Sage Reference Manual: Coding Theory

Release 6.6

The Sage Development Team

April 18, 2015

1 Index of Codes **1**

2 Linear Codes **3**

3 Linear code constructions **33**
3.1 Functions 34

4 Guava error-correcting code constructions. **49**
4.1 Functions 49

5 Binary self-dual codes **51**

6 Bounds for Parameters of Codes **55**

7 Canonical forms and automorphisms for linear codes over finite fields. **63**

8 Delsarte, a.k.a. Linear Programming (LP), upper bounds. **67**

9 Huffman Encoding **71**
9.1 Classes and functions 71

10 Indices and Tables **77**

Bibliography **79**

CHAPTER

ONE

INDEX OF CODES

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

- codes.BCHCode
- codes.BinaryGolayCode
- codes.BinaryReedMullerCode
- codes.CyclicCode
- codes.CyclicCodeFromGeneratingPolynomial
- codes.CyclicCodeFromCheckPolynomial
- codes.DuadicCodeEvenPair
- codes.DuadicCodeOddPair
- codes.ExtendedBinaryGolayCode
- codes.ExtendedQuadraticResidueCode
- codes.ExtendedTernaryGolayCode
- codes.HammingCode
- codes.LinearCodeFromCheckMatrix
- codes.QuadraticResidueCode
- codes.QuadraticResidueCodeEvenPair
- codes.QuadraticResidueCodeOddPair
- codes.QuasiQuadraticResidueCode
- codes.RandomLinearCode
- codes.RandomLinearCodeGuava
- codes.ReedSolomonCode
- codes.TernaryGolayCode
- codes.ToricCode
- codes.TrivialCode
- codes.WalshCode

CHAPTER

TWO

LINEAR CODES

VERSION: 1.2

Let F be a finite field. Here, we will denote the finite field with q elements by $\mathbf{F}_q$. A subspace of F^n (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, with rows the basis vectors. It is called the generator matrix of C. The rows of the parity check matrix of C are a basis for the code,

$$C^* = \{v \in GF(q)^n \mid v \cdot c = 0,\ for\ all\ c \in C\},$$

called the dual space of C.

If $F = \mathbf{F}_2$ then C is called a binary code. If $F = \mathbf{F}_q$ then C is called a q-ary code. The elements of a code C are called codewords.

Let C, D be linear codes of length n and dimension k. There are several notions of equivalence for linear codes:

C and D are

- 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 ϕ of units of length n such that $(c_{\pi(0)}\phi_0^{-1}, \ldots, c_{\pi(n-1)}\phi_{n-1}^{-1}) \in D$ for all $c \in C$.
- semilinear equivalent, if there is some permutation $\pi \in S_n$, a vector ϕ of units of length n and a field automorphism α such that $(\alpha(c_{\pi(0)})\phi_0^{-1}, \ldots, \alpha(c_{\pi(n-1)})\phi_{n-1}^{-1}) \in D$ for all $c \in C$.

These are group actions. If one of these group elements sends the linear code C to itself, then we will call it an automorphism. Depending on the group action we will call those groups:

- permuation automorphism group
- monomial automorphism group (every linear Hamming isometry is a monomial transformation of the ambient space, for $n \geq 3$)
- automorphism group (every semilinear Hamming isometry is a semimonomial transformation of the ambient space, for $n \geq 3$)

This file contains

1. LinearCode class definition; LinearCodeFromVectorspace conversion function,
2. The spectrum (weight distribution), covering_radius, minimum distance programs (calling Steve Linton's or CJ Tjhal's C programs), characteristic_function, and several implementations of the Duursma zeta function (sd_zeta_polynomial, zeta_polynomial, zeta_function, chinen_polynomial, for example),
3. interface with best_known_linear_code_www (interface with codetables.de since A. Brouwer's online tables have been disabled), bounds_minimum_distance which call tables in GUAVA (updated May 2006) created by Cen Tjhai instead of the online internet tables,

4. generator_matrix, generator_matrix_systematic, information_set, list, parity_check_matrix, decode, dual_code, extended_code, shortened, punctured, genus, binomial_moment, and divisor methods for LinearCode,
5. Boolean-valued functions such as "==", is_self_dual, is_self_orthogonal, is_subcode, is_permutation_automorphism, is_permutation_equivalent (which interfaces with Robert Miller's partition refinement code),
6. permutation methods: is_permutation_automorphism, permutation_automorphism_group, permuted_code, standard_form, module_composition_factors,
7. design-theoretic methods: assmus_mattson_designs (implementing Assmus-Mattson Theorem),
8. code constructions, such as HammingCode and ToricCode, are in a separate `code_constructions.py` module; in the separate `guava.py` module, you will find constructions, such as RandomLinearCodeGuava and BinaryReedMullerCode, wrapped from the corresponding GUAVA codes.

EXAMPLES:

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

To be added:

1. More wrappers
2. GRS codes and special decoders.
3. P^1 Goppa codes and group actions on P^1 RR space codes.

REFERENCES:

- [HP] W. C. Huffman and V. Pless, Fundamentals of error-correcting codes, Cambridge Univ. Press, 2003.
- [Gu] GUAVA manual, http://www.gap-system.org/Packages/guava.html

AUTHORS:

- David Joyner (2005-11-22, 2006-12-03): initial version
- William Stein (2006-01-23): Inclusion in Sage
- David Joyner (2006-01-30, 2006-04): small fixes
- David Joyner (2006-07): added documentation, group-theoretical methods, ToricCode
- David Joyner (2006-08): hopeful latex fixes to documentation, added list and __iter__ methods to LinearCode and examples, added hamming_weight function, fixed random method to return a vector, TrivialCode, fixed subtle bug in dual_code, added galois_closure method, fixed mysterious bug in permutation_automorphism_group (GAP was over-using "G" somehow?)

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

TESTS:

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

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

Bases: sage.modules.module.Module

Linear codes over a finite field or finite ring.

A *linear code* is a subspace of a vector space over a finite field. It can be defined by one of its basis or equivalently a generator matrix (a $k \times n$ matrix of full rank k).

See Wikipedia article Linear_code for more information.

INPUT:

- `generator_matrix` – 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`)
- `d` – (optional, default: `None`) the minimum distance of the code

---

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

EXAMPLES:

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

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

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

Another example.:

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

AUTHORS:

- David Joyner (11-2005)

ambient_space()

Returns the ambient vector space of $self$.

EXAMPLES:

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

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

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

dual $[n, n-k, d^*]$ code C^*. Fix a t, $0 < t < d$, and let

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

Assume $s \leq d - t$.

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

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

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

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

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

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

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

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

EXAMPLES:

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

REFERENCE:

•[HP] W. C. Huffman and V. Pless, Fundamentals of ECC, Cambridge Univ. Press, 2003.

automorphism_group_gens(*equivalence='semilinear'*)

Return generators of the automorphism group of `self`.

INPUT:

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

OUTPUT:

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

EXAMPLES:

```
sage: C = codes.HammingCode(3,GF(4,"z"));
sage: C.automorphism_group_gens()
([((1, 1, 1, z, z + 1, z + 1, z + 1, z, z, 1, 1, 1, z, z, z + 1, z, z, z + 1, z + 1, z + 1,
      Defn: z |--> z + 1), ((1, 1, 1, z, z + 1, 1, 1, z, z, z + 1, z, z, z + 1, z + 1, z + 1
      Defn: z |--> z), ((z, z, z, z, z, z, z, z, z, z, z, z, z, z, z, z, z, z, z, z); (),
      Defn: z |--> z)], 362880)
sage: C.automorphism_group_gens(equivalence="linear")
([((z, z, 1, 1, z + 1, z, z + 1, z, z, z + 1, 1, 1, 1, z + 1, z, z, z + 1, z + 1, 1, 1, z);
      Defn: z |--> z), ((z + 1, 1, z, 1, 1, z + 1, z + 1, z, 1, z, z + 1, z, z + 1, z + 1, z
      Defn: z |--> z), ((z + 1, z + 1, z + 1, z + 1, z + 1, z + 1, z + 1, z + 1, z + 1, z +
      Defn: z |--> z)], 181440)
sage: C.automorphism_group_gens(equivalence="permutational")
([((1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1); (1,11)(3,10)(4,9)(5,7)(1
      Defn: z |--> z), ((1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1); (2,
      Defn: z |--> z), ((1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1); (1,
      Defn: z |--> z), ((1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1); (2,
      Defn: z |--> z)], 64)
```

basis()

Returns a basis of $self$.

EXAMPLES:

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

binomial_moment(i)

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

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

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

EXAMPLES:

```
sage: C = codes.HammingCode(3,GF(2))
sage: C.binomial_moment(2)
0
```

```
sage: C.binomial_moment(4)     # long time
35
```

Warning: This is slow.

REFERENCE:

- I. Duursma, "Combinatorics of the two-variable zeta function", Finite fields and applications, 109-136, Lecture Notes in Comput. Sci., 2948, Springer, Berlin, 2004.

canonical_representative(*equivalence='semilinear'*)

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

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

INPUT:

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

OUTPUT:

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

EXAMPLES:

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

Check that the transporter element is correct:

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

Check if an equivalent code has the same canonical representative:

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

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

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

cardinality()

Return the size of this code.

EXAMPLES:

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

characteristic()

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

EXAMPLES:

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

characteristic_polynomial()

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

EXAMPLES:

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

REFERENCES:

- van Lint, Introduction to coding theory, 3rd ed., Springer-Verlag GTM, 86, 1999.

check_mat(**args*, ***kwds*)

Deprecated: Use `parity_check_matrix()` instead. See trac ticket #17973 for details.

chinen_polynomial()

Returns the Chinen zeta polynomial of the code.

EXAMPLES:

```
sage: C = codes.HammingCode(3,GF(2))
sage: C.chinen_polynomial()       # long time
1/5*(2*sqrt(2)*t^3 + 2*sqrt(2)*t^2 + 2*t^2 + sqrt(2)*t + 2*t + 1)/(sqrt(2) + 1)
sage: C = codes.TernaryGolayCode()
sage: C.chinen_polynomial()       # long time
1/7*(3*sqrt(3)*t^3 + 3*sqrt(3)*t^2 + 3*t^2 + sqrt(3)*t + 3*t + 1)/(sqrt(3) + 1)
```

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

REFERENCES:

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

covering_radius()

Wraps Guava's `CoveringRadius` command.

The covering radius of a linear code C is the smallest number r with the property that each element v of the ambient vector space of C has at most a distance r to the code C. So for each vector v there must be an element c of C with $d(v, c) \leq r$. A binary linear code with reasonable small covering radius is often referred to as a covering code.

For example, if C is a perfect code, the covering radius is equal to t, the number of errors the code can correct, where $d = 2t + 1$, with d the minimum distance of C.

EXAMPLES:

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

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

Decodes the received vector `right` to an element c in this code.

Optional algorithms are "guava", "nearest neighbor" or "syndrome". The `algorithm="guava"` wraps GUAVA's `Decodeword`. Hamming codes have a special decoding algorithm; otherwise, `"syndrome"` decoding is used.

INPUT:

- `right` - Vector of length the length of this code
- `algorithm` - Algorithm to use, one of `"syndrome"`, `"nearest neighbor"`, and `"guava"` (default: `"syndrome"`)

OUTPUT:

- The codeword in this code closest to `right`.

EXAMPLES:

```
sage: C = codes.HammingCode(3,GF(2))
sage: MS = MatrixSpace(GF(2),1,7)
sage: F = GF(2); a = F.gen()
sage: v1 = [a,a,F(0),a,a,F(0),a]
sage: C.decode(v1)
(1, 1, 0, 1, 0, 0, 1)
sage: C.decode(v1,algorithm="nearest neighbor")
(1, 1, 0, 1, 0, 0, 1)
sage: C.decode(v1,algorithm="guava")  # optional - gap_packages (Guava package)
(1, 1, 0, 1, 0, 0, 1)
sage: v2 = matrix([[a,a,F(0),a,a,F(0),a]])
sage: C.decode(v2)
(1, 1, 0, 1, 0, 0, 1)
sage: v3 = vector([a,a,F(0),a,a,F(0),a])
sage: c = C.decode(v3); c
(1, 1, 0, 1, 0, 0, 1)
sage: c in C
True
sage: C = codes.HammingCode(2,GF(5))
sage: v = vector(GF(5),[1,0,0,2,1,0])
sage: C.decode(v)
(1, 0, 0, 2, 2, 0)
sage: F.<a> = GF(4)
sage: C = codes.HammingCode(2,F)
sage: v = vector(F, [1,0,0,a,1])
sage: C.decode(v)
(a + 1, 0, 0, a, 1)
sage: C.decode(v, algorithm="nearest neighbor")
(a + 1, 0, 0, a, 1)
sage: C.decode(v, algorithm="guava")  # optional - gap_packages (Guava package)
(a + 1, 0, 0, a, 1)
```

Does not work for very long codes since the syndrome table grows too large.

TESTS:

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

```
sage: (C.decode(v)).is_immutable()
True
```

dimension()

Returns the dimension of this code.

EXAMPLES:

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

direct_sum(*other*)

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

EXAMPLES:

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

divisor()

Returns the divisor of a code, which is the smallest integer $d_0 > 0$ such that each $A_i > 0$ iff i is divisible by d_0.

EXAMPLES:

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

dual_code()

This computes the dual code Cd of the code C,

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

Does not call GAP.

EXAMPLES:

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

extended_code()

If `self` is a linear code of length n defined over F then this returns the code of length $n + 1$ where the last digit c_n satisfies the check condition $c_0 + ... + c_n = 0$. If `self` is an $[n, k, d]$ binary code then the extended code $C^\vee$ is an $[n + 1, k, d^\vee]$ code, where $d^= d$ (if d is even) and $d^\vee = d + 1$ (if d is odd).

EXAMPLES:

```
sage: C = codes.HammingCode(3,GF(4,'a'))
sage: C
Linear code of length 21, dimension 18 over Finite Field in a of size 2^2
sage: Cx = C.extended_code()
sage: Cx
Linear code of length 22, dimension 18 over Finite Field in a of size 2^2
```

galois_closure(*F0*)

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

EXAMPLES:

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

gen_mat(**args*, ***kwds*)

Deprecated: Use `generator_matrix()` instead. See trac ticket #17973 for details.

gen_mat_systematic(**args*, ***kwds*)

Deprecated: Use `generator_matrix_systematic()` instead. See trac ticket #17973 for details.

generator_matrix()

Return a generator matrix of this code.

EXAMPLES:

```
sage: C1 = codes.HammingCode(3,GF(2))
sage: C1.generator_matrix()
[1 0 0 0 0 1 1]
[0 1 0 0 1 0 1]
[0 0 1 0 1 1 0]
[0 0 0 1 1 1 1]
sage: C2 = codes.HammingCode(2,GF(4,"a"))
sage: C2.generator_matrix()
[    1     0     0 a + 1     a]
[    0     1     0     1     1]
[    0     0     1     a a + 1]
```

generator_matrix_systematic()

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.

EXAMPLES:

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

```
sage: code.generator_matrix_systematic()
[1 2 0]
[0 0 1]
```

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

EXAMPLES:

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

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

EXAMPLES:

```
sage: C1 = codes.HammingCode(3,GF(2)); C1
Linear code of length 7, dimension 4 over Finite Field of size 2
sage: C1.genus()
1
sage: C2 = codes.HammingCode(2,GF(4,"a")); C2
Linear code of length 5, dimension 3 over Finite Field in a of size 2^2
sage: C2.genus()
0
```

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

information_set()
Return an information set of the code.

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

OUTPUT:

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

EXAMPLES:

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

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

EXAMPLES:

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

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

EXAMPLES:

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

is_permutation_equivalent(*other*, *algorithm=None*)

Returns `True` if `self` and `other` are permutation equivalent codes and `False` otherwise.

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

Uses Robert Miller's double coset partition refinement work.

EXAMPLES:

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

is_self_dual()

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

EXAMPLES:

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

is_self_orthogonal()

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

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

EXAMPLES:

```
sage: C = codes.ExtendedBinaryGolayCode()
sage: C.is_self_orthogonal()
True
```

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

is_subcode(*other*)

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

EXAMPLES:

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

length()

Returns the length of this code.

EXAMPLES:

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

list()

Return a list of all elements of this linear code.

EXAMPLES:

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

minimum_distance(*algorithm=None*)

Returns the minimum distance of this linear code.

By default, this uses a GAP kernel function (in C and not part of Guava) written by Steve Linton. If

algorithm="guava" is set and q is 2 or 3 then this uses a very fast program written in C written by CJ Tjhal. (This is much faster, except in some small examples.)

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

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

INPUT:

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

OUTPUT:

- Integer, minimum distance of this code

EXAMPLES:

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

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

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

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

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

Another example.:

```
sage: C = codes.HammingCode(2,GF(4,"a")); C
Linear code of length 5, dimension 3 over Finite Field in a of size 2^2
sage: C.minimum_distance()
3
```

TESTS:

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

module_composition_factors(*gp*)

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

EXAMPLES:

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

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

parity_check_matrix()

Returns the parity check matrix of `self`.

EXAMPLES:

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

permutation_automorphism_group(*algorithm='partition'*)

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

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

INPUT:

- `algorithm` - If `"gap"` then GAP's MatrixAutomorphism function (written by Thomas Breuer) is used. The implementation combines an idea of mine with an improvement suggested by Cary Huffman. If `"gap+verbose"` then code-theoretic data is printed out at several stages of the computation. If `"partition"` then the (default) partition refinement algorithm of Robert Miller is used. Finally, if `"codecan"` then the partition refinement algorithm of Thomas Feulner is used, which also computes a canonical representative of `self` (call `canonical_representative()` to access it).

OUTPUT:

- Permutation automorphism group

EXAMPLES:

```
sage: MS = MatrixSpace(GF(2),4,8)
sage: G  = MS([[1,0,0,0,1,1,1,0],[0,1,1,1,0,0,0,0],[0,0,0,0,0,0,0,1],[0,0,0,0,0,1,0,0]])
sage: C  = LinearCode(G)
sage: C
Linear code of length 8, dimension 4 over Finite Field of size 2
```

```
sage: G = C.permutation_automorphism_group()
sage: G.order()
144
sage: GG = C.permutation_automorphism_group("codecan")
sage: GG == G
True
```

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

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

Other examples:

```
sage: C = codes.ExtendedBinaryGolayCode()
sage: G = C.permutation_automorphism_group()
sage: G.order()
244823040
sage: C = codes.HammingCode(5, GF(2))
sage: G = C.permutation_automorphism_group()
sage: G.order()
9999360
sage: C = codes.HammingCode(2,GF(3)); C
Linear code of length 4, dimension 2 over Finite Field of size 3
sage: C.permutation_automorphism_group(algorithm="partition")
Permutation Group with generators [(1,3,4)]
sage: C = codes.HammingCode(2,GF(4,"z")); C
Linear code of length 5, dimension 3 over Finite Field in z of size 2^2
sage: G = C.permutation_automorphism_group(algorithm="partition"); G
Permutation Group with generators [(1,3)(4,5), (1,4)(3,5)]
sage: GG = C.permutation_automorphism_group(algorithm="codecan") # long time
sage: GG == G # long time
True
sage: C.permutation_automorphism_group(algorithm="gap")  # optional - gap_packages (Guava pa
sage: C = codes.TernaryGolayCode()
sage: C.permutation_automorphism_group(algorithm="gap")  # optional - gap_packages (Guava pa
Permutation Group with generators [(3,4)(5,7)(6,9)(8,11), (3,5,8)(4,11,7)(6,9,10), (2,3)(4,6
```

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

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

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

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

permuted_code(*p*)

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

EXAMPLES:

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

punctured(*L*)

Returns the code punctured at the positions L, $L \subset \{1, 2, ..., n\}$. If this code C is of length n in GF(q) then the code C^L obtained from C by puncturing at the positions in L is the code of length $n - L$ consisting of codewords of C which have their $i - th$ coordinate deleted if $i \in L$ and left alone if $i \notin L$:

$$C^L = \{(c_{i_1}, ..., c_{i_N}) \mid (c_1, ..., c_n) \in C\},$$

where $\{1, 2, ..., n\} - T = \{i_1, ..., i_N\}$. In particular, if $L = \{j\}$ then C^L is simply the code obtainen from C by deleting the $j - th$ coordinate of each codeword. The code C^L is called the punctured code at L. The dimension of C^L can decrease if $|L| > d - 1$.

INPUT:

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

OUTPUT:

- Linear code, the punctured code described above

EXAMPLES:

```
sage: C = codes.HammingCode(3,GF(2))
sage: C.punctured([1,2])
Linear code of length 5, dimension 4 over Finite Field of size 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(3,GF(4,'a'))
sage: C.random_element() # random test
(1, 0, 0, a + 1, 1, a, a, a + 1, a + 1, 1, 1, 0, a + 1, a, 0, a, a, 0, a, a, 1)
```

Passes extra positional or keyword arguments through:

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

TESTS:

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

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

redundancy_matrix(*C*)

If C is a linear [n,k,d] code then this function returns a $kimes(n-k)$ matrix A such that G = (I,A) generates a code (in standard form) equivalent to C. If C is already in standard form and G = (I,A) is its generator matrix then this function simply returns that A.

OUTPUT:

- Matrix, the redundancy matrix

EXAMPLES:

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

sd_duursma_data(*C*, *i*)

Returns the Duursma data v and m of this formally s.d. code C and the type number i in (1,2,3,4). Does *not* check if this code is actually sd.

INPUT:

- `i` - Type number

OUTPUT:

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

REFERENCES:

EXAMPLES:

```
sage: MS = MatrixSpace(GF(2),2,4)
sage: G = MS([1,1,0,0,0,0,1,1])
sage: C = LinearCode(G)
sage: C == C.dual_code()  # checks that C is self dual
True
sage: for i in [1,2,3,4]: print C.sd_duursma_data(i)
[2, -1]
[2, -3]
[2, -2]
[2, -1]
```

sd_duursma_q(*C*, *i*, *d0*)

INPUT:

- `C` - sd code; does *not* check if C is actually an sd code
- `i` - Type number, one of 1,2,3,4
- `d0` - Divisor, the smallest integer such that each $A_i > 0$ iff i is divisible by $d0$

OUTPUT:

- Coefficients $q_0, q_1, \ldots$ of $q(T)$ as in Duursma [D]

REFERENCES:

- [D] - I. Duursma, "Extremal weight enumerators and ultraspherical polynomials"

EXAMPLES:

```
sage: C1 = codes.HammingCode(3,GF(2))
sage: C2 = C1.extended_code(); C2
Linear code of length 8, dimension 4 over Finite Field of size 2
sage: C2.is_self_dual()
True
sage: C2.sd_duursma_q(1,1)
2/5*T^2 + 2/5*T + 1/5
sage: C2.sd_duursma_q(3,1)
3/5*T^4 + 1/5*T^3 + 1/15*T^2 + 1/15*T + 1/15
```

sd_zeta_polynomial(*C*, *typ=1*)

Returns the Duursma zeta function of a self-dual code using the construction in [D].

INPUT:

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

OUTPUT:

- Polynomial

EXAMPLES:

```
sage: C1 = codes.HammingCode(3,GF(2))
sage: C2 = C1.extended_code(); C2
Linear code of length 8, dimension 4 over Finite Field of size 2
sage: C2.is_self_dual()
True
sage: C2.sd_zeta_polynomial()
2/5*T^2 + 2/5*T + 1/5
sage: C2.zeta_polynomial()
2/5*T^2 + 2/5*T + 1/5
sage: P = C2.sd_zeta_polynomial(); P(1)
1
sage: F.<z> = GF(4,"z")
sage: MS = MatrixSpace(F, 3, 6)
sage: G = MS([[1,0,0,1,z,z],[0,1,0,z,1,z],[0,0,1,z,z,1]])
sage: C = LinearCode(G)  # the "hexacode"
sage: C.sd_zeta_polynomial(4)
1
```

It is a general fact about Duursma zeta polynomials that $P(1) = 1$.

REFERENCES:

- [D] I. Duursma, "Extremal weight enumerators and ultraspherical polynomials"

shortened(*L*)

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

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

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

INPUT:

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

OUTPUT:

- Linear code, the shortened code described above

EXAMPLES:

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

spectrum(*algorithm=None*)

Returns the spectrum of `self` as a list.

The default algorithm uses a GAP kernel function (in C) written by Steve Linton.

INPUT:

- `algorithm` - None, `"gap"`, `"leon"`, or `"binary"`; defaults to `"gap"` except in the binary case. If `"gap"` then uses the GAP function, if `"leon"` then uses Jeffrey Leon's software via Guava, and if `"binary"` then uses Sage native Cython code
- List, the spectrum

The optional algorithm (`"leon"`) may create a stack smashing error and a traceback but should return the correct answer. It appears to run much faster than the GAP algorithm in some small examples and much slower than the GAP algorithm in other larger examples.

EXAMPLES:

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

```
sage: C = codes.HammingCode(2,GF(5)); C
Linear code of length 6, dimension 4 over Finite Field of size 5
sage: C.spectrum() == C.spectrum(algorithm="leon")   # optional - gap_packages (Guava packag
True
sage: C = codes.HammingCode(2,GF(7)); C
Linear code of length 8, dimension 6 over Finite Field of size 7
sage: C.spectrum() == C.spectrum(algorithm="leon")   # optional - gap_packages (Guava packag
True
```

standard_form()

Returns the standard form of this linear code.

An $[n, k]$ linear code with generator matrix G in standard form is the row-reduced echelon form of G is (I, A), where I denotes the $k \times k$ identity matrix and A is a $k \times (n - k)$ block. This method returns a pair (C, p) where C is a code permutation equivalent to `self` and p in S_n, with n the length of C, is the permutation sending `self` to C. This does not call GAP.

Thanks to Frank Luebeck for (the GAP version of) this code.

EXAMPLES:

```
sage: C = codes.HammingCode(3,GF(2))
sage: C.generator_matrix()
[1 0 0 0 0 1 1]
[0 1 0 0 1 0 1]
[0 0 1 0 1 1 0]
[0 0 0 1 1 1 1]
sage: Cs,p = C.standard_form()
sage: p
()
sage: MS = MatrixSpace(GF(3),3,7)
sage: G = MS([[1,0,0,0,1,1,0],[0,1,0,1,0,1,0],[0,0,0,0,0,0,1]])
sage: C = LinearCode(G)
sage: Cs, p = C.standard_form()
sage: p
(3,7)
sage: Cs.generator_matrix()
 [1 0 0 0 1 1 0]
 [0 1 0 1 0 1 0]
 [0 0 1 0 0 0 0]
```

support()

Returns the set of indices j where A_j is nonzero, where spectrum(self) = $[A_0, A_1, ..., A_n]$.

OUTPUT:

- List of integers

EXAMPLES:

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

weight_distribution(*algorithm=None*)

Returns the spectrum of `self` as a list.

The default algorithm uses a GAP kernel function (in C) written by Steve Linton.

INPUT:

- `algorithm` - None, `"gap"`, `"leon"`, or `"binary"`; defaults to `"gap"` except in the binary case. If `"gap"` then uses the GAP function, if `"leon"` then uses Jeffrey Leon's software via Guava, and if `"binary"` then uses Sage native Cython code
- List, the spectrum

The optional algorithm (`"leon"`) may create a stack smashing error and a traceback but should return the correct answer. It appears to run much faster than the GAP algorithm in some small examples and much slower than the GAP algorithm in other larger examples.

EXAMPLES:

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

weight_enumerator(*names='xy'*, *name2=None*)

Returns the weight enumerator of the code.

INPUT:

- `names` - String of length 2, containing two variable names (default: `"xy"`). Alternatively, it can be a variable name or a string, or a tuple of variable names or strings.
- `name2` - string or symbolic variable (default: `None`). If `name2` is provided then it is assumed that `names` contains only one variable.

OUTPUT:

- Polynomial over $\mathbf{Q}$

EXAMPLES:

```
sage: C = codes.HammingCode(3,GF(2))
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: (var1, var2) = var('var1, var2')
sage: C.weight_enumerator((var1, var2))
var1^7 + 7*var1^4*var2^3 + 7*var1^3*var2^4 + var2^7
sage: C.weight_enumerator(var1, var2)
var1^7 + 7*var1^4*var2^3 + 7*var1^3*var2^4 + var2^7
```

zero()

Return the zero vector.

EXAMPLES:

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

zeta_function(*name='T'*)

Returns the Duursma zeta function of the code.

INPUT:

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

OUTPUT:

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

EXAMPLES:

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

zeta_polynomial(*name='T'*)

Returns the Duursma zeta polynomial of this code.

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

INPUT:

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

OUTPUT:

- Polynomial over $\mathbf{Q}$

EXAMPLES:

```
sage: C = codes.HammingCode(3,GF(2))
sage: C.zeta_polynomial()
2/5*T^2 + 2/5*T + 1/5
sage: C = best_known_linear_code(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(4,GF(2))
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:

- I. Duursma, "From weight enumerators to zeta functions", in Discrete Applied Mathematics, vol. 111, no. 1-2, pp. 55-73, 2001.

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

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

INPUT:

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

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

EXAMPLES:

```
sage: V = VectorSpace(GF(2), 8)
sage: L = V.subspace([[1,1,1,1,0,0,0,0],[0,0,0,0,1,1,1,1]])
sage: C = LinearCodeFromVectorSpace(L)
sage: C.generator_matrix()
[1 1 1 1 0 0 0 0]
[0 0 0 0 1 1 1 1]
sage: C.minimum_distance()
4
```

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

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

sage.coding.linear_code.**best_known_linear_code**(*n*, *k*, *F*)

Returns the best known (as of 11 May 2006) linear code of length `n`, dimension `k` over field `F`. The function uses the tables described in `bounds_minimum_distance` to construct this code.

This does not require an internet connection.

EXAMPLES:

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

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

sage.coding.linear_code.**best_known_linear_code_www**(*n*, *k*, *F*, *verbose=False*)

Explains the construction of the best known linear code over GF(q) with length n and dimension k, courtesy of the www page 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:

- Text about why the bounds are as given

EXAMPLES:

```
sage: L = best_known_linear_code_www(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.linear_code.**bounds_minimum_distance**(*n*, *k*, *F*)

Calculates a lower and upper bound for the minimum distance of an optimal linear code with word length `n` and dimension `k` over the field `F`.

The function returns a 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.win.tue.nl/ aeb/voorlincod.html or use the Sage function `minimum_distance_why` for the most recent data.) These tables contain lower and upper bounds for $q = 2$ (when `n <= 257`), $q = 3$ (when `n <= 243`), $q = 4$ (`n <= 256`). (Current as of 11 May 2006.) For codes over other fields and for larger word lengths, trivial bounds are used.

This does not require an internet connection. The format of the output is a little non-intuitive. Try `bounds_minimum_distance(10,5,GF(2))` for an example.

This function requires optional GAP package (Guava).

EXAMPLES:

```
sage: print bounds_minimum_distance(10,5,GF(2)) # optional - gap_packages (Guava package)
rec(
  construction :=
   [ <Operation "ShortenedCode">,
```

```
        [
            [ <Operation "UUVCode">,
                [
                    [ <Operation "DualCode">,
                        [ [ <Operation "RepetitionCode">, [ 8, 2 ] ] ] ],
                    [ <Operation "UUVCode">,
                        [
                            [ <Operation "DualCode">,
                                [ [ <Operation "RepetitionCode">, [ 4, 2 ] ] ] ]
                              , [ <Operation "RepetitionCode">, [ 4, 2 ] ] ]
                    ] ], [ 1, 2, 3, 4, 5, 6 ] ] ],
    k := 5,
    lowerBound := 4,
    lowerBoundExplanation := ...
    n := 10,
    q := 2,
    references := rec(
         ),
    upperBound := 4,
    upperBoundExplanation := ... )
```

sage.coding.linear_code.**code2leon**(*C*)

Writes a file in Sage's temp directory representing the code C, returning the absolute path to the file. This is the Sage translation of the GuavaToLeon command in Guava's codefun.gi file.

INPUT:

- C - a linear code (over GF(p), p < 11)

OUTPUT:

- Absolute path to the file written

EXAMPLES:

```
sage: C = codes.HammingCode(3,GF(2)); C
Linear code of length 7, dimension 4 over Finite Field of size 2
sage: file_loc = sage.coding.linear_code.code2leon(C)
sage: f = open(file_loc); print f.read()
LIBRARY code;
code=seq(2,4,7,seq(
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
));
FINISH;
sage: f.close()
```

sage.coding.linear_code.**min_wt_vec_gap**(*Gmat*, *n*, *k*, *F*, *algorithm=None*)

Returns a minimum weight vector of the code generated by Gmat.

Uses C programs written by Steve Linton in the kernel of GAP, so is fairly fast. The option algorithm="guava" requires Guava. The default algorithm requires GAP but not Guava.

INPUT:

- Gmat - String representing a GAP generator matrix G of a linear code
- n - Length of the code generated by G

- k - Dimension of the code generated by G
- F - Base field

OUTPUT:

- Minimum weight vector of the code generated by `Gmat`

REMARKS:

- The code in the default case allows one (for free) to also compute the message vector m such that $mG = v$, and the (minimum) distance, as a triple. however, this output is not implemented.
- The binary case can presumably be done much faster using Robert Miller's code (see the docstring for the spectrum method). This is also not (yet) implemented.

EXAMPLES:

```
sage: Gstr = "Z(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: sage.coding.linear_code.min_wt_vec_gap(Gstr,7,4,GF(2))
(0, 1, 0, 1, 0, 1, 0)
```

This output is different but still a minimum weight vector:

```
sage: sage.coding.linear_code.min_wt_vec_gap(Gstr,7,4,GF(2),algorithm="guava")     # optional - g
(0, 0, 1, 0, 1, 1, 0)
```

Here `Gstr` is a generator matrix of the Hamming [7,4,3] binary code.

AUTHORS:

- David Joyner (11-2005)

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

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

INPUT:

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

EXAMPLES:

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

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

```
Linear code of length 6, dimension 2 over Finite Field of size 2
Linear code of length 6, dimension 2 over Finite Field of size 2
Linear code of length 7, dimension 3 over Finite Field of size 2
Linear code of length 6, dimension 1 over Finite Field of size 2
```

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

```
sage: for B in self_orthogonal_binary_codes(7,3,4):
...    print B; print B.generator_matrix()
...
Linear code of length 4, dimension 1 over Finite Field of size 2
[1 1 1 1]
Linear code of length 6, dimension 2 over Finite Field of size 2
[1 1 1 1 0 0]
[0 1 0 1 1 1]
Linear code of length 7, dimension 3 over Finite Field of size 2
[1 0 1 1 0 1 0]
[0 1 0 1 1 1 0]
[0 0 1 0 1 1 1]
```

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

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

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

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

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

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

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

INPUT:

- `Gmat` - String representing a GAP generator matrix G of a linear code
- `n` - Integer greater than 1, representing the number of columns of G (i.e., the length of the linear code)
- `F` - Finite field (in Sage), base field the code

OUTPUT:

- Spectrum of the associated code

EXAMPLES:

```
sage: Gstr = 'Z(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: F = GF(2)
sage: sage.coding.linear_code.wtdist_gap(Gstr, 7, F)
[1, 0, 0, 7, 7, 0, 0, 1]
```

Here `Gstr` is a generator matrix of the Hamming [7,4,3] binary code.

ALGORITHM:

Uses C programs written by Steve Linton in the kernel of GAP, so is fairly fast.

AUTHORS:

- David Joyner (2005-11)

CHAPTER

THREE

LINEAR CODE CONSTRUCTIONS

This file contains constructions of error-correcting codes which are pure Python/Sage and not obtained from wrapping GUAVA functions. The GUAVA wrappers are in guava.py.

All codes available here can be accessed through the `codes` object:

```
sage: codes.HammingCode(3,GF(2))
Linear code of length 7, dimension 4 over Finite Field of size 2
```

Let F be a finite field with q elements. Here's a constructive definition of a cyclic code of length n.

1. Pick a monic polynomial $g(x) \in F[x]$ dividing $x^n - 1$. This is called the generating polynomial of the code.
2. For each polynomial $p(x) \in F[x]$, compute $p(x)g(x) \pmod{x^n - 1}$. Denote the answer by $c_0 + c_1 x + ... + c_{n-1}x^{n-1}$.
3. $\mathbf{c} = (c_0, c_1, ..., c_{n-1})$ is a codeword in C. Every codeword in C arises in this way (from some $p(x)$).

The polynomial notation for the code is to call $c_0 + c_1 x + ... + c_{n-1}x^{n-1}$ the codeword (instead of $(c_0, c_1, ..., c_{n-1})$). The polynomial $h(x) = (x^n - 1)/g(x)$ is called the check polynomial of C.

Let n be a positive integer relatively prime to q and let α be a primitive n-th root of unity. Each generator polynomial g of a cyclic code C of length n has a factorization of the form

$$g(x) = (x - \alpha^{k_1})...(x - \alpha^{k_r}),$$

where $\{k_1, ..., k_r\} \subset \{0, ..., n-1\}$. The numbers α^{k_i}, $1 \leq i \leq r$, are called the zeros of the code C. Many families of cyclic codes (such as BCH codes and the quadratic residue codes) are defined using properties of the zeros of C.

- BCHCode - A 'Bose-Chaudhuri-Hockenghem code' (or BCH code for short) is the largest possible cyclic code of length n over field F=GF(q), whose generator polynomial has zeros (which contain the set) $Z = \{a^i \mid i \in C_b \cup ...C_{b+delta-2}\}$, where a is a primitive n^{th} root of unity in the splitting field $GF(q^m)$, b is an integer $0 \leq b \leq n - delta + 1$ and m is the multiplicative order of q modulo n. The default here is $b = 0$ (unlike Guava, which has default $b = 1$). Here C_k are the cyclotomic codes.
- BinaryGolayCode, ExtendedBinaryGolayCode, TernaryGolayCode, ExtendedTernaryGolayCode the well-known"extremal" Golay codes, http://en.wikipedia.org/wiki/Golay_code
- cyclic codes - CyclicCodeFromGeneratingPolynomial (= CyclicCode), CyclicCodeFromCheckPolynomial, http://en.wikipedia.org/wiki/Cyclic_code
- DuadicCodeEvenPair, DuadicCodeOddPair: Constructs the "even (resp. odd) pair" of duadic codes associated to the "splitting" S1, S2 of n. This is a special type of cyclic code whose generator is determined by S1, S2. See chapter 6 in [HP].
- HammingCode - the well-known Hamming code, http://en.wikipedia.org/wiki/Hamming_code
- LinearCodeFromCheckMatrix - for specifying the code using the check matrix instead of the generator matrix.

- QuadraticResidueCodeEvenPair, QuadraticResidueCodeOddPair: Quadratic residue codes of a given odd prime length and base ring either don't exist at all or occur as 4-tuples - a pair of "odd-like" codes and a pair of "even-like" codes. If n 2 is prime then (Theorem 6.6.2 in [HP]) a QR code exists over GF(q) iff q is a quadratic residue mod n. Here 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. QuadraticResidueCode (a special case) and ExtendedQuadraticResidueCode are included as well.
- RandomLinearCode - Repeatedly applies Sage's random_element applied to the ambient MatrixSpace of the generator matrix until a full rank matrix is found.
- ReedSolomonCode - Given a finite field F of order q, let n and k be chosen such that $1 \leq k \leq n \leq q$. Pick n distinct elements of F, denoted $\{x_1, x_2, ..., x_n\}$. Then, the codewords are obtained by evaluating every polynomial in $F[x]$ of degree less than k at each x_i.
- ToricCode - Let P denote a list of lattice points in $\mathbf{Z}^d$ and let T denote a listing of all points in $(F^x)^d$. Put $n = |T|$ and let k denote the dimension of the vector space of functions $V = \mathrm{Span}\{x^e \mid e \in P\}$. The associated toric code C is the evaluation code which is the image of the evaluation map $eval_T : V \to F^n$, where x^e is the multi-index notation.
- WalshCode - a binary linear $[2^m, m, 2^{m-1}]$ code related to Hadamard matrices. http://en.wikipedia.org/wiki/Walsh_code

REFERENCES:

AUTHOR:

- David Joyner (2007-05): initial version
- " (2008-02): added cyclic codes, Hamming codes
- " (2008-03): added BCH code, LinearCodeFromCheckmatrix, ReedSolomonCode, WalshCode, DuadicCodeEvenPair, DuadicCodeOddPair, QR codes (even and odd)
- " (2008-09) fix for bug in BCHCode reported by F. Voloch
- " (2008-10) small docstring changes to WalshCode and walsh_matrix

3.1 Functions

sage.coding.code_constructions.**BCHCode**(*n*, *delta*, *F*, *b=0*)

A 'Bose-Chaudhuri-Hockenghem code' (or BCH code for short) is the largest possible cyclic code of length n over field F=GF(q), whose generator polynomial has zeros (which contain the set) $Z = \{a^b, a^{b+1}, ..., a^{b+delta-2}\}$, where a is a primitive n^{th} root of unity in the splitting field $GF(q^m)$, b is an integer $0 \leq b \leq n - delta + 1$ and m is the multiplicative order of q modulo n. (The integers $b, ..., b + delta - 2$ typically lie in the range $1, ..., n - 1$.) The integer $delta \geq 1$ is called the "designed distance". The length n of the code and the size q of the base field must be relatively prime. The generator polynomial is equal to the least common multiple of the minimal polynomials of the elements of the set Z above.

Special cases are b=1 (resulting codes are called 'narrow-sense' BCH codes), and $n = q^m - 1$ (known as 'primitive' BCH codes).

It may happen that several values of delta give rise to the same BCH code. The largest one is called the Bose distance of the code. The true minimum distance, d, of the code is greater than or equal to the Bose distance, so $d \geq delta$.

EXAMPLES:

```
sage: FF.<a> = GF(3^2,"a")
sage: x = PolynomialRing(FF,"x").gen()
sage: L = [b.minpoly() for b in [a,a^2,a^3]]; g = LCM(L)
```

```
sage: f = x^(8)-1
sage: g.divides(f)
True
sage: C = codes.CyclicCode(8,g); C
Linear code of length 8, dimension 4 over Finite Field of size 3
sage: C.minimum_distance()
4
sage: C = codes.BCHCode(8,3,GF(3),1); C
Linear code of length 8, dimension 4 over Finite Field of size 3
sage: C.minimum_distance()
4
sage: C = codes.BCHCode(8,3,GF(3)); C
Linear code of length 8, dimension 5 over Finite Field of size 3
sage: C.minimum_distance()
3
sage: C = codes.BCHCode(26, 5, GF(5), b=1); C
Linear code of length 26, dimension 10 over Finite Field of size 5
```

sage.coding.code_constructions.**BinaryGolayCode**()

BinaryGolayCode() returns a binary Golay code. This is a perfect [23,12,7] code. It is also (equivalent to) a cyclic code, with generator polynomial $g(x) = 1 + x^2 + x^4 + x^5 + x^6 + x^{10} + x^{11}$. Extending it yields the extended Golay code (see ExtendedBinaryGolayCode).

EXAMPLE:

```
sage: C = codes.BinaryGolayCode()
sage: C
Linear code of length 23, dimension 12 over Finite Field of size 2
sage: C.minimum_distance()
7
sage: C.minimum_distance(algorithm='gap') # long time, check d=7
7
```

AUTHORS:

- David Joyner (2007-05)

sage.coding.code_constructions.**CyclicCode**(*n*, *g*, *ignore=True*)

If g is a polynomial over GF(q) which divides $x^n - 1$ then this constructs the code "generated by g" (ie, the code associated with the principle ideal gR in the ring $R = GF(q)[x]/(x^n - 1)$ in the usual way).

The option "ignore" says to ignore the condition that (a) the characteristic of the base field does not divide the length (the usual assumption in the theory of cyclic codes), and (b) g must divide $x^n - 1$. If ignore=True, instead of returning an error, a code generated by $gcd(x^n - 1, g)$ is created.

EXAMPLES:

```
sage: P.<x> = PolynomialRing(GF(3),"x")
sage: g = x-1
sage: C = codes.CyclicCodeFromGeneratingPolynomial(4,g); C
Linear code of length 4, dimension 3 over Finite Field of size 3
sage: P.<x> = PolynomialRing(GF(4,"a"),"x")
sage: g = x^3+1
sage: C = codes.CyclicCodeFromGeneratingPolynomial(9,g); C
Linear code of length 9, dimension 6 over Finite Field in a of size 2^2
sage: P.<x> = PolynomialRing(GF(2),"x")
sage: g = x^3+x+1
sage: C = codes.CyclicCodeFromGeneratingPolynomial(7,g); C
Linear code of length 7, dimension 4 over Finite Field of size 2
sage: C.generator_matrix()
```

```
[1 1 0 1 0 0 0]
[0 1 1 0 1 0 0]
[0 0 1 1 0 1 0]
[0 0 0 1 1 0 1]
sage: g = x+1
sage: C = codes.CyclicCodeFromGeneratingPolynomial(4,g); C
Linear code of length 4, dimension 3 over Finite Field of size 2
sage: C.generator_matrix()
[1 1 0 0]
[0 1 1 0]
[0 0 1 1]
```

On the other hand, CyclicCodeFromPolynomial(4,x) will produce a ValueError including a traceback error message: "x must divide $x^4 - 1$". You will also get a ValueError if you type

```
sage: P.<x> = PolynomialRing(GF(4,"a"),"x")
sage: g = x^2+1
```

followed by CyclicCodeFromGeneratingPolynomial(6,g). You will also get a ValueError if you type

```
sage: P.<x> = PolynomialRing(GF(3),"x")
sage: g = x^2-1
sage: C = codes.CyclicCodeFromGeneratingPolynomial(5,g); C
Linear code of length 5, dimension 4 over Finite Field of size 3
```

followed by C = CyclicCodeFromGeneratingPolynomial(5,g,False), with a traceback message including "x^2+2 must divide $x^5 - 1$".

sage.coding.code_constructions.**CyclicCodeFromCheckPolynomial**(*n*, *h*, *ignore=True*)

If h is a polynomial over GF(q) which divides $x^n - 1$ then this constructs the code "generated by $g = (x^n - 1)/h$" (ie, the code associated with the principle ideal gR in the ring $R = GF(q)[x]/(x^n - 1)$ in the usual way). The option "ignore" says to ignore the condition that the characteristic of the base field does not divide the length (the usual assumption in the theory of cyclic codes).

EXAMPLES:

```
sage: P.<x> = PolynomialRing(GF(3),"x")
sage: C = codes.CyclicCodeFromCheckPolynomial(4,x + 1); C
Linear code of length 4, dimension 1 over Finite Field of size 3
sage: C = codes.CyclicCodeFromCheckPolynomial(4,x^3 + x^2 + x + 1); C
Linear code of length 4, dimension 3 over Finite Field of size 3
sage: C.generator_matrix()
[2 1 0 0]
[0 2 1 0]
[0 0 2 1]
```

sage.coding.code_constructions.**CyclicCodeFromGeneratingPolynomial**(*n*, *g*, *ignore=True*)

If g is a polynomial over GF(q) which divides $x^n - 1$ then this constructs the code "generated by g" (ie, the code associated with the principle ideal gR in the ring $R = GF(q)[x]/(x^n - 1)$ in the usual way).

The option "ignore" says to ignore the condition that (a) the characteristic of the base field does not divide the length (the usual assumption in the theory of cyclic codes), and (b) g must divide $x^n - 1$. If ignore=True, instead of returning an error, a code generated by $gcd(x^n - 1, g)$ is created.

EXAMPLES:

```
sage: P.<x> = PolynomialRing(GF(3),"x")
sage: g = x-1
sage: C = codes.CyclicCodeFromGeneratingPolynomial(4,g); C
```

```
Linear code of length 4, dimension 3 over Finite Field of size 3
sage: P.<x> = PolynomialRing(GF(4,"a"),"x")
sage: g = x^3+1
sage: C = codes.CyclicCodeFromGeneratingPolynomial(9,g); C
Linear code of length 9, dimension 6 over Finite Field in a of size 2^2
sage: P.<x> = PolynomialRing(GF(2),"x")
sage: g = x^3+x+1
sage: C = codes.CyclicCodeFromGeneratingPolynomial(7,g); C
Linear code of length 7, dimension 4 over Finite Field of size 2
sage: C.generator_matrix()
[1 1 0 1 0 0 0]
[0 1 1 0 1 0 0]
[0 0 1 1 0 1 0]
[0 0 0 1 1 0 1]
sage: g = x+1
sage: C = codes.CyclicCodeFromGeneratingPolynomial(4,g); C
Linear code of length 4, dimension 3 over Finite Field of size 2
sage: C.generator_matrix()
[1 1 0 0]
[0 1 1 0]
[0 0 1 1]
```

On the other hand, CyclicCodeFromPolynomial(4,x) will produce a ValueError including a traceback error message: "x must divide $x^4 - 1$". You will also get a ValueError if you type

```
sage: P.<x> = PolynomialRing(GF(4,"a"),"x")
sage: g = x^2+1
```

followed by CyclicCodeFromGeneratingPolynomial(6,g). You will also get a ValueError if you type

```
sage: P.<x> = PolynomialRing(GF(3),"x")
sage: g = x^2-1
sage: C = codes.CyclicCodeFromGeneratingPolynomial(5,g); C
Linear code of length 5, dimension 4 over Finite Field of size 3
```

followed by C = CyclicCodeFromGeneratingPolynomial(5,g,False), with a traceback message including "x^2+2 must divide $x^5 - 1$".

sage.coding.code_constructions.**DuadicCodeEvenPair**(*F*, *S1*, *S2*)

Constructs the "even pair" of duadic codes associated to the "splitting" (see the docstring for `is_a_splitting` for the definition) S1, S2 of n.

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

EXAMPLES:

```
sage: from sage.coding.code_constructions import is_a_splitting
sage: n = 11; q = 3
sage: C = Zmod(n).cyclotomic_cosets(q); C
[[0], [1, 3, 4, 5, 9], [2, 6, 7, 8, 10]]
sage: S1 = C[1]
sage: S2 = C[2]
sage: is_a_splitting(S1,S2,11)
True
sage: codes.DuadicCodeEvenPair(GF(q),S1,S2)
(Linear code of length 11, dimension 5 over Finite Field of size 3,
 Linear code of length 11, dimension 5 over Finite Field of size 3)
```

sage.coding.code_constructions.**DuadicCodeOddPair**(*F*, *S1*, *S2*)
Constructs the "odd pair" of duadic codes associated to the "splitting" S1, S2 of n.

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

EXAMPLES:

```
sage: from sage.coding.code_constructions import is_a_splitting
sage: n = 11; q = 3
sage: C = Zmod(n).cyclotomic_cosets(q); C
[[0], [1, 3, 4, 5, 9], [2, 6, 7, 8, 10]]
sage: S1 = C[1]
sage: S2 = C[2]
sage: is_a_splitting(S1,S2,11)
True
sage: codes.DuadicCodeOddPair(GF(q),S1,S2)
(Linear code of length 11, dimension 6 over Finite Field of size 3,
 Linear code of length 11, dimension 6 over Finite Field of size 3)
```

This is consistent with Theorem 6.1.3 in [HP].

sage.coding.code_constructions.**ExtendedBinaryGolayCode**()
ExtendedBinaryGolayCode() returns the extended binary Golay code. This is a perfect [24,12,8] code. This code is self-dual.

EXAMPLES:

```
sage: C = codes.ExtendedBinaryGolayCode()
sage: C
Linear code of length 24, dimension 12 over Finite Field of size 2
sage: C.minimum_distance()
8
sage: C.minimum_distance(algorithm='gap') # long time, check d=8
8
```

AUTHORS:

- David Joyner (2007-05)

sage.coding.code_constructions.**ExtendedQuadraticResidueCode**(*n*, *F*)
The extended quadratic residue code (or XQR code) is obtained from a QR code by adding a check bit to the last coordinate. (These codes have very remarkable properties such as large automorphism groups and duality properties - see [HP], Section 6.6.3-6.6.4.)

INPUT:

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

OUTPUT: Returns an extended quadratic residue code.

EXAMPLES:

```
sage: C1 = codes.QuadraticResidueCode(7,GF(2))
sage: C2 = C1.extended_code()
sage: C3 = codes.ExtendedQuadraticResidueCode(7,GF(2)); C3
Linear code of length 8, dimension 4 over Finite Field of size 2
sage: C2 == C3
True
sage: C = codes.ExtendedQuadraticResidueCode(17,GF(2))
```

```
sage: C
Linear code of length 18, dimension 9 over Finite Field of size 2
sage: C3 = codes.QuadraticResidueCodeOddPair(7,GF(2))[0]
sage: C3x = C3.extended_code()
sage: C4 = codes.ExtendedQuadraticResidueCode(7,GF(2))
sage: C3x == C4
True
```

AUTHORS:

- David Joyner (07-2006)

sage.coding.code_constructions.**ExtendedTernaryGolayCode**()

ExtendedTernaryGolayCode returns a ternary Golay code. This is a self-dual perfect [12,6,6] code.

EXAMPLES:

```
sage: C = codes.ExtendedTernaryGolayCode()
sage: C
Linear code of length 12, dimension 6 over Finite Field of size 3
sage: C.minimum_distance()
6
sage: C.minimum_distance(algorithm='gap') # long time, check d=6
6
```

AUTHORS:

- David Joyner (11-2005)

sage.coding.code_constructions.**HammingCode**(r, F)

Implements the Hamming codes.

The r^{th} Hamming code over $F = GF(q)$ is an $[n, k, d]$ code with length $n = (q^r - 1)/(q - 1)$, dimension $k = (q^r - 1)/(q - 1) - r$ and minimum distance $d = 3$. The parity check matrix of a Hamming code has rows consisting of all nonzero vectors of length r in its columns, modulo a scalar factor so no parallel columns arise. A Hamming code is a single error-correcting code.

INPUT:

- `r` - an integer 2
- `F` - a finite field.

OUTPUT: Returns the r-th q-ary Hamming code.

EXAMPLES:

```
sage: codes.HammingCode(3,GF(2))
Linear code of length 7, dimension 4 over Finite Field of size 2
sage: C = codes.HammingCode(3,GF(3)); C
Linear code of length 13, dimension 10 over Finite Field of size 3
sage: C.minimum_distance()
3
sage: C.minimum_distance(algorithm='gap') # long time, check d=3
3
sage: C = codes.HammingCode(3,GF(4,'a')); C
Linear code of length 21, dimension 18 over Finite Field in a of size 2^2
```

While the `codes` object now gathers all code constructors, `HammingCode` is still available in the global namespace:

```
sage: HammingCode(3,GF(2))
doctest:...: DeprecationWarning: This method soon will not be available in that way anymore. To
```

```
See http://trac.sagemath.org/15445 for details.
Linear code of length 7, dimension 4 over Finite Field of size 2
```

sage.coding.code_constructions.**LinearCodeFromCheckMatrix**(*H*)

A linear [n,k]-code C is uniquely determined by its generator matrix G and check matrix H. We have the following short exact sequence

$$0 \to \mathbf{F}^k \xrightarrow{G} \mathbf{F}^n \xrightarrow{H} \mathbf{F}^{n-k} \to 0.$$

("Short exact" means (a) the arrow G is injective, i.e., G is a full-rank $k \times n$ matrix, (b) the arrow H is surjective, and (c) $\text{image}(G) = \text{kernel}(H)$.)

EXAMPLES:

```
sage: C = codes.HammingCode(3,GF(2))
sage: H = C.parity_check_matrix(); H
[1 0 1 0 1 0 1]
[0 1 1 0 0 1 1]
[0 0 0 1 1 1 1]
sage: codes.LinearCodeFromCheckMatrix(H) == C
True
sage: C = codes.HammingCode(2,GF(3))
sage: H = C.parity_check_matrix(); H
[1 0 1 1]
[0 1 1 2]
sage: codes.LinearCodeFromCheckMatrix(H) == C
True
sage: C = codes.RandomLinearCode(10,5,GF(4,"a"))
sage: H = C.parity_check_matrix()
sage: codes.LinearCodeFromCheckMatrix(H) == C
True
```

sage.coding.code_constructions.**QuadraticResidueCode**(*n*, *F*)

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

See QuadraticResidueCodeEvenPair and QuadraticResidueCodeOddPair for a more general construction.

INPUT:

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

OUTPUT: Returns a quadratic residue code.

EXAMPLES:

```
sage: C = codes.QuadraticResidueCode(7,GF(2))
sage: C
Linear code of length 7, dimension 4 over Finite Field of size 2
sage: C = codes.QuadraticResidueCode(17,GF(2))
sage: C
Linear code of length 17, dimension 9 over Finite Field of size 2
sage: C1 = codes.QuadraticResidueCodeOddPair(7,GF(2))[0]
sage: C2 = codes.QuadraticResidueCode(7,GF(2))
sage: C1 == C2
True
sage: C1 = codes.QuadraticResidueCodeOddPair(17,GF(2))[0]
sage: C2 = codes.QuadraticResidueCode(17,GF(2))
sage: C1 == C2
True
```

AUTHORS:

- David Joyner (11-2005)

sage.coding.code_constructions.**QuadraticResidueCodeEvenPair**(n, F)

Quadratic residue codes of a given odd prime length and base ring either don't exist at all or occur as 4-tuples - a pair of "odd-like" codes and a pair of "even-like" codes. If $n > 2$ is prime then (Theorem 6.6.2 in [HP]) a QR code exists over $GF(q)$ iff q is a quadratic residue mod n.

They are constructed as "even-like" duadic codes associated the splitting (Q,N) mod n, where Q is the set of non-zero quadratic residues and N is the non-residues.

EXAMPLES:

```
sage: codes.QuadraticResidueCodeEvenPair(17,GF(13))
(Linear code of length 17, dimension 8 over Finite Field of size 13,
 Linear code of length 17, dimension 8 over Finite Field of size 13)
sage: codes.QuadraticResidueCodeEvenPair(17,GF(2))
(Linear code of length 17, dimension 8 over Finite Field of size 2,
 Linear code of length 17, dimension 8 over Finite Field of size 2)
sage: codes.QuadraticResidueCodeEvenPair(13,GF(9,"z"))
(Linear code of length 13, dimension 6 over Finite Field in z of size 3^2,
 Linear code of length 13, dimension 6 over Finite Field in z of size 3^2)
sage: C1,C2 = codes.QuadraticResidueCodeEvenPair(7,GF(2))
sage: C1.is_self_orthogonal()
True
sage: C2.is_self_orthogonal()
True
sage: C3 = codes.QuadraticResidueCodeOddPair(17,GF(2))[0]
sage: C4 = codes.QuadraticResidueCodeEvenPair(17,GF(2))[1]
sage: C3 == C4.dual_code()
True
```

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

TESTS:

```
sage: codes.QuadraticResidueCodeEvenPair(14,Zmod(4))
Traceback (most recent call last):
...
ValueError: the argument F must be a finite field
sage: codes.QuadraticResidueCodeEvenPair(14,GF(2))
Traceback (most recent call last):
...
ValueError: the argument n must be an odd prime
sage: codes.QuadraticResidueCodeEvenPair(5,GF(2))
Traceback (most recent call last):
...
ValueError: the order of the finite field must be a quadratic residue modulo n
```

sage.coding.code_constructions.**QuadraticResidueCodeOddPair**(n, F)

Quadratic residue codes of a given odd prime length and base ring either don't exist at all or occur as 4-tuples - a pair of "odd-like" codes and a pair of "even-like" codes. If n 2 is prime then (Theorem 6.6.2 in [HP]) a QR code exists over GF(q) iff q is a quadratic residue mod n.

They are constructed as "odd-like" duadic codes associated the splitting (Q,N) mod n, where Q is the set of non-zero quadratic residues and N is the non-residues.

EXAMPLES:

```
sage: codes.QuadraticResidueCodeOddPair(17,GF(13))
(Linear code of length 17, dimension 9 over Finite Field of size 13,
 Linear code of length 17, dimension 9 over Finite Field of size 13)
sage: codes.QuadraticResidueCodeOddPair(17,GF(2))
(Linear code of length 17, dimension 9 over Finite Field of size 2,
 Linear code of length 17, dimension 9 over Finite Field of size 2)
sage: codes.QuadraticResidueCodeOddPair(13,GF(9,"z"))
(Linear code of length 13, dimension 7 over Finite Field in z of size 3^2,
 Linear code of length 13, dimension 7 over Finite Field in z of size 3^2)
sage: C1 = codes.QuadraticResidueCodeOddPair(17,GF(2))[1]
sage: C1x = C1.extended_code()
sage: C2 = codes.QuadraticResidueCodeOddPair(17,GF(2))[0]
sage: C2x = C2.extended_code()
sage: C2x.spectrum(); C1x.spectrum()
[1, 0, 0, 0, 0, 0, 102, 0, 153, 0, 153, 0, 102, 0, 0, 0, 0, 0, 1]
[1, 0, 0, 0, 0, 0, 102, 0, 153, 0, 153, 0, 102, 0, 0, 0, 0, 0, 1]
sage: C2x == C1x.dual_code()
True
sage: C3 = codes.QuadraticResidueCodeOddPair(7,GF(2))[0]
sage: C3x = C3.extended_code()
sage: C3x.spectrum()
[1, 0, 0, 0, 14, 0, 0, 0, 1]
sage: C3x.is_self_dual()
True
```

This is consistent with Theorem 6.6.14 in [HP].

TESTS:

```
sage: codes.QuadraticResidueCodeOddPair(9,GF(2))
Traceback (most recent call last):
...
ValueError: the argument n must be an odd prime
```

sage.coding.code_constructions.**RandomLinearCode**(*n*, *k*, *F*)

The method used is to first construct a $k \times n$ matrix using Sage's random_element method for the MatrixSpace class. The construction is probabilistic but should only fail extremely rarely.

INPUT: Integers n,k, with $n > k$, and a finite field F

OUTPUT: Returns a "random" linear code with length n, dimension k over field F.

EXAMPLES:

```
sage: C = codes.RandomLinearCode(30,15,GF(2))
sage: C
Linear code of length 30, dimension 15 over Finite Field of size 2
sage: C = codes.RandomLinearCode(10,5,GF(4,'a'))
sage: C
Linear code of length 10, dimension 5 over Finite Field in a of size 2^2
```

AUTHORS:

- David Joyner (2007-05)

sage.coding.code_constructions.**ReedSolomonCode**(*n*, *k*, *F*, *pts=None*)

Given a finite field F of order q, let n and k be chosen such that $1 \leq k \leq n \leq q$. Pick n distinct elements of F, denoted $\{x_1, x_2, ..., x_n\}$. Then, the codewords are obtained by evaluating every polynomial in $F[x]$ of degree less than k at each x_i:

$$C = \{(f(x_1), f(x_2), ..., f(x_n)), f \in F[x], \deg(f) < k\}.$$

C is a $[n, k, n-k+1]$ code. (In particular, C is MDS.)

INPUT: n : the length k : the dimension F : the base ring pts : (optional) list of n points in F (if None then Sage picks n of them in the order given to the elements of F)

EXAMPLES:

```
sage: C = codes.ReedSolomonCode(6,4,GF(7)); C
Linear code of length 6, dimension 4 over Finite Field of size 7
sage: C.minimum_distance()
3
sage: C = codes.ReedSolomonCode(6,4,GF(8,"a")); C
Linear code of length 6, dimension 4 over Finite Field in a of size 2^3
sage: C.minimum_distance()
3
sage: C.minimum_distance(algorithm='gap') # long time, check d=n-k+1
3
sage: F.<a> = GF(3^2,"a")
sage: pts = [0,1,a,a^2,2*a,2*a+1]
sage: len(Set(pts)) == 6 # to make sure there are no duplicates
True
sage: C = codes.ReedSolomonCode(6,4,F,pts); C
Linear code of length 6, dimension 4 over Finite Field in a of size 3^2
sage: C.minimum_distance()
3
```

While the `codes` object now gathers all code constructors, `ReedSolomonCode` is still available in the global namespace:

```
sage: ReedSolomonCode(6,4,GF(7))
doctest:...: DeprecationWarning: This method soon will not be available in that way anymore. To
See http://trac.sagemath.org/15445 for details.
Linear code of length 6, dimension 4 over Finite Field of size 7
```

REFERENCES:

- [W] http://en.wikipedia.org/wiki/Reed-Solomon

`sage.coding.code_constructions.`**`TernaryGolayCode`**`()`

TernaryGolayCode returns a ternary Golay code. This is a perfect [11,6,5] code. It is also equivalent to a cyclic code, with generator polynomial $g(x) = 2 + x^2 + 2x^3 + x^4 + x^5$.

EXAMPLES:

```
sage: C = codes.TernaryGolayCode()
sage: C
Linear code of length 11, dimension 6 over Finite Field of size 3
sage: C.minimum_distance()
5
sage: C.minimum_distance(algorithm='gap') # long time, check d=5
5
```

AUTHORS:

- David Joyner (2007-5)

`sage.coding.code_constructions.`**`ToricCode`**`(`P, F`)`

Let P denote a list of lattice points in $\mathbf{Z}^d$ and let T denote the set of all points in $(F^x)^d$ (ordered in some fixed way). Put $n = |T|$ and let k denote the dimension of the vector space of functions $V = \mathrm{Span}\{x^e \mid e \in P\}$. The associated toric code C is the evaluation code which is the image of the evaluation map

$$\mathrm{eval}_T : V \to F^n,$$

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

INPUT:

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

OUTPUT: Returns toric code with length n = , dimension k over field F.

EXAMPLES:

```
sage: C = codes.ToricCode([[0,0],[1,0],[2,0],[0,1],[1,1]],GF(7))
sage: C
Linear code of length 36, dimension 5 over Finite Field of size 7
sage: C.minimum_distance()
24
sage: C = codes.ToricCode([[-2,-2],[-1,-2],[-1,-1],[-1,0],[0,-1],[0,0],[0,1],[1,-1],[1,0]],GF(5)
sage: C
Linear code of length 16, dimension 9 over Finite Field of size 5
sage: C.minimum_distance()
6
sage: C = codes.ToricCode([ [0,0],[1,1],[1,2],[1,3],[1,4],[2,1],[2,2],[2,3],[3,1],[3,2],[4,1]],G
sage: C
Linear code of length 49, dimension 11 over Finite Field in a of size 2^3
```

This is in fact a [49,11,28] code over GF(8). If you type next `C.minimum_distance()` and wait overnight (!), you should get 28.

AUTHOR:

- David Joyner (07-2006)

REFERENCES:

sage.coding.code_constructions.**TrivialCode**(F, n)

x.__init__(...) initializes x; see help(type(x)) for signature

sage.coding.code_constructions.**WalshCode**(m)

Returns the binary Walsh code of length 2^m. The matrix of codewords correspond to a Hadamard matrix. This is a (constant rate) binary linear $[2^m, m, 2^{m-1}]$ code.

EXAMPLES:

```
sage: C = codes.WalshCode(4); C
Linear code of length 16, dimension 4 over Finite Field of size 2
sage: C = codes.WalshCode(3); C
Linear code of length 8, dimension 3 over Finite Field of size 2
sage: C.spectrum()
[1, 0, 0, 0, 7, 0, 0, 0]
sage: C.minimum_distance()
4
sage: C.minimum_distance(algorithm='gap') # check d=2^(m-1)
4
```

REFERENCES:

- http://en.wikipedia.org/wiki/Hadamard_matrix
- http://en.wikipedia.org/wiki/Walsh_code

sage.coding.code_constructions.**cyclotomic_cosets**(*q*, *n*, *t=None*)

This method is deprecated.

See the documentation in `cyclotomic_cosets()`.

INPUT: q,n,t positive integers (or t=None) Some type-checking of inputs is performed.

OUTPUT: q-cyclotomic cosets mod n (or, if t is not None, the q-cyclotomic coset mod n containing t)

EXAMPLES:

```
sage: cyclotomic_cosets(2,11)
doctest:...: DeprecationWarning: cyclotomic_cosets(q,n,t) is deprecated.
Use Zmod(n).cyclotomic_cosets(q) or Zmod(n).cyclotomic_cosets(q,[t])
instead. Be careful that this method returns elements of Zmod(n).
See http://trac.sagemath.org/16464 for details.
[[0], [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]]

sage: Zmod(11).cyclotomic_cosets(2)
[[0], [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]]

sage: cyclotomic_cosets(5,11)
[[0], [1, 3, 4, 5, 9], [2, 6, 7, 8, 10]]
sage: cyclotomic_cosets(5,11,3)
[1, 3, 4, 5, 9]
```

sage.coding.code_constructions.**is_a_splitting**(*S1*, *S2*, *n*, *return_automorphism=False*)

Check wether `(S1,S2)` is a splitting of $\mathbf{Z}/n\mathbf{Z}$.

A splitting of $R = \mathbf{Z}/n\mathbf{Z}$ is a pair of subsets of R which is a partition of $R\backslash\{0\}$ and such that there exists an element r of R such that $rS_1 = S_2$ and $rS_2 = S_1$ (where rS is the point-wise multiplication of the elements of S by r).

Splittings are useful for computing idempotents in the quotient ring $Q = GF(q)[x]/(x^n - 1)$.

INPUT:

- `S1, S2` – disjoint sublists partitioning `[1, 2, ..., n-1]`
- `n` (integer)
- `return_automorphism` (boolean) – whether to return the automorphism exchanging S_1 and S_2.

OUTPUT:

If `return_automorphism is False` (default) the function returns boolean values.

Otherwise, it returns a pair `(b, r)` where `b` is a boolean indicating whether $S1$, $S2$ is a splitting of n, and r is such that $rS_1 = S_2$ and $rS_2 = S_1$ (if b is `False`, r is equal to `None`).

EXAMPLES:

```
sage: from sage.coding.code_constructions import is_a_splitting
sage: is_a_splitting([1,2],[3,4],5)
True
sage: is_a_splitting([1,2],[3,4],5,return_automorphism=True)
(True, 4)

sage: is_a_splitting([1,3],[2,4,5,6],7)
False
sage: is_a_splitting([1,3,4],[2,5,6],7)
False

sage: for P in SetPartitions(6,[3,3]):
```

```
....:     res,aut= is_a_splitting(P[0],P[1],7,return_automorphism=True)
....:     if res:
....:         print aut, P[0], P[1]
6 {1, 2, 3} {4, 5, 6}
3 {1, 2, 4} {3, 5, 6}
6 {1, 3, 5} {2, 4, 6}
6 {1, 4, 5} {2, 3, 6}
```

We illustrate now how to find idempotents in quotient rings:

```
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: F = GF(q)
sage: P.<x> = PolynomialRing(F,"x")
sage: I = Ideal(P,[x^n-1])
sage: Q.<x> = QuotientRing(P,I)
sage: i1 = -sum([x^i for i in S1]); i1
2*x^9 + 2*x^5 + 2*x^4 + 2*x^3 + 2*x
sage: i2 = -sum([x^i for i in S2]); i2
2*x^10 + 2*x^8 + 2*x^7 + 2*x^6 + 2*x^2
sage: i1^2 == i1
True
sage: i2^2 == i2
True
sage: (1-i1)^2 == 1-i1
True
sage: (1-i2)^2 == 1-i2
True
```

We return to dealing with polynomials (rather than elements of quotient rings), so we can construct cyclic codes:

```
sage: P.<x> = PolynomialRing(F,"x")
sage: i1 = -sum([x^i for i in S1])
sage: i2 = -sum([x^i for i in S2])
sage: i1_sqrd = (i1^2).quo_rem(x^n-1)[1]
sage: i1_sqrd  == i1
True
sage: i2_sqrd = (i2^2).quo_rem(x^n-1)[1]
sage: i2_sqrd  == i2
True
sage: C1 = codes.CyclicCodeFromGeneratingPolynomial(n,i1)
sage: C2 = codes.CyclicCodeFromGeneratingPolynomial(n,1-i2)
sage: C1.dual_code() == C2
True
```

This is a special case of Theorem 6.4.3 in [HP].

sage.coding.code_constructions.**lift2smallest_field**(*a*)

INPUT: a is an element of a finite field GF(q)

OUTPUT: the element b of the smallest subfield F of GF(q) for which F(b)=a.

EXAMPLES:

```
sage: from sage.coding.code_constructions import lift2smallest_field
sage: FF.<z> = GF(3^4,"z")
sage: a = z^10
sage: lift2smallest_field(a)
(2*z + 1, Finite Field in z of size 3^2)
sage: a = z^40
sage: lift2smallest_field(a)
(2, Finite Field of size 3)
```

AUTHORS:

- John Cremona

sage.coding.code_constructions.**lift2smallest_field2**(*a*)

INPUT: a is an element of a finite field GF(q) OUTPUT: the element b of the smallest subfield F of GF(q) for which F(b)=a.

EXAMPLES:

```
sage: from sage.coding.code_constructions import lift2smallest_field2
sage: FF.<z> = GF(3^4,"z")
sage: a = z^40
sage: lift2smallest_field2(a)
(2, Finite Field of size 3)
sage: FF.<z> = GF(2^4,"z")
sage: a = z^15
sage: lift2smallest_field2(a)
(1, Finite Field of size 2)
```

Warning: Since coercion (the FF(b) step) has a bug in it, this *only works* in the case when you *know* F is a prime field.

AUTHORS:

- David Joyner

sage.coding.code_constructions.**permutation_action**(*g*, *v*)

Returns permutation of rows g*v. Works on lists, matrices, sequences and vectors (by permuting coordinates). The code requires switching from i to i+1 (and back again) since the SymmetricGroup is, by convention, the symmetric group on the "letters" 1, 2, ..., n (not 0, 1, ..., n-1).

EXAMPLES:

```
sage: V = VectorSpace(GF(3),5)
sage: v = V([0,1,2,0,1])
sage: G = SymmetricGroup(5)
sage: g = G([(1,2,3)])
sage: permutation_action(g,v)
(1, 2, 0, 0, 1)
sage: g = G([()])
sage: permutation_action(g,v)
(0, 1, 2, 0, 1)
sage: g = G([(1,2,3,4,5)])
sage: permutation_action(g,v)
(1, 2, 0, 1, 0)
sage: L = Sequence([1,2,3,4,5])
sage: permutation_action(g,L)
[2, 3, 4, 5, 1]
sage: MS = MatrixSpace(GF(3),3,7)
sage: A = MS([[1,0,0,0,1,1,0],[0,1,0,1,0,1,0],[0,0,0,0,0,0,1]])
```

```
sage: S5 = SymmetricGroup(5)
sage: g = S5([(1,2,3)])
sage: A
[1 0 0 0 1 1 0]
[0 1 0 1 0 1 0]
[0 0 0 0 0 0 1]
sage: permutation_action(g,A)
[0 1 0 1 0 1 0]
[0 0 0 0 0 0 1]
[1 0 0 0 1 1 0]
```

It also works on lists and is a "left action":

```
sage: v = [0,1,2,0,1]
sage: G = SymmetricGroup(5)
sage: g = G([(1,2,3)])
sage: gv = permutation_action(g,v); gv
[1, 2, 0, 0, 1]
sage: permutation_action(g,v) == g(v)
True
sage: h = G([(3,4)])
sage: gv = permutation_action(g,v)
sage: hgv = permutation_action(h,gv)
sage: hgv == permutation_action(h*g,v)
True
```

AUTHORS:

- David Joyner, licensed under the GPL v2 or greater.

sage.coding.code_constructions.**walsh_matrix**(*m0*)

This is the generator matrix of a Walsh code. The matrix of codewords correspond to a Hadamard matrix.

EXAMPLES:

```
sage: walsh_matrix(2)
[0 0 1 1]
[0 1 0 1]
sage: walsh_matrix(3)
[0 0 0 0 1 1 1 1]
[0 0 1 1 0 0 1 1]
[0 1 0 1 0 1 0 1]
sage: C = LinearCode(walsh_matrix(4)); C
Linear code of length 16, dimension 4 over Finite Field of size 2
sage: C.spectrum()
[1, 0, 0, 0, 0, 0, 0, 0, 15, 0, 0, 0, 0, 0, 0, 0, 0]
```

This last code has minimum distance 8.

REFERENCES:

- http://en.wikipedia.org/wiki/Hadamard_matrix

CHAPTER

FOUR

GUAVA ERROR-CORRECTING CODE CONSTRUCTIONS.

This module only contains Guava wrappers (Guava is an optional GAP package).

AUTHORS:

- David Joyner (2005-11-22, 2006-12-03): initial version
- Nick Alexander (2006-12-10): factor GUAVA code to guava.py
- David Joyner (2007-05): removed Golay codes, toric and trivial codes and placed them in code_constructions; renamed RandomLinearCode to RandomLinearCodeGuava
- David Joyner (2008-03): removed QR, XQR, cyclic and ReedSolomon codes
- David Joyner (2009-05): added "optional package" comments, fixed some docstrings to to be sphinx compatible

4.1 Functions

sage.coding.guava.**BinaryReedMullerCode**(*r*, *k*)

The binary 'Reed-Muller code' with dimension k and order r is a code with length 2^k and minimum distance $2^k - r$ (see for example, section 1.10 in [HP]). By definition, the r^{th} order binary Reed-Muller code of length $n = 2^m$, for $0 \leq r \leq m$, is the set of all vectors $(f(p) \mid p$ $in GF(2)^m)$, where f is a multivariate polynomial of degree at most r in m variables.

INPUT:

- `r, k` – positive integers with $2^k > r$.

OUTPUT:

Returns the binary 'Reed-Muller code' with dimension k and order r.

EXAMPLE:

```
sage: C = codes.BinaryReedMullerCode(2,4); C  # optional - gap_packages (Guava package)
Linear code of length 16, dimension 11 over Finite Field of size 2
sage: C.minimum_distance()              # optional - gap_packages (Guava package)
4
sage: C.generator_matrix()                          # optional - gap_packages (Guava package)
[1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
[0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1]
[0 0 0 0 1 1 1 1 0 0 0 0 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]
[0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1]
[0 0 0 0 0 0 0 0 0 0 1 1 0 0 1 1]
[0 0 0 0 0 0 0 0 0 1 0 1 0 1 0 1]
[0 0 0 0 0 0 1 1 0 0 0 0 0 0 1 1]
```

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

AUTHOR: David Joyner (11-2005)

sage.coding.guava.**QuasiQuadraticResidueCode**(p)

A (binary) quasi-quadratic residue code (or QQR code), as defined by Proposition 2.2 in [BM], has a generator matrix in the block form $G = (Q, N)$. Here Q is a $p \times p$ circulant matrix whose top row is $(0, x_1, ..., x_{p-1})$, where $x_i = 1$ if and only if i is a quadratic residue $\mod p$, and N is a $p \times p$ circulant matrix whose top row is $(0, y_1, ..., y_{p-1})$, where $x_i + y_i = 1$ for all i.

INPUT:

- p – a prime > 2.

OUTPUT:

Returns a QQR code of length $2p$.

EXAMPLES:

```
sage: C = codes.QuasiQuadraticResidueCode(11); C    # optional - gap_packages (Guava package)
Linear code of length 22, dimension 11 over Finite Field of size 2
```

REFERENCES:

These are self-orthogonal in general and self-dual when p
$equiv 3$
$pmod 4$.

AUTHOR: David Joyner (11-2005)

sage.coding.guava.**RandomLinearCodeGuava**(n, k, F)

The method used is to first construct a $k \times n$ matrix of the block form (I, A), where I is a $k \times k$ identity matrix and A is a $k \times (n - k)$ matrix constructed using random elements of F. Then the columns are permuted using a randomly selected element of the symmetric group S_n.

INPUT:

- n, k – integers with $n > k > 1$.

OUTPUT:

Returns a "random" linear code with length n, dimension k over field F.

EXAMPLES:

```
sage: C = codes.RandomLinearCodeGuava(30,15,GF(2)); C      # optional - gap_packages (Guava pack
Linear code of length 30, dimension 15 over Finite Field of size 2
sage: C = codes.RandomLinearCodeGuava(10,5,GF(4,'a')); C      # optional - gap_packages (Guava p
Linear code of length 10, dimension 5 over Finite Field in a of size 2^2
```

AUTHOR: David Joyner (11-2005)

CHAPTER

FIVE

BINARY SELF-DUAL CODES

This module implements functions useful for studying binary self-dual codes. The main function is `self_dual_codes_binary`, which is a case-by-case list of entries, each represented by a Python dictionary.

Format of each entry: a Python dictionary with keys "order autgp", "spectrum", "code", "Comment", "Type", where

- "code" - a sd code C of length n, dim n/2, over GF(2)
- "order autgp" - order of the permutation automorphism group of C
- "Type" - the type of C (which can be "I" or "II", in the binary case)
- "spectrum" - the spectrum [A0,A1,...,An]
- "Comment" - possibly an empty string.

Python dictionaries were used since they seemed to be both human-readable and allow others to update the database easiest.

- The following double for loop can be time-consuming but should be run once in awhile for testing purposes. It should only print True and have no trace-back errors:

```
for n in [4,6,8,10,12,14,16,18,20,22]:
     C = self_dual_codes_binary(n); m = len(C.keys())
     for i in range(m):
         C0 = C["%s"%n]["%s"%i]["code"]
         print n, '  ',i, '    ',C["%s"%n]["%s"%i]["spectrum"] == C0.spectrum()
         print C0 == C0.dual_code()
         G = C0.automorphism_group_binary_code()
         print C["%s"%n]["%s"%i]["order autgp"] == G.order()
```

- To check if the "Riemann hypothesis" holds, run the following code:

```
R = PolynomialRing(CC,"T")
T = R.gen()
for n in [4,6,8,10,12,14,16,18,20,22]:
     C = self_dual_codes_binary(n); m = len(C["%s"%n].keys())
     for i in range(m):
         C0 = C["%s"%n]["%s"%i]["code"]
         if C0.minimum_distance()>2:
             f = R(C0.sd_zeta_polynomial())
             print n,i,[z[0].abs() for z in f.roots()]
```

You should get lists of numbers equal to 0.707106781186548.

Here's a rather naive construction of self-dual codes in the binary case:

For even m, let A_m denote the mxm matrix over GF(2) given by adding the all 1's matrix to the identity matrix (in `MatrixSpace(GF(2),m,m)` of course). If M_1, ..., M_r are square matrices, let $diag(M_1, M_2, ..., M_r)$ denote

the"block diagonal" matrix with the M_i 's on the diagonal and 0's elsewhere. Let $C(m_1, ..., m_r, s)$ denote the linear code with generator matrix having block form $G = (I, A)$, where $A = diag(A_{m_1}, A_{m_2}, ..., A_{m_r}, I_s)$, for some (even) m_i 's and s, where $m_1 + m_2 + ... + m_r + s = n/2$. Note: Such codes $C(m_1, ..., m_r, s)$ are SD.

SD codes not of this form will be called (for the purpose of documenting the code below) "exceptional". Except when n is "small", most sd codes are exceptional (based on a counting argument and table 9.1 in the Huffman+Pless [HP], page 347).

AUTHORS:

- David Joyner (2007-08-11)

REFERENCES:

- [HP] W. C. Huffman, V. Pless, Fundamentals of Error-Correcting Codes, Cambridge Univ. Press, 2003.
- [P] V. Pless, "A classification of self-orthogonal codes over GF(2)", Discrete Math 3 (1972) 209-246.

`sage.coding.sd_codes.I2(n)`
x.__init__(...) initializes x; see help(type(x)) for signature

`sage.coding.sd_codes.MS(n)`
For internal use; returns the floor(n/2) x n matrix space over GF(2).

EXAMPLES:

```
sage: import sage.coding.sd_codes as sd_codes
sage: sd_codes.MS(2)
Full MatrixSpace of 1 by 2 dense matrices over Finite Field of size 2
sage: sd_codes.MS(3)
Full MatrixSpace of 1 by 3 dense matrices over Finite Field of size 2
sage: sd_codes.MS(8)
Full MatrixSpace of 4 by 8 dense matrices over Finite Field of size 2
```

`sage.coding.sd_codes.MS2(n)`
For internal use; returns the floor(n/2) x floor(n/2) matrix space over GF(2).

EXAMPLES:

```
sage: import sage.coding.sd_codes as sd_codes
sage: sd_codes.MS2(8)
Full MatrixSpace of 4 by 4 dense matrices over Finite Field of size 2
```

`sage.coding.sd_codes.matA(n)`
For internal use; returns a list of square matrices over GF(2) (a_{ij}) of sizes 0 x 0, 1 x 1, ..., n x n which are of the form $(a_{ij} = 1) + (a_{ij} = \delta_{ij})$.

EXAMPLES:

```
sage: import sage.coding.sd_codes as sd_codes
sage: sd_codes.matA(4)
[
                  [0 1 1]
          [0 1]   [1 0 1]
[], [0], [1 0], [1 1 0]
]
```

`sage.coding.sd_codes.matId(n)`
For internal use; returns a list of identity matrices over GF(2) of sizes (floor(n/2)-j) x (floor(n/2)-j) for j = 0 ... (floor(n/2)-1).

EXAMPLES:

```
sage: import sage.coding.sd_codes as sd_codes
sage: sd_codes.matId(6)
[
[1 0 0]
[0 1 0]  [1 0]
[0 0 1], [0 1], [1]
]
```

sage.coding.sd_codes.**self_dual_codes_binary**(*n*)

Returns the dictionary of inequivalent sd codes of length n.

For n=4 even, returns the sd codes of a given length, up to (perm) equivalence, the (perm) aut gp, and the type.

The number of inequiv "diagonal" sd binary codes in the database of length n is ("diagonal" is defined by the conjecture above) is the same as the restricted partition number of n, where only integers from the set 1,4,6,8,... are allowed. This is the coefficient of x^n in the series expansion $(1-x)^{-1}\prod_{2^\infty(1-x^{2j})^{-1}}$. Typing the command f = (1-x)(-1)*prod([(1-x(2*j))(-1) for j in range(2,18)]) into Sage, we obtain for the coeffs of x^4, x^6, ... [1, 1, 2, 2, 3, 3, 5, 5, 7, 7, 11, 11, 15, 15, 22, 22, 30, 30, 42, 42, 56, 56, 77, 77, 101, 101, 135, 135, 176, 176, 231] These numbers grow too slowly to account for all the sd codes (see Huffman+Pless' Table 9.1, referenced above). In fact, in Table 9.10 of [HP], the number B_n of inequivalent sd binary codes of length n is given:

```
n   2 4 6 8 10 12 14 16 18 20 22 24  26  28  30
B_n 1 1 1 2  2  3  4  7  9 16 25 55 103 261 731
```

According to http://oeis.org/classic/A003179, the next 2 entries are: 3295, 24147.

EXAMPLES:

```
sage: C = self_dual_codes_binary(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 = self_dual_codes_binary(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
```

CHAPTER

SIX

BOUNDS FOR PARAMETERS OF CODES

This module provided some upper and lower bounds for the parameters of codes.

AUTHORS:

- David Joyner (2006-07): initial implementation.
- William Stein (2006-07): minor editing of docs and code (fixed bug in elias_bound_asymp)
- David Joyner (2006-07): fixed dimension_upper_bound to return an integer, added example to elias_bound_asymp.
- " (2009-05): removed all calls to Guava but left it as an option.
- Dima Pasechnik (2012-10): added LP bounds.

Let F be a finite field (we denote the finite field with q elements by $\mathbf{F}_q$). A subset C of $V = F^n$ is called a code of length n. A subspace of V (with the standard basis) is called a linear code of length n. If its dimension is denoted k then we typically store a basis of C as a $k \times n$ matrix (the rows are the basis vectors). If $F = \mathbf{F}_2$ then C is called a binary code. If F has q elements then C is called a q-ary code. The elements of a code C are called codewords. The information rate of C is

$$R = \frac{\log_q |C|}{n},$$

where $|C|$ denotes the number of elements of C. If $\mathbf{v} = (v_1, v_2, ..., v_n)$, $\mathbf{w} = (w_1, w_2, ..., w_n)$ are vectors in $V = F^n$ then we define

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

to be the Hamming distance between $\mathbf{v}$ and $\mathbf{w}$. The function $d : V \times V \to \mathbf{N}$ is called the Hamming metric. The weight of a vector (in the Hamming metric) is $d(\mathbf{v}, \mathbf{0})$. The minimum distance of a linear code is the smallest non-zero weight of a codeword in C. The relatively minimum distance is denoted

$$\delta = d/n.$$

A linear code with length n, dimension k, and minimum distance d is called an $[n, k, d]_q$-code and n, k, d are called its parameters. A (not necessarily linear) code C with length n, size $M = |C|$, and minimum distance d is called an $(n, M, d)_q$-code (using parentheses instead of square brackets). Of course, $k = \log_q(M)$ for linear codes.

What is the "best" code of a given length? Let F be a finite field with q elements. Let $A_q(n, d)$ denote the largest M such that there exists a (n, M, d) code in F^n. Let $B_q(n, d)$ (also denoted $A_q^{lin}(n, d)$) denote the largest k such that there exists a $[n, k, d]$ code in F^n. (Of course, $A_q(n, d) \geq B_q(n, d)$.) Determining $A_q(n, d)$ and $B_q(n, d)$ is one of the main problems in the theory of error-correcting codes.

These quantities related to solving a generalization of the childhood game of "20 questions".

GAME: Player 1 secretly chooses a number from 1 to M (M is large but fixed). Player 2 asks a series of "yes/no questions" in an attempt to determine that number. Player 1 may lie at most e times ($e \geq 0$ is fixed). What is the

minimum number of "yes/no questions" Player 2 must ask to (always) be able to correctly determine the number Player 1 chose?

If feedback is not allowed (the only situation considered here), call this minimum number $g(M, e)$.

Lemma: For fixed e and M, $g(M, e)$ is the smallest n such that $A_2(n, 2e + 1) \geq M$.

Thus, solving the solving a generalization of the game of "20 questions" is equivalent to determining $A_2(n, d)$! Using Sage, you can determine the best known estimates for this number in 2 ways:

1. **Indirectly, using best_known_linear_code_www(n, k, F),** which connects to the website http://www.codetables.de by Markus Grassl;
2. **codesize_upper_bound(n,d,q), dimension_upper_bound(n,d,q),** and best_known_linear_code(n, k, F).

The output of `best_known_linear_code()`, `best_known_linear_code_www()`, or `dimension_upper_bound()` would give only special solutions to the GAME because the bounds are applicable to only linear codes. The output of `codesize_upper_bound()` would give the best possible solution, that may belong to a linear or nonlinear code.

This module implements:

- codesize_upper_bound(n,d,q), for the best known (as of May, 2006) upper bound A(n,d) for the size of a code of length n, minimum distance d over a field of size q.
- dimension_upper_bound(n,d,q), an upper bound $B(n, d) = B_q(n, d)$ for the dimension of a linear code of length n, minimum distance d over a field of size q.
- gilbert_lower_bound(n,q,d), a lower bound for number of elements in the largest code of min distance d in $\mathbf{F}_q^n$.
- gv_info_rate(n,delta,q), $log_q(GLB)/n$, where GLB is the Gilbert lower bound and delta = d/n.
- gv_bound_asymp(delta,q), asymptotic analog of Gilbert lower bound.
- plotkin_upper_bound(n,q,d)
- plotkin_bound_asymp(delta,q), asymptotic analog of Plotkin bound.
- griesmer_upper_bound(n,q,d)
- elias_upper_bound(n,q,d)
- elias_bound_asymp(delta,q), asymptotic analog of Elias bound.
- hamming_upper_bound(n,q,d)
- hamming_bound_asymp(delta,q), asymptotic analog of Hamming bound.
- singleton_upper_bound(n,q,d)
- singleton_bound_asymp(delta,q), asymptotic analog of Singleton bound.
- mrrw1_bound_asymp(delta,q), "first" asymptotic McEliese-Rumsey-Rodemich-Welsh bound for the information rate.
- Delsarte (a.k.a. Linear Programming (LP)) upper bounds.

PROBLEM: In this module we shall typically either (a) seek bounds on k, given n, d, q, (b) seek bounds on R, delta, q (assuming n is "infinity").

TODO:

- Johnson bounds for binary codes.
- mrrw2_bound_asymp(delta,q), "second" asymptotic McEliese-Rumsey-Rodemich-Welsh bound for the information rate.

REFERENCES:

- C. Huffman, V. Pless, Fundamentals of error-correcting codes, Cambridge Univ. Press, 2003.

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

This computes the minimum value of the upper bound using the methods of Singleton, Hamming, Plotkin, and Elias.

If algorithm="gap" then this returns the best known upper bound $A(n,d) = A_q(n,d)$ for the size of a code of length n, minimum distance d over a field of size q. The function first checks for trivial cases (like d=1 or n=d), and if the value is in the built-in table. Then it calculates the minimum value of the upper bound using the algorithms of Singleton, Hamming, Johnson, Plotkin and Elias. If the code is binary, $A(n, 2\ell-1) = A(n+1, 2\ell)$, so the function takes the minimum of the values obtained from all algorithms for the parameters $(n, 2\ell-1)$ and $(n+1, 2\ell)$. This wraps GUAVA's (i.e. GAP's package Guava) UpperBound(n, d, q).

If algorithm="LP" then this returns the Delsarte (a.k.a. Linear Programming) upper bound.

EXAMPLES:

```
sage: codesize_upper_bound(10,3,2)
93
sage: codesize_upper_bound(24,8,2,algorithm="LP")
4096
sage: codesize_upper_bound(10,3,2,algorithm="gap")  # optional - gap_packages (Guava package)
85
sage: codesize_upper_bound(11,3,4,algorithm=None)
123361
sage: codesize_upper_bound(11,3,4,algorithm="gap")  # optional - gap_packages (Guava package)
123361
sage: codesize_upper_bound(11,3,4,algorithm="LP")
109226
```

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

Returns an upper bound $B(n,d) = B_q(n,d)$ for the dimension of a linear code of length n, minimum distance d over a field of size q. Parameter "algorithm" has the same meaning as in `codesize_upper_bound()`

EXAMPLES:

```
sage: dimension_upper_bound(10,3,2)
6
sage: dimension_upper_bound(30,15,4)
13
sage: dimension_upper_bound(30,15,4,algorithm="LP")
12
```

sage.coding.code_bounds.**elias_bound_asymp**(*delta*, *q*)

Computes the asymptotic Elias bound for the information rate, provided $0 < \delta < 1 - 1/q$.

EXAMPLES:

```
sage: elias_bound_asymp(1/4,2)
0.39912396330...
```

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

Returns the Elias upper bound for number of elements in the largest code of minimum distance d in $\mathbf{F}_q^n$. Wraps GAP's UpperBoundElias.

EXAMPLES:

```
sage: elias_upper_bound(10,2,3)
232
sage: elias_upper_bound(10,2,3,algorithm="gap")  # optional - gap_packages (Guava package)
232
```

sage.coding.code_bounds.**entropy**(*x*, *q=2*)

Computes the entropy at x on the q-ary symmetric channel.

INPUT:

- x - real number in the interval $[0, 1]$.
- q - (default: 2) integer greater than 1. This is the base of the logarithm.

EXAMPLES:

```
sage: entropy(0, 2)
0
sage: entropy(1/5,4)
1/5*log(3)/log(4) - 4/5*log(4/5)/log(4) - 1/5*log(1/5)/log(4)
sage: entropy(1, 3)
log(2)/log(3)
```

Check that values not within the limits are properly handled:

```
sage: entropy(1.1, 2)
Traceback (most recent call last):
...
ValueError: The entropy function is defined only for x in the interval [0, 1]
sage: entropy(1, 1)
Traceback (most recent call last):
...
ValueError: The value q must be an integer greater than 1
```

sage.coding.code_bounds.**entropy_inverse**(*x*, *q=2*)

Find the inverse of the q-ary entropy function at the point x.

INPUT:

- x – real number in the interval $[0, 1]$.
- q - (default: 2) integer greater than 1. This is the base of the logarithm.

OUTPUT:

Real number in the interval $[0, 1 - 1/q]$. The function has multiple values if we include the entire interval $[0, 1]$; hence only the values in the above interval is returned.

EXAMPLES:

```
sage: from sage.coding.code_bounds import entropy_inverse
sage: entropy_inverse(0.1)
0.012986862055848683
sage: entropy_inverse(1)
1/2
sage: entropy_inverse(0, 3)
0
sage: entropy_inverse(1, 3)
2/3
```

sage.coding.code_bounds.**gilbert_lower_bound**(*n*, *q*, *d*)

Returns lower bound for number of elements in the largest code of minimum distance d in $\mathbf{F}_q^n$.

EXAMPLES:

```
sage: gilbert_lower_bound(10,2,3)
128/7
```

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

Returns the Griesmer upper bound for number of elements in the largest code of minimum distance d in $\mathbf{F}_q^n$. Wraps GAP's UpperBoundGriesmer.

EXAMPLES:

```
sage: griesmer_upper_bound(10,2,3)
128
sage: griesmer_upper_bound(10,2,3,algorithm="gap")  # optional - gap_packages (Guava package)
128
```

sage.coding.code_bounds.**gv_bound_asymp**(*delta*, *q*)

Computes the asymptotic GV bound for the information rate, R.

EXAMPLES:

```
sage: RDF(gv_bound_asymp(1/4,2))
0.18872187554086...
sage: f = lambda x: gv_bound_asymp(x,2)
sage: plot(f,0,1)
Graphics object consisting of 1 graphics primitive
```

sage.coding.code_bounds.**gv_info_rate**(*n*, *delta*, *q*)

GV lower bound for information rate of a q-ary code of length n minimum distance delta*n

EXAMPLES:

```
sage: RDF(gv_info_rate(100,1/4,3))
0.36704992608261894
```

sage.coding.code_bounds.**hamming_bound_asymp**(*delta*, *q*)

Computes the asymptotic Hamming bound for the information rate.

EXAMPLES:

```
sage: RDF(hamming_bound_asymp(1/4,2))
0.456435556800...
sage: f = lambda x: hamming_bound_asymp(x,2)
sage: plot(f,0,1)
Graphics object consisting of 1 graphics primitive
```

sage.coding.code_bounds.**hamming_upper_bound**(*n*, *q*, *d*)

Returns the Hamming upper bound for number of elements in the largest code of minimum distance d in $\mathbf{F}_q^n$. Wraps GAP's UpperBoundHamming.

The Hamming bound (also known as the sphere packing bound) returns an upper bound on the size of a code of length n, minimum distance d, over a field of size q. The Hamming bound is obtained by dividing the contents of the entire space $\mathbf{F}_q^n$ by the contents of a ball with radius floor((d-1)/2). As all these balls are disjoint, they can never contain more than the whole vector space.

$$M \leq \frac{q^n}{V(n,e)},$$

where M is the maximum number of codewords and $V(n,e)$ is equal to the contents of a ball of radius e. This bound is useful for small values of d. Codes for which equality holds are called perfect.

EXAMPLES:

```
sage: hamming_upper_bound(10,2,3)
93
```

sage.coding.code_bounds.**mrrw1_bound_asymp**(*delta*, *q*)

Computes the first asymptotic McEliese-Rumsey-Rodemich-Welsh bound for the information rate, provided $0 < \delta < 1 - 1/q$.

EXAMPLES:

```
sage: mrrw1_bound_asymp(1/4,2)
0.3545789026652697
```

sage.coding.code_bounds.**plotkin_bound_asymp**(*delta*, *q*)

Computes the asymptotic Plotkin bound for the information rate, provided $0 < \delta < 1 - 1/q$.

EXAMPLES:

```
sage: plotkin_bound_asymp(1/4,2)
1/2
```

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

Returns Plotkin upper bound for number of elements in the largest code of minimum distance d in $\mathbf{F}_q^n$.

The algorithm="gap" option wraps Guava's UpperBoundPlotkin.

EXAMPLES:

```
sage: plotkin_upper_bound(10,2,3)
192
sage: plotkin_upper_bound(10,2,3,algorithm="gap")  # optional - gap_packages (Guava package)
192
```

sage.coding.code_bounds.**singleton_bound_asymp**(*delta*, *q*)

Computes the asymptotic Singleton bound for the information rate.

EXAMPLES:

```
sage: singleton_bound_asymp(1/4,2)
3/4
sage: f = lambda x: singleton_bound_asymp(x,2)
sage: plot(f,0,1)
Graphics object consisting of 1 graphics primitive
```

sage.coding.code_bounds.**singleton_upper_bound**(*n*, *q*, *d*)

Returns the Singleton upper bound for number of elements in the largest code of minimum distance d in $\mathbf{F}_q^n$. Wraps GAP's UpperBoundSingleton.

This bound is based on the shortening of codes. By shortening an (n, M, d) code d-1 times, an $(n-d+1, M, 1)$ code results, with $M \leq q^n - d + 1$. Thus

$$M \leq q^{n-d+1}.$$

Codes that meet this bound are called maximum distance separable (MDS).

EXAMPLES:

```
sage: singleton_upper_bound(10,2,3)
256
```

sage.coding.code_bounds.**volume_hamming**(*n*, *q*, *r*)

Returns number of elements in a Hamming ball of radius r in $\mathbf{F}_q^n$. Agrees with Guava's SphereContent(n,r,GF(q)).

EXAMPLES:

```
sage: volume_hamming(10,2,3)
176
```

CHAPTER

SEVEN

CANONICAL FORMS AND AUTOMORPHISMS FOR LINEAR CODES OVER FINITE FIELDS.

We implemented the algorithm described in [Feu2009] which computes, a unique code (canonical form) in the equivalence class of a given linear code $C \leq \mathbf{F}_q^n$. Furthermore, this algorithm will return the automorphism group of C, too. You will find more details about the algorithm in the documentation of the class `LinearCodeAutGroupCanLabel`.

The equivalence of codes is modeled as a group action by the group $G = \mathbf{F}_q^{*n} \rtimes (Aut(\mathbf{F}_q) \times S_n)$ on the set of subspaces of $\mathbf{F}_q^n$. The group G will be called the semimonomial group of degree n.

The algorithm is started by initializing the class `LinearCodeAutGroupCanLabel`. When the object gets available, all computations are already finished and you can access the relevant data using the member functions:

- `get_canonical_form()`
- `get_transporter()`
- `get_autom_gens()`

People do also use some weaker notions of equivalence, namely **permutational** equivalence and monomial equivalence (**linear** isometries). These can be seen as the subgroups S_n and $\mathbf{F}_q^{*n} \rtimes S_n$ of G. If you are interested in one of these notions, you can just pass the optional parameter `algorithm_type`.

A second optional parameter `P` allows you to restrict the group of permutations S_n to a subgroup which respects the coloring given by `P`.

AUTHORS:

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

REFERENCES:

EXAMPLES:

```
sage: from sage.coding.codecan.autgroup_can_label import LinearCodeAutGroupCanLabel
sage: C = codes.HammingCode(3, GF(3)).dual_code()
sage: P = LinearCodeAutGroupCanLabel(C)
sage: P.get_canonical_form().generator_matrix()
[1 0 0 0 0 1 1 1 1 1 1 1 1]
[0 1 0 1 1 0 0 1 1 2 2 1 2]
[0 0 1 1 2 1 2 1 2 1 2 0 0]
sage: LinearCode(P.get_transporter()*C.generator_matrix()) == P.get_canonical_form()
True
sage: A = P.get_autom_gens()
sage: all( [ LinearCode(a*C.generator_matrix()) == C for a in A])
True
sage: P.get_autom_order() == GL(3, GF(3)).order()
True
```

If the dimension of the dual code is smaller, we will work on this code:

```
sage: C2 = codes.HammingCode(3, GF(3))
sage: P2 = LinearCodeAutGroupCanLabel(C2)
sage: P2.get_canonical_form().parity_check_matrix() == P.get_canonical_form().generator_matrix()
True
```

There is a specialization of this algorithm to pass a coloring on the coordinates. This is just a list of lists, telling the algorithm which columns do share the same coloring:

```
sage: C = codes.HammingCode(3, GF(4, 'a')).dual_code()
sage: P = LinearCodeAutGroupCanLabel(C, P=[ [0], [1], range(2, C.length()) ])
sage: P.get_autom_order()
864
sage: A = [a.get_perm() for a in P.get_autom_gens()]
sage: H = SymmetricGroup(21).subgroup(A)
sage: H.orbits()
[[1], [2], [3, 5, 4], [6, 10, 13, 20, 17, 9, 8, 11, 18, 15, 14, 16, 12, 19, 21, 7]]
```

We can also restrict the group action to linear isometries:

```
sage: P = LinearCodeAutGroupCanLabel(C, algorithm_type="linear")
sage: P.get_autom_order() == GL(3, GF(4, 'a')).order()
True
```

and to the action of the symmetric group only:

```
sage: P = LinearCodeAutGroupCanLabel(C, algorithm_type="permutational")
sage: P.get_autom_order() == C.permutation_automorphism_group().order()
True
```

class sage.coding.codecan.autgroup_can_label.**LinearCodeAutGroupCanLabel**(*C*, *P=None*, *algorithm_type='semilinear'*)

Canonical representatives and automorphism group computation for linear codes over finite fields.

There are several notions of equivalence for linear codes: Let C, D be linear codes of length n and dimension k. C and D are said to be

- permutational equivalent, if there is some permutation $\pi \in S_n$ such that $(c_{\pi(0)}, \ldots, c_{\pi(n-1)}) \in D$ for all $c \in C$.
- linear equivalent, if there is some permutation $\pi \in S_n$ and a vector $\phi \in \mathbf{F}_q^{*n}$ of units of length n such that $(c_{\pi(0)}\phi_0^{-1}, \ldots, c_{\pi(n-1)}\phi_{n-1}^{-1}) \in D$ for all $c \in C$.
- semilinear equivalent, if there is some permutation $\pi \in S_n$, a vector ϕ of units of length n and a field automorphism α such that $(\alpha(c_{\pi(0)})\phi_0^{-1}, \ldots, \alpha(c_{\pi(n-1)})\phi_{n-1}^{-1}) \in D$ for all $c \in C$.

These are group actions. This class provides an algorithm that will compute a unique representative D in the orbit of the given linear code C. Furthermore, the group element g with $g * C = D$ and the automorphism group of C will be computed as well.

There is also the possibility to restrict the permutational part of this action to a Young subgroup of S_n. This could be achieved by passing a partition P (as a list of lists) of the set $\{0, \ldots, n-1\}$. This is an option which is also available in the computation of a canonical form of a graph, see sage.graphs.generic_graph.GenericGraph.canonical_label().

EXAMPLES:

```
sage: from sage.coding.codecan.autgroup_can_label import LinearCodeAutGroupCanLabel
sage: C = codes.HammingCode(3, GF(3)).dual_code()
sage: P = LinearCodeAutGroupCanLabel(C)
sage: P.get_canonical_form().generator_matrix()
[1 0 0 0 0 1 1 1 1 1 1 1 1]
[0 1 0 1 1 0 0 1 1 2 2 1 2]
[0 0 1 1 2 1 2 1 2 1 2 0 0]
sage: LinearCode(P.get_transporter()*C.generator_matrix()) == P.get_canonical_form()
True
sage: a = P.get_autom_gens()[0]
sage: (a*C.generator_matrix()).echelon_form() == C.generator_matrix().echelon_form()
True
sage: P.get_autom_order() == GL(3, GF(3)).order()
True
```

get_PGammaL_gens()

Return the set of generators translated to the group $P\Gamma L(k, q)$.

There is a geometric point of view of code equivalence. A linear code is identified with the multiset of points in the finite projective geometry $PG(k-1, q)$. The equivalence of codes translates to the natural action of $P\Gamma L(k, q)$. Therefore, we may interpret the group as a subgroup of $P\Gamma L(k, q)$ as well.

EXAMPLES:

```
sage: from sage.coding.codecan.autgroup_can_label import LinearCodeAutGroupCanLabel
sage: C = codes.HammingCode(3, GF(4, 'a')).dual_code()
sage: A = LinearCodeAutGroupCanLabel(C).get_PGammaL_gens()
sage: Gamma = C.generator_matrix()
sage: N = [ x.monic() for x in Gamma.columns() ]
sage: all([ (g[0]*n.apply_map(g[1])).monic() in N for n in N for g in A])
True
```

get_PGammaL_order()

Return the size of the automorphism group as a subgroup of $P\Gamma L(k, q)$.

There is a geometric point of view of code equivalence. A linear code is identified with the multiset of points in the finite projective geometry $PG(k-1, q)$. The equivalence of codes translates to the natural action of $P\Gamma L(k, q)$. Therefore, we may interpret the group as a subgroup of $P\Gamma L(k, q)$ as well.

EXAMPLES:

```
sage: from sage.coding.codecan.autgroup_can_label import LinearCodeAutGroupCanLabel
sage: C = codes.HammingCode(3, GF(4, 'a')).dual_code()
sage: LinearCodeAutGroupCanLabel(C).get_PGammaL_order() == GL(3, GF(4, 'a')).order()*2/3
True
```

get_autom_gens()

Return a generating set for the automorphism group of the code.

EXAMPLES:

```
sage: from sage.coding.codecan.autgroup_can_label import LinearCodeAutGroupCanLabel
sage: C = codes.HammingCode(3, GF(2)).dual_code()
sage: A = LinearCodeAutGroupCanLabel(C).get_autom_gens()
sage: Gamma = C.generator_matrix().echelon_form()
sage: all([(g*Gamma).echelon_form() == Gamma for g in A])
True
```

get_autom_order()

Return the size of the automorphism group of the code.

EXAMPLES:

```
sage: from sage.coding.codecan.autgroup_can_label import LinearCodeAutGroupCanLabel
sage: C = codes.HammingCode(3, GF(2)).dual_code()
sage: LinearCodeAutGroupCanLabel(C).get_autom_order()
168
```

get_canonical_form()

Return the canonical orbit representative we computed.

EXAMPLES:

```
sage: from sage.coding.codecan.autgroup_can_label import LinearCodeAutGroupCanLabel
sage: C = codes.HammingCode(3, GF(3)).dual_code()
sage: CF1 = LinearCodeAutGroupCanLabel(C).get_canonical_form()
sage: s = SemimonomialTransformationGroup(GF(3), C.length()).an_element()
sage: C2 = LinearCode(s*C.generator_matrix())
sage: CF2 = LinearCodeAutGroupCanLabel(C2).get_canonical_form()
sage: CF1 == CF2
True
```

get_transporter()

Return the element which maps the code to its canonical form.

EXAMPLES:

```
sage: from sage.coding.codecan.autgroup_can_label import LinearCodeAutGroupCanLabel
sage: C = codes.HammingCode(3, GF(2)).dual_code()
sage: P = LinearCodeAutGroupCanLabel(C)
sage: g = P.get_transporter()
sage: D = P.get_canonical_form()
sage: (g*C.generator_matrix()).echelon_form() == D.generator_matrix().echelon_form()
True
```

CHAPTER

EIGHT

DELSARTE, A.K.A. LINEAR PROGRAMMING (LP), UPPER BOUNDS.

This module provides LP upper bounds for the parameters of codes. Exact LP solver, PPL, is used by defaut, ensuring that no rounding/overflow problems occur.

AUTHORS:

- Dmitrii V. (Dima) Pasechnik (2012-10): initial implementation.

sage.coding.delsarte_bounds.**Krawtchouk**(*n*, *q*, *l*, *i*)

Compute `K^{n,q}_l(i)`, the Krawtchouk polynomial: see Wikipedia article Kravchuk_polynomials. It is given by

$$K_l^{n,q}(i) = \sum_{j=0}^{l} (-1)^j (q-1)^{(l-j)} \binom{i}{j} \binom{n-i}{l-j}$$

EXAMPLES:

```
sage: Krawtchouk(24,2,5,4)
2224
sage: Krawtchouk(12300,4,5,6)
567785569973042442072
```

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 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 nonlinear. Otherwise, `q=q_base^m` and the code is linear over `F_{q_base}`.

- `return_data` – if `True`, return a triple `(W,LP,bound)`, where `W` is a weights vector, and `LP` the Delsarte bound LP; both of them are Sage LP data. `W` need not be a weight distribution of a code, or, if `isinteger==False`, even have integer entries.
- `solver` – the LP/ILP solver to be used. Defaults to `PPL`. It is arbitrary precision, thus there will be no rounding errors. With other solvers (see `MixedIntegerLinearProgram` for the list), you are on your own!
- `isinteger` – if `True`, uses an integer programming solver (ILP), rather that an LP solver. Can be very slow if set to `True`.

EXAMPLES:

The bound on dimension of linear F_2-codes of length 11 and minimal distance 6:

```
sage: delsarte_bound_additive_hamming_space(11, 6, 2)
3
sage: a,p,val=delsarte_bound_additive_hamming_space(11, 6, 2,
sage: [j for i,j in p.get_values(a).iteritems()]
[1, 0, 0, 0, 0, 0, 5, 2, 0, 0, 0, 0]
```

The bound on the dimension of linear F_4-codes of length 11 and minimal distance 3:

```
sage: delsarte_bound_additive_hamming_space(11,3,4)
8
```

The bound on the F_2-dimension of additive F_4-codes of length 11 and minimal distance 3:

```
sage: delsarte_bound_additive_hamming_space(11,3,4,q_base=2)
16
```

Such a d_star is not possible:

```
sage: delsarte_bound_additive_hamming_space(11,3,4,d_star=9)
Solver exception:  'PPL : There is no feasible solution' ()
False
```

sage.coding.delsarte_bounds.**delsarte_bound_hamming_space**(*n*, *d*, *q*, *return_data=False*, *solver='PPL'*)

Find the classical Delsarte bound [1] on codes in Hamming space `H_q^n` of minimal distance `d`

INPUT:

- `n` – the code length
- `d` – the (lower bound on) minimal distance of the code
- `q` – the size of the alphabet
- **`return_data` – if `True`, return a triple `(W,LP,bound)`, where `W` is** a weights vector, and `LP` the Delsarte 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!

EXAMPLES:

The bound on the size of the F_2-codes of length 11 and minimal distance 6:

[1] P. Delsarte, An algebraic approach to the association schemes of coding theory, Philips Res. Rep., Suppl., vol. 10, 1973.

```
sage: delsarte_bound_hamming_space(11, 6, 2)
12
sage: a, p, val = delsarte_bound_hamming_space(11, 6, 2, return_data=True)
sage: [j for i,j in p.get_values(a).iteritems()]
[1, 0, 0, 0, 0, 0, 11, 0, 0, 0, 0, 0]
```

The bound on the size of the F_2-codes of length 24 and minimal distance 8, i.e. parameters of the extened binary Golay code:

```
sage: a,p,x=delsarte_bound_hamming_space(24,8,2,return_data=True)
sage: x
4096
sage: [j for i,j in p.get_values(a).iteritems()]
[1, 0, 0, 0, 0, 0, 0, 0, 759, 0, 0, 0, 2576, 0, 0, 0, 759, 0, 0, 0, 0, 0, 0, 0, 1]
```

The bound on the size of F_4-codes of length 11 and minimal distance 3:

```
sage: delsarte_bound_hamming_space(11,3,4)
327680/3
```

Such an input is invalid:

```
sage: delsarte_bound_hamming_space(11,3,-4)
Solver exception:  'PPL : There is no feasible solution' ()
False
```

REFERENCES:

CHAPTER

NINE

HUFFMAN ENCODING

This module implements functionalities relating to Huffman encoding and decoding.

AUTHOR:

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

9.1 Classes and functions

class sage.coding.source_coding.huffman.**Huffman**(*source*)
Bases: sage.structure.sage_object.SageObject

This class implements the basic functionalities of Huffman codes.

It can build a Huffman code from a given string, or from the information of a dictionary associating to each key (the elements of the alphabet) a weight (most of the time, a probability value or a number of occurrences).

INPUT:

- source – can be either
  - A string from which the Huffman encoding should be created.
  - A dictionary that associates to each symbol of an alphabet a numeric value. If we consider the frequency of each alphabetic symbol, then source is considered as the frequency table of the alphabet with each numeric (non-negative integer) value being the number of occurrences of a symbol. The numeric values can also represent weights of the symbols. In that case, the numeric values are not necessarily integers, but can be real numbers.

In order to construct a Huffman code for an alphabet, we use exactly one of the following methods:

1. Let source be a string of symbols over an alphabet and feed source to the constructor of this class. Based on the input string, a frequency table is constructed that contains the frequency of each unique symbol in source. The alphabet in question is then all the unique symbols in source. A significant implication of this is that any subsequent string that we want to encode must contain only symbols that can be found in source.
2. Let source be the frequency table of an alphabet. We can feed this table to the constructor of this class. The table source can be a table of frequencies or a table of weights.

Examples:

```
sage: from sage.coding.source_coding.huffman import Huffman, frequency_table
sage: h1 = Huffman("There once was a french fry")
sage: for letter, code in h1.encoding_table().iteritems():
...       print "'"+ letter + "' : " + code
'a' : 0111
```

```
' ' : 00
'c' : 1010
'e' : 100
'f' : 1011
'h' : 1100
'o' : 11100
'n' : 1101
's' : 11101
'r' : 010
'T' : 11110
'w' : 11111
'y' : 0110
```

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

```
sage: ft = frequency_table("There once was a french fry"); ft
{' ': 5,
 'T': 1,
 'a': 2,
 'c': 2,
 'e': 4,
 'f': 2,
 'h': 2,
 'n': 2,
 'o': 1,
 'r': 3,
 's': 1,
 'w': 1,
 'y': 1}
sage: h2 = Huffman(ft)
```

Once `h1` has been trained, and hence possesses an encoding table, it is possible to obtain the Huffman encoding of any string (possibly the same) using this code:

```
sage: encoded = h1.encode("There once was a french fry"); encoded
'11110110010001010000111001101101010000111110111111010001110010110101001101101011000010110100110
```

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

```
sage: h1.decode(encoded)
'There once was a french fry'
```

Obviously, if we try to decode a string using a Huffman instance which has been trained on a different sample (and hence has a different encoding table), we are likely to get some random-looking string:

```
sage: h3 = Huffman("There once were two french fries")
sage: h3.decode(encoded)
' wehnefetrhft ne ewrowrirTc'
```

This does not look like our original string.

Instead of using frequency, we can assign weights to each alphabetic symbol:

```
sage: from sage.coding.source_coding.huffman import Huffman
sage: T = {"a":45, "b":13, "c":12, "d":16, "e":9, "f":5}
sage: H = Huffman(T)
sage: L = ["deaf", "bead", "fab", "bee"]
sage: E = []
sage: for e in L:
...       E.append(H.encode(e))
```

```
...       print E[-1]
...
111110101100
10111010111
11000101
10111011101
sage: D = []
sage: for e in E:
...       D.append(H.decode(e))
...       print D[-1]
...
deaf
bead
fab
bee
sage: D == L
True
```

decode(*string*)

Decode the given string using the current encoding table.

INPUT:

- `string` – a string of Huffman encodings.

OUTPUT:

- The Huffman decoding of `string`.

EXAMPLES:

This is how a string is encoded and then decoded:

```
sage: from sage.coding.source_coding.huffman import Huffman
sage: str = "Sage is my most favorite general purpose computer algebra system"
sage: h = Huffman(str)
sage: encoded = h.encode(str); encoded
'00000110100010101011000011101010011100101010011011011100111101110010110100001011011111000000
sage: h.decode(encoded)
'Sage is my most favorite general purpose computer algebra system'
```

TESTS:

Of course, the string one tries to decode has to be a binary one. If not, an exception is raised:

```
sage: h.decode('I clearly am not a binary string')
Traceback (most recent call last):
...
ValueError: Input must be a binary string.
```

encode(*string*)

Encode the given string based on the current encoding table.

INPUT:

- `string` – a string of symbols over an alphabet.

OUTPUT:

- A Huffman encoding of `string`.

EXAMPLES:

This is how a string is encoded and then decoded:

```
sage: from sage.coding.source_coding.huffman import Huffman
sage: str = "Sage is my most favorite general purpose computer algebra system"
sage: h = Huffman(str)
sage: encoded = h.encode(str); encoded
'000001101000101010110000111010100111001010100110110111001111011100101101000010110111110000
sage: h.decode(encoded)
'Sage is my most favorite general purpose computer algebra system'
```

encoding_table()

Returns the current encoding table.

INPUT:

- None.

OUTPUT:

- A dictionary associating an alphabetic symbol to a Huffman encoding.

EXAMPLES:

```
sage: from sage.coding.source_coding.huffman import Huffman
sage: str = "Sage is my most favorite general purpose computer algebra system"
sage: h = Huffman(str)
sage: T = sorted(h.encoding_table().items())
sage: for symbol, code in T:
...       print symbol, code
...
  101
S 00000
a 1101
b 110001
c 110000
e 010
f 110010
g 0001
i 10000
l 10011
m 0011
n 110011
o 0110
p 0010
r 1111
s 1110
t 0111
u 10001
v 00001
y 10010
```

tree()

Returns the Huffman tree corresponding to the current encoding.

INPUT:

- None.

OUTPUT:

- The binary tree representing a Huffman code.

EXAMPLES:

```
sage: from sage.coding.source_coding.huffman import Huffman
sage: str = "Sage is my most favorite general purpose computer algebra system"
sage: h = Huffman(str)
sage: T = h.tree(); T
Digraph on 39 vertices
sage: T.show(figsize=[20,20])
```

sage.coding.source_coding.huffman.**frequency_table**(*string*)

Return the frequency table corresponding to the given string.

INPUT:

- `string` – a string of symbols over some alphabet.

OUTPUT:

- A table of frequency of each unique symbol in `string`. If `string` is an empty string, return an empty table.

EXAMPLES:

The frequency table of a non-empty string:

```
sage: from sage.coding.source_coding.huffman import frequency_table
sage: str = "Stop counting my characters!"
sage: T = sorted(frequency_table(str).items())
sage: for symbol, code in T:
...       print symbol, code
...
  3
! 1
S 1
a 2
c 3
e 1
g 1
h 1
i 1
m 1
n 2
o 2
p 1
r 2
s 1
t 3
u 1
y 1
```

The frequency of an empty string:

```
sage: frequency_table("")
{}
```

CHAPTER

TEN

INDICES AND TABLES

- Index
- Module Index
- Search Page

[D] I. Duursma, “Extremal weight enumerators and ultraspherical polynomials”

[HP] W. C. Huffman, V. Pless, Fundamentals of Error-Correcting Codes, Cambridge Univ. Press, 2003.

[J] D. Joyner, Toric codes over finite fields, Applicable Algebra in Engineering, Communication and Computing, 15, (2004), p. 63-79.

[BM] Bazzi and Mitter, {it Some constructions of codes from group actions}, (preprint March 2003, available on Mitter’s MIT website).

[Jresidue] D. Joyner, {it On quadratic residue codes and hyperelliptic curves}, (preprint 2006)

[Feu2009] T. Feulner. The Automorphism Groups of Linear Codes and Canonical Representatives of Their Semilinear Isometry Classes. Advances in Mathematics of Communications 3 (4), pp. 363-383, Nov 2009

C

sage.coding.code_bounds, 55
sage.coding.code_constructions, 33
sage.coding.codecan.autgroup_can_label, 63
sage.coding.codes_catalog, 1
sage.coding.delsarte_bounds, 67
sage.coding.guava, 49
sage.coding.linear_code, 3
sage.coding.sd_codes, 51
sage.coding.source_coding.huffman, 71

A

ambient_space() (sage.coding.linear_code.LinearCode method), 6
assmus_mattson_designs() (sage.coding.linear_code.LinearCode method), 6
automorphism_group_gens() (sage.coding.linear_code.LinearCode method), 7

B

basis() (sage.coding.linear_code.LinearCode method), 8
BCHCode() (in module sage.coding.code_constructions), 34
best_known_linear_code() (in module sage.coding.linear_code), 27
best_known_linear_code_www() (in module sage.coding.linear_code), 28
BinaryGolayCode() (in module sage.coding.code_constructions), 35
BinaryReedMullerCode() (in module sage.coding.guava), 49
binomial_moment() (sage.coding.linear_code.LinearCode method), 8
bounds_minimum_distance() (in module sage.coding.linear_code), 28

C

canonical_representative() (sage.coding.linear_code.LinearCode method), 9
cardinality() (sage.coding.linear_code.LinearCode method), 9
characteristic() (sage.coding.linear_code.LinearCode method), 10
characteristic_polynomial() (sage.coding.linear_code.LinearCode method), 10
check_mat() (sage.coding.linear_code.LinearCode method), 10
chinen_polynomial() (sage.coding.linear_code.LinearCode method), 10
code2leon() (in module sage.coding.linear_code), 29
codesize_upper_bound() (in module sage.coding.code_bounds), 57
covering_radius() (sage.coding.linear_code.LinearCode method), 10
CyclicCode() (in module sage.coding.code_constructions), 35
CyclicCodeFromCheckPolynomial() (in module sage.coding.code_constructions), 36
CyclicCodeFromGeneratingPolynomial() (in module sage.coding.code_constructions), 36
cyclotomic_cosets() (in module sage.coding.code_constructions), 44

D

decode() (sage.coding.linear_code.LinearCode method), 11
decode() (sage.coding.source_coding.huffman.Huffman method), 73
delsarte_bound_additive_hamming_space() (in module sage.coding.delsarte_bounds), 67
delsarte_bound_hamming_space() (in module sage.coding.delsarte_bounds), 68
dimension() (sage.coding.linear_code.LinearCode method), 12
dimension_upper_bound() (in module sage.coding.code_bounds), 57

direct_sum() (sage.coding.linear_code.LinearCode method), 12
divisor() (sage.coding.linear_code.LinearCode method), 12
DuadicCodeEvenPair() (in module sage.coding.code_constructions), 37
DuadicCodeOddPair() (in module sage.coding.code_constructions), 37
dual_code() (sage.coding.linear_code.LinearCode method), 12

E

elias_bound_asymp() (in module sage.coding.code_bounds), 57
elias_upper_bound() (in module sage.coding.code_bounds), 57
encode() (sage.coding.source_coding.huffman.Huffman method), 73
encoding_table() (sage.coding.source_coding.huffman.Huffman method), 74
entropy() (in module sage.coding.code_bounds), 57
entropy_inverse() (in module sage.coding.code_bounds), 58
extended_code() (sage.coding.linear_code.LinearCode method), 12
ExtendedBinaryGolayCode() (in module sage.coding.code_constructions), 38
ExtendedQuadraticResidueCode() (in module sage.coding.code_constructions), 38
ExtendedTernaryGolayCode() (in module sage.coding.code_constructions), 39

F

frequency_table() (in module sage.coding.source_coding.huffman), 75

G

galois_closure() (sage.coding.linear_code.LinearCode method), 13
gen_mat() (sage.coding.linear_code.LinearCode method), 13
gen_mat_systematic() (sage.coding.linear_code.LinearCode method), 13
generator_matrix() (sage.coding.linear_code.LinearCode method), 13
generator_matrix_systematic() (sage.coding.linear_code.LinearCode method), 13
gens() (sage.coding.linear_code.LinearCode method), 14
genus() (sage.coding.linear_code.LinearCode method), 14
get_autom_gens() (sage.coding.codecan.autgroup_can_label.LinearCodeAutGroupCanLabel method), 65
get_autom_order() (sage.coding.codecan.autgroup_can_label.LinearCodeAutGroupCanLabel method), 65
get_canonical_form() (sage.coding.codecan.autgroup_can_label.LinearCodeAutGroupCanLabel method), 66
get_PGammaL_gens() (sage.coding.codecan.autgroup_can_label.LinearCodeAutGroupCanLabel method), 65
get_PGammaL_order() (sage.coding.codecan.autgroup_can_label.LinearCodeAutGroupCanLabel method), 65
get_transporter() (sage.coding.codecan.autgroup_can_label.LinearCodeAutGroupCanLabel method), 66
gilbert_lower_bound() (in module sage.coding.code_bounds), 58
griesmer_upper_bound() (in module sage.coding.code_bounds), 58
gv_bound_asymp() (in module sage.coding.code_bounds), 59
gv_info_rate() (in module sage.coding.code_bounds), 59

H

hamming_bound_asymp() (in module sage.coding.code_bounds), 59
hamming_upper_bound() (in module sage.coding.code_bounds), 59
HammingCode() (in module sage.coding.code_constructions), 39
Huffman (class in sage.coding.source_coding.huffman), 71

I

I2() (in module sage.coding.sd_codes), 52
information_set() (sage.coding.linear_code.LinearCode method), 14
is_a_splitting() (in module sage.coding.code_constructions), 45

is_galois_closed() (sage.coding.linear_code.LinearCode method), 14
is_permutation_automorphism() (sage.coding.linear_code.LinearCode method), 14
is_permutation_equivalent() (sage.coding.linear_code.LinearCode method), 15
is_self_dual() (sage.coding.linear_code.LinearCode method), 15
is_self_orthogonal() (sage.coding.linear_code.LinearCode method), 15
is_subcode() (sage.coding.linear_code.LinearCode method), 16

K

Krawtchouk() (in module sage.coding.delsarte_bounds), 67

L

length() (sage.coding.linear_code.LinearCode method), 16
lift2smallest_field() (in module sage.coding.code_constructions), 46
lift2smallest_field2() (in module sage.coding.code_constructions), 47
LinearCode (class in sage.coding.linear_code), 5
LinearCodeAutGroupCanLabel (class in sage.coding.codecan.autgroup_can_label), 64
LinearCodeFromCheckMatrix() (in module sage.coding.code_constructions), 40
LinearCodeFromVectorSpace() (in module sage.coding.linear_code), 27
list() (sage.coding.linear_code.LinearCode method), 16

M

matA() (in module sage.coding.sd_codes), 52
matId() (in module sage.coding.sd_codes), 52
min_wt_vec_gap() (in module sage.coding.linear_code), 29
minimum_distance() (sage.coding.linear_code.LinearCode method), 16
module_composition_factors() (sage.coding.linear_code.LinearCode method), 17
mrrw1_bound_asymp() (in module sage.coding.code_bounds), 59
MS() (in module sage.coding.sd_codes), 52
MS2() (in module sage.coding.sd_codes), 52

P

parity_check_matrix() (sage.coding.linear_code.LinearCode method), 18
permutation_action() (in module sage.coding.code_constructions), 47
permutation_automorphism_group() (sage.coding.linear_code.LinearCode method), 18
permuted_code() (sage.coding.linear_code.LinearCode method), 19
plotkin_bound_asymp() (in module sage.coding.code_bounds), 60
plotkin_upper_bound() (in module sage.coding.code_bounds), 60
punctured() (sage.coding.linear_code.LinearCode method), 20

Q

QuadraticResidueCode() (in module sage.coding.code_constructions), 40
QuadraticResidueCodeEvenPair() (in module sage.coding.code_constructions), 41
QuadraticResidueCodeOddPair() (in module sage.coding.code_constructions), 41
QuasiQuadraticResidueCode() (in module sage.coding.guava), 50

R

random_element() (sage.coding.linear_code.LinearCode method), 20
RandomLinearCode() (in module sage.coding.code_constructions), 42
RandomLinearCodeGuava() (in module sage.coding.guava), 50
redundancy_matrix() (sage.coding.linear_code.LinearCode method), 20

ReedSolomonCode() (in module sage.coding.code_constructions), 42

S

sage.coding.code_bounds (module), 55
sage.coding.code_constructions (module), 33
sage.coding.codecan.autgroup_can_label (module), 63
sage.coding.codes_catalog (module), 1
sage.coding.delsarte_bounds (module), 67
sage.coding.guava (module), 49
sage.coding.linear_code (module), 3
sage.coding.sd_codes (module), 51
sage.coding.source_coding.huffman (module), 71
sd_duursma_data() (sage.coding.linear_code.LinearCode method), 21
sd_duursma_q() (sage.coding.linear_code.LinearCode method), 21
sd_zeta_polynomial() (sage.coding.linear_code.LinearCode method), 22
self_dual_codes_binary() (in module sage.coding.sd_codes), 53
self_orthogonal_binary_codes() (in module sage.coding.linear_code), 30
shortened() (sage.coding.linear_code.LinearCode method), 22
singleton_bound_asymp() (in module sage.coding.code_bounds), 60
singleton_upper_bound() (in module sage.coding.code_bounds), 60
spectrum() (sage.coding.linear_code.LinearCode method), 23
standard_form() (sage.coding.linear_code.LinearCode method), 24
support() (sage.coding.linear_code.LinearCode method), 24

T

TernaryGolayCode() (in module sage.coding.code_constructions), 43
ToricCode() (in module sage.coding.code_constructions), 43
tree() (sage.coding.source_coding.huffman.Huffman method), 74
TrivialCode() (in module sage.coding.code_constructions), 44

V

volume_hamming() (in module sage.coding.code_bounds), 60

W

walsh_matrix() (in module sage.coding.code_constructions), 48
WalshCode() (in module sage.coding.code_constructions), 44
weight_distribution() (sage.coding.linear_code.LinearCode method), 24
weight_enumerator() (sage.coding.linear_code.LinearCode method), 25
wtdist_gap() (in module sage.coding.linear_code), 31

Z

zero() (sage.coding.linear_code.LinearCode method), 26
zeta_function() (sage.coding.linear_code.LinearCode method), 26
zeta_polynomial() (sage.coding.linear_code.LinearCode method), 26