## CONTENTS

1 Basic Coding Theory objects .......... 1
2 Catalogs .......... 57
3 Code constructions .......... 67
4 Derived Code Constructions .......... 129
5 Methods and Operations related to Linear Codes .......... 141
6 Source coding .......... 149
7 Other modules .......... 155
8 Indices and Tables .......... 191
   Bibliography .......... 193
   Python Module Index .......... 195
   Index .......... 197
1.1 Generic structures for linear codes

1.1.1 Linear Codes

Let $F = F_2$ 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.

1.1.2 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
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
[5, 3] linear code over GF(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 https://doc.sagemath.org/html/en/thematic_tutorials/coding_theory.html

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

REFERENCES:

- [HP2003]
- [Gu]

AUTHORS:

- David Joyner (2005-11-22, 2006-12-03): initial version
• 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 (2006-09): modified decode syntax, fixed bug in is_galois_closed, added LinearCode_from_vectorspace, extended_code, zeta_function

• David Joyner (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 (2008-03): 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

```python
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:
```

1.1. Generic structures for linear codes
• 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.

• Add the following two lines on the class level:

```
_registered_encoders = {}
_registered_decoders = {}
```

• 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 a generic implementation of the method __eq__ which uses the gener-
ator matrix and is quite slow. In subclasses you are encouraged to override __eq__ and __hash__.

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:
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: sorted(C.decoders_available())
['InformationSet', 'MyDecoder', 'NearestNeighbor', 'Syndrome']
```

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

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

### 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: sorted(C.encoders_available())
['MyEncoder', 'Systematic']
```

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

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

`ambient_space()`

Returns the ambient vector space of `self`.

EXAMPLES:

```python
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 [HP2003]): 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

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

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 $\lambda$ 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

```python
In [78]: t = given
In [79]: v = n
In [80]: k = i  # (k not to be confused with dim(C))
In [81]: b = Ai
In [82]: lambda_ = b*binomial(k,t)/binomial(v,t)  # (by Theorem 8.1.6, p 294, in [HP2003]_)
```

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.GolayCode(GF(2))  # 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:

Note, this result can depend on the PRNG state in libgap in a way that depends on which packages are loaded, so we must re-seed GAP to ensure a consistent result for this example:

sage: libgap.set_seed(0)
0
sage: C = codes.HammingCode(GF(4, 'z'), 3)
sage: C.automorphism_group_gens()
(((1, z, z + 1, z, l, 1, l, z, z + 1, z, z, 1, 1, z, z + 1, z, z, 1, z, z + 1, z, z, z + 1, z); (1,5,18,7,11,8)(2,12,21)(3,20,14,10,19,15)(4,9)(13,17,16), Ring
endomorphism of Finite Field in z of size 2^2
Defn: z |--> z + 1),
((1, l, z, z + 1, z, 1, l, z, z + 1, z, z, 1, 1, z, z + 1, z, z, 1, z, z + 1, z, z, z + 1, z); (2,11)(3,13)(4,14)(5,20)(6,17)(8,15)(16,19), Ring
endomorphism of Finite Field in z of size 2^2
Defn: z |--> z + 1),
((z, z, z, z, z, z, 1, l, 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, 1, 1, z, z + 1, z, z + 1, z + 1, 1, z + 1, z + 1, 1, 1, z + 1, z + 1, 1, 1, z, z, z + 1, z + 1, z); (1,6)(2,20,9,16)(3,10,8,11)(4,15,21,5)(12,17)(13,14,19,18),
Ring endomorphism of Finite Field in z of size 2^2
Defn: z |--> z),
((z, 1, 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)
base_field()

Return the base field of self.

EXAMPLES:

```python
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], [0,0,1,1,1,1,1]])
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:

```python
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)]
```

(continues on next page)
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 \( b_i(C) = (\binom{n}{i}, d + i)^{-1} B_{d+i}(C). \)) 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 \( (\text{equivalence}='\text{semilinear}') \)

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:

```python
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`:

```python
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:

```python
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:

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

characteristic_polynomial()  
Returns the characteristic polynomial of a linear code, as defined in [Lin1999].

EXAMPLES:

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

chinen_polynomial()  
Returns the Chinen zeta polynomial of the code.

EXAMPLES:

```python
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.GolayCode(GF(3), False)
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:


**covering_radius()**

Return the minimal integer \( r \) such that any element in the ambient space of \( \text{self} \) has distance at most \( r \) to a codeword of \( \text{self} \).

This method requires the optional GAP package Guava.

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

*Note:* This method is currently not implemented on codes over base fields of cardinality greater than 256 due to limitations in the underlying algorithm of GAP.

**EXAMPLES:**

```
sage: C = codes.HammingCode(GF(2), 5)
sage: C.covering_radius()  # optional - gap_packages (Guava package)
1
sage: C = codes.random_linear_code(GF(263), 5, 1)
sage: C.covering_radius()  # optional - gap_packages (Guava package)
Traceback (most recent call last):
...  
NotImplementedError: the GAP algorithm that Sage is using is limited to __ computing with fields of size at most 256
```

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

Corrects the errors in \( \text{word} \) and returns a codeword.

**INPUT:**

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

**OUTPUT:**

- A vector of \( \text{self} \).

**EXAMPLES:**

```
sage: G = Matrix(GF(2),[[1,1,0,0,0,0,0],[1,0,0,1,1,0,0],[0,1,0,1,0,1,0],[1,1,0,0,1,0,0]])
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: sorted(C.decoders_available())
['InformationSet', 'NearestNeighbor', 'Syndrome']
sage: C.decode_to_code(w_err, 'NearestNeighbor')
(1, 1, 0, 0, 1, 1, 0)

code_to_message(word, decoder_name=None, *args, **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.
• args, 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: sorted(C.decoders_available())
['InformationSet', 'NearestNeighbor', 'Syndrome']
sage: C.decode_to_message(word, 'NearestNeighbor')
(0, 1, 1, 0)

decoder (decoder_name=None, *args, **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.
• args, 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:
If the name of a decoder which is not known by `self` is passed, an exception will be raised:

```
sage: sorted(C.decoders_available())
['InformationSet', 'NearestNeighbor', 'Syndrome']
sage: C.decoder('Try')
Traceback (most recent call last):
...
ValueError: There is no Decoder named 'Try'. The known Decoders are: ['InformationSet', 'NearestNeighbor', 'Syndrome']
```

Some decoders take extra arguments. If the user forgets to supply these, the error message attempts to be helpful:

```
sage: C.decoder('InformationSet')
Traceback (most recent call last):
...
ValueError: Constructing the InformationSet decoder failed, possibly due to missing or incorrect parameters. The constructor requires the arguments ['number_errors']. It takes the optional arguments ['algorithm']. It accepts unspecified arguments as well. See the documentation of sage.coding.information_set_decoder.LinearCodeInformationSetDecoder for more details.
```

### decoders_available(classes=False)

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

**INPUT:**
- `classes` – (default: `False`) if `classes` is set to `True`, return instead a `dict` mapping available decoder name to the associated decoder class.

**OUTPUT:** a list of strings, or a `dict` mapping strings to classes.

**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()
['InformationSet', 'NearestNeighbor', 'Syndrome']
sage: dictionary = C.decoders_available(True)
sage: sorted(dictionary.keys())
['InformationSet', 'NearestNeighbor', 'Syndrome']
sage: dictionary['NearestNeighbor']
<class 'sage.coding.linear_code.LinearCodeNearestNeighborDecoder'>
```

### 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:**

```python
sage: C1 = codes.HammingCode(GF(2), 3)
sage: C2 = C1.direct_sum(C1); C2
[14, 8] linear code over GF(2)
sage: C3 = C1.direct_sum(C2); C3
[21, 12] linear code over GF(2)
```

**divisor**()

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

**EXAMPLES:**

```python
sage: C = codes.GolayCode(GF(2))
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:**

```python
sage: C = codes.HammingCode(GF(2), 3)
sage: C.dual_code()
[7, 3] linear code over GF(2)
sage: C = codes.HammingCode(GF(4, 'a'), 3)
sage: C.dual_code()
[21, 3] linear code over GF(4)
```

**encode**(word, encoder_name=None, *args, **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.
- **args, 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.

One can use the following shortcut to encode a word

```python
C(word)
```

**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],[1,1,0,1,0,1,0]])
sage: C = LinearCode(G)
sage: word = vector((0, 1, 1, 0))
sage: C.encode(word)
(1, 1, 0, 0, 1, 1, 0)
sage: C(word)
(1, 1, 0, 0, 1, 1, 0)
```

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

```
sage: sorted(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, *args, **kwargs)

Returns an encoder of self.

The returned encoder provided by this method is cached.

This method 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.

• args, 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],[1,1,0,1,0,1,0]])
sage: C = LinearCode(G)
sage: C.encoder()
Generator matrix-based encoder for [7, 4] linear code over GF(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:
```python
sage: sorted(C.encoders_available())
['GeneratorMatrix', 'Systematic']
sage: C.encoder('NonExistingEncoder')
Traceback (most recent call last):
  ...
ValueError: There is no Encoder named 'NonExistingEncoder'. The known
  Encoders are: ['GeneratorMatrix', 'Systematic']
```

Some encoders take extra arguments. If the user incorrectly supplies these, the error message attempts to be helpful:

```python
sage: C.encoder('Systematic', strange_parameter=True)
Traceback (most recent call last):
  ...
ValueError: Constructing the Systematic encoder failed, possibly due to
  missing or incorrect parameters.
The constructor requires no arguments.
It takes the optional arguments ['systematic_positions'].
See the documentation of sage.coding.linear_code.LinearCodeSystematicEncoder
  for more details.
```

**encoders_available**(classes=False)

Returns a list of the available encoders’ names for self.

INPUT:

- `classes` – (default: False) if classes is set to True, return instead a dict mapping available encoder name to the associated encoder class.

OUTPUT: a list of strings, or a dict mapping strings to classes.

EXAMPLES:

```python
sage: G = Matrix(GF(2), [[1,1,0,0,0,0,0],[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: dictionary = C.encoders_available(True)
sage: sorted(dictionary.items())
[('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:

```python
sage: C = codes.HammingCode(GF(4,'a'), 3)
sage: C
[21, 18] Hamming Code over GF(4)
sage: Cx = C.extended_code()
sage: Cx
Extension of [21, 18] Hamming Code over GF(4)
```

**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:

```python
sage: C = codes.HammingCode(GF(4,'a'), 3)
sage: Cc = C.galois_closure(GF(2))
sage: C; Cc
[21, 18] Hamming Code over GF(4)
[21, 20] linear code over GF(4)
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($\text{encoder\_name}=None, **\text{kwargs}$)

Returns a generator matrix of self.

INPUT:

- $\text{encoder\_name}$ – (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.

- $\text{kwargs}$ – all additional arguments are forwarded to the construction of the encoder that is used.

EXAMPLES:

```python
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]
```

gens()

Returns the generators of this code as a list of vectors.

EXAMPLES:

```python
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),
 (0, 1, 1, 1, 1, 0, 0), (1, 1, 0, 1, 1, 1, 0), (1, 1, 1, 0, 1, 1, 0), (1, 0, 1, 1, 1, 0, 0)]
```

genus()

Returns the “Duursma genus” of the code, $\gamma_C = n + 1 - k - d$.

EXAMPLES:

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

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

1.1. Generic structures for linear codes 17
**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:**

```python
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:**

```python
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:**

```python
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:**

```python
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]])
(continues on next page)```
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.CyclicCode(length = 7, generator_pol = g); C1
[7, 4] Cyclic Code over GF(2)
sage: C2 = codes.HammingCode(GF(2), 3); C2
[7, 4] Hamming Code over GF(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 $C^d$ has minimum weight $\geq 3$, i.e. when no two coordinate positions of $C$ are linearly independent (cf. definition 3 from [BS2011] or 9.8.1 from [BH12]).

**EXAMPLES:**

```
sage: C = codes.GolayCode(GF(2), False)
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
```

**is_self_dual**()

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

**EXAMPLES:**
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:

```python
sage: C = codes.GolayCode(GF(2))
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:

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

list()
Return a list of all elements of this linear code.

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

minimum_distance(algorithm=None)
Returns the minimum distance of self.

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 – (default: None) the name of the algorithm to use to perform minimum distance computation. If set to None, GAP methods will be used. algorithm can be: - "Guava", which will use optional GAP package Guava

OUTPUT:
• Integer, minimum distance of this code

EXAMPLES:
MS = MatrixSpace(GF(3),4,7)
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]])
C = LinearCode(G)
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.:
C.minimum_distance(algorithm="gap")
3
C.minimum_distance(algorithm="guava") # optional - gap_packages (Guava package)
3

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

EXAMPLES:
MS = MatrixSpace(GF(2),4,8)
G = MS([[1,0,0,0,1,1,1,0], [0,1,1,0,0,0,0,0], [0,0,0,0,0,0,0,1], [0,0,0,0,0,0,0,1]])
(continues on next page)
Now type “C.module_composition_factors(gp)” to get the record printed.

\texttt{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 $C$.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: C = codes.HammingCode(GF(2), 3)
sage: Cperp = C.dual_code()
sage: C; Cperp
[7, 4] Hamming Code over GF(2)
[7, 3] linear code over GF(2)
sage: C.generator_matrix()
[1 0 0 0 1 1]
[0 1 0 1 0 1]
[0 0 1 0 1 0]
[0 0 0 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 1 1]
[0 1 0 1 0 1]
[0 0 1 0 1 0]
[0 0 0 1 1 1]
sage: Cperp.generator_matrix()
[1 0 1 0 1 0 1]
[0 1 0 0 0 1 1]
[0 0 0 1 1 1 1]
\end{verbatim}

\texttt{permutation_automorphism_group}(algorithm='partition')

If $C$ is an $[n, k, d]$ code over $F$, this function computes the subgroup $Aut(C) \subseteq S_n$ 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 \texttt{automorphism_group_gens()} to compute generators of the full automorphism group.

\textbf{INPUT:}

\begin{itemize}
  \item \texttt{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 \texttt{canonical_representative()} to access it).
\end{itemize}

\textbf{OUTPUT:}

\begin{itemize}
  \item Permutation automorphism group
\end{itemize}

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

Other examples:

```python
sage: C = codes.GolayCode(GF(2))
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 GF(3)
sage: G = C.permutation_automorphism_group(algorithm="partition")
Permutation Group with generators [(1,3,4)]
sage: GG = C.permutation_automorphism_group(algorithm="codecan")
# long time
sage: GG == G
True
sage: C = codes.GolayCode(GF(3), True)
sage: G = C.permutation_automorphism_group(algorithm="gap")
# optional - gap_packages (Guava package)
Permutation Group with generators [(1,3)(4,5), (1,4)(3,5)]
sage: G = C.permutation_automorphism_group(algorithm="gap")
# optional - gap_packages (Guava package)
Permutation Group with generators [(5,7)(6,11)(8,9)(10,12), (4,6,11)(5,8,12)(7,10,9), (3,4)(6,8)(9,11)(10,12), (2,3)(6,11)(8,12)(9,10), (1,2)(5,10)(7,12)(8,9)]
```
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]
```

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

**permutated_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
[7, 4] linear code over GF(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])
Puncturing of [7, 4] Hamming Code over GF(2) 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, a, 0, 0, 0, 0, a + 1, 0, 0, 0, 0, 0, 0, 0, 0, a + 1, a + 1, 1, 0, 0)
```

Passes extra positional or keyword arguments through:

```
sage: C.random_element(probs=.5, distribution='1/n')  # random test
(1, 0, a, 0, 0, 0, a + 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, a + 1, a + 1, 1, 0, 0)
```
\textbf{rate()} \\
Return the ratio of the number of information symbols to the code length.

EXAMPLES:

\begin{verbatim}
sage: C = codes.HammingCode(GF(2), 3)
sage: C.rate()
4/7
\end{verbatim}

\textbf{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 \texttt{AbstractLinearCode.systematic_generator_matrix()}.

OUTPUT:

\begin{itemize}
  \item An \(k \times (n - k)\) matrix.
\end{itemize}

EXAMPLES:

\begin{verbatim}
sage: C = codes.HammingCode(GF(2), 3)
sage: C.generator_matrix()
\[
\begin{pmatrix}
1 & 0 & 0 & 0 & 0 & 1 & 1 \\
0 & 1 & 0 & 0 & 1 & 0 & 1 \\
0 & 0 & 1 & 0 & 1 & 1 & 0 \\
0 & 0 & 0 & 1 & 1 & 1 & 1
\end{pmatrix}
\]
sage: C.redundancy_matrix()
\[
\begin{pmatrix}
0 & 1 & 1 \\
1 & 0 & 1 \\
1 & 1 & 0 \\
1 & 1 & 1
\end{pmatrix}
\]
sage: C = LinearCode(matrix(GF(3),2,[1,2,0,
3,1,1]))
sage: C.systematic_generator_matrix()
\[
\begin{pmatrix}
1 & 2 & 0 \\
0 & 0 & 1
\end{pmatrix}
\]
sage: C.redundancy_matrix()
\[
\begin{pmatrix}
2 \\
0
\end{pmatrix}
\]
\end{verbatim}

\textbf{relative_distance()} \\
Return the ratio of the minimum distance to the code length.

EXAMPLES:

\begin{verbatim}
sage: C = codes.HammingCode(GF(2), 3)
sage: C.relative_distance()
3/7
\end{verbatim}

\textbf{shortened}(L) \\
Returns the code shortened at the positions \(L\), where \(L \subset \{1, 2, \ldots, n\}\).

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 [HP2003], $C_L$ is $((C^L)^L)^L$. 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:**

```python
sage: C = codes.HammingCode(GF(2), 3)
sage: C.shortened([1,2])
[5, 2] linear code over GF(2)
```

```python
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:**

```python
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 GF(4)
sage: C.weight_distribution()
[1, 0, 0, 30, 15, 18]
sage: C = codes.HammingCode(GF(2), 3); C
[7, 4] Hamming Code over GF(2)
sage: C.weight_distribution(algorithm="leon")  # optional - gap_packages
[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")
```
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 1 1 0]
[0 1 0 1 0 1 0]
[0 0 1 0 0 0 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],
    [0,0,1,0,0,0,1]]))
sage: Cs, p = C.standard_form()
sage: p
[1, 2, 7, 3, 4, 5, 6]
sage: Cs.generator_matrix()
[1 0 0 0 1 1]
[0 1 0 1 0 1]
[0 0 1 0 0 0]
sage: C = codes.HammingCode(GF(3), 3); C
[13, 10] Hamming Code over GF(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 GF(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 GF(7)
sage: C.weight_distribution() == C.weight_distribution(algorithm="leon") # optional - gap_packages (Guava package)
True
```
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], [1,1,0,1,0,1,0, \rightarrow 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:

```python
sage: G = matrix(GF(3), [[1, 2, 1, 0], [2, 1, 1, 1]])
```

```python
sage: C = LinearCode(G)
sage: C.generator_matrix()
[1 2 1 0]
[2 1 1 1]
```

```python
sage: C.systematic_generator_matrix()
[1 2 0 1]
[0 0 1 2]
```

Specific systematic positions can also be requested:

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

```python
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 \(c\). 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:

```python
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]])
```

```python
sage: C = LinearCode(G)
sage: c = vector(GF(2), (1, 1, 0, 0, 1, 1, 0))
sage: C.unencode(c)
(0, 1, 1, 0)
```

```python
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:**

```python
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 GF(4)
sage: C.weight_distribution()
[1, 0, 0, 30, 15, 18]
sage: C = codes.HammingCode(GF(2), 3); C
[7, 4] Hamming Code over GF(2)
sage: C.weight_distribution(algorithm="leon")  # optional - gap_packages
[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 GF(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 GF(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 GF(7)
sage: C.weight_distribution() == C.weight_distribution(algorithm="leon")  # optional - gap_packages (Guava package)
True
```

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

Return the weight enumerator polynomial of self.

This is the bivariate, homogeneous polynomial in \(x\) and \(y\) whose coefficient to \(x^iy^{n-i}\) 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.
• 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 \( \mathbb{Z} \).

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: 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
```

An example of a code with a non-symmetrical weight enumerator:

```
sage: C = codes.GolayCode(GF(3), extended=False)
sage: C.weight_enumerator()
24*x^11 + 110*x^9*y^2 + 330*x^8*y^3 + 132*x^6*y^5 + 132*x^5*y^6 + y^11
```

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.weight_enumerator() # indirect doctest
(0, 0, 0, 0, 0, 0)
sage: C.zero() # 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 \( \mathbb{Q}(T) \)

EXAMPLES:

```
sage: C = codes.HammingCode(GF(2), 3)
sage: C.zeta_function()
(1/5*T^2 + 1/5*T + 1/10)/(T^2 - 3/2*T + 1/2)
```

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 $\mathbb{Q}$

EXAMPLES:

```python
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:

```python
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],
         [1,0,0,0,0,0,0]]
)  
sage: C = LinearCode(G)

[7, 4] linear code over GF(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]

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],
         [1,0,0,0,0,0,0]]
)  
sage: C = LinearCode(G)

[7, 4] linear code over GF(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)  
[23, 12] linear code over GF(23)

Another example:

sage: C = codes.HammingCode(GF(7), 3)
sage: LinearCode(C)  
[57, 54] Hamming Code over GF(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:

```python
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]
```

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:

```python
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,\rightarrow 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:

```python
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,\rightarrow 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, ˓→0,1,0,1,0]]
)sage: C = LinearCode(G)
sage: D = codes.decoders.LinearCodeNearestNeighborDecoder(C)
sage: D.decoding_radius()
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 
  \(\text{maximum_error_weight}\).

• 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.

\(\text{maximum_error_weight}\) need never exceed the covering radius of the code, since there are then always 
lower-weight errors with the same syndrome. If one sets \(\text{maximum_error_weight}\) 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 \(\text{decoding_radius}\) after construction.

If \(\text{maximum_error_weight}\) 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.

Note: 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 \(\text{maximum_error_weight}\) is kept unspecified, it will be set to \(n-k\), where \(n\) is 
the length of \(\text{code}\) and \(k\) its dimension.

EXAMPLES:
\begin{verbatim}
sage: G = Matrix(GF(3), [[1,0,0,1,0,1,0,1,2],[0,1,0,2,2,0,1,1,0],[0,0,1,0,2,2,1,1,0]])
sage: C = LinearCode(G)
sage: D = codes.decoders.LinearCodeSyndromeDecoder(C)
sage: D
Syndrome decoder for [9, 3] linear code over GF(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:

\begin{verbatim}
sage: D = codes.decoders.LinearCodeSyndromeDecoder(C, maximum_error_weight=2)
sage: D
Syndrome decoder for [9, 3] linear code over GF(3) handling errors of weight up to 2
\end{verbatim}

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):

\begin{verbatim}
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)
\end{verbatim}

In the following examples, we illustrate how the choice of \texttt{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 \texttt{maximum_error_weight} smaller than both the covering radius and half the minimum distance:

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

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, \texttt{maximum_error_weight} is chosen to be bigger than half the minimum distance, but lower than the covering radius:

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

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 \texttt{maximum_error_weight} is bigger than both the covering radius and half the minimum distance:
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.

\section*{decode_to_code \((r)\)}

Corrects the errors in \( r \) and returns a codeword.

**INPUT:**

- \( r \) - a codeword of \( self \)

**OUTPUT:**

- a vector of \( self \)'s message space

**EXAMPLES:**

\begin{verbatim}
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
\end{verbatim}

\section*{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:**

\begin{verbatim}
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,\rightarrow-2,1,2]])
sage: C = LinearCode(G)
sage: D = codes.decoders.LinearCodeSyndromeDecoder(C)
sage: D.decoding_radius()
4
\end{verbatim}

\section*{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:**

\section*{1.1. Generic structures for linear codes}
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()
Return the syndrome lookup table of self.

EXAMPLES:

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]])
sage: C = LinearCode(G)
sage: D = codes.decoders.LinearCodeSyndromeDecoder(C)
sage: D.syndrome_table()
{(0, 0, 0): (0, 0, 0, 0, 0, 0, 0, 0),
 (0, 0, 1): (0, 0, 0, 1, 0, 0, 0, 0),
 (0, 1, 0): (0, 1, 0, 0, 0, 0, 0, 0),
 (0, 1, 1): (0, 0, 0, 0, 0, 1, 0, 0),
 (1, 0, 0): (1, 0, 0, 0, 0, 0, 0, 0),
 (1, 0, 1): (0, 0, 0, 0, 1, 0, 0, 0),
 (1, 1, 0): (0, 0, 1, 0, 0, 0, 0, 0),
 (1, 1, 1): (0, 0, 0, 0, 0, 0, 0, 1) }

class sage.coding.linear_code.LinearCodeSystematicEncoder (code,
 systemic_positions=None)
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 $0$th symbol of a message will then be at position systematic_positions[0],
the $1$st 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]])
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 [%d, 1] Repetition Code over GF(%s)" % (self.length(), self.base_field().cardinality())
    ....:
    sage: DualRepetitionCode(GF(3), 5).generator_matrix()
    \[
    \begin{bmatrix}
    1 & 0 & 0 & 0 & 2 \\
    0 & 1 & 0 & 2 & 1 \\
    0 & 0 & 1 & 2 & 0 \\
    \end{bmatrix}
    \]
```

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

```
sage: class BadCodeFamily(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: BadCodeFamily(GF(3), 5).generator_matrix()
```

(continues on next page)
generator_matrix()

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

Systematic form here means that a subset 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
sage: G = Matrix(GF(2), [[1,1,0,0,0],
                    [1,0,1,1,0],
                    [0,1,0,1,0],
                    [1,1,1,0,1]])

sage: C = LinearCode(G)

sage: E = codes.encoders.LinearCodeSystematicEncoder(C)

sage: E.generator_matrix()
[1 0 0 0 0 0 1]
[0 1 0 0 1 0 0]
[0 0 1 0 1 1 1]
[0 0 1 1 1 1 1]
```

We can ask for different systematic positions:

```sage
sage: E2 = codes.encoders.LinearCodeSystematicEncoder(C, systematic_positions=[5,4,3,2])

sage: E2.generator_matrix()
[1 0 0 0 0 0 1]
[0 1 0 0 1 0 0]
[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
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 0 1]
[0 0 1 0 0 1 0]
[0 0 0 1 0 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), [[1,0,0,0,1,0,0], [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)
1.2 Base class for Channels and commonly used 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:

  ```python
  Chan.transmit(msg)
  ```

  It can also be written in a more convenient way:

  ```python
  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:

  ```python
  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

```python
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

```python
input_space()
Returns the input space of self.
EXAMPLES:
```
sage: n_err = 2
sage: Chan = channels.StaticErrorRateChannel(GF(59)^6, n_err)

output_space()

Returns the output space of self.

EXAMPLES:

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:

We can check that the input msg is not modified:

If we transmit a vector which is not in the input space of self:

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 \(\text{"self"}\).

This is an abstract method which should be reimplemented in all the subclasses of Channel.

class sage.coding.channel_constructions.ErrorErasureChannel\((space, \text{number_errors}, \text{number_erasures})\)

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
• \(\text{number_errors}\) – 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.
• \(\text{number_erasures}\) – 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:

\[
\text{sage}: \text{n\_err, n\_era} = 2, 2 \\
\text{sage}: \text{Chan} = \text{channels.ErrorErasureChannel}(	ext{GF(59)}^40, \text{n\_err, n\_era}) \\
\text{sage}: \text{Chan} \\
\text{Error-and-erasure channel creating 2 errors and 2 erasures} \\
\text{of input space Vector space of dimension 40 over Finite Field of size 59} \\
\text{and output space The Cartesian product of (Vector space of dimension 40 over Finite Field of size 59, Vector space of dimension 40 over Finite Field of size 2)}
\]

We can also pass the number of errors and erasures as a couple of integers:

\[
\text{sage}: \text{n\_err, n\_era} = (1, 10), (1, 10) \\
\text{sage}: \text{Chan} = \text{channels.ErrorErasureChannel}(	ext{GF(59)}^40, \text{n\_err, n\_era}) \\
\text{sage}: \text{Chan} \\
\text{Error-and-erasure channel creating between 1 and 10 errors and between 1 and 10 erasures of input space Vector space of dimension 40 over Finite Field of size 59 and output space The Cartesian product of (Vector space of dimension 40 over Finite Field of size 59, Vector space of dimension 40 over Finite Field of size 2)}
\]

number_erasures\()

Returns the number of erasures created by \text{self}.

EXAMPLES:

\[
\text{sage}: \text{n\_err, n\_era} = 0, 3 \\
\text{sage}: \text{Chan} = \text{channels.ErrorErasureChannel}(	ext{GF(59)}^6, \text{n\_err, n\_era}) \\
\text{sage}: \text{Chan.number_erasures()} \\
(3, 3)
\]
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 \( \Sigma \) with \( |\Sigma| = q \) and an error probability \( \epsilon \), a q-ary symmetric channel sends an element of \( \Sigma \) 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 \( \Sigma^n \), and “transmits” each element of the vector independently in the above manner.

    Though \( \Sigma \) is usually taken to be a finite field, this implementation allows any structure for which Sage can represent \( \Sigma^n \) and for which \( \Sigma \) has a random_element() method. However, beware that if \( \Sigma \) 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: \( \Sigma^n \).
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**

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:**

```python
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)
```

```python
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:

```python
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:

```python
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:**

```python
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:

```python
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:

```python
sage: V = VectorSpace(GF(2), 1000)
sage: Chan = channels.StaticErrorRateChannel(V, 367)
sage: c = V.random_element()
sage: (c - Chan(c)).hamming_weight()
367
```

\[ \text{sage.coding.channel_constructions. format_interval}(t) \]

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

\[ \text{sage.coding.channel_constructions.random_error_vector}(n, F, error_positions) \]

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)
```

1.3 Base class for Decoders

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

```python
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.

```python
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()
[7, 4] linear code over GF(2)
```

class decoder()
Returns the connected encoder of self.

EXAMPLES:
sage: G = Matrix(GF(2),=[[1,1,0,0,0,0,0],[1,0,1,0,1,0,0],[0,1,0,1,0,1,0],[1,1,0,0,1,0,1],[0,0,0,0,0,0,0]])
sage: C = LinearCode(G)
sage: D = C.decoder()
sage: D.connected_encoder()
Generator matrix-based encoder for [7, 4] linear code over GF(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),=[[1,1,0,0,0,0,0],[1,0,1,0,1,0,0],[0,1,0,1,0,1,0],[1,1,0,0,1,0,1],[0,0,0,0,0,0,0]])
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)`
Decode `r` to the message space of `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),=[[1,1,0,0,0,0,0],[1,0,1,0,1,0,0],[0,1,0,1,0,1,0],[1,1,0,0,1,0,1],[0,0,0,0,0,0,0]])
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.

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

The types of a decoder are a set of labels commonly associated with decoders which describe the nature
and behaviour of the decoding algorithm. It should be considered as an informal descriptor but can be
coarsely relied upon for e.g. program logic.

The following are the most common types and a brief definition:

<table>
<thead>
<tr>
<th>Decoder type</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>always-succeed</td>
<td>The decoder always returns a closest codeword if the number of errors is up to the decoding radius.</td>
</tr>
<tr>
<td>bounded-distance</td>
<td>Any vector with Hamming distance at most (\text{decoding_radius}()) to a codeword is decodable to some codeword. If might-fail is also a type, then this is not a guarantee but an expectancy.</td>
</tr>
<tr>
<td>complete</td>
<td>The decoder decodes every word in the ambient space of the code.</td>
</tr>
<tr>
<td>dynamic</td>
<td>Some of the decoder’s types will only be determined at construction time (depends on the parameters).</td>
</tr>
<tr>
<td>half-minimum-distance</td>
<td>The decoder corrects up to half the minimum distance, or a specific lower bound thereof.</td>
</tr>
<tr>
<td>hard-decision</td>
<td>The decoder uses no information on which positions are more likely to be in error or not.</td>
</tr>
<tr>
<td>list-decoder</td>
<td>The decoder outputs a list of likely codewords, instead of just a single codeword.</td>
</tr>
<tr>
<td>might-fail</td>
<td>The decoder can fail at decoding even within its usual promises, e.g. bounded distance.</td>
</tr>
<tr>
<td>not-always-closest</td>
<td>The decoder does not guarantee to always return a closest codeword.</td>
</tr>
<tr>
<td>probabilistic</td>
<td>The decoder has internal randomness which can affect running time and the decoding result.</td>
</tr>
<tr>
<td>soft-decision</td>
<td>As part of the input, the decoder takes reliability information on which positions are more likely to be in error. Such a decoder only works for specific channels.</td>
</tr>
</tbody>
</table>

EXAMPLES:

We call it on a class:

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

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'}
```

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:

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

`input_space()`

Returns the input space of `self`.

EXAMPLES:

```python
sage: G = Matrix(GF(2), [[1,1,0,0,0,0,0],[1,0,1,1,0,0,0],[0,1,0,1,1,0,0],[1,1,0,1,0,1,0],[1,1,0,0,0,1,0]])
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:

```python
sage: G = Matrix(GF(2), [[1,1,0,0,0,0,0],[1,0,1,1,0,0,0],[0,1,0,1,1,0,0],[1,1,0,1,0,1,0],[1,1,0,0,0,1,0]])
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.

### 1.4 Base class for Encoders

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

```python
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:

• [Nie]

code()

Returns the code for this `Encoder`.

EXAMPLES:

```sage
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,-0,1,0,1,0]])
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.

One can use the following shortcut to encode a word with an encoder E:

```
E(word)
```

**INPUT:**

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

**OUTPUT:**

• a vector of `code()`.

**EXAMPLES:**

```sage
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,-0,1,0,1,0]])
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:
```python
sage: word = random_vector(GF(7), 4)
sage: E.encode(word)
Traceback (most recent call last):
  ...  
ArithmeticError: reduction modulo 2 not defined
```

### 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:**

```python
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:**

```python
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)

Return the message corresponding to the codeword `c`.

This is the inverse of `encode()`.

**INPUT:**

- `c` – a codeword of `code()`.
- `nocheck=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:**

```python
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)
```
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 [Nie]

INPUT:

- \( c \) – a codeword of \( \text{code()} \).

OUTPUT:

- an element of the message space of \( \text{self} \).

EXAMPLES:

\[
\begin{align*}
sage: & G = \text{Matrix}(\text{GF}(2), \begin{bmatrix} 1,1,1,0,0,0,0,0,1,0,0,1,0,0,1,1,1,0,0,1,0,1,0,0,1 \end{bmatrix}) \\
sage: & C = \text{LinearCode}(G) \\
sage: & c = \text{vector}(\text{GF}(2), \begin{bmatrix} 1,1,0,0,1,1,0 \end{bmatrix}) \\
sage: & c \in C \\
& \text{True} \\
sage: & E = \text{codes.encoders.LinearCodeGeneratorMatrixEncoder}(C) \\
sage: & E.\text{unencode}(c) \\
& (0,1,1,0)
\end{align*}
\]

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

\[
\begin{align*}
sage: & c = \text{vector}(\text{GF}(2), \begin{bmatrix} 1,1,0,0,1,1,1 \end{bmatrix}) \\
sage: & c \in C \\
& \text{False} \\
sage: & E = \text{codes.encoders.LinearCodeGeneratorMatrixEncoder}(C) \\
sage: & E.\text{unencode}_\text{nocheck}(c) \\
& (0,1,1,0) \\
sage: & m = \text{vector}(\text{GF}(2), \begin{bmatrix} 0,1,1,0 \end{bmatrix}) \\
sage: & c1 = E.\text{encode}(m) \\
sage: & c == c1 \\
& \text{False}
\end{align*}
\]

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

**Bases:** `exceptions.Exception`

Special exception class to indicate an error during encoding or unencoding.
2.1 Index of Channels: the information theoretic notion of transmission

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

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

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

```sage```
from sage.coding.channels_catalog import *
```sage```

2.2 Index of code constructions

The codes object may be used to access the codes that Sage can build.
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>DuadicCodeEvenPair()</code></td>
<td>Constructs the “even pair” of duadic codes associated to the “splitting” (see the docstring for <code>is_a_splitting</code> for the definition) S1, S2 of n.</td>
</tr>
<tr>
<td><code>DuadicCodeOddPair()</code></td>
<td>Constructs the “odd pair” of duadic codes associated to the “splitting” S1, S2 of n.</td>
</tr>
<tr>
<td><code>ExtendedQuadraticResidueCode()</code></td>
<td>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 [HP2003], Section 6.6.3-6.6.4.)</td>
</tr>
<tr>
<td><code>QuadraticResidueCode()</code></td>
<td>A quadratic residue code (or QR code) is a cyclic code whose generator polynomial is the product of the polynomials $x - \alpha^i$ ($\alpha$ is a primitive $n^{th}$ root of unity; $i$ ranges over the set of quadratic residues modulo $n$).</td>
</tr>
<tr>
<td><code>QuadraticResidueCodeEvenPair()</code></td>
<td>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 $n &gt; 2$ is prime then (Theorem 6.6.2 in [HP2003]) a QR code exists over $GF(q)$ iff q is a quadratic residue mod n.</td>
</tr>
<tr>
<td><code>QuadraticResidueCodeOddPair()</code></td>
<td>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 [HP2003]) a QR code exists over $GF(q)$ iff q is a quadratic residue mod n.</td>
</tr>
<tr>
<td><code>QuasiQuadraticResidueCode()</code></td>
<td>A (binary) quasi-quadratic residue code (or QQR code).</td>
</tr>
<tr>
<td><code>RandomLinearCodeGuava()</code></td>
<td>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 $S_n$.</td>
</tr>
<tr>
<td><code>ReedMullerCode()</code></td>
<td>Returns a Reed-Muller code.</td>
</tr>
<tr>
<td><code>ReedSolomonCode()</code></td>
<td>Construct a classical Reed-Solomon code.</td>
</tr>
<tr>
<td><code>ToricCode()</code></td>
<td>Let $P$ denote a list of lattice points in $\mathbb{Z}^d$ and let $T$ denote the set of all points in $(F^*)^d$ (ordered in some fixed way). Put $n =</td>
</tr>
<tr>
<td><code>WalshCode()</code></td>
<td>Return the binary Walsh code of length $2^m$.</td>
</tr>
<tr>
<td><code>from_parity_check_matrix()</code></td>
<td>Return the linear code that has $H$ as a parity check matrix.</td>
</tr>
<tr>
<td><code>random_linear_code()</code></td>
<td>Generate a random linear code of length length, dimension dimension and over the field $F$.</td>
</tr>
</tbody>
</table>

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

```python
sage: from sage.coding.codes_catalog import *
```

## 2.3 Index of decoders

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

It is usually not necessary to access these directly: rather, the `decoder` method directly on a code allows you to construct all compatible decoders for that code (`sage.coding.linear_code(AbstractLinearCode.decoder())`).

**Extended code decoders**

- `extended_code.ExtendedCodeOriginalCodeDecoder`
Subfield subcode decoder - subfield_subcode.SubfieldSubcodeOriginalCodeDecoder

Generalized Reed-Solomon code decoders
  • grs.GRSBerlekampWelchDecoder
  • grs.GRSErrorErasureDecoder
  • grs.GRSGaoDecoder
  • guruswami_sudan.gs_decoder.GRSGuruswamiSudanDecoder
  • grs.GRSGlobalSyndromeDecoder

Generic decoders
  • linear_code.LinearCodeNearestNeighborDecoder
  • linear_code.LinearCodeSyndromeDecoder
  • information_set_decoder.LinearCodeInformationSetDecoder

Cyclic code decoder
  • cyclic_code.CyclicCodeSurroundingBCHDecoder

BCH code decoder
  • bch.BCHUnderlyingGRSDecoder

Punctured codes decoders
  • punctured_code.PuncturedCodeOriginalCodeDecoder

---

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

```sage```
sage: from sage.coding.decoders_catalog import *
```sage```

---

2.4 Index of encoders

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

Cyclic code encoders
  • cyclic_code.CyclicCodePolynomialEncoder
  • cyclic_code.CyclicCodeVectorEncoder

Extended code encoders
  • extended_code.ExtendedCodeExtendedMatrixEncoder

Generic encoders
  • linear_code.LinearCodeGeneratorMatrixEncoder
  • linear_code.LinearCodeSystematicEncoder

Generalized Reed-Solomon code encoders
  • grs.GRSEvaluationVectorEncoder
  • grs.GRSEvaluationPolynomialEncoder

Punctured codes encoders
• punctured_code.PuncturedCodePuncturedMatrixEncoder

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

```python
sage: from sage.coding.encoders_catalog import *
```

### 2.5 Index of bounds on the parameters of codes

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>codesize_upper_bound()</code></td>
<td>Returns an upper bound on the number of codewords in a (possibly non-linear) code.</td>
</tr>
<tr>
<td><code>delsarte_bound_additive_hamming_space()</code></td>
<td>Find a modified Delsarte bound on additive codes in Hamming space $\mathbb{H}_{q^n}$ of minimal distance $d$.</td>
</tr>
<tr>
<td><code>delsarte_bound_hamming_space()</code></td>
<td>Find the Delsarte bound [De1973] on codes in Hamming space $\mathbb{H}_{q^n}$ of minimal distance $d$.</td>
</tr>
<tr>
<td><code>dimension_upper_bound()</code></td>
<td>Returns an upper bound for the dimension of a linear code.</td>
</tr>
<tr>
<td><code>elias_bound_asymptotic()</code></td>
<td>The asymptotic Elias bound for the information rate.</td>
</tr>
<tr>
<td><code>elias_upper_bound()</code></td>
<td>Returns the Elias upper bound.</td>
</tr>
<tr>
<td><code>entropy()</code></td>
<td>Computes the entropy at $x$ on the $q$-ary symmetric channel.</td>
</tr>
<tr>
<td><code>gilbert_lower_bound()</code></td>
<td>Returns the Gilbert-Varshamov lower bound.</td>
</tr>
<tr>
<td><code>griesmer_upper_bound()</code></td>
<td>Returns the Griesmer upper bound.</td>
</tr>
<tr>
<td><code>gv_bound_asymptotic()</code></td>
<td>The asymptotic Gilbert-Varshamov bound for the information rate, $R$.</td>
</tr>
<tr>
<td><code>gv_info_rate()</code></td>
<td>The Gilbert-Varshamov lower bound for information rate.</td>
</tr>
<tr>
<td><code>hamming_bound_asymptotic()</code></td>
<td>The asymptotic Hamming Bound for the information rate.</td>
</tr>
<tr>
<td><code>hamming_upper_bound()</code></td>
<td>Returns the Hamming upper bound.</td>
</tr>
<tr>
<td><code>krawtchouk()</code></td>
<td>Compute $K^{n,q}_l(x)$, the Krawtchouk (a.k.a. Kravchuk) polynomial.</td>
</tr>
<tr>
<td><code>mrrw1_bound_asymptotic()</code></td>
<td>The first asymptotic McEliese-Rumsey-Rodemich-Welsh bound.</td>
</tr>
<tr>
<td><code>plotkin_bound_asymptotic()</code></td>
<td>The asymptotic Plotkin bound for the information rate.</td>
</tr>
<tr>
<td><code>plotkin_upper_bound()</code></td>
<td>Returns the Plotkin upper bound.</td>
</tr>
<tr>
<td><code>singleton_bound_asymptotic()</code></td>
<td>The asymptotic Singleton bound for the information rate.</td>
</tr>
<tr>
<td><code>singleton_upper_bound()</code></td>
<td>Returns the Singleton upper bound.</td>
</tr>
<tr>
<td><code>volume_hamming()</code></td>
<td>Returns the number of elements in a Hamming ball.</td>
</tr>
</tbody>
</table>

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

```python
sage: from sage.coding.bounds_catalog import *
```

### 2.6 Databases and accessors of online databases for coding theory

```python
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://www.codetables.de/](http://www.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:

```python
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:

```python
sage: codes.databases.best_linear_code_in_guava(10, 5, GF(2))  # long time;
   → # optional - gap_packages (Guava package)
[10, 5] linear code over GF(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 $n \leq 257$), $q = 3$ (when $n \leq 243$), $q = 4$ ($n \leq 256$). (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:

```python
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 "UUUVCode">,
      [ <Operation "DualCode">,
      [ <Operation "UUUVCode">,
        [ <Operation "DualCode">,
    [ 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:

```python
sage: for B in codes.databases.self_orthogonal_binary_codes(7, 3):
    print(B)
[2, 1] linear code over GF(2)
[4, 2] linear code over GF(2)
[6, 3] linear code over GF(2)
[4, 1] linear code over GF(2)
[6, 2] linear code over GF(2)
[6, 2] linear code over GF(2)
[7, 3] linear code over GF(2)
[6, 1] linear code over GF(2)
```

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

```python
sage: for B in codes.databases.self_orthogonal_binary_codes(7, 3, 4):
    print(B); print(B.generator_matrix())
[4, 1] linear code over GF(2)
[1 1 1 1]
[6, 2] linear code over GF(2)
[1 1 1 0 0]
[0 1 0 1 1]
[7, 3] linear code over GF(2)
[1 0 1 1 0]
[0 1 0 1 1]
[0 0 1 0 1]
```

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

```python
sage: for B in codes.databases.self_orthogonal_binary_codes(7, 2, 4):
    print(B); print(B.generator_matrix())
[4, 1] linear code over GF(2)
[1 1 1 1]
[6, 2] linear code over GF(2)
[1 1 1 0 0]
[0 1 0 1 1]
```

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

```python
sage: for B in codes.databases.self_orthogonal_binary_codes(8, 4, equal=True):
    ....:     print(B); print(B.generator_matrix())

[8, 4] linear code over GF(2)
[1 0 0 1 0 0 0 0]
[0 1 0 0 1 0 0 0]
[0 0 1 0 1 0 0 0]
[0 0 0 0 0 0 1 1]
[8, 4] linear code over GF(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(codes.databases.self_orthogonal_binary_codes(8, 4, 1, equal=True))
Traceback (most recent call last):
...
ValueError: b (1) must be a positive even integer.
```

### 2.7 Database of two-weight codes

This module stores a database of two-weight codes.
\[
\begin{array}{cccccc}
q = 2 & n = 68 & k = 8 & w_1 = 32 & w_2 = 40 & \text{Shared by Eric Chen [ChenDB].} \\
q = 2 & n = 85 & k = 8 & w_1 = 40 & w_2 = 48 & \text{Shared by Eric Chen [ChenDB].} \\
q = 2 & n = 70 & k = 9 & w_1 = 32 & w_2 = 40 & \text{Found by Axel Kohnert [Koh2007] and shared by Alfred Wassermann.} \\
q = 2 & n = 73 & k = 9 & w_1 = 32 & w_2 = 40 & \text{Shared by Eric Chen [ChenDB].} \\
q = 2 & n = 219 & k = 9 & w_1 = 96 & w_2 = 112 & \text{Shared by Eric Chen [ChenDB].} \\
q = 2 & n = 198 & k = 10 & w_1 = 96 & w_2 = 112 & \text{Shared by Eric Chen [ChenDB].} \\
q = 3 & n = 15 & k = 4 & w_1 = 9 & w_2 = 12 & \text{Shared by Eric Chen [ChenDB].} \\
q = 3 & n = 55 & k = 5 & w_1 = 36 & w_2 = 45 & \text{Shared by Eric Chen [ChenDB].} \\
q = 3 & n = 56 & k = 6 & w_1 = 36 & w_2 = 45 & \text{Shared by Eric Chen [ChenDB].} \\
q = 3 & n = 84 & k = 6 & w_1 = 54 & w_2 = 63 & \text{Shared by Eric Chen [ChenDB].} \\
q = 3 & n = 98 & k = 6 & w_1 = 63 & w_2 = 72 & \text{Shared by Eric Chen [ChenDB].} \\
q = 3 & n = 126 & k = 6 & w_1 = 81 & w_2 = 90 & \text{Shared by Eric Chen [ChenDB].} \\
q = 3 & n = 140 & k = 6 & w_1 = 90 & w_2 = 99 & \text{Found by Axel Kohnert [Koh2007] and shared by Alfred Wassermann.} \\
q = 3 & n = 154 & k = 6 & w_1 = 99 & w_2 = 108 & \text{Shared by Eric Chen [ChenDB].} \\
q = 3 & n = 168 & k = 6 & w_1 = 108 & w_2 = 117 & \text{From [Di2000]} \\
q = 4 & n = 34 & k = 4 & w_1 = 24 & w_2 = 28 & \text{Shared by Eric Chen [ChenDB].} \\
q = 4 & n = 121 & k = 5 & w_1 = 88 & w_2 = 96 & \text{From [Di2000]} \\
q = 4 & n = 132 & k = 5 & w_1 = 96 & w_2 = 104 & \text{From [Di2000]} \\
q = 4 & n = 143 & k = 5 & w_1 = 104 & w_2 = 112 & \text{From [Di2000]} \\
q = 5 & n = 39 & k = 4 & w_1 = 30 & w_2 = 35 & \text{From Bouyukliev and Simonis ([BS2003], Theorem 4.1)} \\
q = 5 & n = 52 & k = 4 & w_1 = 40 & w_2 = 45 & \text{Shared by Eric Chen [ChenDB].} \\
q = 5 & n = 65 & k = 4 & w_1 = 50 & w_2 = 55 & \text{Shared by Eric Chen [ChenDB].} \\
\end{array}
\]

REFERENCE:
- [BS2003]
- [ChenDB]
- [Koh2007]
- [Di2000]
CHAPTER
THREE

CODE CONSTRUCTIONS

The named code families below are represented in Sage by their own classes, allowing specialised implementations of e.g. decoding or computation of properties:

3.1 Reed-Solomon codes and Generalized Reed-Solomon codes

Given $n$ different evaluation points $\alpha_1, \ldots, \alpha_n$ from some finite field $F$, the corresponding Reed-Solomon code (RS code) of dimension $k$ is the set:

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

An RS code is often called “classical” if $alpha_i = \alpha^{i-1}$ and $\alpha$ is a primitive $n$‘th root of unity.

More generally, given also $n$ “column multipliers” $\beta_1, \ldots, \beta_n$, the corresponding Generalized Reed-Solomon code (GRS code) of dimension $k$ is the set:

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

Here is a list of all content related to GRS codes:

- `GeneralizedReedSolomonCode`, the class for GRS codes
- `ReedSolomonCode()`, function for constructing classical Reed-Solomon 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

```python
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 [HJ2004] pp. 51-52 for details.

    INPUT:
```
• code – a code associated to this decoder

EXAMPLES:

```python
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] Reed-Solomon Code over GF(59)
```

Actually, we can construct the decoder from C directly:

```python
sage: D = C.decoder("BerlekampWelch")
sage: D
Berlekamp-Welch decoder for [40, 12, 29] Reed-Solomon Code over GF(59)
```

decode_to_code(r)
Correct 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:**

```python
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.decode_to_code(y)
```

decode_to_message(r)
Decode 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:**

```python
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_message(y)
True
```
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

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.GRSBerlekampWelchDecoder(C)
 sage: D.decoding_radius()
14

class sage.coding.grs.GRSErrorErasureDecoder (code)

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

Let $C$ be a GRS code of length $n$ and dimension $k$. Considering $y$ a codeword with at most $t$ errors ($t$ being the $\lfloor \frac{d-1}{2} \rfloor$ decoding radius), and $e$ the erasure vector, this decoder works as follows:

• Puncture the erased coordinates which are identified in $e$.

• Create a new GRS code of length $n - w(e)$, where $w$ is the Hamming weight function, and dimension $k$.

• Use Gao decoder over this new code one the punctured word built on the first step.

• Recover the original message from the decoded word computed on the previous step.

• Encode this message using an encoder over $C$.

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] Reed-Solomon Code over GF(59)

Actually, we can construct the decoder from $C$ directly:
```python
sage: D = C.decoder("ErrorErasure")
```
```
sage: D
```
Error-Erasure decoder for [40, 12, 29] Reed-Solomon Code over GF(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 $F_2$ 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:**

```python
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

### 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:**

```python
sage: F = GF(59)
sage: n, k = 40, 12
sage: C = codes.GeneralizedReedSolomonCode(F.list()[:n], k)
sage: D = codes.decoders.GRSErrorErasureDecoder(C)
```
If we receive too many erasures, it returns an exception as codeword will be impossible to decode:

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

```python
class sage.coding.grs.GRSEvaluationPolynomialEncoder (code, polynomial_ring=None)
```

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

Let \( C \) be a GRS code of length \( n \) and dimension \( k \) over some finite field \( F \). We denote by \( \alpha_i \) its evaluations points and by \( \beta_i \) its column multipliers, where \( 1 \leq i \leq n \). Let \( p \) be a polynomial of degree at most \( k - 1 \) in \( F[x] \) be the message.

The encoding of \( m \) will be the following codeword:

\[
(\beta_1 \times p(\alpha_1), \ldots, \beta_n \times p(\alpha_n)).
\]

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:

```python
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] Reed-Solomon Code over GF(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:

```python
sage: E = C.encoder("EvaluationPolynomial")
sage: E
Evaluation polynomial-style encoder for [40, 12, 29] Reed-Solomon Code over GF(59)
```

We can also specify another polynomial ring:

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

```python
encode (p)
```

Transform the polynomial \( p \) into a codeword of \( \text{code}() \).

One can use the following shortcut to encode a word with an encoder \( E \):
E(word)

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

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

```sage
code: p = x^10
code: 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
code: Qy.<y> = QQ[]
code: p = y^2 + 1
code: 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
```

message_space()

Return the message space of self

EXAMPLES:

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

class polynomial_ring()

Return the message space of self

EXAMPLES:

```sage
code: F = GF(11)
code: n, k = 10, 5
code: C = codes.GeneralizedReedSolomonCode(F.list()[:n], k)
code: E = C.encoder("EvaluationPolynomial")
code: E.polynomial_ring()
Univariate Polynomial Ring in x over Finite Field of size 11
```
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)
Return 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:
• a 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
(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 that encodes vectors into codewords.

Let C be a GRS code of length n and dimension k over some finite field F. We denote by \( \alpha_i \) its evaluations points and by \( \beta_i \) its column multipliers, where \( 1 \leq i \leq n \). Let \( m = (m_1, \ldots, m_k) \), a vector over F, be the message. We build a polynomial using the coordinates of m as coefficients:

\[
p = \sum_{i=1}^{m} m_i \times x^i.
\]

The encoding of m will be the following codeword:

\[
(\beta_1 \times p(\alpha_1), \ldots, \beta_n \times p(\alpha_n)).
\]
• code – the associated code of this encoder

EXAMPLES:

```python
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] Reed-Solomon Code over GF(59)
```

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

```python
sage: E = C.encoder("EvaluationVector")
sage: E
Evaluation vector-style encoder for [40, 12, 29] Reed-Solomon Code over GF(59)
```

**generator_matrix()**

Return a generator matrix of `self`

Considering a GRS code of length `n`, dimension `k`, with evaluation points \( (\alpha_1, \ldots, \alpha_n) \) and column multipliers \( (\beta_1, \ldots, \beta_n) \), its generator matrix \( G \) is built using the following formula:

\[
G = [g_{i,j}], g_{i,j} = \beta_j \times \alpha_i^j.
\]

This matrix is a Vandermonde matrix.

EXAMPLES:

```python
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]
```

```python
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 [Ga02] for details.

**INPUT:**

• code – the associated code of this decoder

**EXAMPLES:**

```python
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] Reed-Solomon Code over GF(59)
```
Actually, we can construct the decoder from \( C \) directly:

```
sage: D = C.decoder("Gao")
sage: D
Gao decoder for [40, 12, 29] Reed-Solomon Code over GF(59)
```

**decode_to_code** \((r)\)
Correct 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: D.connected_encoder().unencode(c) == D.decode_to_code(y)
True
```

**decode_to_message** \((r)\)
Decode \( r \) 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
```
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.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 [Rot2006], 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.GRSGaoDecoder(C)
sage: D
Key equation decoder for [40, 12, 29] Reed-Solomon Code over GF(59)
```

Actually, we can construct the decoder from C directly:
```
sage: D = C.decoder("KeyEquationSyndrome")
sage: D
Key equation decoder for [40, 12, 29] Reed-Solomon Code over GF(59)
```

decode_to_code(r)
Correct 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:
```python
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
```

def decode_to_message(r)
    Decode 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:**

```python
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
```

def decoding_radius()
    Return maximal number of errors that self can decode

    **OUTPUT:**
    - the number of errors as an integer

    **EXAMPLES:**

```python
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.

3.1. Reed-Solomon codes and Generalized Reed-Solomon codes
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:

Often, one constructs a Reed-Solomon code by taking all non-zero elements of the field as evaluation points, and specifying no column multipliers (see also `ReedSolomonCode()` for constructing classical Reed-Solomon codes directly):

```python
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] Reed-Solomon Code over GF(7)
```

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

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

It is also possible to specify the column multipliers:

```python
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 GF(59)
```

column_multipliers()

Return the vector of column multipliers of `self`.

EXAMPLES:

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

covering_radius()

Return 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

decke_to_message(r)
Decodes \( r \) to an element in message space of \( \text{self} \).

**Note:** If the code associated to \( \text{self} \) 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 \( \text{self} \)

**OUTPUT:**
- a vector of \( \text{self} \) 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()
Return the dual code of \( \text{self} \), 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()[1:40], 12, colmults)
sage: Cd = C.dual_code(); Cd
[40, 28, 13] Generalized Reed-Solomon Code over GF(59)

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

sage: C == Cd.dual_code()
True

evaluation_points()
Return the vector of field elements used for the polynomial evaluations.

**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)

is_generalized()
Return whether \( \text{self} \) is a Generalized Reed-Solomon code or a regular Reed-Solomon code.
self is a Generalized Reed-Solomon code if its column multipliers are not all 1.

EXAMPLES:

```python
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)
sage: C.is_generalized()
False
sage: colmults = [1, 2, 3, 4, 5, 6, 7, 8, 9, 1]
sage: C2 = codes.GeneralizedReedSolomonCode(F.list()[:n], k, colmults)
sage: C2.is_generalized()
True
```

minimum_distance()  
Return the minimum distance between any two words in self.

Since a GRS code is always Maximum-Distance-Separable (MDS), this returns \( C.\text{length}() - C.\text{dimension}() + 1 \).

EXAMPLES:

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

multipliers_product()  
Return 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 are also the column multipliers of the parity check matrix of self.

EXAMPLES:

```python
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()  
Return the parity check matrix of self.

EXAMPLES:

```python
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()
Return the list of column multipliers of the parity check matrix of self. They are also column multipliers of the generator matrix for the dual GRS code of self.

EXAMPLES:

```python
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()
Return 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:

```python
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]
```

sage.coding.grs.ReedSolomonCode(base_field, length, dimension, primitive_root=None)
Construct a classical Reed-Solomon code.

A classical $[n,k]$ Reed-Solomon code over GF($q$) with $1 \leq k \leq n$ and $n|(q - 1)$ is a Reed-Solomon code whose evaluation points are the consecutive powers of a primitive $n$'th root of unity $\alpha$, i.e. $\alpha_i = \alpha^{i-1}$, where $\alpha_1,\ldots,\alpha_n$ are the evaluation points. A classical Reed-Solomon codes has all column multipliers equal 1.

Classical Reed-Solomon codes are cyclic, unlike most Generalized Reed-Solomon codes.

Use GeneralizedReedSolomonCode if you instead wish to construct non-classical Reed-Solomon and Generalized Reed-Solomon codes.

INPUT:

- base_field – the finite field for which to build the classical Reed-Solomon code.
- length – the length of the classical Reed-Solomon code. Must divide $q - 1$ where $q$ is the cardinality of base_field.
- dimension – the dimension of the resulting code.
- primitive_root – (default: None) a primitive $n$'th root of unity to use for constructing the classical Reed-Solomon code. If not supplied, one will be computed and can be recovered as C.evaluation_points()[1] where C is the code returned by this method.

EXAMPLES:

```python
sage: C = codes.ReedSolomonCode(GF(7), 6, 3); C
[6, 3, 4] Reed-Solomon Code over GF(7)
```

This code is cyclic as can be seen by coercing it into a cyclic code:

```python
sage: Ccyc = codes.CyclicCode(code=C); Ccyc
[6, 3] Cyclic Code over GF(7)
```

(continues on next page)
Another example over an extension field:

```
sage: C = codes.ReedSolomonCode(GF(64,'a'), 9, 4); C
[9, 4, 6] Reed-Solomon Code over GF(64)
```

The primitive \( n \)’th root of unity can be recovered as the 2nd evaluation point of the code:

```
sage: alpha = C.evaluation_points()[1]; alpha
a^5 + a^4 + a^2 + a
```

We can also supply a different primitive \( n \)’th root of unity:

```
sage: beta = alpha^2; beta
a^4 + a
sage: beta.multiplicative_order()
9
sage: D = codes.ReedSolomonCode(GF(64), 9, 4, primitive_root=beta); D
[9, 4, 6] Reed-Solomon Code over GF(64)
sage: C == D
False
```

### 3.2 Hamming Code

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

REFERENCES:

• [Rot2006]

```python
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 GF(7)
```

`minimum_distance()`

Return the minimum distance of `self`.

It is always 3 as `self` is a Hamming Code.

EXAMPLES:

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

**parity_check_matrix()**

Return 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 [Rot2006], pages 47-48
• the dedicated paragraph in [HP2003], page 30

**EXAMPLES:**

```python
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 1 1 1 1 1 1 2 2 2]
```

### 3.3 Golay code

Golay codes are a set of four specific codes (binary Golay code, extended binary Golay code, ternary Golay and extended ternary Golay code), known to have some very interesting properties: for example, binary and ternary Golay codes are perfect codes, while their extended versions are self-dual codes.

**REFERENCES:**

• [HP2003] pp. 31-33 for a definition of Golay codes.
• Wikipedia article Golay_code

```python
class sage.coding.golay_code.GolayCode(base_field, extended=True)
Bases: sage.coding.linear_code.AbstractLinearCode
Representation of a Golay Code.

INPUT:

• `base_field` – The base field over which the code is defined. Can only be GF(2) or GF(3).
• `extended` – (default: True) if set to True, creates an extended Golay code.

**EXAMPLES:**

```python
sage: codes.GolayCode(GF(2))
[24, 12, 8] Extended Golay code over GF(2)
```

Another example with the perfect binary Golay code:

```python
sage: codes.GolayCode(GF(2), False)
[23, 12, 7] Golay code over GF(2)
```

**covering_radius()**

Return the covering radius of `self`.

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

**3.3. Golay code**
The covering radii of all Golay codes are known, and are thus returned by this method without performing any computation.

**EXAMPLES:**

```python
sage: C = codes.GolayCode(GF(2))
sage: C.covering_radius()
4
sage: C = codes.GolayCode(GF(2),False)
4
sage: C = codes.GolayCode(GF(3))
sage: C.covering_radius()
3
sage: C = codes.GolayCode(GF(3),False)
sage: C.covering_radius()
2
```

dual_code()

Return the dual code of self.

If self is an extended Golay code, self is returned. Otherwise, it returns the output of `sage.coding.linear_code.AbstractLinearCode.dual_code()`

**EXAMPLES:**

```python
sage: C = codes.GolayCode(GF(2), extended=True)
sage: Cd = C.dual_code(); Cd
[24, 12, 8] Extended Golay code over GF(2)
sage: Cd == C
True
```

generator_matrix()

Return a generator matrix of self.

Generator matrices of all Golay codes are known, and are thus returned by this method without performing any computation.

**EXAMPLES:**

```python
sage: C = codes.GolayCode(GF(2), extended=True)
sage: C.generator_matrix()
```

minimum_distance()

Return the minimum distance of self.

```python
sage: C = codes.GolayCode(GF(2), extended=True)
sage: C.minimum_distance()
84
```
The minimum distance of Golay codes is already known, and is thus returned immediately without computing anything.

EXAMPLES:

```python
sage: C = codes.GolayCode(GF(2))
sage: C.minimum_distance()
8
```

**parity_check_matrix()**

Return 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 $C$.

Parity check matrices of all Golay codes are known, and are thus returned by this method without performing any computation.

EXAMPLES:

```python
sage: C = codes.GolayCode(GF(3), extended=False)
sage: C.parity_check_matrix()
[1 0 0 0 1 2 2 2 1 0]
[0 1 0 0 0 1 2 2 2 1]
[0 0 1 0 0 2 1 2 0 1 2]
[0 0 0 1 0 1 1 0 1 1 1]
[0 0 0 1 2 2 2 1 0 1]
```

**weight_distribution()**

Return the list whose $i$'th entry is the number of words of weight $i$ in self.

The weight distribution of all Golay codes are known, and are thus returned by this method without performing any computation MWS (67, 69)

EXAMPLES:

```python
sage: C = codes.GolayCode(GF(3))
sage: C.weight_distribution()
[1, 0, 0, 0, 0, 0, 264, 0, 0, 440, 0, 0, 24]
```

### 3.4 Parity-check code

A simple way of detecting up to one error is to use the device of adding a parity check to ensure that the sum of the digits in a transmitted word is even.

A parity-check code of dimension $k$ over $F_q$ is the set: \{(m_1, m_2, ..., m_k, -\sum_{i=1}^{k} m_i) \mid (m_1, m_2, ..., m_k) \in F_q^k\}

REFERENCE:

```python
class sage.coding.parity_check_code.ParityCheckCode(base_field=Finite Field of size 2, dimension=7)
```

Representation of a parity-check code.

INPUT:

- `base_field` – the base field over which self is defined.
- `dimension` – the dimension of self.
EXAMPLES:

```python
sage: C = codes.ParityCheckCode(GF(5), 7)
sage: C
[8, 7] parity-check code over GF(5)
```

**minimum_distance()**

Return the minimum distance of self.

It is always 2 as self is a parity-check code.

EXAMPLES:

```python
sage: C = codes.ParityCheckCode(GF(5), 7)
sage: C.minimum_distance()
2
```

class sage.coding.parity_check_code.ParityCheckCodeGeneratorMatrixEncoder(
    code)

Bases: sage.coding.linear_code.LinearCodeGeneratorMatrixEncoder

Encoder for parity-check codes which uses a generator matrix to obtain codewords.

**INPUT:**

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

**EXAMPLES:**

```python
sage: C = codes.ParityCheckCode(GF(5), 7)
sage: E = codes.encoders.ParityCheckCodeGeneratorMatrixEncoder(C)
sage: E
Generator matrix-based encoder for [8, 7] parity-check code over GF(5)
```

Actually, we can construct the encoder from C directly:

```python
sage: E = C.encoder("ParityCheckCodeGeneratorMatrixEncoder")
sage: E
Generator matrix-based encoder for [8, 7] parity-check code over GF(5)
```

class sage.coding.parity_check_code.ParityCheckCodeStraightforwardEncoder(
    code)

Bases: sage.coding.encoder.Encoder

Encoder for parity-check codes which computes the sum of message symbols and appends its opposite to the message to obtain codewords.

**INPUT:**

```python
```
• code – the associated code of this encoder.

**EXAMPLES:**

```python
sage: C = codes.ParityCheckCode(GF(5), 7)
sage: E = codes.encoders.ParityCheckCodeStraightforwardEncoder(C)
sage: E
Parity-check encoder for the [8, 7] parity-check code over GF(5)
```

Actually, we can construct the encoder from \(C\) directly:

```python
sage: E = C.encoder("ParityCheckCodeStraightforwardEncoder")
sage: E
Parity-check encoder for the [8, 7] parity-check code over GF(5)
```

**encode** *(message)*

Transform the vector *message* into a codeword of *code()*.  

**INPUT:**

• *message* – A *self.code().dimension()*-vector from the *message space* of *self*.

**OUTPUT:**

• A codeword in associated code of *self*.

**EXAMPLES:**

```python
sage: C = codes.ParityCheckCode(GF(5), 7)
sage: message = vector(C.base_field(), [1, 0, 4, 2, 0, 3, 2])
sage: C.encode(message)
(1, 0, 4, 2, 0, 3, 2, 3)
```

**message_space** ()

Return the message space of *self*.

**EXAMPLES:**

```python
sage: C = codes.ParityCheckCode(GF(5), 7)
sage: E = codes.encoders.ParityCheckCodeStraightforwardEncoder(C)
sage: E.message_space()
Vector space of dimension 7 over Finite Field of size 5
```

**unencode_nocheck** *(word)*

Return the message corresponding to the vector *word*.

Use this method with caution: it does not check if *word* belongs to the code.

**INPUT:**

• *word* – A *self.code().length()*-vector from the ambiant space of *self*.

**OUTPUT:**

• A vector corresponding to the *self.code().dimension()*-first symbols in *word*.

**EXAMPLES:**

```python
sage: C = codes.ParityCheckCode(GF(5), 7)
sage: word = vector(C.base_field(), [1, 0, 4, 2, 0, 3, 2, 3])
sage: E = codes.encoders.ParityCheckCodeStraightforwardEncoder(C)
sage: E.unencode_nocheck(word)
(1, 0, 4, 2, 0, 3, 2)
```
3.5 Reed-Muller code

Given integers $m, r$ and a finite field $F$, the corresponding Reed-Muller Code is the set:

$$\{(f(\alpha_i) | \alpha_i \in F^m) | f \in F[x_1, x_2, \ldots, x_m], \deg f \leq r\}$$

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 $r \leq m$
- `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)

```python
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 a binary Reed-Muller code, 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
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
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:**
```python
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:**

```python
sage: C = codes.BinaryReedMullerCode(2, 4)
sage: C.order()
2
```

```python
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 :meth:`ReedMullerCode`

**Note:** It is better to use the aforementioned method rather than calling this class directly, as :meth:`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:**

```python
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:**

```python
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:

```python
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:

```python
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:

```python
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:

```python
sage: C.length()
9
sage: C.dimension()
6
```
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 appropriate 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:

\[
(f(\alpha_{11}, \alpha_{12}, \ldots, \alpha_{1m}), f(\alpha_{21}, \alpha_{22}, \ldots, \alpha_{2m}), \ldots, f(\alpha_{q1}, \alpha_{q2}, \ldots, \alpha_{qm})), \text{ with } \alpha_{ij} = \beta_i \mod q \forall (i, j)
\]

**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 number of variables 2 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 number of variables over Finite Field of size 3
```

Actually, we can construct the encoder from \( C \) directly:

```
sage: E=codes.encoders.ReedMullerPolynomialEncoder(C)
sage: E
Evaluation polynomial-style encoder for Reed-Muller Code of order 2 and number of variables in Multivariate Polynomial Ring in x1, x2 over Finite Field of size 3
```
```python
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:**

```python
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:

```python
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:

```python
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:**

```python
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:

```python
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:

```python
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:

```python
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:

```python
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 + x1 + 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 \( \mathbb{F} \). 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 \( x_1 \times x_2 \), 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 \( \mathbb{F} \), which corresponds to the polynomial \( f = \sum_{i=1}^{k} v_i \times x_i \).

Let \( (\beta_1, \beta_2, \ldots, \beta_q) \) be the elements of \( \mathbb{F} \) ordered as they are returned by Sage when calling \( F\text{.list()} \).

The aforementioned polynomial \( f \) is encoded as:

\[
(f(\alpha_{11}, \alpha_{12}, \ldots, \alpha_{1m}), f(\alpha_{21}, \alpha_{22}, \ldots, \alpha_{2m}), \ldots, f(\alpha_{qm}, \alpha_{q2}, \ldots, \alpha_{qm}), \text{with } \alpha_{ij} = \beta_i \mod q \forall (i, j))
\]

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

(continues on next page)
3.6 Cyclic Code

Let $F$ be a field. A $[n, k]$ code $C$ over $F$ is called cyclic if every cyclic shift of a codeword is also a codeword [R06]:

$$\forall c \in C, c = (c_0, c_1, \ldots, c_{n-1}) \in C \Rightarrow (c_{n-1}, c_0, \ldots, c_{n-2}) \in C$$

Let $c = (c_0, c_1, \ldots, c_{n-1})$ be a codeword of $C$. This codeword can be seen as a polynomial over $F_2[x]$ as follows: $\sum_{i=0}^{n-1} c_i x^i$. There is a unique monic polynomial $g(x)$ such that for every $c(x) \in F_2[x]$ of degree less than $n - 1$, we have $c(x) \in C \iff g(x)|c(x)$. This polynomial is called the generator polynomial of $C$.

For now, only single-root cyclic codes (i.e. whose length $n$ and field order $q$ are coprimes) are implemented.

REFERENCES:

```python
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)]
```
• **field** – (default: None) the base field of self.

• **primitive_root** – (default: None) the primitive root of the splitting field which contains the roots of the generator polynomial. It has to be of multiplicative order length over this field. If the splitting field is not field, it also have to be a polynomial in \(zx\), where \(x\) is the degree of the extension over the prime field. For instance, over GF(16), it must be a polynomial in \(z^4\).

**EXAMPLES:**

We can construct a CyclicCode object using three different methods. First (1), we provide a generator polynomial and a code length:

```
sage: F.<x> = GF(2)[]
sage: n = 7
sage: g = x ** 3 + x + 1
sage: C = codes.CyclicCode(generator_pol = g, length = n)
sage: C
[7, 4] Cyclic Code over GF(2)
```

We can also provide a code (2). In that case, the program will try to extract a generator polynomial (see `find_generator_polynomial()` for details):

```
sage: C = codes.GeneralizedReedSolomonCode(GF(8, 'a').list()[1:], 4)
sage: Cc = codes.CyclicCode(code = C)
sage: Cc
[7, 4] Cyclic Code over GF(8)
```

Finally, we can give (a subset of) a defining set for the code (3). In this case, the generator polynomial will be computed:

```
sage: F = GF(16, 'a')
sage: n = 15
sage: D = [14,1,2,11,12]
sage: C = codes.CyclicCode(length = n, field = F, D = [1,2])
sage: C
[15, 13] Cyclic Code over GF(16)
```

**bch_bound**(arithmetic=False)

Returns the BCH bound of self which is a bound on self minimum distance.

See `sage.coding.cyclic_code.bch_bound()` for details.

**INPUT:**

• **arithmetic** – (default: False), if it is set to True, then it computes the BCH bound using the longest arithmetic sequence definition

**OUTPUT:**

• \((\text{delta} + 1, (l, c))\) – such that \(\text{delta} + 1\) is the BCH bound, and \(l, c\) are the parameters of the largest arithmetic sequence

**EXAMPLES:**

```
sage: F = GF(16, 'a')
sage: n = 15
sage: D = [14,1,2,11,12]
sage: C = codes.CyclicCode(field = F, length = n, D = D)
sage: C.bch_bound()
(3, (1, 1))
```
sage: F = GF(16, 'a')
sage: n = 15
sage: D = [14,1,2,11,12]
sage: C = codes.CyclicCode(field = F, length = n, D = D)
sage: C.bch_bound(True)
(4, (2, 12))

check_polynomial()

Returns the check polynomial of self.

Let $C$ be a cyclic code of length $n$ and $g$ its generator polynomial. The following: $h = \frac{x^n - 1}{g(x)}$ is called $C$’s check polynomial.

EXAMPLES:

sage: F.<x> = GF(2)[]
sage: n = 7
sage: g = x ** 3 + x + 1
sage: C = codes.CyclicCode(generator_pol = g, length = n)
sage: h = C.check_polynomial()
sage: h == (x**n - 1)/C.generator_polynomial()
True

defining_set (primitive_root=None)

Returns the set of exponents of the roots of self’s generator polynomial over the extension field. Of course, it depends on the choice of the primitive root of the splitting field.

INPUT:

• primitive_root (optional) – a primitive root of the extension field

EXAMPLES:

We provide a defining set at construction time:

sage: F = GF(16, 'a')
sage: n = 15
sage: C = codes.CyclicCode(length=n, field=F, D=[1,2])
sage: C.defining_set()
[1, 2]

If the defining set was provided by the user, it might have been expanded at construction time. In this case, the expanded defining set will be returned:

sage: C = codes.CyclicCode(length=13, field=F, D=[1, 2])
sage: C.defining_set()
[1, 2, 3, 5, 6, 9]

If a generator polynomial was passed at construction time, the defining set is computed using this polynomial:

sage: R.<x> = F[]
sage: n = 7
sage: g = x ** 3 + x + 1
sage: C = codes.CyclicCode(generator_pol=g, length=n)
sage: C.defining_set()
[1, 2, 4]

Both operations give the same result:
Another one, in a reversed order:

```
sage: n = 13
sage: C1 = codes.CyclicCode(length=n, field=F, D=[1, 2])
sage: g = C1.generator_polynomial()
sage: C2 = codes.CyclicCode(generator_pol=g, length=n)
sage: C1.defining_set() == C2.defining_set()
True
```

**field_embedding()**

Returns the base field embedding into the splitting field.

**EXAMPLES:**

```
sage: F.<x> = GF(2)[]
sage: n = 7
sage: g = x ** 3 + x + 1
sage: C = codes.CyclicCode(generator_pol=g, length=n)
sage: C.field_embedding()
Relative field extension between Finite Field in z3 of size 2^3 and Finite
   →Field of size 2
```

**generator_polynomial()**

Returns the generator polynomial of self.

**EXAMPLES:**

```
sage: F.<x> = GF(2)[]
sage: n = 7
sage: g = x ** 3 + x + 1
sage: C = codes.CyclicCode(generator_pol=g, length=n)
sage: C.generator_polynomial()
x^3 + x + 1
```

**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: F.<x> = GF(2)[]
sage: n = 7
sage: g = x ** 3 + x + 1
sage: C = codes.CyclicCode(generator_pol=g, length=n)
sage: C.parity_check_matrix()
[1 0 1 1 1 0 0]
[0 1 0 1 1 1 0]
[0 0 1 0 1 1 1]
```

**primitive_root()**

Returns the primitive root of the splitting field that is used to build the defining set of the code.

If it has not been specified by the user, it is set by default with the output of the `zeta` method of the splitting field.
EXAMPLES:

```python
sage: F.<x> = GF(2)[]
sage: n = 7
sage: g = x ** 3 + x + 1
sage: C = codes.CyclicCode(generator_pol = g, length=n)
sage: C.primitive_root()
z3
sage: F = GF(16, 'a')
sage: n = 15
sage: a = F.gen()
sage: Cc = codes.CyclicCode(length = n, field = F, D = [1,2], primitive_root = a^2 + 1)
sage: Cc.primitive_root()
a^2 + 1
```

`sage.surrounding_bch_code()`

Returns the surrounding BCH code of self.

EXAMPLES:

```python
sage: C = codes.CyclicCode(field=GF(2), length=63, D=[1, 7, 17])
sage: C.dimension() 45
sage: CC = C.surrounding_bch_code()
sage: CC
[63, 51] BCH Code over GF(2) with designed distance 3
sage: all(r in CC for r in C.generator_matrix())
True
```

class sage.coding.cyclic_code.CyclicCodePolynomialEncoder(code)

Bases: sage.coding.encoder.Encoder

An encoder encoding polynomials into codewords.

Let \( C \) be a cyclic code over some finite field \( F \), and let \( g \) be its generator polynomial.

This encoder encodes any polynomial \( p \in F[x]_{<k} \) by computing \( c = p \times g \) and returning the vector of its coefficients.

INPUT:

- `code` – The associated code of this encoder

EXAMPLES:

```python
sage: F.<x> = GF(2)[]
sage: n = 7
sage: g = x ** 3 + x + 1
sage: C = codes.CyclicCode(generator_pol = g, length=n)
sage: E = codes.encoders.CyclicCodePolynomialEncoder(C)
sage: E
Polynomial-style encoder for [7, 4] Cyclic Code over GF(2)
```

`encode(p)`

Transforms \( p \) into an element of the associated code of self.

INPUT:

- `p` – A polynomial from self message space
OUTPUT:
• A codeword in associated code of self

EXAMPLES:
```
sage: F.<x> = GF(2)[]
sage: n = 7
sage: g = x ** 3 + x + 1
sage: C = codes.CyclicCode(generator_pol = g, length = n)
sage: E = codes.encoders.CyclicCodePolynomialEncoder(C)
sage: m = x ** 2 + 1
sage: E.encode(m)
(1, 1, 1, 0, 0, 1, 0)
```

message_space()  
Returns the message space of self

EXAMPLES:
```
sage: F.<x> = GF(2)[]
sage: n = 7
sage: g = x ** 3 + x + 1
sage: C = codes.CyclicCode(generator_pol = g, length = n)
sage: E = codes.encoders.CyclicCodePolynomialEncoder(C)
sage: E.message_space()  
Univariate Polynomial Ring in x over Finite Field of size 2 (using GF2X)
```

unencode_nocheck(c)  
Returns the message corresponding to c. Does not check if c belongs to the code.

INPUT:
• c – A vector with the same length as the code

OUTPUT:
• An element of the message space

EXAMPLES:
```
sage: F.<x> = GF(2)[]
sage: n = 7
sage: g = x ** 3 + x + 1
sage: C = codes.CyclicCode(generator_pol = g, length = n)
sage: E = codes.encoders.CyclicCodePolynomialEncoder(C)
sage: c = vector(GF(2), (1, 1, 1, 0, 0, 1, 0))
sage: E.unencode_nocheck(c)
x^2 + 1
```

class sage.coding.cyclic_code.CyclicCodeSurroundingBCHDecoder(code, **kwargs)
Bases: sage.coding.decoder.Decoder

A decoder which decodes through the surrounding BCH code of the cyclic code.

INPUT:
• code – The associated code of this decoder.
• **kwargs – All extra arguments are forwarded to the BCH decoder

EXAMPLES:
```python
sage: C = codes.CyclicCode(field=GF(16), length=15, D=[14, 1, 2, 11, 12])
sage: D = codes.decoders.CyclicCodeSurroundingBCHDecoder(C)
sage: D
Decoder through the surrounding BCH code of the [15, 10] Cyclic Code over GF(16)
```

**bch_code()**

Returns the surrounding BCH code of `sage.coding.encoder.Encoder.code()`.

**EXAMPLES:**

```python
sage: C = codes.CyclicCode(field=GF(16), length=15, D=[14, 1, 2, 11, 12])
sage: D = codes.decoders.CyclicCodeSurroundingBCHDecoder(C)
sage: D.bch_code()
[15, 12] BCH Code over GF(16) with designed distance 4
```

**bch_decoder()**

Returns the decoder that will be used over the surrounding BCH code.

**EXAMPLES:**

```python
sage: C = codes.CyclicCode(field=GF(16), length=15, D=[14, 1, 2, 11, 12])
sage: D = codes.decoders.CyclicCodeSurroundingBCHDecoder(C)
sage: D.bch_decoder()
Decoder through the underlying GRS code of [15, 12] BCH Code over GF(16) with designed distance 4
```

**decode_to_code(y)**

Decodes `r` to an element in `sage.coding.encoder.Encoder.code()`.

**EXAMPLES:**

```python
sage: F = GF(16, 'a')
sage: C = codes.CyclicCode(field=F, length=15, D=[14, 1, 2, 11, 12])
sage: a = F.gen()
sage: D = codes.decoders.CyclicCodeSurroundingBCHDecoder(C)
sage: y = vector(F, [0, a^3, a^3 + a^2 + a, 1, a^2 + 1, a^3 + a^2 + 1, a^3 + a^2 + a, a^3 + a^2 + a, a^2 + a, a^3 + a^2 + a, a^2 + a, a^3 + a^2 + a, a^2 + 1, a^3 + a, a^3 + a])
sage: D.decode_to_code(y) in C
True
```

**decoding_radius()**

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

**EXAMPLES:**

```python
sage: C = codes.CyclicCode(field=GF(16), length=15, D=[14, 1, 2, 11, 12])
sage: D = codes.decoders.CyclicCodeSurroundingBCHDecoder(C)
sage: D.decoding_radius()
1
```

**class** `sage.coding.cyclic_code.CyclicCodeVectorEncoder(code)`

**Bases:** `sage.coding.encoder.Encoder`

An encoder which can encode vectors into codewords.

Let \( C \) be a cyclic code over some finite field \( F \), and let \( g \) be its generator polynomial.

Let \( m = (m_1, m_2, \ldots, m_k) \) be a vector in \( F^k \). This codeword can be seen as a polynomial over \( F[x] \), as follows: \( P_m = \sum_{i=0}^{k-1} m_i x^i \).
To encode \(m\), this encoder does the following multiplication: \(P_m \times g\).

**INPUT:**
- `code` – The associated code of this encoder

**EXAMPLES:**

```python
sage: F.<x> = GF(2)[]
sage: n = 7
sage: g = x ** 3 + x + 1
sage: C = codes.CyclicCode(generator_pol = g, length = n)
sage: E = codes.encoders.CyclicCodeVectorEncoder(C)
sage: E
Vector-style encoder for [7, 4] Cyclic Code over GF(2)
```

**encode** \((m)\)

Transforms \(m\) into an element of the associated code of \(self\).

**INPUT:**
- \(m\) – an element from \(self\)’s message space

**OUTPUT:**
- A codeword in the associated code of \(self\)

**EXAMPLES:**

```python
sage: F.<x> = GF(2)[]
sage: n = 7
sage: g = x ** 3 + x + 1
sage: C = codes.CyclicCode(generator_pol = g, length = n)
sage: E = codes.encoders.CyclicCodeVectorEncoder(C)
sage: m = vector(GF(2), (1, 0, 1, 0))
sage: E.encode(m)
(1, 1, 1, 0, 0, 1, 0)
```

**generator_matrix** ()

Returns a generator matrix of \(self\)

**EXAMPLES:**

```python
sage: F.<x> = GF(2)[]
sage: n = 7
sage: g = x ** 3 + x + 1
sage: C = codes.CyclicCode(generator_pol = g, length = n)
sage: E = codes.encoders.CyclicCodeVectorEncoder(C)
sage: E.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]
```

**message_space** ()

Returns the message space of \(self\)

**EXAMPLES:**

```python
sage: F.<x> = GF(2)[]
sage: n = 7
```

(continues on next page)

(continued from previous page)

sage: g = x ** 3 + x + 1
sage: C = codes.CyclicCode(generator_pol = g, length = n)
sage: E = codes.encoders.CyclicCodeVectorEncoder(C)
sage: E.message_space()
Vector space of dimension 4 over Finite Field of size 2

unencode_nocheck(c)
Returns the message corresponding to c. Does not check if c belongs to the code.
INPUT:
• c – A vector with the same length as the code
OUTPUT:
• An element of the message space
EXAMPLES:

sage: F.<x> = GF(2)[]
sage: n = 7
sage: g = x ** 3 + x + 1
sage: C = codes.CyclicCode(generator_pol = g, length = n)
sage: E = codes.encoders.CyclicCodeVectorEncoder(C)
sage: c = vector(GF(2), (1, 1, 1, 0, 0, 1, 0))
sage: E.unencode_nocheck(c)
(1, 0, 1, 0)

sage.coding.cyclic_code.bch_bound(n, D, arithmetic=False)
Returns the BCH bound obtained for a cyclic code of length n and defining set D.

Consider a cyclic code C, with defining set D, length n, and minimum distance d. We have the following bound, called BCH bound, on d: \( d \geq \delta + 1 \), where \( \delta \) is the length of the longest arithmetic sequence (modulo n) of elements in D.

That is, if \( \exists c, \gcd(c, n) = 1 \) such that \( \{l, l + c, \ldots, l + (\delta - 1) \times c\} \subseteq D \), then \( d \geq \delta + 1 \) [1]

The BCH bound is often known in the particular case \( c = 1 \). The user can specify by setting arithmetic = False.

Note: As this is a specific use case of the BCH bound, it is not available in the global namespace. Call it by using sage.coding.cyclic_code.bch_bound. You can also load it into the global namespace by typing from sage.coding.cyclic_code import bch_bound.

INPUT:
• n – an integer
• D – a list of integers
• arithmetic – (default: False), if it is set to True, then it computes the BCH bound using the longest arithmetic sequence definition

OUTPUT:
• (delta + 1, (l, c)) – such that delta + 1 is the BCH bound, and l, c are the parameters of the longest arithmetic sequence (see below)

EXAMPLES:
sage: n = 15
sage: D = [14,1,2,11,12]
sage: sage.coding.cyclic_code.bch_bound(n, D)
(3, (1, 1))

sage: n = 15
sage: D = [14,1,2,11,12]
sage: sage.coding.cyclic_code.bch_bound(n, D, True)
(4, (2, 12))

sage.coding.cyclic_code.find_generator_polynomial(code, check=True)
Returns a possible generator polynomial for code.

If the code is cyclic, the generator polynomial is the gcd of all the polynomial forms of the codewords. Conversely, if this gcd exactly generates the code code, then code is cyclic.

If check is set to True, then it also checks that the code is indeed cyclic. Otherwise it doesn’t.

INPUT:
• code – a linear code
• check – whether the cyclicity should be checked

OUTPUT:
• the generator polynomial of code (if the code is cyclic).

EXAMPLES:

```python
sage: from sage.coding.cyclic_code import find_generator_polynomial
sage: C = codes.GeneralizedReedSolomonCode(GF(8, 'a').list()[1:], 4)
sage: find_generator_polynomial(C)
x^3 + (a^2 + 1)*x^2 + a*x + a^2 + 1
```

### 3.7 BCH Code

Let $F = GF(q)$ and $\Phi$ be the splitting field of $x^n - 1$ over $F$, with $n$ a positive integer. Let also $\alpha$ be an element of multiplicative order $n$ in $\Phi$. Finally, let $b, \delta, \ell$ be integers such that $0 \leq b \leq n$, $1 \leq \delta \leq n$ and $\alpha^\ell$ generates the multiplicative group $\Phi^\times$.

A BCH code over $F$ with designed distance $\delta$ is a cyclic code whose codewords $c(x) \in F[x]$ satisfy $c(\alpha^a) = 0$, for all integers $a$ in the arithmetic sequence $b, b + \ell, b + 2 \times \ell, \ldots, b + (\delta - 2) \times \ell$.

```python
class sage.coding.bch.BCHCode
```
Bases: `sage.coding.cyclic_code.CyclicCode`

Representation of a BCH code seen as a cyclic code.

INPUT:
• base_field – the base field for this code
• length – the length of the code
• designed_distance – the designed minimum distance of the code
• primitive_root – (default: None) the primitive root to use when creating the set of roots for the generating polynomial over the splitting field. It has to be of multiplicative order length over this field.
If the splitting field is not field, it also has to be a polynomial in \(zx\), where \(x\) is the degree of the extension field. For instance, over \(GF(16)\), it has to be a polynomial in \(z^4\).

- **offset** – (default: 1) the first element in the defining set
- **jump_size** – (default: 1) the jump size between two elements of the defining set. It must be coprime with the multiplicative order of \(\text{primitive_root}\).
- **b** – (default: 0) is exactly the same as \(\text{offset}\). It is only here for retro-compatibility purposes with the old signature of \(\text{codes.BCHCode}\) and will be removed soon.

**EXAMPLES:**

As explained above, BCH codes can be built through various parameters:

```python
sage: C = codes.BCHCode(GF(2), 15, 7, offset=1)
sage: C
[15, 5] BCH Code over GF(2) with designed distance 7
sage: C.generator_polynomial()
x^10 + x^8 + x^5 + x^4 + x^2 + x + 1
sage: C = codes.BCHCode(GF(2), 15, 4, offset=1, jump_size=8)
sage: C
[15, 7] BCH Code over GF(2) with designed distance 4
sage: C.generator_polynomial()
x^8 + x^7 + x^6 + x^4 + 1
```

BCH codes are cyclic, and can be interfaced into the CyclicCode class. The smallest GRS code which contains a given BCH code can also be computed, and these two codes may be equal:

```python
sage: C = codes.BCHCode(GF(16), 15, 7)
sage: R = C.bch_to_grs()
sage: codes.CyclicCode(code=R) == codes.CyclicCode(code=C)
True
```

The \(\delta = 15\), 1 cases (trivial codes) also work:

```python
sage: C = codes.BCHCode(GF(16), 15, 1)
sage: C.dimension()
15
sage: C.defining_set()
[]
sage: C.generator_polynomial()
1
sage: C = codes.BCHCode(GF(16), 15, 15)
sage: C.dimension()
1
```

**bch_to_grs()**

Returns the underlying GRS code from which self was derived.

**EXAMPLES:**

```python
sage: C = codes.BCHCode(GF(2), 15, 3)
sage: RS = C.bch_to_grs()
sage: RS
[15, 13, 3] Reed-Solomon Code over GF(16)
sage: C.generator_matrix() * RS.parity_check_matrix().transpose() == 0
True
```
designed_distance()  
Returns the designed distance of self.

EXAMPLES:

```sage
C = codes.BCHCode(GF(2), 15, 4)
sage: C.designed_distance()
4
```

jump_size()  
Returns the jump size between two consecutive elements of the defining set of self.

EXAMPLES:

```sage
C = codes.BCHCode(GF(2), 15, 4, jump_size = 2)
sage: C.jump_size()
2
```

offset()  
Returns the offset which was used to compute the elements in the defining set of self.

EXAMPLES:

```sage
C = codes.BCHCode(GF(2), 15, 4, offset = 1)
sage: C.offset()
1
```

class sage.coding.bch.BCHUnderlyingGRSDecoder (code, grs_decoder='KeyEquationSyndrome', **kwargs)

Bases: sage.coding.decoder.Decoder

A decoder which decodes through the underlying sage.coding.grs.GeneralizedReedSolomonCode code of the provided BCH code.

INPUT:

• code – The associated code of this decoder.
• grs_decoder – The string name of the decoder to use over the underlying GRS code
• **kwargs – All extra arguments are forwarded to the GRS decoder

bch_word_to_grs(c)  
Returns c converted as a codeword of grs_code().

EXAMPLES:

```sage
C = codes.BCHCode(GF(2), 15, 3)
sage: D = codes.decoders.BCHUnderlyingGRSDecoder(C)
sage: c = C.random_element()
sage: y = D.bch_word_to_grs(c)
sage: y.parent()
Vector space of dimension 15 over Finite Field in z4 of size 2^4
sage: y in D.grs_code()
True
```

decode_to_code(y)  
Decodes y to a codeword in sage.coding.decoder.Decoder.code().

EXAMPLES:
```python
sage: F = GF(4, 'a')
sage: a = F.gen()
sage: C = codes.BCHCode(F, 15, 3, jump_size=2)
sage: D = codes.decoders.BCHUnderlyingGRSDecoder(C)
sage: y = vector(F, [a, a + 1, 1, a + 1, 0, 1, a + 1, a, a + 1, 1, a])
sage: D.decode_to_code(y)
(a, a + 1, 1, a + 1, 1, a, a + 1, a + 1, 0, 1, a + 1, 1, 1, 1, a)
sage: D.decode_to_code(y) in C
True
```

We check that it still works when, while list-decoding, the GRS decoder output some words which do not lie in the BCH code:

```python
sage: C = codes.BCHCode(GF(2), 31, 15)
sage: D = codes.decoders.BCHUnderlyingGRSDecoder(C, "GuruswamiSudan", tau=8)
sage: c = vector(GF(2), [1, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 1, 1, 0, 0])
sage: y = vector(GF(2), [1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 1, 0, 0, 1, 0, 1, 1, 1, 0, 0])
sage: print(c in C and (c-y).hamming_weight() == 8)
True
sage: Dgrs.decode_to_code(y)
[(1, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 1, 1, 0, 0),
 (1, z5^3 + z5^2 + z5 + 1, z5^4 + z5^2 + z5, z5^4 + z5^3 + 1, 0, 0, 1, 0, 1, l, l, 1, 1, 1, 0, 0)]
sage: D.decode_to_code(y) == [c]
True
```

decoding_radius()  
Returns maximal number of errors that self can decode.

EXAMPLES:

```python
sage: C = codes.BCHCode(GF(4, 'a'), 15, 3, jump_size=2)
sage: D = codes.decoders.BCHUnderlyingGRSDecoder(C)
sage: D.decoding_radius()
1
```

grs_code()  
Returns the underlying GRS code of sage.coding.decoder.Decoder.code().

Note: Let us explain what is the underlying GRS code of a BCH code of length $n$ over $F$ with parameters $b, \delta, \ell$. Let $c \in F^n$ and $\alpha$ a primitive root of the splitting field. We know:

\[
 c \in \text{BCH} \iff \sum_{i=0}^{n-1} c_i (\alpha^{b+j})^i = 0, \quad j = 0, \ldots, \delta - 2 \iff Hc = 0
\]

3.7. BCH Code 107
where $H = A \times D$ with:

$$
A = \begin{pmatrix}
1 & \ldots & 1 \\
(\alpha^0 \times \ell)^{\delta - 2} & \ldots & (\alpha^{(n-1)} \times \ell)^{\delta - 2}
\end{pmatrix}
$$

$$
D = \begin{pmatrix}
1 & 0 & \ldots & 0 \\
0 & \alpha^b & \ldots & 0 \\
\ldots & \ldots & \ldots & \ldots \\
0 & \ldots & 0 & \alpha^{b(n-1)}
\end{pmatrix}
$$

The BCH code is orthogonal to the GRS code $C'$ of dimension $\delta - 1$ with evaluation points $\{1 = \alpha^0 \times \ell, \ldots, \alpha^{(n-1)} \times \ell\}$ and associated multipliers $\{1 = \alpha^0 \times b, \ldots, \alpha^{(n-1)} \times b\}$. The underlying GRS code is the dual code of $C'$.

**EXAMPLES:**

```python
sage: C = codes.BCHCode(GF(2), 15, 3)
sage: D = codes.decoders.BCHUnderlyingGRSDecoder(C)
sage: D.grs_code()
[15, 13, 3] Reed-Solomon Code over GF(16)
```

**grs_decoder()**

Returns the decoder used to decode words of $grs\_code()$.

**EXAMPLES:**

```python
sage: C = codes.BCHCode(GF(4, 'a'), 15, 3, jump_size=2)
sage: D = codes.decoders.BCHUnderlyingGRSDecoder(C)
sage: D.grs_decoder()
Key equation decoder for [15, 13, 3] Generalized Reed-Solomon Code over GF(16)
```

**grs_word_to_bch(c)**

Returns $c$ converted as a codeword of $sage.coding.decoder.Decoder.code()$.

**EXAMPLES:**

```python
sage: C = codes.BCHCode(GF(4, 'a'), 15, 3, jump_size=2)
sage: D = codes.decoders.BCHUnderlyingGRSDecoder(C)
sage: Cgrs = D.grs_code()
sage: Fgrs = Cgrs.base_field()
sage: b = Fgrs.gen()
sage: c = vector(Fgrs, [0, b^2 + b, 1, b^2 + b, 0, 1, 1, b^2 + b, 0, 0, b^2 + b + 1, b^2 + b, 0, 1])
sage: D.grs_word_to_bch(c)
(0, a, 1, a, 0, 1, 1, 1, a, 0, 0, a + 1, a, 0, 1)
```

In contrast, for some code families Sage can only construct their generator matrix and has no other a priori knowledge on them:

### 3.8 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.GeneralizedReedSolomonCode`. 
All codes available here can be accessed through the `codes` object:

```python
sage: codes.random_linear_code(GF(2), 5, 2)

[5, 2] linear code over GF(2)
```

REFERENCES:

- [HP2003]

AUTHOR:

- David Joyner (2007-05): initial version
- " (2008-02): added cyclic codes, Hamming codes
- " (2008-03): added BCH code, LinearCodeFromCheckmatrix, ReedSolomonCode, WalshCode, DuadicCodeEvenPair, DuadicCodeOddPair, ReedSolomonCode, 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

```python
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) $S_1, S_2$ of $n$.

**Warning:** Maybe the splitting should be associated to a sum of $q$-cyclotomic cosets mod $n$, where $q$ is a prime.

```python
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)
([11, 5] Cyclic Code over GF(3),
 [11, 5] Cyclic Code over GF(3))
```

```python
sage.coding.code_constructions.DuadicCodeOddPair(F, S1, S2)
```

Constructs the “odd pair” of duadic codes associated to the “splitting” $S_1, S_2$ of $n$.

**Warning:** Maybe the splitting should be associated to a sum of $q$-cyclotomic cosets mod $n$, where $q$ is a prime.

```python
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]
```

(continues on next page)
\texttt{sage: \_is\_a\_splitting(S1,S2,11)}
\texttt{True}
\texttt{sage: codes.DuadicCodeOddPair(GF(q),S1,S2)}
\texttt{([11, 6] Cyclic Code over GF(3), [11, 6] Cyclic Code over GF(3))}

This is consistent with Theorem 6.1.3 in [HP2003].

\texttt{sage.coding.code\_constructions.\texttt{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 [HP2003], Section 6.6.3-6.6.4.)

\textbf{INPUT:}

- \texttt{n} - an odd prime
- \texttt{F} - a finite prime field \texttt{F} whose order must be a quadratic residue modulo \texttt{n}.

\textbf{OUTPUT:} Returns an extended quadratic residue code.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: C1 = codes.QuadraticResidueCode(7,GF(2))
sage: C2 = C1.extended_code()
sage: C3 = codes.ExtendedQuadraticResidueCode(7,GF(2)); C3
Extension of [7, 4] Cyclic Code over GF(2)
sage: C2 == C3
True
sage: C = codes.ExtendedQuadraticResidueCode(17,GF(2))
sage: C
Extension of [17, 9] Cyclic Code over GF(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
\end{verbatim}

\textbf{AUTHORS:}

- David Joyner (07-2006)

\texttt{sage.coding.code\_constructions.\texttt{QuadraticResidueCode}(n, F)}

A quadratic residue code (or QR code) is a cyclic code whose generator polynomial is the product of the polynomials \( x - \alpha^i \) (\( \alpha \) is a primitive \( n^{th} \) root of unity; \( i \) ranges over the set of quadratic residues modulo \( n \)).

See \texttt{QuadraticResidueCodeEvenPair} and \texttt{QuadraticResidueCodeOddPair} for a more general construction.

\textbf{INPUT:}

- \texttt{n} - an odd prime
- \texttt{F} - a finite prime field \texttt{F} whose order must be a quadratic residue modulo \texttt{n}.

\textbf{OUTPUT:} Returns a quadratic residue code.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: C = codes.QuadraticResidueCode(7,GF(2))
sage: C
[7, 4] Cyclic Code over GF(2)
\end{verbatim}
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 [HP2003]) 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:

```python
sage: codes.QuadraticResidueCodeEvenPair(17, GF(13))  # known bug (#25896)
([17, 8] Cyclic Code over GF(13),
 [17, 8] Cyclic Code over GF(13))
sage: codes.QuadraticResidueCodeEvenPair(17, GF(2))
([17, 8] Cyclic Code over GF(2),
 [17, 8] Cyclic Code over GF(2))
sage: codes.QuadraticResidueCodeEvenPair(13, GF(9, "z"))  # known bug (#25896)
([13, 6] Cyclic Code over GF(9),
 [13, 6] Cyclic Code over GF(9))
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.systematic_generator_matrix() == C4.dual_code().systematic_generator_matrix()
True
```

This is consistent with Theorem 6.6.9 and Exercise 365 in [HP2003].

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 [HP2003]) 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:
This is consistent with Theorem 6.6.14 in [HP2003].

\texttt{sage.coding.code_constructions.ToricCode(} \texttt{P, F)\texttt{}}

Let \( P \) denote a list of lattice points in \( \mathbb{Z}^d \) 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 \( V = \text{Span}\{x^e \mid e \in P\} \). The associated toric code \( C \) is the evaluation code which is the image of the evaluation map

\[
eval_T : V \to F^n,
\]

where \( x^e \) is the multi-index notation \( (x = (x_1, \ldots, x_d), e = (e_1, \ldots, e_d), \) and \( x^e = x_1^{e_1} \cdots x_d^{e_d} \)), where \( \text{eval}_T(f(x)) = (f(t_1), \ldots, f(t_n)) \), and where \( T = \{t_1, \ldots, t_n\} \). This function returns the toric codes discussed in [Joy2004].

\textbf{INPUT:}

- \( P \) - all the integer lattice points in a polytope defining the toric variety.
- \( F \) - a finite field.

\textbf{OUTPUT:} Returns toric code with length \( n = \), dimension \( k \) over field \( F \).

\textbf{EXAMPLES:}

\begin{verbatim}
sage: C = codes.ToricCode([[0,0],[1,0],[2,0],[0,1],[1,1]],GF(7))
sage: C [36, 5] linear code over GF(7)
sage: C.minimum_distance() 24
sage: C = codes.ToricCode([[-2,-2],[-1,-2],[-1,-1],[-1,0],[0,-1],[0,0],[1,-1],[1,0]],GF(5))
sage: C [16, 9] linear code over GF(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 [49, 11] linear code over GF(8)
\end{verbatim}
This is in fact a $[49,11,28]$ code over $\text{GF}(8)$. If you type `next C.minimum_distance()` and wait overnight (!), you should get 28.

**AUTHOR:**
- David Joyner (07-2006)

```python
sage: C = codes.WalshCode(4); C
[16, 4] linear code over GF(2)
sage: C = codes.WalshCode(3); C
[8, 3] linear code over GF(2)
sage: C.spectrum()
[1, 0, 0, 0, 7, 0, 0, 0, 0]
sage: C.minimum_distance()
4
sage: C.minimum_distance(algorithm='gap')  # check $d=2^{m-1}$
4
```

**REFERENCES:**
- Wikipedia article Hadamard_matrix
- Wikipedia article Walsh_code

```python
sage: C = codes.HammingCode(GF(2), 3); C
[7, 4] Hamming Code over GF(2)
```

```python
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
[7, 4] linear code over GF(2)
sage: C2.systematic_generator_matrix() == C.systematic_generator_matrix()
True
```

```python
sage: V = VectorSpace(GF(3), 5)
sage: v = V([0, 1, 2, 0, 1])
sage: G = SymmetricGroup(5)
sage: g = G([(1, 2, 3)])
```

(continues on next page)
sage: permutation_action(g,v)
(1, 2, 0, 0, 1)
sage: g = G([(1,2,3,4,5)])
sage: permutation_action(g,v)
(0, 1, 2, 0, 1)
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: 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(\texttt{F, length, dimension})

Generate a random linear code of length \texttt{length}, dimension \texttt{dimension} and over the field \texttt{F}.

This function is Las Vegas probabilistic: always correct, usually fast. Random matrices over the \texttt{F} are drawn until one with full rank is hit.

If \texttt{F} is infinite, the distribution of the elements in the random generator matrix will be random according to the distribution of \texttt{F.random_element()}.

EXAMPLES:

\begin{verbatim}
sage: C = codes.random_linear_code(GF(2), 10, 3)
sage: C
[10, 3] linear code over GF(2)
\end{verbatim}
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:

```python
sage: walsh_matrix(2)
[0 0 1 1]
[0 1 0 1]
sage: walsh_matrix(3)
[0 0 0 1 1 1 1]
[0 0 1 0 0 1 1]
[0 1 0 1 0 1 0]
sage: C = LinearCode(walsh_matrix(4)); C
[16, 4] linear code over GF(2)
sage: C.spectrum()
[1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

This last code has minimum distance 8.

REFERENCES:

- Wikipedia article Hadamard_matrix

### 3.9 Constructions of generator matrices using the GUAVA package for GAP

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 be sphinx compatible

REFERENCES:

sage.coding.guava.QuasiQuadraticResidueCode(p)
A (binary) quasi-quadratic residue code (or QQR code).

Follows the definition of Proposition 2.2 in [BM]. The code 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.
Returns a QQR code of length $2p$.

EXAMPLES:

```python
sage: C = codes.QuasiQuadraticResidueCode(11); C   # optional - gap_packages
    (Guava package)
[22, 11] linear code over GF(2)
```

These are self-orthogonal in general and self-dual when $p$ is an odd prime.

AUTHOR: David Joyner (11-2005)

```python
sage: C = codes.RandomLinearCodeGuava(30,15,GF(2)); C   # optional - gap_packages
    (Guava package)
[30, 15] linear code over GF(2)
```

```python
sage: C = codes.RandomLinearCodeGuava(10,5,GF(4,'a')); C   # optional - gap_packages
    (Guava package)
[10, 5] linear code over GF(4)
```

AUTHOR: David Joyner (11-2005)

### 3.10 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 $[A_0,A_1,\ldots,A_n]$
- “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]:

\[
\text{C} = \text{self_dual_binary_codes}(n); \quad m = \text{len}(\text{C}.\text{keys})
\]

\[
\text{for} \quad i \quad \text{in} \quad \text{range}(m):
\]

\[
\text{C0} = \text{C}["%s" % n]["%s" % i]["code"]
\]

\[
\text{print}([n, i, \text{C}["%s" % n]["%s" % i]["spectrum"] \text{== C0.spectrum()}])
\]

\[
\text{print}(\text{C0} == \text{C0.dual_code()})
\]

\[
\text{G} = \text{C0.automorphism_group_binary_code()}
\]

\[
\text{print}(\text{C}["%s" % n]["%s" % i]["order autgp"] \text{== G.order()})
\]

To check if the “Riemann hypothesis” holds, run the following code:

\[
\begin{align*}
R &= \text{PolynomialRing}(\text{CC}, \text{"T"}) \\
T &= R.\text{gen}()
\end{align*}
\]

\[
\text{for} \quad n \quad \text{in} \quad [4, 6, 8, 10, 12, 14, 16, 18, 20, 22]:
\]

\[
\text{C} = \text{self_dual_binary_codes}(n); \quad m = \text{len}(\text{C}["%s" % n].\text{keys})
\]

\[
\text{for} \quad i \quad \text{in} \quad \text{range}(m):
\]

\[
\text{C0} = \text{C}["%s" % n]["%s" % i]["code"]
\]

\[
\text{if} \quad \text{C0.minimum_distance()} \text{>2}:
\]

\[
\text{f} = R(\text{C0.sd_zeta_polynomial}())
\]

\[
\text{print}([n, i, \text{[z[0].abs() for z in f.roots()]}])
\]

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 \( \text{MatrixSpace}(\text{GF}(2), m, m) \) of course). If \( M_1, \ldots, M_r \) are square matrices, let \( \text{diag}(M_1, M_2, \ldots, M_r) \) denote the “block diagonal” matrix with the \( M_i \)’s on the diagonal and 0’s elsewhere. Let \( C(m_1, \ldots, m_r, s) \) denote the linear code with generator matrix having block form \( G = (I, A) \), where \( A = \text{diag}(A_{m_1}, A_{m_2}, \ldots, A_{m_r}, I_s) \), for some (even) \( m_i \)’s and \( s \), where \( m_1 + m_2 + \ldots + m_r + s = n/2 \). Note: Such codes \( C(m_1, \ldots, m_r, s) \) 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 the Huffman+Pless [HP2003], page 347).

AUTHORS:

• David Joyner (2007-08-11)

REFERENCES:


\[
\text{sage.coding.self_dual_codes.self_dual_binary_codes}(n)
\]

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, \ldots are allowed. This is the coefficient of \( x^n \) in the series expansion \((1-x)^{-1} \prod_{2^j}(1-x^{2^j})^{-1}\). Typing the command 
\[
f = (1-x)(-1) \ast \text{prod}([1-(1-x(2^j))(-1) \text{for } j \text{ in } \text{range}(2,18)])
\]
into Sage, we obtain for the coeffs of \( x^4, x^6, \ldots \) [1, 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] These numbers grow too slowly to account for all the sd codes (see Huffman+Pless’ Table 9.1, referenced above). In fact, in Table 9.10 of [HP2003], the number \( B_n \) of inequivalent sd binary codes of length \( n \) is given:
According to http://oeis.org/classic/A003179, the next 2 entries are: 3295, 24147.

EXAMPLES:

```python
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
```

### 3.11 Optimized low-level binary code representation

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

$$g(a_i) + g(b_i) = g((a_i + b_i)).$$

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.

**EXAMPLES:**

```python
sage: import sage.coding.binary_code
sage: from sage.coding.binary_code import *

sage: M = Matrix(GF(2), [[1,1,1,1]])
(continues on next page)```
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

EXAMPLES:

sage: from sage.coding.binary_code import *
sage: B = BinaryCode(codes.GolayCode(GF(2)).generator_matrix())
sage: B
Binary [24,12] linear code, generator matrix
  [1000000000000101011110011]
  [0100000000001111101001010]
  [0010000000001100011101101]
  [0001000000001100111001101]
  [0000100000001101110011101]
  [0000010000001101111011101]
  [0000001000001101111110101]
  [0000000100001101111111011]
  [0000000010001101111111101]
  [0000000001001101111111110]
  [0000000000101001111111111]
  [0000000000010101111111111]
  [0000000000001011111111111]
  [0000000000000111111111111]
sage: B.apply_permutation(list(range(11,-1,-1)) + list(range(12, 24)))
sage: B
Binary [24,12] linear code, generator matrix
  [000000000001101011100011]

(continues on next page)
matrix()  
Returns the generator matrix of the BinaryCode, i.e. the code is the rowspace of B.matrix().

EXAMPLES:

```
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
```
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.

EXAMPLES:

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

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.

EXAMPLES:

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

(continues on next page)
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.

EXAMPLES:

```python
sage: from sage.coding.binary_code import *
sage: BC = BinaryCodeClassifier()
sage: B = BinaryCode(codes.GolayCode(GF(2)).generator_matrix())
sage: B.apply_permutation(list(range(24,-1,-1)))
sage: B
Binary [24,12] linear code, generator matrix
[011001110101000000000000]
[100000000000110011100101]
[001000000001010001111000]
[000100000000101100110100]
[000010000000100110010101]
[000001000000011000110110]
[000000100000110011101101]
[000000010000111111011001]
[000000001000010011111011]
[000000000100010011111011]
[000000000010001011111111]
[sage: BC.put_in_canonical_form(B)]
sage: B
Binary [24,12] linear code, generator matrix
[100000000000110011100101]
[010000000000110011100101]
[001000000001010001111000]
[000100000000101100110100]
[000010000000100110010101]
[000001000000011000110110]
[000000100000110011101101]
[000000010000111111011001]
[000000001000010011111011]
[000000000100010011111011]
[000000000010001011111111]
```

class sage.coding.binary_code.OrbitPartition

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.

See Wikipedia article 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:**

```python
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)

sage: P._split_vertex(0, 1)

sage: P._refine(1, [[0,0]], B)

sage: P._split_vertex(1, 2)

sage: P._refine(2, [[0,1]], B)

sage: P._split_vertex(2, 3)

sage: P._refine(3, [[0,2]], B)

sage: P._split_vertex(4, 4)

sage: P._refine(4, [[0,4]], B)

sage: P._is_discrete(4)

sage: Q = PartitionStack(P)

sage: Q._clear(4)

sage: Q._split_vertex(5, 4)

sage: Q._refine(4, [[0,4]], B)

sage: Q._is_discrete(4)

sage: Q.cmp(P, B)
```

**print** **basis**( )

**EXAMPLES:**

```python
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
```

(continues on next page)
(continued from previous page)

```python
({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})
```

```python
sage: P._find_basis()
sage: P.print_basis()
basis_locations:
  4
  8
  0
  11
```

**print_data()**

Prints all data for self.

**EXAMPLES:**

```python
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
  -1
col_degs:
  0
  0
  0
  0
```

(continues on next page)
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 \(B' = \text{extspan}(B, 1)\) (note that this guarantees that all the vectors in the span of the output have even weight).

Tests the WordPermutation structs for at least \(t\_limit=5.0\) 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 \(i \rightarrow L[i]\). 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 Word-
   Permutations. Takes a random word and two random permutations, and tests that the result of per-
  muting by the composition is correct.

3. **Tests create_inv_word_perm and create_id_word_perm, which create a** WordPermutation as the in-
   verse 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.

---

```
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],
    ....: [0,0,0,0,0,0,1,1,1,1,1,1,1,1,0,0,0],
    ....: [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, 68, 0, 85, 0, 68, 0, 34, 0, 0, 0, 0, 0, 0]
```
Sage supports the following derived code constructions. If the constituent code is from a special code family, the derived codes inherit e.g. decoding or minimum distance capabilities:

### 4.1 Subfield subcode

Let $C$ be a $[n, k]$ code over $F_q$. Let $C_s = \{ c \mid \forall i, c_i \in F_q \}$, $c_i$ being the $i$-th coordinate of $c$.

$C_s$ is called the subfield subcode of $C$ over $F_q$.

```python
from sage.coding.subfield_subcode import SubfieldSubcode

C = codes.random_linear_code(GF(16, 'aa'), 7, 3)
Cs = SubfieldSubcode(C, GF(4, 'a'))
```

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:**

```python
sage: C = codes.random_linear_code(GF(16, 'aa'), 7, 3)
sage: Cs = codes.SubfieldSubcode(C, GF(4, 'a'))
```

- `dimension()`
  - Returns the dimension of `self`.

- `dimension_lower_bound()`
  - Returns a lower bound for the dimension of `self`.

**EXAMPLES:**

- `dimension_lower_bound()`
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()
[7, 3] linear code over GF(16)

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 0 a + 1 0]
[0 0 1 0 0 0 0 0 0 0 0 a + 1 a]
[ 0 0 0 1 0 0 0 0 0 0 0 0 a + 1]
[ 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]
[ 0 0 0 0 0 0 1 0 0 0 a a a]

(continues on next page)
Decoder decoding through a decoder over the original code of \( \text{code} \).

**INPUT:**

- `\text{code}` – The associated code of this decoder
- `\text{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:**

```python
code: C = codes.GeneralizedReedSolomonCode(GF(16, 'aa').list()[:13], 5)
code: Cs = codes.SubfieldSubcode(C, GF(4, 'a'))
code: D = codes.decoders.SubfieldSubcodeOriginalCodeDecoder(Cs)
code: Chan = channels.StaticErrorRateChannel(Cs.ambient_space(), D.decoding_radius())
code: c = Cs.random_element()
code: y = Chan(c)
code: c == D.decode_to_code(y)
```

```
True
```

- `\text{code}` – The associated code of this decoder
- `\text{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 decoder's `\text{sage.coding.decoder.}

Decoder.decoding_radius()` method.

**EXAMPLES:**

```python
code: C = codes.GeneralizedReedSolomonCode(GF(16, 'aa').list()[:13], 5)
code: Cs = codes.SubfieldSubcode(C, GF(4, 'a'))
code: D = codes.decoders.SubfieldSubcodeOriginalCodeDecoder(Cs)
code: Chan = channels.StaticErrorRateChannel(Cs.ambient_space(), D.decoding_radius())
code: c = Cs.random_element()
code: y = Chan(c)
code: c == D.decode_to_code(y)
```

```
True
```
```python
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:**

```python
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] Reed-Solomon Code over GF(16)
```

### 4.2 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:**

```python
sage: C = codes.random_linear_code(GF(7), 11, 5)
sage: Cp = codes.PuncturedCode(C, 3)
sage: Cp
Puncturing of [11, 5] linear code over GF(7) on position(s) [3]
sage: Cp = codes.PuncturedCode(C, {3, 5})
sage: Cp
Puncturing of [11, 5] linear code over GF(7) on position(s) [3, 5]
```

**dimension()**

Returns the dimension of `self`.

**EXAMPLES:**

```python
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:**

```python
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:**

```python
sage: C = codes.random_linear_code(GF(7), 11, 5)
sage: Cp = codes.PuncturedCode(C, 3)
sage: Cp.original_code()
[11, 5] linear code over GF(7)
```

**punctured_positions()**
Returns the list of positions which were punctured on the original code.

**EXAMPLES:**

```python
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:**

- `args, 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] Reed-Solomon Code over GF(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()
[9, 5] linear code over GF(2)
```

---

**class** `sage.coding.punctured_code.PuncturedCodeOriginalCodeDecoder`

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:**

```python
sage: C = codes.GeneralizedReedSolomonCode(GF(16, 'a').list()[:15], 7)
sage: Cp = codes.PuncturedCode(C, 3)
sage: codes.decoders.PuncturedCodeOriginalCodeDecoder(Cp)
Decoder of Puncturing of [15, 7, 9] Reed-Solomon Code over GF(16) on
---position(s) [3] through Error-Erasure decoder for [15, 7, 9] Reed-
---Solomon Code over GF(16)
```

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:

```python
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:

```python
sage: C = codes.GeneralizedReedSolomonCode(GF(16, 'a').list()[:15], 7)
sage: Cp = codes.PuncturedCode(C, 3)
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:**

```python
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:**

4.2. Punctured code
```python
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
```

```python
original_decoder()

Returns the decoder over the original code that will be used to decode words of `sage.coding.decoder.Decoder.code()`.

EXAMPLES:
```python
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] Reed-Solomon Code over GF(16)
```

```python
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:
```python
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 Puncturing of [11, 5] linear code over GF(7) on position(s) [3]
```

generator_matrix()

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

EXAMPLES:
```python
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 5 2 6 0 6]
[0 1 0 0 5 2 2 1 1]
[0 0 1 0 6 2 4 0 4]
[0 0 0 1 0 6 3 3 3]
[0 0 0 0 1 0 1 3 4 3]
```

### 4.3 Extended code

Let $C$ be a linear code of length $n$ over $\mathbb{F}_q$. The extended code of $C$ is the code

$$\hat{C} = \{ x_1 x_2 \ldots x_{n+1} \in \mathbb{F}_q^{n+1} \mid x_1 x_2 \ldots x_n \in C \text{ with } x_1 + x_2 + \cdots + x_{n+1} = 0 \}.$$
See [HP2003] (pp 15-16) for details.

```python
class sage.coding.extended_code.ExtendedCode(C):
    Bases: sage.coding.linear_code.AbstractLinearCode

Representation of an extended code.

INPUT:

- C – A linear code

EXAMPLES:

```python
sage: C = codes.random_linear_code(GF(7), 11, 5)
sage: Ce = codes.ExtendedCode(C)
sage: Ce
Extension of [11, 5] linear code over GF(7)
```

original_code()

Returns the code which was extended to get self.

EXAMPLES:

```python
sage: C = codes.random_linear_code(GF(7), 11, 5)
sage: Ce = codes.ExtendedCode(C)
sage: Ce.original_code()
[11, 5] linear code over GF(7)
```

parity_check_matrix()

Returns a parity check matrix of self.

This matrix is computed directly from `original_code()`.

EXAMPLES:

```python
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:

```python
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

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 = C.random_element()
    sage: Chan = channels.StaticErrorRateChannel(Ce.ambient_space(), D.decoding_radius())
    sage: y = Chan(c)
    (continues on next page)
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
```

\texttt{decoding\_radius(*args, **kwargs)}

Returns maximal number of errors that \texttt{self} can decode.

\textbf{INPUT:}

- \texttt{*args, **kwargs} - arguments and optional arguments are forwarded to original decoder's \texttt{decoding\_radius} method.

\textbf{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
```

\texttt{original\_decoder()}

Returns the decoder over the original code that will be used to decode words of \texttt{sage.coding.decoder.Decoder.code().}

\textbf{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] Reed-Solomon Code over GF(16)
```

Other derived constructions that simply produce the modified generator matrix can be found among the methods of a constructed code.
5.1 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`. 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 $G = (GL(k,q) \times F_q^n) \rtimes Aut(F_q)$ on the set $X = (F_q^k)^n$ of $k \times n$ matrices over $F_q$ (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 $G_{\Pi(I')(x)}$. 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 $(F_q^n \rtimes Aut(F_q)) \rtimes S_n = (F_q^n \rtimes Aut(F_q)) \times (Aut(F_q) \times S_n)$.

AUTHORS:

- Thomas Feulner (2012-11-15): initial version

REFERENCES:

- [Feu2009]

EXAMPLES:

Get the canonical form of the Simplex code:
```python
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:

```python
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:

```python
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
```

REFERENCES:

```python
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_generic with $G = (GL(k,q) \times F_q^n) \rtimes Aut(F_q)$.

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 $2 \times 2$ 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 $\varphi_i$ is equal to the entries of the corresponding diagonal entry of $A$.
- $\alpha$ is a power of $\tau^{frob\_pow}$, where $\tau$ denotes the Frobenius automorphism of the finite field $F_q$.

See [Feu2009] for more details.

```
```
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
```

```python
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 1 1 1 1 1 1 1 1]
[0 1 0 1 0 0 1 2 2 2 0 0]
[0 0 1 2 1 2 1 2 0 0 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: A = P.get_autom_gens()
sage: all( [(a*mat).echelon_form() == mat.echelon_form() for a in A])
True
```
```python
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:

```python
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: CF1 = P.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:

```python
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
```

5.2 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 $C \leq F_3^n$. 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 $G = F_3^{*n} \rtimes (\text{Aut}(F_3) \times S_n)$ on the set of subspaces of $F_3^n$. 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 $\mathbb{F}_q^n \rtimes S_n$ 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

EXAMPLES:

```python
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 1 1 1 1]
[0 1 0 1 1 0 1 1 2 1 2]
[0 0 1 2 1 2 1 2 0 1 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:

```python
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:

```python
sage: C = codes.HammingCode(GF(4, 'a'), 3).dual_code()
sage: P = LinearCodeAutGroupCanLabel(C, P=[ [0], [1], list(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, 19, 9, 21, 16, 14, 11, 20, 15, 8, 10, 12, 7, 13, 18, 17]]
```

We can also restrict the group action to linear isometries:

```python
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:

```
```
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 $\phi \in \mathbb{F}_q^*$ 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 $\phi$ of units of length $n$ and a field automorphism $\alpha$ 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:

```python
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 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 $PGL(k, q)$.

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 $PGL(k, q)$. Therefore, we may interpret the group as a subgroup of $PGL(k, q)$ as well.

EXAMPLES:
```python
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 $\text{PGL}(k,q)$.

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 $\text{PG}(k-1,q)$. The equivalence of codes translates to the natural action of $\text{PGL}(k,q)$. Therefore, we may interpret the group as a subgroup of $\text{PGL}(k,q)$ as well.

**EXAMPLES:**

```python
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:**

```python
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:**

```python
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:**

```python
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()
```

(continues on next page)

5.2. Canonical forms and automorphisms for linear codes over finite fields
```python
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:**

```python
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
```
6.1 Huffman Encoding

This module implements functionalities relating to Huffman encoding and decoding.

AUTHOR:

• Nathann Cohen (2010-05): initial version.

6.1.1 Classes and functions

```python
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 fre-
        quency 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.

    In either case, the alphabet must consist of at least two symbols.

    Examples:
```
We can obtain the same result by "training" the Huffman code with the following table of frequency:

```python
sage: ft = frequency_table("There once was a french fry")
sage: sorted(ft.items())
[(' ', 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)]
```

```python
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:

```python
sage: encoded = h1.encode("There once was a french fry"); encoded
...
\[\text{string encoded...}\]
```

We can decode the above encoded string in the following way:

```python
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:

```python
sage: h3 = Huffman("There once were two french fries")
sage: h3.decode(encoded)
'eierffcoeft TfewrnwrTrsc'
```
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
˓→'1100001101000101010110000111110100111001110100110110111101111011100111101000010110111010000011101010100010100000001011101 ...
˓→'
sage: h.decode(encoded)
'Sage is my most favorite general purpose computer algebra system'
```

**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:

```python
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
'11000110100010101101110111101100110111101110111011011011001110111011011011110111011011011111100110111010011011101001011100101100011001010010011101011101011101010000100011110110100011100111001011011101000010110111010000011101010100010100000001011101 ...

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:

```python
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("{} {}\n".format(symbol, code))
101 $ 110000
101 a 1101
101 b 11001
101 c 110010
101 e 010
101 f 11011
101 g 0001
101 i 10000
101 l 10001
101 m 0011
101 n 00000
101 o 0110
101 p 0010
101 r 1110
101 s 1111
101 t 0111
101 u 10010
101 v 00001
101 y 10011
```

tree()

Returns the Huffman tree corresponding to the current encoding.

INPUT:

• None.

OUTPUT:
• The binary tree representing a Huffman code.

**EXAMPLES:**

```python
def from sage.coding.source_coding.huffman import Huffman
sage: str = "Sage is my 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])
```

```python
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:

```python
def 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:

```python
sage: frequency_table("")
defaultdict(<... 'int'>, {})
```

**6.1. Huffman Encoding**
7.1 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).

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:

```python
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:

```python
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:**

```python
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:**

```python
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:**

```python
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:**

```python
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:

```python
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:

```python
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() Return \( m \), the extension degree of the absolute field over the relative field.

EXAMPLES:

```python
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:

```python
sage: from sage.coding.relative_finite_field_extension import *

sage: Fqm.<aa> = GF(16)
sage: Fq.<a> = GF(4)
```

(continues on next page)
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:

```python
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:

```python
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:

```python
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:

```python
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:

```python
sage: from sage.coding.relativeFiniteFieldExtension 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)
```

### 7.2 Guruswami-Sudan decoder for (Generalized) Reed-Solomon codes

REFERENCES:
- [GS1999]
- [Nie2013]

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

```python
class sage.coding.guruswami_sudan.gs_decoder.GRSGuruswamiSudanDecoder:
```

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 [GS1999] 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:
  – "Alekhnovich" – uses Alekhnovich’s algorithm.
  – "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] Reed-Solomon Code over GF(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] Reed-Solomon Code over GF(251)
˓→decoding 97 errors with parameters (1, 2)
```

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

```python
sage: D = codes.decoders.GRSGuruswamiSudanDecoder(C, parameters = (1,2), root_finder="RothRuckenstein")
```

Actually, we can construct the decoder from C directly:

```python
sage: D = C.decoder("GuruswamiSudan", tau = 97)
```

### 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:**

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

### 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.GRGuruswamiSudanDecoder(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

defending_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.defending_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.defending_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 [Nie2013] page 49, definition 3.3 and proposition 3.4 for details.

INPUT:

- tau – an integer, number of errors 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.GRGuruswamiSudanDecoder.gs_satisfactory(tau, s, l, n_k=(n, k))
True
One can also pass a GRS code:

```python
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:

```python
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:

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

Same if one provides none of these:

```python
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
```

The static method `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
  - \( s, l \) are the multiplicity parameter, respectively list size parameter giving this radius.
EXAMPLES:

```python
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:

```python
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:

```python
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:

```python
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:

```python
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:

```python
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`.

164 Chapter 7. Other modules
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_value). See roth_ruckenstein_root_finder() for an example.
```

7.2. Guruswami-Sudan decoder for (Generalized) Reed-Solomon codes
EXAMPLES:

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

`sage.coding.guruswami_sudan.gs_decoder.alekhnovich_root_finder(p, maxd=None, precision=None)`

Wrapper for Alekhnovich’s algorithm to compute the roots of a polynomial with coefficients in \( \mathbb{F}[x] \).

`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 \( \mathbb{F}[x] \).

7.3 Interpolation algorithms for the Guruswami-Sudan decoder

**AUTHORS:**
- Johan S. R. Nielsen, original implementation (see [Nie] 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 $\mathbb{F}_q[x]$ 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 $(x_i, y_i)$ such that we seek $Q$ with $(x_i,y_i)$ 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:

```python
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: Q = gs_interpolation_lee_osullivan(points, tau, params, wy)
sage: Q / Q.lc()  # make monic
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 $(x_i, y_i)$ such that we seek $Q$ with $(x_i,y_i)$ 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 $\mathbb{F}(11)$ with multiplicity 2 and list size 4:

```python
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 + y^2 - 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:

```python
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 $\mathbb{F}_q[x]$ 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 $\mathbb{F}_q[x][y]$:

$$P_i(x,y) = G(x)[i-\lfloor a \rfloor](y - R(x))^{\lfloor i-\lfloor a \rfloor \rfloor},$$

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 $(x_i, y_i)$ such that we seek $Q$ with $(x_i, y_i)$ 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:

```python
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 + 9]
 [10*x^7 + 4*x^6 + 9*x^4 + 7*x^3 + 2*x^2 + 9*x + 9]]
```

7.4 Guruswami-Sudan utility methods

AUTHORS:

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

sage.coding.guruswami_sudan.utils.gilt\(\(x\)\)

Returns the greatest integer smaller than \(x\).

EXAMPLES:

```python
sage: from sage.coding.guruswami_sudan.utils import gilt
sage: gilt(43)
42
```

It works with any type of numbers (not only integers):

```python
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:

```python
sage: sage.coding.guruswami_sudan.utils.johnson_radius(250, 181)
-5*sqrt(690) + 250
```

sage.coding.guruswami_sudan.utils.ligt\(\(x\)\)

Returns the least integer greater than \(x\).

EXAMPLES:

```python
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 \( \text{len} \).

**INPUT:**

- \( p \) – a polynomial
- \( \text{len} \) – an integer. If \( \text{len} \) is smaller than the degree of \( p \), the returned list will be of size degree of \( p \), else it will be of size \( \text{len} \).

**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.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 \(ax^2 + bx + c = 0\).

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:

```
solve_degree2_to_integer_range(50, 5, 42)
(-2, -1)
```

## 7.5 Information-set decoding for linear codes

Information-set decoding is a probabilistic decoding strategy that essentially tries to guess \( k \) correct positions in the received word, where \( k \) is the dimension of the code. A codeword agreeing with the received word on the guessed position can easily be computed, and their difference is one possible error vector. A “correct” guess is assumed when this error vector has low Hamming weight.

This simple algorithm is not very efficient in itself, but there are numerous refinements to the strategy that make it very capable over rather large codes. Still, the decoding algorithm is exponential in dimension of the code and the log of the field size.

The ISD strategy requires choosing how many errors is deemed acceptable. One choice could be \( d/2 \), where \( d \) is the minimum distance of the code, but sometimes \( d \) is not known, or sometimes more errors are expected. If one chooses anything above \( d/2 \), the algorithm does not guarantee to return a nearest codeword.
AUTHORS:

- David Lucas, Johan Rosenkilde, Yann Laigle-Chapuy (2016-02, 2017-06): initial version

```python
class sage.coding.information_set_decoder.InformationSetAlgorithm:
    Bases: sage.structure.sage_object.SageObject

    Abstract class for algorithms for sage.coding.information_set_decoder.
    LinearCodeInformationSetDecoder.

    To sub-class this class, override decode and calibrate, and call the super constructor from __init__.

    INPUT:

    - code -- A linear code for which to decode.
    - number_errors -- an integer, the maximal number of errors to accept as correct decoding. An interval
      can also be specified by giving a pair of integers, where both end values are taken to be in the interval.
    - algorithm_name -- A name for the specific ISD algorithm used (used for printing).
    - parameters -- (optional) A dictionary for setting the parameters of this ISD algorithm. Note that sanity
      checking this dictionary for the individual sub-classes should be done in the sub-class constructor.

    EXAMPLES:

    sage: from sage.coding.information_set_decoder import LeeBrickellISDAlgorithm
    sage: LeeBrickellISDAlgorithm(codes.GolayCode(GF(2)), (0,4))
    ISD Algorithm (Lee-Brickell) for [24, 12, 8] Extended Golay code over GF(2)
    → decoding up to 4 errors

    A minimal working example of how to sub-class:

    sage: from sage.coding.information_set_decoder import InformationSetAlgorithm
    sage: from sage.coding.decoder import DecodingError
    sage: class MinimalISD(InformationSetAlgorithm):
    ...     def __init__(self, code, decoding_interval):
    ...         super(MinimalISD, self).__init__(code, decoding_interval,
    ...             "MinimalISD")
    ...     def calibrate(self):
    ...         self._parameters = {} # calibrate parameters here
    ...         self._time_estimate = 10.0 # calibrated time estimate
    ...     def decode(self, r):
    ...         # decoding algorithm here
    ...         raise DecodingError("I failed")
    sage: MinimalISD(codes.GolayCode(GF(2)), (0,4))
    ISD Algorithm (MinimalISD) for [24, 12, 8] Extended Golay code over GF(2)
    → decoding up to 4 errors
```

calibrate()

Uses test computations to estimate optimal values for any parameters this ISD algorithm may take.

Must be overridden by sub-classes.

If self._parameters_specified is False, this method shall set self._parameters to the best parameters estimated. It shall always set self._time_estimate to the time estimate of using
EXAMPLES:

```python
sage: from sage.coding.information_set_decoder import LeeBrickellISDAlgorithm
sage: C = codes.GolayCode(GF(2))
sage: A = LeeBrickellISDAlgorithm(C, (0,3))
sage: A.calibrate()
#random
{'search_size': 1}
```

code()

Return the code associated to this ISD algorithm.

EXAMPLES:

```python
sage: from sage.coding.information_set_decoder import LeeBrickellISDAlgorithm
sage: C = codes.GolayCode(GF(2))
sage: A = LeeBrickellISDAlgorithm(C, (0,3))
sage: A.code()
[24, 12, 8] Extended Golay code over GF(2)
```

decode(r)

Decode a received word using this ISD decoding algorithm.

Must be overridden by sub-classes.

EXAMPLES:

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

```python
sage: from sage.coding.information_set_decoder import LeeBrickellISDAlgorithm
sage: A = LeeBrickellISDAlgorithm(C, (2,2))
```

```python
sage: r = vector(GF(2), [1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

```python
sage: A.decode(r)
(0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)
```

decoding_interval()

A pair of integers specifying the interval of number of errors this ISD algorithm will attempt to correct.

The interval includes both end values.

EXAMPLES:

```python
sage: C = codes.GolayCode(GF(2))
```

```python
sage: from sage.coding.information_set_decoder import LeeBrickellISDAlgorithm
sage: A = LeeBrickellISDAlgorithm(C, (0,2))
```

```python
sage: A.decoding_interval()
(0, 2)
```

name()

Return the name of this ISD algorithm.

EXAMPLES:
sage: C = codes.GolayCode(GF(2))
sage: from sage.coding.information_set_decoder import LeeBrickellISDAlgorithm
sage: A = LeeBrickellISDAlgorithm(C, (0,2))
sage: A.name()
'Lee-Brickell'

parameters()  
Return any parameters this ISD algorithm uses.

If the parameters have not already been set, efficient values will first be calibrated and returned.

EXAMPLES:

sage: C = codes.GolayCode(GF(2))
sage: from sage.coding.information_set_decoder import LeeBrickellISDAlgorithm
sage: A = LeeBrickellISDAlgorithm(C, (0,4), search_size=3)
sage: A.parameters()
{'search_size': 3}

If not set, calibration will determine a sensible value:

sage: A = LeeBrickellISDAlgorithm(C, (0,4))
sage: A.parameters()  #random
{'search_size': 1}

time_estimate()  
Estimate for how long this ISD algorithm takes to perform a single decoding.

The estimate is for a received word whose number of errors is within the decoding interval of this ISD algorithm.

EXAMPLES:

sage: C = codes.GolayCode(GF(2))
sage: from sage.coding.information_set_decoder import LeeBrickellISDAlgorithm
sage: A = LeeBrickellISDAlgorithm(C, (0,2))
sage: A.time_estimate()  #random
0.0008162108571427874

class sage.coding.information_set_decoder.LeeBrickellISDAlgorithm(code, 
decoding_interval, 
search_size=None)

Bases: sage.coding.information_set_decoder.InformationSetAlgorithm

The Lee-Brickell algorithm for information-set decoding.

For a description of the information-set decoding paradigm (ISD), see sage.coding.information_set_decoder.LinearCodeInformationSetDecoder.

This implements the Lee-Brickell variant of ISD, see [LB1988] for the original binary case, and [Pet2010] for the $q$-ary extension.

Let $C$ be a $[n,k]$-linear code over $GF(q)$, and let $r \in GF(q)^n$ be a received word in a transmission. We seek the codeword whose Hamming distance from $r$ is minimal. Let $p$ and $w$ be integers, such that $0 \leq p \leq w$. Let $G$ be a generator matrix of $C$, and for any set of indices $I$, we write $G_I$ for the matrix formed by the columns of $G$ indexed by $I$. The Lee-Brickell ISD loops the following until it is successful:

1. Choose an information set $I$ of $C$.
2. Compute $r' = r - r_I \times G_I^{-1} \times G$
3. Consider every size-\(p\) subset of \(I, \{a_1, \ldots, a_p\}\). For each \(m = (m_1, \ldots, m_p) \in GF(q)^p\), compute the error vector \(e = r' - \sum_{i=1}^{p} m_i \times g_{a_i}\).

4. If \(e\) has a Hamming weight at most \(w\), return \(r - e\).

INPUT:

- `code` – A linear code for which to decode.
- `decoding_interval` – a pair of integers specifying an interval of number of errors to correct. Includes both end values.
- `search_size` – (optional) the size of subsets to use on step 3 of the algorithm as described above. Usually a small number. It has to be at most the largest allowed number of errors. A good choice will be approximated if this option is not set; see `sage.coding.LeeBrickellISDAlgorithm.calibrate()` for details.

EXAMPLES:

```python
sage: C = codes.GolayCode(GF(2))
sage: from sage.coding.information_set_decoder import LeeBrickellISDAlgorithm
sage: A = LeeBrickellISDAlgorithm(C, (0,4)); A
ISD Algorithm (Lee-Brickell) for [24, 12, 8] Extended Golay code over GF(2) → decoding up to 4 errors
sage: C = codes.GolayCode(GF(2))
sage: A = LeeBrickellISDAlgorithm(C, (2,3)); A
ISD Algorithm (Lee-Brickell) for [24, 12, 8] Extended Golay code over GF(2) → decoding between 2 and 3 errors
```

calibrate()

Run some test computations to estimate the optimal search size.

Let \(p\) be the search size. We should simply choose \(p\) such that the average expected time is minimal. The algorithm succeeds when it chooses an information set with at least \(k - p\) correct positions, where \(k\) is the dimension of the code and \(p\) the search size. The expected number of trials we need before this occurs is:

\[
\frac{n}{\rho} \sum_{i=0}^{p} \binom{n - \tau}{k - i} \binom{\tau}{i}
\]

Here \(\rho\) is the fraction of \(k\) subsets of indices which are information sets. If \(T\) is the average time for steps 1 and 2 (including selecting \(I\) until an information set is found), while \(P(i)\) is the time for the body of the for-loop in step 3 for \(m\) of weight \(i\), then each information set trial takes roughly time \(T + \sum_{i=0}^{p} P(i)(\binom{k}{i})q^{-1}\), where \(F_q\) is the base field.

The values \(T\) and \(P\) are here estimated by running a few test computations similar to those done by the decoding algorithm. We don’t explicitly estimate \(\rho\).

OUTPUT: Does not output anything but sets private fields used by `sage.coding.information_set_decoder.InformationSetAlgorithm.parameters()` and `sage.coding.information_set_decoder.InformationSetAlgorithm.time_estimate()`.

EXAMPLES:

```python
sage: from sage.coding.information_set_decoder import LeeBrickellISDAlgorithm
sage: C = codes.GolayCode(GF(2))
sage: A = LeeBrickellISDAlgorithm(C, (0,3)); A
ISD Algorithm (Lee-Brickell) for [24, 12, 8] Extended Golay code over GF(2) → decoding up to 3 errors
```
If we specify the parameter at construction time, calibrate does not override this choice:

```plaintext
sage: A = LeeBrickellISDAlgorithm(C, (0,3), search_size=2); A
ISD Algorithm (Lee-Brickell) for [24, 12, 8] Extended Golay code over GF(2) → decoding up to 3 errors
sage: A.parameters()
{'search_size': 2}
sage: A.calibrate()
sage: A.parameters()
{'search_size': 2}
sage: A.time_estimate() #random
0.0008162108571427874
```

`decode(r)`

The Lee-Brickell algorithm as described in the class doc.

Note that either parameters must be given at construction time or `sage.coding.information_set_decoder.InformationSetAlgorithm.calibrate()` should be called before calling this method.

INPUT:

- `r` – a received word, i.e. a vector in the ambient space of `decoder.Decoder.code()`.

OUTPUT: A codeword whose distance to `r` satisfies `self.decoding_interval()`.

EXAMPLES:

```plaintext
sage: M = matrix(GF(2),
             [[1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0],
              [0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 1, 1],
              [0, 0, 0, 1, 0, 1, 0, 1, 1, 0, 0, 0],
              [0, 0, 0, 0, 1, 1, 0, 0, 1, 1, 0, 1],
              [0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 1]])
sage: C = codes.LinearCode(M)
sage: from sage.coding.information_set_decoder import LeeBrickellISDAlgorithm
sage: A = LeeBrickellISDAlgorithm(C, (2,2))
sage: c = C.random_element()
sage: Chan = channels.StaticErrorRateChannel(C.ambient_space(), 2)
sage: r = Chan(c)
sage: c_out = A.decode(r)
sage: (r - c).hamming_weight() == 2
True
```

```python
class sage.coding.information_set_decoder.LinearCodeInformationSetDecoder

    sage: C = codes.LinearCode(M)
    sage: from sage.coding.information_set_decoder import LeeBrickellISDAlgorithm
    sage: A = LeeBrickellISDAlgorithm(C, (2,2))
    sage: c = C.random_element()
    sage: Chan = channels.StaticErrorRateChannel(C.ambient_space(), 2)
    sage: r = Chan(c)
    sage: c_out = A.decode(r)
    sage: (r - c).hamming_weight() == 2
    True

class sage.coding.information_set_decoder.LinearCodeInformationSetDecoder

    Bases: sage.coding.decoder.Decoder

    Information-set decoder for any linear code.

7.5. Information-set decoding for linear codes
Information-set decoding is a probabilistic decoding strategy that essentially tries to guess \( k \) correct positions in the received word, where \( k \) is the dimension of the code. A codeword agreeing with the received word on the guessed position can easily be computed, and their difference is one possible error vector. A “correct” guess is assumed when this error vector has low Hamming weight.

The ISD strategy requires choosing how many errors is deemed acceptable. One choice could be \( d/2 \), where \( d \) is the minimum distance of the code, but sometimes \( d \) is not known, or sometimes more errors are expected. If one chooses anything above \( d/2 \), the algorithm does not guarantee to return a nearest codeword.

This simple algorithm is not very efficient in itself, but there are numerous refinements to the strategy. Specifying which strategy to use among those that Sage knows is done using the algorithm keyword. If this is not set, an efficient choice will be made for you.

The various ISD algorithms all need to select a number of parameters. If you choose a specific algorithm to use, you can pass these parameters as named parameters directly to this class’ constructor. If you don’t, efficient choices will be calibrated for you.

**Warning:** If there is no codeword within the specified decoding distance, then the decoder may never terminate, or it may raise a `sage.coding.decoder.DecodingError` exception, depending on the ISD algorithm used.

**INPUT:**

- **code** – A linear code for which to decode.
- **number_errors** – an integer, the maximal number of errors to accept as correct decoding. An interval can also be specified by giving a pair of integers, where both end values are taken to be in the interval.
- **algorithm** – (optional) the string name of the ISD algorithm to employ. If this is not set, an appropriate one will be chosen. A constructed `sage.coding.information_set_decoder.InformationSetAlgorithm` object may also be given. In this case `number_errors` must match that of the passed algorithm.
- ****kwargs** – (optional) any number of named arguments passed on to the ISD algorithm. Such are usually not required, and they can only be set if `algorithm` is set to a specific algorithm. See the documentation for each individual ISD algorithm class for information on any named arguments they may accept.

The easiest way to access this documentation is to first construct the decoder without passing any named arguments, then accessing the ISD algorithm using `sage.coding.information_set_decoder.LinearCodeInformationSetDecoder.algorithm()`, and then reading the ? help on the constructed object.

**EXAMPLES:**

The principal way to access this class is through the `sage.code.linear_code.AbstractLinearCode.decoder()` method:

```
sage: C = codes.GolayCode(GF(3))
sage: D = C.decoder("InformationSet", 2); D
Information-set decoder (Lee-Brickell) for [12, 6, 6] Extended Golay code over
˓→GF(3) decoding up to 2 errors
```

You can specify which algorithm you wish to use, and you should do so in order to pass special parameters to it:

```
sage: C = codes.GolayCode(GF(3))
sage: D2 = C.decoder("InformationSet", 2, algorithm="Lee-Brickell", search_
˓→size=2); D2
Information-set decoder (Lee-Brickell) for [12, 6, 6] Extended Golay code over
˓→GF(3) decoding up to 2 errors
```

(continues on next page)
If you specify an algorithm which is not known, you get a friendly error message:

```
sage: C.decoder("InformationSet", 2, algorithm="NoSuchThing")
Traceback (most recent call last):
...
ValueError: Unknown ISD algorithm 'NoSuchThing'. The known algorithms are ['Lee-Brickell'].
```

You can also construct an ISD algorithm separately and pass that. This is mostly useful if you write your own ISD algorithms:

```
sage: from sage.coding.information_set_decoder import LeeBrickellISDAlgorithm
sage: A = LeeBrickellISDAlgorithm(C, (0, 2))
sage: D = C.decoder("InformationSet", 2, algorithm=A); D
```

When passing an already constructed ISD algorithm, you can’t also pass parameters to the ISD algorithm when constructing the decoder:

```
sage: C.decoder("InformationSet", 2, algorithm=A, search_size=2)
Traceback (most recent call last):
...
ValueError: ISD algorithm arguments are not allowed when supplying a constructed ISD algorithm
```

We can also information-set decode non-binary codes:

```
sage: C = codes.GolayCode(GF(3))
sage: D = C.decoder("InformationSet", 2); D
```

There are two other ways to access this class:

```
sage: D = codes.decoders.LinearCodeInformationSetDecoder(C, 2); D
```

```
sage: from sage.coding.information_set_decoder import LinearCodeInformationSetDecoder
sage: D = LinearCodeInformationSetDecoder(C, 2); D
```

```
algorithm()  
        Return the ISD algorithm used by this ISD decoder.
```

EXAMPLES:
sage: C = codes.GolayCode(GF(2))
sage: D = C.decoder("InformationSet", (2,4), "Lee-Brickell")
sage: D.algorithm()
ISD Algorithm (Lee-Brickell) for [24, 12, 8] Extended Golay code over GF(2)
→decoding between 2 and 4 errors

**decode_to_code** *(r)*
Decodes a received word with respect to the associated code of this decoder.

**Warning:** If there is no codeword within the decoding radius of this decoder, this method may never terminate, or it may raise a `sage.coding.decoder.DecodingError` exception, depending on the ISD algorithm used.

**INPUT:**

- *r* – a vector in the ambient space of `decoder.Decoder.code()`.

**OUTPUT:** a codeword of `decoder.Decoder.code()`.

**EXAMPLES:**

```
sage: M = matrix(GF(2),
[1,0,0,0,0,0,1,0,1,0,1,1,0,0,1],
[0,1,0,0,0,1,1,1,1,0,0,0,1,1,1],
[0,0,1,0,0,0,0,1,0,1,1,1,1,1,0],
[0,0,0,1,0,0,1,0,1,0,0,1,1,0,1],
[0,0,0,0,1,0,0,0,1,0,1,0,1,0,1])
sage: C = LinearCode(M)
sage: c = C.random_element()
sage: Chan = channels.StaticErrorRateChannel(C.ambient_space(), 2)
sage: r = Chan(c)
sage: D = C.decoder('InformationSet', 2)
sage: c == D.decode_to_code(r)
True
```

Information-set decoding a non-binary code:

```
sage: C = codes.GolayCode(GF(3)); C
[12, 6, 6] Extended Golay code over GF(3)
sage: c = C.random_element()
sage: Chan = channels.StaticErrorRateChannel(C.ambient_space(), 2)
sage: r = Chan(c)
sage: D = C.decoder('InformationSet', 2)
sage: c == D.decode_to_code(r)
True
```

Let’s take a bigger example, for which syndrome decoding or nearest-neighbor decoding would be infeasible: the `[59,30]` Quadratic Residue code over $\mathbb{F}_3$ has true minimum distance 17, so we can correct 8 errors:

```
sage: C = codes.QuadraticResidueCode(59, GF(3))
sage: c = C.random_element()
sage: Chan = channels.StaticErrorRateChannel(C.ambient_space(), 2)
sage: r = Chan(c)
sage: D = C.decoder('InformationSet', 8)
sage: c == D.decode_to_code(r)  # long time
True
```
decoding_interval()  
A pair of integers specifying the interval of number of errors this decoder will attempt to correct. 
The interval includes both end values. 
EXAMPLES:

```
sage: C = codes.GolayCode(GF(2))
sage: D = C.decoder("InformationSet", 2)
sage: D.decoding_interval()
(0, 2)
```

decoding_radius()  
Return the maximal number of errors this decoder can decode. 
EXAMPLES:

```
sage: C = codes.GolayCode(GF(2))
sage: D = C.decoder("InformationSet", 2)
sage: D.decoding_radius()
2
```

static known_algorithms(dictionary=False)  
Return the list of ISD algorithms that Sage knows. 
Passing any of these to the constructor of sage.coding.information_set_decoder.LinearCodeInformationSetDecoder will make the ISD decoder use that algorithm. 
INPUT: 
• dictionary - optional. If set to True, return a dict mapping decoding algorithm name to its class. 
OUTPUT: a list of strings or a dict from string to ISD algorithm class. 
EXAMPLES:

```
sage: from sage.coding.information_set_decoder import LinearCodeInformationSetDecoder
sage: sorted(LinearCodeInformationSetDecoder.known_algorithms())
['Lee-Brickell']
```

7.6 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_asym) 
• David Joyner (2006-07): fixed dimension_upper_bound to return an integer, added example to elias_bound_asym. 
• " (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 set of size $q$. A subset $C$ of $V = F^n$ is called a code of length $n$. Often one considers the case where $F$ is a finite field, denoted by $\mathbb{F}_q$. Then $V$ is an $F$-vector space. 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 $F = \mathbb{F}_2$ 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 $v = (v_1, v_2, ..., v_n)$, $w = (w_1, w_2, ..., w_n)$ are elements of $V = F^n$ then we define

$$d(v, w) = |\{i \mid 1 \leq i \leq n, v_i \neq w_i\}|$$

to be the Hamming distance between $v$ and $w$. The function $d : V \times V \to N$ is called the Hamming metric. The weight of an element (in the Hamming metric) is $d(v, 0)$, where 0 is a distinguished element of $F$; in particular it is 0 if $F$ is a field. 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 $A_q(n, d)$ denote the largest $M$ such that there exists a $(n, M, d)$ code in $F^n$. Let $B_q(n, d)$ (also denoted $A^\text{lin}_q(n, d)$) 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 [HP2003] and [Lin1999].

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 $q$. 

180 Chapter 7. Other modules
• `gilbert_lower_bound(n,q,d)`, a lower bound for number of elements in the largest code of min distance $d$ in $F_q^n$.
• `gv_info_rate(n, delta, q)`, $\log_q(\text{GLB})/n$, where GLB is the Gilbert lower bound and $\delta = d/n$.
• `gv_bound_asympt(delta, q)`, asymptotic analog of Gilbert lower bound.
• `plotkin_upper_bound(n,q,d)`
• `plotkin_bound_asympt(delta, q)`, asymptotic analog of Plotkin bound.
• `griesmer_upper_bound(n,q,d)`
• `elias_upper_bound(n,q,d)`
• `elias_bound_asympt(delta, q)`, asymptotic analog of Elias bound.
• `hamming_upper_bound(n,q,d)`
• `hamming_bound_asympt(delta, q)`, asymptotic analog of Hamming bound.
• `singleton_upper_bound(n,q,d)`
• `singleton_bound_asympt(delta, q)`, asymptotic analog of Singleton bound.
• `mrrw1_bound_asympt(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_asympt(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`)  

Returns an upper bound on the number of codewords in a (possibly non-linear) code.

This function computes the minimum value of the upper bounds 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:

```python
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)
```

(continues on next page)
sage.coding.code_bounds.dimension_upper_bound(n, d, q, algorithm=None)

Return an upper bound for the dimension of a linear code.

Return 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_asympt(\( \delta \), \( q \))

The asymptotic Elias bound for the information rate.

This only makes sense when \( 0 < \delta < 1 - 1/q \).

EXAMPLES:

sage: codes.bounds.elias_bound_asympt(1/4, 2)
0.39912396330...

sage.coding.code_bounds.elias_upper_bound(n, q, d, algorithm=None)

Returns the Elias upper bound.

Returns the Elias upper bound for number of elements in the largest code of minimum distance \( d \) in \( \mathbb{F}^n_q \), cf. [HP2003]. If the method is “gap”, it 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:
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)
   0.012986862055...
   sage: entropy_inverse(1)
   0.5
   sage: entropy_inverse(0, 3)
   0
   sage: entropy_inverse(1, 3)
   0.6666666666666667
```

```sage
coding.code_bounds.gilbert_lower_bound(n, q, d)
   Returns the Gilbert-Varshamov lower bound.

   Returns the Gilbert-Varshamov lower bound for number of elements in a largest code of minimum distance d in F_q^n. See Wikipedia article Gilbert-Varshamov_bound

   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.

   Returns the Griesmer upper bound for the number of elements in a largest linear code of minimum distance d in F_q^n, cf. [HP2003]. If the method is “gap”, it wraps GAP’s UpperBoundGriesmer.
```

7.6. Bounds for Parameters of Codes 183
The bound states:

\[ n \geq \sum_{i=0}^{k-1} \lceil \frac{d}{q} \rceil^i. \]

**EXAMPLES:**

The bound is reached for the ternary Golay codes:

```python
sage: codes.bounds.griesmer_upper_bound(12, 3, 6)
sage: codes.bounds.griesmer_upper_bound(11, 3, 5)
```

```python
sage: codes.bounds.griesmer_upper_bound(10, 2, 3)
sage: codes.bounds.griesmer_upper_bound(10, 2, 3, algorithm="gap") # optional - gap_packages (Guava package)
```

**sage.coding.code_bounds.gv_bound_asym** \((\delta, q)\)

The asymptotic Gilbert-Varshamov bound for the information rate, \(R\).

**EXAMPLES:**

```python
sage: RDF(codes.bounds.gv_bound_asym(1/4, 2))
sage: f = lambda x: codes.bounds.gv_bound_asym(x, 2)
sage: plot(f, 0, 1)
```

**sage.coding.code_bounds.gv_info_rate** \((n, \delta, q)\)

The Gilbert-Varshamov lower bound for information rate.

The Gilbert-Varshamov lower bound for information rate of a \(q\)-ary code of length \(n\) and minimum distance \(n\delta\).

**EXAMPLES:**

```python
sage: RDF(codes.bounds.gv_info_rate(100, 1/4, 3)) # abs tol 1e-15
```

**sage.coding.code_bounds.hamming_bound_asym** \((\delta, q)\)

The asymptotic Hamming bound for the information rate.

**EXAMPLES:**

```python
sage: RDF(codes.bounds.hamming_bound_asym(1/4, 2))
sage: f = lambda x: codes.bounds.hamming_bound_asym(x, 2)
sage: plot(f, 0, 1)
```

**sage.coding.code_bounds.hamming_upper_bound** \((n, q, d)\)

Returns the Hamming upper bound.

Returns the Hamming upper bound for number of elements in the largest code of length \(n\) and minimum distance \(d\) over alphabet of size \(q\).

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 an alphabet of size \(q\). The Hamming bound is obtained by dividing the
contents of the entire Hamming space $q^n$ by the contents of a ball with radius $\text{floor}((d - 1)/2)$. 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. See e.g. [HP2003].

EXAMPLES:

```python
sage: codes.bounds.hamming_upper_bound(10,2,3)
93
```

```python
sage.coding.code_bounds.mrrw1_bound_asymptotic(delta, q)
```

The first asymptotic McEliese-Rumsey-Rodemich-Welsh bound.

This only makes sense when $0 < \delta < 1 - 1/q$.

EXAMPLES:

```python
sage: codes.bounds.mrrw1_bound_asymptotic(1/4,2)  # abs tol 4e-16
0.3545789026652697
```

```python
sage.coding.code_bounds.plotkin_bound_asymptotic(delta, q)
```

The asymptotic Plotkin bound for the information rate.

This only makes sense when $0 < \delta < 1 - 1/q$.

EXAMPLES:

```python
sage: codes.bounds.plotkin_bound_asymptotic(1/4,2)
1/2
```

```python
sage.coding.code_bounds.plotkin_upper_bound(n, q, d, algorithm=None)
```

Returns the Plotkin upper bound.

Returns the Plotkin upper bound for the number of elements in a largest code of minimum distance $d$ in $\mathbb{F}_q^n$. More precisely this is a generalization of Plotkin’s result for $q = 2$ to bigger $q$ due to Berlekamp.

The algorithm="gap" option wraps Guava’s UpperBoundPlotkin.

EXAMPLES:

```python
sage: codes.bounds.plotkin_upper_bound(10,2,3)
192
```

```python
sage: codes.bounds.plotkin_upper_bound(10,2,3,algorithm="gap")  # optional - gap_packages (Guava package)
192
```

```python
sage.coding.code_bounds.singleton_bound_asymptotic(delta, q)
```

The asymptotic Singleton bound for the information rate.

EXAMPLES:

```python
sage: codes.bounds.singleton_bound_asymptotic(1/4,2)
3/4
```

```python
f = lambda x: codes.bounds.singleton_bound_asymptotic(x,2)
sage: plot(f,0,1)
```

Graphics object consisting of 1 graphics primitive

7.6. Bounds for Parameters of Codes

185
sage.coding.code_bounds.singleton_upper_bound(n, q, d)
Returns the Singleton upper bound.
Returns the Singleton upper bound for number of elements in a largest code of minimum distance \(d\) in \(F_q^n\).
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 the number of elements in a Hamming ball.
Returns the number of elements in a Hamming ball of radius \(r\) in \(F_q^n\).

EXAMPLES:

```
sage: codes.bounds.volume_hamming(10,2,3)
176
```

### 7.7 Delsarte, a.k.a. Linear Programming (LP), upper bounds

This module provides LP upper bounds for the parameters of codes, introduced in [De1973].
The exact LP solver PPL is used by default, ensuring that no rounding/overflow problems occur.

AUTHORS:


REFERENCES:

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, $q = q_{\text{base}}^m$ and the code is linear over $F_{\text{base}}$.

• **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:

```python
sage: codes.bounds.delsarte_bound_additive_hamming_space(11, 6, 2)
3
```

```python
sage: a,p,val = codes.bounds.delsarte_bound_additive_hamming_space(11, 6, 2, return_data=True)
sage: [j for i,j in p.get_values(a).items()]
[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:

```python
sage: codes.bounds.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:

```python
sage: codes.bounds.delsarte_bound_additive_hamming_space(11, 3, 4, q_base=2)
16
```

Such a $d_{\text{star}}$ is not possible:

```python
sage: codes.bounds.delsarte_bound_additive_hamming_space(11, 3, 4, d_star=9)
Solver exception: PPL : There is no feasible solution
False
```

Find the Delsarte bound [De1973] on codes in Hamming space $H_{\mathbb{Q}}^n$ 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: codes.bounds.delsarte_bound_hamming_space(11, 6, 2)
12
sage: a, p, val = codes.bounds.delsarte_bound_hamming_space(11, 6, 2, return_data=True)
sage: [j for i,j in p.get_values(a).items()]
[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 extended binary Golay code:

```
sage: a, p, x = codes.bounds.delsarte_bound_hamming_space(24, 8, 2, return_data=True)
sage: x
4096
sage: [j for i,j in p.get_values(a).items()]
[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: codes.bounds.delsarte_bound_hamming_space(11, 3, 4)
327680/3
```

An improvement of a known upper bound (150) from https://www.win.tue.nl/~aeb/codes/binary-1.html

```
sage: a, p, x = codes.bounds.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).items()] # 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, 0]
```

Note that a usual LP, without integer variables, won’t do the trick

```
sage: codes.bounds.delsarte_bound_hamming_space(23,10,2).n(20)
151.86
```

Such an input is invalid:

```
sage: codes.bounds.delsarte_bound_hamming_space(11,3,-4)
Solver exception: PPL : There is no feasible solution
False
```

---

`sage.coding.delsarte_bounds.krawtchouk`(*n*, *q*, *l*, *x*, check=True)

Compute $K^{n,q}_{l-1}(x)$, the Krawtchouk (a.k.a. Kravchuk) polynomial.

See Wikipedia article Kravchuk polynomials.

It is defined by the generating function

$$ (1 + (q - 1)z)^{n-x}(1 - z)^x = \sum_l K^n_l(x)z^l $$
and is equal to

\[ K_n^{n,q}(x) = \sum_{j=0}^{l} (-1)^j (q-1)^{(l-j)} \binom{x}{j} \binom{n-x}{l-j}, \]

INPUT:

- \( n, q, x \) – arbitrary numbers
- \( l \) – a nonnegative integer
- \( \text{check} \) – check the input for correctness. True by default. Otherwise, pass it as it is. Use \( \text{check}=\text{False} \) at your own risk.

EXAMPLES:

```
sage: codes.bounds.krawtchouk(24,2,5,4)
2224
sage: codes.bounds.krawtchouk(12300,4,5,6)
567785569973042442072
```
CHAPTER EIGHT

INDICES AND TABLES

- Index
- Module Index
- Search Page


[R06] Ron Roth, Introduction to Coding Theory, Cambridge University Press, 2006


sage.coding.bch, 104
sage.coding.binary_code, 118
sage.coding.bounds_catalog, 60
sage.coding.channel_constructions, 42
sage.coding.channels_catalog, 57
sage.coding.code_bounds, 179
sage.coding.code_constructions, 108
sage.coding.codecan.autgroup_can_label, 144
sage.coding.codecan.codecan, 141
sage.coding.codes_catalog, 57
sage.coding.cyclic_code, 95
sage.coding.databases, 60
sage.coding.decoder, 49
sage.coding.decoders_catalog, 58
sage.coding.delsarte_bounds, 186
sage.coding.encoder, 52
sage.coding.encoders_catalog, 59
sage.coding.extended_code, 136
sage.coding.golay_code, 83
sage.coding.grs, 67
sage.coding.guava, 115
sage.coding.guruswami_sudan.gs_decoder, 159
sage.coding.guruswami_sudan.interpolation, 166
sage.coding.guruswami_sudan.utils, 169
sage.coding.hamming_code, 82
sage.coding.information_set_decoder, 170
sage.coding.linear_code, 1
sage.coding.parity_check_code, 85
sage.coding.punctured_code, 132
sage.coding.reed_muller_code, 88
sage.coding.relativeFiniteField_extension, 155
sage.coding.self_dual_codes, 116
sage.coding.source_coding.huffman, 149
sage.coding.subfield_subcode, 129
sage.coding.two_weight_db, 64
INDEX

A

absolute_field() (sage.coding.relative_finite_field_extension.RelativeFiniteFieldExtension method), 156
absolute_field_basis() (sage.coding.relative_finite_field_extension.RelativeFiniteFieldExtension method), 156
absolute_field_degree() (sage.coding.relative_finite_field_extension.RelativeFiniteFieldExtension method), 156
absolute_field_representation() (sage.coding.relative_finite_field_extension.RelativeFiniteFieldExtension method), 156
AbstractLinearCode (class in sage.coding.linear_code), 3
add_decoder() (sage.coding.linear_code.AbstractLinearCode method), 4
add_encoder() (sage.coding.linear_code.AbstractLinearCode method), 5
alekhnovich_root_finder() (in module sage.coding.guruswami_sudan.gs_decoder), 166
algorithm() (sage.coding.information_set_decoder.LinearCodeInformationSetDecoder method), 177
ambient_space() (sage.coding.linear_code.AbstractLinearCode method), 6
apply_permutation() (sage.coding.binary_code.BinaryCode method), 119
assmus_mattson_designs() (sage.coding.linear_code.AbstractLinearCode method), 6
automorphism_group_gens() (sage.coding.linear_code.AbstractLinearCode method), 7

B

base_field() (sage.coding.linear_code.AbstractLinearCode method), 8
basis() (sage.coding.linear_code.AbstractLinearCode method), 8
bch_bound() (in module sage.coding.cyclic_code), 103
bch_bound() (sage.coding.cyclic_code.CyclicCode method), 96
bch_code() (sage.coding.cyclic_code.CyclicCodeSurroundingBCHDecoder method), 101
bch_decoder() (sage.coding.cyclic_code.CyclicCodeSurroundingBCHDecoder method), 101
bch_to_grs() (sage.coding.bch.BCHCode method), 105
bch_word_to_grs() (sage.coding.bch.BCHUnderlyingGRSDecoder method), 106
BCHCode (class in sage.coding.bch), 104
BCHUnderlyingGRSDecoder (class in sage.coding.bch), 106
best_linear_code_in_codetables_dot_de() (in module sage.coding.databases), 60
best_linear_code_in_guava() (in module sage.coding.databases), 61
BinaryCode (class in sage.coding.binary_code), 118
BinaryCodeClassifier (class in sage.coding.binary_code), 122
BinaryReedMullerCode (class in sage.coding.reed_muller_code), 88
binomial_moment() (sage.coding.linear_code.AbstractLinearCode method), 9
bounds_on_minimum_distance_in_guava() (in module sage.coding.databases), 62

C

calibrate() (sage.coding.information_set_decoder.InformationSetAlgorithm method), 171
calibrate() (sage.coding.information_set_decoder.LeeBrickellISDAlgorithm method), 174
canonical_representative() (sage.coding.linear_code.AbstractLinearCode method), 9
cardinality() (sage.coding.linear_code.AbstractLinearCode method), 10
cast_into_relative_field() (sage.coding.relative_finite_field_extension.RelativeFiniteFieldExtension method), 156
Channel (class in sage.coding.channel_constructions), 42
characteristic() (sage.coding.linear_code.AbstractLinearCode method), 10
characteristic_polynomial() (sage.coding.linear_code.AbstractLinearCode method), 10
check_polynomial() (sage.coding.cyclic_code.CyclicCode method), 97
chinen_polynomial() (sage.coding.linear_code.AbstractLinearCode method), 10
cmp() (sage.coding.binary_code.PartitionStack method), 124
code() (sage.coding.decoder.Decoder method), 49
code() (sage.coding.encoder.Encoder method), 53
code() (sage.coding.information_set_decoder.InformationSetAlgorithm method), 172
codesize_upper_bound() (in module sage.coding.code_bounds), 181
column_blocks() (sage.coding.codecan.codecan.InnerGroup method), 142
column_multipliers() (sage.coding.grs.GeneralizedReedSolomonCode method), 78
connected_encoder() (sage.coding.decoder.Decoder method), 49
covering_radius() (sage.coding.golay_code.GolayCode method), 83
covering_radius() (sage.coding.grs.GeneralizedReedSolomonCode method), 78
covering_radius() (sage.coding.linear_code.AbstractLinearCode method), 11
CyclicCode (class in sage.coding.cyclic_code), 95
CyclicCodePolynomialEncoder (class in sage.coding.cyclic_code), 99
CyclicCodeSurroundingBCHDecoder (class in sage.coding.cyclic_code), 100
CyclicCodeVectorEncoder (class in sage.coding.cyclic_code), 101

D
decode() (sage.coding.information_set_decoder.InformationSetAlgorithm method), 172
decode() (sage.coding.information_set_decoder.LeeBrickellISDAlgorithm method), 175
decode() (sage.coding.source_coding.huffman.Huffman method), 151
decode_to_code() (sage.coding.bch.BCHUnderlyingGRSDecoder method), 106
decode_to_code() (sage.coding.cyclic_code.CyclicCodeSurroundingBCHDecoder method), 101
decode_to_code() (sage.coding.decoder.Decoder method), 50
decode_to_code() (sage.coding.extended_code.ExtendedCodeOriginalCodeDecoder method), 138
decode_to_code() (sage.coding.grs.GRSBerlekampWelchDecoder method), 68
decode_to_code() (sage.coding.grs.GRSGaoDecoder method), 75
decode_to_code() (sage.coding.grs.GRSKeyEquationSyndromeDecoder method), 76
decode_to_code() (sage.coding.guruswami_sudan.gs_decoder.GSGuruswamiSudanDecoder method), 161
decode_to_code() (sage.coding.information_set_decoder.LinearCodeInformationSetDecoder method), 178
decode_to_code() (sage.coding.linear_code.AbstractLinearCode method), 11
decode_to_code() (sage.coding.linear_code.LinearCodeNearestNeighborDecoder method), 34
decode_to_code() (sage.coding.linear_code.LinearCodeSyndromeDecoder method), 37
decode_to_code() (sage.coding.punctured_code.PuncturedCodeOriginalCodeDecoder method), 135
decode_to_message() (sage.coding.subfield_subcode.SubfieldSubcodeOriginalCodeDecoder method), 131
decode_to_message() (sage.coding.decoder.Decoder method), 50
decode_to_message() (sage.coding.grs.GeneralizedReedSolomonCode method), 79
decode_to_message() (sage.coding.grs.GRSBerlekampWelchDecoder method), 68
decode_to_message() (sage.coding.grs.GRSErrorErasureDecoder method), 70
decode_to_message() (sage.coding.grs.GRSGaoDecoder method), 75
decode_to_message() (sage.coding.grs.GRSKeyEquationSyndromeDecoder method), 77
decode_to_message() (sage.coding.guruswami_sudan.gs_decoder.GSGuruswamiSudanDecoder method), 161
decode_to_message() (sage.coding.linear_code.AbstractLinearCode method), 12
Decoder (class in sage.coding.decoder), 49
decoder() (sage.coding.linear_code.AbstractLinearCode method), 12
decoder_type() (sage.coding.decoder.Decoder class method), 50
decoders_available() (sage.coding.linear_code.AbstractLinearCode method), 13
decoding_interval() (sage.coding.information_set_decoder.InformationSetAlgorithm method), 172
decoding_interval() (sage.coding.information_set_decoder.LinearCodeInformationSetDecoder method), 178
decoding_radius() (sage.coding.bch.BCHUnderlyingGRSDecoder method), 107
decoding_radius() (sage.coding.cyclic_code.CyclicCodeSurroundingBCHDecoder method), 101
decoding_radius() (sage.coding.decoder.Decoder method), 51
decoding_radius() (sage.coding.extended_code.ExtendedCodeOriginalCodeDecoder method), 139
decoding_radius() (sage.coding.grs.GRSErasureDecoder method), 70
decoding_radius() (sage.coding.grs.GRSKeyEquationSyndromeDecoder method), 77
decoding_radius() (sage.coding.information_set_decoder.LinearCodeInformationSetDecoder method), 179
decoding_radius() (sage.coding.linear_code.LinearCodeNearestNeighborDecoder method), 35
decoding_radius() (sage.coding.linear_code.LinearCodeSyndromeDecoder method), 37
decoding_radius() (sage.coding.punctured_code.PuncturedCodeOriginalCodeDecoder method), 135
decoding_radius() (sage.coding.subfield_subcode.SubfieldSubcodeOriginalCodeDecoder method), 131
DecodingError, 52
defining_set() (sage.coding.cyclic_code.CyclicCode method), 97
delsarte_bound_additive_hamming_space() (in module sage.coding.delsarte_bounds), 186
delsarte_bound_hamming_space() (in module sage.coding.delsarte_bounds), 187
designed_distance() (sage.coding.bch.BCHCode method), 105
dimension() (sage.coding.linear_code.AbstractLinearCode method), 13
dimension() (sage.coding.punctured_code.PuncturedCode method), 132
dimension() (sage.coding.subfield_subcode.SubfieldSubcode method), 129
dimension_lower_bound() (sage.coding.subfield_subcode.SubfieldSubcode method), 129
dimension_upper_bound() (in module sage.coding.code_bounds), 182
dimension_upper_bound() (sage.coding.subfield_subcode.SubfieldSubcode method), 130
direct_sum() (sage.coding.linear_code.AbstractLinearCode method), 13
divisor() (sage.coding.linear_code.AbstractLinearCode method), 14
DuadicCodeEvenPair() (in module sage.coding.code_constructions), 109
DuadicCodeOddPair() (in module sage.coding.code_constructions), 109
dual_code() (sage.coding.golay_code.GolayCode method), 84
dual_code() (sage.coding.grs.GeneralizedReedSolomonCode method), 79
dual_code() (sage.coding.linear_code.AbstractLinearCode method), 14
encode() (sage.coding.parity_check_code.ParityCheckCodeStraightforwardEncoder method), 87
encode() (sage.coding.punctured_code.PuncturedCode method), 132
encode() (sage.coding.reed_muller_code.ReedMullerPolynomialEncoder method), 92
encode() (sage.coding.source_coding.huffman.Huffman method), 151
Encoder (class in sage.coding.encoder), 52
encoder() (sage.coding.linear_code.AbstractLinearCode method), 15
encoders_available() (sage.coding.linear_code.AbstractLinearCode method), 16
encoding_table() (sage.coding.source_coding.huffman.Huffman method), 152
EncodingError, 55
entropy() (in module sage.coding.code_bounds), 182
entropy_inverse() (in module sage.coding.code_bounds), 183
error_probability() (sage.coding.channel_constructions.QarySymmetricChannel method), 46
ErrorErasureChannel (class in sage.coding.channel_constructions), 44
evaluation_points() (sage.coding.grs.GeneralizedReedSolomonCode method), 79
extended_code() (sage.coding.linear_code.AbstractLinearCode method), 16
ExtendedCode (class in sage.coding.extended_code), 137
ExtendedCodeExtendedMatrixEncoder (class in sage.coding.extended_code), 137
ExtendedCodeOriginalCodeDecoder (class in sage.coding.extended_code), 138
ExtendedQuadraticResidueCode() (in module sage.coding.code_constructions), 110
extension_degree() (sage.coding.relative_finite_field_extension.RelativeFiniteFieldExtension method), 157

d field_embedding() (sage.coding.cyclic_code.CyclicCode method), 98
find_generator_polynomial() (in module sage.coding.cyclic_code), 104
format_interval() (in module sage.coding.channel_constructions), 48
frequency_table() (in module sage.coding.source_coding.huffman), 153
from_parity_check_matrix() (in module sage.coding.code_constructions), 113

G
galois_closure() (sage.coding.linear_code.AbstractLinearCode method), 16
GeneralizedReedSolomonCode (class in sage.coding.grs), 77
generate_children() (sage.coding.binary_code.BinaryCodeClassifier method), 122
generator_matrix() (sage.coding.cyclic_code.CyclicCodeVectorEncoder method), 102
generator_matrix() (sage.coding.encoder.Encoder method), 54
generator_matrix() (sage.coding.extended_code.ExtendedCodeExtendedMatrixEncoder method), 138
generator_matrix() (sage.coding.extended_code.LinearCodeGeneratorMatrixEncoder method), 34
generator_matrix() (sage.coding.linear_code.LinearCodeSystematicEncoder method), 40
generator_matrix() (sage.coding.parity_check_code.ParityCheckCodeGeneratorMatrixEncoder method), 86
generator_matrix() (sage.coding.punctured_code.PuncturedCodePuncturedMatrixEncoder method), 136
generator_matrix() (sage.coding.reed_muller_code.ReedMullerVectorEncoder method), 94
generator_polynomial() (sage.coding.cyclic_code.CyclicCode method), 98
gens() (sage.coding.linear_code.AbstractLinearCode method), 17
genus() (sage.coding.linear_code.AbstractLinearCode method), 17
get_autom_gens() (sage.coding.codecan.autgroup_can_label.LinearCodeAutGroupCanLabel method), 147
get_autom_order() (sage.coding.codecan.autgroup_can_label.LinearCodeAutGroupCanLabel method), 147
get_autom_order_inner_stabilizer() (sage.coding.codecan.codecan.PartitionRefinementLinearCode method), 143
get_canonical_form() (sage.coding.codecan.autgroup_can_label.LinearCodeAutGroupCanLabel method), 147
get_canonical_form() (sage.coding.codecan.codecan.PartitionRefinementLinearCode method), 144
get_frob_pow() (sage.coding.codecan.codecan.InnerGroup method), 143
get_PGammaL_gens() (sage.coding.codecan.autgroup_can_label.LinearCodeAutGroupCanLabel method), 146
get_PGammaL_order() (sage.coding.codecan.autgroup_can_label.LinearCodeAutGroupCanLabel method), 147
generate_transporter() (sage.coding.codecan.codecan.PartitionRefinementLinearCode method), 144
get_transporter() (sage.coding.codecan.codecan.PartitionRefinementLinearCode method), 144
gilbert_lower_bound() (in module sage.coding.code_bounds), 183
gilt() (in module sage.coding.guruswami_sudan.utils), 169
GolayCode (class in sage.coding.golay_code), 83
griesmer_upper_bound() (in module sage.coding.code_bounds), 183
grs_code() (sage.coding.bch.BCHUnderlyingGRSDecoder method), 107
grs_decoder() (sage.coding.bch.BCHUnderlyingGRSDecoder method), 108
grs_word_to_bch() (sage.coding.bch.BCHUnderlyingGRSDecoder method), 108
GRSBERLEKAMPWELCHDECODER (class in sage.coding.grs), 67
GRSErrortErasureDecoder (class in sage.coding.grs), 69
GRSEvaluationPolynomialEncoder (class in sage.coding.grs), 71
GRSEvaluationVectorEncoder (class in sage.coding.grs), 73
GRSGaoDecoder (class in sage.coding.grs), 74
GRSGuruswamiSudanDecoder (class in sage.coding.guruswami_sudan.gs_decoder), 159
GRSKeyEquationSyndromeDecoder (class in sage.coding.grs), 76
gs_interpolation_lee_osullivan() (in module sage.coding.guruswami_sudan.interpolation), 166
gs_interpolation_final() (in module sage.coding.guruswami_sudan.interpolation), 167
gs_satisfactory() (sage.coding.guruswami_sudan.gs_decoder.GRSGuruswamiSudanDecoder static method), 162
guruswami_sudan_decoding_radius() (sage.coding.guruswami_sudan.gs_decoder.GRSGuruswamiSudanDecoder static method), 163
gv_bound_asympt() (in module sage.coding.code_bounds), 184
gv_info_rate() (in module sage.coding.code_bounds), 184

H
hamming_bound_asympt() (in module sage.coding.code_bounds), 184
hamming_upper_bound() (in module sage.coding.code_bounds), 184
HammingCode (class in sage.coding.hamming_code), 82
Huffman (class in sage.coding.source_coding.huffman), 149

I
information_set() (sage.coding.linear_code.AbstractLinearCode method), 17
InformationSetAlgorithm (class in sage.coding.information_set_decoder), 171
InnerGroup (class in sage.coding.codecan.codecan), 142
input_space() (sage.coding.channel_constructions.Channel method), 42
input_space() (sage.coding.decoder.Decoder method), 52
interpolation_algorithm() (sage.coding.guruswami_sudan.gs_decoder.GRSGuruswamiSudanDecoder method), 164
is_galois_closed() (sage.coding.linear_code.AbstractLinearCode method), 18
is_generalized() (sage.coding.grs.GeneralizedReedSolomonCode method), 79
is_in_relative_field() (sage.coding.relative_finite_field_extension.RelativeFiniteFieldExtension method), 157
is_information_set() (sage.coding.linear_code.AbstractLinearCode method), 18
is_permutation_automorphism() (sage.coding.linear_code.AbstractLinearCode method), 18
is_permutation_equivalent() (sage.coding.linear_code.AbstractLinearCode method), 19
is_projective() (sage.coding.linear_code.AbstractLinearCode method), 19
is_self_dual() (sage.coding.linear_code.AbstractLinearCode method), 19
is_self_orthogonal() (sage.coding.linear_code.AbstractLinearCode method), 20
is_subcode() (sage.coding.linear_code.AbstractLinearCode method), 20

J
johnson_radius() (in module sage.coding.guruswami_sudan.utils), 169
jump_size() (sage.coding.bch.BCHCode method), 106

K
known_algorithms() (sage.coding.information_set_decoder.LinearCodeInformationSetDecoder static method), 179
krawtchouk() (in module sage.coding.delsarte_bounds), 188

L
lee_osullivan_module() (in module sage.coding.guruswami_sudan.interpolation), 168
LeeBrickellISDAlgorithm (class in sage.coding.information_set_decoder), 173
length() (sage.coding.linear_code.AbstractLinearCode method), 20
ligt() (in module sage.coding.guruswami_sudan.utils), 169
LinearCode (class in sage.coding.linear_code), 32
LinearCodeAutGroupCanLabel (class in sage.coding.codecan.autgroup_can_label), 146
LinearCodeGeneratorMatrixEncoder (class in sage.coding.linear_code), 34
LinearCodeInformationSetDecoder (class in sage.coding.information_set_decoder), 175
LinearCodeNearestNeighborDecoder (class in sage.coding.linear_code), 34
LinearCodeSyndromeDecoder (class in sage.coding.linear_code), 35
LinearCodeSystematicEncoder (class in sage.coding.linear_code), 38
list() (sage.coding.linear_code.AbstractLinearCode method), 21
list_size() (sage.coding.guruswami_sudan.gs_decoder.GRSGuruswamiSudanDecoder method), 164

M
matrix() (sage.coding.binary_code.BinaryCode method), 120
maximum_error_weight() (sage.coding.linear_code.LinearCodeSyndromeDecoder method), 37
message_space() (sage.coding.cyclic_code.CyclicCodePolynomialEncoder method), 100
message_space() (sage.coding.cyclic_code.CyclicCodeVectorEncoder method), 102
message_space() (sage.coding.decoder.Decoder method), 52
message_space() (sage.coding.encoder.Encoder method), 54
message_space() (sage.coding.grs.GRSEvaluationPolynomialEncoder method), 72
message_space() (sage.coding.parity_check_code.JuntaCodeStraightforwardEncoder method), 87
message_space() (sage.coding.parity_check_code.ParityCheckCodePolynomialEncoder method), 92
minimum_distance() (sage.coding.golay_code.GolayCode method), 84
minimum_distance() (sage.coding.grs.GeneralizedReedSolomonCode method), 80
minimum_distance() (sage.coding.hamming_code.HammingCode method), 82
minimum_distance() (sage.coding.linear_code.AbstractLinearCode method), 21
minimum_distance() (sage.coding.parity_check_code.ParityCheckCode method), 86
minimum_distance() (sage.coding.reed_muller_code.BinaryReedMullerCode method), 88
minimum_distance() (sage.coding.reed_muller_code.QAryReedMullerCode method), 89
module_composition_factors() (sage.coding.linear_code.AbstractLinearCode method), 21
mrrw1_bound_asympt() (in module sage.coding.code_bounds), 185
multiplicity() (sage.coding.guruswami_sudan.gs_decoder.GRSGuruswamiSudanDecoder method), 164
multipliers_product() (sage.coding.grs.GeneralizedReedSolomonCode method), 80
n_k_params() (in module sage.coding.guruswami_sudan.gs_decoder), 166
name() (sage.coding.information_set_decoder.InformationSetAlgorithm method), 172
number_erasures() (sage.coding.channel_constructions.ErrorErasureChannel method), 44
number_errors() (sage.coding.channel_constructions.ErrorErasureChannel method), 44
number_errors() (sage.coding.channel_constructions.StaticErrorRateChannel method), 47
number_of_variables() (sage.coding.reed_muller_code.BinaryReedMullerCode method), 88
number_of_variables() (sage.coding.reed_muller_code.QAryReedMullerCode method), 89
offset() (sage.coding.bch.BCHCode method), 106
 OrbitPartition (class in sage.coding.binary_code), 123
order() (sage.coding.reed_muller_code.BinaryReedMullerCode method), 89
order() (sage.coding.reed_muller_code.QAryReedMullerCode method), 90
original_code() (sage.coding.extended_code.ExtendedCode method), 137
original_code() (sage.coding.punctured_code.PuncturedCode method), 133
original_code() (sage.coding.subfield_subcode.SubfieldSubcode method), 130
original_decoder() (sage.coding.extended_code.ExtendedCodeOriginalCodeDecoder method), 139
original_decoder() (sage.coding.punctured_code.PuncturedCodeOriginalCodeDecoder method), 136
output_space() (sage.coding.channel_constructions.Channel method), 43
parameters() (sage.coding.guruswami_sudan.gs_decoder.GRSGuruswamiSudanDecoder method), 164
parameters() (sage.coding.information_set_decoder.InformationSetAlgorithm method), 173
parameters_given_tau() (sage.coding.guruswami_sudan.gs_decoder.GRSGuruswamiSudanDecoder static method), 165
parity_check_matrix() (sage.coding.cyclic_code.CyclicCode method), 98
parity_check_matrix() (sage.coding.extended_code.ExtendedCode method), 137
parity_check_matrix() (sage.coding.golay_code.GolayCode method), 85
parity_check_matrix() (sage.coding.grs.GeneralizedReedSolomonCode method), 80
parity_check_matrix() (sage.coding.hamming_code.HammingCode method), 83
parity_check_matrix() (sage.coding.linear_code.AbstractLinearCode method), 22
parity_check_matrix() (sage.coding.subfield_subcode.SubfieldSubcode method), 130
parity_column_multipliers() (sage.coding.grs.GeneralizedReedSolomonCode method), 80
ParityCheckCode (class in sage.coding.parity_check_code), 85
ParityCheckCodeGeneratorMatrixEncoder (class in sage.coding.parity_check_code), 86
ParityCheckCodeStraightforwardEncoder (class in sage.coding.parity_check_code), 86
PartitionRefinementLinearCode (class in sage.coding.codecan.codecan), 143
PartitionStack (class in sage.coding.binary_code), 123
permutation_action() (in module sage.coding.code_constructions), 113
permutation_automorphism_group() (sage.coding.linear_code.AbstractLinearCode method), 22
permuted_code() (sage.coding.linear_code.AbstractLinearCode method), 24
plotkin_bound_asympt() (in module sage.coding.code_bounds), 185
plotkin_upper_bound() (in module sage.coding.code_bounds), 185
points() (sage.coding.reed_muller_code.ReedMullerPolynomialEncoder method), 92
points() (sage.coding.reed_muller_code.ReedMullerVectorEncoder method), 94
polynomial_ring() (sage.coding.grs.GRSEvaluationPolynomialEncoder method), 72
polynomial_ring() (sage.coding.reed_muller_code.ReedMullerPolynomialEncoder method), 93
polynomial_to_list() (in module sage.coding.guruswami_sudan.utils), 170
prime_field() (sage.coding.relative_finite_field_extension.RelativeFiniteFieldExtension method), 158
primitive_root() (sage.coding.cyclic_code.CyclicCode method), 98
print_basis() (sage.coding.binary_code.PartialReduction method), 124
print_data() (sage.coding.binary_code.BinaryCode method), 120
print_data() (sage.coding.binary_code.PartialReduction method), 125
probability_of_at_most_t_errors() (sage.coding.channel_constructions.QarySymmetricChannel method), 46
probability_of_exactly_t_errors() (sage.coding.channel_constructions.QarySymmetricChannel method), 46
punctured() (sage.coding.linear_code.AbstractLinearCode method), 24
punctured_positions() (sage.coding.punctured_code.PuncturedCode method), 133
PuncturedCode (class in sage.coding.punctured_code), 132
PuncturedCodeOriginalCodeDecoder (class in sage.coding.punctured_code), 134
PuncturedCodePuncturedMatrixEncoder (class in sage.coding.punctured_code), 136
put_in_canonical_form() (sage.coding.binary_code.BinaryCodeClassifier method), 123
put_in_std_form() (sage.coding.binary_code.BinaryCode method), 121

Q
QaryReedMullerCode (class in sage.coding.reed_muller_code), 89
QarySymmetricChannel (class in sage.coding.channel_constructions), 45
QuadraticResidueCode() (in module sage.coding.code_constructions), 110
QuadraticResidueCodeEvenPair() (in module sage.coding.code_constructions), 111
QuadraticResidueCodeOddPair() (in module sage.coding.code_constructions), 111
QuasiQuadraticResidueCode() (in module sage.coding.guava), 115

R
random_element() (sage.coding.extended_code.ExtendedCode method), 137
random_element() (sage.coding.linear_code.AbstractLinearCode method), 24
random_element() (sage.coding.punctured_code.PuncturedCode method), 133
random_error_vector() (in module sage.coding.channel_constructions), 48
random_linear_code() (in module sage.coding.code_constructions), 114
RandomLinearCodeGuava() (in module sage.coding.guava), 116
rate() (sage.coding.linear_code.AbstractLinearCode method), 24
redundancy_matrix() (sage.coding.linear_code.AbstractLinearCode method), 25
ReedMullerCode() (in module sage.coding.reed_muller_code), 90
ReedMullerPolynomialEncoder (class in sage.coding.reed_muller_code), 91
ReedMullerVectorEncoder (class in sage.coding.reed_muller_code), 93
ReedSolomonCode() (in module sage.coding.grs), 81
relative_distance() (sage.coding.linear_code.AbstractLinearCode method), 25
relative_field() (sage.coding.relative_finite_field_extension.RelativeFiniteFieldExtension method), 158
relative_field_basis() (sage.coding.relative_finite_field_extension.RelativeFiniteFieldExtension method), 158
relative_field_degree() (sage.coding.relative_finite_field_extension.RelativeFiniteFieldExtension method), 158
relative_field_representation() (sage.coding.relative_finite_field_extension.RelativeFiniteFieldExtension method), 158
RelativeFiniteFieldExtension (class in sage.coding.relative_finite_field_extension), 155
rootfinding_algorithm() (sage.coding.guruswami_sudan.gs_decoder.GRSGuruswamiSudanDecoder method), 165
roth_ruckenstein_root_finder() (in module sage.coding.guruswami_sudan.gs_decoder), 166

S
sage.coding.bch (module), 104
sage.coding.binary_code (module), 118
sage.coding.bounds_catalog (module), 60
sage.coding.channel_constructions (module), 42
sage.coding.channels_catalog (module), 57
sage.coding.code_bounds (module), 179
sage.coding.code_constructions (module), 108
sage.coding.codecan.autgroup_can_label (module), 144
sage.coding.codecan.codecan (module), 141
sage.coding.codes_catalog (module), 57
sage.coding.cyclic_code (module), 95
sage.coding.databases (module), 60
sage.coding.decoder (module), 49
sage.coding.decoders_catalog (module), 58
sage.coding.encoder (module), 52
sage.coding.encoders_catalog (module), 59
sage.coding.extended_code (module), 136
sage.coding.golay_code (module), 83
sage.coding.grs (module), 67
sage.coding.guava (module), 115
sage.coding.guruswami_sudan.gs_decoder (module), 159
sage.coding.guruswami_sudan.interpolation (module), 166
sage.coding.guruswami_sudan.utils (module), 169
sage.coding.hamming_code (module), 82
sage.coding.information_set_decoder (module), 170
sage.coding.linear_code (module), 1
sage.coding.parity_check_code (module), 85
sage.coding.punctured_code (module), 132
sage.coding.reed_muller_code (module), 88
sage.coding.relative_finite_field_extension (module), 155
sage.coding.self_dual_codes (module), 116
sage.coding.source_coding.huffman (module), 149
sage.coding.subfield_subcode (module), 129
sage.coding.two_weight_db (module), 64
self_dual_binary_codes() (in module sage.coding.self_dual_codes), 117
self_orthogonal_binary_codes() (in module sage.coding.databases), 62
shortened() (sage.coding.linear_code.AbstractLinearCode method), 25
singleton_bound_asympt() (in module sage.coding.code_bounds), 185
singleton_upper_bound() (in module sage.coding.code_bounds), 185
solve_degree2_to_integer_range() (in module sage.coding.guruswami_sudan.utils), 170
spectrum() (sage.coding.linear_code.AbstractLinearCode method), 26
standard_form() (sage.coding.linear_code.AbstractLinearCode method), 27
StaticErrorRateChannel (class in sage.coding.channel_constructions), 47
structured_representation() (sage.coding.punctured_code.PuncturedCode method), 134
SubfieldSubcode (class in sage.coding.subfield_subcode), 129
SubfieldSubcodeOriginalCodeDecoder (class in sage.coding.subfield_subcode), 131
support() (sage.coding.linear_code.AbstractLinearCode method), 27
surrounding_bch_code() (sage.coding.cyclic_code.CyclicCode method), 99
syndrome() (sage.coding.linear_code.AbstractLinearCode method), 28
syndrome_table() (sage.coding.linear_code.LinearCodeSyndromeDecoder method), 38
systematic_generator_matrix() (sage.coding.linear_code.AbstractLinearCode method), 28
systematic_permutation() (sage.coding.linear_code.LinearCodeSystematicEncoder method), 40

Index 205
systematic_positions() (sage.coding.linear_code.LinearCodeSystematicEncoder method), 41

test_expand_to_ortho_basis() (in module sage.coding.binary_code), 126
test_word_perms() (in module sage.coding.binary_code), 126
time_estimate() (sage.coding.information_set_decoder.InformationSetAlgorithm method), 173
ToricCode() (in module sage.coding.code_constructions), 112
transmit() (sage.coding.channel_constructions.Channel method), 43
transmit_unsafe() (sage.coding.channel_constructions.Channel method), 43
testword_permute() (sage.coding.channel_constructions.ErrorErasureChannel method), 45
test_word_unsafe() (sage.coding.channel_constructions.QarySymmetricChannel method), 46
transmit_unsafe() (sage.coding.channel_constructions.StaticErrorRateChannel method), 47
tree() (sage.coding.source_coding.huffman.Huffman method), 152

unencode() (sage.coding.encoder.Encoder method), 54
unencode() (sage.coding.linear_code.AbstractLinearCode method), 29
unencode_nocheck() (sage.coding.cyclic_code.CyclicCodePolynomialEncoder method), 100
unencode_nocheck() (sage.coding.cyclic_code.CyclicCodeVectorEncoder method), 103
unencode_nocheck() (sage.coding.encoder.Encoder method), 55
unencode_nocheck() (sage.coding.grs.GRSEvaluationPolynomialEncoder method), 73
unencode_nocheck() (sage.coding.parity_check_code.ParityCheckCodeStraightforwardEncoder method), 87
unencode_nocheck() (sage.coding.reed_muller_code.ReedMullerPolynomialEncoder method), 93

volume_hamming() (in module sage.coding.code_bounds), 186

walsh_matrix() (in module sage.coding.code_constructions), 115
WalshCode() (in module sage.coding.code_constructions), 113
weight_dist() (in module sage.coding.binary_code), 127
weight_distribution() (sage.coding.golay_code.GolayCode method), 85
weight_distribution() (sage.coding.grs.GeneralizedReedSolomonCode method), 81
weight_distribution() (sage.coding.linear_code.AbstractLinearCode method), 29
weight_enumerator() (sage.coding.linear_code.AbstractLinearCode method), 30

zero() (sage.coding.linear_code.AbstractLinearCode method), 31
zeta_function() (sage.coding.linear_code.AbstractLinearCode method), 31
zeta_polynomial() (sage.coding.linear_code.AbstractLinearCode method), 31