q = 3
sage: C = Zmod(n).cyclotomic_cosets(q); C
[[0], [1, 3, 4, 5, 9], [2, 6, 7, 8, 10]]
sage: S1 = C[1]
sage: S2 = C[2]
sage: _is_a_splitting(S1,S2,11)
True
sage: codes.DuadicCodeOddPair(GF(q),S1,S2)
(Linear code of length 11, dimension 6 over Finite Field of size 3,
 Linear code of length 11, dimension 6 over Finite Field of size 3)
```

This is consistent with Theorem 6.1.3 in *[HP]*.

sage.coding.code_constructions. **ExtendedBinaryGolayCode** ( )

ExtendedBinaryGolayCode() returns the extended binary Golay code. This is a perfect [24,12,8] code. This code is self-dual.

EXAMPLES:

```
sage: C = codes.ExtendedBinaryGolayCode()
sage: C
Linear code of length 24, dimension 12 over Finite Field of size 2
sage: C.minimum_distance()
8
sage: C.minimum_distance(algorithm='gap') # long time, check d=8
8
```

AUTHORS:

- David Joyner (2007-05)

sage.coding.code_constructions. **ExtendedQuadraticResidueCode** (n, F)

The extended quadratic residue code (or XQR code) is obtained from a QR code by adding a check bit to the last coordinate. (These codes have very remarkable properties such as large automorphism groups and duality properties - see *[HP]*, Section 6.6.3-6.6.4.)

INPUT:

- n - an odd prime
- F - a finite prime field F whose order must be a quadratic residue modulo n.

OUTPUT: Returns an extended quadratic residue code.

EXAMPLES:

```
sage: C1 = codes.QuadraticResidueCode(7,GF(2))
sage: C2 = C1.extended_code()
sage: C3 = codes.ExtendedQuadraticResidueCode(7,GF(2)); C3
Extended code coming from Linear code of length 7, dimension 4 over Finite Field␣
↪of size 2
sage: C2 == C3
True
sage: C = codes.ExtendedQuadraticResidueCode(17,GF(2))
sage: C
Extended code coming from Linear code of length 17, dimension 9 over Finite Field␣
↪of size 2
sage: C3 = codes.QuadraticResidueCodeOddPair(7,GF(2))[0]
sage: C3x = C3.extended_code()
sage: C4 = codes.ExtendedQuadraticResidueCode(7,GF(2))
sage: C3x == C4
True
```

AUTHORS:

- David Joyner (07-2006)

sage.coding.code_constructions. **ExtendedTernaryGolayCode** ( )

ExtendedTernaryGolayCode returns a ternary Golay code. This is a self-dual perfect [12,6,6] code.

EXAMPLES:

```
sage: C = codes.ExtendedTernaryGolayCode()
sage: C
Linear code of length 12, dimension 6 over Finite Field of size 3
sage: C.minimum_distance()
6
sage: C.minimum_distance(algorithm='gap') # long time, check d=6
6
```

AUTHORS:

- David Joyner (11-2005)

sage.coding.code_constructions. **QuadraticResidueCode** (n, F)

A quadratic residue code (or QR code) is a cyclic code whose generator polynomial is the product of the polynomials $x - \alpha^i$ (α is a primitive n^{th} root of unity; i ranges over the set of quadratic residues modulo n).

See QuadraticResidueCodeEvenPair and QuadraticResidueCodeOddPair for a more general construction.

INPUT:

- n - an odd prime

- F - a finite prime field F whose order must be a quadratic residue modulo n.

OUTPUT: Returns a quadratic residue code.

EXAMPLES:

```
sage: C = codes.QuadraticResidueCode(7,GF(2))
sage: C
Linear code of length 7, dimension 4 over Finite Field of size 2
sage: C = codes.QuadraticResidueCode(17,GF(2))
sage: C
Linear code of length 17, dimension 9 over Finite Field of size 2
sage: C1 = codes.QuadraticResidueCodeOddPair(7,GF(2))[0]
sage: C2 = codes.QuadraticResidueCode(7,GF(2))
sage: C1 == C2
True
sage: C1 = codes.QuadraticResidueCodeOddPair(17,GF(2))[0]
sage: C2 = codes.QuadraticResidueCode(17,GF(2))
sage: C1 == C2
True
```

AUTHORS:

- David Joyner (11-2005)

sage.coding.code_constructions. **QuadraticResidueCodeEvenPair** (n, F)

Quadratic residue codes of a given odd prime length and base ring either don't exist at all or occur as 4-tuples - a pair of "odd-like" codes and a pair of "even-like" codes. If $n > 2$ is prime then (Theorem 6.6.2 in *[HP]*) a QR code exists over $GF(q)$ iff q is a quadratic residue mod n.

They are constructed as "even-like" duadic codes associated the splitting (Q,N) mod n, where Q is the set of non-zero quadratic residues and N is the non-residues.

EXAMPLES:

```
sage: codes.QuadraticResidueCodeEvenPair(17,GF(13))
(Linear code of length 17, dimension 8 over Finite Field of size 13,
 Linear code of length 17, dimension 8 over Finite Field of size 13)
sage: codes.QuadraticResidueCodeEvenPair(17,GF(2))
(Linear code of length 17, dimension 8 over Finite Field of size 2,
 Linear code of length 17, dimension 8 over Finite Field of size 2)
sage: codes.QuadraticResidueCodeEvenPair(13,GF(9,"z"))
(Linear code of length 13, dimension 6 over Finite Field in z of size 3^2,
 Linear code of length 13, dimension 6 over Finite Field in z of size 3^2)
sage: C1,C2 = codes.QuadraticResidueCodeEvenPair(7,GF(2))
sage: C1.is_self_orthogonal()
True
sage: C2.is_self_orthogonal()
True
sage: C3 = codes.QuadraticResidueCodeOddPair(17,GF(2))[0]
sage: C4 = codes.QuadraticResidueCodeEvenPair(17,GF(2))[1]
sage: C3 == C4.dual_code()
True
```

This is consistent with Theorem 6.6.9 and Exercise 365 in *[HP]*.

TESTS:

```
sage: codes.QuadraticResidueCodeEvenPair(14,Zmod(4))
Traceback (most recent call last):
...
```

```
ValueError: the argument F must be a finite field
sage: codes.QuadraticResidueCodeEvenPair(14,GF(2))
Traceback (most recent call last):
...
ValueError: the argument n must be an odd prime
sage: codes.QuadraticResidueCodeEvenPair(5,GF(2))
Traceback (most recent call last):
...
ValueError: the order of the finite field must be a quadratic residue modulo n
```

sage.coding.code_constructions. **QuadraticResidueCodeOddPair** (n, F)

Quadratic residue codes of a given odd prime length and base ring either don't exist at all or occur as 4-tuples - a pair of "odd-like" codes and a pair of "even-like" codes. If n 2 is prime then (Theorem 6.6.2 in *[HP]*) a QR code exists over GF(q) iff q is a quadratic residue mod n.

They are constructed as "odd-like" duadic codes associated the splitting (Q,N) mod n, where Q is the set of non-zero quadratic residues and N is the non-residues.

EXAMPLES:

```
sage: codes.QuadraticResidueCodeOddPair(17,GF(13))
(Linear code of length 17, dimension 9 over Finite Field of size 13,
 Linear code of length 17, dimension 9 over Finite Field of size 13)
sage: codes.QuadraticResidueCodeOddPair(17,GF(2))
(Linear code of length 17, dimension 9 over Finite Field of size 2,
 Linear code of length 17, dimension 9 over Finite Field of size 2)
sage: codes.QuadraticResidueCodeOddPair(13,GF(9,"z"))
(Linear code of length 13, dimension 7 over Finite Field in z of size 3^2,
 Linear code of length 13, dimension 7 over Finite Field in z of size 3^2)
sage: C1 = codes.QuadraticResidueCodeOddPair(17,GF(2))[1]
sage: C1x = C1.extended_code()
sage: C2 = codes.QuadraticResidueCodeOddPair(17,GF(2))[0]
sage: C2x = C2.extended_code()
sage: C2x.spectrum(); C1x.spectrum()
[1, 0, 0, 0, 0, 0, 102, 0, 153, 0, 153, 0, 102, 0, 0, 0, 0, 0, 1]
[1, 0, 0, 0, 0, 0, 102, 0, 153, 0, 153, 0, 102, 0, 0, 0, 0, 0, 1]
sage: C3 = codes.QuadraticResidueCodeOddPair(7,GF(2))[0]
sage: C3x = C3.extended_code()
sage: C3x.spectrum()
[1, 0, 0, 0, 14, 0, 0, 0, 1]
```

This is consistent with Theorem 6.6.14 in *[HP]*.

TESTS:

```
sage: codes.QuadraticResidueCodeOddPair(9,GF(2))
Traceback (most recent call last):
...
ValueError: the argument n must be an odd prime
```

sage.coding.code_constructions. **RandomLinearCode** (n, k, F)

Deprecated alias of *random_linear_code()* .

EXAMPLES:

```
sage: C = codes.RandomLinearCode(10, 3, GF(2))
doctest:...: DeprecationWarning: codes.RandomLinearCode(n, k, F) is deprecated.␣
↪Please use codes.random_linear_code(F, n, k) instead
See http://trac.sagemath.org/21165 for details.
```

```
sage: C
Linear code of length 10, dimension 3 over Finite Field of size 2
sage: C.generator_matrix().rank()
3
```

sage.coding.code_constructions. **ReedSolomonCode** (*n*, *k*, *F*, *pts=None*)

sage.coding.code_constructions. **TernaryGolayCode** ( )

TernaryGolayCode returns a ternary Golay code. This is a perfect [11,6,5] code. It is also equivalent to a cyclic code, with generator polynomial $g(x) = 2 + x^2 + 2x^3 + x^4 + x^5$.

EXAMPLES:

```
sage: C = codes.TernaryGolayCode()
sage: C
Linear code of length 11, dimension 6 over Finite Field of size 3
sage: C.minimum_distance()
5
sage: C.minimum_distance(algorithm='gap') # long time, check d=5
5
```

AUTHORS:

- David Joyner (2007-5)

sage.coding.code_constructions. **ToricCode** (*P*, *F*)

Let P denote a list of lattice points in $\mathbf{Z}^d$ and let T denote the set of all points in $(F^x)^d$ (ordered in some fixed way). Put $n = |T|$ and let k denote the dimension of the vector space of functions $V = \mathrm{Span}\{x^e \mid e \in P\}$. The associated toric code C is the evaluation code which is the image of the evaluation map

$$\mathrm{eval_T} : V \to F^n,$$

where x^e is the multi-index notation ($x = (x_1, ..., x_d)$, $e = (e_1, ..., e_d)$, and $x^e = x_1^{e_1} ... x_d^{e_d}$), where $eval_T(f(x)) = (f(t_1), ..., f(t_n))$, and where $T = \{t_1, ..., t_n\}$. This function returns the toric codes discussed in *[J]*.

INPUT:

- `P` - all the integer lattice points in a polytope defining the toric variety.
- `F` - a finite field.

OUTPUT: Returns toric code with length n = , dimension k over field F.

EXAMPLES:

```
sage: C = codes.ToricCode([[0,0],[1,0],[2,0],[0,1],[1,1]],GF(7))
sage: C
Linear code of length 36, dimension 5 over Finite Field of size 7
sage: C.minimum_distance()
24
sage: C = codes.ToricCode([[-2,-2],[-1,-2],[-1,-1],[-1,0],[0,-1],[0,0],[0,1],[1,-
↪1],[1,0]],GF(5))
sage: C
Linear code of length 16, dimension 9 over Finite Field of size 5
sage: C.minimum_distance()
6
sage: C = codes.ToricCode([␣
↪[0,0],[1,1],[1,2],[1,3],[1,4],[2,1],[2,2],[2,3],[3,1],[3,2],[4,1]],GF(8,"a"))
sage: C
Linear code of length 49, dimension 11 over Finite Field in a of size 2^3
```

This is in fact a [49,11,28] code over GF(8). If you type next `C.minimum_distance()` and wait overnight (!), you should get 28.

AUTHOR:

- David Joyner (07-2006)

REFERENCES:

sage.coding.code_constructions. **WalshCode** (m)

Returns the binary Walsh code of length 2^m. The matrix of codewords correspond to a Hadamard matrix. This is a (constant rate) binary linear $[2^m, m, 2^{m-1}]$ code.

EXAMPLES:

```
sage: C = codes.WalshCode(4); C
Linear code of length 16, dimension 4 over Finite Field of size 2
sage: C = codes.WalshCode(3); C
Linear code of length 8, dimension 3 over Finite Field of size 2
sage: C.spectrum()
[1, 0, 0, 0, 7, 0, 0, 0]
sage: C.minimum_distance()
4
sage: C.minimum_distance(algorithm='gap') # check d=2^(m-1)
4
```

REFERENCES:

- http://en.wikipedia.org/wiki/Hadamard_matrix
- http://en.wikipedia.org/wiki/Walsh_code

sage.coding.code_constructions. **from_parity_check_matrix** (H)

Return the linear code that has `H` as a parity check matrix.

If `H` has dimensions $h \times n$ then the linear code will have dimension $n - h$ and length n.

EXAMPLES:

```
sage: C = codes.HammingCode(GF(2), 3); C
[7, 4] Hamming Code over Finite Field of size 2
sage: H = C.parity_check_matrix(); H
[1 0 1 0 1 0 1]
[0 1 1 0 0 1 1]
[0 0 0 1 1 1 1]
sage: C2 = codes.from_parity_check_matrix(H); C2
Linear code of length 7, dimension 4 over Finite Field of size 2
sage: C2.systematic_generator_matrix() == C.systematic_generator_matrix()
True
```

sage.coding.code_constructions. **lift2smallest_field2** (a)

INPUT: a is an element of a finite field GF(q)

OUTPUT: the element b of the smallest subfield F of GF(q) for which F(b)=a.

EXAMPLES:

```
sage: from sage.coding.code_constructions import lift2smallest_field2
sage: FF.<z> = GF(3^4,"z")
sage: a = z^40
sage: lift2smallest_field2(a)
doctest:...: DeprecationWarning: lift2smallest_field2 will be removed in a future␣
↪release of Sage. Consider using sage.coding.code_constructions._lift2smallest_
↪field instead, though this is private and may be removed in the future without␣
↪deprecation warning. If you care about this functionality being in Sage,␣
↪[illegible] public.
```

```
See http://trac.sagemath.org/21165 for details.
(2, Finite Field of size 3)
sage: FF.<z> = GF(2^4,"z")
sage: a = z^15
sage: lift2smallest_field2(a)
(1, Finite Field of size 2)
```

Warning: Since coercion (the FF(b) step) has a bug in it, this *only works* in the case when you *know* F is a prime field.

AUTHORS:

- David Joyner

sage.coding.code_constructions.**permutation_action**(*g*, *v*)

Returns permutation of rows g*v. Works on lists, matrices, sequences and vectors (by permuting coordinates). The code requires switching from i to i+1 (and back again) since the SymmetricGroup is, by convention, the symmetric group on the "letters" 1, 2, ..., n (not 0, 1, ..., n-1).

EXAMPLES:

```
sage: V = VectorSpace(GF(3),5)
sage: v = V([0,1,2,0,1])
sage: G = SymmetricGroup(5)
sage: g = G([(1,2,3)])
sage: permutation_action(g,v)
(1, 2, 0, 0, 1)
sage: g = G([()])
sage: permutation_action(g,v)
(0, 1, 2, 0, 1)
sage: g = G([(1,2,3,4,5)])
sage: permutation_action(g,v)
(1, 2, 0, 1, 0)
sage: L = Sequence([1,2,3,4,5])
sage: permutation_action(g,L)
[2, 3, 4, 5, 1]
sage: MS = MatrixSpace(GF(3),3,7)
sage: A = MS([[1,0,0,0,1,1,0],[0,1,0,1,0,1,0],[0,0,0,0,0,0,1]])
sage: S5 = SymmetricGroup(5)
sage: g = S5([(1,2,3)])
sage: A
[1 0 0 0 1 1 0]
[0 1 0 1 0 1 0]
[0 0 0 0 0 0 1]
sage: permutation_action(g,A)
[0 1 0 1 0 1 0]
[0 0 0 0 0 0 1]
[1 0 0 0 1 1 0]
```

It also works on lists and is a "left action":

```
sage: v = [0,1,2,0,1]
sage: G = SymmetricGroup(5)
sage: g = G([(1,2,3)])
sage: gv = permutation_action(g,v); gv
[1, 2, 0, 0, 1]
```

```
sage: permutation_action(g,v) == g(v)
True
sage: h = G([(3,4)])
sage: gv = permutation_action(g,v)
sage: hgv = permutation_action(h,gv)
sage: hgv == permutation_action(h*g,v)
True
```

AUTHORS:

- David Joyner, licensed under the GPL v2 or greater.

sage.coding.code_constructions. **random_linear_code** (*F*, *length*, *dimension*)

Generate a random linear code of length `length` , dimension `dimension` and over the field `F` .

This function is Las Vegas probabilistic: always correct, usually fast. Random matrices over the `F` are drawn until one with full rank is hit.

If `F` is infinite, the distribution of the elements in the random generator matrix will be random according to the distribution of `F.random_element()` .

EXAMPLES:

```
sage: C = codes.random_linear_code(GF(2), 10, 3)
sage: C
Linear code of length 10, dimension 3 over Finite Field of size 2
sage: C.generator_matrix().rank()
3
```

sage.coding.code_constructions. **walsh_matrix** (*m0*)

This is the generator matrix of a Walsh code. The matrix of codewords correspond to a Hadamard matrix.

EXAMPLES:

```
sage: walsh_matrix(2)
[0 0 1 1]
[0 1 0 1]
sage: walsh_matrix(3)
[0 0 0 0 1 1 1 1]
[0 0 1 1 0 0 1 1]
[0 1 0 1 0 1 0 1]
sage: C = LinearCode(walsh_matrix(4)); C
Linear code of length 16, dimension 4 over Finite Field of size 2
sage: C.spectrum()
[1, 0, 0, 0, 0, 0, 0, 0, 15, 0, 0, 0, 0, 0, 0, 0, 0]
```

This last code has minimum distance 8.

REFERENCES:

- http://en.wikipedia.org/wiki/Hadamard_matrix

2.9 Punctured code

Let C be a linear code. Let C_i be the set of all words of C with the i-th coordinate being removed. C_i is the punctured code of C on the i-th position.

class sage.coding.punctured_code. **PuncturedCode** (*C*, *positions*)

Bases: *sage.coding.linear_code.AbstractLinearCode*

Representation of a punctured code.

- `C` – A linear code
- `positions` – the positions where `C` will be punctured. It can be either an integer if one need to puncture only one position, a list or a set of positions to puncture. If the same position is passed several times, it will be considered only once.

EXAMPLES:

```
sage: C = codes.random_linear_code(GF(7), 11, 5)
sage: Cp = codes.PuncturedCode(C, 3)
sage: Cp
Punctured code coming from Linear code of length 11, dimension 5 over Finite␣
↪Field of size 7 punctured on position(s) [3]

sage: Cp = codes.PuncturedCode(C, {3, 5})
sage: Cp
Punctured code coming from Linear code of length 11, dimension 5 over Finite␣
↪Field of size 7 punctured on position(s) [3, 5]
```

dimension ( )

Returns the dimension of `self` .

EXAMPLES:

```
sage: set_random_seed(42)
sage: C = codes.random_linear_code(GF(7), 11, 5)
sage: Cp = codes.PuncturedCode(C, 3)
sage: Cp.dimension()
5
```

encode (*m*, *original_encode=False*, *encoder_name=None*, ***kwargs*)

Transforms an element of the message space into an element of the code.

INPUT:

- `m` – a vector of the message space of the code.
- `original_encode` – (default: `False`) if this is set to `True` , `m` will be encoded using an Encoder of `self` 's *`original_code()`* . This allow to avoid the computation of a generator matrix for `self` .
- `encoder_name` – (default: `None`) Name of the encoder which will be used to encode `word` . The default encoder of `self` will be used if default value is kept

OUTPUT:

- an element of `self`

EXAMPLES:

```
sage: M = matrix(GF(7), [[1, 0, 0, 0, 3, 4, 6], [0, 1, 0, 6, 1, 6, 4], [0, 0,␣
↪1, 5, 2, 2, 4]])
sage: C_original = LinearCode(M)
sage: Cp = codes.PuncturedCode(C_original, 2)
sage: m = vector(GF(7), [1, 3, 5])
sage: Cp.encode(m)
 (1, 3, 5, 5, 0, 2)
```

original_code ( )

Returns the linear code which was punctured to get `self` .

EXAMPLES:

```
sage: C = codes.random_linear_code(GF(7), 11, 5)
sage: Cp = codes.PuncturedCode(C, 3)
sage: Cp.original_code()
Linear code of length 11, dimension 5 over Finite Field of size 7
```

punctured_positions ( )

Returns the list of positions which were punctured on the original code.

EXAMPLES:

```
sage: C = codes.random_linear_code(GF(7), 11, 5)
sage: Cp = codes.PuncturedCode(C, 3)
sage: Cp.punctured_positions()
{3}
```

random_element (**args*, ***kwds*)

Returns a random codeword of `self` .

This method does not trigger the computation of `self` 's `sage.coding.linear_code.generator_matrix()` .

INPUT:

- `agrs` , `kwds` - extra positional arguments passed to `sage.modules.free_module.random_element()` .

EXAMPLES:

```
sage: C = codes.random_linear_code(GF(7), 11, 5)
sage: Cp = codes.PuncturedCode(C, 3)
sage: Cp.random_element() in Cp
True
```

structured_representation ( )

Returns `self` as a structured code object.

If `self` has a specific structured representation (e.g. a punctured GRS code is a GRS code too), it will return this representation, else it returns a *sage.coding.linear_code.LinearCode* .

EXAMPLES:

We consider a GRS code:

```
sage: C_grs = codes.GeneralizedReedSolomonCode(GF(59).list()[:40], 12)
```

A punctured GRS code is still a GRS code:

```
sage: Cp_grs = codes.PuncturedCode(C_grs, 3)
sage: Cp_grs.structured_representation()
[39, 12, 28] Generalized Reed-Solomon Code over Finite Field of size 59
```

Another example with structureless linear codes:

```
sage: set_random_seed(42)
sage: C_lin  = codes.random_linear_code(GF(2), 10, 5)
sage: Cp_lin = codes.PuncturedCode(C_lin, 2)
sage: Cp_lin.structured_representation()
Linear code of length 9, dimension 5 over Finite Field of size 2
```

class sage.coding.punctured_code. **PuncturedCodeOriginalCodeDecoder** (*code*, *strategy=None*, *original_decoder=None*, ***kwargs*)

Bases: *sage.coding.decoder.Decoder*

Decoder decoding through a decoder over the original code of its punctured code.

INPUT:

- `code` – The associated code of this encoder
- `strategy` – (default: `None`) the strategy used to decode. The available strategies are:
  - **`'error-erasure'` – uses an error-erasure decoder over the original code if available,** fails otherwise.
  - **`'random-values'` – fills the punctured positions with random elements** in `code` 's base field and tries to decode using the default decoder of the original code
  - **`'try-all'` – fills the punctured positions with every possible combination of** symbols until decoding succeeds, or until every combination have been tried
  - **`None` – uses `error-erasure` if an error-erasure decoder is available,** switch to `random-values` behaviour otherwise
- `original_decoder` – (default: `None`) the decoder that will be used over the original code. It has to be a decoder object over the original code. This argument takes precedence over `strategy` : if both `original_decoder` and `strategy` are filled, `self` will use the `original_decoder` to decode over the original code. If `original_decoder` is set to `None` , it will use the decoder picked by `strategy` .
- `**kwargs` – all extra arguments are forwarded to original code's decoder

EXAMPLES:

```
sage: C = codes.GeneralizedReedSolomonCode(GF(16, 'a').list()[:15], 7)
sage: Cp = codes.PuncturedCode(C, 3)
sage: codes.decoders.PuncturedCodeOriginalCodeDecoder(Cp)
Decoder of Punctured code coming from [15, 7, 9] Generalized Reed-
↪Solomon Code over Finite Field in a of size 2^4 punctured on␣
↪position(s) [3] through Error-Erasure decoder for [15, 7, 9]␣
↪Generalized Reed-Solomon Code over Finite Field in a of size 2^4
```

As seen above, if all optional are left blank, and if an error-erasure decoder is available, it will be chosen as the original decoder. Now, if one forces `strategy `` to ``'try-all'` or `'random-values'` , the default decoder of the original code will be chosen, even if an error-erasure is available:

```
sage: C = codes.GeneralizedReedSolomonCode(GF(16, 'a').list()[:15], 7)
sage: Cp = codes.PuncturedCode(C, 3)
sage: D = codes.decoders.PuncturedCodeOriginalCodeDecoder(Cp, strategy=
↪"try-all")
sage: "error-erasure" in D.decoder_type()
False
```

And if one fills `original_decoder` and `strategy` fields with contradictory elements, the `original_decoder` takes precedence:

```
sage: C = codes.GeneralizedReedSolomonCode(GF(16, 'a').list()[:15], 7)
sage: Cp = codes.PuncturedCode(C, 3)
sage: Dor = C.decoder("Gao")
sage: D = codes.decoders.PuncturedCodeOriginalCodeDecoder(Cp, original_
↪decoder = Dor, strategy="error-erasure")
sage: D.original_decoder() == Dor
True
```

decode_to_code (*y*)

Decodes `y` to an element in *sage.coding.decoder.Decoder.code()* .

EXAMPLES:

```
sage: C = codes.GeneralizedReedSolomonCode(GF(16, 'a').list()[:15], 7)
sage: Cp = codes.PuncturedCode(C, 3)
sage: D = codes.decoders.PuncturedCodeOriginalCodeDecoder(Cp)
sage: c = Cp.random_element()
sage: Chan = channels.StaticErrorRateChannel(Cp.ambient_space(), 3)
sage: y = Chan(c)
sage: y in Cp
False
sage: D.decode_to_code(y) == c
True
```

decoding_radius (*number_erasures=None*)

Returns maximal number of errors that `self` can decode.

EXAMPLES:

```
sage: C = codes.GeneralizedReedSolomonCode(GF(16, 'a').list()[:15], 7)
sage: Cp = codes.PuncturedCode(C, 3)
sage: D = codes.decoders.PuncturedCodeOriginalCodeDecoder(Cp)
sage: D.decoding_radius(2)
2
```

original_decoder ( )

Returns the decoder over the original code that will be used to decode words of *sage.coding.decoder.Decoder.code()* .

EXAMPLES:

```
sage: C = codes.GeneralizedReedSolomonCode(GF(16, 'a').list()[:15], 7)
sage: Cp = codes.PuncturedCode(C, 3)
sage: D = codes.decoders.PuncturedCodeOriginalCodeDecoder(Cp)
sage: D.original_decoder()
Error-Erasure decoder for [15, 7, 9] Generalized Reed-Solomon Code over␣
↪Finite Field in a of size 2^4
```

class `sage.coding.punctured_code.` **PuncturedCodePuncturedMatrixEncoder** (*code*)

Bases: *sage.coding.encoder.Encoder*

Encoder using original code generator matrix to compute the punctured code's one.

INPUT:

- `code` – The associated code of this encoder.

EXAMPLES:: sage: C = codes.random_linear_code(GF(7), 11, 5) sage: Cp = codes.PuncturedCode(C, 3) sage: E =

codes.encoders.PuncturedCodePuncturedMatrixEncoder(Cp) sage: E Punctured matrix-based encoder for the Punctured code coming from Linear code of length 11, dimension 5 over Finite Field of size 7 punctured on position(s) [3]

generator_matrix ( )

Returns a generator matrix of the associated code of `self` .

EXAMPLES:

```
sage: set_random_seed(10)
sage: C = codes.random_linear_code(GF(7), 11, 5)
sage: Cp = codes.PuncturedCode(C, 3)
sage: E = codes.encoders.PuncturedCodePuncturedMatrixEncoder(Cp)
sage: E.generator_matrix()
[1 0 0 0 0 5 2 6 0 6]
[0 1 0 0 0 5 2 2 1 1]
[0 0 1 0 0 6 2 4 0 4]
[0 0 0 1 0 0 6 3 3 3]
[0 0 0 0 1 0 1 3 4 3]
```

2.10 Extended code

Let C be a linear code of length n over $\mathbb{F}_q$. The extended code of C is the code

$$\hat{C} = \{x_1 x_2 \dots x_{n+1} \in \mathbb{F}_q^{n+1} \,|\, x_1 x_2 \dots x_n \in C \text{ with } x_1 + x_2 + \cdots + x_{n+1} = 0\}.$$

See *[HP]* (pp 15-16) for details.

class sage.coding.extended_code. **ExtendedCode** (C)

Bases: *sage.coding.linear_code.AbstractLinearCode*

Representation of an extended code.

INPUT:

- `C` – A linear code

EXAMPLES:

```
sage: C = codes.random_linear_code(GF(7), 11, 5)
sage: Ce = codes.ExtendedCode(C)
sage: Ce
Extended code coming from Linear code of length 11, dimension 5 over Finite Field␣
↪of size 7
```

original_code ( )

Returns the code which was extended to get `self` .

EXAMPLES:

```
sage: C = codes.random_linear_code(GF(7), 11, 5)
sage: Ce = codes.ExtendedCode(C)
sage: Ce.original_code()
Linear code of length 11, dimension 5 over Finite Field of size 7
```

parity_check_matrix ( )

Returns a parity check matrix of `self` .

This matrix is computed directly from *original_code()* .

EXAMPLES:

```
sage: C = LinearCode(matrix(GF(2),[[1,0,0,1,1],\
                                   [0,1,0,1,0],\
                                   [0,0,1,1,1]]))
sage: C.parity_check_matrix()
[1 0 1 0 1]
[0 1 0 1 1]
sage: Ce = codes.ExtendedCode(C)
sage: Ce.parity_check_matrix()
[1 1 1 1 1 1]
[1 0 1 0 1 0]
[0 1 0 1 1 0]
```

random_element ( )

Returns a random element of `self` .

This random element is computed directly from the original code, and does not compute a generator matrix of `self` in the process.

EXAMPLES:

```
sage: C = codes.random_linear_code(GF(7), 9, 5)
sage: Ce = codes.ExtendedCode(C)
sage: c = Ce.random_element() #random
sage: c in Ce
True
```

class sage.coding.extended_code. **ExtendedCodeExtendedMatrixEncoder** (*code*)

Bases: *sage.coding.encoder.Encoder*

Encoder using original code's generator matrix to compute the extended code's one.

INPUT:

- `code` – The associated code of `self` .

generator_matrix ( )

Returns a generator matrix of the associated code of `self` .

EXAMPLES:

```
sage: C = LinearCode(matrix(GF(2),[[1,0,0,1,1],\
                                   [0,1,0,1,0],\
                                   [0,0,1,1,1]]))
sage: Ce = codes.ExtendedCode(C)
sage: E = codes.encoders.ExtendedCodeExtendedMatrixEncoder(Ce)
sage: E.generator_matrix()
[1 0 0 1 1 1]
[0 1 0 1 0 0]
[0 0 1 1 1 1]
```

class sage.coding.extended_code. **ExtendedCodeOriginalCodeDecoder** (*code*, *original_decoder=None*, ***kwargs*)

Bases: *sage.coding.decoder.Decoder*

Decoder which decodes through a decoder over the original code.

INPUT:

- `code` – The associated code of this decoder

- `original_decoder` – (default: `None`) the decoder that will be used over the original code. It has to be a decoder object over the original code. If `original_decoder` is set to `None`, it will use the default decoder of the original code.
- `**kwargs` – all extra arguments are forwarded to original code's decoder

EXAMPLES:

```
sage: C = codes.GeneralizedReedSolomonCode(GF(16, 'a').list()[:15], 7)
sage: Ce = codes.ExtendedCode(C)
sage: D = codes.decoders.ExtendedCodeOriginalCodeDecoder(Ce)
sage: D
Decoder of Extended code coming from [15, 7, 9] Generalized Reed-Solomon Code␣
↪over Finite Field in a of size 2^4 through Gao decoder for [15, 7, 9]␣
↪Generalized Reed-Solomon Code over Finite Field in a of size 2^4
```

decode_to_code(*y*, ***kwargs*)

Decodes `y` to an element in *sage.coding.decoder.Decoder.code()*.

EXAMPLES:

```
sage: C = codes.GeneralizedReedSolomonCode(GF(16, 'a').list()[:15], 7)
sage: Ce = codes.ExtendedCode(C)
sage: D = codes.decoders.ExtendedCodeOriginalCodeDecoder(Ce)
sage: c = Ce.random_element()
sage: Chan = channels.StaticErrorRateChannel(Ce.ambient_space(), D.decoding_
↪radius())
sage: y = Chan(c)
sage: y in Ce
False
sage: D.decode_to_code(y) == c
True
```

Another example, with a list decoder:

```
sage: C = codes.GeneralizedReedSolomonCode(GF(16, 'a').list()[:15], 7)
sage: Ce = codes.ExtendedCode(C)
sage: Dgrs = C.decoder('GuruswamiSudan', tau = 4)
sage: D = codes.decoders.ExtendedCodeOriginalCodeDecoder(Ce, original_decoder␣
↪= Dgrs)
sage: c = Ce.random_element()
sage: Chan = channels.StaticErrorRateChannel(Ce.ambient_space(), D.decoding_
↪radius())
sage: y = Chan(c)
sage: y in Ce
False
sage: c in D.decode_to_code(y)
True
```

decoding_radius(**args*, ***kwargs*)

Returns maximal number of errors that `self` can decode.

INPUT:

- `*args`, `**kwargs` – arguments and optional arguments are forwarded to original decoder's `decoding_radius` method.

EXAMPLES:

```
sage: C = codes.GeneralizedReedSolomonCode(GF(16, 'a').list()[:15], 7)
sage: Ce = codes.ExtendedCode(C)
sage: D = codes.decoders.ExtendedCodeOriginalCodeDecoder(Ce)
sage: D.decoding_radius()
4
```

original_decoder ( )

Returns the decoder over the original code that will be used to decode words of *sage.coding.decoder.Decoder.code()* .

EXAMPLES:

```
sage: C = codes.GeneralizedReedSolomonCode(GF(16, 'a').list()[:15], 7)
sage: Ce = codes.ExtendedCode(C)
sage: D = codes.decoders.ExtendedCodeOriginalCodeDecoder(Ce)
sage: D.original_decoder()
Gao decoder for [15, 7, 9] Generalized Reed-Solomon Code over Finite Field in␣
↪a of size 2^4
```

2.11 Enumerating binary self-dual codes

This module implements functions useful for studying binary self-dual codes. The main function is `self_dual_binary_codes` , which is a case-by-case list of entries, each represented by a Python dictionary.

Format of each entry: a Python dictionary with keys "order autgp", "spectrum", "code", "Comment", "Type", where

- "code" - a sd code C of length n, dim n/2, over GF(2)
- "order autgp" - order of the permutation automorphism group of C
- "Type" - the type of C (which can be "I" or "II", in the binary case)
- "spectrum" - the spectrum [A0,A1,...,An]
- "Comment" - possibly an empty string.

Python dictionaries were used since they seemed to be both human-readable and allow others to update the database easiest.

- The following double for loop can be time-consuming but should be run once in awhile for testing purposes. It should only print True and have no trace-back errors:

```
for n in [4,6,8,10,12,14,16,18,20,22]:
    C = self_dual_binary_codes(n); m = len(C.keys())
    for i in range(m):
        C0 = C["%s"%n]["%s"%i]["code"]
        print([n,i,C["%s"%n]["%s"%i]["spectrum"] == C0.spectrum()])
        print(C0 == C0.dual_code())
        G = C0.automorphism_group_binary_code()
        print(C["%s" % n]["%s" % i]["order autgp"] == G.order())
```

- To check if the "Riemann hypothesis" holds, run the following code:

```
R = PolynomialRing(CC,"T")
T = R.gen()
for n in [4,6,8,10,12,14,16,18,20,22]:
    C = self_dual_binary_codes(n); m = len(C["%s"%n].keys())
    for i in range(m):
```

```
        C0 = C["%s"%n]["%s"%i]["code"]
        if C0.minimum_distance()>2:
            f = R(C0.sd_zeta_polynomial())
            print([n,i,[z[0].abs() for z in f.roots()]])
```

You should get lists of numbers equal to 0.707106781186548.

Here's a rather naive construction of self-dual codes in the binary case:

For even m, let A_m denote the mxm matrix over GF(2) given by adding the all 1's matrix to the identity matrix (in `MatrixSpace(GF(2),m,m)` of course). If M_1, ..., M_r are square matrices, let $diag(M_1, M_2, ..., M_r)$ denote the"block diagonal" matrix with the M_i 's on the diagonal and 0's elsewhere. Let $C(m_1, ..., m_r, s)$ denote the linear code with generator matrix having block form $G = (I, A)$, where $A = diag(A_{m_1}, A_{m_2}, ..., A_{m_r}, I_s)$, for some (even) m_i 's and s, where $m_1 + m_2 + ... + m_r + s = n/2$. Note: Such codes $C(m_1, ..., m_r, s)$ are SD.

SD codes not of this form will be called (for the purpose of documenting the code below) "exceptional". Except when n is "small", most sd codes are exceptional (based on a counting argument and table 9.1 in *[HP]*, page 347).

AUTHORS:

- David Joyner (2007-08-11)

REFERENCES:

- [P] V. Pless, "A classification of self-orthogonal codes over GF(2)", Discrete Math 3 (1972) 209-246.

`sage.coding.self_dual_codes.` **`self_dual_binary_codes`** `(n)`

Returns the dictionary of inequivalent binary self dual codes of length n.

For n=4 even, returns the sd codes of a given length, up to (perm) equivalence, the (perm) aut gp, and the type.

The number of inequiv "diagonal" sd binary codes in the database of length n is ("diagonal" is defined by the conjecture above) is the same as the restricted partition number of n, where only integers from the set 1,4,6,8,... are allowed. This is the coefficient of x^n in the series expansion $(1-x)^{-1} \prod_{2^\infty (1-x^{2j})^{-1}}$. Typing the command f = (1-x)(-1)*prod([(1-x(2*j))(-1) for j in range(2,18)]) into Sage, we obtain for the coeffs of x^4, x^6, ... [1, 1, 2, 2, 3, 3, 5, 5, 7, 7, 11, 11, 15, 15, 22, 22, 30, 30, 42, 42, 56, 56, 77, 77, 101, 101, 135, 135, 176, 176, 231] These numbers grow too slowly to account for all the sd codes (see *[HP]* Table 9.1). In fact, in Table 9.10 of *[HP]*, the number B_n of inequivalent sd binary codes of length n is given:

n	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30
B_n	1	1	1	2	2	3	4	7	9	16	25	55	103	261	731

According to http://oeis.org/classic/A003179, the next 2 entries are: 3295, 24147.

EXAMPLES:

```
sage: C = codes.databases.self_dual_binary_codes(10)
sage: C["10"]["0"]["code"] == C["10"]["0"]["code"].dual_code()
True
sage: C["10"]["1"]["code"] == C["10"]["1"]["code"].dual_code()
True
sage: len(C["10"].keys()) # number of inequiv sd codes of length 10
2
sage: C = codes.databases.self_dual_binary_codes(12)
sage: C["12"]["0"]["code"] == C["12"]["0"]["code"].dual_code()
True
sage: C["12"]["1"]["code"] == C["12"]["1"]["code"].dual_code()
True
```

```
sage: C["12"]["2"]["code"] == C["12"]["2"]["code"].dual_code()
True
```

2.12 Guava error-correcting code constructions

This module only contains Guava wrappers (Guava is an optional GAP package).

AUTHORS:

- David Joyner (2005-11-22, 2006-12-03): initial version
- Nick Alexander (2006-12-10): factor GUAVA code to guava.py
- David Joyner (2007-05): removed Golay codes, toric and trivial codes and placed them in code_constructions; renamed RandomLinearCode to RandomLinearCodeGuava
- David Joyner (2008-03): removed QR, XQR, cyclic and ReedSolomon codes
- David Joyner (2009-05): added "optional package" comments, fixed some docstrings to to be sphinx compatible

2.12.1 Functions

sage.coding.guava. **QuasiQuadraticResidueCode** (p)

A (binary) quasi-quadratic residue code (or QQR code), as defined by Proposition 2.2 in *[BM]*, has a generator matrix in the block form $G = (Q, N)$. Here Q is a $p \times p$ circulant matrix whose top row is $(0, x_1, ..., x_{p-1})$, where $x_i = 1$ if and only if i is a quadratic residue $\mod p$, and N is a $p \times p$ circulant matrix whose top row is $(0, y_1, ..., y_{p-1})$, where $x_i + y_i = 1$ for all i.

INPUT:

- p – a prime > 2.

OUTPUT:

Returns a QQR code of length $2p$.

EXAMPLES:

```
sage: C = codes.QuasiQuadraticResidueCode(11); C    # optional - gap_packages
↪(Guava package)
Linear code of length 22, dimension 11 over Finite Field of size 2
```

REFERENCES:

These are self-orthogonal in general and self-dual when p
$equiv 3$
$pmod 4$.

AUTHOR: David Joyner (11-2005)

sage.coding.guava. **RandomLinearCodeGuava** (n, k, F)

The method used is to first construct a $k \times n$ matrix of the block form (I, A), where I is a $k \times k$ identity matrix and A is a $k \times (n - k)$ matrix constructed using random elements of F. Then the columns are permuted using a randomly selected element of the symmetric group S_n.

INPUT:

- n, k – integers with $n > k > 1$.

OUTPUT:

Returns a "random" linear code with length n, dimension k over field F.

EXAMPLES:

```
sage: C = codes.RandomLinearCodeGuava(30,15,GF(2)); C      # optional - gap_
↪packages (Guava package)
Linear code of length 30, dimension 15 over Finite Field of size 2
sage: C = codes.RandomLinearCodeGuava(10,5,GF(4,'a')); C      # optional - gap_
↪packages (Guava package)
Linear code of length 10, dimension 5 over Finite Field in a of size 2^2
```

AUTHOR: David Joyner (11-2005)

2.13 Fast binary code routines

Some computations with linear binary codes. Fix a basis for $GF(2)^n$. A linear binary code is a linear subspace of $GF(2)^n$, together with this choice of basis. A permutation $g \in S_n$ of the fixed basis gives rise to a permutation of the vectors, or words, in $GF(2)^n$, sending (w_i) to $(w_{g(i)})$. The permutation automorphism group of the code C is the set of permutations of the basis that bijectively map C to itself. Note that if g is such a permutation, then

$$g(a_i) + g(b_i) = (a_{g(i)} + b_{g(i)}) = g((a_i) + (b_i)).$$

Over other fields, it is also required that the map be linear, which as per above boils down to scalar multiplication. However, over $GF(2)$, the only scalars are 0 and 1, so the linearity condition has trivial effect.

AUTHOR:

- Robert L Miller (Oct-Nov 2007)
- compiled code data structure
- union-find based orbit partition
- optimized partition stack class
- NICE-based partition refinement algorithm
- canonical generation function

class sage.coding.binary_code. **BinaryCode**
Bases: object

Minimal, but optimized, binary code object.

EXAMPLE:

```
sage: import sage.coding.binary_code
sage: from sage.coding.binary_code import *
sage: M = Matrix(GF(2), [[1,1,1,1]])
sage: B = BinaryCode(M)      # create from matrix
sage: C = BinaryCode(B, 60) # create using glue
sage: D = BinaryCode(C, 240)
sage: E = BinaryCode(D, 85)
sage: B
Binary [4,1] linear code, generator matrix
[1111]
sage: C
Binary [6,2] linear code, generator matrix
```

```
[111100]
[001111]
sage: D
Binary [8,3] linear code, generator matrix
[11110000]
[00111100]
[00001111]
sage: E
Binary [8,4] linear code, generator matrix
[11110000]
[00111100]
[00001111]
[10101010]

sage: M = Matrix(GF(2), [[1]*32])
sage: B = BinaryCode(M)
sage: B
Binary [32,1] linear code, generator matrix
[11111111111111111111111111111111]
```

apply_permutation (*labeling*)

Apply a column permutation to the code.

INPUT:

- labeling – a list permutation of the columns

EXAMPLE:

```
sage: from sage.coding.binary_code import *
sage: B = BinaryCode(codes.ExtendedBinaryGolayCode().generator_matrix())
sage: B
Binary [24,12] linear code, generator matrix
[100000000000101011100011]
[010000000000111110010010]
[001000000000110100101011]
[000100000000110001110110]
[000010000000110011011001]
[000001000000011001101101]
[000000100000001100110111]
[000000010000101101111000]
[000000001000010110111100]
[000000000100001011011110]
[000000000010101110001101]
[000000000001010111000111]
sage: B.apply_permutation(list(range(11,-1,-1)) + list(range(12, 24)))
sage: B
Binary [24,12] linear code, generator matrix
[000000000001101011100011]
[000000000010111110010010]
[000000000100110100101011]
[000000001000110001110110]
[000000010000110011011001]
[000000100000011001101101]
[000001000000001100110111]
[000010000000101101111000]
[000100000000010110111100]
[001000000000001011011110]
[010000000000101110001101]
```

```
[100000000000010111000111]
```

matrix ( )

Returns the generator matrix of the BinaryCode, i.e. the code is the rowspace of B.matrix().

EXAMPLE:

```
sage: M = Matrix(GF(2), [[1,1,1,1,0,0],[0,0,1,1,1,1]])
sage: from sage.coding.binary_code import *
sage: B = BinaryCode(M)
sage: B.matrix()
[1 1 1 1 0 0]
[0 0 1 1 1 1]
```

print_data ( )

Print all data for `self` .

EXAMPLES:

```
sage: import sage.coding.binary_code
sage: from sage.coding.binary_code import *
sage: M = Matrix(GF(2), [[1,1,1,1]])
sage: B = BinaryCode(M)
sage: C = BinaryCode(B, 60)
sage: D = BinaryCode(C, 240)
sage: E = BinaryCode(D, 85)
sage: B.print_data() # random - actually "print(P.print_data())"
ncols: 4
nrows: 1
nwords: 2
radix: 32
basis:
1111
words:
0000
1111
sage: C.print_data() # random - actually "print(P.print_data())"
ncols: 6
nrows: 2
nwords: 4
radix: 32
basis:
111100
001111
words:
000000
111100
001111
110011
sage: D.print_data() # random - actually "print(P.print_data())"
ncols: 8
nrows: 3
nwords: 8
radix: 32
basis:
11110000
00111100
00001111
```

```
words:
00000000
11110000
00111100
11001100
00001111
11111111
00110011
11000011
sage: E.print_data() # random - actually "print(P.print_data())"
ncols: 8
nrows: 4
nwords: 16
radix: 32
basis:
11110000
00111100
00001111
10101010
words:
00000000
11110000
00111100
11001100
00001111
11111111
00110011
11000011
10101010
01011010
10010110
01100110
10100101
01010101
10011001
01101001
```

put_in_std_form ( )

Put the code in binary form, which is defined by an identity matrix on the left, augmented by a matrix of data.

EXAMPLE:

```
sage: from sage.coding.binary_code import *
sage: M = Matrix(GF(2), [[1,1,1,1,0,0],[0,0,1,1,1,1]])
sage: B = BinaryCode(M); B
Binary [6,2] linear code, generator matrix
[111100]
[001111]
sage: B.put_in_std_form(); B
0
Binary [6,2] linear code, generator matrix
[101011]
[010111]
```

class sage.coding.binary_code. **BinaryCodeClassifier**

Bases: object

generate_children (*B*, *n*, *d=2*)

Use canonical augmentation to generate children of the code B.

INPUT:

- B – a BinaryCode
- n – limit on the degree of the code
- d – test whether new vector has weight divisible by d. If d==4, this ensures that all doubly-even canonically augmented children are generated.

EXAMPLE:

```
sage: from sage.coding.binary_code import *
sage: BC = BinaryCodeClassifier()
sage: B = BinaryCode(Matrix(GF(2), [[1,1,1,1]]))
sage: BC.generate_children(B, 6, 4)
[
[1 1 1 1 0 0]
[0 1 0 1 1 1]
]
```

Note: The function `codes.databases.self_orthogonal_binary_codes` makes heavy use of this function.

MORE EXAMPLES:

```
sage: soc_iter = codes.databases.self_orthogonal_binary_codes(12, 6, 4)
sage: L = list(soc_iter)
sage: for n in range(0, 13):
....:     s = 'n=%2d : '%n
....:     for k in range(1,7):
....:         s += '%3d '%len([C for C in L if C.length() == n and C.
↪dimension() == k])
....:     print(s)
n= 0 :   0   0   0   0   0   0
n= 1 :   0   0   0   0   0   0
n= 2 :   0   0   0   0   0   0
n= 3 :   0   0   0   0   0   0
n= 4 :   1   0   0   0   0   0
n= 5 :   0   0   0   0   0   0
n= 6 :   0   1   0   0   0   0
n= 7 :   0   0   1   0   0   0
n= 8 :   1   1   1   1   0   0
n= 9 :   0   0   0   0   0   0
n=10 :   0   1   1   1   0   0
n=11 :   0   0   1   1   0   0
n=12 :   1   2   3   4   2   0
```

put_in_canonical_form (*B*)

Puts the code into canonical form.

Canonical form is obtained by performing row reduction, permuting the pivots to the front so that the generator matrix is of the form: the identity matrix augmented to the right by arbitrary data.

EXAMPLE:

```
sage: from sage.coding.binary_code import *
sage: BC = BinaryCodeClassifier()
sage: B = BinaryCode(codes.ExtendedBinaryGolayCode().generator_matrix())
sage: B.apply_permutation(list(range(24,-1,-1)))
sage: B
Binary [24,12] linear code, generator matrix
[011000111010100000000000]
[001001001111100000000001]
[011010100101100000000010]
[001101110001100000000100]
[010011011001100000001000]
[010110110011000000010000]
[011101100110000000100000]
[000011110110100001000000]
[000111101101000010000000]
[001111011010000100000000]
[010110001110101000000000]
[011100011101010000000000]
sage: BC.put_in_canonical_form(B)
sage: B
Binary [24,12] linear code, generator matrix
[100000000000000011001111001]
[010000000000000010100011111]
[001000000000000011110100010]
[000100000000010110101010010]
[000010000000010110010101001]
[000001000000010001101101001]
[000000100000011000110110001]
[000000010000011111001001001]
[000000001000010101110011001]
[000000000100010011011110001]
[000000000010001011110101001]
[000000000001001101101110001]
```

class sage.coding.binary_code. **OrbitPartition**

Bases: object

Structure which keeps track of which vertices are equivalent under the part of the automorphism group that has already been seen, during search. Essentially a disjoint-set data structure*, which also keeps track of the minimum element and size of each cell of the partition, and the size of the partition.

- http://en.wikipedia.org/wiki/Disjoint-set_data_structure

class sage.coding.binary_code. **PartitionStack**

Bases: object

Partition stack structure for traversing the search tree during automorphism group computation.

cmp (*other*, *CG*)

EXAMPLES:

```
sage: import sage.coding.binary_code
sage: from sage.coding.binary_code import *
sage: M = Matrix(GF(2),␣
↪[[1,1,1,1,0,0,0,0],[0,0,1,1,1,1,0,0],[0,0,0,0,1,1,1,1],[1,0,1,0,1,0,1,0]])
sage: B = BinaryCode(M)
sage: P = PartitionStack(4, 8)
sage: P._refine(0, [[0,0],[1,0]], B)
181
```

```
sage: P._split_vertex(0, 1)
0
sage: P._refine(1, [[0,0]], B)
290
sage: P._split_vertex(1, 2)
1
sage: P._refine(2, [[0,1]], B)
463
sage: P._split_vertex(2, 3)
2
sage: P._refine(3, [[0,2]], B)
1500
sage: P._split_vertex(4, 4)
4
sage: P._refine(4, [[0,4]], B)
1224
sage: P._is_discrete(4)
1
sage: Q = PartitionStack(P)
sage: Q._clear(4)
sage: Q._split_vertex(5, 4)
4
sage: Q._refine(4, [[0,4]], B)
1224
sage: Q._is_discrete(4)
1
sage: Q.cmp(P, B)
0
```

print_basis ( )

EXAMPLE:

```
sage: import sage.coding.binary_code
sage: from sage.coding.binary_code import *
sage: P = PartitionStack(4, 8)
sage: P._dangerous_dont_use_set_ents_lvls(list(range(8)), list(range(7))+[-
↪1], [4,7,12,11,1,9,3,0,2,5,6,8,10,13,14,15], [0]*16)
sage: P
({4},{7},{12},{11},{1},{9},{3},{0},{2},{5},{6},{8},{10},{13},{14},{15})  ({0},
↪{1,2,3,4,5,6,7})
({4},{7},{12},{11},{1},{9},{3},{0},{2},{5},{6},{8},{10},{13},{14},{15})  ({0},
↪{1},{2,3,4,5,6,7})
({4},{7},{12},{11},{1},{9},{3},{0},{2},{5},{6},{8},{10},{13},{14},{15})  ({0},
↪{1},{2},{3,4,5,6,7})
({4},{7},{12},{11},{1},{9},{3},{0},{2},{5},{6},{8},{10},{13},{14},{15})  ({0},
↪{1},{2},{3},{4,5,6,7})
({4},{7},{12},{11},{1},{9},{3},{0},{2},{5},{6},{8},{10},{13},{14},{15})  ({0},
↪{1},{2},{3},{4},{5,6,7})
({4},{7},{12},{11},{1},{9},{3},{0},{2},{5},{6},{8},{10},{13},{14},{15})  ({0},
↪{1},{2},{3},{4},{5},{6,7})
({4},{7},{12},{11},{1},{9},{3},{0},{2},{5},{6},{8},{10},{13},{14},{15})  ({0},
↪{1},{2},{3},{4},{5},{6},{7})
sage: P._find_basis()
sage: P.print_basis()
basis_locations:
4
8
0
```

```
11
```

print_data ( )

Prints all data for self.

EXAMPLE:

```
sage: import sage.coding.binary_code
sage: from sage.coding.binary_code import *
sage: P = PartitionStack(2, 6)
sage: print(P.print_data())
nwords:4
nrows:2
ncols:6
radix:32
wd_ents:
0
1
2
3
wd_lvls:
12
12
12
-1
col_ents:
0
1
2
3
4
5
col_lvls:
12
12
12
12
12
-1
col_degs:
0
0
0
0
0
0
col_counts:
0
0
0
0
col_output:
0
0
0
0
0
0
```

```
wd_degs:
0
0
0
0
wd_counts:
0
0
0
0
0
0
0
wd_output:
0
0
0
0
```

sage.coding.binary_code. **test_expand_to_ortho_basis** (*B=None*)

This function is written in pure C for speed, and is tested from this function.

INPUT:

- B – a BinaryCode in standard form

OUTPUT:

An array of codewords which represent the expansion of a basis for B to a basis for $(B')^{\perp}$, where $B' = B$ if the all-ones vector 1 is in B, otherwise $B' = extspan(B, 1)$ (note that this guarantees that all the vectors in the span of the output have even weight).

TESTS:

```
sage: from sage.coding.binary_code import test_expand_to_ortho_basis, BinaryCode
sage: M = Matrix(GF(2), [[1,1,1,1,1,1,0,0,0,0],[0,0,1,1,1,1,1,1,1,1]])
sage: B = BinaryCode(M)
sage: B.put_in_std_form()
0
sage: test_expand_to_ortho_basis(B=B)
INPUT CODE:
Binary [10,2] linear code, generator matrix
[1010001111]
[0101111111]
Expanding to the basis of an orthogonal complement...
Basis:
0010000010
0001000010
0000100001
0000010001
0000001001
```

sage.coding.binary_code. **test_word_perms** (*t_limit=5.0*)

Tests the WordPermutation structs for at least t_limit seconds.

These are structures written in pure C for speed, and are tested from this function, which performs the following tests:

1. **Tests create_word_perm, which creates a WordPermutation from a Python** list L representing a permutation i –> L[i]. Takes a random word and permutes it by a random list permutation, and tests that

the result agrees with doing it the slow way.

1b. Tests create_array_word_perm, which creates a WordPermutation from a C array. Does the same as above.

2. **Tests create_comp_word_perm, which creates a WordPermutation as a** composition of two WordPermutations. Takes a random word and two random permutations, and tests that the result of permuting by the composition is correct.

3. **Tests create_inv_word_perm and create_id_word_perm, which create a** WordPermutation as the inverse and identity permutations, resp. Takes a random word and a random permutation, and tests that the result permuting by the permutation and its inverse in either order, and permuting by the identity both return the original word.

Note: The functions permute_word_by_wp and dealloc_word_perm are implicitly involved in each of the above tests.

TESTS:

```
sage: from sage.coding.binary_code import test_word_perms
sage: test_word_perms()  # long time (5s on sage.math, 2011)
```

sage.coding.binary_code. **weight_dist** (*M*)

Computes the weight distribution of the row space of M.

EXAMPLES:

```
sage: from sage.coding.binary_code import weight_dist
sage: M = Matrix(GF(2),[
....:  [1,1,1,1,1,1,1,1,0,0,0,0,0,0,0,0],
....:  [0,0,0,0,1,1,1,1,1,1,1,1,0,0,0,0],
....:  [0,0,0,0,0,0,0,0,1,1,1,1,1,1,1,1],
....:  [0,0,1,1,0,0,1,1,0,0,1,1,0,0,1,1],
....:  [0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1]])
sage: weight_dist(M)
[1, 0, 0, 0, 0, 0, 0, 0, 30, 0, 0, 0, 0, 0, 0, 0, 1]
sage: M = Matrix(GF(2),[
....:  [1,1,1,1,1,1,1,1,0,0,0,0,0,0,0,0],
....:  [0,0,0,0,0,0,1,1,1,1,1,1,1,1,0,0,0],
....:  [0,0,0,0,0,1,0,1,0,0,0,1,1,1,1,1,1],
....:  [0,0,0,1,1,0,0,0,0,1,1,0,1,1,0,1,1]])
sage: weight_dist(M)
[1, 0, 0, 0, 0, 0, 0, 0, 11, 0, 0, 0, 4, 0, 0, 0, 0, 0]
sage: M=Matrix(GF(2),[
....:  [1,0,0,1,1,1,1,0,0,1,0,0,0,0,0,0,0],
....:  [0,1,0,0,1,1,1,1,0,0,1,0,0,0,0,0,0],
....:  [0,0,1,0,0,1,1,1,1,0,0,1,0,0,0,0,0],
....:  [0,0,0,1,0,0,1,1,1,1,0,0,1,0,0,0,0],
....:  [0,0,0,0,1,0,0,1,1,1,1,0,0,1,0,0,0],
....:  [0,0,0,0,0,1,0,0,1,1,1,1,0,0,1,0,0],
....:  [0,0,0,0,0,0,1,0,0,1,1,1,1,0,0,1,0],
....:  [0,0,0,0,0,0,0,1,0,0,1,1,1,1,0,0,1]])
sage: weight_dist(M)
[1, 0, 0, 0, 0, 0, 68, 0, 85, 0, 68, 0, 34, 0, 0, 0, 0, 0]
```

2.14 Reed-Muller code

Given integers m, r and a finite field F, the corresponding Reed-Muller Code is the set:

$$\{(f(\alpha_i) \mid \alpha_i \in F^m) \mid f \in F[x_1, x_2, \ldots, x_m], \deg f \leq r\}$$

This file contains the following elements:

- *QAryReedMullerCode*, the class for Reed-Muller codes over non-binary field of size q and $r < q$
- *BinaryReedMullerCode*, the class for Reed-Muller codes over binary field and $r <= m$
- *ReedMullerVectorEncoder*, an encoder with a vectorial message space (for both the two code classes)
- *ReedMullerPolynomialEncoder*, an encoder with a polynomial message space (for both the code classes)

class sage.coding.reed_muller_code. **BinaryReedMullerCode** (*order*, *num_of_var*)
Bases: *sage.coding.linear_code.AbstractLinearCode*

Representation of a binary Reed-Muller code.

For details on the definition of Reed-Muller codes, refer to *ReedMullerCode()*.

Note: It is better to use the aforementioned method rather than calling this class directly, as *ReedMullerCode()* creates either a binary or a q-ary Reed-Muller code according to the arguments it receives.

INPUT:

- `order` – The order of the Reed-Muller Code, i.e., the maximum degree of the polynomial to be used in the code.
- `num_of_var` – The number of variables used in the polynomial.

EXAMPLES:

A binary Reed-Muller code can be constructed by simply giving the order of the code and the number of variables:

```
sage: C = codes.BinaryReedMullerCode(2, 4)
sage: C
Binary Reed-Muller Code of order 2 and number of variables 4
```

minimum_distance ( )
Returns the minimum distance of `self`. The minimum distance of a binary Reed-Muller code of order d and number of variables m is q^{m-d}

EXAMPLES:

```
sage: C = codes.BinaryReedMullerCode(2, 4)
sage: C.minimum_distance()
4
```

number_of_variables ( )
Returns the number of variables of the polynomial ring used in `self`.

EXAMPLES:

```
sage: C = codes.BinaryReedMullerCode(2, 4)
sage: C.number_of_variables()
4
```

order ( )

Returns the order of `self` . Order is the maximum degree of the polynomial used in the Reed-Muller code.

EXAMPLES:

```
sage: C = codes.BinaryReedMullerCode(2, 4)
sage: C.order()
2
```

class sage.coding.reed_muller_code. **QAryReedMullerCode** (*base_field*, *order*, *num_of_var*)

Bases: *sage.coding.linear_code.AbstractLinearCode*

Representation of a q-ary Reed-Muller code.

For details on the definition of Reed-Muller codes, refer to *ReedMullerCode()* .

Note: It is better to use the aforementioned method rather than calling this class directly, as *ReedMullerCode()* creates either a binary or a q-ary Reed-Muller code according to the arguments it receives.

INPUT:

- `base_field` – A finite field, which is the base field of the code.
- `order` – The order of the Reed-Muller Code, i.e., the maximum degree of the polynomial to be used in the code.
- `num_of_var` – The number of variables used in polynomial.

Warning: For now, this implementation only supports Reed-Muller codes whose order is less than q.

EXAMPLES:

```
sage: from sage.coding.reed_muller_code import QAryReedMullerCode
sage: F = GF(3)
sage: C = QAryReedMullerCode(F, 2, 2)
sage: C
Reed-Muller Code of order 2 and 2 variables over Finite Field of size 3
```

minimum_distance ( )

Returns the minimum distance between two words in `self` .

The minimum distance of a q-ary Reed-Muller code with order d and number of variables m is $(q-d)q^{m-1}$

EXAMPLES:

```
sage: from sage.coding.reed_muller_code import QAryReedMullerCode
sage: F = GF(5)
sage: C = QAryReedMullerCode(F, 2, 4)
sage: C.minimum_distance()
375
```

number_of_variables ( )

Returns the number of variables of the polynomial ring used in `self` .

EXAMPLES:

```
sage: from sage.coding.reed_muller_code import QAryReedMullerCode
sage: F = GF(59)
sage: C = QAryReedMullerCode(F, 2, 4)
sage: C.number_of_variables()
4
```

order ( )

Returns the order of `self` .

Order is the maximum degree of the polynomial used in the Reed-Muller code.

EXAMPLES:

```
sage: from sage.coding.reed_muller_code import QAryReedMullerCode
sage: F = GF(59)
sage: C = QAryReedMullerCode(F, 2, 4)
sage: C.order()
2
```

sage.coding.reed_muller_code. **ReedMullerCode** (*base_field*, *order*, *num_of_var*)

Returns a Reed-Muller code.

A Reed-Muller Code of order r and number of variables m over a finite field F is the set:

$$\{(f(\alpha_i) \mid \alpha_i \in F^m) \mid f \in F[x_1, x_2, \ldots, x_m], \deg f \leq r\}$$

INPUT:

- `base_field` – The finite field F over which the code is built.
- **`order` – The order of the Reed-Muller Code, which is the maximum** degree of the polynomial to be used in the code.
- `num_of_var` – The number of variables used in polynomial.

Warning: For now, this implementation only supports Reed-Muller codes whose order is less than q. Binary Reed-Muller codes must have their order less than or equal to their number of variables.

EXAMPLES:

We build a Reed-Muller code:

```
sage: F = GF(3)
sage: C = codes.ReedMullerCode(F, 2, 2)
sage: C
Reed-Muller Code of order 2 and 2 variables over Finite Field of size 3
```

We ask for its parameters:

```
sage: C.length()
9
sage: C.dimension()
6
sage: C.minimum_distance()
3
```

If one provides a finite field of size 2, a Binary Reed-Muller code is built:

```
sage: F = GF(2)
sage: C = codes.ReedMullerCode(F, 2, 2)
sage: C
Binary Reed-Muller Code of order 2 and number of variables 2
```

class sage.coding.reed_muller_code. **ReedMullerPolynomialEncoder** (*code*, *polynomial_ring=None*)

Bases: *sage.coding.encoder.Encoder*

Encoder for Reed-Muller codes which encodes appropiate multivariate polynomials into codewords.

Consider a Reed-Muller code of order r, number of variables m, length n, dimension k over some finite field F. Let those variables be $(x_1, x_2, \ldots, x_m)$. We order the monomials by lowest power on lowest index variables. If we have three monomials $x_1 \times x_2$, $x_1 \times x_2^2$ and $x_1^2 \times x_2$, the ordering is: $x_1 \times x_2 < x_1 \times x_2^2 < x_1^2 \times x_2$

Let now f be a polynomial of the multivariate polynomial ring $F[x_1, \ldots, x_m]$.

Let $(\beta_1, \beta_2, \ldots, \beta_q)$ be the elements of F ordered as they are returned by Sage when calling `F.list()` .

The aforementioned polynomial f is encoded as:

$(f(\alpha_{11}, \alpha_{12}, \ldots, \alpha_{1m}), f(\alpha_{21}, \alpha_{22}, \ldots, \alpha_{2m}), \ldots, f(\alpha_{q^m 1}, \alpha_{q^m 2}, \ldots, \alpha_{q^m m},$ with $\alpha_{ij} = \beta_{i \bmod q^j} \forall (i, j)$

INPUT:

- `code` – The associated code of this encoder.

-`polynomial_ring` – (default:`None`) The polynomial ring from which the message is chosen. If this is set to `None` , a polynomial ring in x will be built from the code parameters.

EXAMPLES:

```
sage: C1=codes.ReedMullerCode(GF(2), 2, 4)
sage: E1=codes.encoders.ReedMullerPolynomialEncoder(C1)
sage: E1
Evaluation polynomial-style encoder for Binary Reed-Muller Code of order 2 and␣
↪number of variables 4
sage: C2=codes.ReedMullerCode(GF(3), 2, 2)
sage: E2=codes.encoders.ReedMullerPolynomialEncoder(C2)
sage: E2
Evaluation polynomial-style encoder for Reed-Muller Code of order 2 and 2␣
↪variables over Finite Field of size 3
```

We can also pass a predefined polynomial ring:

```
sage: R=PolynomialRing(GF(3), 2, 'y')
sage: C=codes.ReedMullerCode(GF(3), 2, 2)
sage: E=codes.encoders.ReedMullerPolynomialEncoder(C, R)
sage: E
Evaluation polynomial-style encoder for Reed-Muller Code of order 2 and 2␣
↪variables over Finite Field of size 3
```

Actually, we can construct the encoder from `C` directly:

```
sage: E = C1.encoder("EvaluationPolynomial")
sage: E
Evaluation polynomial-style encoder for Binary Reed-Muller Code of order 2 and␣
↪number of variables 4
```

encode (*p*)

Transforms the polynomial `p` into a codeword of `code()`.

INPUT:

- `p` – A polynomial from the message space of `self` of degree less than `self.code().order()`.

OUTPUT:

- A codeword in associated code of `self`

EXAMPLES:

```
sage: F = GF(3)
sage: Fx.<x0,x1> = F[]
sage: C = codes.ReedMullerCode(F, 2, 2)
sage: E = C.encoder("EvaluationPolynomial")
sage: p = x0*x1 + x1^2 + x0 + x1 + 1
sage: c = E.encode(p); c
(1, 2, 0, 0, 2, 1, 1, 1, 1)
sage: c in C
True
```

If a polynomial with good monomial degree but wrong monomial degree is given,an error is raised:

```
sage: p = x0^2*x1
sage: E.encode(p)
Traceback (most recent call last):
...
ValueError: The polynomial to encode must have degree at most 2
```

If `p` is not an element of the proper polynomial ring, an error is raised:

```
sage: Qy.<y1,y2> = QQ[]
sage: p = y1^2 + 1
sage: E.encode(p)
Traceback (most recent call last):
...
ValueError: The value to encode must be in Multivariate Polynomial Ring in␣
↪x0, x1 over Finite Field of size 3
```

message_space ( )

Returns the message space of `self`

EXAMPLES:

```
sage: F = GF(11)
sage: C = codes.ReedMullerCode(F, 2, 4)
sage: E = C.encoder("EvaluationPolynomial")
sage: E.message_space()
Multivariate Polynomial Ring in x0, x1, x2, x3 over Finite Field of size 11
```

points ( )

Returns the evaluation points in the appropriate order as used by `self` when encoding a message.

EXAMPLES:

```
sage: F = GF(3)
sage: Fx.<x0,x1> = F[]
sage: C = codes.ReedMullerCode(F, 2, 2)
sage: E = C.encoder("EvaluationPolynomial")
sage: E.points()
[(0, 0), (1, 0), (2, 0), (0, 1), (1, 1), (2, 1), (0, 2), (1, 2), (2, 2)]
```

polynomial_ring ( )

Returns the polynomial ring associated with `self`

EXAMPLES:

```
sage: F = GF(11)
sage: C = codes.ReedMullerCode(F, 2, 4)
sage: E = C.encoder("EvaluationPolynomial")
sage: E.polynomial_ring()
Multivariate Polynomial Ring in x0, x1, x2, x3 over Finite Field of size 11
```

unencode_nocheck (*c*)

Returns the message corresponding to the codeword `c` .

Use this method with caution: it does not check if `c` belongs to the code, and if this is not the case, the output is unspecified. Instead, use `unencode()` .

INPUT:

- `c` – A codeword of `code()` .

OUTPUT:

- An polynomial of degree less than `self.code().order()` .

EXAMPLES:

```
sage: F = GF(3)
sage: C = codes.ReedMullerCode(F, 2, 2)
sage: E = C.encoder("EvaluationPolynomial")
sage: c = vector(F, (1, 2, 0, 0, 2, 1, 1, 1, 1))
sage: c in C
True
sage: p = E.unencode_nocheck(c); p
x0*x1 + x1^2 + x0 + x1 + 1
sage: E.encode(p) == c
True
```

Note that no error is thrown if `c` is not a codeword, and that the result is undefined:

```
sage: c = vector(F, (1, 2, 0, 0, 2, 1, 0, 1, 1))
sage: c in C
False
sage: p = E.unencode_nocheck(c); p
-x0*x1 - x1^2 + x0 + 1
sage: E.encode(p) == c
False
```

class sage.coding.reed_muller_code. **ReedMullerVectorEncoder** (*code*)

Bases: *sage.coding.encoder.Encoder*

Encoder for Reed-Muller codes which encodes vectors into codewords.

Consider a Reed-Muller code of order r, number of variables m, length n, dimension k over some finite field F. Let those variables be $(x_1, x_2, \ldots, x_m)$. We order the monomials by lowest power on lowest index variables. If we have three monomials $x_1 \times x_2$, $x_1 \times x_2^2$ and $x_1^2 \times x_2$, the ordering is: $x_1 \times x_2 < x_1 \times x_2^2 < x_1^2 \times x_2$

Let now $(v_1, v_2, \ldots, v_k)$ be a vector of F, which corresponds to the polynomial $f = \Sigma_{i=1}^{k} v_i \times x_i$.

Let $(\beta_1, \beta_2, \ldots, \beta_q)$ be the elements of F ordered as they are returned by Sage when calling `F.list()`.

The aforementioned polynomial f is encoded as:

$(f(\alpha_{11}, \alpha_{12}, \ldots, \alpha_{1m}), f(\alpha_{21}, \alpha_{22}, \ldots, \alpha_{2m}), \ldots, f(\alpha_{q^m 1}, \alpha_{q^m 2}, \ldots, \alpha_{q^m m}$, with $\alpha_{ij} = \beta_{i \bmod q^j} \forall (i, j)$

INPUT:

- `code` – The associated code of this encoder.

EXAMPLES:

```
sage: C1=codes.ReedMullerCode(GF(2), 2, 4)
sage: E1=codes.encoders.ReedMullerVectorEncoder(C1)
sage: E1
Evaluation vector-style encoder for Binary Reed-Muller Code of order 2 and number␣
↪of variables 4
sage: C2=codes.ReedMullerCode(GF(3), 2, 2)
sage: E2=codes.encoders.ReedMullerVectorEncoder(C2)
sage: E2
Evaluation vector-style encoder for Reed-Muller Code of order 2 and 2 variables␣
↪over Finite Field of size 3
```

Actually, we can construct the encoder from `C` directly:

```
sage: C=codes.ReedMullerCode(GF(2), 2, 4)
sage: E = C.encoder("EvaluationVector")
sage: E
Evaluation vector-style encoder for Binary Reed-Muller Code of order 2 and number␣
↪of variables 4
```

generator_matrix ( )

Returns a generator matrix of `self`

EXAMPLES:

```
sage: F = GF(3)
sage: C = codes.ReedMullerCode(F, 2, 2)
sage: E = codes.encoders.ReedMullerVectorEncoder(C)
sage: E.generator_matrix()
[1 1 1 1 1 1 1 1 1]
[0 1 2 0 1 2 0 1 2]
[0 0 0 1 1 1 2 2 2]
[0 1 1 0 1 1 0 1 1]
[0 0 0 0 1 2 0 2 1]
[0 0 0 1 1 1 1 1 1]
```

points ( )

Returns the points of F^m, where F is base field and m is the number of variables, in order of which polynomials are evaluated on.

EXAMPLES:

```
sage: F = GF(3)
sage: Fx.<x0,x1> = F[]
```

```
sage: C = codes.ReedMullerCode(F, 2, 2)
sage: E = C.encoder("EvaluationVector")
sage: E.points()
[(0, 0), (1, 0), (2, 0), (0, 1), (1, 1), (2, 1), (0, 2), (1, 2), (2, 2)]
```

CHAPTER

THREE

BOUNDS ON CODES

3.1 Bounds for Parameters of Codes

This module provided some upper and lower bounds for the parameters of codes.

AUTHORS:

- David Joyner (2006-07): initial implementation.
- William Stein (2006-07): minor editing of docs and code (fixed bug in elias_bound_asymp)
- David Joyner (2006-07): fixed dimension_upper_bound to return an integer, added example to elias_bound_asymp.
- ” (2009-05): removed all calls to Guava but left it as an option.
- Dima Pasechnik (2012-10): added LP bounds.

Let F be a finite field (we denote the finite field with q elements by $\mathbf{F}_q$). A subset C of $V = F^n$ is called a code of length n. A subspace of V (with the standard basis) is called a linear code of length n. If its dimension is denoted k then we typically store a basis of C as a $k \times n$ matrix (the rows are the basis vectors). If $F = \mathbf{F}_2$ then C is called a binary code. If F has q elements then C is called a q-ary code. The elements of a code C are called codewords. The information rate of C is

$$R = \frac{\log_q |C|}{n},$$

where $|C|$ denotes the number of elements of C. If $\mathbf{v} = (v_1, v_2, ..., v_n)$, $\mathbf{w} = (w_1, w_2, ..., w_n)$ are vectors in $V = F^n$ then we define

$$d(\mathbf{v}, \mathbf{w}) = |\{i \mid 1 \leq i \leq n,\ v_i \neq w_i\}|$$

to be the Hamming distance between $\mathbf{v}$ and $\mathbf{w}$. The function $d : V \times V \to \mathbf{N}$ is called the Hamming metric. The weight of a vector (in the Hamming metric) is $d(\mathbf{v}, \mathbf{0})$. The minimum distance of a linear code is the smallest non-zero weight of a codeword in C. The relatively minimum distance is denoted

$$\delta = d/n.$$

A linear code with length n, dimension k, and minimum distance d is called an $[n, k, d]_q$-code and n, k, d are called its parameters. A (not necessarily linear) code C with length n, size $M = |C|$, and minimum distance d is called an $(n, M, d)_q$-code (using parentheses instead of square brackets). Of course, $k = \log_q(M)$ for linear codes.

What is the “best” code of a given length? Let F be a finite field with q elements. Let $A_q(n, d)$ denote the largest M such that there exists a (n, M, d) code in F^n. Let $B_q(n, d)$ (also denoted $A_q^{lin}(n, d)$) denote the largest k such that there exists a $[n, k, d]$ code in F^n. (Of course, $A_q(n, d) \geq B_q(n, d)$.) Determining $A_q(n, d)$ and $B_q(n, d)$ is one of the main problems in the theory of error-correcting codes. For more details see *[HP]* and *[vL]*.

These quantities related to solving a generalization of the childhood game of "20 questions".

GAME: Player 1 secretly chooses a number from 1 to M (M is large but fixed). Player 2 asks a series of "yes/no questions" in an attempt to determine that number. Player 1 may lie at most e times ($e \geq 0$ is fixed). What is the minimum number of "yes/no questions" Player 2 must ask to (always) be able to correctly determine the number Player 1 chose?

If feedback is not allowed (the only situation considered here), call this minimum number $g(M, e)$.

Lemma: For fixed e and M, $g(M, e)$ is the smallest n such that $A_2(n, 2e + 1) \geq M$.

Thus, solving the solving a generalization of the game of "20 questions" is equivalent to determining $A_2(n, d)$! Using Sage, you can determine the best known estimates for this number in 2 ways:

1. **Indirectly, using best_known_linear_code_www(n, k, F),** which connects to the website http://www.codetables.de by Markus Grassl;
2. **codesize_upper_bound(n,d,q), dimension_upper_bound(n,d,q),** and best_known_linear_code(n, k, F).

The output of `best_known_linear_code()`, `best_known_linear_code_www()`, or *dimension_upper_bound()* would give only special solutions to the GAME because the bounds are applicable to only linear codes. The output of *codesize_upper_bound()* would give the best possible solution, that may belong to a linear or nonlinear code.

This module implements:

- codesize_upper_bound(n,d,q), for the best known (as of May, 2006) upper bound A(n,d) for the size of a code of length n, minimum distance d over a field of size q.
- dimension_upper_bound(n,d,q), an upper bound $B(n, d) = B_q(n, d)$ for the dimension of a linear code of length n, minimum distance d over a field of size q.
- gilbert_lower_bound(n,q,d), a lower bound for number of elements in the largest code of min distance d in $\mathbf{F}_q^n$.
- gv_info_rate(n,delta,q), $log_q(GLB)/n$, where GLB is the Gilbert lower bound and delta = d/n.
- gv_bound_asymp(delta,q), asymptotic analog of Gilbert lower bound.
- plotkin_upper_bound(n,q,d)
- plotkin_bound_asymp(delta,q), asymptotic analog of Plotkin bound.
- griesmer_upper_bound(n,q,d)
- elias_upper_bound(n,q,d)
- elias_bound_asymp(delta,q), asymptotic analog of Elias bound.
- hamming_upper_bound(n,q,d)
- hamming_bound_asymp(delta,q), asymptotic analog of Hamming bound.
- singleton_upper_bound(n,q,d)
- singleton_bound_asymp(delta,q), asymptotic analog of Singleton bound.
- mrrw1_bound_asymp(delta,q), "first" asymptotic McEliese-Rumsey-Rodemich-Welsh bound for the information rate.
- Delsarte (a.k.a. Linear Programming (LP)) upper bounds.

PROBLEM: In this module we shall typically either (a) seek bounds on k, given n, d, q, (b) seek bounds on R, delta, q (assuming n is "infinity").

TODO:

- Johnson bounds for binary codes.

- mrrw2_bound_asymp(delta,q), "second" asymptotic McEliese-Rumsey-Rodemich-Welsh bound for the information rate.

sage.coding.code_bounds. **codesize_upper_bound** (*n*, *d*, *q*, *algorithm=None*)

This computes the minimum value of the upper bound using the methods of Singleton, Hamming, Plotkin, and Elias.

If algorithm="gap" then this returns the best known upper bound $A(n, d) = A_q(n, d)$ for the size of a code of length n, minimum distance d over a field of size q. The function first checks for trivial cases (like d=1 or n=d), and if the value is in the built-in table. Then it calculates the minimum value of the upper bound using the algorithms of Singleton, Hamming, Johnson, Plotkin and Elias. If the code is binary, $A(n, 2\ell-1) = A(n+1, 2\ell)$, so the function takes the minimum of the values obtained from all algorithms for the parameters $(n, 2\ell - 1)$ and $(n + 1, 2\ell)$. This wraps GUAVA's (i.e. GAP's package Guava) UpperBound(n, d, q).

If algorithm="LP" then this returns the Delsarte (a.k.a. Linear Programming) upper bound.

EXAMPLES:

```
sage: codes.bounds.codesize_upper_bound(10,3,2)
93
sage: codes.bounds.codesize_upper_bound(24,8,2,algorithm="LP")
4096
sage: codes.bounds.codesize_upper_bound(10,3,2,algorithm="gap")  # optional - gap_
↪packages (Guava package)
85
sage: codes.bounds.codesize_upper_bound(11,3,4,algorithm=None)
123361
sage: codes.bounds.codesize_upper_bound(11,3,4,algorithm="gap")  # optional - gap_
↪packages (Guava package)
123361
sage: codes.bounds.codesize_upper_bound(11,3,4,algorithm="LP")
109226
```

sage.coding.code_bounds. **dimension_upper_bound** (*n*, *d*, *q*, *algorithm=None*)

Returns an upper bound $B(n, d) = B_q(n, d)$ for the dimension of a linear code of length n, minimum distance d over a field of size q. Parameter "algorithm" has the same meaning as in *codesize_upper_bound()*

EXAMPLES:

```
sage: codes.bounds.dimension_upper_bound(10,3,2)
6
sage: codes.bounds.dimension_upper_bound(30,15,4)
13
sage: codes.bounds.dimension_upper_bound(30,15,4,algorithm="LP")
12
```

sage.coding.code_bounds. **elias_bound_asymp** (*delta*, *q*)

Computes the asymptotic Elias bound for the information rate, provided $0 < \delta < 1 - 1/q$.

EXAMPLES:

```
sage: codes.bounds.elias_bound_asymp(1/4,2)
0.39912396330...
```

sage.coding.code_bounds. **elias_upper_bound** (*n*, *q*, *d*, *algorithm=None*)

Returns the Elias upper bound for number of elements in the largest code of minimum distance d in $\mathbf{F}_q^n$. Wraps GAP's UpperBoundElias.

EXAMPLES:

```
sage: codes.bounds.elias_upper_bound(10,2,3)
232
sage: codes.bounds.elias_upper_bound(10,2,3,algorithm="gap")  # optional - gap_
↪packages (Guava package)
232
```

sage.coding.code_bounds. **entropy** (*x*, *q=2*)

Computes the entropy at x on the q-ary symmetric channel.

INPUT:

- `x` - real number in the interval $[0, 1]$.
- `q` - (default: 2) integer greater than 1. This is the base of the logarithm.

EXAMPLES:

```
sage: codes.bounds.entropy(0, 2)
0
sage: codes.bounds.entropy(1/5,4)
1/5*log(3)/log(4) - 4/5*log(4/5)/log(4) - 1/5*log(1/5)/log(4)
sage: codes.bounds.entropy(1, 3)
log(2)/log(3)
```

Check that values not within the limits are properly handled:

```
sage: codes.bounds.entropy(1.1, 2)
Traceback (most recent call last):
...
ValueError: The entropy function is defined only for x in the interval [0, 1]
sage: codes.bounds.entropy(1, 1)
Traceback (most recent call last):
...
ValueError: The value q must be an integer greater than 1
```

sage.coding.code_bounds. **entropy_inverse** (*x*, *q=2*)

Find the inverse of the `q` -ary entropy function at the point `x` .

INPUT:

- `x` – real number in the interval $[0, 1]$.
- `q` - (default: 2) integer greater than 1. This is the base of the logarithm.

OUTPUT:

Real number in the interval $[0, 1 - 1/q]$. The function has multiple values if we include the entire interval $[0, 1]$; hence only the values in the above interval is returned.

EXAMPLES:

```
sage: from sage.coding.code_bounds import entropy_inverse
sage: entropy_inverse(0.1)
0.012986862055848683
sage: entropy_inverse(1)
1/2
sage: entropy_inverse(0, 3)
0
sage: entropy_inverse(1, 3)
2/3
```

sage.coding.code_bounds. **gilbert_lower_bound** (*n*, *q*, *d*)
Returns lower bound for number of elements in the largest code of minimum distance d in $\mathbf{F}_q^n$.

EXAMPLES:

```
sage: codes.bounds.gilbert_lower_bound(10,2,3)
128/7
```

sage.coding.code_bounds. **griesmer_upper_bound** (*n*, *q*, *d*, *algorithm=None*)
Returns the Griesmer upper bound for number of elements in the largest code of minimum distance d in $\mathbf{F}_q^n$. Wraps GAP's UpperBoundGriesmer.

EXAMPLES:

```
sage: codes.bounds.griesmer_upper_bound(10,2,3)
128
sage: codes.bounds.griesmer_upper_bound(10,2,3,algorithm="gap")  # optional - gap_
↪packages (Guava package)
128
```

sage.coding.code_bounds. **gv_bound_asymp** (*delta*, *q*)
Computes the asymptotic GV bound for the information rate, R.

EXAMPLES:

```
sage: RDF(codes.bounds.gv_bound_asymp(1/4,2))
0.18872187554086...
sage: f = lambda x: codes.bounds.gv_bound_asymp(x,2)
sage: plot(f,0,1)
Graphics object consisting of 1 graphics primitive
```

sage.coding.code_bounds. **gv_info_rate** (*n*, *delta*, *q*)
GV lower bound for information rate of a q-ary code of length n minimum distance delta*n

EXAMPLES:

```
sage: RDF(codes.bounds.gv_info_rate(100,1/4,3))  # abs tol 1e-15
0.36704992608261894
```

sage.coding.code_bounds. **hamming_bound_asymp** (*delta*, *q*)
Computes the asymptotic Hamming bound for the information rate.

EXAMPLES:

```
sage: RDF(codes.bounds.hamming_bound_asymp(1/4,2))
0.456435556800...
sage: f = lambda x: codes.bounds.hamming_bound_asymp(x,2)
sage: plot(f,0,1)
Graphics object consisting of 1 graphics primitive
```

sage.coding.code_bounds. **hamming_upper_bound** (*n*, *q*, *d*)
Returns the Hamming upper bound for number of elements in the largest code of minimum distance d in $\mathbf{F}_q^n$. Wraps GAP's UpperBoundHamming.

The Hamming bound (also known as the sphere packing bound) returns an upper bound on the size of a code of length n, minimum distance d, over a field of size q. The Hamming bound is obtained by dividing the contents of the entire space $\mathbf{F}_q^n$ by the contents of a ball with radius floor((d-1)/2). As all these balls are disjoint, they can never contain more than the whole vector space.

$$M \leq \frac{q^n}{V(n,e)},$$

where M is the maximum number of codewords and $V(n, e)$ is equal to the contents of a ball of radius e. This bound is useful for small values of d. Codes for which equality holds are called perfect.

EXAMPLES:

```
sage: codes.bounds.hamming_upper_bound(10,2,3)
93
```

sage.coding.code_bounds. **mrrw1_bound_asymp** (*delta*, *q*)

Computes the first asymptotic McEliese-Rumsey-Rodemich-Welsh bound for the information rate, provided $0 < \delta < 1 - 1/q$.

EXAMPLES:

```
sage: codes.bounds.mrrw1_bound_asymp(1/4,2)    # abs tol 4e-16
0.3545789026652697
```

sage.coding.code_bounds. **plotkin_bound_asymp** (*delta*, *q*)

Computes the asymptotic Plotkin bound for the information rate, provided $0 < \delta < 1 - 1/q$.

EXAMPLES:

```
sage: codes.bounds.plotkin_bound_asymp(1/4,2)
1/2
```

sage.coding.code_bounds. **plotkin_upper_bound** (*n*, *q*, *d*, *algorithm=None*)

Returns Plotkin upper bound for number of elements in the largest code of minimum distance d in $\mathbf{F}_q^n$.

The algorithm="gap" option wraps Guava's UpperBoundPlotkin.

EXAMPLES:

```
sage: codes.bounds.plotkin_upper_bound(10,2,3)
192
sage: codes.bounds.plotkin_upper_bound(10,2,3,algorithm="gap")  # optional - gap_
↪packages (Guava package)
192
```

sage.coding.code_bounds. **singleton_bound_asymp** (*delta*, *q*)

Computes the asymptotic Singleton bound for the information rate.

EXAMPLES:

```
sage: codes.bounds.singleton_bound_asymp(1/4,2)
3/4
sage: f = lambda x: codes.bounds.singleton_bound_asymp(x,2)
sage: plot(f,0,1)
Graphics object consisting of 1 graphics primitive
```

sage.coding.code_bounds. **singleton_upper_bound** (*n*, *q*, *d*)

Returns the Singleton upper bound for number of elements in the largest code of minimum distance d in $\mathbf{F}_q^n$. Wraps GAP's UpperBoundSingleton.

This bound is based on the shortening of codes. By shortening an (n, M, d) code d-1 times, an $(n-d+1, M, 1)$ code results, with $M \leq q^n - d + 1$. Thus

$$M \leq q^{n-d+1}.$$

Codes that meet this bound are called maximum distance separable (MDS).

EXAMPLES:

```
sage: codes.bounds.singleton_upper_bound(10,2,3)
256
```

sage.coding.code_bounds. **volume_hamming** (*n*, *q*, *r*)

Returns number of elements in a Hamming ball of radius r in $\mathbf{F}_q^n$. Agrees with Guava's SphereContent(n,r,GF(q)).

EXAMPLES:

```
sage: codes.bounds.volume_hamming(10,2,3)
176
```

3.2 Delsarte, a.k.a. Linear Programming (LP), upper bounds

This module provides LP upper bounds for the parameters of codes, introduced in *[De73]*.

The exact LP solver PPL is used by default, ensuring that no rounding/overflow problems occur.

AUTHORS:

- Dmitrii V. (Dima) Pasechnik (2012-10): initial implementation. Minor fixes (2015)

sage.coding.delsarte_bounds. **Kravchuk** (*n*, *q*, *l*, *x*, *check=True*)

Compute `K^{n,q}_l(x)` , the Krawtchouk (a.k.a. Kravchuk) polynomial.

See Wikipedia article Kravchuk_polynomials.

It is defined by the generating function

$$(1+(q-1)z)^{n-x}(1-z)^x = \sum_l K_l^{n,q}(x) z^l$$

and is equal to

$$K_l^{n,q}(x) = \sum_{j=0}^{l} (-1)^j (q-1)^{(l-j)} \binom{x}{j} \binom{n-x}{l-j},$$

INPUT:

- `n, q, x` – arbitrary numbers
- `l` – a nonnegative integer
- `check` – check the input for correctness. `True` by default. Otherwise, pass it as it is. Use `check=False` at your own risk.

EXAMPLES:

```
sage: Krawtchouk(24,2,5,4)
2224
sage: Krawtchouk(12300,4,5,6)
567785569973042442072
```

TESTS:

check that the bug reported on trac ticket #19561 is fixed:

```
sage: Krawtchouk(3,2,3,3)
-1
sage: Krawtchouk(int(3),int(2),int(3),int(3))
-1
sage: Krawtchouk(int(3),int(2),int(3),int(3),check=False)
-1.0
sage: Kravchuk(24,2,5,4)
2224
```

other unusual inputs

```
sage: Krawtchouk(sqrt(5),1-I*sqrt(3),3,55.3).n()
211295.892797... + 1186.42763...*I
sage: Krawtchouk(-5/2,7*I,3,-1/10)
480053/250*I - 357231/400
sage: Krawtchouk(1,1,-1,1)
Traceback (most recent call last):
...
ValueError: l must be a nonnegative integer
sage: Krawtchouk(1,1,3/2,1)
Traceback (most recent call last):
...
TypeError: no conversion of this rational to integer
```

sage.coding.delsarte_bounds. **Krawtchouk** (*n*, *q*, *l*, *x*, *check=True*)

Compute `K^{n,q}_l(x)` , the Krawtchouk (a.k.a. Kravchuk) polynomial.

See Wikipedia article Kravchuk_polynomials.

It is defined by the generating function

$$(1+(q-1)z)^{n-x}(1-z)^x = \sum_l K_l^{n,q}(x) z^l$$

and is equal to

$$K_l^{n,q}(x) = \sum_{j=0}^{l} (-1)^j (q-1)^{(l-j)} \binom{x}{j} \binom{n-x}{l-j},$$

INPUT:

- `n,q,x` – arbitrary numbers
- `l` – a nonnegative integer
- `check` – check the input for correctness. `True` by default. Otherwise, pass it as it is. Use `check=False` at your own risk.

EXAMPLES:

```
sage: Krawtchouk(24,2,5,4)
2224
sage: Krawtchouk(12300,4,5,6)
567785569973042442072
```

TESTS:

check that the bug reported on trac ticket #19561 is fixed:

```
sage: Krawtchouk(3,2,3,3)
-1
sage: Krawtchouk(int(3),int(2),int(3),int(3))
-1
sage: Krawtchouk(int(3),int(2),int(3),int(3),check=False)
-1.0
sage: Kravchuk(24,2,5,4)
2224
```

other unusual inputs

```
sage: Krawtchouk(sqrt(5),1-I*sqrt(3),3,55.3).n()
211295.892797... + 1186.42763...*I
sage: Krawtchouk(-5/2,7*I,3,-1/10)
480053/250*I - 357231/400
sage: Krawtchouk(1,1,-1,1)
Traceback (most recent call last):
...
ValueError: l must be a nonnegative integer
sage: Krawtchouk(1,1,3/2,1)
Traceback (most recent call last):
...
TypeError: no conversion of this rational to integer
```

sage.coding.delsarte_bounds. **delsarte_bound_additive_hamming_space** (*n*, *d*, *q*, *d_star=1*, *q_base=0*, *return_data=False*, *solver='PPL'*, *isinteger=False*)

Find a modified Delsarte bound on additive codes in Hamming space `H_q^n` of minimal distance `d`

Find the Delsarte LP bound on `F_{q_base}` -dimension of additive codes in Hamming space `H_q^n` of minimal distance `d` with minimal distance of the dual code at least `d_star` . If `q_base` is set to non-zero, then `q` is a power of `q_base` , and the code is, formally, linear over `F_{q_base}` . Otherwise it is assumed that `q_base==q` .

INPUT:

- `n` – the code length
- `d` – the (lower bound on) minimal distance of the code
- `q` – the size of the alphabet
- `d_star` – the (lower bound on) minimal distance of the dual code; only makes sense for additive codes.
- `q_base` – if `0` , the code is assumed to be linear. Otherwise, `q=q_base^m` and the code is linear over `F_{q_base}` .
- `return_data` – if `True` , return a triple `(W,LP,bound)` , where `W` is a weights vector, and `LP` the Delsarte bound LP; both of them are Sage LP data. `W` need not be a weight distribution of a code, or, if `isinteger==False` , even have integer entries.
- `solver` – the LP/ILP solver to be used. Defaults to `PPL` . It is arbitrary precision, thus there will be no rounding errors. With other solvers (see `MixedIntegerLinearProgram` for the list), you are on your own!

- `isinteger` – if `True`, uses an integer programming solver (ILP), rather that an LP solver. Can be very slow if set to `True`.

EXAMPLES:

The bound on dimension of linear F_2-codes of length 11 and minimal distance 6:

```
sage: delsarte_bound_additive_hamming_space(11, 6, 2)
3
sage: a,p,val=delsarte_bound_additive_hamming_space(11, 6, 2,
 ↪                  return_data=True)
sage: [j for i,j in p.get_values(a).iteritems()]
[1, 0, 0, 0, 0, 0, 5, 2, 0, 0, 0, 0]
```

The bound on the dimension of linear F_4-codes of length 11 and minimal distance 3:

```
sage: delsarte_bound_additive_hamming_space(11,3,4)
8
```

The bound on the F_2-dimension of additive F_4-codes of length 11 and minimal distance 3:

```
sage: delsarte_bound_additive_hamming_space(11,3,4,q_base=2)
16
```

Such a `d_star` is not possible:

```
sage: delsarte_bound_additive_hamming_space(11,3,4,d_star=9)
Solver exception: PPL : There is no feasible solution
False
```

TESTS:

```
sage: a,p,x=delsarte_bound_additive_hamming_space(19,15,7,return_
 ↪data=True,isinteger=True)
sage: [j for i,j in p.get_values(a).iteritems()]
[1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 307, 0, 0, 1, 34]
sage: delsarte_bound_additive_hamming_space(19,15,7,solver='glpk')
3
sage: delsarte_bound_additive_hamming_space(19,15,7,isinteger=True,solver='glpk')
3
```

sage.coding.delsarte_bounds. **delsarte_bound_hamming_space**(*n*, *d*, *q*, *return_data=False*, *solver='PPL'*, *isinteger=False*)

Find the Delsarte bound *[De73]* on codes in Hamming space `H_q^n` of minimal distance `d`

INPUT:

- `n` – the code length
- `d` – the (lower bound on) minimal distance of the code
- `q` – the size of the alphabet
- `return_data` – if `True`, return a triple `(W,LP,bound)`, where `W` is a weights vector, and `LP` the Delsarte upper bound LP; both of them are Sage LP data. `W` need not be a weight distribution of a code.
- `solver` – the LP/ILP solver to be used. Defaults to `PPL`. It is arbitrary precision, thus there will be no rounding errors. With other solvers (see `MixedIntegerLinearProgram` for the list), you are on your own!

- `isinteger` – if `True`, uses an integer programming solver (ILP), rather that an LP solver. Can be very slow if set to `True`.

EXAMPLES:

The bound on the size of the F_2-codes of length 11 and minimal distance 6:

```
sage: delsarte_bound_hamming_space(11, 6, 2)
12
sage: a, p, val = delsarte_bound_hamming_space(11, 6, 2, return_data=True)
sage: [j for i,j in p.get_values(a).iteritems()]
[1, 0, 0, 0, 0, 0, 11, 0, 0, 0, 0, 0]
```

The bound on the size of the F_2-codes of length 24 and minimal distance 8, i.e. parameters of the extened binary Golay code:

```
sage: a,p,x=delsarte_bound_hamming_space(24,8,2,return_data=True)
sage: x
4096
sage: [j for i,j in p.get_values(a).iteritems()]
[1, 0, 0, 0, 0, 0, 0, 0, 759, 0, 0, 0, 2576, 0, 0, 0, 759, 0, 0, 0, 0, 0, 0, 0, 1]
```

The bound on the size of F_4-codes of length 11 and minimal distance 3:

```
sage: delsarte_bound_hamming_space(11,3,4)
327680/3
```

An improvement of a known upper bound (150) from http://www.win.tue.nl/~aeb/codes/binary-1.html

```
sage: a,p,x= delsarte_bound_hamming_space(23,10,2,return_
 ↪data=True,isinteger=True); x # long time
148
sage: [j for i,j in p.get_values(a).iteritems()]
 ↪    # long time
[1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 95, 0, 2, 0, 36, 0, 14, 0, 0, 0, 0, 0, 0, 0]
```

Note that a usual LP, without integer variables, won't do the trick

```
sage: delsarte_bound_hamming_space(23,10,2).n(20)
151.86
```

Such an input is invalid:

```
sage: delsarte_bound_hamming_space(11,3,-4)
Solver exception: PPL : There is no feasible solution
False
```

REFERENCES:

CHAPTER

FOUR

CHANNELS AND RELATED CONSTRUCTIONS

4.1 Channels

Given an input space and an output space, a channel takes element from the input space (the message) and transforms it into an element of the output space (the transmitted message).

In Sage, Channels simulate error-prone transmission over communication channels, and we borrow the nomenclature from communication theory, such as "transmission" and "positions" as the elements of transmitted vectors. Transmission can be achieved with two methods:

- *Channel.transmit()* . Considering a channel `Chan` and a message `msg` , transmitting `msg` with `Chan` can be done this way:

```
Chan.transmit(msg)
```

  It can also be written in a more convenient way:

```
Chan(msg)
```

- `transmit_unsafe()` . This does the exact same thing as `transmit()` except that it does not check if `msg` belongs to the input space of `Chan` :

```
Chan.transmit_unsafe(msg)
```

This is useful in e.g. an inner-loop of a long simulation as a lighter-weight alternative to *Channel.transmit()* .

This file contains the following elements:

- *Channel* , the abstract class for Channels
- *StaticErrorRateChannel* , which creates a specific number of errors in each transmitted message
- *ErrorErasureChannel* , which creates a specific number of errors and a specific number of erasures in each transmitted message

class `sage.coding.channel_constructions.` **`Channel`** (*input_space*, *output_space*)
Bases: `sage.structure.sage_object.SageObject`

Abstract top-class for Channel objects.

All channel objects must inherit from this class. To implement a channel subclass, one should do the following:

•inherit from this class,

•call the super constructor,

•override *transmit_unsafe()* .

While not being mandatory, it might be useful to reimplement representation methods (_repr_ and _latex_).

This abstract class provides the following parameters:

- input_space – the space of the words to transmit
- output_space – the space of the transmitted words

input_space ( )

Returns the input space of self .

EXAMPLES:

```
sage: n_err = 2
sage: Chan = channels.StaticErrorRateChannel(GF(59)^6, n_err)
sage: Chan.input_space()
Vector space of dimension 6 over Finite Field of size 59
```

output_space ( )

Returns the output space of self .

EXAMPLES:

```
sage: n_err = 2
sage: Chan = channels.StaticErrorRateChannel(GF(59)^6, n_err)
sage: Chan.output_space()
Vector space of dimension 6 over Finite Field of size 59
```

transmit (*message*)

Returns message , modified accordingly with the algorithm of the channel it was transmitted through.

Checks if message belongs to the input space, and returns an exception if not. Note that message itself is never modified by the channel.

INPUT:

- message – a vector

OUTPUT:

- a vector of the output space of self

EXAMPLES:

```
sage: F = GF(59)^6
sage: n_err = 2
sage: Chan = channels.StaticErrorRateChannel(F, n_err)
sage: msg = F((4, 8, 15, 16, 23, 42))
sage: set_random_seed(10)
sage: Chan.transmit(msg)
(4, 8, 4, 16, 23, 53)
```

We can check that the input msg is not modified:

```
sage: msg
(4, 8, 15, 16, 23, 42)
```

If we transmit a vector which is not in the input space of self :

```
sage: n_err = 2
sage: Chan = channels.StaticErrorRateChannel(GF(59)^6, n_err)
```

```
sage: msg = (4, 8, 15, 16, 23, 42)
sage: Chan.transmit(msg)
Traceback (most recent call last):
...
TypeError: Message must be an element of the input space for the given channel
```

Note: One can also call directly `Chan(message)`, which does the same as `Chan.transmit(message)`

transmit_unsafe (*message*)

Returns `message`, modified accordingly with the algorithm of the channel it was transmitted through.

This method does not check if `message` belongs to the input space of''self''.

This is an abstract method which should be reimplemented in all the subclasses of Channel.

class sage.coding.channel_constructions. **ErrorErasureChannel** (*space*, *number_errors*, *number_erasures*)

Bases: *sage.coding.channel_constructions.Channel*

Channel which adds errors and erases several positions in any message it transmits.

The output space of this channel is a Cartesian product between its input space and a VectorSpace of the same dimension over GF(2)

INPUT:

- `space` – the input and output space
- `number_errors` – the number of errors created in each transmitted message. It can be either an integer of a tuple. If an tuple is passed as an argument, the number of errors will be a random integer between the two bounds of this tuple.
- `number_erasures` – the number of erasures created in each transmitted message. It can be either an integer of a tuple. If an tuple is passed as an argument, the number of erasures will be a random integer between the two bounds of this tuple.

EXAMPLES:

We construct a ErrorErasureChannel which adds 2 errors and 2 erasures to any transmitted message:

```
sage: n_err, n_era = 2, 2
sage: Chan = channels.ErrorErasureChannel(GF(59)^40, n_err, n_era)
sage: Chan
Error-and-erasure channel creating 2 errors and 2 erasures
of input space Vector space of dimension 40 over Finite Field of size 59
and output space The Cartesian product of (Vector space of dimension 40
over Finite Field of size 59, Vector space of dimension 40 over Finite Field of␣
↪size 2)
```

We can also pass the number of errors and erasures as a couple of integers:

```
sage: n_err, n_era = (1, 10), (1, 10)
sage: Chan = channels.ErrorErasureChannel(GF(59)^40, n_err, n_era)
sage: Chan
Error-and-erasure channel creating between 1 and 10 errors and
between 1 and 10 erasures of input space Vector space of dimension 40
over Finite Field of size 59 and output space The Cartesian product of
(Vector space of dimension 40 over Finite Field of size 59,
Vector space of dimension 40 over Finite Field of size 2)
```

number_erasures ( )

Returns the number of erasures created by `self` .

EXAMPLES:

```
sage: n_err, n_era = 0, 3
sage: Chan = channels.ErrorErasureChannel(GF(59)^6, n_err, n_era)
sage: Chan.number_erasures()
(3, 3)
```

number_errors ( )

Returns the number of errors created by `self` .

EXAMPLES:

```
sage: n_err, n_era = 3, 0
sage: Chan = channels.ErrorErasureChannel(GF(59)^6, n_err, n_era)
sage: Chan.number_errors()
(3, 3)
```

transmit_unsafe (*message*)

Returns `message` with as many errors as `self._number_errors` in it, and as many erasures as `self._number_erasures` in it.

If `self._number_errors` was passed as an tuple for the number of errors, it will pick a random integer between the bounds of the tuple and use it as the number of errors. It does the same with `self._number_erasures` .

All erased positions are set to 0 in the transmitted message. It is guaranteed that the erasures and the errors will never overlap: the received message will always contains exactly as many errors and erasures as expected.

This method does not check if `message` belongs to the input space of‘‘self‘‘.

INPUT:

- `message` – a vector

OUTPUT:

- a couple of vectors, namely:
  - the transmitted message, which is `message` with erroneous and erased positions
  - the erasure vector, which contains `1` at the erased positions of the transmitted message , 0 elsewhere.

EXAMPLES:

```
sage: F = GF(59)^11
sage: n_err, n_era = 2, 2
sage: Chan = channels.ErrorErasureChannel(F, n_err, n_era)
sage: msg = F((3, 14, 15, 9, 26, 53, 58, 9, 7, 9, 3))
sage: set_random_seed(10)
sage: Chan.transmit_unsafe(msg)
((31, 0, 15, 9, 38, 53, 58, 9, 0, 9, 3), (0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0))
```

class sage.coding.channel_constructions. **QarySymmetricChannel** (*space*, *epsilon*)

Bases: *sage.coding.channel_constructions.Channel*

The q-ary symmetric, memoryless communication channel.

Given an alphabet Σ with $|\Sigma| = q$ and an error probability ϵ, a q-ary symmetric channel sends an element of Σ into the same element with probability $1 - \epsilon$, and any one of the other $q - 1$ elements with probability $\frac{\epsilon}{q-1}$. This implementation operates over vectors in Σ^n, and "transmits" each element of the vector independently in the above manner.

Though Σ is usually taken to be a finite field, this implementation allows any structure for which Sage can represent Σ^n and for which Σ has a $random_element()$ method. However, beware that if Σ is infinite, errors will not be uniformly distributed (since $random_element()$ does not draw uniformly at random).

The input space and the output space of this channel are the same: Σ^n.

INPUT:

- `space` – the input and output space of the channel. It has to be $GF(q)^n$ for some finite field $GF(q)$.
- `epsilon` – the transmission error probability of the individual elements.

EXAMPLES:

We construct a QarySymmetricChannel which corrupts 30% of all transmitted symbols:

```
sage: epsilon = 0.3
sage: Chan = channels.QarySymmetricChannel(GF(59)^50, epsilon)
sage: Chan
q-ary symmetric channel with error probability 0.300000000000000,
of input and output space Vector space of dimension 50 over Finite Field of size␣
↪59
```

error_probability ( )

Returns the error probability of a single symbol transmission of `self` .

EXAMPLES:

```
sage: epsilon = 0.3
sage: Chan = channels.QarySymmetricChannel(GF(59)^50, epsilon)
sage: Chan.error_probability()
0.300000000000000
```

probability_of_at_most_t_errors (*t*)

Returns the probability `self` has to return at most `t` errors.

INPUT:

- `t` – an integer

EXAMPLES:

```
sage: epsilon = 0.3
sage: Chan = channels.QarySymmetricChannel(GF(59)^50, epsilon)
sage: Chan.probability_of_at_most_t_errors(20)
0.952236164579467
```

probability_of_exactly_t_errors (*t*)

Returns the probability `self` has to return exactly `t` errors.

INPUT:

- `t` – an integer

EXAMPLES:

```
sage: epsilon = 0.3
sage: Chan = channels.QarySymmetricChannel(GF(59)^50, epsilon)
sage: Chan.probability_of_exactly_t_errors(15)
0.122346861835401
```

transmit_unsafe (*message*)

Returns `message` where each of the symbols has been changed to another from the alphabet with probability *error_probability()* .

This method does not check if `message` belongs to the input space of"self".

INPUT:

- `message` – a vector

EXAMPLES:

```
sage: F = GF(59)^11
sage: epsilon = 0.3
sage: Chan = channels.QarySymmetricChannel(F, epsilon)
sage: msg = F((3, 14, 15, 9, 26, 53, 58, 9, 7, 9, 3))
sage: set_random_seed(10)
sage: Chan.transmit_unsafe(msg)
(3, 14, 15, 53, 12, 53, 58, 9, 55, 9, 3)
```

class sage.coding.channel_constructions. **StaticErrorRateChannel** (*space*, *number_errors*)

Bases: *sage.coding.channel_constructions.Channel*

Channel which adds a static number of errors to each message it transmits.

The input space and the output space of this channel are the same.

INPUT:

- `space` – the space of both input and output
- `number_errors` – the number of errors added to each transmitted message It can be either an integer of a tuple. If a tuple is passed as argument, the number of errors will be a random integer between the two bounds of the tuple.

EXAMPLES:

We construct a StaticErrorRateChannel which adds 2 errors to any transmitted message:

```
sage: n_err = 2
sage: Chan = channels.StaticErrorRateChannel(GF(59)^40, n_err)
sage: Chan
Static error rate channel creating 2 errors, of input and output space
Vector space of dimension 40 over Finite Field of size 59
```

We can also pass a tuple for the number of errors:

```
sage: n_err = (1, 10)
sage: Chan = channels.StaticErrorRateChannel(GF(59)^40, n_err)
sage: Chan
Static error rate channel creating between 1 and 10 errors,
of input and output space Vector space of dimension 40 over Finite Field of size␣
↪59
```

number_errors ( )

Returns the number of errors created by `self` .

EXAMPLES:

```
sage: n_err = 3
sage: Chan = channels.StaticErrorRateChannel(GF(59)^6, n_err)
sage: Chan.number_errors()
(3, 3)
```

transmit_unsafe (*message*)

Returns `message` with as many errors as `self._number_errors` in it.

If `self._number_errors` was passed as a tuple for the number of errors, it will pick a random integer between the bounds of the tuple and use it as the number of errors.

This method does not check if `message` belongs to the input space of‘‘self‘‘.

INPUT:

- `message` – a vector

OUTPUT:

- a vector of the output space

EXAMPLES:

```
sage: F = GF(59)^6
sage: n_err = 2
sage: Chan = channels.StaticErrorRateChannel(F, n_err)
sage: msg = F((4, 8, 15, 16, 23, 42))
sage: set_random_seed(10)
sage: Chan.transmit_unsafe(msg)
(4, 8, 4, 16, 23, 53)
```

This checks that trac ticket #19863 is fixed:

```
sage: V = VectorSpace(GF(2), 1000)
sage: Chan = channels.StaticErrorRateChannel(V, 367)
sage: c = V.random_element()
sage: (c - Chan(c)).hamming_weight()
367
```

sage.coding.channel_constructions. **format_interval** (*t*)

Returns a formatted string representation of `t` .

This method should be called by any representation function in Channel classes.

Note: This is a helper function, which should only be used when implementing new channels.

INPUT:

- `t` – a list or a tuple

OUTPUT:

- a string

TESTS:

```
sage: from sage.coding.channel_constructions import format_interval
sage: t = (5, 5)
sage: format_interval(t)
```

```
'5'

sage: t = (2, 10)
sage: format_interval(t)
'between 2 and 10'
```

sage.coding.channel_constructions. **random_error_vector** (*n*, *F*, *error_positions*)

Return a vector of length `n` over `F` filled with random non-zero coefficients at the positions given by `error_positions`.

Note: This is a helper function, which should only be used when implementing new channels.

INPUT:

- `n` – the length of the vector
- `F` – the field over which the vector is defined
- `error_positions` – the non-zero positions of the vector

OUTPUT:

- a vector of `F`

AUTHORS:

This function is taken from codinglib (https://bitbucket.org/jsrn/codinglib/) and was written by Johan Nielsen.

EXAMPLES:

```
sage: from sage.coding.channel_constructions import random_error_vector
sage: random_error_vector(5, GF(2), [1,3])
(0, 1, 0, 1, 0)
```

CHAPTER

FIVE

SOURCE CODING

5.1 Huffman Encoding

This module implements functionalities relating to Huffman encoding and decoding.

AUTHOR:

- Nathann Cohen (2010-05): initial version.

5.1.1 Classes and functions

class sage.coding.source_coding.huffman. **Huffman** (*source*)
Bases: sage.structure.sage_object.SageObject

This class implements the basic functionalities of Huffman codes.

It can build a Huffman code from a given string, or from the information of a dictionary associating to each key (the elements of the alphabet) a weight (most of the time, a probability value or a number of occurrences).

INPUT:

- source – can be either
  - A string from which the Huffman encoding should be created.
  - A dictionary that associates to each symbol of an alphabet a numeric value. If we consider the frequency of each alphabetic symbol, then source is considered as the frequency table of the alphabet with each numeric (non-negative integer) value being the number of occurrences of a symbol. The numeric values can also represent weights of the symbols. In that case, the numeric values are not necessarily integers, but can be real numbers.

In order to construct a Huffman code for an alphabet, we use exactly one of the following methods:

1. Let source be a string of symbols over an alphabet and feed source to the constructor of this class. Based on the input string, a frequency table is constructed that contains the frequency of each unique symbol in source . The alphabet in question is then all the unique symbols in source . A significant implication of this is that any subsequent string that we want to encode must contain only symbols that can be found in source .
2. Let source be the frequency table of an alphabet. We can feed this table to the constructor of this class. The table source can be a table of frequencies or a table of weights.

Examples:

```
sage: from sage.coding.source_coding.huffman import Huffman, frequency_table
sage: h1 = Huffman("There once was a french fry")
sage: for letter, code in h1.encoding_table().iteritems():
....:     print("'{}' : {}".format(letter, code))
'a' : 0111
' ' : 00
'c' : 1010
'e' : 100
'f' : 1011
'h' : 1100
'o' : 11100
'n' : 1101
's' : 11101
'r' : 010
'T' : 11110
'w' : 11111
'y' : 0110
```

We can obtain the same result by "training" the Huffman code with the following table of frequency:

```
sage: ft = frequency_table("There once was a french fry"); ft
{' ': 5,
 'T': 1,
 'a': 2,
 'c': 2,
 'e': 4,
 'f': 2,
 'h': 2,
 'n': 2,
 'o': 1,
 'r': 3,
 's': 1,
 'w': 1,
 'y': 1}
sage: h2 = Huffman(ft)
```

Once `h1` has been trained, and hence possesses an encoding table, it is possible to obtain the Huffman encoding of any string (possibly the same) using this code:

```
sage: encoded = h1.encode("There once was a french fry"); encoded

↪'11110110010001010000111001101101010000111110111111010001110010110101001101101011000010110100
↪'
```

We can decode the above encoded string in the following way:

```
sage: h1.decode(encoded)
'There once was a french fry'
```

Obviously, if we try to decode a string using a Huffman instance which has been trained on a different sample (and hence has a different encoding table), we are likely to get some random-looking string:

```
sage: h3 = Huffman("There once were two french fries")
sage: h3.decode(encoded)
' wehnefetrhft ne ewrowrirTc'
```

This does not look like our original string.

Instead of using frequency, we can assign weights to each alphabetic symbol:

```
sage: from sage.coding.source_coding.huffman import Huffman
sage: T = {"a":45, "b":13, "c":12, "d":16, "e":9, "f":5}
sage: H = Huffman(T)
sage: L = ["deaf", "bead", "fab", "bee"]
sage: E = []
sage: for e in L:
....:     E.append(H.encode(e))
....:     print(E[-1])
111110101100
10111010111
11000101
10111011101
sage: D = []
sage: for e in E:
....:     D.append(H.decode(e))
....:     print(D[-1])
deaf
bead
fab
bee
sage: D == L
True
```

decode (*string*)

Decode the given string using the current encoding table.

INPUT:

- `string` – a string of Huffman encodings.

OUTPUT:

- The Huffman decoding of `string` .

EXAMPLES:

This is how a string is encoded and then decoded:

```
sage: from sage.coding.source_coding.huffman import Huffman
sage: str = "Sage is my most favorite general purpose computer algebra system"
sage: h = Huffman(str)
sage: encoded = h.encode(str); encoded

 ↪'000001101000101010110000111010100111001010100110110111001111011100101101000010110111111000
 ↪'
sage: h.decode(encoded)
'Sage is my most favorite general purpose computer algebra system'
```

TESTS:

Of course, the string one tries to decode has to be a binary one. If not, an exception is raised:

```
sage: h.decode('I clearly am not a binary string')
Traceback (most recent call last):
...
ValueError: Input must be a binary string.
```

encode (*string*)

Encode the given string based on the current encoding table.

INPUT:

- `string` – a string of symbols over an alphabet.

OUTPUT:

- A Huffman encoding of `string`.

EXAMPLES:

This is how a string is encoded and then decoded:

```
sage: from sage.coding.source_coding.huffman import Huffman
sage: str = "Sage is my most favorite general purpose computer algebra system"
sage: h = Huffman(str)
sage: encoded = h.encode(str); encoded

 ↪'000001101000101010110000111010100111001010100110110111001111011100101101000010110111110000
 ↪'
sage: h.decode(encoded)
'Sage is my most favorite general purpose computer algebra system'
```

encoding_table ( )

Returns the current encoding table.

INPUT:

- None.

OUTPUT:

- A dictionary associating an alphabetic symbol to a Huffman encoding.

EXAMPLES:

```
sage: from sage.coding.source_coding.huffman import Huffman
sage: str = "Sage is my most favorite general purpose computer algebra system"
sage: h = Huffman(str)
sage: T = sorted(h.encoding_table().items())
sage: for symbol, code in T:
....:     print("{} {}".format(symbol, code))
  101
S 00000
a 1101
b 110001
c 110000
e 010
f 110010
g 0001
i 10000
l 10011
m 0011
n 110011
o 0110
p 0010
r 1111
s 1110
t 0111
u 10001
v 00001
y 10010
```

tree ( )

Returns the Huffman tree corresponding to the current encoding.

INPUT:

•None.

OUTPUT:

•The binary tree representing a Huffman code.

EXAMPLES:

```
sage: from sage.coding.source_coding.huffman import Huffman
sage: str = "Sage is my most favorite general purpose computer algebra system"
sage: h = Huffman(str)
sage: T = h.tree(); T
Digraph on 39 vertices
sage: T.show(figsize=[20,20])
```

sage.coding.source_coding.huffman. **frequency_table** (*string*)

Return the frequency table corresponding to the given string.

INPUT:

•`string` – a string of symbols over some alphabet.

OUTPUT:

•A table of frequency of each unique symbol in `string` . If `string` is an empty string, return an empty table.

EXAMPLES:

The frequency table of a non-empty string:

```
sage: from sage.coding.source_coding.huffman import frequency_table
sage: str = "Stop counting my characters!"
sage: T = sorted(frequency_table(str).items())
sage: for symbol, code in T:
....:     print("{} {}".format(symbol, code))
  3
! 1
S 1
a 2
c 3
e 1
g 1
h 1
i 1
m 1
n 2
o 2
p 1
r 2
s 1
t 3
u 1
y 1
```

The frequency of an empty string:

```
sage: frequency_table("")
{}
```

CHAPTER

SIX

CANONICAL FORMS

6.1 Canonical forms and automorphism group computation for linear codes over finite fields

We implemented the algorithm described in *[Feu2009]* which computes the unique semilinearly isometric code (canonical form) in the equivalence class of a given linear code `C` . Furthermore, this algorithm will return the automorphism group of `C` , too.

The algorithm should be started via a further class *`LinearCodeAutGroupCanLabel`* . This class removes duplicated columns (up to multiplications by units) and zero columns. Hence, we can suppose that the input for the algorithm developed here is a set of points in $PG(k-1,q)$.

The implementation is based on the class `sage.groups.perm_gps.partn_ref2.refinement_generic.PartitionRef` . See the description of this algorithm in `sage.groups.perm_gps.partn_ref2.refinement_generic` . In the language given there, we have to implement the group action of $G = (GL(k,q) \times \mathbf{F}_q^{*n}) \rtimes Aut(\mathbf{F}_q)$ on the set $X = (\mathbf{F}_q^k)^n$ of $k \times n$ matrices over $\mathbf{F}_q$ (with the above restrictions).

The derived class here implements the stabilizers $G_{\Pi^{(I)}(x)}$ of the projections $\Pi^{(I)}(x)$ of x to the coordinates specified in the sequence I. Furthermore, we implement the inner minimization, i.e. the computation of a canonical form of the projection $\Pi^{(I)}(x)$ under the action of $G_{\Pi^{(I^{(i-1)})}(x)}$. Finally, we provide suitable homomorphisms of group actions for the refinements and methods to compute the applied group elements in $G \rtimes S_n$.

The algorithm also uses Jeffrey Leon's idea of maintaining an invariant set of codewords which is computed in the beginning, see `_init_point_hyperplane_incidence()` . An example for such a set is the set of all codewords of weight $\leq w$ for some uniquely defined w. In our case, we interpret the codewords as a set of hyperplanes (via the corresponding information word) and compute invariants of the bipartite, colored derived subgraph of the point-hyperplane incidence graph, see `PartitionRefinementLinearCode._point_refine()` and `PartitionRefinementLinearCode._hyp_refine()` .

Since we are interested in subspaces (linear codes) instead of matrices, our group elements returned in *`PartitionRefinementLinearCode.get_transporter()`* and *`PartitionRefinementLinearCode.get_autom_gens()`* will be elements in the group $(\mathbf{F}_q^{*n} \rtimes Aut(\mathbf{F}_q)) \rtimes S_n = (\mathbf{F}_q^{*n} \rtimes (Aut(\mathbf{F}_q) \times S_n)$.

AUTHORS:

- Thomas Feulner (2012-11-15): initial version

EXAMPLES:

Get the canonical form of the Simplex code:

```
sage: from sage.coding.codecan.codecan import PartitionRefinementLinearCode
sage: mat = codes.HammingCode(GF(3), 3).dual_code().generator_matrix()
sage: P = PartitionRefinementLinearCode(mat.ncols(), mat)
```

```
sage: cf = P.get_canonical_form(); cf
[1 0 0 0 0 1 1 1 1 1 1 1 1]
[0 1 0 1 1 0 0 1 1 2 2 1 2]
[0 0 1 1 2 1 2 1 2 1 2 0 0]
```

The transporter element is a group element which maps the input to its canonical form:

```
sage: cf.echelon_form() == (P.get_transporter() * mat).echelon_form()
True
```

The automorphism group of the input, i.e. the stabilizer under this group action, is returned by generators:

```
sage: P.get_autom_order_permutation() == GL(3, GF(3)).order()/(len(GF(3))-1)
True
sage: A = P.get_autom_gens()
sage: all( [(a*mat).echelon_form() == mat.echelon_form() for a in A])
True
```

class sage.coding.codecan.codecan. **InnerGroup**

Bases: object

This class implements the stabilizers $G_{\Pi^{(I)}(x)}$ described in sage.groups.perm_gps.partn_ref2.refinement_gener with $G = (GL(k,q) \times \mathbf{F}_q^n) \rtimes Aut(\mathbf{F}_q)$.

Those stabilizers can be stored as triples:

- rank - an integer in $\{0, \ldots, k\}$
- **row_partition - a partition of** $\{0, \ldots, k-1\}$ **with** discrete cells for all integers $i \geq rank$.
- frob_pow an integer in $\{0, \ldots, r-1\}$ if $q = p^r$

The group $G_{\Pi^{(I)}(x)}$ contains all elements $(A, \varphi, \alpha) \in G$, where

- A is a 2×2 blockmatrix, whose upper left matrix is a $k \times k$ diagonal matrix whose entries $A_{i,i}$ are constant on the cells of the partition row_partition . The lower left matrix is zero. And the right part is arbitrary.
- The support of the columns given by $i \in I$ intersect exactly one cell of the partition. The entry φ_i is equal to the entries of the corresponding diagonal entry of A.
- α **is a power of** $\tau^{frob_{pow}}$**, where** τ **denotes the** Frobenius automorphism of the finite field $\mathbf{F}_q$.

See *[Feu2009]* for more details.

column_blocks (*mat*)

Let mat be a matrix which is stabilized by self having no zero columns. We know that for each column of mat there is a uniquely defined cell in self.row_partition having a nontrivial intersection with the support of this particular column.

This function returns a partition (as list of lists) of the columns indices according to the partition of the rows given by self .

EXAMPLES:

```
sage: from sage.coding.codecan.codecan import InnerGroup
sage: I = InnerGroup(3)
sage: mat = Matrix(GF(3), [[0,1,0],[1,0,0], [0,0,1]])
sage: I.column_blocks(mat)
[[1], [0], [2]]
```

get_frob_pow ( )

Return the power of the Frobenius automorphism which generates the corresponding component of `self` .

EXAMPLES:

```
sage: from sage.coding.codecan.codecan import InnerGroup
sage: I = InnerGroup(10)
sage: I.get_frob_pow()
1
```

sage.coding.codecan.codecan. **OP_represent** (*n*, *merges*, *perm*)

Demonstration and testing.

TESTS:

```
sage: from sage.groups.perm_gps.partn_ref.automorphism_group_canonical_label␣
↪import OP_represent
sage: OP_represent(9, [(0,1),(2,3),(3,4)], [1,2,0,4,3,6,7,5,8])
Allocating OrbitPartition...
Allocation passed.
Checking that each element reports itself as its root.
Each element reports itself as its root.
Merging:
Merged 0 and 1.
Merged 2 and 3.
Merged 3 and 4.
Done merging.
Finding:
0 -> 0, root: size=2, mcr=0, rank=1
1 -> 0
2 -> 2, root: size=3, mcr=2, rank=1
3 -> 2
4 -> 2
5 -> 5, root: size=1, mcr=5, rank=0
6 -> 6, root: size=1, mcr=6, rank=0
7 -> 7, root: size=1, mcr=7, rank=0
8 -> 8, root: size=1, mcr=8, rank=0
Allocating array to test merge_perm.
Allocation passed.
Merging permutation: [1, 2, 0, 4, 3, 6, 7, 5, 8]
Done merging.
Finding:
0 -> 0, root: size=5, mcr=0, rank=2
1 -> 0
2 -> 0
3 -> 0
4 -> 0
5 -> 5, root: size=3, mcr=5, rank=1
6 -> 5
7 -> 5
8 -> 8, root: size=1, mcr=8, rank=0
Deallocating OrbitPartition.
Done.
```

sage.coding.codecan.codecan. **PS_represent** (*partition*, *splits*)

Demonstration and testing.

TESTS:

```
sage: from sage.groups.perm_gps.partn_ref.automorphism_group_canonical_label␣
 ↪import PS_represent
sage: PS_represent([[6],[3,4,8,7],[1,9,5],[0,2]], [6,1,8,2])
Allocating PartitionStack...
Allocation passed:
(0 1 2 3 4 5 6 7 8 9)
Checking that entries are in order and correct level.
Everything seems in order, deallocating.
Deallocated.
Creating PartitionStack from partition [[6], [3, 4, 8, 7], [1, 9, 5], [0, 2]].
PartitionStack's data:
entries -> [6, 3, 4, 8, 7, 1, 9, 5, 0, 2]
levels -> [0, 10, 10, 10, 0, 10, 10, 0, 10, -1]
depth = 0, degree = 10
(6|3 4 8 7|1 9 5|0 2)
Checking PS_is_discrete:
False
Checking PS_num_cells:
4
Checking PS_is_mcr, min cell reps are:
[6, 3, 1, 0]
Checking PS_is_fixed, fixed elements are:
[6]
Copying PartitionStack:
(6|3 4 8 7|1 9 5|0 2)
Checking for consistency.
Everything is consistent.
Clearing copy:
(0 3 4 8 7 1 9 5 6 2)
Splitting point 6 from original:
0
(6|3 4 8 7|1 9 5|0 2)
Splitting point 1 from original:
5
(6|3 4 8 7|1|5 9|0 2)
Splitting point 8 from original:
1
(6|8|3 4 7|1|5 9|0 2)
Splitting point 2 from original:
8
(6|8|3 4 7|1|5 9|2|0)
Getting permutation from PS2->PS:
[6, 1, 0, 8, 3, 9, 2, 7, 4, 5]
Finding first smallest:
Minimal element is 5, bitset is:
0000010001
Finding element 1:
Location is: 5
Bitset is:
0100000000
Deallocating PartitionStacks.
Done.
```

class sage.coding.codecan.codecan. **PartitionRefinementLinearCode**

Bases: sage.groups.perm_gps.partn_ref2.refinement_generic.PartitionRefinement_generic

See *sage.coding.codecan.codecan*.

EXAMPLES:

```
sage: from sage.coding.codecan.codecan import PartitionRefinementLinearCode
sage: mat = codes.HammingCode(GF(3), 3).dual_code().generator_matrix()
sage: P = PartitionRefinementLinearCode(mat.ncols(), mat)
sage: cf = P.get_canonical_form(); cf
[1 0 0 0 0 1 1 1 1 1 1 1 1]
[0 1 0 1 1 0 0 1 1 2 2 1 2]
[0 0 1 1 2 1 2 1 2 1 2 0 0]
```

```
sage: cf.echelon_form() == (P.get_transporter() * mat).echelon_form()
True
```

```
sage: P.get_autom_order_permutation() == GL(3, GF(3)).order()/(len(GF(3))-1)
True
sage: A = P.get_autom_gens()
sage: all( [(a*mat).echelon_form() == mat.echelon_form() for a in A])
True
```

get_autom_gens ( )

Return generators of the automorphism group of the initial matrix.

EXAMPLES:

```
sage: from sage.coding.codecan.codecan import PartitionRefinementLinearCode
sage: mat = codes.HammingCode(GF(3), 3).dual_code().generator_matrix()
sage: P = PartitionRefinementLinearCode(mat.ncols(), mat)
sage: A = P.get_autom_gens()
sage: all( [(a*mat).echelon_form() == mat.echelon_form() for a in A])
True
```

get_autom_order_inner_stabilizer ( )

Return the order of the stabilizer of the initial matrix under the action of the inner group G.

EXAMPLES:

```
sage: from sage.coding.codecan.codecan import PartitionRefinementLinearCode
sage: mat = codes.HammingCode(GF(3), 3).dual_code().generator_matrix()
sage: P = PartitionRefinementLinearCode(mat.ncols(), mat)
sage: P.get_autom_order_inner_stabilizer()
2
sage: mat2 = Matrix(GF(4, 'a'), [[1,0,1], [0,1,1]])
sage: P2 = PartitionRefinementLinearCode(mat2.ncols(), mat2)
sage: P2.get_autom_order_inner_stabilizer()
6
```

get_canonical_form ( )

Return the canonical form for this matrix.

EXAMPLES:

```
sage: from sage.coding.codecan.codecan import PartitionRefinementLinearCode
sage: mat = codes.HammingCode(GF(3), 3).dual_code().generator_matrix()
sage: P1 = PartitionRefinementLinearCode(mat.ncols(), mat)
sage: CF1 = P1.get_canonical_form()
sage: s = SemimonomialTransformationGroup(GF(3), mat.ncols()).an_element()
sage: P2 = PartitionRefinementLinearCode(mat.ncols(), s*mat)
sage: CF1 == P2.get_canonical_form()
True
```

get_transporter ( )

Return the transporter element, mapping the initial matrix to its canonical form.

EXAMPLES:

```
sage: from sage.coding.codecan.codecan import PartitionRefinementLinearCode
sage: mat = codes.HammingCode(GF(3), 3).dual_code().generator_matrix()
sage: P = PartitionRefinementLinearCode(mat.ncols(), mat)
sage: CF = P.get_canonical_form()
sage: t = P.get_transporter()
sage: (t*mat).echelon_form() == CF.echelon_form()
True
```

sage.coding.codecan.codecan. **SC_test_list_perms** (*L*, *n*, *limit*, *gap*, *limit_complain*, *test_contains*)

Test that the permutation group generated by list perms in L of degree n is of the correct order, by comparing with GAP. Don't test if the group is of size greater than limit.

TESTS:

```
sage: from sage.groups.perm_gps.partn_ref.automorphism_group_canonical_label␣
↪import SC_test_list_perms
sage: limit = 10^7
sage: def test_Sn_on_m_points(n, m, gap, contains):
....:     perm1 = [1,0] + list(range(2, m))
....:     perm2 = [(i+1)%n for i in range( n )] + list(range(n,m))
....:     SC_test_list_perms([perm1, perm2], m, limit, gap, 0, contains)
sage: for i in range(2,9):
....:     test_Sn_on_m_points(i,i,1,0)
sage: for i in range(2,9):
....:     test_Sn_on_m_points(i,i,0,1)
sage: for i in range(2,9):           # long time
....:     test_Sn_on_m_points(i,i,1,1) # long time
sage: test_Sn_on_m_points(8,8,1,1)
sage: def test_stab_chain_fns_1(n, gap, contains):
....:     perm1 = sum([[2*i+1,2*i] for i in range(n)], [])
....:     perm2 = [(i+1)%(2*n) for i in range( 2*n)]
....:     SC_test_list_perms([perm1, perm2], 2*n, limit, gap, 0, contains)
sage: for n in range(1,11):
...     test_stab_chain_fns_1(n, 1, 0)
sage: for n in range(1,11):
...     test_stab_chain_fns_1(n, 0, 1)
sage: for n in range(1,9):              # long time
...     test_stab_chain_fns_1(n, 1, 1)  # long time
sage: test_stab_chain_fns_1(11, 1, 1)
sage: def test_stab_chain_fns_2(n, gap, contains):
...     perms = []
...     for p,e in factor(n):
...         perm1 = [(p*(i//p)) + ((i+1)%p) for i in range(n)]
...         perms.append(perm1)
...     SC_test_list_perms(perms, n, limit, gap, 0, contains)
sage: for n in range(2,11):
...     test_stab_chain_fns_2(n, 1, 0)
sage: for n in range(2,11):
...     test_stab_chain_fns_2(n, 0, 1)
sage: for n in range(2,11):            # long time
...     test_stab_chain_fns_2(n, 1, 1) # long time
sage: test_stab_chain_fns_2(11, 1, 1)
```

```
sage: def test_stab_chain_fns_3(n, gap, contains):
...       perm1 = [(-i)%n for i in range( n )]
...       perm2 = [(i+1)%n for i in range( n )]
...       SC_test_list_perms([perm1, perm2], n, limit, gap, 0, contains)
sage: for n in range(2,20):
...       test_stab_chain_fns_3(n, 1, 0)
sage: for n in range(2,20):
...       test_stab_chain_fns_3(n, 0, 1)
sage: for n in range(2,14):             # long time
...       test_stab_chain_fns_3(n, 1, 1) # long time
sage: test_stab_chain_fns_3(20, 1, 1)
sage: def test_stab_chain_fns_4(n, g, gap, contains):
....:     perms = []
....:     for _ in range(g):
....:         perm = list(range(n))
....:         shuffle(perm)
....:         perms.append(perm)
....:     SC_test_list_perms(perms, n, limit, gap, 0, contains)
sage: for n in range(4,9):                  # long time
...       test_stab_chain_fns_4(n, 1, 1, 0) # long time
...       test_stab_chain_fns_4(n, 2, 1, 0) # long time
...       test_stab_chain_fns_4(n, 2, 1, 0) # long time
...       test_stab_chain_fns_4(n, 2, 1, 0) # long time
...       test_stab_chain_fns_4(n, 2, 1, 0) # long time
...       test_stab_chain_fns_4(n, 3, 1, 0) # long time
sage: for n in range(4,9):
...       test_stab_chain_fns_4(n, 1, 0, 1)
...       for j in range(6):
...           test_stab_chain_fns_4(n, 2, 0, 1)
...       test_stab_chain_fns_4(n, 3, 0, 1)
sage: for n in range(4,8):                  # long time
...       test_stab_chain_fns_4(n, 1, 1, 1) # long time
...       test_stab_chain_fns_4(n, 2, 1, 1) # long time
...       test_stab_chain_fns_4(n, 2, 1, 1) # long time
...       test_stab_chain_fns_4(n, 3, 1, 1) # long time
sage: test_stab_chain_fns_4(8, 2, 1, 1)
sage: def test_stab_chain_fns_5(n, gap, contains):
....:     perms = []
....:     m = n//3
....:     perm1 = list(range(2*m))
....:     shuffle(perm1)
....:     perm1 += range(2*m,n)
....:     perm2 = range(m,n)
....:     shuffle(perm2)
....:     perm2 = range(m) + perm2
....:     SC_test_list_perms([perm1, perm2], n, limit, gap, 0, contains)
sage: for n in [4..9]:                      # long time
...       for _ in range(2):                # long time
...           test_stab_chain_fns_5(n, 1, 0) # long time
sage: for n in [4..8]:                      # long time
...       test_stab_chain_fns_5(n, 0, 1)    # long time
sage: for n in [4..9]:                      # long time
....:     test_stab_chain_fns_5(n, 1, 1)      # long time
sage: def random_perm(x):
....:     shuffle(x)
....:     return x
sage: def test_stab_chain_fns_6(m,n,k, gap, contains):
....:     perms = []
```

```
....:     for i in range(k):
....:         perm = sum([random_perm(list(range(i*(n//m),min(n,(i+1)*(n//m)))))␣
↪for i in range(m)], [])
....:         perms.append(perm)
....:     SC_test_list_perms(perms, m*(n//m), limit, gap, 0, contains)
sage: for m in range(2,9):                          # long time
...     for n in range(m,3*m):                      # long time
...         for k in range(1,3):                    # long time
...             test_stab_chain_fns_6(m,n,k, 1, 0) # long time
sage: for m in range(2,10):
...     for n in range(m,4*m):
...         for k in range(1,3):
...             test_stab_chain_fns_6(m,n,k, 0, 1)
sage: test_stab_chain_fns_6(10,20,2, 1, 1)
sage: test_stab_chain_fns_6(8,16,2, 1, 1)
sage: test_stab_chain_fns_6(6,36,2, 1, 1)
sage: test_stab_chain_fns_6(4,40,3, 1, 1)
sage: test_stab_chain_fns_6(4,40,2, 1, 1)
sage: def test_stab_chain_fns_7(n, cop, gap, contains):
....:     perms = []
....:     for i in range(0,n//2,2):
....:         p = list(range(n))
....:         p[i] = i+1
....:         p[i+1] = i
....:     if cop:
....:         perms.append([c for c in p])
....:     else:
....:         perms.append(p)
....:     SC_test_list_perms(perms, n, limit, gap, 0, contains)
sage: for n in [6..14]:
....:     test_stab_chain_fns_7(n, 1, 1, 0)
....:     test_stab_chain_fns_7(n, 0, 1, 0)
sage: for n in [6..30]:
....:     test_stab_chain_fns_7(n, 1, 0, 1)
....:     test_stab_chain_fns_7(n, 0, 0, 1)
sage: for n in [6..14]:                          # long time
....:     test_stab_chain_fns_7(n, 1, 1, 1) # long time
....:     test_stab_chain_fns_7(n, 0, 1, 1) # long time
sage: test_stab_chain_fns_7(20, 1, 1, 1)
sage: test_stab_chain_fns_7(20, 0, 1, 1)
```

6.2 Canonical forms and automorphisms for linear codes over finite fields

We implemented the algorithm described in *[Feu2009]* which computes, a unique code (canonical form) in the equivalence class of a given linear code $C \leq \mathbf{F}_q^n$. Furthermore, this algorithm will return the automorphism group of C, too. You will find more details about the algorithm in the documentation of the class `LinearCodeAutGroupCanLabel` .

The equivalence of codes is modeled as a group action by the group $G = \mathbf{F}_q^{*n} \rtimes (Aut(\mathbf{F}_q) \times S_n)$ on the set of subspaces of $\mathbf{F}_q^n$. The group G will be called the semimonomial group of degree n.

The algorithm is started by initializing the class `LinearCodeAutGroupCanLabel` . When the object gets available, all computations are already finished and you can access the relevant data using the member functions:

- `get_canonical_form()`

- *get_transporter()*
- *get_autom_gens()*

People do also use some weaker notions of equivalence, namely **permutational** equivalence and monomial equivalence (**linear** isometries). These can be seen as the subgroups S_n and $\mathbf{F}_q^{*n} \rtimes S_n$ of G. If you are interested in one of these notions, you can just pass the optional parameter `algorithm_type` .

A second optional parameter `P` allows you to restrict the group of permutations S_n to a subgroup which respects the coloring given by `P` .

AUTHORS:

- Thomas Feulner (2012-11-15): initial version

REFERENCES:

EXAMPLES:

```
sage: from sage.coding.codecan.autgroup_can_label import LinearCodeAutGroupCanLabel
sage: C = codes.HammingCode(GF(3), 3).dual_code()
sage: P = LinearCodeAutGroupCanLabel(C)
sage: P.get_canonical_form().generator_matrix()
[1 0 0 0 0 1 1 1 1 1 1 1 1]
[0 1 0 1 1 0 0 1 1 2 2 1 2]
[0 0 1 1 2 1 2 1 2 1 2 0 0]
sage: LinearCode(P.get_transporter()*C.generator_matrix()) == P.get_canonical_form()
True
sage: A = P.get_autom_gens()
sage: all( [ LinearCode(a*C.generator_matrix()) == C for a in A])
True
sage: P.get_autom_order() == GL(3, GF(3)).order()
True
```

If the dimension of the dual code is smaller, we will work on this code:

```
sage: C2 = codes.HammingCode(GF(3), 3)
sage: P2 = LinearCodeAutGroupCanLabel(C2)
sage: P2.get_canonical_form().parity_check_matrix() == P.get_canonical_form().
↪generator_matrix()
True
```

There is a specialization of this algorithm to pass a coloring on the coordinates. This is just a list of lists, telling the algorithm which columns do share the same coloring:

```
sage: C = codes.HammingCode(GF(4, 'a'), 3).dual_code()
sage: P = LinearCodeAutGroupCanLabel(C, P=[ [0], [1], range(2, C.length()) ])
sage: P.get_autom_order()
864
sage: A = [a.get_perm() for a in P.get_autom_gens()]
sage: H = SymmetricGroup(21).subgroup(A)
sage: H.orbits()
[[1], [2], [3, 5, 4], [6, 10, 13, 20, 17, 9, 8, 11, 18, 15, 14, 16, 12, 19, 21, 7]]
```

We can also restrict the group action to linear isometries:

```
sage: P = LinearCodeAutGroupCanLabel(C, algorithm_type="linear")
sage: P.get_autom_order() == GL(3, GF(4, 'a')).order()
True
```

and to the action of the symmetric group only:

```
sage: P = LinearCodeAutGroupCanLabel(C, algorithm_type="permutational")
sage: P.get_autom_order() == C.permutation_automorphism_group().order()
True
```

class sage.coding.codecan.autgroup_can_label. **LinearCodeAutGroupCanLabel** (*C*, *P=None*, *algorithm_type='semilinear'*)

Canonical representatives and automorphism group computation for linear codes over finite fields.

There are several notions of equivalence for linear codes: Let C, D be linear codes of length n and dimension k. C and D are said to be

- permutational equivalent, if there is some permutation $\pi \in S_n$ such that $(c_{\pi(0)}, \ldots, c_{\pi(n-1)}) \in D$ for all $c \in C$.
- linear equivalent, if there is some permutation $\pi \in S_n$ and a vector $\phi \in \mathbf{F}_q^{*n}$ of units of length n such that $(c_{\pi(0)}\phi_0^{-1}, \ldots, c_{\pi(n-1)}\phi_{n-1}^{-1}) \in D$ for all $c \in C$.
- semilinear equivalent, if there is some permutation $\pi \in S_n$, a vector ϕ of units of length n and a field automorphism α such that $(\alpha(c_{\pi(0)})\phi_0^{-1}, \ldots, \alpha(c_{\pi(n-1)})\phi_{n-1}^{-1}) \in D$ for all $c \in C$.

These are group actions. This class provides an algorithm that will compute a unique representative D in the orbit of the given linear code C. Furthermore, the group element g with $g * C = D$ and the automorphism group of C will be computed as well.

There is also the possibility to restrict the permutational part of this action to a Young subgroup of S_n. This could be achieved by passing a partition P (as a list of lists) of the set $\{0, \ldots, n-1\}$. This is an option which is also available in the computation of a canonical form of a graph, see sage.graphs.generic_graph.GenericGraph.canonical_label() .

EXAMPLES:

```
sage: from sage.coding.codecan.autgroup_can_label import␣
↪LinearCodeAutGroupCanLabel
sage: C = codes.HammingCode(GF(3), 3).dual_code()
sage: P = LinearCodeAutGroupCanLabel(C)
sage: P.get_canonical_form().generator_matrix()
[1 0 0 0 0 1 1 1 1 1 1 1 1]
[0 1 0 1 1 0 0 1 1 2 2 1 2]
[0 0 1 1 2 1 2 1 2 1 2 0 0]
sage: LinearCode(P.get_transporter()*C.generator_matrix()) == P.get_canonical_
↪form()
True
sage: a = P.get_autom_gens()[0]
sage: (a*C.generator_matrix()).echelon_form() == C.generator_matrix().echelon_
↪form()
True
sage: P.get_autom_order() == GL(3, GF(3)).order()
True
```

get_PGammaL_gens ( )

Return the set of generators translated to the group $P\Gamma L(k, q)$.

There is a geometric point of view of code equivalence. A linear code is identified with the multiset of points in the finite projective geometry $PG(k-1, q)$. The equivalence of codes translates to the natural action of $P\Gamma L(k, q)$. Therefore, we may interpret the group as a subgroup of $P\Gamma L(k, q)$ as well.

EXAMPLES:

```
sage: from sage.coding.codecan.autgroup_can_label import␣
↪LinearCodeAutGroupCanLabel
sage: C = codes.HammingCode(GF(4, 'a'), 3).dual_code()
sage: A = LinearCodeAutGroupCanLabel(C).get_PGammaL_gens()
sage: Gamma = C.generator_matrix()
sage: N = [ x.monic() for x in Gamma.columns() ]
sage: all([ (g[0]*n.apply_map(g[1])).monic() in N for n in N for g in A])
True
```

get_PGammaL_order ( )

Return the size of the automorphism group as a subgroup of $P\Gamma L(k, q)$.

There is a geometric point of view of code equivalence. A linear code is identified with the multiset of points in the finite projective geometry $PG(k-1, q)$. The equivalence of codes translates to the natural action of $P\Gamma L(k, q)$. Therefore, we may interpret the group as a subgroup of $P\Gamma L(k, q)$ as well.

EXAMPLES:

```
sage: from sage.coding.codecan.autgroup_can_label import␣
↪LinearCodeAutGroupCanLabel
sage: C = codes.HammingCode(GF(4, 'a'), 3).dual_code()
sage: LinearCodeAutGroupCanLabel(C).get_PGammaL_order() == GL(3, GF(4, 'a')).
↪order()*2/3
True
```

get_autom_gens ( )

Return a generating set for the automorphism group of the code.

EXAMPLES:

```
sage: from sage.coding.codecan.autgroup_can_label import␣
↪LinearCodeAutGroupCanLabel
sage: C = codes.HammingCode(GF(2), 3).dual_code()
sage: A = LinearCodeAutGroupCanLabel(C).get_autom_gens()
sage: Gamma = C.generator_matrix().echelon_form()
sage: all([(g*Gamma).echelon_form() == Gamma for g in A])
True
```

get_autom_order ( )

Return the size of the automorphism group of the code.

EXAMPLES:

```
sage: from sage.coding.codecan.autgroup_can_label import␣
↪LinearCodeAutGroupCanLabel
sage: C = codes.HammingCode(GF(2), 3).dual_code()
sage: LinearCodeAutGroupCanLabel(C).get_autom_order()
168
```

get_canonical_form ( )

Return the canonical orbit representative we computed.

EXAMPLES:

```
sage: from sage.coding.codecan.autgroup_can_label import␣
↪LinearCodeAutGroupCanLabel
sage: C = codes.HammingCode(GF(3), 3).dual_code()
sage: CF1 = LinearCodeAutGroupCanLabel(C).get_canonical_form()
sage: s = SemimonomialTransformationGroup(GF(3), C.length()).an_element()
```

```
sage: C2 = LinearCode(s*C.generator_matrix())
sage: CF2 = LinearCodeAutGroupCanLabel(C2).get_canonical_form()
sage: CF1 == CF2
True
```

get_transporter ( )

Return the element which maps the code to its canonical form.

EXAMPLES:

```
sage: from sage.coding.codecan.autgroup_can_label import␣
↪LinearCodeAutGroupCanLabel
sage: C = codes.HammingCode(GF(2), 3).dual_code()
sage: P = LinearCodeAutGroupCanLabel(C)
sage: g = P.get_transporter()
sage: D = P.get_canonical_form()
sage: (g*C.generator_matrix()).echelon_form() == D.generator_matrix().echelon_
↪form()
True
```

CHAPTER

SEVEN

OTHER TOOLS

7.1 Management of relative finite field extensions

Considering a *absolute field* F_{q^m} and a *relative_field* F_q, with $q = p^s$, p being a prime and s, m being integers, this file contains a class to take care of the representation of F_{q^m}-elements as F_q-elements.

Warning: As this code is experimental, a warning is thrown when a relative finite field extension is created for the first time in a session (see `sage.misc.superseded.experimental`).

TESTS:

```
sage: from sage.coding.relative_finite_field_extension import *
sage: Fqm.<aa> = GF(16)
sage: Fq.<a> = GF(4)
sage: RelativeFiniteFieldExtension(Fqm, Fq)
doctest:...: FutureWarning: This class/method/function is marked as experimental.␣
↪It, its functionality or its interface might change without a formal deprecation.
See http://trac.sagemath.org/20284 for details.
Relative field extension between Finite Field in aa of size 2^4 and Finite Field in␣
↪a of size 2^2
```

class `sage.coding.relative_finite_field_extension.` **`RelativeFiniteFieldExtension`** (*absolute_field*, *relative_field*, *embedding=None*)

Bases: `sage.structure.sage_object.SageObject`

Considering p a prime number, n an integer and three finite fields F_p, F_q and F_{q^m}, this class contains a set of methods to manage the representation of elements of the relative extension F_{q^m} over F_q.

INPUT:

- `absolute_field` , `relative_field` – two finite fields, `relative_field` being a subfield of `absolute_field`
- `embedding` – (default: `None`) an homomorphism from `relative_field` to `absolute_field` . If `None` is provided, it will default to the first homomorphism of the list of homomorphisms Sage can build.

EXAMPLES:

```
sage: from sage.coding.relative_finite_field_extension import *
sage: Fqm.<aa> = GF(16)
sage: Fq.<a> = GF(4)
sage: RelativeFiniteFieldExtension(Fqm, Fq)
Relative field extension between Finite Field in aa of size 2^4 and Finite Field␣
↪in a of size 2^2
```

It is possible to specify the embedding to use from `relative_field` to `absolute_field`:

```
sage: Fqm.<aa> = GF(16)
sage: Fq.<a> = GF(4)
sage: FE = RelativeFiniteFieldExtension(Fqm, Fq, embedding=Hom(Fq, Fqm)[1])
sage: FE.embedding() == Hom(Fq, Fqm)[1]
True
```

absolute_field ( )

Returns the absolute field of `self`.

EXAMPLES:

```
sage: from sage.coding.relative_finite_field_extension import *
sage: Fqm.<aa> = GF(16)
sage: Fq.<a> = GF(4)
sage: FE = RelativeFiniteFieldExtension(Fqm, Fq)
sage: FE.absolute_field()
Finite Field in aa of size 2^4
```

absolute_field_basis ( )

Returns a basis of the absolute field over the prime field.

EXAMPLES:

```
sage: from sage.coding.relative_finite_field_extension import *
sage: Fqm.<aa> = GF(16)
sage: Fq.<a> = GF(4)
sage: FE = RelativeFiniteFieldExtension(Fqm, Fq)
sage: FE.absolute_field_basis()
[1, aa, aa^2, aa^3]
```

absolute_field_degree ( )

Let F_p be the base field of our absolute field F_{q^m}. Returns sm where $p^{sm} = q^m$

EXAMPLES:

```
sage: from sage.coding.relative_finite_field_extension import *
sage: Fqm.<aa> = GF(16)
sage: Fq.<a> = GF(4)
sage: FE = RelativeFiniteFieldExtension(Fqm, Fq)
sage: FE.absolute_field_degree()
4
```

absolute_field_representation (*a*)

Returns an absolute field representation of the relative field vector `a`.

INPUT:

- `a` – a vector in the relative extension field

EXAMPLES:

```
sage: from sage.coding.relative_finite_field_extension import *
sage: Fqm.<aa> = GF(16)
sage: Fq.<a> = GF(4)
sage: FE = RelativeFiniteFieldExtension(Fqm, Fq)
sage: b = aa^3 + aa^2 + aa + 1
sage: rel = FE.relative_field_representation(b)
sage: FE.absolute_field_representation(rel) == b
True
```

cast_into_relative_field (*b*, *check=True*)

Casts an absolute field element into the relative field (if possible). This is the inverse function of the field embedding.

INPUT:

- b – an element of the absolute field which also lies in the relative field.

EXAMPLES:

```
sage: from sage.coding.relative_finite_field_extension import *
sage: Fqm.<aa> = GF(16)
sage: Fq.<a> = GF(4)
sage: FE = RelativeFiniteFieldExtension(Fqm, Fq)
sage: phi = FE.embedding()
sage: b = aa^2 + aa
sage: FE.is_in_relative_field(b)
True
sage: FE.cast_into_relative_field(b)
a
sage: phi(FE.cast_into_relative_field(b)) == b
True
```

embedding ( )

Returns the embedding which is used to go from the relative field to the absolute field.

EXAMPLES:

```
sage: from sage.coding.relative_finite_field_extension import *
sage: Fqm.<aa> = GF(16)
sage: Fq.<a> = GF(4)
sage: FE = RelativeFiniteFieldExtension(Fqm, Fq)
sage: FE.embedding()
Ring morphism:
 From: Finite Field in a of size 2^2
 To:   Finite Field in aa of size 2^4
 Defn: a |--> aa^2 + aa
```

extension_degree ( )

Returns m, teh extension degree of the absiolute field over the relative field.

EXAMPLES:

```
sage: from sage.coding.relative_finite_field_extension import *
sage: Fqm.<aa> = GF(64)
sage: Fq.<a> = GF(4)
sage: FE = RelativeFiniteFieldExtension(Fqm, Fq)
sage: FE.extension_degree()
3
```

is_in_relative_field (*b*)
Returns `True` if `b` is in the relative field.

INPUT:

- `b` – an element of the absolute field.

EXAMPLES:

```
sage: from sage.coding.relative_finite_field_extension import *
sage: Fqm.<aa> = GF(16)
sage: Fq.<a> = GF(4)
sage: FE = RelativeFiniteFieldExtension(Fqm, Fq)
sage: FE.is_in_relative_field(aa^2 + aa)
True
sage: FE.is_in_relative_field(aa^3)
False
```

prime_field ( )
Returns the base field of our absolute and relative fields.

EXAMPLES:

```
sage: from sage.coding.relative_finite_field_extension import *
sage: Fqm.<aa> = GF(16)
sage: Fq.<a> = GF(4)
sage: FE = RelativeFiniteFieldExtension(Fqm, Fq)
sage: FE.prime_field()
Finite Field of size 2
```

relative_field ( )
Returns the relative field of `self` .

EXAMPLES:

```
sage: from sage.coding.relative_finite_field_extension import *
sage: Fqm.<aa> = GF(16)
sage: Fq.<a> = GF(4)
sage: FE = RelativeFiniteFieldExtension(Fqm, Fq)
sage: FE.relative_field()
Finite Field in a of size 2^2
```

relative_field_basis ( )
Returns a basis of the relative field over the prime field.

EXAMPLES:

```
sage: from sage.coding.relative_finite_field_extension import *
sage: Fqm.<aa> = GF(16)
sage: Fq.<a> = GF(4)
sage: FE = RelativeFiniteFieldExtension(Fqm, Fq)
sage: FE.relative_field_basis()
[1, a]
```

relative_field_degree ( )
Let F_p be the base field of our relative field F_q. Returns s where $p^s = q$

EXAMPLES:

```
sage: from sage.coding.relative_finite_field_extension import *
sage: Fqm.<aa> = GF(16)
sage: Fq.<a> = GF(4)
sage: FE = RelativeFiniteFieldExtension(Fqm, Fq)
sage: FE.relative_field_degree()
2
```

relative_field_representation (*b*)

Returns a vector representation of the field element `b` in the basis of the absolute field over the relative field.

INPUT:

- `b` – an element of the absolute field

EXAMPLES:

```
sage: from sage.coding.relative_finite_field_extension import *
sage: Fqm.<aa> = GF(16)
sage: Fq.<a> = GF(4)
sage: FE = RelativeFiniteFieldExtension(Fqm, Fq)
sage: b = aa^3 + aa^2 + aa + 1
sage: FE.relative_field_representation(b)
(1, a + 1)
```

CHAPTER

EIGHT

DEPRECATED MODULES

8.1 Deprecated name for sage.coding.self_dual_codes

CHAPTER

NINE

INDICES AND TABLES

- Index
- Module Index
- Search Page

[Nielsen] Johan S. R. Nielsen, (https://bitbucket.org/jsrn/codinglib/)

[BS03] I. Bouyukliev and J. Simonis, Some new results on optimal codes over F_5, Designs, Codes and Cryptography 30, no. 1 (2003): 97-111, http://www.moi.math.bas.bg/moiuser/~iliya/pdf_site/gf5srev.pdf,

[ChenDB] Eric Chen, Online database of two-weight codes, http://moodle.tec.hkr.se/~chen/research/2-weight-codes/search.php

[Kohnert07] A. Kohnert, Constructing two-weight codes with prescribed groups of automorphisms, Discrete applied mathematics 155, no. 11 (2007): 1451-1457. http://linearcodes.uni-bayreuth.de/twoweight/

[Disset00] L. Dissett, Combinatorial and computational aspects of finite geometries, 2000, https://tspace.library.utoronto.ca/bitstream/1807/14575/1/NQ49844.pdf

[HP] W. C. Huffman and V. Pless, Fundamentals of error-correcting codes, Cambridge Univ. Press, 2003.

[Gu] GUAVA manual, http://www.gap-system.org/Packages/guava.html

[Du04] I. Duursma, "Combinatorics of the two-variable zeta function", Finite fields and applications, 109-136, Lecture Notes in Comput. Sci., 2948, Springer, Berlin, 2004.

[vL] J. van Lint, Introduction to coding theory, 3rd ed., Springer-Verlag GTM, 86, 1999.

[BS11] E. Byrne and A. Sneyd, On the Parameters of Codes with Two Homogeneous Weights. WCC 2011-Workshop on coding and cryptography, pp. 81-90. 2011. https://hal.inria.fr/inria-00607341/document

[D] I. Duursma "Extremal weight enumerators and ultraspherical polynomials" Discrete Mathematics 268 (1), 103-127

[Du01] I. Duursma, "From weight enumerators to zeta functions", in Discrete Applied Mathematics, vol. 111, no. 1-2, pp. 55-73, 2001.

[HJ04] Tom Hoeholdt and Joern Justesen, A Course In Error-Correcting Codes, EMS, 2004

[G02] Shuhong Gao, A new algorithm for decoding Reed-Solomon Codes, January 31, 2002

[R06] Ron Roth, Introduction to Coding Theory, Cambridge University Press, 2006

[GS99] Venkatesan Guruswami and Madhu Sudan, Improved Decoding of Reed-Solomon Codes and Algebraic-Geometric Codes, 1999

[N13] Johan S. R. Nielsen, List Decoding of Algebraic Codes, Ph.D. Thesis, Technical University of Denmark, 2013

[J] D. Joyner, Toric codes over finite fields, Applicable Algebra in Engineering, Communication and Computing, 15, (2004), p. 63-79.

[BM] Bazzi and Mitter, {it Some constructions of codes from group actions}, (preprint March 2003, available on Mitter's MIT website).

[Jresidue] D. Joyner, {it On quadratic residue codes and hyperelliptic curves}, (preprint 2006)

[De73] P. Delsarte, An algebraic approach to the association schemes of coding theory, Philips Res. Rep., Suppl., vol. 10, 1973.

[Feu2009] T. Feulner. The Automorphism Groups of Linear Codes and Canonical Representatives of Their Semilinear Isometry Classes. Advances in Mathematics of Communications 3 (4), pp. 363-383, Nov 2009

C

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z