Coding Theory

Release 10.2

The Sage Development Team

Dec 06, 2023

CONTENTS

Coding theory is the mathematical theory for algebraic and combinatorial codes used for forward error correction in communications theory. Sage provides an extensive library of objects and algorithms in coding theory.

Basic objects in coding theory are codes, channels, encoders, and decoders. The following modules provide the base classes defining them.

CHAPTER

ONE

CODES

Class supporting methods available for any type of code (linear, non-linear) and over any metric (Hamming, rank).

There are further abstract classes representing certain types of codes. For linear codes, *AbstractLinearCodeNoMetric* contains all the methods that any linear code can use regardless of its metric. Inheriting from this class are base classes for linear codes over specific metrics. For example, *AbstractLinearCode* is a base class for all linear codes over the Hamming metric.

Take the class *HammingCode*. This class inherits from *AbstractLinearCode*, since it is a linear code over the Hamming metric. *AbstractLinearCode* then inherits from *AbstractLinearCodeNoMetric*, since it is a linear code. Finally, this class inherits from *AbstractCode*, since it is a code.

The following diagram shows the inheritance relationship in the coding module:

```
AbstractCode
+ AbstractLinearCodeNoMetric
| + AbstractLinearCode
| | + ParityCheckCode
| | + HammingCode
| | + CyclicCode
| | + BCHCode
| | + GolayCode
| | + ReedMullerCode
| | + GeneralizedReedSolomonCode
| | + GoppaCode
| + AbstractLinearRankMetricCode
```

Any class inheriting from AbstractCode can use the encode/decode framework.

The encoder/decoder framework within the coding module offers the creation and use of encoders/decoders independently of codes. An encoder encodes a message into a codeword. A decoder decodes a word into a codeword or a message, possibly with error-correction.

Instead of creating specific encoders/decoders for every code family, some encoders/decoders can be used by multiple code families. The encoder/decoder framework enables just that. For example, *LinearCodeGeneratorMatrixEncoder* can be used by any code that has a generator matrix. Similarly, *LinearCodeNearestNeighborDecoder* can be used for any linear code with Hamming metric.

When creating a new code family, investigate the encoder/decoder catalogs, `codes.encoders` and `codes.decoders`, to see if there are suitable encoders/decoders for your code family already implemented. If this is the case, follow the instructions in *AbstractCode* to set these up.

A new encoder must have the following methods:

- `encode` – method encoding a message into a codeword
- `unencode` – method decoding a codeword into a message

- `message_space` – ambient space of messages that can be encoded
- `code` – code of the encoder

For more information about the Encoder class, see *Encoder*

A new decoder must have the following methods:

- `decode_to_code` or `decode_to_message` – method decoding a word from the input space into either a codeword or a message
- `input_space` – ambient space of words that can be decoded
- `code` – code of the decoder

For more information about the Decoder class, see *Decoder*

class sage.coding.abstract_code.**AbstractCode**(*length*, *default_encoder_name=None*, *default_decoder_name=None*, *metric='Hamming'*)

Bases: `Parent`

Abstract class for codes.

This class contains all the methods that can be used on any code and on any code family. As opposed to *sage.coding.linear_code.AbstractLinearCode*, this class makes no assumptions about linearity, metric, finiteness or the number of alphabets.

The abstract notion of "code" that is implicitly used for this class is any enumerable subset of a cartesian product $A_1 \times A_2 \times \ldots \times A_n$ for some sets A_i. Note that this class makes no attempt to directly represent the code in this fashion, allowing subclasses to make the appropriate choices. The notion of metric is also not mathematically enforced in any way, and is simply stored as a string value.

Every code-related class should inherit from this abstract class.

To implement a code, you need to:

- inherit from *AbstractCode*
- call *AbstractCode* `__init__` method in the subclass constructor. Example: `super().__init__(length, "EncoderName", "DecoderName", "metric")`. "EncoderName" and "DecoderName" are set to `None` by default, a generic code class such as AbstractCode does not necessarily have to have general encoders/decoders. However, if you want to use the encoding/decoding methods, you have to add these.
- since this class does not specify any category, it is highly recommended to set up the category framework in the subclass. To do this, use the `Parent.__init__(self, base, facade, category)` function in the subclass constructor. A good example is in *sage.coding.linear_code.AbstractLinearCode*.
- it is also recommended to override the `ambient_space` method, which is required by `__call__`
- to use the encoder/decoder framework, one has to set up the category and related functions `__iter__` and `__contains__`. A good example is in *sage.coding.linear_code.AbstractLinearCode*.
- add the following two lines on the class level:

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

- fill the dictionary of its encoders in `sage.coding.__init__.py` file. Example: I want to link the encoder `MyEncoderClass` to `MyNewCodeClass` under the name `MyEncoderName`. All I need to do is to write this line in the `__init__.py` file: `MyNewCodeClass._registered_encoders["NameOfMyEncoder"] = MyEncoderClass` and all instances of `MyNewCodeClass` will be able to use instances of `MyEncoderClass`.

- fill the dictionary of its decoders in `sage.coding.__init__` file. Example: I want to link the encoder `MyDecoderClass` to `MyNewCodeClass` under the name `MyDecoderName`. All I need to do is to write this line in the `__init__.py` file: `MyNewCodeClass._registered_decoders["NameOfMyDecoder"] = MyDecoderClass` and all instances of `MyNewCodeClass` will be able to use instances of `MyDecoderClass`.

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

add_decoder(*name*, *decoder*)

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

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

INPUT:

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

EXAMPLES:

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

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

We now create a new code:

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

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

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

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

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

add_encoder(*name*, *encoder*)

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

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

INPUT:

- `name` – the string name for the encoder

- encoder – the class name of the encoder

EXAMPLES:

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

```
sage: class MyEncoder(sage.coding.encoder.Encoder):
....:     def __init__(self, code):
....:         super().__init__(code)
....:     def _repr_(self):
....:         return "MyEncoder encoder with associated code %s" % self.code()
```

We now create a new code:

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

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

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

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

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

ambient_space()

Return an error stating `ambient_space` of `self` is not implemented.

This method is required by `__call__()`.

EXAMPLES:

```
sage: from sage.coding.abstract_code import AbstractCode
sage: class MyCode(AbstractCode):
....:     def __init__(self, length):
....:         super().__init__(length)
sage: C = MyCode(3)
sage: C.ambient_space()
Traceback (most recent call last):
...
NotImplementedError: No ambient space implemented for this code.
```

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

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

INPUT:

- `word` – an element in the ambient space as `self`
- `decoder_name` – (default: `None`) Name of the decoder which will be used to decode `word`. The default decoder of `self` will be used if default value is kept.
- `args`, `kwargs` – all additional arguments are forwarded to *decoder()*

OUTPUT:

- A vector of `self`.

EXAMPLES:

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

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

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

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

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

INPUT:

- `word` – an element in the ambient space as `self`
- `decoder_name` – (default: `None`) Name of the decoder which will be used to decode `word`. The default decoder of `self` will be used if default value is kept.
- `args`, `kwargs` – all additional arguments are forwarded to `decoder()`

OUTPUT:

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

EXAMPLES:

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

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

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

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

Return a decoder of `self`.

INPUT:

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

OUTPUT:

- a decoder object

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

EXAMPLES:

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

If there is no decoder for the code, we return an error:

```
sage: from sage.coding.abstract_code import AbstractCode
sage: class MyCodeFamily(AbstractCode):
....:   def __init__(self, length, field):
....:       sage.coding.abstract_code.AbstractCode.__init__(self, length)
....:       Parent.__init__(self, base=field, facade=False, category=Sets())
....:       self._field = field
....:   def field(self):
....:       return self._field
....:   def _repr_(self):
....:       return "%d dummy code over GF(%s)" % (self.length(), self.field().
↪cardinality())
sage: D = MyCodeFamily(5, GF(2))
sage: D.decoder()
Traceback (most recent call last):
...
NotImplementedError: No decoder implemented for this code.
```

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

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

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

```
sage: C.decoder('InformationSet')
Traceback (most recent call last):
...
ValueError: Constructing the InformationSet decoder failed,
possibly due to missing or incorrect parameters.
The constructor requires the arguments ['number_errors'].
It takes the optional arguments ['algorithm'].
```

(continues on next page)

(continued from previous page)

```
It accepts unspecified arguments as well. See the documentation of
sage.coding.information_set_decoder.LinearCodeInformationSetDecoder
for more details.
```

decoders_available(*classes=False*)

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

INPUT:

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

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

EXAMPLES:

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

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

encode(*word*, *encoder_name=None*, **args*, ***kwargs*)

Transforms an element of a message space into a codeword.

INPUT:

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

One can use the following shortcut to encode a word

```
C(word)
```

OUTPUT:

- a vector of `self`.

EXAMPLES:

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

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

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

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

Returns an encoder of `self`.

The returned encoder provided by this method is cached.

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

INPUT:

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

OUTPUT:

- an Encoder object.

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

EXAMPLES:

```
sage: G = Matrix(GF(2), [[1,1,1,0,0,0,0], [1,0,0,1,1,0,0],
....:                     [0,1,0,1,0,1,0], [1,1,0,1,0,0,1]])
sage: C = LinearCode(G)
sage: C.encoder()
Generator matrix-based encoder for [7, 4] linear code over GF(2)
```

If there is no encoder for the code, we return an error:

```
sage: from sage.coding.abstract_code import AbstractCode
sage: class MyCodeFamily(AbstractCode):
....:   def __init__(self, length, field):
....:       sage.coding.abstract_code.AbstractCode.__init__(self, length)
....:       Parent.__init__(self, base=field, facade=False, category=Sets())
....:       self._field = field
....:   def field(self):
....:       return self._field
....:   def _repr_(self):
....:       return "%d dummy code over GF(%s)" % (self.length(),
....:                                       self.field().cardinality())
sage: D = MyCodeFamily(5, GF(2))
sage: D.encoder()
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
NotImplementedError: No encoder implemented for this code.
```

We check that the returned encoder is cached:

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

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

```
sage: sorted(C.encoders_available())
['GeneratorMatrix', 'Systematic']
sage: C.encoder('NonExistingEncoder')
Traceback (most recent call last):
...
ValueError: There is no Encoder named 'NonExistingEncoder'.
The known Encoders are: ['GeneratorMatrix', 'Systematic']
```

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

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

encoders_available(*classes=False*)

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

INPUT:

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

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

EXAMPLES:

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

length()

Returns the length of this code.

EXAMPLES:

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

list()

Return a list of all elements of this code.

EXAMPLES:

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

metric()

Return the metric of `self`.

EXAMPLES:

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

random_element(**args, **kwds*)

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

OUTPUT:

- Random element of the vector space of this code

EXAMPLES:

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

Passes extra positional or keyword arguments through:

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

unencode(*c, encoder_name=None, nocheck=False, **kwargs*)

Returns the message corresponding to `c`.

This is the inverse of *encode()*.

INPUT:

- `c` – a codeword of `self`.
- `encoder_name` – (default: `None`) name of the decoder which will be used to decode `word`. The default decoder of `self` will be used if default value is kept.

- nocheck – (default: False) checks if c is in self. You might set this to True to disable the check for saving computation. Note that if c is not in self and nocheck = True, then the output of *unencode()* is not defined (except that it will be in the message space of self).
- kwargs – all additional arguments are forwarded to the construction of the encoder that is used.

OUTPUT:

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

EXAMPLES:

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

CHAPTER

TWO

CHANNELS

Given an input space and an output space, a channel takes element from the input space (the message) and transforms it into an element of the output space (the transmitted message).

In Sage, Channels simulate error-prone transmission over communication channels, and we borrow the nomenclature from communication theory, such as "transmission" and "positions" as the elements of transmitted vectors. Transmission can be achieved with two methods:

- *Channel.transmit()*. Considering a channel `Chan` and a message `msg`, transmitting `msg` with `Chan` can be done this way:

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

  It can also be written in a more convenient way:

  ```
  Chan(msg)
  ```

- `transmit_unsafe()`. This does the exact same thing as `transmit()` except that it does not check if `msg` belongs to the input space of `Chan`:

  ```
  Chan.transmit_unsafe(msg)
  ```

This is useful in e.g. an inner-loop of a long simulation as a lighter-weight alternative to *Channel.transmit()*.

This file contains the following elements:

- *Channel*, the abstract class for Channels
- *StaticErrorRateChannel*, which creates a specific number of errors in each transmitted message
- *ErrorErasureChannel*, which creates a specific number of errors and a specific number of erasures in each transmitted message

class sage.coding.channel.**Channel**(*input_space*, *output_space*)

Bases: `SageObject`

Abstract top-class for Channel objects.

All channel objects must inherit from this class. To implement a channel subclass, one should do the following:

- inherit from this class,
- call the super constructor,
- override *transmit_unsafe()*.

While not being mandatory, it might be useful to reimplement representation methods (`_repr_` and `_latex_`).

This abstract class provides the following parameters:

- `input_space` – the space of the words to transmit
- `output_space` – the space of the transmitted words

input_space()

Return the input space of `self`.

EXAMPLES:

```
sage: n_err = 2
sage: Chan = channels.StaticErrorRateChannel(GF(59)^6, n_err)
sage: Chan.input_space()
Vector space of dimension 6 over Finite Field of size 59
```

output_space()

Return the output space of `self`.

EXAMPLES:

```
sage: n_err = 2
sage: Chan = channels.StaticErrorRateChannel(GF(59)^6, n_err)
sage: Chan.output_space()
Vector space of dimension 6 over Finite Field of size 59
```

transmit(*message*)

Return `message`, modified accordingly with the algorithm of the channel it was transmitted through.

Checks if `message` belongs to the input space, and returns an exception if not. Note that `message` itself is never modified by the channel.

INPUT:

- `message` – a vector

OUTPUT:

- a vector of the output space of `self`

EXAMPLES:

```
sage: F = GF(59)^6
sage: n_err = 2
sage: Chan = channels.StaticErrorRateChannel(F, n_err)
sage: msg = F((4, 8, 15, 16, 23, 42))
sage: set_random_seed(10)
sage: Chan.transmit(msg)
(4, 8, 4, 16, 23, 53)
```

We can check that the input `msg` is not modified:

```
sage: msg
(4, 8, 15, 16, 23, 42)
```

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

```
sage: n_err = 2
sage: Chan = channels.StaticErrorRateChannel(GF(59)^6, n_err)
sage: msg = (4, 8, 15, 16, 23, 42)
```

(continues on next page)

(continued from previous page)

```
sage: Chan.transmit(msg)
Traceback (most recent call last):
...
TypeError: Message must be an element of the input space for the given channel
```

Note: One can also call directly `Chan(message)`, which does the same as `Chan.transmit(message)`

transmit_unsafe(*message*)

Return `message`, modified accordingly with the algorithm of the channel it was transmitted through.

This method does not check if `message` belongs to the input space of `self`.

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

EXAMPLES:

```
sage: n_err = 2
sage: Chan = channels.StaticErrorRateChannel(GF(59)^6, n_err)
sage: v = Chan.input_space().random_element()
sage: Chan.transmit_unsafe(v)  # random
(1, 33, 46, 18, 20, 49)
```

class sage.coding.channel.**ErrorErasureChannel**(*space*, *number_errors*, *number_erasures*)

Bases: *Channel*

Channel which adds errors and erases several positions in any message it transmits.

The output space of this channel is a Cartesian product between its input space and a VectorSpace of the same dimension over $\mathbf{F}_2$.

INPUT:

- `space` – the input and output space
- `number_errors` – the number of errors created in each transmitted message. It can be either an integer of a tuple. If an tuple is passed as an argument, the number of errors will be a random integer between the two bounds of this tuple.
- `number_erasures` – the number of erasures created in each transmitted message. It can be either an integer of a tuple. If an tuple is passed as an argument, the number of erasures will be a random integer between the two bounds of this tuple.

EXAMPLES:

We construct a ErrorErasureChannel which adds 2 errors and 2 erasures to any transmitted message:

```
sage: n_err, n_era = 2, 2
sage: Chan = channels.ErrorErasureChannel(GF(59)^40, n_err, n_era)
sage: Chan
Error-and-erasure channel creating 2 errors and 2 erasures
of input space Vector space of dimension 40 over Finite Field of size 59
and output space The Cartesian product of (Vector space of dimension 40
over Finite Field of size 59, Vector space of dimension 40 over Finite Field of
size 2)
```

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

```
sage: n_err, n_era = (1, 10), (1, 10)
sage: Chan = channels.ErrorErasureChannel(GF(59)^40, n_err, n_era)
sage: Chan
Error-and-erasure channel creating between 1 and 10 errors and
between 1 and 10 erasures of input space Vector space of dimension 40
over Finite Field of size 59 and output space The Cartesian product of
(Vector space of dimension 40 over Finite Field of size 59,
Vector space of dimension 40 over Finite Field of size 2)
```

number_erasures()

Return the number of erasures created by `self`.

EXAMPLES:

```
sage: n_err, n_era = 0, 3
sage: Chan = channels.ErrorErasureChannel(GF(59)^6, n_err, n_era)
sage: Chan.number_erasures()
(3, 3)
```

number_errors()

Return the number of errors created by `self`.

EXAMPLES:

```
sage: n_err, n_era = 3, 0
sage: Chan = channels.ErrorErasureChannel(GF(59)^6, n_err, n_era)
sage: Chan.number_errors()
(3, 3)
```

transmit_unsafe(*message*)

Return `message` with as many errors as `self._number_errors` in it, and as many erasures as `self._number_erasures` in it.

If `self._number_errors` was passed as an tuple for the number of errors, it will pick a random integer between the bounds of the tuple and use it as the number of errors. It does the same with `self._number_erasures`.

All erased positions are set to 0 in the transmitted message. It is guaranteed that the erasures and the errors will never overlap: the received message will always contains exactly as many errors and erasures as expected.

This method does not check if `message` belongs to the input space of `self`.

INPUT:

- `message` – a vector

OUTPUT:

- a couple of vectors, namely:
  - the transmitted message, which is `message` with erroneous and erased positions
  - the erasure vector, which contains 1 at the erased positions of the transmitted message and `0` elsewhere.

EXAMPLES:

```
sage: F = GF(59)^11
sage: n_err, n_era = 2, 2
sage: Chan = channels.ErrorErasureChannel(F, n_err, n_era)
sage: msg = F((3, 14, 15, 9, 26, 53, 58, 9, 7, 9, 3))
sage: set_random_seed(10)
sage: Chan.transmit_unsafe(msg)
((31, 0, 15, 9, 38, 53, 58, 9, 0, 9, 3), (0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0))
```

class `sage.coding.channel.`**`QarySymmetricChannel`**(*space*, *epsilon*)

Bases: *`Channel`*

The q-ary symmetric, memoryless communication channel.

Given an alphabet Σ with $|\Sigma| = q$ and an error probability ϵ, a q-ary symmetric channel sends an element of Σ into the same element with probability $1 - \epsilon$, and any one of the other $q - 1$ elements with probability $\frac{\epsilon}{q-1}$. This implementation operates over vectors in Σ^n, and "transmits" each element of the vector independently in the above manner.

Though Σ is usually taken to be a finite field, this implementation allows any structure for which Sage can represent Σ^n and for which Σ has a `random_element()` method. However, beware that if Σ is infinite, errors will not be uniformly distributed (since `random_element()` does not draw uniformly at random).

The input space and the output space of this channel are the same: Σ^n.

INPUT:

- `space` – the input and output space of the channel. It has to be $\mathbf{F}_q^n$ for some finite field $\mathbf{F}_q$.
- `epsilon` – the transmission error probability of the individual elements.

EXAMPLES:

We construct a *`QarySymmetricChannel`* which corrupts 30% of all transmitted symbols:

```
sage: epsilon = 0.3
sage: Chan = channels.QarySymmetricChannel(GF(59)^50, epsilon)
sage: Chan
q-ary symmetric channel with error probability 0.300000000000000,
 of input and output space
  Vector space of dimension 50 over Finite Field of size 59
```

`error_probability`()

Return the error probability of a single symbol transmission of `self`.

EXAMPLES:

```
sage: epsilon = 0.3
sage: Chan = channels.QarySymmetricChannel(GF(59)^50, epsilon)
sage: Chan.error_probability()
0.300000000000000
```

`probability_of_at_most_t_errors`(*t*)

Return the probability `self` has to return at most `t` errors.

INPUT:

- `t` – an integer

EXAMPLES:

```
sage: epsilon = 0.3
sage: Chan = channels.QarySymmetricChannel(GF(59)^50, epsilon)
sage: Chan.probability_of_at_most_t_errors(20)
0.952236164579467
```

probability_of_exactly_t_errors(*t*)

Return the probability `self` has to return exactly `t` errors.

INPUT:

- `t` – an integer

EXAMPLES:

```
sage: epsilon = 0.3
sage: Chan = channels.QarySymmetricChannel(GF(59)^50, epsilon)
sage: Chan.probability_of_exactly_t_errors(15)
0.122346861835401
```

transmit_unsafe(*message*)

Return `message` where each of the symbols has been changed to another from the alphabet with probability *error_probability()*.

This method does not check if `message` belongs to the input space of `self`.

INPUT:

- `message` – a vector

EXAMPLES:

```
sage: F = GF(59)^11
sage: epsilon = 0.3
sage: Chan = channels.QarySymmetricChannel(F, epsilon)
sage: msg = F((3, 14, 15, 9, 26, 53, 58, 9, 7, 9, 3))
sage: set_random_seed(10)
sage: Chan.transmit_unsafe(msg)
(3, 14, 15, 53, 12, 53, 58, 9, 55, 9, 3)
```

class sage.coding.channel.**StaticErrorRateChannel**(*space*, *number_errors*)

Bases: *Channel*

Channel which adds a static number of errors to each message it transmits.

The input space and the output space of this channel are the same.

INPUT:

- `space` – the space of both input and output
- `number_errors` – the number of errors added to each transmitted message It can be either an integer of a tuple. If a tuple is passed as argument, the number of errors will be a random integer between the two bounds of the tuple.

EXAMPLES:

We construct a *StaticErrorRateChannel* which adds 2 errors to any transmitted message:

```
sage: n_err = 2
sage: Chan = channels.StaticErrorRateChannel(GF(59)^40, n_err)
sage: Chan
Static error rate channel creating 2 errors, of input and output space
Vector space of dimension 40 over Finite Field of size 59
```

We can also pass a tuple for the number of errors:

```
sage: n_err = (1, 10)
sage: Chan = channels.StaticErrorRateChannel(GF(59)^40, n_err)
sage: Chan
Static error rate channel creating between 1 and 10 errors,
of input and output space Vector space of dimension 40 over Finite Field of size 59
```

number_errors()

Return the number of errors created by `self`.

EXAMPLES:

```
sage: n_err = 3
sage: Chan = channels.StaticErrorRateChannel(GF(59)^6, n_err)
sage: Chan.number_errors()
(3, 3)
```

transmit_unsafe(*message*)

Return `message` with as many errors as `self._number_errors` in it.

If `self._number_errors` was passed as a tuple for the number of errors, it will pick a random integer between the bounds of the tuple and use it as the number of errors.

This method does not check if `message` belongs to the input space of `self`.

INPUT:

- `message` – a vector

OUTPUT:

- a vector of the output space

EXAMPLES:

```
sage: F = GF(59)^6
sage: n_err = 2
sage: Chan = channels.StaticErrorRateChannel(F, n_err)
sage: msg = F((4, 8, 15, 16, 23, 42))
sage: set_random_seed(10)
sage: Chan.transmit_unsafe(msg)
(4, 8, 4, 16, 23, 53)
```

This checks that github issue #19863 is fixed:

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

sage.coding.channel.**format_interval**(*t*)

Return a formatted string representation of `t`.

This method should be called by any representation function in Channel classes.

Note: This is a helper function, which should only be used when implementing new channels.

INPUT:

- `t` – a list or a tuple

OUTPUT:

- a string

sage.coding.channel.**random_error_vector**(*n*, *F*, *error_positions*)

Return a vector of length `n` over `F` filled with random non-zero coefficients at the positions given by `error_positions`.

Note: This is a helper function, which should only be used when implementing new channels.

INPUT:

- `n` – the length of the vector
- `F` – the field over which the vector is defined
- `error_positions` – the non-zero positions of the vector

OUTPUT:

- a vector of `F`

AUTHORS:

This function is taken from codinglib (https://bitbucket.org/jsrn/codinglib/) and was written by Johan Nielsen.

EXAMPLES:

```
sage: from sage.coding.channel import random_error_vector
sage: random_error_vector(5, GF(2), [1,3])
(0, 1, 0, 1, 0)
```

CHAPTER

THREE

ENCODERS

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

AUTHORS:

- David Lucas (2015): initial version

class sage.coding.encoder.**Encoder**(*code*)

Bases: SageObject

Abstract top-class for *Encoder* objects.

Every encoder class for linear codes (of any metric) should inherit from this abstract class.

To implement an encoder, you need to:

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

REFERENCES:

- [Nie]

code()

Returns the code for this *Encoder*.

EXAMPLES:

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

(continues on next page)

(continued from previous page)

```
sage: E.code() == C
True
```

encode(*word*)

Transforms an element of the message space into a codeword.

This is a default implementation which assumes that the message space of the encoder is F^k, where F is `sage.coding.linear_code_no_metric.AbstractLinearCodeNoMetric.base_field()` and k is `sage.coding.linear_code_no_metric.AbstractLinearCodeNoMetric.dimension()`. If this is not the case, this method should be overwritten by the subclass.

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

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

```
E(word)
```

INPUT:

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

OUTPUT:

- a vector of `code()`.

EXAMPLES:

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

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

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

generator_matrix()

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

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

EXAMPLES:

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

message_space()

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

EXAMPLES:

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

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

Return the message corresponding to the codeword `c`.

This is the inverse of *encode()*.

INPUT:

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

OUTPUT:

- an element of the message space of `self`

EXAMPLES:

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

unencode_nocheck(*c*)

Returns the message corresponding to `c`.

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

AUTHORS:

This function is taken from codinglib [Nie]

INPUT:

- c – a codeword of `code()`.

OUTPUT:

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

EXAMPLES:

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

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

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

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

Bases: `Exception`

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

CHAPTER

FOUR

DECODERS

Representation of an error-correction algorithm for a code.

AUTHORS:

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

class sage.coding.decoder.**Decoder**(*code*, *input_space*, *connected_encoder_name*)

Bases: SageObject

Abstract top-class for *Decoder* objects.

Every decoder class for linear codes (of any metric) should inherit from this abstract class.

To implement an decoder, you need to:

- inherit from *Decoder*
- call Decoder.__init__ in the subclass constructor. Example: super().__init__(code, input_space, connected_encoder_name). By doing that, your subclass will have all the parameters described above initialized.
- Then, you need to override one of decoding methods, either *decode_to_code()* or *decode_to_message()*. You can also override the optional method *decoding_radius()*.
- By default, comparison of *Decoder* (using methods __eq__ and __ne__) are by memory reference: if you build the same decoder twice, they will be different. If you need something more clever, override __eq__ and __ne__ in your subclass.
- As *Decoder* is not designed to be instantiated, it does not have any representation methods. You should implement _repr_ and _latex_ methods in the subclass.

code()

Return the code for this *Decoder*.

EXAMPLES:

```
sage: G = Matrix(GF(2), [[1,1,1,0,0,0,0], [1,0,0,1,1,0,0],
....:                     [0,1,0,1,0,1,0], [1,1,0,1,0,0,1]])
sage: C = LinearCode(G)
sage: D = C.decoder()
sage: D.code()
[7, 4] linear code over GF(2)
```

connected_encoder()

Return the connected encoder of self.

EXAMPLES:

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

decode_to_code(*r*)

Correct the errors in r and return a codeword.

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

INPUT:

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

OUTPUT:

- a vector of *code()*.

EXAMPLES:

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

decode_to_message(*r*)

Decode r to the message space of *connected_encoder()*.

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

INPUT:

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

OUTPUT:

- a vector of *message_space()*.

EXAMPLES:

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

(continues on next page)

(continued from previous page)

```
sage: C = LinearCode(G)
sage: word = vector(GF(2), (1, 1, 0, 0, 1, 1, 0))
sage: w_err = word + vector(GF(2), (1, 0, 0, 0, 0, 0, 0))
sage: D = C.decoder()
sage: D.decode_to_message(w_err)
(0, 1, 1, 0)
```

classmethod decoder_type()

Returns the set of types of `self`.

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

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

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

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

EXAMPLES:

We call it on a class:

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

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

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

decoding_radius(***kwargs*)

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

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

EXAMPLES:

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

input_space()

Return the input space of `self`.

EXAMPLES:

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

message_space()

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

EXAMPLES:

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

exception sage.coding.decoder.**DecodingError**

Bases: `Exception`

Special exception class to indicate an error during decoding.

Catalogs for available constructions of the basic objects and for bounds on the parameters of linear codes are provided.

CHAPTER

FIVE

INDEX OF CHANNELS

Channels in Sage implement the information theoretic notion of transmission of messages.

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

- *channel.ErrorErasureChannel*
- *channel.QarySymmetricChannel*
- *channel.StaticErrorRateChannel*

To import these names into the global namespace, use:

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

CHAPTER

SIX

INDEX OF CODE CONSTRUCTIONS

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

6.1 Families of Codes (Rich representation)

ParityCheckCode()	Parity check codes
CyclicCode()	Cyclic codes
BCHCode()	BCH Codes
GeneralizedReedSolomonCode()	Generalized Reed-Solomon codes
ReedSolomonCode()	Reed-Solomon codes
BinaryReedMullerCode()	Binary Reed-Muller codes
ReedMullerCode()	q-ary Reed-Muller codes
HammingCode()	Hamming codes
GolayCode()	Golay codes
GoppaCode()	Goppa codes
KasamiCode()	Kasami codes

6.2 Families of Codes (Generator matrix representation)

DuadicCodeEvenPair()	Duadic codes, even pair
DuadicCodeOddPair()	Duadic codes, odd pair
QuadraticResidueCode()	Quadratic residue codes
ExtendedQuadraticResidueCode()	Extended quadratic residue codes
QuadraticResidueCodeEvenPair()	Even-like quadratic residue codes
QuadraticResidueCodeOddPair()	Odd-like quadratic residue codes
QuasiQuadraticResidueCode()	Quasi quadratic residue codes (Requires GAP/Guava)
ToricCode()	Toric codes
WalshCode()	Walsh codes
from_parity_check_matrix()	Construct a code from a parity check matrix
random_linear_code()	Construct a random linear code
RandomLinearCodeGuava()	Construct a random linear code through Guava (Requires GAP/Guava)

6.3 Derived Codes

SubfieldSubcode()	Subfield subcodes
ExtendedCode()	Extended codes
PuncturedCode()	Puncturedcodes

To import these names into the global namespace, use:

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

CHAPTER

SEVEN

INDEX OF DECODERS

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

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

Extended code decoder

- *extended_code.ExtendedCodeOriginalCodeDecoder*

Subfield subcode decoder

- *subfield_subcode.SubfieldSubcodeOriginalCodeDecoder*

Generalized Reed-Solomon code decoders

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

Generic decoders

- *linear_code.LinearCodeNearestNeighborDecoder*
- *linear_code.LinearCodeSyndromeDecoder*
- *information_set_decoder.LinearCodeInformationSetDecoder*

Cyclic code decoder

- *cyclic_code.CyclicCodeSurroundingBCHDecoder*

BCH code decoder

- *bch_code.BCHUnderlyingGRSDecoder*

Punctured code decoder

- *punctured_code.PuncturedCodeOriginalCodeDecoder*

Evaluation and differential AG code decoders

- *ag_code_decoders.EvaluationAGCodeUniqueDecoder*
- *ag_code_decoders.DifferentialAGCodeUniqueDecoder*

To import these names into the global namespace, use:

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

CHAPTER

EIGHT

INDEX OF ENCODERS

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

Cyclic code encoders

- *cyclic_code.CyclicCodePolynomialEncoder*
- *cyclic_code.CyclicCodeVectorEncoder*

Extended code encoder

- *extended_code.ExtendedCodeExtendedMatrixEncoder*

Generic encoders

- *linear_code.LinearCodeGeneratorMatrixEncoder*
- *linear_code_no_metric.LinearCodeSystematicEncoder*

Generalized Reed-Solomon code encoders

- *grs_code.GRSEvaluationVectorEncoder*
- *grs_code.GRSEvaluationPolynomialEncoder*

Punctured code encoder

- *punctured_code.PuncturedCodePuncturedMatrixEncoder*

To import these names into the global namespace, use:

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

CHAPTER

NINE

INDEX OF BOUNDS ON THE PARAMETERS OF CODES

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

codesize_upper_bound()	Return an upper bound on the number of codewords in a (possibly non-linear) code.
delsarte_bound_Q_matrix()	Delsarte bound on a code with Q matrix `q` and lower bound on min. dist. `d`.
delsarte_bound_additive_hamming_space()	Find a modified Delsarte bound on additive codes in Hamming space H_q^n of minimal distance d
delsarte_bound_constant_weight_code()	Find the Delsarte bound on a constant weight code.
delsarte_bound_hamming_space()	Find the Delsarte bound on codes in `H_q^n` of minimal distance `d`
dimension_upper_bound()	Return an upper bound for the dimension of a linear code.
eberlein()	Compute $E_k^{w,n}(x)$, the Eberlein polynomial.
elias_bound_asymp()	The asymptotic Elias bound for the information rate.
elias_upper_bound()	Return the Elias upper bound.
entropy()	Compute the entropy at x on the q-ary symmetric channel.
gilbert_lower_bound()	Return the Gilbert-Varshamov lower bound.
griesmer_upper_bound()	Return the Griesmer upper bound.
gv_bound_asymp()	The asymptotic Gilbert-Varshamov bound for the information rate, R.
gv_info_rate()	The Gilbert-Varshamov lower bound for information rate.
hamming_bound_asymp()	The asymptotic Hamming bound for the information rate.
hamming_upper_bound()	Return the Hamming upper bound.
krawtchouk()	Compute $K_l^{n,q}(x)$, the Krawtchouk (a.k.a. Kravchuk) polynomial.
mrrw1_bound_asymp()	The first asymptotic McEliese-Rumsey-Rodemich-Welsh bound.
plotkin_bound_asymp()	The asymptotic Plotkin bound for the information rate.
plotkin_upper_bound()	Return the Plotkin upper bound.
singleton_bound_asymp()	The asymptotic Singleton bound for the information rate.
singleton_upper_bound()	Return the Singleton upper bound.
volume_hamming()	Return the number of elements in a Hamming ball.

To import these names into the global namespace, use:

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

CHAPTER

TEN

LINEAR CODES

The following module is a base class for linear code objects regardless their metric.

10.1 Generic structures for linear codes of any metric

Class supporting methods available for linear codes over any metric (Hamming, rank).

class sage.coding.linear_code_no_metric.**AbstractLinearCodeNoMetric**(*base_field*, *length*, *default_encoder_name*, *default_decoder_name*, *metric='Hamming'*)

Bases: *AbstractCode*, Module

Abstract class for linear codes of any metric.

This class contains all the methods that can be used on any linear code of any metric. Every abstract class of linear codes over some metric (e.g. abstract class for linear codes over the Hamming metric, *sage.coding.linear_code.AbstractLinearCode*) should inherit from this class.

To create a new class of linear codes over some metrics, you need to:

- inherit from AbstractLinearCodeNoMetric
- call AbstractCode `__init__` method in the subclass constructor. Example: `super().__init__(length, "EncoderName", "DecoderName", "metric")`.
- add the following two lines on the class level:

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

- fill the dictionary of its encoders in `sage.coding.__init__.py` file. Example: I want to link the encoder `MyEncoderClass` to `MyNewCodeClass` under the name `MyEncoderName`. All I need to do is to write this line in the `__init__.py` file: `MyNewCodeClass._registered_encoders["NameOfMyEncoder"] = MyEncoderClass` and all instances of `MyNewCodeClass` will be able to use instances of `MyEncoderClass`.
- fill the dictionary of its decoders in `sage.coding.__init__` file. Example: I want to link the encoder `MyDecoderClass` to `MyNewCodeClass` under the name `MyDecoderName`. All I need to do is to write this line in the `__init__.py` file: `MyNewCodeClass._registered_decoders["NameOfMyDecoder"] = MyDecoderClass` and all instances of `MyNewCodeClass` will be able to use instances of `MyDecoderClass`.
- create a generic constructor representative of you abstract class. This generic constructor is a class for unstructured linear codes given by some generator and considered over the given metric. A good example

of this is *sage.coding.linear_code.LinearCode*, which is a generic constructor for *sage.coding.linear_code.AbstractLinearCode*, an abstract class for linear codes over the Hamming metric.

- set a private field in the `__init__` method specifying the generic constructor, (e.g. `MyAbstractCode._generic_constructor = MyCode`)

It is assumed that the subclass codes are linear over `base_field`. To test this, it is recommended to add a test suite test to the generic constructor. To do this, create a representative of your code C and run `TestSuite(C).run()`. A good example of this is in *sage.coding.linear_code.LinearCode*.

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

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

ambient_space()

Return the ambient vector space of `self`.

EXAMPLES:

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

base_field()

Return the base field of `self`.

EXAMPLES:

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

basis()

Return a basis of `self`.

OUTPUT:

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

EXAMPLES:

```
sage: C = codes.HammingCode(GF(2), 3)
sage: C.basis()
[
(1, 0, 0, 0, 0, 1, 1),
(0, 1, 0, 0, 1, 0, 1),
(0, 0, 1, 0, 1, 1, 0),
(0, 0, 0, 1, 1, 1, 1)
]
sage: C.basis().universe()
Vector space of dimension 7 over Finite Field of size 2
```

cardinality()

Return the size of this code.

EXAMPLES:

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

dimension()

Return 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
```

dual_code()

Return the dual code $C^{\perp}$ of the code C,

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

EXAMPLES:

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

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

Return a generator matrix of `self`.

INPUT:

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

EXAMPLES:

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

gens()

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

EXAMPLES:

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

information_set()

Return an information set of the code.

Return value of this method is cached.

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

OUTPUT:

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

EXAMPLES:

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

is_information_set(*positions*)

Return whether the given positions form an information set.

INPUT:

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

OUTPUT:

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

EXAMPLES:

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

is_permutation_automorphism(*g*)

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

EXAMPLES:

```
sage: C = codes.HammingCode(GF(3), 3)
sage: g = SymmetricGroup(13).random_element()
sage: C.is_permutation_automorphism(g)
0
```

(continues on next page)

(continued from previous page)

```
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_self_dual()

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

EXAMPLES:

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

is_self_orthogonal()

Return 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.GolayCode(GF(2))
sage: C.is_self_orthogonal()
True
sage: C = codes.HammingCode(GF(2), 3)
sage: C.is_self_orthogonal()
False
sage: C = codes.QuasiQuadraticResidueCode(11)  # optional - gap_package_guava
sage: C.is_self_orthogonal()                   # optional - gap_package_guava
True
```

is_subcode(*other*)

Return True if self is a subcode of other.

EXAMPLES:

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

(continues on next page)

(continued from previous page)

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

parity_check_matrix()

Return the parity check matrix of `self`.

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

EXAMPLES:

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

permuted_code(*p*)

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

EXAMPLES:

```
sage: C = codes.HammingCode(GF(2), 3)
sage: G = C.permutation_automorphism_group(); G
Permutation Group with generators
 [(4,5)(6,7), (4,6)(5,7), (2,3)(6,7), (2,4)(3,5), (1,2)(5,6)]
sage: g = G("(2,3)(6,7)")
sage: Cg = C.permuted_code(g)
sage: Cg
[7, 4] linear code over GF(2)
sage: C.generator_matrix() == Cg.systematic_generator_matrix()
True
```

rate()

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

EXAMPLES:

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

redundancy_matrix()

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

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

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

OUTPUT:

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

EXAMPLES:

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

standard_form(*return_permutation=True*)

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

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

INPUT:

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

OUTPUT:

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

EXAMPLES:

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

syndrome(*r*)

Return the syndrome of `r`.

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

INPUT:

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

OUTPUT:

- a column vector

EXAMPLES:

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

(continues on next page)

(continued from previous page)

```
↪1]])
sage: C  = LinearCode(G)
sage: r = vector(GF(2), (1,0,1,0,1,0,1))
sage: r in C
True
sage: C.syndrome(r)
(0, 0, 0)
```

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

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

Syndrome computation works fine on bigger fields:

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

systematic_generator_matrix(*systematic_positions=None*)

Return a systematic generator matrix of the code.

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

INPUT:

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

EXAMPLES:

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

Specific systematic positions can also be requested:

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

zero()

Return the zero vector of `self`.

EXAMPLES:

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

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

Bases: `Encoder`

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

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

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

INPUT:

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

EXAMPLES:

The following demonstrates the basic usage of `LinearCodeSystematicEncoder`:

```
sage: LCSE = codes.encoders.LinearCodeSystematicEncoder
sage: G = Matrix(GF(2), [[1,1,1,0,0,0,0,0],\
                         [1,0,0,1,1,0,0,0],\
                         [0,1,0,1,0,1,0,0],\
                         [1,1,0,1,0,0,1,1]])
sage: C = LinearCode(G)
sage: E = LCSE(C)
sage: E.generator_matrix()
[1 0 0 0 0 1 1 1]
[0 1 0 0 1 0 1 1]
[0 0 1 0 1 1 0 0]
[0 0 0 1 1 1 1 1]
sage: E2 = LCSE(C, systematic_positions=[5,4,3,2])
sage: E2.generator_matrix()
[1 0 0 0 0 1 1 1]
[0 1 0 0 1 0 1 1]
[1 1 0 1 0 0 1 1]
[1 1 1 0 0 0 0 0]
```

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

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

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

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

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

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

generator_matrix()

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

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

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

EXAMPLES:

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

We can ask for different systematic positions:

```
sage: E2 = LCSE(C, systematic_positions=[5,4,3,2])
sage: E2.generator_matrix()
[1 0 0 0 0 1 1]
[0 1 0 0 1 0 1]
[1 1 0 1 0 0 1]
[1 1 1 0 0 0 0]
```

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

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

systematic_permutation()

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

EXAMPLES:

```
sage: C = LinearCode(matrix(GF(2), [[1,0,0,0,1,1,0],\
                                    [0,1,0,1,0,1,0],\
                                    [0,0,0,0,0,0,1]]))
sage: E = codes.encoders.LinearCodeSystematicEncoder(C)
sage: E.systematic_positions()
(0, 1, 6)
sage: E.systematic_permutation()
[1, 2, 7, 3, 4, 5, 6]
```

systematic_positions()

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

EXAMPLES:

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

We take another matrix with a less nice shape:

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

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

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

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

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

There is a number of representatives of linear codes over a specific metric.

10.2 Generic structures for linear codes over the Hamming metric

10.2.1 Linear Codes

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

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

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

10.2.2 Families or Generic codes

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

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

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

AbstractLinearCode

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

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

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

LinearCode

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

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

```
sage: M = matrix(GF(2), [[1, 0, 0, 1, 0],\
                         [0, 1, 0, 1, 1],\
                         [0, 0, 1, 1, 1]])
sage: C = codes.LinearCode(M)
sage: C
[5, 3] linear code over GF(2)
```

(continues on next page)

(continued from previous page)

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

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

Further references

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

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

REFERENCES:

- [HP2003]
- [Gu]

AUTHORS:

- David Joyner (2005-11-22, 2006-12-03): initial version
- 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): github issue #3893: `random_element()` should pass on `*args` and `**kwds`.
- Thomas Feulner (2012-11): github issue #13723: deprecation of `hamming_weight()`
- Thomas Feulner (2013-10): added methods to compute a canonical representative and the automorphism group

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

Bases: *`AbstractLinearCodeNoMetric`*

Abstract base class for linear codes.

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

To implement a linear code, you need to:

- inherit from *`AbstractLinearCode`*
- call *`AbstractLinearCode`* `__init__` method in the subclass constructor. Example: `super().__init__(base_field, length, "EncoderName", "DecoderName")`. By doing that, your subclass will have its `length` parameter initialized and will be properly set as a member of the category framework. You need of course to complete the constructor by adding any additional parameter needed to describe properly the code defined in the subclass.
- Add the following two lines on the class level:

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

- fill the dictionary of its encoders in `sage.coding.__init__.py` file. Example: I want to link the encoder `MyEncoderClass` to `MyNewCodeClass` under the name `MyEncoderName`. All I need to do is to write this line in the `__init__.py` file: `MyNewCodeClass._registered_encoders["NameOfMyEncoder"] = MyEncoderClass` and all instances of `MyNewCodeClass` will be able to use instances of `MyEncoderClass`.
- fill the dictionary of its decoders in `sage.coding.__init__` file. Example: I want to link the encoder `MyDecoderClass` to `MyNewCodeClass` under the name `MyDecoderName`. All I need to do is to write this line in the `__init__.py` file: `MyNewCodeClass._registered_decoders["NameOfMyDecoder"] = MyDecoderClass` and all instances of `MyNewCodeClass` will be able to use instances of `MyDecoderClass`.

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

Note: *AbstractLinearCode* has a generic implementation of the method `__eq__` which uses the generator matrix and is quite slow. In subclasses you are encouraged to override `__eq__` and `__hash__`.

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

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

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

Assmus and Mattson Theorem (section 8.4, page 303 of [HP2003]): Let $A_0, A_1, ..., A_n$ be the weights of the codewords in a binary linear $[n, k, d]$ code C, and let $A_0^*, A_1^*, ..., A_n^*$ be the weights of the codewords in its dual $[n, n-k, d^*]$ code C^*. Fix a t, $0 < t < d$, and let

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

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
```

(continues on next page)

(continued from previous page)

```
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 [HP2003]_)
```

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

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

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

EXAMPLES:

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

automorphism_group_gens(*equivalence='semilinear'*)

Return generators of the automorphism group of `self`.

INPUT:

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

OUTPUT:

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

EXAMPLES:

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

```
sage: libgap.set_seed(0)                                                #␣
↪optional - sage.libs.gap
0
sage: C = codes.HammingCode(GF(4, 'z'), 3)
```

(continues on next page)

(continued from previous page)

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

binomial_moment(*i*)

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

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

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

EXAMPLES:

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

Warning: This is slow.

REFERENCE:

- [Du2004]

canonical_representative(*equivalence='semilinear'*)

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

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

INPUT:

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

OUTPUT:

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

EXAMPLES:

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

Check that the transporter element is correct:

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

Check if an equivalent code has the same canonical representative:

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

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

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

characteristic()

Return the characteristic of the base ring of `self`.

EXAMPLES:

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

characteristic_polynomial()

Return the characteristic polynomial of a linear code, as defined in [Lin1999].

EXAMPLES:

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

chinen_polynomial()

Return the Chinen zeta polynomial of the code.

EXAMPLES:

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

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

REFERENCES:

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

construction_x(*other*, *aux*)

Construction X applied to `self=C_1`, `other=C_2` and `aux=C_a`.

`other` must be a subcode of `self`.

If C_1 is a $[n, k_1, d_1]$ linear code and C_2 is a $[n, k_2, d_2]$ linear code, then $k_1 > k_2$ and $d_1 < d_2$. C_a must be a $[n_a, k_a, d_a]$ linear code, such that $k_a + k_2 = k_1$ and $d_a + d_1 \leq d_2$.

The method will then return a $[n+n_a, k_1, d_a+d_1]$ linear code.

EXAMPLES:

```
sage: C = codes.BCHCode(GF(2),15,7)
sage: C
[15, 5] BCH Code over GF(2) with designed distance 7
sage: D = codes.BCHCode(GF(2),15,5)
sage: D
[15, 7] BCH Code over GF(2) with designed distance 5
sage: C.is_subcode(D)
True
sage: C.minimum_distance()
7
sage: D.minimum_distance()
5
sage: aux = codes.HammingCode(GF(2),2)
sage: aux = aux.dual_code()
sage: aux.minimum_distance()
2
sage: Cx = D.construction_x(C,aux)
sage: Cx
[18, 7] linear code over GF(2)
sage: Cx.minimum_distance()
7
```

cosetGraph()

Return the coset graph of this linear code.

The coset graph of a linear code C is the graph whose vertices are the cosets of C, considered as a subgroup of the additive group of the ambient vector space, and two cosets are adjacent if they have representatives that differ in exactly one coordinate.

EXAMPLES:

```
sage: C = codes.GolayCode(GF(3))
sage: G = C.cosetGraph()
sage: G.is_distance_regular()
True
sage: C = codes.KasamiCode(8,2)
sage: G = C.cosetGraph()
sage: G.is_distance_regular()
True
```

ALGORITHM:

Instead of working with cosets we compute a (direct sum) complement of C. Let P be the projection of the cosets to the newly found subspace. Then two vectors are adjacent if they differ by $\lambda P(e_i)$ for some i.

covering_radius()

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

This method requires the optional GAP package Guava.

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

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

EXAMPLES:

```
sage: C = codes.HammingCode(GF(2), 5)
sage: C.covering_radius()                         # optional - gap_package_
↪guava
...1

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

direct_sum(*other*)

Return the direct sum of the codes `self` and `other`.

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

EXAMPLES:

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

divisor()

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

EXAMPLES:

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

extended_code()

Return `self` as an extended code.

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

EXAMPLES:

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

(continues on next page)

(continued from previous page)

```
sage: Cx = C.extended_code()
sage: Cx
Extension of [21, 18] Hamming Code over GF(4)
```

galois_closure(*F0*)

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

EXAMPLES:

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

genus()

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

EXAMPLES:

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

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

is_galois_closed()

Checks if `self` is equal to its Galois closure.

EXAMPLES:

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

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

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

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

Uses Robert Miller's double coset partition refinement work.

EXAMPLES:

```
sage: P.<x> = PolynomialRing(GF(2),"x")
sage: g = x^3 + x + 1
sage: C1 = codes.CyclicCode(length=7, generator_pol=g); C1
[7, 4] Cyclic Code over GF(2)
sage: C2 = codes.HammingCode(GF(2), 3); C2
[7, 4] Hamming Code over GF(2)
sage: C1.is_permutation_equivalent(C2)
True
sage: C1.is_permutation_equivalent(C2, algorithm="verbose")
(True, (3,4)(5,7,6))
sage: C1 = codes.random_linear_code(GF(2), 10, 5)
sage: C2 = codes.random_linear_code(GF(3), 10, 5)
sage: C1.is_permutation_equivalent(C2)
False
```

is_projective()

Test whether the code is projective.

A linear code C over a field is called *projective* when its dual Cd has minimum weight ≥ 3, i.e. when no two coordinate positions of C are linearly independent (cf. definition 3 from [BS2011] or 9.8.1 from [BH2012]).

EXAMPLES:

```
sage: C = codes.GolayCode(GF(2), False)
sage: C.is_projective()
True
sage: C.dual_code().minimum_distance()
8
```

A non-projective code:

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

juxtapose(*other*)

Juxtaposition of `self` and `other`

The two codes must have equal dimension.

EXAMPLES:

```
sage: C1 = codes.HammingCode(GF(2), 3)
sage: C2 = C1.juxtapose(C1)
sage: C2
[14, 4] linear code over GF(2)
```

minimum_distance(*algorithm=None*)

Return the minimum distance of `self`.

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

INPUT:

- algorithm – (default: None) the name of the algorithm to use to perform minimum distance computation. algorithm can be:
  - None, to use GAP methods (but not Guava)
  - "Guava", to use the optional GAP package Guava

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
```

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")                                    #␣
↪optional - sage.libs.gap
3
sage: libgap.SetAllInfoLevels(0)        # to suppress extra info messages   #␣
↪optional - sage.libs.gap
sage: C.minimum_distance(algorithm="guava")          # optional - gap_package_
↪guava
...3
```

module_composition_factors(*gp*)

Print the GAP record of the Meataxe composition factors module.

This is displayed 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()
sage: C.module_composition_factors(gp)                                       #␣
↪optional - sage.libs.gap
[ rec(
      IsIrreducible := true,
      IsOverFiniteField := true,
...) ]
```

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
[8, 4] linear code over GF(2)
sage: G = C.permutation_automorphism_group()                                    #␣
↪optional - sage.groups
sage: G.order()                                                                 #␣
↪optional - sage.groups
144
sage: GG = C.permutation_automorphism_group("codecan")                          #␣
↪optional - sage.groups
sage: GG == G                                                                   #␣
↪optional - sage.groups
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.GolayCode(GF(3))
sage: M11 = MathieuGroup(11)                                                    #␣
↪optional - sage.groups
sage: M11.order()                                                               #␣
↪optional - sage.groups
7920
sage: G = C.permutation_automorphism_group()  # long time (6s on sage.math,␣
↪2011)   # optional - sage.groups
sage: G.is_isomorphic(M11)                    # long time                       #␣
↪optional - sage.groups
True
sage: GG = C.permutation_automorphism_group("codecan")  # long time             #␣
↪optional - sage.groups
sage: GG == G # long time                                                       #␣
↪optional - sage.groups
True
```

Other examples:

```
sage: C = codes.GolayCode(GF(2))
sage: G = C.permutation_automorphism_group()                               #␣
↪optional - sage.groups
sage: G.order()                                                            #␣
↪optional - sage.groups
244823040
sage: C = codes.HammingCode(GF(2), 5)
sage: G = C.permutation_automorphism_group()                               #␣
↪optional - sage.groups
sage: G.order()                                                            #␣
↪optional - sage.groups
9999360
sage: C = codes.HammingCode(GF(3), 2); C
[4, 2] Hamming Code over GF(3)
sage: C.permutation_automorphism_group(algorithm="partition")              #␣
↪optional - sage.groups
Permutation Group with generators [(1,3,4)]
sage: C = codes.HammingCode(GF(4,"z"), 2); C
[5, 3] Hamming Code over GF(4)
sage: G = C.permutation_automorphism_group(algorithm="partition"); G       #␣
↪optional - sage.groups
Permutation Group with generators [(1,3)(4,5), (1,4)(3,5)]
sage: GG = C.permutation_automorphism_group(algorithm="codecan")    # long time,
↪ optional - sage.groups
sage: GG == G                                                       # long time,
↪ optional - sage.groups
True
sage: C.permutation_automorphism_group(algorithm="gap")  # optional - gap_
↪package_guava sage.groups
Permutation Group with generators [(1,3)(4,5), (1,4)(3,5)]
sage: C = codes.GolayCode(GF(3), True)
sage: C.permutation_automorphism_group(algorithm="gap")  # optional - gap_
↪package_guava sage.groups
Permutation Group with generators
 [(5,7)(6,11)(8,9)(10,12), (4,6,11)(5,8,12)(7,10,9), (3,4)(6,8)(9,11)(10,12),
  (2,3)(6,11)(8,12)(9,10), (1,2)(5,10)(7,12)(8,9)]
```

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

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

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

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

product_code(*other*)

Combines `self` with `other` to give the tensor product code.

If `self` is a $[n_1, k_1, d_1]$-code and `other` is a $[n_2, k_2, d_2]$-code, the product is a $[n_1 n_2, k_1 k_2, d_1 d_2]$-code.

Note that the two codes have to be over the same field.

EXAMPLES:

```
sage: C = codes.HammingCode(GF(2), 3)
sage: C
[7, 4] Hamming Code over GF(2)
sage: D = codes.ReedMullerCode(GF(2), 2, 2)
sage: D
Binary Reed-Muller Code of order 2 and number of variables 2
sage: A = C.product_code(D)
sage: A
[28, 16] linear code over GF(2)
sage: A.length() == C.length()*D.length()
True
sage: A.dimension() == C.dimension()*D.dimension()
True
sage: A.minimum_distance() == C.minimum_distance()*D.minimum_distance()
True
```

punctured(*L*)

Return a *sage.coding.punctured_code* object from L.

INPUT:

- L – List of positions to puncture

OUTPUT:

- an instance of *sage.coding.punctured_code*

EXAMPLES:

```
sage: C = codes.HammingCode(GF(2), 3)
sage: C.punctured([1,2])
Puncturing of [7, 4] Hamming Code over GF(2) on position(s) [1, 2]
```

relative_distance()

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

EXAMPLES:

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

shortened(*L*)

Return 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 [HP2003], C_L is $((C^\perp)^L)^\perp$. This is used in the construction below.

INPUT:

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

OUTPUT:

- Linear code, the shortened code described above

EXAMPLES:

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

spectrum(*algorithm=None*)

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

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

INPUT:

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

OUTPUT:

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

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

EXAMPLES:

```
sage: MS = MatrixSpace(GF(2),4,7)
sage: G = MS([[1,1,1,0,0,0,0],[1,0,0,1,1,0,0],[0,1,0,1,0,1,0],[1,1,0,1,0,0,1]])
sage: C = LinearCode(G)
sage: C.weight_distribution()
[1, 0, 0, 7, 7, 0, 0, 1]
sage: F.<z> = GF(2^2,"z")
sage: C = codes.HammingCode(F, 2); C
[5, 3] Hamming Code over GF(4)
sage: C.weight_distribution()
[1, 0, 0, 30, 15, 18]
sage: C = codes.HammingCode(GF(2), 3); C
[7, 4] Hamming Code over GF(2)
sage: C.weight_distribution(algorithm="leon")   # optional - gap_package_guava
[1, 0, 0, 7, 7, 0, 0, 1]
sage: C.weight_distribution(algorithm="gap")                                  #␣
↪optional - sage.libs.gap
[1, 0, 0, 7, 7, 0, 0, 1]
sage: C.weight_distribution(algorithm="binary")
[1, 0, 0, 7, 7, 0, 0, 1]

sage: # optional - gap_package_guava
sage: C = codes.HammingCode(GF(3), 3); C
[13, 10] Hamming Code over GF(3)
sage: C.weight_distribution() == C.weight_distribution(algorithm="leon")
True
sage: C = codes.HammingCode(GF(5), 2); C
```

(continues on next page)

(continued from previous page)

```
[6, 4] Hamming Code over GF(5)
sage: C.weight_distribution() == C.weight_distribution(algorithm="leon")
True
sage: C = codes.HammingCode(GF(7), 2); C
[8, 6] Hamming Code over GF(7)
sage: C.weight_distribution() == C.weight_distribution(algorithm="leon")
True
```

support()

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

OUTPUT:

- List of integers

EXAMPLES:

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

u_u_plus_v_code(*other*)

Return the $(u|u+v)$-construction with `self=u` and `other=v`.

This returns the code obtained through $(u|u+v)$-construction with `self` as u and `other` as v. Note that u and v must have equal lengths. For u a $[n, k_1, d_1]$-code and v a $[n, k_2, d_2]$-code this returns a $[2n, k_1+k_2, d]$-code, where $d = \min(2d_1, d_2)$.

EXAMPLES:

```
sage: C1 = codes.HammingCode(GF(2), 3)
sage: C2 = codes.HammingCode(GF(2), 3)
sage: D = C1.u_u_plus_v_code(C2)
sage: D
[14, 8] linear code over GF(2)
```

weight_distribution(*algorithm=None*)

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

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

INPUT:

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

OUTPUT:

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

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

EXAMPLES:

```
sage: MS = MatrixSpace(GF(2),4,7)
sage: G = MS([[1,1,1,0,0,0,0],[1,0,0,1,1,0,0],[0,1,0,1,0,1,0],[1,1,0,1,0,0,1]])
sage: C = LinearCode(G)
sage: C.weight_distribution()
[1, 0, 0, 7, 7, 0, 0, 1]
sage: F.<z> = GF(2^2,"z")
sage: C = codes.HammingCode(F, 2); C
[5, 3] Hamming Code over GF(4)
sage: C.weight_distribution()
[1, 0, 0, 30, 15, 18]
sage: C = codes.HammingCode(GF(2), 3); C
[7, 4] Hamming Code over GF(2)
sage: C.weight_distribution(algorithm="leon")   # optional - gap_package_guava
[1, 0, 0, 7, 7, 0, 0, 1]
sage: C.weight_distribution(algorithm="gap")                                  #␣
↪optional - sage.libs.gap
[1, 0, 0, 7, 7, 0, 0, 1]
sage: C.weight_distribution(algorithm="binary")
[1, 0, 0, 7, 7, 0, 0, 1]

sage: # optional - gap_package_guava
sage: C = codes.HammingCode(GF(3), 3); C
[13, 10] Hamming Code over GF(3)
sage: C.weight_distribution() == C.weight_distribution(algorithm="leon")
True
sage: C = codes.HammingCode(GF(5), 2); C
[6, 4] Hamming Code over GF(5)
sage: C.weight_distribution() == C.weight_distribution(algorithm="leon")
True
sage: C = codes.HammingCode(GF(7), 2); C
[8, 6] Hamming Code over GF(7)
sage: C.weight_distribution() == C.weight_distribution(algorithm="leon")
True
```

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

Return the weight enumerator polynomial of `self`.

This is the bivariate, homogeneous polynomial in x and y whose coefficient to $x^i y^{n-i}$ is the number of codewords of `self` of Hamming weight i. Here, n is the length of `self`.

INPUT:

- `names` – (default: `"xy"`) The names of the variables in the homogeneous polynomial. Can be given as a single string of length 2, or a single string with a comma, or as a tuple or list of two strings.
- `bivariate` – (default: `True`) Whether to return a bivariate, homogeneous polynomial or just a univariate polynomial. If set to `False`, then `names` will be interpreted as a single variable name and default to `"x"`.

OUTPUT:

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

EXAMPLES:

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

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

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

zeta_function(*name='T'*)

Return the Duursma zeta function of the code.

INPUT:

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

OUTPUT:

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

EXAMPLES:

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

zeta_polynomial(*name='T'*)

Return the Duursma zeta polynomial of this code.

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

INPUT:

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

OUTPUT:

- Polynomial over $\mathbf{Q}$

EXAMPLES:

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

(continues on next page)

(continued from previous page)

```
sage: C = codes.databases.best_linear_code_in_guava(6, 3, GF(2))     # optional -
↪ gap_package_guava
sage: C.minimum_distance()                                            # optional -
↪ gap_package_guava
3
sage: C.zeta_polynomial()                                             # optional -
↪ gap_package_guava
2/5*T^2 + 2/5*T + 1/5

sage: C = codes.HammingCode(GF(2), 4)
sage: C.zeta_polynomial()
16/429*T^6 + 16/143*T^5 + 80/429*T^4 + 32/143*T^3 + 30/143*T^2 + 2/13*T + 1/13

sage: F.<z> = GF(4,"z")
sage: MS = MatrixSpace(F, 3, 6)
sage: G = MS([[1,0,0,1,z,z],[0,1,0,z,1,z],[0,0,1,z,z,1]])
sage: C = LinearCode(G)  # the "hexacode"
sage: C.zeta_polynomial()
1
```

REFERENCES:

- [Du2001]

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

Bases: *AbstractLinearCode*

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

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

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

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

INPUT:

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

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

EXAMPLES:

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

(continues on next page)

(continued from previous page)

```
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
[7, 4] linear code over GF(5)
```

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

```
sage: C = codes.GeneralizedReedSolomonCode(GF(23).list(), 12)
sage: LinearCode(C)
[23, 12] linear code over GF(23)
```

Another example:

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

AUTHORS:

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

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

Return a generator matrix of `self`.

INPUT:

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

EXAMPLES:

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

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

Bases: *Encoder*

Encoder based on generator_matrix for Linear codes.

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

INPUT:

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

generator_matrix()

Return a generator matrix of the associated code of self.

EXAMPLES:

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

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

Bases: *Decoder*

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

INPUT:

- code – A code associated to this decoder

decode_to_code(*r*)

Corrects the errors in word and returns a codeword.

INPUT:

- r – a codeword of self

OUTPUT:

- a vector of self's message space

EXAMPLES:

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

(continues on next page)

(continued from previous page)

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

decoding_radius()

Return maximal number of errors `self` can decode.

EXAMPLES:

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

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

Bases: *Decoder*

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

The decoding algorithm works as follows:

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

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

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

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

INPUT:

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

EXAMPLES:

```
sage: G = Matrix(GF(3), [[1,0,0,1,0,1,0,1,2],
....:                    [0,1,0,2,2,0,1,1,0],
```

(continues on next page)

(continued from previous page)

```
....:                       [0,0,1,0,2,2,2,1,2]])
sage: C = LinearCode(G)
sage: D = codes.decoders.LinearCodeSyndromeDecoder(C)
sage: D
Syndrome decoder for [9, 3] linear code over GF(3) handling errors of weight up to 4
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

decode_to_code(*r*)

Correct the errors in `word` and return a codeword.

INPUT:

- `r` – a codeword of `self`

OUTPUT:

- a vector of `self`'s message space

EXAMPLES:

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

decoding_radius()

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

EXAMPLES:

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

maximum_error_weight()

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

Same as *decoding_radius()*.

EXAMPLES:

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

syndrome_table()

Return the syndrome lookup table of `self`.

EXAMPLES:

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

10.3 Generic structures for linear codes over the rank metric

10.3.1 Rank Metric

In coding theory, the most common metric is the Hamming metric, where distance between two codewords is given by the number of positions in which they differ. An alternative to this is the rank metric. Take two fields, $\mathbf{F}_q$ and $\mathbf{F}_{q^m}$, and define a code C to be a set of vectors of length n with entries from $\mathbf{F}_{q^m}$. Let c be a codeword. We can represent it as an $m \times n$ matrix M over $\mathbf{F}_q$.

A detailed description on the relationship between the two representations can be found in *sage.coding.linear_rank_metric.to_matrix_representation()* and *sage.coding.linear_rank_metric.from_matrix_representation()*.

We can define a metric using the rank of the matrix representation of the codewords. A distance between two codewords a, b is the rank of the matrix representation of $a - b$. A weight of a codeword c is the rank of the matrix representation of c.

This module allows representing rank metric codes which are linear over the big field $\mathbf{F}_{q^m}$, i.e. the usual linearity condition when the codewords are considered in vector form. One can also consider rank metric codes which are only linear over $\mathbf{F}_q$, but these are not currently supported in SageMath.

Note that linear rank metric codes per the definition of this file are mathematically just linear block codes, and so could be considered as a *sage.coding.linear_code.LinearCode*. However, since most of the functionality of that class is specific to the Hamming metric, the two notions are implemented as entirely different in SageMath. If you wish to investigate Hamming-metric properties of a linear rank metric code C, you can easily convert it by calling C_hamm = LinearCode(C).

10.3.2 Linear Rank Metric Code and Gabidulin Codes

The class *sage.coding.linear_rank_metric.LinearRankMetricCode* is the analog of *sage.coding.linear_code.LinearCode*, i.e. it is a generator matrix-based representation of a linear rank metric code without specific knowledge on the structure of the code.

Gabidulin codes are the main family of structured linear rank metric codes. These codes are the rank-metric analog of Reed-Solomon codes.

AbstractLinearRankMetricCode

This is a base class designed to contain methods, features and parameters shared by every linear rank metric code. For instance, generic algorithms for computing the minimum distance, etc. Many of these algorithms are slow, e.g. exponential in the code length. It also contains methods for swapping between vector and matrix representation of elements.

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

See *sage.coding.linear_rank_metric.AbstractLinearRankMetricCode* for details and examples.

LinearRankMetricCode

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

A LinearRankMetricCode is instantiated by providing a generator:

```
sage: G = Matrix(GF(64), [[1,1,0], [0,0,1]])
sage: C = codes.LinearRankMetricCode(G, GF(4))
sage: C
[3, 2] linear rank metric code over GF(64)/GF(4)
sage: C.generator_matrix()
[1 1 0]
[0 0 1]
sage: c = vector(GF(64), (1, 1, 1))
sage: c in C
True
```

Further references

Read more about rank metric and Gabidulin codes

AUTHORS:

- Marketa Slukova (2019-08-16): initial version

class sage.coding.linear_rank_metric.**AbstractLinearRankMetricCode**(*base_field*, *sub_field*, *length*, *default_encoder_name*, *default_decoder_name*, *basis=None*)

Bases: *AbstractLinearCodeNoMetric*

Abstract class for linear rank metric codes.

This class contains methods that can be used on families of linear rank metric codes. Every linear rank metric code class should inherit from this abstract class.

This class is intended for codes which are linear over the `base_field`.

Codewords of rank metric codes have two representations. They can either be written as a vector of length n over $\mathbf{F}_{q^m}$, or an $m \times n$ matrix over $\mathbf{F}_q$. This implementation principally uses the vector representation. However, one can always get the matrix representation using the `sage.coding.linear_rank_metric.AbstractLinearRankMetricCode.to_matrix()` method. To go back to a vector, use the `sage.coding.linear_rank_metric.AbstractLinearRankMetricCode.from_matrix()` method.

Instructions on how to make a new family of rank metric codes is analogous to making a new family of linear codes over the Hamming metric, instructions for which are in *sage.coding.linear_code.AbstractLinearCode*. For an example on, see `sage.coding.linear_rank_metric.AbstractLinearRankMetricCode.__init__()`

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

extension_degree()

Return m, the degree of the field extension of `self`.

Let `base_field` be $\mathbf{F}_{q^m}$ and `sub_field` be $\mathbf{F}_q$. Then this function returns m.

EXAMPLES:

```
sage: G = Matrix(GF(64), [[1,1,0], [0,0,1]])
sage: C = codes.LinearRankMetricCode(G, GF(4))
sage: C.extension_degree()
3
```

field_extension()

Return the field extension of `self`.

Let `base_field` be some field $\mathbf{F}_{q^m}$ and `sub_field` $\mathbf{F}_q$. This function returns the vector space of dimension m over $\mathbf{F}_q$.

EXAMPLES:

```
sage: G = Matrix(GF(64), [[1,1,0], [0,0,1]])
sage: C = codes.LinearRankMetricCode(G, GF(4))
sage: C.field_extension()
Vector space of dimension 3 over Finite Field in z2 of size 2^2
```

matrix_form_of_vector(*word*)

Return the matrix representation of a word.

INPUT:

- `word` – a vector over the `base_field` of `self`

EXAMPLES:

```
sage: G = Matrix(GF(64), [[1,1,0], [0,0,1]])
sage: C = codes.LinearRankMetricCode(G, GF(4))
sage: x = GF(64).gen()
sage: a = vector(GF(64), (x + 1, x + 1, 1))
sage: C.matrix_form_of_vector(a)
[1 1 1]
[1 1 0]
[0 0 0]
```

minimum_distance()

Return the minimum distance of `self`.

This algorithm simply iterates over all the elements of the code and returns the minimum weight.

EXAMPLES:

```
sage: F.<a> = GF(8)
sage: G = Matrix(F, [[1,a,a^2,0]])
sage: C = codes.LinearRankMetricCode(G, GF(2))
sage: C.minimum_distance()
3
```

rank_distance_between_vectors(*left*, *right*)

Return the rank of the matrix of `left` - `right`.

INPUT:

- `left` – a vector over the `base_field` of `self`
- `right` – a vector over the `base_field` of `self`

EXAMPLES:

```
sage: G = Matrix(GF(64), [[1,1,0], [0,0,1]])
sage: C = codes.LinearRankMetricCode(G, GF(4))
sage: x = GF(64).gen()
sage: a = vector(GF(64), (x + 1, x + 1, 1))
sage: b = vector(GF(64), (1, 0, 0))
sage: C.rank_distance_between_vectors(a, b)
2
```

rank_weight_of_vector(*word*)

Return the weight of the word, i.e. its rank.

INPUT:

- `word` – a vector over the `base_field` of `self`

EXAMPLES:

```
sage: G = Matrix(GF(64), [[1,1,0], [0,0,1]])
sage: C = codes.LinearRankMetricCode(G, GF(4))
sage: x = GF(64).gen()
sage: a = vector(GF(64), (x + 1, x + 1, 1))
sage: C.rank_weight_of_vector(a)
2
```

sub_field()

Return the sub field of `self`.

EXAMPLES:

```
sage: G = Matrix(GF(64), [[1,1,0], [0,0,1]])
sage: C = codes.LinearRankMetricCode(G, GF(4))
sage: C.sub_field()
Finite Field in z2 of size 2^2
```

vector_form_of_matrix(*word*)

Return the vector representation of a word.

INPUT:

- `word` – a matrix over the `sub_field` of `self`

EXAMPLES:

```
sage: G = Matrix(GF(64), [[1,1,0], [0,0,1]])
sage: C = codes.LinearRankMetricCode(G, GF(4))
sage: x = GF(64).gen()
sage: m = Matrix(GF(4), [[1, 1, 1], [1, 1, 0], [0, 0, 0]])
sage: C.vector_form_of_matrix(m)
(z6 + 1, z6 + 1, 1)
```

class sage.coding.linear_rank_metric.**LinearRankMetricCode**(*generator*, *sub_field=None*, *basis=None*)

Bases: *AbstractLinearRankMetricCode*

Linear rank metric codes over a finite field, represented using a generator matrix.

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

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

EXAMPLES:

```
sage: G = Matrix(GF(64), [[1,1,0], [0,0,1]])
sage: C = codes.LinearRankMetricCode(G, GF(4))
sage: C
[3, 2] linear rank metric code over GF(64)/GF(4)
sage: C.base_field()
Finite Field in z6 of size 2^6
sage: C.sub_field()
Finite Field in z2 of size 2^2
sage: C.length()
3
sage: C.dimension()
2
sage: C[2]
(z6, z6, 0)
sage: E = codes.encoders.LinearCodeGeneratorMatrixEncoder(C)
sage: word = vector(C.base_field(), [1, 0])
```

(continues on next page)

(continued from previous page)

```
sage: E(word)
(1, 1, 0)
```

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

Return a generator matrix of `self`.

INPUT:

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

EXAMPLES:

```
sage: G = Matrix(GF(64), [[1,1,0], [0,0,1]])
sage: C = codes.LinearRankMetricCode(G, GF(4))
sage: C.generator_matrix()
[1 1 0]
[0 0 1]
```

class sage.coding.linear_rank_metric.**LinearRankMetricCodeNearestNeighborDecoder**(*code*)

Bases: *Decoder*

Construct a decoder for Linear Rank Metric Codes.

This decoder will decode to the nearest codeword found.

decode_to_code(*r*)

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

INPUT:

- `r` – a codeword of `self`

OUTPUT:

- a vector of `self`'s message space

EXAMPLES:

```
sage: F.<a> = GF(4)
sage: G = Matrix(F, [[1,1,0]])
sage: C = codes.LinearRankMetricCode(G, GF(2))
sage: D = codes.decoders.LinearRankMetricCodeNearestNeighborDecoder(C)
sage: D.decode_to_code(vector(F, [a, a, 1]))
(a, a, 0)
```

decoding_radius()

Return maximal number of errors `self` can decode.

EXAMPLES:

```
sage: F.<a> = GF(8)
sage: G = Matrix(F, [[1,a,a^2,0]])
sage: C = codes.LinearRankMetricCode(G, GF(2))
sage: D = codes.decoders.LinearRankMetricCodeNearestNeighborDecoder(C)
sage: D.decoding_radius()
1
```

sage.coding.linear_rank_metric.**from_matrix_representation**(*w*, *base_field=None*, *basis=None*)

Return a vector representation of a matrix `w` over `base_field` in terms of `basis`.

Given an $m \times n$ matrix over $\mathbf{F}_q$ and some `basis` of $\mathbf{F}_{q^m}$ over $\mathbf{F}_q$, we can represent each of its columns as an element of $\mathbf{F}_{q^m}$, yielding a vector of length n over $\mathbf{F}_q$.

In case `base_field` is not given, we take $\mathbf{F}_{q^m}$, the field extension of $\mathbf{F}_q$ of degree m, the number of rows of `w`.

INPUT:

- `w` – a matrix over some field $\mathbf{F}_q$
- `base_field` – (default: `None`) an extension field of $\mathbf{F}_q$. If not specified, it is the field $\mathbf{F}_{q^m}$, where m is the number of rows of `w`.
- `basis` – (default: `None`) a basis of $\mathbf{F}_{q^m}$ as a vector space over $\mathbf{F}_q$. If not specified, given that $q = p^s$, let $1, \beta, \ldots, \beta^{sm}$ be the power basis that SageMath uses to represent $\mathbf{F}_{q^m}$. The default basis is then $1, \beta, \ldots, \beta^{m-1}$.

EXAMPLES:

```
sage: from sage.coding.linear_rank_metric import from_matrix_representation
sage: m = Matrix(GF(4), [[1, 1, 1], [1, 1, 0], [0, 0, 0]])
sage: from_matrix_representation(m)
(z6 + 1, z6 + 1, 1)

sage: v = vector(GF(4), (1, 0, 0))
sage: from_matrix_representation(v)
Traceback (most recent call last):
...
TypeError: Input must be a matrix
```

sage.coding.linear_rank_metric.**rank_distance**(*a*, *b*, *sub_field=None*, *basis=None*)

Return the rank of `a - b` as a matrix over `sub_field`.

Take two vectors `a`, `b` over some field $\mathbf{F}_{q^m}$. This function converts them to matrices over $\mathbf{F}_q$ and calculates the rank of their difference.

If `sub_field` is not specified, we take the prime subfield $\mathbf{F}_q$ of $\mathbf{F}_{q^m}$.

INPUT:

- `a` – a vector over some field $\mathbf{F}_{q^m}$
- `b` – a vector over some field $\mathbf{F}_{q^m}$
- `sub_field` – (default: `None`) a sub field of $\mathbf{F}_{q^m}$. If not specified, it is the prime subfield $\mathbf{F}_p$ of $\mathbf{F}_{q^m}$.
- `basis` – (default: `None`) a basis of $\mathbf{F}_{q^m}$ as a vector space over `sub_field`. If not specified, given that $q = p^s$, let $1, \beta, \ldots, \beta^{sm}$ be the power basis that SageMath uses to represent $\mathbf{F}_{q^m}$. The default basis is then $1, \beta, \ldots, \beta^{m-1}$.

EXAMPLES:

```
sage: from sage.coding.linear_rank_metric import rank_distance
sage: x = GF(64).gen()
sage: a = vector(GF(64), (x + 1, x + 1, 1))
sage: b = vector(GF(64), (1, 0, 0))
sage: rank_distance(a, b, GF(4))
2
```

(continues on next page)

(continued from previous page)

```
sage: c = vector(GF(4), (1, 0, 0))
sage: rank_distance(a, c, GF(4))
Traceback (most recent call last):
...
ValueError: The base field of (z6 + 1, z6 + 1, 1) and (1, 0, 0) has to be the same

sage: d = Matrix(GF(64), (1, 0, 0))
sage: rank_distance(a, d, GF(64))
Traceback (most recent call last):
...
TypeError: Both inputs have to be vectors

sage: e = vector(GF(64), (1, 0))
sage: rank_distance(a, e, GF(64))
Traceback (most recent call last):
...
ValueError: The length of (z6 + 1, z6 + 1, 1) and (1, 0) has to be the same
```

sage.coding.linear_rank_metric.**rank_weight**(*c*, *sub_field=None*, *basis=None*)

Return the rank of c as a matrix over sub_field.

If c is a vector over some field $\mathbf{F}_{q^m}$, the function converts it into a matrix over $\mathbf{F}_q$.

INPUT:

- c – a vector over some field $\mathbf{F}_{q^m}$; or a matrix over $\mathbf{F}_q$
- sub_field – (default: None) a sub field of $\mathbf{F}_{q^m}$. If not specified, it is the prime subfield $\mathbf{F}_p$ of $\mathbf{F}_{q^m}$.
- basis – (default: None) a basis of $\mathbf{F}_{q^m}$ as a vector space over sub_field. If not specified, given that $q = p^s$, let $1, \beta, \dots, \beta^{sm}$ be the power basis that SageMath uses to represent $\mathbf{F}_{q^m}$. The default basis is then $1, \beta, \dots, \beta^{m-1}$.

EXAMPLES:

```
sage: from sage.coding.linear_rank_metric import rank_weight
sage: x = GF(64).gen()
sage: a = vector(GF(64), (x + 1, x + 1, 1))
sage: rank_weight(a, GF(4))
2
```

sage.coding.linear_rank_metric.**to_matrix_representation**(*v*, *sub_field=None*, *basis=None*)

Return a matrix representation of v over sub_field in terms of basis.

Let $(b_1, b_2, \dots, b_m)$, $b_i \in GF(q^m)$, be a basis of $\mathbf{F}_{q^m}$ as a vector space over $\mathbf{F}_q$. Take an element $x \in \mathbf{F}_{q^m}$. We can write x as $x = u_1 b_1 + u_2 b_2 + \dots u_m b_m$, where $u_i \in GF(q)$. This way we can represent an element from $\mathbf{F}_{q^m}$ as a vector of length m over $\mathbf{F}_q$.

Given a vector v of length n over some field $\mathbf{F}_{q^m}$, we can represent each entry as a vector of length m, yielding an $m \times n$ matrix over sub_field. In case sub_field is not given, we take the prime subfield $\mathbf{F}_p$ of $\mathbf{F}_{q^m}$.

INPUT:

- v – a vector over some field $\mathbf{F}_{q^m}$
- sub_field – (default: None) a sub field of $\mathbf{F}_{q^m}$. If not specified, it is the prime subfield $\mathbf{F}_p$ of $\mathbf{F}_{q^m}$.

- basis – (default: None) a basis of $\mathbf{F}_{q^m}$ as a vector space over sub_field. If not specified, given that $q = p^s$, let $1, \beta, \ldots, \beta^{sm}$ be the power basis that SageMath uses to represent $\mathbf{F}_{q^m}$. The default basis is then $1, \beta, \ldots, \beta^{m-1}$.

EXAMPLES:

```
sage: from sage.coding.linear_rank_metric import to_matrix_representation
sage: x = GF(64).gen()
sage: a = vector(GF(64), (x + 1, x + 1, 1))
sage: to_matrix_representation(a, GF(4))
[1 1 1]
[1 1 0]
[0 0 0]

sage: m = Matrix(GF(4), [[1, 1, 1], [1, 1, 0], [0, 0, 0]])
sage: to_matrix_representation(m)
Traceback (most recent call last):
...
TypeError: Input must be a vector
```

CHAPTER

ELEVEN

FAMILIES OF LINEAR CODES

Famous families of codes, listed below, are represented in Sage by their own classes. For some of them, implementations of special decoding algorithms or computations for structural invariants are available.

11.1 Parity-check code

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

A parity-check code of dimension k over $\mathbf{F}_q$ is the set: $\{(m_1, m_2, \dots, m_k, -\Sigma_{i=1}^{k} m_i) \mid (m_1, m_2, \dots, m_k) \in \mathbf{F}_q^k\}$

REFERENCE:

- [Wel1988]

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

Bases: *AbstractLinearCode*

Representation of a parity-check code.

INPUT:

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

EXAMPLES:

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

minimum_distance()

Return the minimum distance of self.

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

EXAMPLES:

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

class sage.coding.parity_check_code.**ParityCheckCodeGeneratorMatrixEncoder**(*code*)

Bases: *LinearCodeGeneratorMatrixEncoder*

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

INPUT:

- code – the associated code of this encoder.

EXAMPLES:

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

Actually, we can construct the encoder from C directly:

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

generator_matrix()

Return a generator matrix of self.

EXAMPLES:

```
sage: C = codes.ParityCheckCode(GF(5),7)
sage: E = codes.encoders.ParityCheckCodeGeneratorMatrixEncoder(C)
sage: E.generator_matrix()
[1 0 0 0 0 0 0 4]
[0 1 0 0 0 0 0 4]
[0 0 1 0 0 0 0 4]
[0 0 0 1 0 0 0 4]
[0 0 0 0 1 0 0 4]
[0 0 0 0 0 1 0 4]
[0 0 0 0 0 0 1 4]
```

class sage.coding.parity_check_code.**ParityCheckCodeStraightforwardEncoder**(*code*)

Bases: *Encoder*

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

INPUT:

- code – the associated code of this encoder.

EXAMPLES:

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

Actually, we can construct the encoder from C directly:

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

encode(*message*)

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

INPUT:

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

OUTPUT:

- A codeword in associated code of `self`.

EXAMPLES:

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

message_space()

Return the message space of `self`.

EXAMPLES:

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

unencode_nocheck(*word*)

Return the message corresponding to the vector `word`.

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

INPUT:

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

OUTPUT:

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

EXAMPLES:

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

11.2 Hamming codes

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

REFERENCES:

- [Rot2006]

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

Bases: *AbstractLinearCode*

Representation of a Hamming code.

INPUT:

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

EXAMPLES:

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

minimum_distance()

Return the minimum distance of `self`.

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

EXAMPLES:

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

parity_check_matrix()

Return a parity check matrix of `self`.

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

- the note over section 2.3 in [Rot2006], pages 47-48
- the dedicated paragraph in [HP2003], page 30

EXAMPLES:

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

11.3 Cyclic code

Let F be a field. A $[n, k]$ code C over F is called cyclic if every cyclic shift of a codeword is also a codeword [Rot2006]:

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

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

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

class sage.coding.cyclic_code.**CyclicCode**(*generator_pol=None*, *length=None*, *code=None*, *check=True*, *D=None*, *field=None*, *primitive_root=None*)

Bases: *AbstractLinearCode*

Representation of a cyclic code.

We propose three different ways to create a new *CyclicCode*, either by providing:

- the generator polynomial and the length (1)
- an existing linear code. In that case, a generator polynomial will be computed from the provided linear code's parameters (2)
- (a subset of) the defining set of the cyclic code (3)

For now, only single-root cyclic codes are implemented. That is, only cyclic codes such that its length n and field order q are coprimes.

Depending on which behaviour you want, you need to specify the names of the arguments to *CyclicCode*. See EXAMPLES section below for details.

INPUT:

- `generator_pol` – (default: `None`) the generator polynomial of `self`. That is, the highest-degree monic polynomial which divides every polynomial representation of a codeword in `self`.
- `length` – (default: `None`) the length of `self`. It has to be bigger than the degree of `generator_pol`.
- `code` – (default: `None`) a linear code.
- `check` – (default: `False`) a boolean representing whether the cyclicity of `self` must be checked while finding the generator polynomial. See *find_generator_polynomial()* for details.
- `D` – (default: `None`) a list of integers between `0` and `length-1`, corresponding to (a subset of) the defining set of the code. Will be modified if it is not cyclotomic-closed.
- `field` – (default: `None`) the base field of `self`.
- `primitive_root` – (default: `None`) the primitive root of the splitting field which contains the roots of the generator polynomial. It has to be of multiplicative order `length` over this field. If the splitting field is not `field`, it also have to be a polynomial in `zx`, where `x` is the degree of the extension over the prime field. For instance, over `GF(16)`, it must be a polynomial in `z4`.

EXAMPLES:

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

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

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

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

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

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

bch_bound(*arithmetic=False*)

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

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

INPUT:

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

OUTPUT:

- `(delta + 1, (l, c))` – such that `delta + 1` is the BCH bound, and `l, c` are the parameters of the largest arithmetic sequence

EXAMPLES:

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

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

check_polynomial()

Return the check polynomial of `self`.

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

EXAMPLES:

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

defining_set(*primitive_root=None*)

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

INPUT:

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

EXAMPLES:

We provide a defining set at construction time:

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

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

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

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

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

Both operations give the same result:

```
sage: C1 = codes.CyclicCode(length=n, field=F, D=[1, 2, 4])
sage: C1.generator_polynomial() == g
True
```

Another one, in a reversed order:

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

field_embedding()

Return the base field embedding into the splitting field.

EXAMPLES:

```
sage: F.<x> = GF(2)[]
sage: n = 7
sage: g = x ** 3 + x + 1
sage: C = codes.CyclicCode(generator_pol=g, length=n)
sage: C.field_embedding()
Ring morphism:
  From: Finite Field of size 2
  To:   Finite Field in z3 of size 2^3
  Defn: 1 |--> 1
```

generator_polynomial()

Return the generator polynomial of `self`.

EXAMPLES:

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

parity_check_matrix()

Return the parity check matrix of `self`.

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

EXAMPLES:

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

primitive_root()

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

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

EXAMPLES:

```
sage: F.<x> = GF(2)[]
sage: n = 7
sage: g = x ** 3 + x + 1
sage: C = codes.CyclicCode(generator_pol=g, length=n)
sage: C.primitive_root()
z3

sage: F = GF(16, 'a')
sage: n = 15
sage: a = F.gen()
sage: Cc = codes.CyclicCode(length=n, field=F, D=[1,2],
....:                       primitive_root=a^2 + 1)
sage: Cc.primitive_root()
a^2 + 1
```

surrounding_bch_code()

Return the surrounding BCH code of `self`.

EXAMPLES:

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

class sage.coding.cyclic_code.**CyclicCodePolynomialEncoder**(*code*)

Bases: *Encoder*

An encoder encoding polynomials into codewords.

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

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

INPUT:

- code – The associated code of this encoder

EXAMPLES:

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

encode(p)

Transforms p into an element of the associated code of `self`.

INPUT:

- p – A polynomial from self message space

OUTPUT:

- A codeword in associated code of self

EXAMPLES:

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

message_space()

Return the message space of self

EXAMPLES:

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

unencode_nocheck(*c*)

Return the message corresponding to c. Does not check if c belongs to the code.

INPUT:

- c – A vector with the same length as the code

OUTPUT:

- An element of the message space

EXAMPLES:

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

class sage.coding.cyclic_code.**CyclicCodeSurroundingBCHDecoder**(*code*, ***kwargs*)

Bases: *Decoder*

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

INPUT:

- `code` – The associated code of this decoder.
- `**kwargs` – All extra arguments are forwarded to the BCH decoder

EXAMPLES:

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

bch_code()

Return the surrounding BCH code of *sage.coding.encoder.Encoder.code()*.

EXAMPLES:

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

bch_decoder()

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

EXAMPLES:

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

decode_to_code(*y*)

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

EXAMPLES:

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

decoding_radius()

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

EXAMPLES:

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

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

Bases: *Encoder*

An encoder which can encode vectors into codewords.

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

Let $m = (m_1, m_2, \dots, m_k)$ be a vector in F^k. This codeword can be seen as a polynomial over $F[x]$, as follows: $P_m = \Sigma_{i=0}^{k-1} m_i \times x^i$.

To encode m, this encoder does the multiplication $P_m g$.

INPUT:

- code – The associated code of this encoder

EXAMPLES:

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

encode(*m*)

Transforms m into an element of the associated code of self.

INPUT:

- m – an element from self's message space

OUTPUT:

- A codeword in the associated code of self

EXAMPLES:

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

generator_matrix()

Return a generator matrix of self

EXAMPLES:

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

(continues on next page)

(continued from previous page)

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

message_space()

Return the message space of `self`

EXAMPLES:

```
sage: F.<x> = GF(2)[]
sage: n = 7
sage: g = x ** 3 + x + 1
sage: C = codes.CyclicCode(generator_pol=g, length=n)
sage: E = codes.encoders.CyclicCodeVectorEncoder(C)
sage: E.message_space()
Vector space of dimension 4 over Finite Field of size 2
```

unencode_nocheck(*c*)

Return the message corresponding to `c`. Does not check if `c` belongs to the code.

INPUT:

- `c` – A vector with the same length as the code

OUTPUT:

- An element of the message space

EXAMPLES:

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

`sage.coding.cyclic_code.`**bch_bound**(*n*, *D*, *arithmetic=False*)

Return the BCH bound obtained for a cyclic code of length `n` and defining set `D`.

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

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

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

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

INPUT:

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

OUTPUT:

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

EXAMPLES:

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

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

sage.coding.cyclic_code.**find_generator_polynomial**(*code*, *check=True*)

Return a possible generator polynomial for code.

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

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

INPUT:

- code – a linear code
- check – whether the cyclicity should be checked

OUTPUT:

- the generator polynomial of code (if the code is cyclic).

EXAMPLES:

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

11.4 BCH code

Let $F = \mathbf{F}_q$ and Φ be the splitting field of $x^n - 1$ over F, with n a positive integer. Let also α be an element of multiplicative order n in Φ. Finally, let b, δ, ℓ be integers such that $0 \leq b \leq n$, $1 \leq \delta \leq n$ and α^ℓ generates the multiplicative group $\Phi^\times$.

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

class `sage.coding.bch_code.BCHCode`(*base_field*, *length*, *designed_distance*, *primitive_root=None*, *offset=1*, *jump_size=1*, *b=0*)

Bases: `CyclicCode`

Representation of a BCH code seen as a cyclic code.

INPUT:

- `base_field` – the base field for this code
- `length` – the length of the code
- `designed_distance` – the designed minimum distance of the code
- `primitive_root` – (default: `None`) the primitive root to use when creating the set of roots for the generating polynomial over the splitting field. It has to be of multiplicative order `length` over this field. If the splitting field is not `field`, it also has to be a polynomial in `zx`, where `x` is the degree of the extension field. For instance, over $\mathbf{F}_{16}$, it has to be a polynomial in `z4`.
- `offset` – (default: `1`) the first element in the defining set
- `jump_size` – (default: `1`) the jump size between two elements of the defining set. It must be coprime with the multiplicative order of `primitive_root`.
- `b` – (default: `0`) is exactly the same as `offset`. It is only here for retro-compatibility purposes with the old signature of `codes.BCHCode()` and will be removed soon.

EXAMPLES:

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

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

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

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

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

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

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

bch_to_grs()

Return the underlying GRS code from which `self` was derived.

EXAMPLES:

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

designed_distance()

Return the designed distance of `self`.

EXAMPLES:

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

jump_size()

Return the jump size between two consecutive elements of the defining set of `self`.

EXAMPLES:

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

offset()

Return the offset which was used to compute the elements in the defining set of `self`.

EXAMPLES:

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

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

Bases: `Decoder`

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

INPUT:

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

bch_word_to_grs(*c*)

Return `c` converted as a codeword of *grs_code()*.

EXAMPLES:

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

decode_to_code(*y*)

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

EXAMPLES:

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

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

```
sage: C = codes.BCHCode(GF(2), 31, 15)
sage: C
[31, 6] BCH Code over GF(2) with designed distance 15
sage: D = codes.decoders.BCHUnderlyingGRSDecoder(C, "GuruswamiSudan", tau=8)
sage: Dgrs = D.grs_decoder()
sage: c = vector(GF(2), [1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 1, 0, 0, 0,
....:                    1, 0, 1, 0, 1, 1, 1, 1, 0, 1, 1, 0, 1, 0, 0])
sage: y = vector(GF(2), [1, 1, 0, 0, 0, 0, 0, 1, 1, 0, 1, 0, 1, 0, 1, 1,
....:                    1, 0, 0, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0])
sage: print (c in C and (c-y).hamming_weight() == 8)
True
sage: Dgrs.decode_to_code(y)
[(1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 1, 1, 1, 1,
```

(continues on next page)

(continued from previous page)

```
     0, 1, 1, 0, 1, 0, 0),
 (1, z5^3 + z5^2 + z5 + 1, z5^4 + z5^2 + z5, z5^4 + z5^3 + z5^2 + 1, 0, 0,
     z5^4 + z5 + 1, 1, z5^4 + z5^2 + z5, 0, 1, z5^4 + z5, 1, 0, 1, 1, 1, 0,
     0, z5^4 + z5^3 + 1, 1, 0, 1, 1, 1, 1, z5^4 + z5^3 + z5 + 1, 1, 1, 0, 0)]
sage: D.decode_to_code(y) == [c]
True
```

decoding_radius()

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

EXAMPLES:

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

grs_code()

Return the underlying GRS code of *sage.coding.decoder.Decoder.code()*.

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

$$\begin{aligned} c \in \mathrm{BCH} &\iff \sum_{i=0}^{n-1} c_i (\alpha^{b+\ell j})^i = 0, \quad j = 0, \dots, \delta-2 \\ &\iff Hc = 0 \end{aligned}$$

where $H = A \times D$ with:

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

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

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

EXAMPLES:

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

grs_decoder()

Return the decoder used to decode words of *grs_code()*.

EXAMPLES:

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

grs_word_to_bch(*c*)

Return c converted as a codeword of *sage.coding.decoder.Decoder.code()*.

EXAMPLES:

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

11.5 Golay code

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

REFERENCES:

- [HP2003] pp. 31-33 for a definition of Golay codes.
- [MS2011]
- Wikipedia article Golay_code

class sage.coding.golay_code.**GolayCode**(*base_field*, *extended=True*)

Bases: *AbstractLinearCode*

Representation of a Golay Code.

INPUT:

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

EXAMPLES:

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

Another example with the perfect binary Golay code:

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

covering_radius()

Return the covering radius of `self`.

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

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

EXAMPLES:

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

dual_code()

Return the dual code of `self`.

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

EXAMPLES:

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

sage: Cd == C
True
```

generator_matrix()

Return a generator matrix of `self`

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

EXAMPLES:

```
sage: C = codes.GolayCode(GF(2), extended=True)
sage: C.generator_matrix()
[1 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 1 1 1 0 0 0 1 1]
[0 1 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 0 0 1 0 0 1 0]
[0 0 1 0 0 0 0 0 0 0 0 0 1 1 0 1 0 0 1 0 1 0 1 1]
[0 0 0 1 0 0 0 0 0 0 0 0 1 1 0 0 0 1 1 1 0 1 1 0]
[0 0 0 0 1 0 0 0 0 0 0 0 1 1 0 0 1 1 0 1 1 0 0 1]
[0 0 0 0 0 1 0 0 0 0 0 0 0 1 1 0 0 1 1 0 1 1 0 1]
[0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 1 0 0 1 1 0 1 1 1]
[0 0 0 0 0 0 0 1 0 0 0 0 1 0 1 1 0 1 1 1 1 0 0 0]
```

(continues on next page)

(continued from previous page)

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

minimum_distance()

Return the minimum distance of `self`.

The minimum distance of Golay codes is already known, and is thus returned immediately without computing anything.

EXAMPLES:

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

parity_check_matrix()

Return the parity check matrix of `self`.

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

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

EXAMPLES:

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

weight_distribution()

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

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

EXAMPLES:

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

11.6 Reed-Muller code

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

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

This file contains the following elements:

- *QAryReedMullerCode*, the class for Reed-Muller codes over non-binary field of size q and $r < q$
- *BinaryReedMullerCode*, the class for Reed-Muller codes over binary field and $r <= m$
- *ReedMullerVectorEncoder*, an encoder with a vectorial message space (for both the two code classes)
- *ReedMullerPolynomialEncoder*, an encoder with a polynomial message space (for both the code classes)

class sage.coding.reed_muller_code.**BinaryReedMullerCode**(*order*, *num_of_var*)

Bases: *AbstractLinearCode*

Representation of a binary Reed-Muller code.

For details on the definition of Reed-Muller codes, refer to *ReedMullerCode()*.

Note: It is better to use the aforementioned method rather than calling this class directly, as *ReedMullerCode()* creates either a binary or a q-ary Reed-Muller code according to the arguments it receives.

INPUT:

- order – The order of the Reed-Muller Code, i.e., the maximum degree of the polynomial to be used in the code.
- num_of_var – The number of variables used in the polynomial.

EXAMPLES:

A binary Reed-Muller code can be constructed by simply giving the order of the code and the number of variables:

```
sage: C = codes.BinaryReedMullerCode(2, 4)
sage: C
Binary Reed-Muller Code of order 2 and number of variables 4
```

minimum_distance()

Return the minimum distance of self.

The minimum distance of a binary Reed-Muller code of order d and number of variables m is q^{m-d}

EXAMPLES:

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

number_of_variables()

Return the number of variables of the polynomial ring used in self.

EXAMPLES:

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

order()

Return the order of `self`.

Order is the maximum degree of the polynomial used in the Reed-Muller code.

EXAMPLES:

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

class sage.coding.reed_muller_code.**QAryReedMullerCode**(*base_field*, *order*, *num_of_var*)

Bases: *AbstractLinearCode*

Representation of a q-ary Reed-Muller code.

For details on the definition of Reed-Muller codes, refer to *ReedMullerCode()*.

Note: It is better to use the aforementioned method rather than calling this class directly, as *ReedMullerCode()* creates either a binary or a q-ary Reed-Muller code according to the arguments it receives.

INPUT:

- `base_field` – A finite field, which is the base field of the code.
- `order` – The order of the Reed-Muller Code, i.e., the maximum degree of the polynomial to be used in the code.
- `num_of_var` – The number of variables used in polynomial.

Warning: For now, this implementation only supports Reed-Muller codes whose order is less than q.

EXAMPLES:

```
sage: from sage.coding.reed_muller_code import QAryReedMullerCode
sage: F = GF(3)
sage: C = QAryReedMullerCode(F, 2, 2)
sage: C
Reed-Muller Code of order 2 and 2 variables over Finite Field of size 3
```

minimum_distance()

Return the minimum distance between two words in `self`.

The minimum distance of a q-ary Reed-Muller code with order d and number of variables m is $(q-d)q^{m-1}$

EXAMPLES:

```
sage: from sage.coding.reed_muller_code import QAryReedMullerCode
sage: F = GF(5)
sage: C = QAryReedMullerCode(F, 2, 4)
sage: C.minimum_distance()
375
```

number_of_variables()

Return the number of variables of the polynomial ring used in `self`.

EXAMPLES:

```
sage: from sage.coding.reed_muller_code import QAryReedMullerCode
sage: F = GF(59)
sage: C = QAryReedMullerCode(F, 2, 4)
sage: C.number_of_variables()
4
```

order()

Return the order of `self`.

Order is the maximum degree of the polynomial used in the Reed-Muller code.

EXAMPLES:

```
sage: from sage.coding.reed_muller_code import QAryReedMullerCode
sage: F = GF(59)
sage: C = QAryReedMullerCode(F, 2, 4)
sage: C.order()
2
```

sage.coding.reed_muller_code.**ReedMullerCode**(*base_field*, *order*, *num_of_var*)

Return a Reed-Muller code.

A Reed-Muller Code of order r and number of variables m over a finite field F is the set:

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

INPUT:

- `base_field` – The finite field F over which the code is built.
- `order` – The order of the Reed-Muller Code, which is the maximum degree of the polynomial to be used in the code.
- `num_of_var` – The number of variables used in polynomial.

> **Warning:** For now, this implementation only supports Reed-Muller codes whose order is less than q. Binary Reed-Muller codes must have their order less than or equal to their number of variables.

EXAMPLES:

We build a Reed-Muller code:

```
sage: F = GF(3)
sage: C = codes.ReedMullerCode(F, 2, 2)
sage: C
Reed-Muller Code of order 2 and 2 variables over Finite Field of size 3
```

We ask for its parameters:

```
sage: C.length()
9
sage: C.dimension()
6
sage: C.minimum_distance()
3
```

If one provides a finite field of size 2, a Binary Reed-Muller code is built:

```
sage: F = GF(2)
sage: C = codes.ReedMullerCode(F, 2, 2)
sage: C
Binary Reed-Muller Code of order 2 and number of variables 2
```

class sage.coding.reed_muller_code.**ReedMullerPolynomialEncoder**(*code*, *polynomial_ring=None*)

Bases: *Encoder*

Encoder for Reed-Muller codes which encodes appropriate multivariate polynomials into codewords.

Consider a Reed-Muller code of order r, number of variables m, length n, dimension k over some finite field F. Let those variables be $(x_1, x_2, \dots, x_m)$. We order the monomials by lowest power on lowest index variables. If we have three monomials x_1x_2, $x_1x_2^2$ and $x_1^2x_2$, the ordering is: $x_1x_2 < x_1x_2^2 < x_1^2x_2$

Let now f be a polynomial of the multivariate polynomial ring $F[x_1, \dots, x_m]$.

Let $(\beta_1, \beta_2, \dots, \beta_q)$ be the elements of F ordered as they are returned by Sage when calling `F.list()`.

The aforementioned polynomial f is encoded as:

$(f(\alpha_{11}, \alpha_{12}, \dots, \alpha_{1m}), f(\alpha_{21}, \alpha_{22}, \dots, \alpha_{2m}), \dots, f(\alpha_{q^m1}, \alpha_{q^m2}, \dots, \alpha_{q^mm}))$

with $\alpha_{ij} = \beta_{i \bmod q^j}$ for all i, j.

INPUT:

- `code` – The associated code of this encoder.
- `polynomial_ring` – (default:`None`) The polynomial ring from which the message is chosen. If this is set to `None`, a polynomial ring in x will be built from the code parameters.

EXAMPLES:

```
sage: C1 = codes.ReedMullerCode(GF(2), 2, 4)
sage: E1 = codes.encoders.ReedMullerPolynomialEncoder(C1)
sage: E1
Evaluation polynomial-style encoder for
 Binary Reed-Muller Code of order 2 and number of variables 4
sage: C2 = codes.ReedMullerCode(GF(3), 2, 2)
sage: E2 = codes.encoders.ReedMullerPolynomialEncoder(C2)
sage: E2
Evaluation polynomial-style encoder for
 Reed-Muller Code of order 2 and 2 variables over Finite Field of size 3
```

We can also pass a predefined polynomial ring:

```
sage: R = PolynomialRing(GF(3), 2, 'y')
sage: C = codes.ReedMullerCode(GF(3), 2, 2)
sage: E = codes.encoders.ReedMullerPolynomialEncoder(C, R)
sage: E
Evaluation polynomial-style encoder for
 Reed-Muller Code of order 2 and 2 variables over Finite Field of size 3
```

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

```
sage: E = C1.encoder("EvaluationPolynomial")
sage: E
```

(continues on next page)

(continued from previous page)

```
Evaluation polynomial-style encoder for
 Binary Reed-Muller Code of order 2 and number of variables 4
```

encode(*p*)

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

INPUT:

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

OUTPUT:

- A codeword in associated code of self

EXAMPLES:

```
sage: F = GF(3)
sage: Fx.<x0,x1> = F[]
sage: C = codes.ReedMullerCode(F, 2, 2)
sage: E = C.encoder("EvaluationPolynomial")
sage: p = x0*x1 + x1^2 + x0 + x1 + 1
sage: c = E.encode(p); c
(1, 2, 0, 0, 2, 1, 1, 1, 1)
sage: c in C
True
```

If a polynomial with good monomial degree but wrong monomial degree is given, an error is raised:

```
sage: p = x0^2*x1
sage: E.encode(p)
Traceback (most recent call last):
...
ValueError: The polynomial to encode must have degree at most 2
```

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

```
sage: Qy.<y1,y2> = QQ[]
sage: p = y1^2 + 1
sage: E.encode(p)
Traceback (most recent call last):
...
ValueError: The value to encode must be in
Multivariate Polynomial Ring in x0, x1 over Finite Field of size 3
```

message_space()

Return the message space of self

EXAMPLES:

```
sage: F = GF(11)
sage: C = codes.ReedMullerCode(F, 2, 4)
sage: E = C.encoder("EvaluationPolynomial")
sage: E.message_space()
Multivariate Polynomial Ring in x0, x1, x2, x3 over Finite Field of size 11
```

points()

Return the evaluation points in the appropriate order as used by `self` when encoding a message.

EXAMPLES:

```
sage: F = GF(3)
sage: Fx.<x0,x1> = F[]
sage: C = codes.ReedMullerCode(F, 2, 2)
sage: E = C.encoder("EvaluationPolynomial")
sage: E.points()
[(0, 0), (1, 0), (2, 0), (0, 1), (1, 1), (2, 1), (0, 2), (1, 2), (2, 2)]
```

polynomial_ring()

Return the polynomial ring associated with `self`

EXAMPLES:

```
sage: F = GF(11)
sage: C = codes.ReedMullerCode(F, 2, 4)
sage: E = C.encoder("EvaluationPolynomial")
sage: E.polynomial_ring()
Multivariate Polynomial Ring in x0, x1, x2, x3 over Finite Field of size 11
```

unencode_nocheck(*c*)

Return the message corresponding to the codeword `c`.

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

INPUT:

- `c` – A codeword of `code()`.

OUTPUT:

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

EXAMPLES:

```
sage: F = GF(3)
sage: C = codes.ReedMullerCode(F, 2, 2)
sage: E = C.encoder("EvaluationPolynomial")
sage: c = vector(F, (1, 2, 0, 0, 2, 1, 1, 1, 1))
sage: c in C
True
sage: p = E.unencode_nocheck(c); p
x0*x1 + x1^2 + x0 + x1 + 1
sage: E.encode(p) == c
True
```

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

```
sage: c = vector(F, (1, 2, 0, 0, 2, 1, 0, 1, 1))
sage: c in C
False
sage: p = E.unencode_nocheck(c); p
-x0*x1 - x1^2 + x0 + 1
```

(continues on next page)

(continued from previous page)

```
sage: E.encode(p) == c
False
```

class sage.coding.reed_muller_code.**ReedMullerVectorEncoder**(*code*)

Bases: *Encoder*

Encoder for Reed-Muller codes which encodes vectors into codewords.

Consider a Reed-Muller code of order r, number of variables m, length n, dimension k over some finite field F. Let those variables be $(x_1, x_2, \ldots, x_m)$. We order the monomials by lowest power on lowest index variables. If we have three monomials $x_1 x_2$, $x_1 x_2^2$ and $x_1^2 x_2$, the ordering is: $x_1 x_2 < x_1 x_2^2 < x_1^2 x_2$

Let now $(v_1, v_2, \ldots, v_k)$ be a vector of F, which corresponds to the polynomial $f = \Sigma_{i=1}^{k} v_i x_i$.

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

The aforementioned polynomial f is encoded as:

$(f(\alpha_{11}, \alpha_{12}, \ldots, \alpha_{1m}), f(\alpha_{21}, \alpha_{22}, \ldots, \alpha_{2m}), \ldots, f(\alpha_{q^m 1}, \alpha_{q^m 2}, \ldots, \alpha_{q^m m}))$

with $\alpha_{ij} = \beta_{i \bmod q^j}$ for all i, j).

INPUT:

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

EXAMPLES:

```
sage: C1 = codes.ReedMullerCode(GF(2), 2, 4)
sage: E1 = codes.encoders.ReedMullerVectorEncoder(C1)
sage: E1
Evaluation vector-style encoder for
 Binary Reed-Muller Code of order 2 and number of variables 4
sage: C2 = codes.ReedMullerCode(GF(3), 2, 2)
sage: E2 = codes.encoders.ReedMullerVectorEncoder(C2)
sage: E2
Evaluation vector-style encoder for
 Reed-Muller Code of order 2 and 2 variables over Finite Field of size 3
```

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

```
sage: C=codes.ReedMullerCode(GF(2), 2, 4)
sage: E = C.encoder("EvaluationVector")
sage: E
Evaluation vector-style encoder for
 Binary Reed-Muller Code of order 2 and number of variables 4
```

generator_matrix()

Return a generator matrix of `self`

EXAMPLES:

```
sage: F = GF(3)
sage: C = codes.ReedMullerCode(F, 2, 2)
sage: E = codes.encoders.ReedMullerVectorEncoder(C)
sage: E.generator_matrix()
[1 1 1 1 1 1 1 1 1]
```

(continues on next page)

(continued from previous page)

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

points()

Return the points of F^m, where F is the base field and m is the number of variables, in order of which polynomials are evaluated on.

EXAMPLES:

```
sage: F = GF(3)
sage: Fx.<x0,x1> = F[]
sage: C = codes.ReedMullerCode(F, 2, 2)
sage: E = C.encoder("EvaluationVector")
sage: E.points()
[(0, 0), (1, 0), (2, 0), (0, 1), (1, 1), (2, 1), (0, 2), (1, 2), (2, 2)]
```

11.7 Reed-Solomon codes and Generalized Reed-Solomon codes

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

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

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

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

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

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

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

class sage.coding.grs_code.**GRSBerlekampWelchDecoder**(*code*)

Bases: *Decoder*

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

This algorithm recovers the error locator polynomial by solving a linear system. See [HJ2004] pp. 51-52 for details.

INPUT:

- code – a code associated to this decoder

EXAMPLES:

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

Actually, we can construct the decoder from C directly:

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

decode_to_code(*r*)

Correct the errors in r and returns a codeword.

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

INPUT:

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

OUTPUT:

- a vector of self.code()

EXAMPLES:

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

decode_to_message(*r*)

Decode r to an element in message space of self.

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

INPUT:

- r – a codeword of self

OUTPUT:

- a vector of self message space

EXAMPLES:

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

decoding_radius()

Return maximal number of errors that self can decode.

OUTPUT:

- the number of errors as an integer

EXAMPLES:

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

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

Bases: *Decoder*

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

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

- Puncture the erased coordinates which are identified in e.
- Create a new GRS code of length $n - w(e)$, where w is the Hamming weight function, and dimension k.
- Use Gao decoder over this new code one the punctured word built on the first step.
- Recover the original message from the decoded word computed on the previous step.
- Encode this message using an encoder over C.

INPUT:

- code – the associated code of this decoder

EXAMPLES:

```
sage: F = GF(59)
sage: n, k = 40, 12
```

(continues on next page)

(continued from previous page)

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

Actually, we can construct the decoder from C directly:

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

decode_to_message(*word_and_erasure_vector*)

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

INPUT:

- `word_and_erasure_vector` – a tuple whose:
  - first element is an element of the ambient space of the code
  - second element is a vector over $\mathbf{F}_2$ whose length is the same as the code's, containing erasure positions

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

OUTPUT:

- a vector of `self` message space

EXAMPLES:

```
sage: F = GF(59)
sage: n, k = 40, 12
sage: C = codes.GeneralizedReedSolomonCode(F.list()[:n], k)
sage: D = codes.decoders.GRSErrorErasureDecoder(C)
sage: c = C.random_element()
sage: n_era = randint(0, C.minimum_distance() - 2)
sage: Chan = channels.ErrorErasureChannel(C.ambient_space(),
....:                                     D.decoding_radius(n_era), n_era)
sage: y = Chan(c)
sage: D.connected_encoder().unencode(c) == D.decode_to_message(y)
True
```

decoding_radius(*number_erasures*)

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

INPUT:

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

OUTPUT:

- the number of errors as an integer

EXAMPLES:

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

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

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

class sage.coding.grs_code.**GRSEvaluationPolynomialEncoder**(*code*, *polynomial_ring=None*)

Bases: *Encoder*

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

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

The encoding of m will be the following codeword:

$$(\beta_1 p(\alpha_1), \dots, \beta_n p(\alpha_n)).$$

INPUT:

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

EXAMPLES:

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

Actually, we can construct the encoder from C directly:

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

We can also specify another polynomial ring:

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

encode(*p*)

Transform the polynomial `p` into a codeword of `code()`.

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

```
E(word)
```

INPUT:

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

OUTPUT:

- a codeword in associated code of `self`

EXAMPLES:

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

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

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

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

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

message_space()

Return the message space of `self`

EXAMPLES:

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

polynomial_ring()

Return the message space of `self`

EXAMPLES:

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

unencode_nocheck(*c*)

Return the message corresponding to the codeword `c`.

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

INPUT:

- `c` – a codeword of `code()`

OUTPUT:

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

EXAMPLES:

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

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

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

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

Bases: *Encoder*

Encoder for (Generalized) Reed-Solomon codes that encodes vectors into codewords.

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

$$p = \Sigma_{i=1}^{m} m_i x^i.$$

The encoding of m will be the following codeword:

$$(\beta_1 p(\alpha_1), \dots, \beta_n p(\alpha_n)).$$

INPUT:

- code – the associated code of this encoder

EXAMPLES:

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

Actually, we can construct the encoder from C directly:

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

generator_matrix()

Return a generator matrix of self

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

$$G = [g_{i,j}], g_{i,j} = \beta_j \alpha_j^i.$$

This matrix is a Vandermonde matrix.

EXAMPLES:

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

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

Bases: *Decoder*

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

Gao decoding algorithm uses early terminated extended Euclidean algorithm to find the error locator polynomial. See [Ga02] for details.

INPUT:

- code – the associated code of this decoder

EXAMPLES:

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

Actually, we can construct the decoder from C directly:

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

decode_to_code(*r*)

Correct the errors in r and returns a codeword.

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

INPUT:

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

OUTPUT:

- a vector of self.code()

EXAMPLES:

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

decode_to_message(*r*)

Decode r to an element in message space of self.

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

INPUT:

- r – a codeword of self

OUTPUT:

- a vector of self message space

EXAMPLES:

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

decoding_radius()

Return maximal number of errors that `self` can decode

OUTPUT:

- the number of errors as an integer

EXAMPLES:

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

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

Bases: *Decoder*

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

This algorithm uses early terminated extended euclidean algorithm to solve the key equations, as described in [Rot2006], pp. 183-195.

INPUT:

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

EXAMPLES:

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

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

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

decode_to_code(*r*)

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

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

INPUT:

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

OUTPUT:

- a vector of `self.code()`

EXAMPLES:

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

decode_to_message(*r*)

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

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

INPUT:

- `r` – a codeword of `self`

OUTPUT:

- a vector of `self` message space

EXAMPLES:

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

decoding_radius()

Return maximal number of errors that `self` can decode

OUTPUT:

- the number of errors as an integer

EXAMPLES:

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

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

Bases: *AbstractLinearCode*

Representation of a (Generalized) Reed-Solomon code.

INPUT:

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

EXAMPLES:

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

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

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

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

It is also possible to specify the column multipliers:

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

SageMath implements efficient decoding algorithms for GRS codes:

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

column_multipliers()

Return the vector of column multipliers of `self`.

EXAMPLES:

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

covering_radius()

Return the covering radius of `self`.

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

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

EXAMPLES:

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

dual_code()

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

EXAMPLES:

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

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

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

evaluation_points()

Return the vector of field elements used for the polynomial evaluations.

EXAMPLES:

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

is_generalized()

Return whether `self` is a Generalized Reed-Solomon code or a regular Reed-Solomon code.

`self` is a Generalized Reed-Solomon code if its column multipliers are not all 1.

EXAMPLES:

```
sage: F = GF(11)
sage: n, k = 10, 5
sage: C = codes.GeneralizedReedSolomonCode(F.list()[:n], k)
sage: C.column_multipliers()
(1, 1, 1, 1, 1, 1, 1, 1, 1, 1)
sage: C.is_generalized()
False
sage: colmults = [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 1]
sage: C2 = codes.GeneralizedReedSolomonCode(F.list()[:n], k, colmults)
sage: C2.is_generalized()
True
```

minimum_distance()

Return the minimum distance between any two words in `self`.

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

EXAMPLES:

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

multipliers_product()

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

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

EXAMPLES:

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

parity_check_matrix()

Return the parity check matrix of `self`.

EXAMPLES:

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

parity_column_multipliers()

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

EXAMPLES:

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

weight_distribution()

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

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

EXAMPLES:

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

`sage.coding.grs_code.`**ReedSolomonCode**(*base_field*, *length*, *dimension*, *primitive_root=None*)

Construct a classical Reed-Solomon code.

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

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

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

INPUT:

- `base_field` – the finite field for which to build the classical Reed-Solomon code.
- `length` – the length of the classical Reed-Solomon code. Must divide $q-1$ where q is the cardinality of `base_field`.
- `dimension` – the dimension of the resulting code.

- `primitive_root` – (default: `None`) a primitive n'th root of unity to use for constructing the classical Reed-Solomon code. If not supplied, one will be computed and can be recovered as `C.evaluation_points()[1]` where C is the code returned by this method.

EXAMPLES:

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

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

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

sage: Ccyc.generator_polynomial()
x^3 + 3*x^2 + x + 6
```

Another example over an extension field:

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

The primitive n'th root of unity can be recovered as the 2nd evaluation point of the code:

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

We can also supply a different primitive n'th root of unity:

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

11.8 Goppa code

This module implements Goppa codes and an encoder for them.

EXAMPLES:

```
sage: F = GF(2^6)
sage: R.<x> = F[]
sage: g = x^9 + 1
sage: L = [a for a in F.list() if g(a) != 0]
sage: C = codes.GoppaCode(g, L)
sage: C
[55, 16] Goppa code over GF(2)
sage: E = codes.encoders.GoppaCodeEncoder(C)
sage: E
Encoder for [55, 16] Goppa code over GF(2)
```

AUTHORS:

- Filip Ion, Marketa Slukova (2019-06): initial version

class sage.coding.goppa_code.**GoppaCode**(*generating_pol*, *defining_set*)

Bases: *AbstractLinearCode*

Implementation of Goppa codes.

Goppa codes are a generalization of narrow-sense BCH codes. These codes are defined by a generating polynomial g over a finite field $\mathbf{F}_{p^m}$, and a defining set L of elements from $\mathbf{F}_{p^m}$, which are not roots of g. The number of defining elements determines the length of the code.

In binary cases, the minimum distance is $2t+1$, where t is the degree of g.

INPUT:

- generating_pol – a monic polynomial with coefficients in a finite field $\mathbf{F}_{p^m}$, the code is defined over $\mathbf{F}_p$, p must be a prime number
- defining_set – a set of elements of $\mathbf{F}_{p^m}$ that are not roots of g, its cardinality is the length of the code

EXAMPLES:

```
sage: F = GF(2^6)
sage: R.<x> = F[]
sage: g = x^9 + 1
sage: L = [a for a in F.list() if g(a) != 0]
sage: C = codes.GoppaCode(g, L)
sage: C
[55, 16] Goppa code over GF(2)
```

distance_bound()

Return a lower bound for the minimum distance of the code.

Computed using the degree of the generating polynomial of self. The minimum distance is guaranteed to be bigger than or equal to this bound.

EXAMPLES:

```
sage: F = GF(2^3)
sage: R.<x> = F[]
sage: g = x^2 + x + 1
sage: L = [a for a in F.list() if g(a) != 0]
sage: C = codes.GoppaCode(g, L)
sage: C
[8, 2] Goppa code over GF(2)
sage: C.distance_bound()
3
sage: C.minimum_distance()
5
```

parity_check_matrix()

Return a parity check matrix for self.

The element in row t, column i is $h[i](D[i]^t)$, where:

- $h[i]$ – is the inverse of $g(D[i])$
- $D[i]$ – is the i-th element of the defining set

In the resulting $d \times n$ matrix we interpret each entry as an m-column vector and return a $dm \times n$ matrix.

EXAMPLES:

```
sage: F  = GF(2^3)
sage: R.<x> = F[]
sage: g = x^2 + x+ 1
sage: L = [a for a in F.list() if g(a) != 0]
sage: C = codes.GoppaCode(g, L)
sage: C
[8, 2] Goppa code over GF(2)
sage: C.parity_check_matrix()
[1 0 0 0 0 0 0 1]
[0 0 1 0 1 1 1 0]
[0 1 1 1 0 0 1 0]
[0 1 1 1 1 1 1 1]
[0 1 0 1 1 0 1 0]
[0 0 1 1 1 1 0 0]
```

class sage.coding.goppa_code.**GoppaCodeEncoder**(*code*)

Bases: *Encoder*

Encoder for Goppa codes

Encodes words represented as vectors of length k, where k is the dimension of `self`, with entries from $\mathbf{F}_p$, the prime field of the base field of the generating polynomial of `self`, into codewords of length n, with entries from $\mathbf{F}_p$.

EXAMPLES:

```
sage: F = GF(2^3)
sage: R.<x> = F[]
sage: g = x^2 + x + 1
sage: L = [a for a in F.list() if g(a) != 0]
sage: C = codes.GoppaCode(g, L)
sage: C
[8, 2] Goppa code over GF(2)
sage: E = codes.encoders.GoppaCodeEncoder(C)
sage: E
Encoder for [8, 2] Goppa code over GF(2)
sage: word = vector(GF(2), (0, 1))
sage: c = E.encode(word)
sage: c
(0, 1, 1, 1, 1, 1, 1, 0)
sage: c in C
True
```

generator_matrix()

A generator matrix for `self`

Dimension of resulting matrix is $k \times n$, where k is the dimension of `self` and n is the length of `self`.

EXAMPLES:

```
sage: F = GF(2^3)
sage: R.<x> = F[]
```

(continues on next page)

(continued from previous page)

```
sage: g = (x^2 + x + 1)^2
sage: L = [a for a in F.list() if g(a) != 0]
sage: C = codes.GoppaCode(g, L)
sage: C
[8, 2] Goppa code over GF(2)
sage: C.generator_matrix()
[1 0 0 1 0 1 1 1]
[0 1 1 1 1 1 1 0]
```

11.9 Kasami code

This module implements a construction for the extended Kasami codes. The "regular" Kasami codes are obtained from truncating the extended version.

The extended Kasami code with parameters (s, t) is defined as

$$\{v \in \mathbf{F}_2^s \mid \sum_{a \in \mathbf{F}_s} v_a = \sum_{a \in \mathbf{F}_s} a v_a = \sum_{a \in \mathbf{F}_s} a^{t+1} v_a = 0\}$$

It follows that these are subfield subcodes of the code having those three equations as parity checks. The only valid parameters s, t are given by the below, where q is a power of 2

- $s = q^{2j+1}, t = q^m$ with $m \leq j$ and $\gcd(m, 2j+1) = 1$
- $s = q^2, t = q$

The coset graphs of the Kasami codes are distance-regular. In particular, the extended Kasami codes result in distance-regular graphs with intersection arrays

- $[q^{2j+1}, q^{2j+1} - 1, q^{2j+1} - q, q^{2j+1} - q^{2j} + 1; 1, q, q^{2j} - 1, q^{2j+1}]$
- $[q^2, q^2 - 1, q^2 - q, 1; 1, q, q^2 - 1, q^2]$

The Kasami codes result in distance-regular graphs with intersection arrays

- $[q^{2j+1} - 1, q^{2j+1} - q, q^{2j+1} - q^{2j} + 1; 1, q, q^{2j} - 1]$
- $[q^2 - 1, q^2 - q, 1; 1, q, q^2 - 1]$

REFERENCES:

- [BCN1989] p. 358 for a definition.
- [Kas1966a]
- [Kas1966b]
- [Kas1971]

AUTHORS:

- Ivo Maffei (2020-07-09): initial version

class sage.coding.kasami_codes.**KasamiCode**(*s*, *t*, *extended=True*)

Bases: *AbstractLinearCode*

Representation of a Kasami Code.

The extended Kasami code with parameters (s, t) is defined as

$$\{v \in \mathbf{F}_2^s \mid \sum_{a \in \mathbf{F}_s} v_a = \sum_{a \in \mathbf{F}_s} a v_a = \sum_{a \in \mathbf{F}_s} a^{t+1} v_a = 0\}$$

The only valid parameters s, t are given by the below, where q is a power of 2:

- $s = q^{2j+1}$, $t = q^m$ with $m \leq j$ and $\gcd(m, 2j+1) = 1$
- $s = q^2$, $t = q$

The Kasami code (s, t) is obtained from the extended Kasami code (s, t), via truncation of all words.

INPUT:

- `s,t` – (integer) the parameters of the Kasami code
- `extended` – (default: `True`) if set to `True`, creates an extended Kasami code.

EXAMPLES:

```
sage: codes.KasamiCode(16,4)
[16, 9] Extended (16, 4)-Kasami code
sage: _.minimum_distance()
4

sage: codes.KasamiCode(8, 2, extended=False)
[7, 1] (8, 2)-Kasami code

sage: codes.KasamiCode(8,4)
Traceback (most recent call last):
...
ValueError: The parameters(=8,4) are invalid. Check the documentation
```

The extended Kasami code is the extension of the Kasami code:

```
sage: C = codes.KasamiCode(16, 4, extended=False)
sage: Cext = C.extended_code()
sage: D = codes.KasamiCode(16, 4, extended=True)
sage: D.generator_matrix() == Cext.generator_matrix()
True
```

See also:

sage.coding.linear_code.

REFERENCES:

For more information on Kasami codes and their use see [BCN1989] or [Kas1966a], [Kas1966b], [Kas1971]

generator_matrix()

Return a generator matrix of `self`.

EXAMPLES:

```
sage: C = codes.KasamiCode(16, 4, extended=False)
sage: C.generator_matrix()
[1 0 0 0 0 0 0 0 0 0 1 0 0 1 1 1]
[0 1 0 0 0 0 0 0 0 0 1 1 0 1 0 0]
[0 0 1 0 0 0 0 0 0 0 0 1 1 0 1 0]
```

(continues on next page)

(continued from previous page)

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

ALGORITHM:

We build the parity check matrix given by the three equations that the codewords must satisfy. Then we generate the parity check matrix over $\mathbf{F}_2$ and from this the obtain the generator matrix for the extended Kasami codes.

For the Kasami codes, we truncate the last column.

parameters()

Return the parameters s, t of `self`.

EXAMPLES:

```
sage: C = codes.KasamiCode(16, 4, extended=True)
sage: C.parameters()
(16, 4)
sage: D = codes.KasamiCode(16, 4, extended=False)
sage: D.parameters()
(16, 4)
sage: C = codes.KasamiCode(8,2)
sage: C.parameters()
(8, 2)
```

11.10 AG codes

Algebraic geometry codes or shortly AG codes are linear codes defined using functions or differentials on algebraic curves over finite fields. Sage implements evaluation AG codes and differential AG codes as Goppa defined in [Gop1981] and provides decoding algorithms for them in full generality.

EXAMPLES:

```
sage: k.<a> = GF(4)
sage: A.<x,y> = AffineSpace(k, 2)
sage: C = Curve(y^2 + y - x^3)
sage: F = C.function_field()
sage: pls = F.places()
sage: p = C([0,0])
sage: Q, = p.places()
sage: pls.remove(Q)
sage: G = 5*Q
sage: codes.EvaluationAGCode(pls, G)
[8, 5] evaluation AG code over GF(4)
sage: codes.DifferentialAGCode(pls, G)
[8, 3] differential AG code over GF(4)
```

As is well known, the two kinds of AG codes are dual to each other.

```
sage: E = codes.EvaluationAGCode(pls, G)
sage: D = codes.DifferentialAGCode(pls, G)
sage: E.dual_code() == D
True
sage: D.dual_code() == E
True
```

Decoders for both evaluation and differential AG codes are available.

11.10.1 Decoders for AG codes

This module implements decoders for evaluation and differential AG codes.

The implemented algorithm for unique decoding of AG codes, named K, is from [LBO2014] and [Lee2016].

EXAMPLES:

```
sage: F.<a> = GF(9)
sage: A2.<x,y> = AffineSpace(F, 2)
sage: C = Curve(y^3 + y - x^4)
sage: Q, = C.places_at_infinity()
sage: O = C(0,0).place()
sage: pls = C.places()
sage: pls.remove(Q)
sage: pls.remove(O)
sage: G = -O + 18*Q
sage: code = codes.EvaluationAGCode(pls, G)  # long time
sage: code                                   # long time
[26, 15] evaluation AG code over GF(9)
sage: decoder = code.decoder('K')            # long time
sage: tau = decoder.decoding_radius()        # long time
sage: tau                                    # long time
4
```

The `decoder` is now ready for correcting vectors received from a noisy channel:

```
sage: channel = channels.StaticErrorRateChannel(code.ambient_space(), tau)  # long time
sage: message_space = decoder.message_space()                   # long time
sage: message = message_space.random_element()                  # long time
sage: encoder = decoder.connected_encoder()                     # long time
sage: sent_codeword = encoder.encode(message)                   # long time
sage: received_vector = channel(sent_codeword)                  # long time
sage: (received_vector - sent_codeword).hamming_weight()        # long time
4
sage: decoder.decode_to_code(received_vector) == sent_codeword  # long time
True
sage: decoder.decode_to_message(received_vector) == message     # long time
True
```

AUTHORS:

- Kwankyu Lee (2019-03): initial version

class `sage.coding.ag_code_decoders.Decoder_K`

Bases: `object`

Common base class for the implementation of decoding algorithm K for AG codes.

EXAMPLES:

```
sage: F.<a> = GF(4)
sage: P.<x,y> = AffineSpace(F, 2);
sage: C = Curve(y^2 + y - x^3)
sage: pls = C.places()
sage: p = C([0,0])
sage: Q, = p.places()
sage: D = [pl for pl in pls if pl != Q]
sage: G = 5*Q
sage: from sage.coding.ag_code_decoders import EvaluationAGCodeDecoder_K
sage: circuit = EvaluationAGCodeDecoder_K(D, G, Q)
```

decode(*received_vector*, *verbose=False*, *detect_decoding_failure=True*, *detect_Q_polynomial=True*)

Return the message vector that corresponds to the corrected codeword from the received vector.

INPUT:

- `received_vector` – a received vector in the ambient space of the code
- `verbose` – boolean; if `True`, verbose information is printed
- `detect_decoding_failure` – boolean; if `True`, early failure detection is activated
- `detect_Q_polynomial` – boolean; if `True`, a Q-polynomial is detected for fast decoding

If decoding fails for some reason, `DecodingError` is raised. The message contained in the exception indicates the type of the decoding failure.

encode(*message*)

Encode `message` to a codeword.

info

class `sage.coding.ag_code_decoders.`**Decoder_K_extension**

Bases: `object`

Common base class for decoding algorithm K for AG codes via constant field extension.

INPUT:

- `pls` – a list of places of a function field
- `G` – a divisor of the function field
- `Q` – a non-rational place
- `verbose` – if `True`, verbose information is printed

EXAMPLES:

```
sage: A.<x,y> = AffineSpace(GF(4), 2)
sage: C = Curve(y^2 + y - x^3)
sage: pls = C.places()
sage: F = C.function_field()
sage: G = 1*F.get_place(4)
sage: code = codes.EvaluationAGCode(pls, G)
sage: dec = code.decoder('K'); dec  # long time
Unique decoder for [9, 4] evaluation AG code over GF(4)
```

```
sage: P.<x,y> = ProjectiveSpace(GF(4), 1)
sage: C = Curve(P)
sage: pls = C.places()
sage: len(pls)
5
sage: F = C.function_field()
sage: G = F.get_place(2).divisor()
sage: code = codes.EvaluationAGCode(pls, G)
sage: code.decoder('K')
Unique decoder for [5, 3] evaluation AG code over GF(4)
```

decode(*received_vector*, ***kwargs*)

Decode the received vector to a message.

INPUT:

- `received_vector` – a vector in the ambient space of the code

encode(*message*, ***kwargs*)

Encode `message` to a codeword.

info

class sage.coding.ag_code_decoders.**DifferentialAGCodeDecoder_K**

Bases: *Decoder_K*

This class implements the decoding algorithm K for differential AG codes.

INPUT:

- `pls` – a list of places of a function field
- `G` – a divisor of the function field
- `Q` – a rational place not in `pls`
- `verbose` – if `True`, verbose information is printed

EXAMPLES:

```
sage: F.<a> = GF(4)
sage: P.<x,y> = AffineSpace(F, 2);
sage: C = Curve(y^2 + y - x^3)
sage: pls = C.places()
sage: p = C([0,0])
sage: Q, = p.places()
sage: D = [pl for pl in pls if pl != Q]
sage: G = 5*Q
sage: from sage.coding.ag_code_decoders import DifferentialAGCodeDecoder_K
sage: circuit = DifferentialAGCodeDecoder_K(D, G, Q)  # long time
sage: rv = vector([1, a, 1, a, 1, a, a, a + 1])
sage: cw = circuit.encode(circuit.decode(rv))  # long time
sage: rv - cw  # long time
(0, 0, 0, a + 1, 1, 0, 0, 0)
sage: circuit.info['designed_distance']  # long time
5
sage: circuit.info['decoding_radius']  # long time
2
```

class sage.coding.ag_code_decoders.**DifferentialAGCodeDecoder_K_extension**

Bases: *Decoder_K_extension*

This class implements the decoding algorithm K for differential AG codes via constant field extension.

INPUT:

- pls – a list of places of a function field
- G – a divisor of the function field
- Q – a non-rational place
- verbose – if True, verbose information is printed

EXAMPLES:

```
sage: F.<a> = GF(4)
sage: A.<x,y> = AffineSpace(F, 2)
sage: C = Curve(y^2 + y - x^3)
sage: pls = C.places()
sage: F = C.function_field()
sage: G = 1*F.get_place(4)
sage: code = codes.DifferentialAGCode(pls, G)
sage: Q = F.get_place(3)
sage: from sage.coding.ag_code_decoders import DifferentialAGCodeDecoder_K_extension
sage: circuit = DifferentialAGCodeDecoder_K_extension(pls, G, Q)  # long time
sage: cw = code.random_element()
sage: rv = cw + vector([0,0,a,0,0,0,0,0,0])
sage: circuit.encode(circuit.decode(circuit._lift(rv))) == circuit._lift(cw)  #␣
↪long time
True
```

class sage.coding.ag_code_decoders.**DifferentialAGCodeEncoder**(*code*, *decoder=None*)

Bases: *Encoder*

Encoder of a differential AG code.

INPUT:

- code – a differential AG code
- decoder – a decoder of the code

EXAMPLES:

```
sage: F.<a> = GF(4)
sage: P.<x,y> = AffineSpace(F, 2);
sage: C = Curve(y^2 + y - x^3)
sage: F = C.function_field()
sage: pls = F.places()
sage: p = C([0,0])
sage: Q, = p.places()
sage: D = [pl for pl in pls if pl != Q]
sage: G = 5*Q
sage: code = codes.DifferentialAGCode(D, G)
sage: dec = code.decoder('K', Q)  # long time
sage: enc = dec.connected_encoder(); enc  # long time
Encoder for [8, 3] differential AG code over GF(4)
```

encode(*message*)

Return the codeword encoded from the message.

INPUT:

- message – a vector in the message space

EXAMPLES:

```
sage: F.<a> = GF(4)
sage: P.<x,y> = AffineSpace(F, 2)
sage: C = Curve(y^2 + y - x^3)
sage: F = C.function_field()
sage: pls = F.places()
sage: p = C([0,0])
sage: Q, = p.places()
sage: D = [pl for pl in pls if pl != Q]
sage: G = 5*Q
sage: code = codes.DifferentialAGCode(D, G)      # long time
sage: dec = code.decoder('K', Q)                 # long time
sage: enc = dec.connected_encoder()              # long time
sage: msg = enc.message_space().random_element() # long time
sage: codeword = enc.encode(msg)                 # long time
sage: enc.unencode(codeword) == msg              # long time
True
```

unencode_nocheck(*codeword*)

Return the message unencoded from codeword.

INPUT:

- codeword – a vector in the code

EXAMPLES:

```
sage: F.<a> = GF(4)
sage: P.<x,y> = AffineSpace(F, 2)
sage: C = Curve(y^2 + y - x^3)
sage: F = C.function_field()
sage: pls = F.places()
sage: p = C([0,0])
sage: Q, = p.places()
sage: D = [pl for pl in pls if pl != Q]
sage: G = 5*Q
sage: code = codes.DifferentialAGCode(D, G)      # long time
sage: dec = code.decoder('K', Q)                 # long time
sage: enc = dec.connected_encoder()              # long time
sage: msg = enc.message_space().random_element() # long time
sage: codeword = enc.encode(msg)                 # long time
sage: enc.unencode(codeword) in enc.message_space()   # indirect doctest, long␣
↪time
True
```

class sage.coding.ag_code_decoders.**DifferentialAGCodeUniqueDecoder**(*code*, *Q=None*, *basis=None*, *verbose=False*)

Bases: *Decoder*

Unique decoder for a differential AG codes.

INPUT:

- code – an evaluation AG code
- Q – (optional) a place, not one of the places supporting the code
- basis – (optional) a basis of the space of differentials to take residues
- verbose – if True, verbose information is printed

EXAMPLES:

```
sage: F.<a> = GF(4)
sage: P.<x,y> = AffineSpace(F, 2);
sage: C = Curve(y^2 + y - x^3)
sage: F = C.function_field()
sage: pls = F.places()
sage: p = C(0,0)
sage: Q, = p.places()
sage: D = [pl for pl in pls if pl != Q]
sage: G = 5*Q
sage: code = codes.DifferentialAGCode(D, G)
sage: chan = channels.StaticErrorRateChannel(code.ambient_space(), 2)
sage: rv = chan.transmit(code.random_element())  # long time
sage: dec = code.decoder('K', Q)  # long time
sage: enc = dec.connected_encoder()  # long time
sage: enc.encode(dec.decode_to_message(rv)) in code  # long time
True
```

If basis is given, that defines the associated residue encoding map:

```
sage: basis = tuple((G - sum(D)).basis_differential_space())
sage: w = basis[0]
sage: cw = vector(w.residue(p) for p in D)
sage: dec2 = code.decoder('K', Q, basis)  # long time
sage: enc2 = dec2.connected_encoder()  # long time
sage: temp = enc2.unencode(cw); temp  # long time
(1, 0, 0)
sage: enc2.encode(temp) == cw  # long time
True
sage: w = basis[1]
sage: cw = vector(w.residue(p) for p in D)
sage: temp = enc2.unencode(cw); temp  # long time
(0, 1, 0)
sage: enc2.encode(temp) == cw  # long time
True
```

The default basis is given by code.basis_differentials().

connected_encoder(**args*, ***kwargs*)

Return the connected encoder for this decoder.

INPUT:

- args, kwargs – all additional arguments are forwarded to the constructor of the connected encoder

EXAMPLES:

```
sage: F.<a> = GF(4)
sage: P.<x,y> = AffineSpace(F, 2);
sage: C = Curve(y^2 + y - x^3)
sage: F = C.function_field()
sage: pls = F.places()
sage: p = C([0,0])
sage: Q, = p.places()
sage: D = [pl for pl in pls if pl != Q]
sage: G = 5*Q
sage: code = codes.DifferentialAGCode(D, G)   # long time
sage: dec = code.decoder('K', Q)              # long time
sage: dec.connected_encoder()                 # long time
Encoder for [8, 3] differential AG code over GF(4)
```

decode_to_code(*received_vector*, ***kwargs*)

Return the codeword decoded from `received_vector`.

INPUT:

- `received_vector` – a vector in the ambient space of the code
- `verbose` – boolean; if `True`, verbose information on the decoding process is printed

EXAMPLES:

```
sage: F.<a> = GF(4)
sage: P.<x,y> = AffineSpace(F, 2);
sage: C = Curve(y^2 + y - x^3)
sage: F = C.function_field()
sage: pls = F.places()
sage: p = C([0,0])
sage: Q, = p.places()
sage: D = [pl for pl in pls if pl != Q]
sage: G = 5*Q
sage: code = codes.DifferentialAGCode(D, G)   # long time
sage: dec = code.decoder('K', Q)              # long time
sage: enc = dec.connected_encoder()           # long time
sage: code = dec.code()                       # long time
sage: chan = channels.StaticErrorRateChannel(code.ambient_space(), 2) # long␣
↪time
sage: rv = chan.transmit(code.random_element())  # long time
sage: cw = dec.decode_to_code(rv)                # long time
sage: (cw - rv).hamming_weight() == 2            # long time
True
```

decode_to_message(*received_vector*, ***kwargs*)

Return the message decoded from `received_vector`.

INPUT:

- `received_vector` – a vector in the ambient space of the code
- `verbose` – boolean; if `True`, verbose information on the decoding process is printed

EXAMPLES:

```
sage: F.<a> = GF(4)
sage: P.<x,y> = AffineSpace(F, 2);
sage: C = Curve(y^2 + y - x^3)
sage: F = C.function_field()
sage: pls = F.places()
sage: p = C([0,0])
sage: Q, = p.places()
sage: D = [pl for pl in pls if pl != Q]
sage: G = 5*Q
sage: code = codes.DifferentialAGCode(D, G)  # long time
sage: dec = code.decoder('K', Q)             # long time
sage: enc = dec.connected_encoder()          # long time
sage: code = dec.code()                      # long time
sage: chan = channels.StaticErrorRateChannel(code.ambient_space(), 2)  # long␣
↪time
sage: rv = chan.transmit(code.random_element())  # long time
sage: msg = dec.decode_to_message(rv)            # long time
sage: cw = enc.encode(msg)                       # long time
sage: (cw - rv).hamming_weight() == 2            # long time
True
```

decoding_radius()

Return the decoding radius of the decoder.

EXAMPLES:

```
sage: F.<a> = GF(4)
sage: P.<x,y> = AffineSpace(F, 2);
sage: C = Curve(y^2 + y - x^3)
sage: F = C.function_field()
sage: pls = F.places()
sage: p = C([0,0])
sage: Q, = p.places()
sage: D = [pl for pl in pls if pl != Q]
sage: G = 5*Q
sage: code = codes.DifferentialAGCode(D, G)  # long time
sage: dec = code.decoder('K', Q)             # long time
sage: dec.decoding_radius()                  # long time
2
```

class sage.coding.ag_code_decoders.**EvaluationAGCodeDecoder_K**

Bases: *Decoder_K*

This class implements the decoding algorithm K for evaluation AG codes.

INPUT:

- `pls` – a list of places of a function field
- `G` – a divisor of the function field
- `Q` – a rational place not in `pls`
- `verbose` – if `True`, verbose information is printed.

EXAMPLES:

```
sage: F.<a> = GF(4)
sage: P.<x,y> = AffineSpace(F, 2);
sage: C = Curve(y^2 + y - x^3)
sage: pls = C.places()
sage: p = C([0,0])
sage: Q, = p.places()
sage: D = [pl for pl in pls if pl != Q]
sage: G = 5*Q
sage: from sage.coding.ag_code_decoders import EvaluationAGCodeDecoder_K
sage: circuit = EvaluationAGCodeDecoder_K(D, G, Q)
sage: rv = vector([a, 0, 0, a, 1, 1, a + 1, 0])
sage: cw = circuit.encode(circuit.decode(rv))
sage: rv - cw
(a + 1, 0, 0, 0, 0, 0, 0, 0)
sage: circuit.info['designed_distance']
3
sage: circuit.info['decoding_radius']
1
```

class sage.coding.ag_code_decoders.**EvaluationAGCodeDecoder_K_extension**

Bases: *Decoder_K_extension*

This class implements the decoding algorithm K for evaluation AG codes via constant field extension.

INPUT:

- `pls` – a list of places of a function field
- `G` – a divisor of the function field
- `Q` – a non-rational place
- `verbose` – if `True`, verbose information is printed

EXAMPLES:

```
sage: F.<a> = GF(4)
sage: A.<x,y> = AffineSpace(F, 2)
sage: C = Curve(y^2 + y - x^3)
sage: pls = C.places()
sage: F = C.function_field()
sage: G = 1*F.get_place(4)
sage: code = codes.EvaluationAGCode(pls, G)
sage: Q = F.get_place(3)
sage: from sage.coding.ag_code_decoders import EvaluationAGCodeDecoder_K_extension
sage: circuit = EvaluationAGCodeDecoder_K_extension(pls, G, Q)
sage: cw = code.random_element()
sage: rv = cw + vector([0,1,1,0,0,0,0,0,0])
sage: circuit.encode(circuit.decode(circuit._lift(rv))) == circuit._lift(cw)
True
```

class sage.coding.ag_code_decoders.**EvaluationAGCodeEncoder**(*code*, *decoder=None*)

Bases: *Encoder*

Encoder of an evaluation AG code

INPUT:

- `code` – an evaluation AG code
- `decoder` – a decoder of the code

EXAMPLES:

```
sage: F.<a> = GF(4)
sage: P.<x,y> = AffineSpace(F, 2);
sage: C = Curve(y^2 + y - x^3)
sage: F = C.function_field()
sage: pls = F.places()
sage: p = C([0,0])
sage: Q, = p.places()
sage: D = [pl for pl in pls if pl != Q]
sage: G = 5*Q
sage: code = codes.EvaluationAGCode(D, G)
sage: dec = code.decoder('K', Q)
sage: enc = dec.connected_encoder()
sage: enc
Encoder for [8, 5] evaluation AG code over GF(4)
```

encode(*message*)

Return the codeword encoded from the message.

INPUT:

- `message` – a vector in the message space

EXAMPLES:

```
sage: F.<a> = GF(4)
sage: P.<x,y> = AffineSpace(F, 2)
sage: C = Curve(y^2 + y - x^3)
sage: F = C.function_field()
sage: pls = F.places()
sage: p = C([0,0])
sage: Q, = p.places()
sage: D = [pl for pl in pls if pl != Q]
sage: G = 5*Q
sage: code = codes.EvaluationAGCode(D, G)         # long time
sage: dec = code.decoder('K', Q)                  # long time
sage: enc = dec.connected_encoder()               # long time
sage: msg = enc.message_space().random_element() # long time
sage: codeword = enc.encode(msg)                  # long time
sage: enc.unencode(codeword) == msg               # long time
True
```

unencode_nocheck(*codeword*)

Return the message unencoded from `codeword`.

INPUT:

- `codeword` – a vector in the code

EXAMPLES:

```
sage: F.<a> = GF(4)
sage: P.<x,y> = AffineSpace(F, 2)
sage: C = Curve(y^2 + y - x^3)
sage: F = C.function_field()
sage: pls = F.places()
sage: p = C([0,0])
sage: Q, = p.places()
sage: D = [pl for pl in pls if pl != Q]
sage: G = 5*Q
sage: code = codes.EvaluationAGCode(D, G)         # long time
sage: dec = code.decoder('K', Q)                  # long time
sage: enc = dec.connected_encoder()               # long time
sage: msg = enc.message_space().random_element() # long time
sage: codeword = enc.encode(msg)                  # long time
sage: enc.unencode(codeword) in enc.message_space()  # long time, indirect␣
↪doctest
True
```

class sage.coding.ag_code_decoders.**EvaluationAGCodeUniqueDecoder**(*code*, *Q=None*, *basis=None*, *verbose=False*)

Bases: `Decoder`

Unique decoder for evaluation AG codes.

INPUT:

- `code` – an evaluation AG code
- `Q` – (optional) a place, not one of the places supporting the code
- `basis` – (optional) a basis of the space of functions to evaluate
- `verbose` – if `True`, verbose information is printed

EXAMPLES:

```
sage: k.<a> = GF(4)
sage: P.<x,y> = AffineSpace(k, 2);
sage: C = Curve(y^2 + y - x^3)
sage: F = C.function_field()
sage: pls = F.places()
sage: p = C(0,0)
sage: Q, = p.places()
sage: D = [pl for pl in pls if pl != Q]
sage: G = 5*Q
sage: code = codes.EvaluationAGCode(D, G)
sage: dec = code.decoder('K', Q)
sage: enc = dec.connected_encoder()
sage: chan = channels.StaticErrorRateChannel(code.ambient_space(), 1)
sage: rv = chan.transmit(code.random_element())
sage: enc.encode(dec.decode_to_message(rv)) in code
True
```

If `basis` is given, that defines the associated evaluation encoding map:

```
sage: basis = tuple(G.basis_function_space())
sage: dec2 = code.decoder('K', Q, basis)
sage: enc2 = dec2.connected_encoder()
sage: f = basis[0]
sage: cw = vector(f.evaluate(p) for p in D)
sage: enc2.unencode(cw)
(1, 0, 0, 0, 0)
sage: enc2.encode(_) == cw
True
sage: f = basis[1]
sage: cw = vector(f.evaluate(p) for p in D)
sage: enc2.unencode(cw)
(0, 1, 0, 0, 0)
sage: enc2.encode(_) == cw
True
```

The default basis is given by `code.basis_functions()`.

connected_encoder(**args*, ***kwargs*)

Return the connected encoder for this decoder.

INPUT:

- `args`, `kwargs` – all additional arguments are forwarded to the constructor of the connected encoder

EXAMPLES:

```
sage: F.<a> = GF(4)
sage: P.<x,y> = AffineSpace(F, 2)
sage: C = Curve(y^2 + y - x^3)
sage: F = C.function_field()
sage: pls = F.places()
sage: p = C([0,0])
sage: Q, = p.places()
sage: D = [pl for pl in pls if pl != Q]
sage: G = 5*Q
sage: code = codes.EvaluationAGCode(D, G)  # long time
sage: dec = code.decoder('K', Q)           # long time
sage: dec.connected_encoder()              # long time
Encoder for [8, 5] evaluation AG code over GF(4)
```

decode_to_code(*received_vector*, ***kwargs*)

Return the codeword decoded from `received_vector`.

INPUT:

- `received_vector` – a vector in the ambient space of the code
- `verbose` – boolean; if `True`, verbose information on the decoding process is printed

EXAMPLES:

```
sage: F.<a> = GF(4)
sage: P.<x,y> = AffineSpace(F, 2)
sage: C = Curve(y^2 + y - x^3)
sage: F = C.function_field()
```

(continues on next page)

(continued from previous page)

```
sage: pls = F.places()
sage: p = C([0,0])
sage: Q, = p.places()
sage: D = [pl for pl in pls if pl != Q]
sage: G = 5*Q
sage: code = codes.EvaluationAGCode(D, G)  # long time
sage: dec = code.decoder('K', Q)           # long time
sage: code = dec.code()                    # long time
sage: chan = channels.StaticErrorRateChannel(code.ambient_space(), 1)  # long␣
↪time
sage: rv = chan.transmit(code.random_element())  # long time
sage: cw = dec.decode_to_code(rv)                # long time
sage: (cw - rv).hamming_weight() == 1            # long time
True
```

decode_to_message(*received_vector*, ***kwargs*)

Return the message decoded from `received_vector`.

INPUT:

- `received_vector` – a vector in the ambient space of the code
- `verbose` – boolean; if `True`, verbose information on the decoding process is printed

EXAMPLES:

```
sage: F.<a> = GF(4)
sage: P.<x,y> = AffineSpace(F, 2)
sage: C = Curve(y^2 + y - x^3)
sage: F = C.function_field()
sage: pls = F.places()
sage: p = C([0,0])
sage: Q, = p.places()
sage: D = [pl for pl in pls if pl != Q]
sage: G = 5*Q
sage: code = codes.EvaluationAGCode(D, G)  # long time
sage: dec = code.decoder('K', Q)           # long time
sage: enc = dec.connected_encoder()        # long time
sage: code = dec.code()                    # long time
sage: chan = channels.StaticErrorRateChannel(code.ambient_space(), 1)  # long␣
↪time
sage: rv = chan.transmit(code.random_element())  # long time
sage: msg = dec.decode_to_message(rv)            # long time
sage: cw = enc.encode(msg)                       # long time
sage: (cw - rv).hamming_weight() == 1            # long time
True
```

decoding_radius()

Return the decoding radius of the decoder.

EXAMPLES:

```
sage: F.<a> = GF(4)
sage: P.<x,y> = AffineSpace(F, 2)
```

(continues on next page)

(continued from previous page)

```
sage: C = Curve(y^2 + y - x^3)
sage: F = C.function_field()
sage: pls = F.places()
sage: p = C([0,0])
sage: Q, = p.places()
sage: D = [pl for pl in pls if pl != Q]
sage: G = 5*Q
sage: code = codes.EvaluationAGCode(D, G)  # long time
sage: dec = code.decoder('K', Q)           # long time
sage: dec.decoding_radius()                # long time
1
```

A natural generalization of classical Goppa codes is Cartier codes [Cou2014]. Cartier codes are subfield subcodes of differential AG codes.

EXAMPLES:

```
sage: F.<a> = GF(9)
sage: P.<x,y,z> = ProjectiveSpace(F, 2);
sage: C = Curve(x^3*y + y^3*z + x*z^3)
sage: F = C.function_field()
sage: pls = F.places()
sage: Z, = C([0,0,1]).places()
sage: pls.remove(Z)
sage: G = 3*Z
sage: codes.CartierCode(pls, G)  # long time
[9, 4] Cartier code over GF(3)
```

AUTHORS:

- Kwankyu Lee (2019-03): initial version

class sage.coding.ag_code.**AGCode**(*base_field*, *length*, *default_encoder_name*, *default_decoder_name*)

Bases: `AbstractLinearCode`

Base class of algebraic geometry codes.

A subclass of this class is required to define `_function_field` attribute that refers to an abstract functiom field or the function field of the underlying curve used to construct a code of the class.

base_function_field()

Return the function field used to construct the code.

EXAMPLES:

```
sage: k.<a> = GF(4)
sage: A.<x,y> = AffineSpace(k, 2)
sage: C = Curve(y^2 + y - x^3)
sage: F = C.function_field()
sage: pls = F.places()
sage: p = C([0,0])
sage: Q, = p.places()
sage: pls.remove(Q)
sage: G = 5*Q
sage: code = codes.EvaluationAGCode(pls, G)
```

(continues on next page)

(continued from previous page)

```
sage: code.base_function_field()
Function field in y defined by y^2 + y + x^3
```

class sage.coding.ag_code.**CartierCode**(*pls*, *G*, *r=1*, *name=None*)

Bases: *AGCode*

Cartier code defined by rational places `pls` and a divisor `G` of a function field.

INPUT:

- `pls` – a list of rational places
- `G` – a divisor whose support is disjoint from `pls`
- `r` – integer (default: 1)
- `name` – string; name of the generator of the subfield $\mathbf{F}_{p^r}$

OUTPUT: Cartier code over $\mathbf{F}_{p^r}$ where p is the characteristic of the base constant field of the function field

Note that if `r` is 1 the default, then `name` can be omitted.

EXAMPLES:

```
sage: F.<a> = GF(9)
sage: P.<x,y,z> = ProjectiveSpace(F, 2);
sage: C = Curve(x^3*y + y^3*z + x*z^3)
sage: F = C.function_field()
sage: pls = F.places()
sage: Z, = C(0,0,1).places()
sage: pls.remove(Z)
sage: G = 3*Z
sage: code = codes.CartierCode(pls, G)  # long time
sage: code.minimum_distance()           # long time
2
```

designed_distance()

Return the designed distance of the Cartier code.

The designed distance is that of the differential code of which the Cartier code is a subcode.

EXAMPLES:

```
sage: F.<a> = GF(9)
sage: P.<x,y,z> = ProjectiveSpace(F, 2);
sage: C = Curve(x^3*y + y^3*z + x*z^3)
sage: F = C.function_field()
sage: pls = F.places()
sage: Z, = C(0,0,1).places()
sage: pls.remove(Z)
sage: G = 3*Z
sage: code = codes.CartierCode(pls, G)  # long time
sage: code.designed_distance()          # long time
1
```

generator_matrix()

Return a generator matrix of the Cartier code.

EXAMPLES:

```
sage: F.<a> = GF(9)
sage: P.<x,y,z> = ProjectiveSpace(F, 2);
sage: C = Curve(x^3*y + y^3*z + x*z^3)
sage: F = C.function_field()
sage: pls = F.places()
sage: Z, = C(0,0,1).places()
sage: pls.remove(Z)
sage: G = 3*Z
sage: code = codes.CartierCode(pls, G)  # long time
sage: code.generator_matrix()           # long time
[1 0 0 2 2 0 2 2 0]
[0 1 0 2 2 0 2 2 0]
[0 0 1 0 0 0 0 0 2]
[0 0 0 0 0 1 0 0 2]
```

class sage.coding.ag_code.**DifferentialAGCode**(*pls*, *G*)

Bases: *AGCode*

Differential AG code defined by rational places `pls` and a divisor `G`

INPUT:

- `pls` – a list of rational places of a function field
- `G` – a divisor whose support is disjoint from `pls`

If `G` is a place, then it is regarded as a prime divisor.

EXAMPLES:

```
sage: k.<a> = GF(4)
sage: A.<x,y> = AffineSpace(k, 2)
sage: C = A.curve(y^3 + y - x^4)
sage: Q = C.places_at_infinity()[0]
sage: O = C([0,0]).place()
sage: pls = [p for p in C.places() if p not in [O, Q]]
sage: G = -O + 3*Q
sage: codes.DifferentialAGCode(pls, -O + Q)
[3, 2] differential AG code over GF(4)

sage: F = C.function_field()
sage: G = F.get_place(1)
sage: codes.DifferentialAGCode(pls, G)
[3, 1] differential AG code over GF(4)
```

basis_differentials()

Return the basis differentials associated with the generator matrix.

EXAMPLES:

```
sage: k.<a> = GF(4)
sage: A.<x,y> = AffineSpace(k, 2)
sage: C = Curve(y^2 + y - x^3)
sage: F = C.function_field()
sage: pls = F.places()
sage: Q, = C.places_at_infinity()
```

(continues on next page)

(continued from previous page)

```
sage: pls.remove(Q)
sage: code = codes.DifferentialAGCode(pls, 3*Q)
sage: matrix([[w.residue(p) for p in pls]
....:         for w in code.basis_differentials()])
[    1     0     0     0     0 a + 1 a + 1     1]
[    0     1     0     0     0 a + 1     a     0]
[    0     0     1     0     0     a     1     a]
[    0     0     0     1     0     a     0 a + 1]
[    0     0     0     0     1     1     1     1]
```

designed_distance()

Return the designed distance of the differential AG code.

If the code is of dimension zero, then a `ValueError` is raised.

EXAMPLES:

```
sage: k.<a> = GF(4)
sage: A.<x,y> = AffineSpace(k, 2)
sage: C = Curve(y^2 + y - x^3)
sage: F = C.function_field()
sage: pls = F.places()
sage: Q, = C.places_at_infinity()
sage: pls.remove(Q)
sage: code = codes.DifferentialAGCode(pls, 3*Q)
sage: code.designed_distance()
3
```

generator_matrix()

Return a generator matrix of the code.

EXAMPLES:

```
sage: k.<a> = GF(4)
sage: A.<x,y> = AffineSpace(k, 2)
sage: C = Curve(y^2 + y - x^3)
sage: F = C.function_field()
sage: pls = F.places()
sage: Q, = C.places_at_infinity()
sage: pls.remove(Q)
sage: code = codes.DifferentialAGCode(pls, 3*Q)
sage: code.generator_matrix()
[    1     0     0     0     0 a + 1 a + 1     1]
[    0     1     0     0     0 a + 1     a     0]
[    0     0     1     0     0     a     1     a]
[    0     0     0     1     0     a     0 a + 1]
[    0     0     0     0     1     1     1     1]
```

class sage.coding.ag_code.**EvaluationAGCode**(*pls*, *G*)

Bases: *AGCode*

Evaluation AG code defined by rational places `pls` and a divisor `G`.

INPUT:

- `pls` – a list of rational places of a function field

- G – a divisor whose support is disjoint from pls

If G is a place, then it is regarded as a prime divisor.

EXAMPLES:

```
sage: k.<a> = GF(4)
sage: A.<x,y> = AffineSpace(k, 2)
sage: C = Curve(y^2 + y - x^3)
sage: F = C.function_field()
sage: pls = F.places()
sage: Q, = C.places_at_infinity()
sage: pls.remove(Q)
sage: G = 5*Q
sage: codes.EvaluationAGCode(pls, G)
[8, 5] evaluation AG code over GF(4)

sage: G = F.get_place(5)
sage: codes.EvaluationAGCode(pls, G)
[8, 5] evaluation AG code over GF(4)
```

basis_functions()

Return the basis functions associated with the generator matrix.

EXAMPLES:

```
sage: k.<a> = GF(4)
sage: A.<x,y> = AffineSpace(k, 2)
sage: C = Curve(y^2 + y - x^3)
sage: F = C.function_field()
sage: pls = F.places()
sage: Q, = C.places_at_infinity()
sage: pls.remove(Q)
sage: code = codes.EvaluationAGCode(pls, 3*Q)
sage: code.basis_functions()
(y + a*x + 1, y + x, (a + 1)*x)
sage: matrix([[f.evaluate(p) for p in pls] for f in code.basis_functions()])
[    1     0     0     1     a a + 1     1     0]
[    0     1     0     1     1     0 a + 1     a]
[    0     0     1     1     a     a a + 1 a + 1]
```

designed_distance()

Return the designed distance of the AG code.

If the code is of dimension zero, then a ValueError is raised.

EXAMPLES:

```
sage: k.<a> = GF(4)
sage: A.<x,y> = AffineSpace(k, 2)
sage: C = Curve(y^2 + y - x^3)
sage: F = C.function_field()
sage: pls = F.places()
sage: Q, = C.places_at_infinity()
sage: pls.remove(Q)
sage: code = codes.EvaluationAGCode(pls, 3*Q)
```

(continues on next page)

(continued from previous page)

```
sage: code.designed_distance()
5
```

generator_matrix()

Return a generator matrix of the code.

EXAMPLES:

```
sage: k.<a> = GF(4)
sage: A.<x,y> = AffineSpace(k, 2)
sage: C = Curve(y^2 + y - x^3)
sage: F = C.function_field()
sage: pls = F.places()
sage: Q, = C.places_at_infinity()
sage: pls.remove(Q)
sage: code = codes.EvaluationAGCode(pls, 3*Q)
sage: code.generator_matrix()
[    1     0     0     1     a a + 1     1     0]
[    0     1     0     1     1     0 a + 1     a]
[    0     0     1     1     a     a a + 1 a + 1]
```

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

11.11 Linear code constructors that do not preserve the structural information

This file contains a variety of constructions which builds the generator matrix of special (or random) linear codes and wraps them in a *sage.coding.linear_code.LinearCode* object. These constructions are therefore not rich objects such as *sage.coding.grs_code.GeneralizedReedSolomonCode*.

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

```
sage: codes.random_linear_code(GF(2), 5, 2)
[5, 2]  linear code over GF(2)
```

REFERENCES:

- [HP2003]

AUTHORS:

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

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)
([11, 5] Cyclic Code over GF(3),
 [11, 5] Cyclic Code over GF(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)
([11, 6] Cyclic Code over GF(3),
 [11, 6] Cyclic Code over GF(3))
```

This is consistent with Theorem 6.1.3 in [HP2003].

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 [HP2003], Section 6.6.3-6.6.4.)

INPUT:

- `n` – an odd prime
- `F` – a finite prime field 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
Extension of [7, 4] Cyclic Code over GF(2)
sage: C2 == C3
True
sage: C = codes.ExtendedQuadraticResidueCode(17, GF(2))
sage: C
Extension of [17, 9] Cyclic Code over GF(2)
sage: C3 = codes.QuadraticResidueCodeOddPair(7, GF(2))[0]
sage: C3x = C3.extended_code()
sage: C4 = codes.ExtendedQuadraticResidueCode(7, GF(2))
sage: C3x == C4
True
```

AUTHORS:

- David Joyner (07-2006)

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 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
[7, 4] Cyclic Code over GF(2)
sage: C = codes.QuadraticResidueCode(17, GF(2))
sage: C
[17, 9] Cyclic Code over GF(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 [HP2003]) a QR code exists over $\mathbf{F}_q$ 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))  # known bug (#25896)
([17, 8] Cyclic Code over GF(13),
 [17, 8] Cyclic Code over GF(13))
sage: codes.QuadraticResidueCodeEvenPair(17, GF(2))
([17, 8] Cyclic Code over GF(2),
 [17, 8] Cyclic Code over GF(2))
sage: codes.QuadraticResidueCodeEvenPair(13, GF(9,"z"))  # known bug (#25896)
([13, 6] Cyclic Code over GF(9),
 [13, 6] Cyclic Code over GF(9))
sage: C1,C2 = codes.QuadraticResidueCodeEvenPair(7, GF(2))
sage: C1.is_self_orthogonal()
True
sage: C2.is_self_orthogonal()
True
sage: C3 = codes.QuadraticResidueCodeOddPair(17, GF(2))[0]
sage: C4 = codes.QuadraticResidueCodeEvenPair(17, GF(2))[1]
sage: C3.systematic_generator_matrix() == C4.dual_code().systematic_generator_
↪matrix()
True
```

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

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

Quadratic residue codes of a given odd prime length and base ring either don't exist at all or occur as 4-tuples - a pair of "odd-like" codes and a pair of "even-like" codes. If n 2 is prime then (Theorem 6.6.2 in [HP2003]) a QR code exists over $\mathbf{F}_q iff`q$ 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))  # known bug (#25896)
([17, 9] Cyclic Code over GF(13),
 [17, 9] Cyclic Code over GF(13))
sage: codes.QuadraticResidueCodeOddPair(17, GF(2))
([17, 9] Cyclic Code over GF(2),
 [17, 9] Cyclic Code over GF(2))
sage: codes.QuadraticResidueCodeOddPair(13, GF(9,"z"))  # known bug (#25896)
([13, 7] Cyclic Code over GF(9),
 [13, 7] Cyclic Code over GF(9))
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()
```

(continues on next page)

(continued from previous page)

```
[1, 0, 0, 0, 0, 0, 102, 0, 153, 0, 153, 0, 102, 0, 0, 0, 0, 0, 1]
[1, 0, 0, 0, 0, 0, 102, 0, 153, 0, 153, 0, 102, 0, 0, 0, 0, 0, 1]
sage: C3 = codes.QuadraticResidueCodeOddPair(7, GF(2))[0]
sage: C3x = C3.extended_code()
sage: C3x.spectrum()
[1, 0, 0, 0, 14, 0, 0, 0, 1]
```

This is consistent with Theorem 6.6.14 in [HP2003].

`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 $\mathrm{eval}_T(f(x)) = (f(t_1), ..., f(t_n))$, and where $T = \{t_1, ..., t_n\}$. This function returns the toric codes discussed in [Joy2004].

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
[36, 5] linear code over GF(7)
sage: C.minimum_distance()
24
sage: C.minimum_distance(algorithm="guava")  # optional - gap_package_guava
...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
[16, 9] linear code over GF(5)
sage: C.minimum_distance()
6
sage: C.minimum_distance(algorithm="guava")  # optional - gap_package_guava
6
sage: C = codes.ToricCode([[0,0],[1,1],[1,2],[1,3],[1,4],[2,1],
....:                      [2,2],[2,3],[3,1],[3,2],[4,1]], GF(8,"a"))
sage: C
[49, 11] linear code over GF(8)
```

This is in fact a [49,11,28] code over $\mathbf{F}_8$. If you type next `C.minimum_distance()` and wait overnight (!), you should get 28.

AUTHOR:

- David Joyner (07-2006)

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

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

REFERENCES:

- Wikipedia article Hadamard_matrix
- Wikipedia article Walsh_code

`sage.coding.code_constructions.from_parity_check_matrix(H)`

Return the linear code that has `H` as a parity check matrix.

If `H` has dimensions $h \times n$ then the linear code will have dimension $n - h$ and length n.

EXAMPLES:

```
sage: C = codes.HammingCode(GF(2), 3); C
[7, 4] Hamming Code over 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: C2 = codes.from_parity_check_matrix(H); C2
[7, 4] linear code over GF(2)
sage: C2.systematic_generator_matrix() == C.systematic_generator_matrix()
True
```

`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, $\ldots$, n (not 0, 1, $\ldots$, $n - 1$).

EXAMPLES:

```
sage: V = VectorSpace(GF(3),5)
sage: v = V([0,1,2,0,1])
sage: G = SymmetricGroup(5)                                                 #␣
↪optional - sage.groups
sage: g = G([(1,2,3)])                                                      #␣
↪optional - sage.groups
```

(continues on next page)

(continued from previous page)

```
sage: permutation_action(g,v)                                              #␣
↪optional - sage.groups
(1, 2, 0, 0, 1)
sage: g = G([()])                                                          #␣
↪optional - sage.groups
sage: permutation_action(g,v)                                              #␣
↪optional - sage.groups
(0, 1, 2, 0, 1)
sage: g = G([(1,2,3,4,5)])                                                 #␣
↪optional - sage.groups
sage: permutation_action(g,v)                                              #␣
↪optional - sage.groups
(1, 2, 0, 1, 0)
sage: L = Sequence([1,2,3,4,5])
sage: permutation_action(g,L)                                              #␣
↪optional - sage.groups
[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)                                               #␣
↪optional - sage.groups
sage: g = S5([(1,2,3)])                                                    #␣
↪optional - sage.groups
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)                                              #␣
↪optional - sage.groups
[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)                                                #␣
↪optional - sage.groups
sage: g = G([(1,2,3)])                                                     #␣
↪optional - sage.groups
sage: gv = permutation_action(g,v); gv                                     #␣
↪optional - sage.groups
[1, 2, 0, 0, 1]
sage: permutation_action(g,v) == g(v)                                      #␣
↪optional - sage.groups
True
sage: h = G([(3,4)])                                                       #␣
↪optional - sage.groups
sage: gv = permutation_action(g,v)                                         #␣
↪optional - sage.groups
sage: hgv = permutation_action(h,gv)                                       #␣
↪optional - sage.groups
```

(continues on next page)

(continued from previous page)

```
sage: hgv == permutation_action(h*g,v)                                      #␣
↪optional - sage.groups
True
```

AUTHORS:

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

`sage.coding.code_constructions.random_linear_code(F, length, dimension)`

Generate a random linear code of length `length`, dimension `dimension` and over the field `F`.

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

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

EXAMPLES:

```
sage: C = codes.random_linear_code(GF(2), 10, 3)
sage: C
[10, 3] linear code over GF(2)
sage: C.generator_matrix().rank()
3
```

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

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

EXAMPLES:

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

- Wikipedia article Hadamard_matrix

11.12 Constructions of generator matrices using the GUAVA package for GAP

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

AUTHORS:

- David Joyner (2005-11-22, 2006-12-03): initial version
- Nick Alexander (2006-12-10): factor GUAVA code to guava.py
- David Joyner (2007-05): removed Golay codes, toric and trivial codes and placed them in code_constructions; renamed RandomLinearCode to RandomLinearCodeGuava
- David Joyner (2008-03): removed QR, XQR, cyclic and ReedSolomon codes
- David Joyner (2009-05): added "optional package" comments, fixed some docstrings to be sphinx compatible
- Dima Pasechnik (2019-11): port to libgap

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

A (binary) quasi-quadratic residue code (or QQR code).

Follows the definition of Proposition 2.2 in [BM2003]. The code has a generator matrix in the block form $G = (Q, N)$. Here Q is a $p \times p$ circulant matrix whose top row is $(0, x_1, ..., x_{p-1})$, where $x_i = 1$ if and only if i is a quadratic residue mod p, and N is a $p \times p$ circulant matrix whose top row is $(0, y_1, ..., y_{p-1})$, where $x_i + y_i = 1$ for all i.

INPUT:

- `p` – a prime > 2.

OUTPUT:

Returns a QQR code of length $2p$.

EXAMPLES:

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

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_package_
↪guava
[30, 15] linear code over GF(2)
sage: C = codes.RandomLinearCodeGuava(10,5,GF(4,'a')); C   # optional - gap_package_
↪guava
[10, 5] linear code over GF(4)
```

AUTHOR: David Joyner (11-2005)

11.13 Enumerating binary self-dual codes

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

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

- `"code"` – a sd code C of length n, dim $n/2$, over $\mathbf{F}_2$
- `"order autgp"` – order of the permutation automorphism group of C
- `"Type"` – the type of C (which can be `"I"` or `"II"`, in the binary case)
- `"spectrum"` – the spectrum $[A_0, A_1, ..., A_n]$
- `"Comment"` – possibly an empty string.

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

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

```
sage: for n in [4,6,8,10,12,14,16,18,20,22]:                 # not tested
....:    C = self_dual_binary_codes(n); m = len(C.keys())
....:    for i in range(m):
....:        C0 = C["%s"%n]["%s"%i]["code"]
....:        print([n,i,C["%s"%n]["%s"%i]["spectrum"] == C0.spectrum()])
....:        print(C0 == C0.dual_code())
....:        G = C0.automorphism_group_binary_code()
....:        print(C["%s" % n]["%s" % i]["order autgp"] == G.order())
```

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

```
sage: R = PolynomialRing(CC,"T")                             # not tested
sage: T = R.gen()                                            # not tested
sage: for n in [4,6,8,10,12,14,16,18,20,22]:                 # not tested
....:      C = self_dual_binary_codes(n); m = len(C["%s"%n].keys())
....:      for i in range(m):
....:          C0 = C["%s"%n]["%s"%i]["code"]
....:          if C0.minimum_distance()>2:
....:              f = R(C0.sd_zeta_polynomial())
....:              print([n,i,[z[0].abs() for z in f.roots()]])
```

You should get lists of numbers equal to 0.707106781186548.

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

For even m, let A_m denote the $m \times m$ matrix over $\mathbf{F}_2$ 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 matrices M_i 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 [HP2003], page 347).

AUTHORS:

- David Joyner (2007-08-11)

REFERENCES:

- [HP2003] 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.self_dual_codes.`**`self_dual_binary_codes`**`(n)`

Return the dictionary of inequivalent binary self dual codes of length n.

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

The number of inequivalent "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 $x^4, x^6, \ldots$ [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 [HP2003], the number B_n of inequivalent sd binary codes of length n is given:

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

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

EXAMPLES:

```
sage: C = codes.databases.self_dual_binary_codes(10)
sage: C["10"]["0"]["code"] == C["10"]["0"]["code"].dual_code()
True
sage: C["10"]["1"]["code"] == C["10"]["1"]["code"].dual_code()
True
sage: len(C["10"].keys())  # number of inequiv sd codes of length 10
2
sage: C = codes.databases.self_dual_binary_codes(12)
sage: C["12"]["0"]["code"] == C["12"]["0"]["code"].dual_code()
True
sage: C["12"]["1"]["code"] == C["12"]["1"]["code"].dual_code()
True
sage: C["12"]["2"]["code"] == C["12"]["2"]["code"].dual_code()
True
```

11.14 Optimized low-level binary code representation

Some computations with linear binary codes. Fix a basis for $\mathbf{F}_2^n$. A linear binary code is a linear subspace of $\mathbf{F}_2^n$, together with this choice of basis. A permutation $g \in S_n$ of the fixed basis gives rise to a permutation of the vectors, or words, in $\mathbf{F}_2^n$, sending (w_i) to $(w_{g(i)})$. The permutation automorphism group of the code C is the set of permutations of the basis that bijectively map C to itself. Note that if g is such a permutation, then

$$g(a_i) + g(b_i) = (a_{g(i)} + b_{g(i)}) = g((a_i) + (b_i)).$$

Over other fields, it is also required that the map be linear, which as per above boils down to scalar multiplication. However, over $\mathbf{F}_2$, the only scalars are 0 and 1, so the linearity condition has trivial effect.

AUTHOR:

- Robert L Miller (Oct-Nov 2007)
- compiled code data structure
- union-find based orbit partition
- optimized partition stack class
- NICE-based partition refinement algorithm
- canonical generation function

class `sage.coding.binary_code.`**`BinaryCode`**

Bases: `object`

Minimal, but optimized, binary code object.

EXAMPLES:

```
sage: import sage.coding.binary_code
sage: from sage.coding.binary_code import *
sage: M = Matrix(GF(2), [[1,1,1,1]])
sage: B = BinaryCode(M)     # create from matrix
sage: C = BinaryCode(B, 60) # create using glue
sage: D = BinaryCode(C, 240)
sage: E = BinaryCode(D, 85)
sage: B
Binary [4,1] linear code, generator matrix
[1111]
sage: C
Binary [6,2] linear code, generator matrix
[111100]
[001111]
sage: D
Binary [8,3] linear code, generator matrix
[11110000]
[00111100]
[00001111]
sage: E
Binary [8,4] linear code, generator matrix
[11110000]
[00111100]
[00001111]
[10101010]
```

(continues on next page)

(continued from previous page)

```
sage: M = Matrix(GF(2), [[1]*32])
sage: B = BinaryCode(M)
sage: B
Binary [32,1] linear code, generator matrix
[11111111111111111111111111111111]
```

apply_permutation(*labeling*)

Apply a column permutation to the code.

INPUT:

- labeling – a list permutation of the columns

EXAMPLES:

```
sage: from sage.coding.binary_code import *
sage: B = BinaryCode(codes.GolayCode(GF(2)).generator_matrix())
sage: B
Binary [24,12] linear code, generator matrix
[100000000000101011100011]
[010000000000111110010010]
[001000000000110100101011]
[000100000000110001110110]
[000010000000110011011001]
[000001000000011001101101]
[000000100000001100110111]
[000000010000101101111000]
[000000001000010110111100]
[000000000100001011011110]
[000000000010101110001101]
[000000000001010111000111]
sage: B.apply_permutation(list(range(11,-1,-1)) + list(range(12, 24)))
sage: B
Binary [24,12] linear code, generator matrix
[000000000001101011100011]
[000000000010111110010010]
[000000000100110100101011]
[000000001000110001110110]
[000000010000110011011001]
[000000100000011001101101]
[000001000000001100110111]
[000010000000101101111000]
[000100000000010110111100]
[001000000000001011011110]
[010000000000101110001101]
[100000000000010111000111]
```

matrix()

Return the generator matrix of the *BinaryCode*, i.e. the code is the rowspace of `B.matrix()`.

EXAMPLES:

```
sage: M = Matrix(GF(2), [[1,1,1,1,0,0], [0,0,1,1,1,1]])
sage: from sage.coding.binary_code import *
sage: B = BinaryCode(M)
sage: B.matrix()
[1 1 1 1 0 0]
[0 0 1 1 1 1]
```

print_data()

Print all data for self.

EXAMPLES:

```
sage: import sage.coding.binary_code
sage: from sage.coding.binary_code import *
sage: M = Matrix(GF(2), [[1,1,1,1]])
sage: B = BinaryCode(M)
sage: C = BinaryCode(B, 60)
sage: D = BinaryCode(C, 240)
sage: E = BinaryCode(D, 85)
sage: B.print_data() # random - actually "print(P.print_data())"
ncols: 4
nrows: 1
nwords: 2
radix: 32
basis:
1111
words:
0000
1111
sage: C.print_data() # random - actually "print(P.print_data())"
ncols: 6
nrows: 2
nwords: 4
radix: 32
basis:
111100
001111
words:
000000
111100
001111
110011
sage: D.print_data() # random - actually "print(P.print_data())"
ncols: 8
nrows: 3
nwords: 8
radix: 32
basis:
11110000
00111100
00001111
words:
00000000
```

(continues on next page)

(continued from previous page)

```
11110000
00111100
11001100
00001111
11111111
00110011
11000011
sage: E.print_data() # random - actually "print(P.print_data())"
ncols: 8
nrows: 4
nwords: 16
radix: 32
basis:
11110000
00111100
00001111
10101010
words:
00000000
11110000
00111100
11001100
00001111
11111111
00110011
11000011
10101010
01011010
10010110
01100110
10100101
01010101
10011001
01101001
```

put_in_std_form()

Put the code in binary form, which is defined by an identity matrix on the left, augmented by a matrix of data.

EXAMPLES:

```
sage: from sage.coding.binary_code import *
sage: M = Matrix(GF(2), [[1,1,1,1,0,0], [0,0,1,1,1,1]])
sage: B = BinaryCode(M); B
Binary [6,2] linear code, generator matrix
[111100]
[001111]
sage: B.put_in_std_form(); B
0
Binary [6,2] linear code, generator matrix
[101011]
[010111]
```

class sage.coding.binary_code.**BinaryCodeClassifier**

Bases: `object`

generate_children(*B*, *n*, *d=2*)

Use canonical augmentation to generate children of the code B.

INPUT:

- `B` – a *BinaryCode*
- `n` – limit on the degree of the code
- `d` – test whether new vector has weight divisible by d. If $d = 4$, this ensures that all doubly-even canonically augmented children are generated.

EXAMPLES:

```
sage: from sage.coding.binary_code import *
sage: BC = BinaryCodeClassifier()
sage: B = BinaryCode(Matrix(GF(2), [[1,1,1,1]]))
sage: BC.generate_children(B, 6, 4)
[
[1 1 1 1 0 0]
[0 1 0 1 1 1]
]
```

Note: The function `codes.databases.self_orthogonal_binary_codes` makes heavy use of this function.

MORE EXAMPLES:

```
sage: soc_iter = codes.databases.self_orthogonal_binary_codes(12, 6, 4)
sage: L = list(soc_iter)
sage: for n in range(13):
....:   s = 'n=%2d : '%n
....:   for k in range(1,7):
....:       s += '%3d '%len([C for C in L
....:                        if C.length() == n and C.dimension() == k])
....:   print(s)
n= 0 :   0   0   0   0   0   0
n= 1 :   0   0   0   0   0   0
n= 2 :   0   0   0   0   0   0
n= 3 :   0   0   0   0   0   0
n= 4 :   1   0   0   0   0   0
n= 5 :   0   0   0   0   0   0
n= 6 :   0   1   0   0   0   0
n= 7 :   0   0   1   0   0   0
n= 8 :   1   1   1   1   0   0
n= 9 :   0   0   0   0   0   0
n=10 :   0   1   1   1   0   0
n=11 :   0   0   1   1   0   0
n=12 :   1   2   3   4   2   0
```

put_in_canonical_form(*B*)

Puts the code into canonical form.

Canonical form is obtained by performing row reduction, permuting the pivots to the front so that the generator matrix is of the form: the identity matrix augmented to the right by arbitrary data.

EXAMPLES:

```
sage: from sage.coding.binary_code import *
sage: BC = BinaryCodeClassifier()
sage: B = BinaryCode(codes.GolayCode(GF(2)).generator_matrix())
sage: B.apply_permutation(list(range(24,-1,-1)))
sage: B
Binary [24,12] linear code, generator matrix
[011000111010100000000000]
[001001001111100000000001]
[011010100101100000000010]
[001101110001100000000100]
[010011011001100000001000]
[010110110011000000010000]
[011101100110000000100000]
[000011110110100001000000]
[000111101101000010000000]
[001111011010000100000000]
[010110001110101000000000]
[011100011101010000000000]
sage: BC.put_in_canonical_form(B)
sage: B
Binary [24,12] linear code, generator matrix
[100000000000001100111001]
[010000000000001010001111]
[001000000000001111010010]
[000100000000010110101010]
[000010000000010110010101]
[000001000000010001101101]
[000000100000011000110110]
[000000010000011111001001]
[000000001000010101110011]
[000000000100010011011110]
[000000000010001011110101]
[000000000001001101101110]
```

class `sage.coding.binary_code.`**`OrbitPartition`**

Bases: `object`

Structure which keeps track of which vertices are equivalent under the part of the automorphism group that has already been seen, during search. Essentially a disjoint-set data structure*, which also keeps track of the minimum element and size of each cell of the partition, and the size of the partition.

See Wikipedia article Disjoint-set_data_structure

class `sage.coding.binary_code.`**`PartitionStack`**

Bases: `object`

Partition stack structure for traversing the search tree during automorphism group computation.

`cmp`(*other*, *CG*)

EXAMPLES:

```
sage: import sage.coding.binary_code
sage: from sage.coding.binary_code import *
sage: M = Matrix(GF(2), [[1,1,1,1,0,0,0,0], [0,0,1,1,1,1,0,0],
....:                    [0,0,0,0,1,1,1,1], [1,0,1,0,1,0,1,0]])
sage: B = BinaryCode(M)
sage: P = PartitionStack(4, 8)
sage: P._refine(0, [[0,0],[1,0]], B)
181
sage: P._split_vertex(0, 1)
0
sage: P._refine(1, [[0,0]], B)
290
sage: P._split_vertex(1, 2)
1
sage: P._refine(2, [[0,1]], B)
463
sage: P._split_vertex(2, 3)
2
sage: P._refine(3, [[0,2]], B)
1500
sage: P._split_vertex(4, 4)
4
sage: P._refine(4, [[0,4]], B)
1224
sage: P._is_discrete(4)
1
sage: Q = PartitionStack(P)
sage: Q._clear(4)
sage: Q._split_vertex(5, 4)
4
sage: Q._refine(4, [[0,4]], B)
1224
sage: Q._is_discrete(4)
1
sage: Q.cmp(P, B)
0
```

print_basis()

EXAMPLES:

```
sage: import sage.coding.binary_code
sage: from sage.coding.binary_code import *
sage: P = PartitionStack(4, 8)
sage: P._dangerous_dont_use_set_ents_lvls(
....:     list(range(8)),
....:     list(range(7)) + [-1],
....:     [4,7,12,11,1,9,3,0,2,5,6,8,10,13,14,15],
....:     [0]*16)
sage: P
({4},{7},{12},{11},{1},{9},{3},{0},{2},{5},{6},{8},{10},{13},{14},{15})  ({0},
↪{1,2,3,4,5,6,7})
({4},{7},{12},{11},{1},{9},{3},{0},{2},{5},{6},{8},{10},{13},{14},{15})  ({0},
↪{1},{2,3,4,5,6,7})
```

(continues on next page)

(continued from previous page)

```
({4},{7},{12},{11},{1},{9},{3},{0},{2},{5},{6},{8},{10},{13},{14},{15})  ({0},
↪{1},{2},{3,4,5,6,7})
({4},{7},{12},{11},{1},{9},{3},{0},{2},{5},{6},{8},{10},{13},{14},{15})  ({0},
↪{1},{2},{3},{4,5,6,7})
({4},{7},{12},{11},{1},{9},{3},{0},{2},{5},{6},{8},{10},{13},{14},{15})  ({0},
↪{1},{2},{3},{4},{5,6,7})
({4},{7},{12},{11},{1},{9},{3},{0},{2},{5},{6},{8},{10},{13},{14},{15})  ({0},
↪{1},{2},{3},{4},{5},{6,7})
({4},{7},{12},{11},{1},{9},{3},{0},{2},{5},{6},{8},{10},{13},{14},{15})  ({0},
↪{1},{2},{3},{4},{5},{6},{7})
sage: P._find_basis()
sage: P.print_basis()
basis_locations:
4
8
0
11
```

print_data()

Prints all data for self.

EXAMPLES:

```
sage: import sage.coding.binary_code
sage: from sage.coding.binary_code import *
sage: P = PartitionStack(2, 6)
sage: print(P.print_data())
nwords:4
nrows:2
ncols:6
radix:32
wd_ents:
0
1
2
3
wd_lvls:
12
12
12
-1
col_ents:
0
1
2
3
4
5
col_lvls:
12
12
12
```

(continues on next page)

(continued from previous page)

```
12
12
-1
col_degs:
0
0
0
0
0
0
col_counts:
0
0
0
0
col_output:
0
0
0
0
0
0
wd_degs:
0
0
0
0
wd_counts:
0
0
0
0
0
0
0
wd_output:
0
0
0
0
```

`sage.coding.binary_code.test_expand_to_ortho_basis(B=None)`

This function is written in pure C for speed, and is tested from this function.

INPUT:

- `B` – a *BinaryCode* in standard form

OUTPUT:

An array of codewords which represent the expansion of a basis for B to a basis for $(B')^\perp$, where $B' = B$ if the all-ones vector 1 is in B, otherwise $B' = \text{span}(B, 1)$ (note that this guarantees that all the vectors in the span of the output have even weight).

sage.coding.binary_code.**test_word_perms**(*t_limit=5.0*)

Test the `WordPermutation` structs for at least `t_limit` seconds.

These are structures written in pure C for speed, and are tested from this function, which performs the following tests:

1. Tests `create_word_perm()`, which creates a `WordPermutation` from a Python list L representing a permutation $i \mapsto L[i]$. Takes a random word and permutes it by a random list permutation, and tests that the result agrees with doing it the slow way.

1b. Tests `create_array_word_perm()`, which creates a
`WordPermutation` from a C array. Does the same as above.

2. Tests `create_comp_word_perm()`, which creates a `WordPermutation` as a composition of two `WordPermutation` objects. Takes a random word and two random permutations, and tests that the result of permuting by the composition is correct.
3. Tests `create_inv_word_perm()` and `create_id_word_perm()`, which create a `WordPermutation` as the inverse and identity permutations, resp. Takes a random word and a random permutation, and tests that the result permuting by the permutation and its inverse in either order, and permuting by the identity both return the original word.

Note: The functions `permute_word_by_wp()` and `dealloc_word_perm()` are implicitly involved in each of the above tests.

sage.coding.binary_code.**weight_dist**(*M*)

Computes the weight distribution of the row space of M.

EXAMPLES:

```
sage: from sage.coding.binary_code import weight_dist
sage: M = Matrix(GF(2), [
....:  [1,1,1,1,1,1,1,1,0,0,0,0,0,0,0,0],
....:  [0,0,0,0,1,1,1,1,1,1,1,1,0,0,0,0],
....:  [0,0,0,0,0,0,0,0,1,1,1,1,1,1,1,1],
....:  [0,0,1,1,0,0,1,1,0,0,1,1,0,0,1,1],
....:  [0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1]])
sage: weight_dist(M)
[1, 0, 0, 0, 0, 0, 0, 0, 30, 0, 0, 0, 0, 0, 0, 0, 1]
sage: M = Matrix(GF(2), [
....:  [1,1,1,1,1,1,1,1,0,0,0,0,0,0,0,0,0],
....:  [0,0,0,0,0,0,1,1,1,1,1,1,1,1,0,0,0],
....:  [0,0,0,0,0,1,0,1,0,0,0,1,1,1,1,1,1],
....:  [0,0,0,1,1,0,0,0,0,1,1,0,1,1,0,1,1]])
sage: weight_dist(M)
[1, 0, 0, 0, 0, 0, 0, 0, 11, 0, 0, 0, 4, 0, 0, 0, 0, 0]
sage: M = Matrix(GF(2), [
....:  [1,0,0,1,1,1,1,0,0,1,0,0,0,0,0,0,0],
....:  [0,1,0,0,1,1,1,1,0,0,1,0,0,0,0,0,0],
....:  [0,0,1,0,0,1,1,1,1,0,0,1,0,0,0,0,0],
....:  [0,0,0,1,0,0,1,1,1,1,0,0,1,0,0,0,0],
....:  [0,0,0,0,1,0,0,1,1,1,1,0,0,1,0,0,0],
....:  [0,0,0,0,0,1,0,0,1,1,1,1,0,0,1,0,0],
```

(continues on next page)

(continued from previous page)

```
....:  [0,0,0,0,0,0,1,0,0,1,1,1,1,0,0,1,0],
....:  [0,0,0,0,0,0,0,1,0,0,1,1,1,1,0,0,1]])
sage: weight_dist(M)
[1, 0, 0, 0, 0, 0, 68, 0, 85, 0, 68, 0, 34, 0, 0, 0, 0, 0]
```

CHAPTER

TWELVE

DERIVED CODE CONSTRUCTIONS

Sage supports the following derived code constructions. If the constituent code is from a special code family, the derived codes inherit structural properties like decoding radius or minimum distance:

12.1 Subfield subcode

Let C be a $[n, k]$ code over $\mathbf{F}_{q^t}$. Let $Cs = \{c \in C | \forall i, c_i \in \mathbf{F}_q\}$, c_i being the i-th coordinate of c.

Cs is called the subfield subcode of C over $\mathbf{F}_q$

class sage.coding.subfield_subcode.**SubfieldSubcode**(*original_code*, *subfield*, *embedding=None*)

Bases: *AbstractLinearCode*

Representation of a subfield subcode.

INPUT:

- original_code – the code self comes from.
- subfield – the base field of self.
- embedding – (default: None) an homomorphism from subfield to original_code's base field. If None is provided, it will default to the first homomorphism of the list of homomorphisms Sage can build.

EXAMPLES:

```
sage: C = codes.random_linear_code(GF(16, 'aa'), 7, 3)
sage: codes.SubfieldSubcode(C, GF(4, 'a'))
Subfield subcode of [7, 3] linear code over GF(16) down to GF(4)
```

dimension()

Return the dimension of self.

EXAMPLES:

```
sage: C = codes.GeneralizedReedSolomonCode(GF(16, 'aa').list()[:13], 5)
sage: Cs = codes.SubfieldSubcode(C, GF(4, 'a'))
sage: Cs.dimension()
3
```

dimension_lower_bound()

Return a lower bound for the dimension of self.

EXAMPLES:

```
sage: C = codes.random_linear_code(GF(16, 'aa'), 7, 3)
sage: Cs = codes.SubfieldSubcode(C, GF(4, 'a'))
sage: Cs.dimension_lower_bound()
-1
```

dimension_upper_bound()

Return an upper bound for the dimension of `self`.

EXAMPLES:

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

embedding()

Return the field embedding between the base field of `self` and the base field of its original code.

EXAMPLES:

```
sage: C = codes.random_linear_code(GF(16, 'aa'), 7, 3)
sage: Cs = codes.SubfieldSubcode(C, GF(4, 'a'))
sage: Cs.embedding()
Ring morphism:
  From: Finite Field in a of size 2^2
  To:   Finite Field in aa of size 2^4
  Defn: a |--> aa^2 + aa
```

original_code()

Return the original code of `self`.

EXAMPLES:

```
sage: C = codes.random_linear_code(GF(16, 'aa'), 7, 3)
sage: Cs = codes.SubfieldSubcode(C, GF(4, 'a'))
sage: Cs.original_code()
[7, 3] linear code over GF(16)
```

parity_check_matrix()

Return a parity check matrix of `self`.

EXAMPLES:

```
sage: C = codes.GeneralizedReedSolomonCode(GF(16, 'aa').list()[:13], 5)
sage: Cs = codes.SubfieldSubcode(C, GF(4, 'a'))
sage: Cs.parity_check_matrix()
[    1     0     0     0     0     0     0     0     0     0     1 a + 1 a + 1]
[    0     1     0     0     0     0     0     0     0     0 a + 1     0     a]
[    0     0     1     0     0     0     0     0     0     0 a + 1     a     0]
[    0     0     0     1     0     0     0     0     0     0     0 a + 1     a]
[    0     0     0     0     1     0     0     0     0     0 a + 1     1 a + 1]
[    0     0     0     0     0     1     0     0     0     0     1     1     1]
[    0     0     0     0     0     0     1     0     0     0     a     a     1]
[    0     0     0     0     0     0     0     1     0     0     a     1     a]
```

(continues on next page)

(continued from previous page)

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

class sage.coding.subfield_subcode.**SubfieldSubcodeOriginalCodeDecoder**(*code*, *original_decoder=None*, ***kwargs*)

Bases: *Decoder*

Decoder decoding through a decoder over the original code of code.

INPUT:

- code – The associated code of this decoder
- original_decoder – (default: None) The decoder that will be used over the original code. It has to be a decoder object over the original code. If it is set to None, the default decoder over the original code will be used.
- **kwargs – All extra arguments are forwarded to original code's decoder

EXAMPLES:

```
sage: C = codes.GeneralizedReedSolomonCode(GF(16, 'aa').list()[:13], 5)
sage: Cs = codes.SubfieldSubcode(C, GF(4, 'a'))
sage: codes.decoders.SubfieldSubcodeOriginalCodeDecoder(Cs)
Decoder of Subfield subcode of [13, 5, 9] Reed-Solomon Code over GF(16) down to␣
↪GF(4)
 through Gao decoder for [13, 5, 9] Reed-Solomon Code over GF(16)
```

decode_to_code(*y*)

Return an error-corrected codeword from y.

EXAMPLES:

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

decoding_radius(***kwargs*)

Return the maximal number of errors self can decode.

INPUT:

- kwargs – Optional arguments are forwarded to original decoder's *sage.coding.decoder.Decoder.decoding_radius()* method.

EXAMPLES:

```
sage: C = codes.GeneralizedReedSolomonCode(GF(16, 'aa').list()[:13], 5)
sage: Cs = codes.SubfieldSubcode(C, GF(4, 'a'))
sage: D = codes.decoders.SubfieldSubcodeOriginalCodeDecoder(Cs)
```

(continues on next page)

(continued from previous page)

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

original_decoder()

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

EXAMPLES:

```
sage: C = codes.GeneralizedReedSolomonCode(GF(16, 'aa').list()[:13], 5)
sage: Cs = codes.SubfieldSubcode(C, GF(4, 'a'))
sage: D = codes.decoders.SubfieldSubcodeOriginalCodeDecoder(Cs)
sage: D.original_decoder()
Gao decoder for [13, 5, 9] Reed-Solomon Code over GF(16)
```

12.2 Punctured code

Let C be a linear code. Let C_i be the set of all words of C with the i-th coordinate being removed. C_i is the punctured code of C on the i-th position.

class sage.coding.punctured_code.**PuncturedCode**(*C*, *positions*)

Bases: *AbstractLinearCode*

Representation of a punctured code.

- C – A linear code
- positions – the positions where C will be punctured. It can be either an integer if one need to puncture only one position, a list or a set of positions to puncture. If the same position is passed several times, it will be considered only once.

EXAMPLES:

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

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

dimension()

Return the dimension of self.

EXAMPLES:

```
sage: set_random_seed(42)
sage: C = codes.random_linear_code(GF(7), 11, 5)
sage: Cp = codes.PuncturedCode(C, 3)
sage: Cp.dimension()
5
```

encode(*m*, *original_encode=False*, *encoder_name=None*, ***kwargs*)

Transforms an element of the message space into an element of the code.

INPUT:

- `m` – a vector of the message space of the code.
- `original_encode` – (default: `False`) if this is set to `True`, `m` will be encoded using an Encoder of `self`'s *original_code()*. This allow to avoid the computation of a generator matrix for `self`.
- `encoder_name` – (default: `None`) Name of the encoder which will be used to encode `word`. The default encoder of `self` will be used if default value is kept

OUTPUT:

- an element of `self`

EXAMPLES:

```
sage: M = matrix(GF(7), [[1, 0, 0, 0, 3, 4, 6],
....:                    [0, 1, 0, 6, 1, 6, 4],
....:                    [0, 0, 1, 5, 2, 2, 4]])
sage: C_original = LinearCode(M)
sage: Cp = codes.PuncturedCode(C_original, 2)
sage: m = vector(GF(7), [1, 3, 5])
sage: Cp.encode(m)
 (1, 3, 5, 5, 0, 2)
```

original_code()

Return the linear code which was punctured to get `self`.

EXAMPLES:

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

punctured_positions()

Return the list of positions which were punctured on the original code.

EXAMPLES:

```
sage: C = codes.random_linear_code(GF(7), 11, 5)
sage: Cp = codes.PuncturedCode(C, 3)
sage: Cp.punctured_positions()
{3}
```

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

Return a random codeword of `self`.

This method does not trigger the computation of `self`'s `sage.coding.linear_code_no_metric.generator_matrix()`.

INPUT:

- `agrs, kwds - extra positional arguments passed to sage.modules.free_module.random_element()`.

EXAMPLES:

```
sage: C = codes.random_linear_code(GF(7), 11, 5)
sage: Cp = codes.PuncturedCode(C, 3)
sage: Cp.random_element() in Cp
True
```

structured_representation()

Return `self` as a structured code object.

If `self` has a specific structured representation (e.g. a punctured GRS code is a GRS code too), it will return this representation, else it returns a *sage.coding.linear_code.LinearCode*.

EXAMPLES:

We consider a GRS code:

```
sage: C_grs = codes.GeneralizedReedSolomonCode(GF(59).list()[:40], 12)
```

A punctured GRS code is still a GRS code:

```
sage: Cp_grs = codes.PuncturedCode(C_grs, 3)
sage: Cp_grs.structured_representation()
[39, 12, 28] Reed-Solomon Code over GF(59)
```

Another example with structureless linear codes:

```
sage: set_random_seed(42)
sage: C_lin  = codes.random_linear_code(GF(2), 10, 5)
sage: Cp_lin = codes.PuncturedCode(C_lin, 2)
sage: Cp_lin.structured_representation()
[9, 5] linear code over GF(2)
```

class sage.coding.punctured_code.**PuncturedCodeOriginalCodeDecoder**(*code*, *strategy=None*, *original_decoder=None*, ***kwargs*)

Bases: *Decoder*

Decoder decoding through a decoder over the original code of its punctured code.

INPUT:

- `code` – The associated code of this encoder
- `strategy` – (default: `None`) the strategy used to decode. The available strategies are:
  - `'error-erasure'` – uses an error-erasure decoder over the original code if available, fails otherwise.
  - `'random-values'` – fills the punctured positions with random elements in `code`'s base field and tries to decode using the default decoder of the original code
  - `'try-all'` – fills the punctured positions with every possible combination of symbols until decoding succeeds, or until every combination have been tried
  - `None` – uses `error-erasure` if an error-erasure decoder is available, switch to `random-values` behaviour otherwise
- `original_decoder` – (default: `None`) the decoder that will be used over the original code. It has to be a decoder object over the original code. This argument takes precedence over `strategy`: if both `original_decoder` and `strategy` are filled, `self` will use the `original_decoder` to decode over the original code. If `original_decoder` is set to `None`, it will use the decoder picked by `strategy`.

- **kwargs – all extra arguments are forwarded to original code's decoder

EXAMPLES:

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

As seen above, if all optional are left blank, and if an error-erasure decoder is available, it will be chosen as the original decoder. Now, if one forces `strategy` to `'try-all'` or `'random-values'`, the default decoder of the original code will be chosen, even if an error-erasure is available:

```
sage: C = codes.GeneralizedReedSolomonCode(GF(16, 'a').list()[:15], 7)
sage: Cp = codes.PuncturedCode(C, 3)
sage: D = codes.decoders.PuncturedCodeOriginalCodeDecoder(Cp, strategy="try-all")
sage: "error-erasure" in D.decoder_type()
False
```

And if one fills `original_decoder` and `strategy` fields with contradictory elements, the `original_decoder` takes precedence:

```
sage: C = codes.GeneralizedReedSolomonCode(GF(16, 'a').list()[:15], 7)
sage: Cp = codes.PuncturedCode(C, 3)
sage: Dor = C.decoder("Gao")
sage: D = codes.decoders.PuncturedCodeOriginalCodeDecoder(Cp, original_decoder=Dor,
....:                                                         strategy="error-erasure")
sage: D.original_decoder() == Dor
True
```

decode_to_code(*y*)

Decode y to an element in *sage.coding.decoder.Decoder.code()*.

EXAMPLES:

```
sage: C = codes.GeneralizedReedSolomonCode(GF(16, 'a').list()[:15], 7)
sage: Cp = codes.PuncturedCode(C, 3)
sage: D = codes.decoders.PuncturedCodeOriginalCodeDecoder(Cp)
sage: c = Cp.random_element()
sage: Chan = channels.StaticErrorRateChannel(Cp.ambient_space(), 3)
sage: y = Chan(c)
sage: y in Cp
False
sage: D.decode_to_code(y) == c
True
```

decoding_radius(*number_erasures=None*)

Return the maximal number of errors that `self` can decode.

EXAMPLES:

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

(continues on next page)

(continued from previous page)

```
sage: D.decoding_radius(2)
2
```

original_decoder()

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

EXAMPLES:

```
sage: C = codes.GeneralizedReedSolomonCode(GF(16, 'a').list()[:15], 7)
sage: Cp = codes.PuncturedCode(C, 3)
sage: D = codes.decoders.PuncturedCodeOriginalCodeDecoder(Cp)
sage: D.original_decoder()
Error-Erasure decoder for [15, 7, 9] Reed-Solomon Code over GF(16)
```

class sage.coding.punctured_code.**PuncturedCodePuncturedMatrixEncoder**(*code*)

Bases: *Encoder*

Encoder using original code generator matrix to compute the punctured code's one.

INPUT:

- code – The associated code of this encoder.

EXAMPLES:

```
sage: C = codes.random_linear_code(GF(7), 11, 5)
sage: Cp = codes.PuncturedCode(C, 3)
sage: E = codes.encoders.PuncturedCodePuncturedMatrixEncoder(Cp)
sage: E
Punctured matrix-based encoder for the
 Puncturing of [11, 5] linear code over GF(7) on position(s) [3]
```

generator_matrix()

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

EXAMPLES:

```
sage: set_random_seed(10)
sage: C = codes.random_linear_code(GF(7), 11, 5)
sage: Cp = codes.PuncturedCode(C, 3)
sage: E = codes.encoders.PuncturedCodePuncturedMatrixEncoder(Cp)
sage: E.generator_matrix()
[1 0 0 0 0 5 2 6 0 6]
[0 1 0 0 0 5 2 2 1 1]
[0 0 1 0 0 6 2 4 0 4]
[0 0 0 1 0 0 6 3 3 3]
[0 0 0 0 1 0 1 3 4 3]
```

12.3 Extended code

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

$$\hat{C} = \{x_1 x_2 \dots x_{n+1} \in \mathbf{F}_q^{n+1} \,|\, x_1 x_2 \dots x_n \in C \text{ with } x_1 + x_2 + \dots + x_{n+1} = 0\}.$$

See [HP2003] (pp 15-16) for details.

class sage.coding.extended_code.**ExtendedCode**(*C*)

Bases: *AbstractLinearCode*

Representation of an extended code.

INPUT:

- C – A linear code

EXAMPLES:

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

original_code()

Return the code which was extended to get self.

EXAMPLES:

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

parity_check_matrix()

Return a parity check matrix of self.

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

EXAMPLES:

```
sage: C = LinearCode(matrix(GF(2),[[1,0,0,1,1],\
                                   [0,1,0,1,0],\
                                   [0,0,1,1,1]]))
sage: C.parity_check_matrix()
[1 0 1 0 1]
[0 1 0 1 1]
sage: Ce = codes.ExtendedCode(C)
sage: Ce.parity_check_matrix()
[1 1 1 1 1 1]
[1 0 1 0 1 0]
[0 1 0 1 1 0]
```

random_element()

Return a random element of self.

This random element is computed directly from the original code, and does not compute a generator matrix of self in the process.

EXAMPLES:

```
sage: C = codes.random_linear_code(GF(7), 9, 5)
sage: Ce = codes.ExtendedCode(C)
sage: c = Ce.random_element() #random
sage: c in Ce
True
```

class sage.coding.extended_code.**ExtendedCodeExtendedMatrixEncoder**(*code*)

Bases: *Encoder*

Encoder using original code's generator matrix to compute the extended code's one.

INPUT:

- code – The associated code of self.

generator_matrix()

Return a generator matrix of the associated code of self.

EXAMPLES:

```
sage: C = LinearCode(matrix(GF(2),[[1,0,0,1,1],\
                                   [0,1,0,1,0],\
                                   [0,0,1,1,1]]))
sage: Ce = codes.ExtendedCode(C)
sage: E = codes.encoders.ExtendedCodeExtendedMatrixEncoder(Ce)
sage: E.generator_matrix()
[1 0 0 1 1 1]
[0 1 0 1 0 0]
[0 0 1 1 1 1]
```

class sage.coding.extended_code.**ExtendedCodeOriginalCodeDecoder**(*code*, *original_decoder=None*, ***kwargs*)

Bases: *Decoder*

Decoder which decodes through a decoder over the original code.

INPUT:

- code – The associated code of this decoder
- original_decoder – (default: None) the decoder that will be used over the original code. It has to be a decoder object over the original code. If original_decoder is set to None, it will use the default decoder of the original code.
- **kwargs – all extra arguments are forwarded to original code's decoder

EXAMPLES:

```
sage: C = codes.GeneralizedReedSolomonCode(GF(16, 'a').list()[:15], 7)
sage: Ce = codes.ExtendedCode(C)
sage: D = codes.decoders.ExtendedCodeOriginalCodeDecoder(Ce)
sage: D
Decoder of Extension of [15, 7, 9] Reed-Solomon Code over GF(16)
 through Gao decoder for [15, 7, 9] Reed-Solomon Code over GF(16)
```

decode_to_code(*y*, ***kwargs*)

Decode y to an element in *sage.coding.decoder.Decoder.code()*.

EXAMPLES:

```
sage: C = codes.GeneralizedReedSolomonCode(GF(16, 'a').list()[:15], 7)
sage: Ce = codes.ExtendedCode(C)
sage: D = codes.decoders.ExtendedCodeOriginalCodeDecoder(Ce)
sage: c = Ce.random_element()
sage: Chan = channels.StaticErrorRateChannel(Ce.ambient_space(),
....:                                          D.decoding_radius())
sage: y = Chan(c)
sage: y in Ce
False
sage: D.decode_to_code(y) == c
True
```

Another example, with a list decoder:

```
sage: C = codes.GeneralizedReedSolomonCode(GF(16, 'a').list()[:15], 7)
sage: Ce = codes.ExtendedCode(C)
sage: Dgrs = C.decoder('GuruswamiSudan', tau=4)
sage: D = codes.decoders.ExtendedCodeOriginalCodeDecoder(Ce,
....:                                                        original_decoder=Dgrs)
sage: c = Ce.random_element()
sage: Chan = channels.StaticErrorRateChannel(Ce.ambient_space(),
....:                                          D.decoding_radius())
sage: y = Chan(c)
sage: y in Ce
False
sage: c in D.decode_to_code(y)
True
```

decoding_radius(**args*, ***kwargs*)

Return maximal number of errors that self can decode.

INPUT:

- *args, **kwargs – arguments and optional arguments are forwarded to original decoder's decoding_radius method.

EXAMPLES:

```
sage: C = codes.GeneralizedReedSolomonCode(GF(16, 'a').list()[:15], 7)
sage: Ce = codes.ExtendedCode(C)
sage: D = codes.decoders.ExtendedCodeOriginalCodeDecoder(Ce)
sage: D.decoding_radius()
4
```

original_decoder()

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

EXAMPLES:

```
sage: C = codes.GeneralizedReedSolomonCode(GF(16, 'a').list()[:15], 7)
sage: Ce = codes.ExtendedCode(C)
sage: D = codes.decoders.ExtendedCodeOriginalCodeDecoder(Ce)
sage: D.original_decoder()
Gao decoder for [15, 7, 9] Reed-Solomon Code over GF(16)
```

Other derived constructions that simply produce the modified generator matrix can be found among the methods of a constructed code.

CHAPTER

THIRTEEN

DECODING

Information-set decoding for linear codes:

13.1 Information-set decoding for linear codes

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

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

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

AUTHORS:

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

class sage.coding.information_set_decoder.**InformationSetAlgorithm**(*code*, *decoding_interval*, *algorithm_name*, *parameters=None*)

Bases: `SageObject`

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

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

INPUT:

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

EXAMPLES:

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

A minimal working example of how to sub-class:

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

calibrate()

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

Must be overridden by sub-classes.

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

EXAMPLES:

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

code()

Return the code associated to this ISD algorithm.

EXAMPLES:

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

decode(*r*)

Decode a received word using this ISD decoding algorithm.

Must be overridden by sub-classes.

EXAMPLES:

```
sage: M = matrix(GF(2), [[1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0],\
                         [0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 1, 1],\
                         [0, 0, 0, 1, 0, 1, 0, 1, 1, 0, 0, 0],\
                         [0, 0, 0, 0, 1, 1, 0, 0, 1, 1, 0, 1],\
                         [0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 1]])
sage: C = codes.LinearCode(M)
sage: from sage.coding.information_set_decoder import LeeBrickellISDAlgorithm
sage: A = LeeBrickellISDAlgorithm(C, (2,2))
sage: r = vector(GF(2), [1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
sage: A.decode(r)
(0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)
```

decoding_interval()

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

The interval includes both end values.

EXAMPLES:

```
sage: C = codes.GolayCode(GF(2))
sage: from sage.coding.information_set_decoder import LeeBrickellISDAlgorithm
sage: A = LeeBrickellISDAlgorithm(C, (0,2))
sage: A.decoding_interval()
(0, 2)
```

name()

Return the name of this ISD algorithm.

EXAMPLES:

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

parameters()

Return any parameters this ISD algorithm uses.

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

EXAMPLES:

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

If not set, calibration will determine a sensible value:

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

time_estimate()

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

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

EXAMPLES:

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

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

Bases: *InformationSetAlgorithm*

The Lee-Brickell algorithm for information-set decoding.

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

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

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

1. Choose an information set I of C.
2. Compute $r' = r - r_I G_I^{-1} G$
3. Consider every size-p subset of I, $\{a_1, \dots, a_p\}$. For each $m = (m_1, \dots, m_p) \in \mathbf{F}_q^p$, compute the error vector $e = r' - \sum_{i=1}^p m_i g_{a_i}$,
4. If e has a Hamming weight at most w, return $r - e$.

INPUT:

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

EXAMPLES:

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

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

(continues on next page)

(continued from previous page)

```
sage: A = LeeBrickellISDAlgorithm(C, (2,3)); A
ISD Algorithm (Lee-Brickell) for [24, 12, 8] Extended Golay code over GF(2)
 decoding between 2 and 3 errors
```

calibrate()

Run some test computations to estimate the optimal search size.

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

$$\binom{n}{k}/(\rho \sum_{i=0}^{p} \binom{n-\tau}{k-i} \binom{\tau}{i})$$

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

The values T and P are here estimated by running a few test computations similar to those done by the decoding algorithm. We don't explicitly estimate ρ.

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

EXAMPLES:

```
sage: from sage.coding.information_set_decoder import LeeBrickellISDAlgorithm
sage: C = codes.GolayCode(GF(2))
sage: A = LeeBrickellISDAlgorithm(C, (0,3)); A
ISD Algorithm (Lee-Brickell) for [24, 12, 8] Extended Golay code over GF(2)
 decoding up to 3 errors
sage: A.calibrate()
sage: A.parameters() #random
{'search_size': 1}
sage: A.time_estimate() #random
0.0008162108571427874
```

If we specify the parameter at construction time, calibrate does not override this choice:

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

decode(*r*)

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

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

INPUT:

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

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

EXAMPLES:

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

class `sage.coding.information_set_decoder.`**`LinearCodeInformationSetDecoder`**(*code*, *number_errors*, *algorithm=None*, ***kwargs*)

Bases: `Decoder`

Information-set decoder for any linear code.

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

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

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

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

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

INPUT:

- code – A linear code for which to decode.
- number_errors – an integer, the maximal number of errors to accept as correct decoding. An interval can also be specified by giving a pair of integers, where both end values are taken to be in the interval.
- algorithm – (optional) the string name of the ISD algorithm to employ. If this is not set, an appropriate one will be chosen. A constructed *sage.coding.information_set_decoder.InformationSetAlgorithm* object may also be given. In this case number_errors must match that of the passed algorithm.
- **kwargs – (optional) any number of named arguments passed on to the ISD algorithm. Such are usually not required, and they can only be set if algorithm is set to a specific algorithm. See the documentation for each individual ISD algorithm class for information on any named arguments they may accept. The easiest way to access this documentation is to first construct the decoder without passing any named arguments, then accessing the ISD algorithm using *sage.coding.information_set_decoder.LinearCodeInformationSetDecoder.algorithm()*, and then reading the ? help on the constructed object.

EXAMPLES:

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

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

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

```
sage: C = codes.GolayCode(GF(3))
sage: D2 = C.decoder("InformationSet", 2, algorithm="Lee-Brickell", search_size=2);
↪D2
Information-set decoder (Lee-Brickell) for [12, 6, 6] Extended Golay code over GF(3)
 decoding up to 2 errors
sage: D2.algorithm()
ISD Algorithm (Lee-Brickell) for [12, 6, 6] Extended Golay code over GF(3)
 decoding up to 2 errors
sage: D2.algorithm().parameters()
{'search_size': 2}
```

If you specify an algorithm which is not known, you get a friendly error message:

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

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

```
sage: from sage.coding.information_set_decoder import LeeBrickellISDAlgorithm
sage: A = LeeBrickellISDAlgorithm(C, (0, 2))
sage: D = C.decoder("InformationSet", 2, algorithm=A); D
Information-set decoder (Lee-Brickell) for [12, 6, 6] Extended Golay code over GF(3)
 decoding up to 2 errors
```

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

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

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

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

There are two other ways to access this class:

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

sage: from sage.coding.information_set_decoder import␣
↪LinearCodeInformationSetDecoder
sage: D = LinearCodeInformationSetDecoder(C, 2); D
Information-set decoder (Lee-Brickell) for [12, 6, 6] Extended Golay code over GF(3)
 decoding up to 2 errors
```

algorithm()

Return the ISD algorithm used by this ISD decoder.

EXAMPLES:

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

decode_to_code(*r*)

Decodes a received word with respect to the associated code of this decoder.

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

INPUT:

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

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

EXAMPLES:

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

Information-set decoding a non-binary code:

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

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

```
sage: C = codes.QuadraticResidueCode(59, GF(3))
sage: c = C.random_element()
sage: Chan = channels.StaticErrorRateChannel(C.ambient_space(), 2)
sage: r = Chan(c)
sage: D = C.decoder('InformationSet', 8)
sage: c == D.decode_to_code(r) # long time
True
```

decoding_interval()

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

The interval includes both end values.

EXAMPLES:

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

decoding_radius()

Return the maximal number of errors this decoder can decode.

EXAMPLES:

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

(continues on next page)

(continued from previous page)

```
sage: D.decoding_radius()
2
```

static known_algorithms(*dictionary=False*)

Return the list of ISD algorithms that Sage knows.

Passing any of these to the constructor of *sage.coding.information_set_decoder.LinearCodeInformationSetDecoder* will make the ISD decoder use that algorithm.

INPUT:

- `dictionary` - optional. If set to `True`, return a `dict` mapping decoding algorithm name to its class.

OUTPUT: a list of strings or a `dict` from string to ISD algorithm class.

EXAMPLES:

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

Guruswami-Sudan interpolation-based list decoding for Reed-Solomon codes:

13.2 Guruswami-Sudan decoder for (Generalized) Reed-Solomon codes

REFERENCES:

- [GS1999]
- [Nie2013]

AUTHORS:

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

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

Bases: *Decoder*

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

The Guruswami-Sudan algorithm is a polynomial time algorithm to decode beyond half the minimum distance of the code. It can decode up to the Johnson radius which is $n - \sqrt(n(n-d))$, where n, d is the length, respectively minimum distance of the RS code. See [GS1999] for more details. It is a list-decoder meaning that it returns a list of all closest codewords or their corresponding message polynomials. Note that the output of the `decode_to_code` and `decode_to_message` methods are therefore lists.

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

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

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

INPUT:

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

  You can also supply your own function to perform the interpolation. See NOTE section for details on the signature of this function.
- `root_finder` – (default: `None`) the rootfinding algorithm that will be used. The following possibilities are currently available:
  - `"Alekhnovich"` – uses Alekhnovich's algorithm.
  - `"RothRuckenstein"` – uses Roth-Ruckenstein algorithm.
  - `None` – one of the above will be chosen based on the size of the code and the parameters.

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

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

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

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

EXAMPLES:

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

One can specify multiplicity and list size instead of `tau`:

```
sage: D = codes.decoders.GRSGuruswamiSudanDecoder(C, parameters=(1,2)); D
Guruswami-Sudan decoder for [250, 70, 181] Reed-Solomon Code over GF(251)
 decoding 97 errors with parameters (1, 2)
```

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

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

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

```
sage: D = codes.decoders.GRSGuruswamiSudanDecoder(C, parameters=(1,2),
....:                                              root_finder="RothRuckenstein"); D
Guruswami-Sudan decoder for [250, 70, 181] Reed-Solomon Code over GF(251)
 decoding 97 errors with parameters (1, 2)
```

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

```
sage: D = C.decoder("GuruswamiSudan", tau=97); D
Guruswami-Sudan decoder for [250, 70, 181] Reed-Solomon Code over GF(251)
 decoding 97 errors with parameters (1, 2)
```

decode_to_code(*r*)

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

INPUT:

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

EXAMPLES:

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

decode_to_message(*r*)

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

INPUT:

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

EXAMPLES:

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

decoding_radius()

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

EXAMPLES:

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

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

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

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

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

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

INPUT:

- `tau` – an integer, number of errors one expects Guruswami-Sudan algorithm to correct
- `s` – an integer, multiplicity parameter of Guruswami-Sudan algorithm
- `l` – an integer, list size parameter
- `C` – (default: `None`) a *GeneralizedReedSolomonCode*
- `n_k` – (default: `None`) a tuple of integers, respectively the length and the dimension of the *GeneralizedReedSolomonCode*

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

EXAMPLES:

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

One can also pass a GRS code:

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

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

```
sage: tau, s, l = 118, 47, 80
sage: GSD.gs_satisfactory(tau, s, l, n_k=(n, k))
False
```

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

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

Same if one provides none of these:

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

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

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

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

INPUT:

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

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

OUTPUT:

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

EXAMPLES:

```
sage: GSD = codes.decoders.GRSGuruswamiSudanDecoder
sage: n, k = 250, 70
sage: GSD.guruswami_sudan_decoding_radius(n_k=(n, k))
(118, (47, 89))
```

One parameter can be restricted at a time:

```
sage: n, k = 250, 70
sage: GSD.guruswami_sudan_decoding_radius(n_k=(n, k), s=3)
(109, (3, 5))
sage: GSD.guruswami_sudan_decoding_radius(n_k=(n, k), l=7)
(111, (4, 7))
```

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

```
sage: GSD.guruswami_sudan_decoding_radius(n_k=(n, k), s=2, l=6)
(92, (2, 6))
```

interpolation_algorithm()

Return the interpolation algorithm that will be used.

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

EXAMPLES:

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

list_size()

Return the list size parameter of `self`.

EXAMPLES:

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

multiplicity()

Return the multiplicity parameter of `self`.

EXAMPLES:

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

parameters()

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

EXAMPLES:

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

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

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

INPUT:

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

OUTPUT:

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

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

EXAMPLES:

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

Another example with a bigger decoding radius:

```
sage: tau, n, k = 118, 250, 70
sage: GSD.parameters_given_tau(tau, n_k=(n, k))
(47, 89)
```

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

```
sage: tau = 200
sage: GSD.parameters_given_tau(tau, n_k=(n, k))
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
ValueError: The decoding radius must be less than
the Johnson radius (which is 118.66)
```

rootfinding_algorithm()

Return the rootfinding algorithm that will be used.

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

EXAMPLES:

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

sage.coding.guruswami_sudan.gs_decoder.**alekhnovich_root_finder**(*p*, *maxd=None*, *precision=None*)

Wrapper for Alekhnovich's algorithm to compute the roots of a polynomial with coefficients in $F[x]$.

sage.coding.guruswami_sudan.gs_decoder.**n_k_params**(*C*, *n_k*)

Internal helper function for the *GRSGuruswamiSudanDecoder* class for allowing to specify either a GRS code C or the length and dimensions n, k directly, in all the static functions.

If neither C or n, k were specified to those functions, an appropriate error should be raised. Otherwise, n, k of the code or the supplied tuple directly is returned.

INPUT:

- `C` – A GRS code or `None`
- `n_k` – A tuple (n, k) being length and dimension of a GRS code, or `None`.

OUTPUT:

- `n_k` – A tuple (n, k) being length and dimension of a GRS code.

EXAMPLES:

```
sage: from sage.coding.guruswami_sudan.gs_decoder import n_k_params
sage: n_k_params(None, (10, 5))
(10, 5)
sage: C = codes.GeneralizedReedSolomonCode(GF(11).list()[:10], 5)
sage: n_k_params(C, None)
(10, 5)
sage: n_k_params(None,None)
Traceback (most recent call last):
...
ValueError: Please provide either the code or its length and dimension
sage: n_k_params(C, (12, 2))
Traceback (most recent call last):
...
ValueError: Please provide only the code or its length and dimension
```

sage.coding.guruswami_sudan.gs_decoder.**roth_ruckenstein_root_finder**(*p*, *maxd=None*, *precision=None*)

Wrapper for Roth-Ruckenstein algorithm to compute the roots of a polynomial with coefficients in $F[x]$.

13.3 Interpolation algorithms for the Guruswami-Sudan decoder

AUTHORS:

- Johan S. R. Nielsen, original implementation (see [Nie] for details)
- David Lucas, ported the original implementation in Sage

sage.coding.guruswami_sudan.interpolation.**gs_interpolation_lee_osullivan**(*points*, *tau*, *parameters*, *wy*)

Returns an interpolation polynomial Q(x,y) for the given input using the module-based algorithm of Lee and O'Sullivan.

This algorithm constructs an explicit $(\ell + 1) \times (\ell + 1)$ polynomial matrix whose rows span the $\mathbf{F}_q[x]$ module of all interpolation polynomials. It then runs a row reduction algorithm to find a low-shifted degree vector in this row space, corresponding to a low weighted-degree interpolation polynomial.

INPUT:

- `points` – a list of tuples `(xi, yi)` such that we seek `Q` with `(xi,yi)` being a root of `Q` with multiplicity `s`.
- `tau` – an integer, the number of errors one wants to decode.
- `parameters` – (default: `None`) a pair of integers, where:
  - the first integer is the multiplicity parameter of Guruswami-Sudan algorithm and
  - the second integer is the list size parameter.
- `wy` – an integer, the y-weight, where we seek `Q` of low `(1,wy)` weighted degree.

EXAMPLES:

```
sage: from sage.coding.guruswami_sudan.interpolation import gs_interpolation_lee_
↪osullivan
sage: F = GF(11)
sage: points = [(F(0), F(2)), (F(1), F(5)), (F(2), F(0)), (F(3), F(4)),
....:           (F(4), F(9)), (F(5), F(1)), (F(6), F(9)), (F(7), F(10))]
sage: tau = 1
sage: params = (1, 1)
sage: wy = 1
sage: Q = gs_interpolation_lee_osullivan(points, tau, params, wy)
sage: Q / Q.lc()  # make monic
x^3*y + 2*x^3 - x^2*y + 5*x^2 + 5*x*y - 5*x + 2*y - 4
```

sage.coding.guruswami_sudan.interpolation.**gs_interpolation_linalg**(*points*, *tau*, *parameters*, *wy*)

Compute an interpolation polynomial $Q(x, y)$ for the Guruswami-Sudan algorithm by solving a linear system of equations.

Q is a bivariate polynomial over the field of the points, such that the polynomial has a zero of multiplicity at least s at each of the points, where s is the multiplicity parameter. Furthermore, its `(1, wy)`-weighted degree should be less than `_interpolation_max_weighted_deg(n, tau, wy)`, where `n` is the number of points

INPUT:

- `points` – a list of tuples `(xi, yi)` such that we seek `Q` with `(xi,yi)` being a root of `Q` with multiplicity `s`.
- `tau` – an integer, the number of errors one wants to decode.

- parameters – (default: None) a pair of integers, where:
  - the first integer is the multiplicity parameter of Guruswami-Sudan algorithm and
  - the second integer is the list size parameter.
- wy – an integer, the y-weight, where we seek Q of low (1, wy)-weighted degree.

EXAMPLES:

The following parameters arise from Guruswami-Sudan decoding of an [6,2,5] GRS code over F(11) with multiplicity 2 and list size 4:

```
sage: from sage.coding.guruswami_sudan.interpolation import gs_interpolation_linalg
sage: F = GF(11)
sage: points = [(F(x), F(y))
....:           for (x, y) in [(0, 5), (1, 1), (2, 4), (3, 6), (4, 3), (5, 3)]]
sage: tau = 3
sage: params = (2, 4)
sage: wy = 1
sage: Q = gs_interpolation_linalg(points, tau, params, wy); Q
4*x^5 - 4*x^4*y - 2*x^2*y^3 - x*y^4 + 3*x^4 - 4*x^2*y^2 + 5*y^4 - x^3 + x^2*y
 + 5*x*y^2 - 5*y^3 + 3*x*y - 2*y^2 + x - 4*y + 1
```

We verify that the interpolation polynomial has a zero of multiplicity at least 2 in each point:

```
sage: all( Q(x=a, y=b).is_zero() for (a,b) in points )
True
sage: x,y = Q.parent().gens()
sage: dQdx = Q.derivative(x)
sage: all( dQdx(x=a, y=b).is_zero() for (a,b) in points )
True
sage: dQdy = Q.derivative(y)
sage: all( dQdy(x=a, y=b).is_zero() for (a,b) in points )
True
```

sage.coding.guruswami_sudan.interpolation.**lee_osullivan_module**(*points*, *parameters*, *wy*)

Return the analytically straight-forward basis for the $\mathbf{F}_q[x]$ module containing all interpolation polynomials, as according to Lee and O'Sullivan.

The module is constructed in the following way: Let $R(x)$ be the Lagrange interpolation polynomial through the sought interpolation points (x_i, y_i), i.e. $R(x_i) = y_i$. Let $G(x) = \prod_{i=1}^{n}(x - x_i)$. Then the i'th row of the basis matrix of the module is the coefficient-vector of the following polynomial in $\mathbf{F}_q[x][y]$:

$$P_i(x, y) = G(x)^{[i-s]}(y - R(x))^{i-[i-s]}y^{[i-s]} ,$$

where $[a]$ for real a is a when $a > 0$ and 0 otherwise. It is easily seen that $P_i(x, y)$ is an interpolation polynomial, i.e. it is zero with multiplicity at least s on each of the points (x_i, y_i).

INPUT:

- points – a list of tuples (xi, yi) such that we seek Q with (xi,yi) being a root of Q with multiplicity s.
- parameters – (default: None) a pair of integers, where:
  - the first integer is the multiplicity parameter s of Guruswami-Sudan algorithm and
  - the second integer is the list size parameter.
- wy – an integer, the y-weight, where we seek Q of low (1,wy) weighted degree.

EXAMPLES:

```
sage: from sage.coding.guruswami_sudan.interpolation import lee_osullivan_module
sage: F = GF(11)
sage: points = [(F(0), F(2)), (F(1), F(5)), (F(2), F(0)), (F(3), F(4)),
....:           (F(4), F(9)), (F(5), F(1)), (F(6), F(9)), (F(7), F(10))]
sage: params = (1, 1)
sage: wy = 1
sage: lee_osullivan_module(points, params, wy)
[x^8 + 5*x^7 + 3*x^6 + 9*x^5 + 4*x^4 + 2*x^3 + 9*x   0]
[ 10*x^7 + 4*x^6 + 9*x^4 + 7*x^3 + 2*x^2 + 9*x + 9   1]
```

13.4 Guruswami-Sudan utility methods

AUTHORS:

- Johan S. R. Nielsen, original implementation (see [Nie] for details)
- David Lucas, ported the original implementation in Sage

sage.coding.guruswami_sudan.utils.**gilt**(x)

Returns the greatest integer smaller than x.

EXAMPLES:

```
sage: from sage.coding.guruswami_sudan.utils import gilt
sage: gilt(43)
42
```

It works with any type of numbers (not only integers):

```
sage: gilt(43.041)
43
```

sage.coding.guruswami_sudan.utils.**johnson_radius**(n, d)

Returns the Johnson-radius for the code length n and the minimum distance d.

The Johnson radius is defined as $n - \sqrt{(}n(n - d))$.

INPUT:

- n – an integer, the length of the code
- d – an integer, the minimum distance of the code

EXAMPLES:

```
sage: sage.coding.guruswami_sudan.utils.johnson_radius(250, 181)
-5*sqrt(690) + 250
```

sage.coding.guruswami_sudan.utils.**ligt**(x)

Returns the least integer greater than x.

EXAMPLES:

```
sage: from sage.coding.guruswami_sudan.utils import ligt
sage: ligt(41)
42
```

It works with any type of numbers (not only integers):

```
sage: ligt(41.041)
42
```

sage.coding.guruswami_sudan.utils.**polynomial_to_list**(*p*, *len*)

Returns p as a list of its coefficients of length len.

INPUT:

- p – a polynomial
- len – an integer. If len is smaller than the degree of p, the returned list will be of size degree of p, else it will be of size len.

EXAMPLES:

```
sage: from sage.coding.guruswami_sudan.utils import polynomial_to_list
sage: F.<x> = GF(41)[]
sage: p = 9*x^2 + 8*x + 37
sage: polynomial_to_list(p, 4)
[37, 8, 9, 0]
```

sage.coding.guruswami_sudan.utils.**solve_degree2_to_integer_range**(*a*, *b*, *c*)

Returns the greatest integer range $[i_1, i_2]$ such that $i_1 > x_1$ and $i_2 < x_2$ where x_1, x_2 are the two zeroes of the equation in x: $ax^2 + bx + c = 0$.

If there is no real solution to the equation, it returns an empty range with negative coefficients.

INPUT:

- a, b and c – coefficients of a second degree equation, a being the coefficient of the higher degree term.

EXAMPLES:

```
sage: from sage.coding.guruswami_sudan.utils import solve_degree2_to_integer_range
sage: solve_degree2_to_integer_range(1, -5, 1)
(1, 4)
```

If there is no real solution:

```
sage: solve_degree2_to_integer_range(50, 5, 42)
(-2, -1)
```

CHAPTER

FOURTEEN

AUTOMORPHISM GROUPS OF LINEAR CODES

14.1 Canonical forms and automorphism group computation for linear codes over finite fields

We implemented the algorithm described in [Feu2009] which computes the unique semilinearly isometric code (canonical form) in the equivalence class of a given linear code `C`. Furthermore, this algorithm will return the automorphism group of `C`, too.

The algorithm should be started via a further class `LinearCodeAutGroupCanLabel`. This class removes duplicated columns (up to multiplications by units) and zero columns. Hence, we can suppose that the input for the algorithm developed here is a set of points in $PG(k-1,q)$.

The implementation is based on the class `sage.groups.perm_gps.partn_ref2.refinement_generic.PartitionRefinement_generic`. See the description of this algorithm in `sage.groups.perm_gps.partn_ref2.refinement_generic`. In the language given there, we have to implement the group action of $G = (GL(k,q) \times \mathbf{F}_q^{*n}) \rtimes Aut(\mathbf{F}_q)$ on the set $X = (\mathbf{F}_q^k)^n$ of $k \times n$ matrices over $\mathbf{F}_q$ (with the above restrictions).

The derived class here implements the stabilizers $G_{\Pi^{(I)}(x)}$ of the projections $\Pi^{(I)}(x)$ of x to the coordinates specified in the sequence I. Furthermore, we implement the inner minimization, i.e. the computation of a canonical form of the projection $\Pi^{(I)}(x)$ under the action of $G_{\Pi^{(I^{(i-1)})}(x)}$. Finally, we provide suitable homomorphisms of group actions for the refinements and methods to compute the applied group elements in $G \rtimes S_n$.

The algorithm also uses Jeffrey Leon's idea of maintaining an invariant set of codewords which is computed in the beginning, see `_init_point_hyperplane_incidence()`. An example for such a set is the set of all codewords of weight $\leq w$ for some uniquely defined w. In our case, we interpret the codewords as a set of hyperplanes (via the corresponding information word) and compute invariants of the bipartite, colored derived subgraph of the point-hyperplane incidence graph, see `PartitionRefinementLinearCode._point_refine()` and `PartitionRefinementLinearCode._hyp_refine()`.

Since we are interested in subspaces (linear codes) instead of matrices, our group elements returned in `PartitionRefinementLinearCode.get_transporter()` and `PartitionRefinementLinearCode.get_autom_gens()` will be elements in the group $(\mathbf{F}_q^{*n} \rtimes Aut(\mathbf{F}_q)) \rtimes S_n = (\mathbf{F}_q^{*n} \rtimes (Aut(\mathbf{F}_q) \times S_n)$.

AUTHORS:

- Thomas Feulner (2012-11-15): initial version

REFERENCES:

- [Feu2009]

EXAMPLES:

Get the canonical form of the Simplex code:

```
sage: from sage.coding.codecan.codecan import PartitionRefinementLinearCode
sage: mat = codes.HammingCode(GF(3), 3).dual_code().generator_matrix()
sage: P = PartitionRefinementLinearCode(mat.ncols(), mat)
sage: cf = P.get_canonical_form(); cf
[1 0 0 0 0 1 1 1 1 1 1 1 1]
[0 1 0 1 1 0 0 1 1 2 2 1 2]
[0 0 1 1 2 1 2 1 2 1 2 0 0]
```

The transporter element is a group element which maps the input to its canonical form:

```
sage: cf.echelon_form() == (P.get_transporter() * mat).echelon_form()
True
```

The automorphism group of the input, i.e. the stabilizer under this group action, is returned by generators:

```
sage: P.get_autom_order_permutation() == GL(3, GF(3)).order()/(len(GF(3))-1)
True
sage: A = P.get_autom_gens()
sage: all((a*mat).echelon_form() == mat.echelon_form() for a in A)
True
```

class sage.coding.codecan.codecan.**InnerGroup**

Bases: object

This class implements the stabilizers $G_{\Pi^{(I)}(x)}$ described in sage.groups.perm_gps.partn_ref2.refinement_generic with $G = (GL(k, q) \times \mathbf{F}_q^n) \rtimes Aut(\mathbf{F}_q)$.

Those stabilizers can be stored as triples:

- rank – an integer in $\{0, \dots, k\}$
- row_partition – a partition of $\{0, \dots, k - 1\}$ with discrete cells for all integers $i \geq$ rank.
- frob_pow – an integer s in $\{0, \dots, r - 1\}$ if $q = p^r$

The group $G_{\Pi^{(I)}(x)}$ contains all elements $(A, \varphi, \alpha) \in G$, where

- A is a 2×2 blockmatrix, whose upper left matrix is a $k \times k$ diagonal matrix whose entries $A_{i,i}$ are constant on the cells of the partition row_partition. The lower left matrix is zero. And the right part is arbitrary.
- The support of the columns given by $i \in I$ intersect exactly one cell of the partition. The entry φ_i is equal to the entries of the corresponding diagonal entry of A.
- α is a power of τ^s, where τ denotes the Frobenius automorphism of the finite field $\mathbf{F}_q$ and s = frob_pow.

See [Feu2009] for more details.

column_blocks(*mat*)

Let mat be a matrix which is stabilized by self having no zero columns. We know that for each column of mat there is a uniquely defined cell in self.row_partition having a nontrivial intersection with the support of this particular column.

This function returns a partition (as list of lists) of the columns indices according to the partition of the rows given by self.

EXAMPLES:

```
sage: from sage.coding.codecan.codecan import InnerGroup
sage: I = InnerGroup(3)
```

(continues on next page)

(continued from previous page)

```
sage: mat = Matrix(GF(3), [[0,1,0],[1,0,0], [0,0,1]])
sage: I.column_blocks(mat)
[[1], [0], [2]]
```

get_frob_pow()

Return the power of the Frobenius automorphism which generates the corresponding component of `self`.

EXAMPLES:

```
sage: from sage.coding.codecan.codecan import InnerGroup
sage: I = InnerGroup(10)
sage: I.get_frob_pow()
1
```

class sage.coding.codecan.codecan.**PartitionRefinementLinearCode**

Bases: `PartitionRefinement_generic`

See *sage.coding.codecan.codecan*.

EXAMPLES:

```
sage: from sage.coding.codecan.codecan import PartitionRefinementLinearCode
sage: mat = codes.HammingCode(GF(3), 3).dual_code().generator_matrix()
sage: P = PartitionRefinementLinearCode(mat.ncols(), mat)
sage: cf = P.get_canonical_form(); cf
[1 0 0 0 0 1 1 1 1 1 1 1 1]
[0 1 0 1 1 0 0 1 1 2 2 1 2]
[0 0 1 1 2 1 2 1 2 1 2 0 0]
```

```
sage: cf.echelon_form() == (P.get_transporter() * mat).echelon_form()
True
```

```
sage: P.get_autom_order_permutation() == GL(3, GF(3)).order()/(len(GF(3))-1)
True
sage: A = P.get_autom_gens()
sage: all((a*mat).echelon_form() == mat.echelon_form() for a in A)
True
```

get_autom_gens()

Return generators of the automorphism group of the initial matrix.

EXAMPLES:

```
sage: from sage.coding.codecan.codecan import PartitionRefinementLinearCode
sage: mat = codes.HammingCode(GF(3), 3).dual_code().generator_matrix()
sage: P = PartitionRefinementLinearCode(mat.ncols(), mat)
sage: A = P.get_autom_gens()
sage: all((a*mat).echelon_form() == mat.echelon_form() for a in A)
True
```

get_autom_order_inner_stabilizer()

Return the order of the stabilizer of the initial matrix under the action of the inner group G.

EXAMPLES:

```
sage: from sage.coding.codecan.codecan import PartitionRefinementLinearCode
sage: mat = codes.HammingCode(GF(3), 3).dual_code().generator_matrix()
sage: P = PartitionRefinementLinearCode(mat.ncols(), mat)
sage: P.get_autom_order_inner_stabilizer()
2
sage: mat2 = Matrix(GF(4, 'a'), [[1,0,1], [0,1,1]])
sage: P2 = PartitionRefinementLinearCode(mat2.ncols(), mat2)
sage: P2.get_autom_order_inner_stabilizer()
6
```

get_canonical_form()

Return the canonical form for this matrix.

EXAMPLES:

```
sage: from sage.coding.codecan.codecan import PartitionRefinementLinearCode
sage: mat = codes.HammingCode(GF(3), 3).dual_code().generator_matrix()
sage: P1 = PartitionRefinementLinearCode(mat.ncols(), mat)
sage: CF1 = P1.get_canonical_form()
sage: s = SemimonomialTransformationGroup(GF(3), mat.ncols()).an_element()
sage: P2 = PartitionRefinementLinearCode(mat.ncols(), s*mat)
sage: CF1 == P2.get_canonical_form()
True
```

get_transporter()

Return the transporter element, mapping the initial matrix to its canonical form.

EXAMPLES:

```
sage: from sage.coding.codecan.codecan import PartitionRefinementLinearCode
sage: mat = codes.HammingCode(GF(3), 3).dual_code().generator_matrix()
sage: P = PartitionRefinementLinearCode(mat.ncols(), mat)
sage: CF = P.get_canonical_form()
sage: t = P.get_transporter()
sage: (t*mat).echelon_form() == CF.echelon_form()
True
```

14.2 Canonical forms and automorphisms for linear codes over finite fields

We implemented the algorithm described in [Feu2009] which computes, a unique code (canonical form) in the equivalence class of a given linear code $C \leq \mathbf{F}_q^n$. Furthermore, this algorithm will return the automorphism group of C, too. You will find more details about the algorithm in the documentation of the class *LinearCodeAutGroupCanLabel*.

The equivalence of codes is modeled as a group action by the group $G = \mathbf{F}_q^{*n} \rtimes (Aut(\mathbf{F}_q) \times S_n)$ on the set of subspaces of $\mathbf{F}_q^n$. The group G will be called the semimonomial group of degree n.

The algorithm is started by initializing the class *LinearCodeAutGroupCanLabel*. When the object gets available, all computations are already finished and you can access the relevant data using the member functions:

- *get_canonical_form()*
- *get_transporter()*

- *get_autom_gens()*

People do also use some weaker notions of equivalence, namely **permutational** equivalence and monomial equivalence (**linear** isometries). These can be seen as the subgroups S_n and $\mathbf{F}_q^{*n} \rtimes S_n$ of G. If you are interested in one of these notions, you can just pass the optional parameter `algorithm_type`.

A second optional parameter P allows you to restrict the group of permutations S_n to a subgroup which respects the coloring given by P.

AUTHORS:

- Thomas Feulner (2012-11-15): initial version

EXAMPLES:

```
sage: from sage.coding.codecan.autgroup_can_label import LinearCodeAutGroupCanLabel
sage: C = codes.HammingCode(GF(3), 3).dual_code()
sage: P = LinearCodeAutGroupCanLabel(C)
sage: P.get_canonical_form().generator_matrix()
[1 0 0 0 0 1 1 1 1 1 1 1 1]
[0 1 0 1 1 0 0 1 1 2 2 1 2]
[0 0 1 1 2 1 2 1 2 1 2 0 0]
sage: LinearCode(P.get_transporter()*C.generator_matrix()) == P.get_canonical_form()
True
sage: A = P.get_autom_gens()
sage: all(LinearCode(a*C.generator_matrix()) == C for a in A)
True
sage: P.get_autom_order() == GL(3, GF(3)).order()
True
```

If the dimension of the dual code is smaller, we will work on this code:

```
sage: C2 = codes.HammingCode(GF(3), 3)
sage: P2 = LinearCodeAutGroupCanLabel(C2)
sage: P2.get_canonical_form().parity_check_matrix() == P.get_canonical_form().generator_
↪matrix()
True
```

There is a specialization of this algorithm to pass a coloring on the coordinates. This is just a list of lists, telling the algorithm which columns do share the same coloring:

```
sage: C = codes.HammingCode(GF(4, 'a'), 3).dual_code()
sage: P = LinearCodeAutGroupCanLabel(C, P=[ [0], [1], list(range(2, C.length())) ])
sage: P.get_autom_order()
864
sage: A = [a.get_perm() for a in P.get_autom_gens()]
sage: H = SymmetricGroup(21).subgroup(A)
sage: H.orbits()
((1,),
 (2,),
 (3, 5, 4),
 (6, 19, 16, 21, 9, 10, 15, 8, 20, 11, 14, 13, 7, 12, 18, 17))
```

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'*)

Bases: object

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. The codes C and D are said to be

- permutational equivalent, if there is some permutation $\pi \in S_n$ such that $(c_{\pi(0)}, \ldots, c_{\pi(n-1)}) \in D$ for all $c \in C$.
- linear equivalent, if there is some permutation $\pi \in S_n$ and a vector $\phi \in \mathbf{F}_q^{*n}$ of units of length n such that $(c_{\pi(0)}\phi_0^{-1}, \ldots, c_{\pi(n-1)}\phi_{n-1}^{-1}) \in D$ for all $c \in C$.
- semilinear equivalent, if there is some permutation $\pi \in S_n$, a vector ϕ of units of length n and a field automorphism α such that $(\alpha(c_{\pi(0)})\phi_0^{-1}, \ldots, \alpha(c_{\pi(n-1)})\phi_{n-1}^{-1}) \in D$ for all $c \in C$.

These are group actions. This class provides an algorithm that will compute a unique representative D in the orbit of the given linear code C. Furthermore, the group element g with $g * C = D$ and the automorphism group of C will be computed as well.

There is also the possibility to restrict the permutational part of this action to a Young subgroup of S_n. This could be achieved by passing a partition P (as a list of lists) of the set $\{0, \ldots, n-1\}$. This is an option which is also available in the computation of a canonical form of a graph, see sage.graphs.generic_graph.GenericGraph.canonical_label().

EXAMPLES:

```
sage: from sage.coding.codecan.autgroup_can_label import LinearCodeAutGroupCanLabel
sage: C = codes.HammingCode(GF(3), 3).dual_code()
sage: P = LinearCodeAutGroupCanLabel(C)
sage: P.get_canonical_form().generator_matrix()
[1 0 0 0 0 1 1 1 1 1 1 1 1]
[0 1 0 1 1 0 0 1 1 2 2 1 2]
[0 0 1 1 2 1 2 1 2 1 2 0 0]
sage: LinearCode(P.get_transporter()*C.generator_matrix()) == P.get_canonical_form()
True
sage: a = P.get_autom_gens()[0]
sage: (a*C.generator_matrix()).echelon_form() == C.generator_matrix().echelon_form()
True
sage: P.get_autom_order() == GL(3, GF(3)).order()
True
```

get_PGammaL_gens()

Return the set of generators translated to the group $P\Gamma L(k, q)$.

There is a geometric point of view of code equivalence. A linear code is identified with the multiset of points in the finite projective geometry $PG(k-1,q)$. The equivalence of codes translates to the natural action of $P\Gamma L(k,q)$. Therefore, we may interpret the group as a subgroup of $P\Gamma L(k,q)$ as well.

EXAMPLES:

```
sage: from sage.coding.codecan.autgroup_can_label import␣
↪LinearCodeAutGroupCanLabel
sage: C = codes.HammingCode(GF(4, 'a'), 3).dual_code()
sage: A = LinearCodeAutGroupCanLabel(C).get_PGammaL_gens()
sage: Gamma = C.generator_matrix()
sage: N = [ x.monic() for x in Gamma.columns() ]
sage: all((g[0]*n.apply_map(g[1])).monic() in N for n in N for g in A)
True
```

get_PGammaL_order()

Return the size of the automorphism group as a subgroup of $P\Gamma L(k,q)$.

There is a geometric point of view of code equivalence. A linear code is identified with the multiset of points in the finite projective geometry $PG(k-1,q)$. The equivalence of codes translates to the natural action of $P\Gamma L(k,q)$. Therefore, we may interpret the group as a subgroup of $P\Gamma L(k,q)$ as well.

EXAMPLES:

```
sage: from sage.coding.codecan.autgroup_can_label import␣
↪LinearCodeAutGroupCanLabel
sage: C = codes.HammingCode(GF(4, 'a'), 3).dual_code()
sage: LinearCodeAutGroupCanLabel(C).get_PGammaL_order() == GL(3, GF(4, 'a')).
↪order()*2/3
True
```

get_autom_gens()

Return a generating set for the automorphism group of the code.

EXAMPLES:

```
sage: from sage.coding.codecan.autgroup_can_label import␣
↪LinearCodeAutGroupCanLabel
sage: C = codes.HammingCode(GF(2), 3).dual_code()
sage: A = LinearCodeAutGroupCanLabel(C).get_autom_gens()
sage: Gamma = C.generator_matrix().echelon_form()
sage: all((g*Gamma).echelon_form() == Gamma for g in A)
True
```

get_autom_order()

Return the size of the automorphism group of the code.

EXAMPLES:

```
sage: from sage.coding.codecan.autgroup_can_label import␣
↪LinearCodeAutGroupCanLabel
sage: C = codes.HammingCode(GF(2), 3).dual_code()
sage: LinearCodeAutGroupCanLabel(C).get_autom_order()
168
```

get_canonical_form()

Return the canonical orbit representative we computed.

EXAMPLES:

```
sage: from sage.coding.codecan.autgroup_can_label import␣
↪LinearCodeAutGroupCanLabel
sage: C = codes.HammingCode(GF(3), 3).dual_code()
sage: CF1 = LinearCodeAutGroupCanLabel(C).get_canonical_form()
sage: s = SemimonomialTransformationGroup(GF(3), C.length()).an_element()
sage: C2 = LinearCode(s*C.generator_matrix())
sage: CF2 = LinearCodeAutGroupCanLabel(C2).get_canonical_form()
sage: CF1 == CF2
True
```

get_transporter()

Return the element which maps the code to its canonical form.

EXAMPLES:

```
sage: from sage.coding.codecan.autgroup_can_label import␣
↪LinearCodeAutGroupCanLabel
sage: C = codes.HammingCode(GF(2), 3).dual_code()
sage: P = LinearCodeAutGroupCanLabel(C)
sage: g = P.get_transporter()
sage: D = P.get_canonical_form()
sage: (g*C.generator_matrix()).echelon_form() == D.generator_matrix().echelon_
↪form()
True
```

CHAPTER

FIFTEEN

BOUNDS FOR PARAMETERS OF LINEAR CODES

15.1 Bounds for parameters of codes

This module provided some upper and lower bounds for the parameters of codes.

AUTHORS:

- David Joyner (2006-07): initial implementation.
- William Stein (2006-07): minor editing of docs and code (fixed bug in elias_bound_asymp)
- David Joyner (2006-07): fixed dimension_upper_bound to return an integer, added example to elias_bound_asymp.
- " (2009-05): removed all calls to Guava but left it as an option.
- Dima Pasechnik (2012-10): added LP bounds.

Let F be a finite set of size q. A subset C of $V = F^n$ is called a code of length n. Often one considers the case where F is a finite field, denoted by $\mathbf{F}_q$. Then V is an F-vector space. A subspace of V (with the standard basis) is called a linear code of length n. If its dimension is denoted k then we typically store a basis of C as a $k \times n$ matrix (the rows are the basis vectors). If $F = \mathbf{F}_2$ 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 elements of $V = F^n$ 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 an element (in the Hamming metric) is $d(\mathbf{v}, \mathbf{0})$, where 0 is a distinguished element of F; in particular it is 0 of the field if F is a field. The minimum distance of a linear code is the smallest non-zero weight of a codeword in C. The relatively minimum distance is denoted

$$\delta = d/n.$$

A linear code with length n, dimension k, and minimum distance d is called an $[n, k, d]_q$-code and n, k, d are called its parameters. A (not necessarily linear) code C with length n, size $M = |C|$, and minimum distance d is called an $(n, M, d)_q$-code (using parentheses instead of square brackets). Of course, $k = \log_q(M)$ for linear codes.

What is the "best" code of a given length? Let $A_q(n, d)$ denote the largest M such that there exists a (n, M, d) code in F^n. Let $B_q(n, d)$ (also denoted $A_q^{lin}(n, d)$) denote the largest k such that there exists a $[n, k, d]$ code in F^n. (Of course, $A_q(n, d) \geq B_q(n, d)$.) Determining $A_q(n, d)$ and $B_q(n, d)$ is one of the main problems in the theory of error-correcting codes. For more details see [HP2003] and [Lin1999].

These quantities related to solving a generalization of the childhood game of "20 questions".

GAME: Player 1 secretly chooses a number from 1 to M (M is large but fixed). Player 2 asks a series of "yes/no questions" in an attempt to determine that number. Player 1 may lie at most e times ($e \geq 0$ is fixed). What is the minimum number of "yes/no questions" Player 2 must ask to (always) be able to correctly determine the number Player 1 chose?

If feedback is not allowed (the only situation considered here), call this minimum number $g(M, e)$.

Lemma: For fixed e and M, $g(M, e)$ is the smallest n such that $A_2(n, 2e + 1) \geq M$.

Thus, solving the solving a generalization of the game of "20 questions" is equivalent to determining $A_2(n, d)$! Using Sage, you can determine the best known estimates for this number in 2 ways:

1. Indirectly, using `best_known_linear_code_www(n, k, F)`, which connects to the website http://www.codetables.de by Markus Grassl;
2. `codesize_upper_bound(n,d,q)`, `dimension_upper_bound(n,d,q)`, and `best_known_linear_code(n, k, F)`.

The output of `best_known_linear_code()`, `best_known_linear_code_www()`, or *`dimension_upper_bound()`* would give only special solutions to the GAME because the bounds are applicable to only linear codes. The output of *`codesize_upper_bound()`* would give the best possible solution, that may belong to a linear or nonlinear code.

This module implements:

- `codesize_upper_bound(n,d,q)`, for the best known (as of May, 2006) upper bound $A(n, d)$ for the size of a code of length n, minimum distance d over a field of size q.
- `dimension_upper_bound(n,d,q)`, an upper bound $B(n, d) = B_q(n, d)$ for the dimension of a linear code of length n, minimum distance d over a field of size q.
- `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, δ, q (assuming n is "infinity").

Todo:

- Johnson bounds for binary codes.
- mrrw2_bound_asymp(delta,q), "second" asymptotic McEliese-Rumsey-Rodemich-Welsh bound for the information rate.

sage.coding.code_bounds.**codesize_upper_bound**(*n*, *d*, *q*, *algorithm=None*)

Return an upper bound on the number of codewords in a (possibly non-linear) code.

This function computes the minimum value of the upper bounds of Singleton, Hamming, Plotkin, and Elias.

If `algorithm="gap"`, then this returns the best known upper bound $A(n, d) = A_q(n, d)$ for the size of a code of length n, minimum distance d over a field of size q. The function first checks for trivial cases (like $d = 1$ or $n = d$), and if the value is in the built-in table. Then it calculates the minimum value of the upper bound using the algorithms of Singleton, Hamming, Johnson, Plotkin and Elias. If the code is binary, $A(n, 2\ell-1) = A(n+1, 2\ell)$, so the function takes the minimum of the values obtained from all algorithms for the parameters $(n, 2\ell - 1)$ and $(n + 1, 2\ell)$. This wraps GUAVA's (i.e. GAP's package Guava) `UpperBound(n, d, q)`.

If `algorithm="LP"`, then this returns the Delsarte (a.k.a. Linear Programming) upper bound.

EXAMPLES:

```
sage: codes.bounds.codesize_upper_bound(10,3,2)
93
sage: codes.bounds.codesize_upper_bound(24,8,2,algorithm="LP")
4096
sage: codes.bounds.codesize_upper_bound(10,3,2,algorithm="gap")  # optional - gap_
↪package_guava
85
sage: codes.bounds.codesize_upper_bound(11,3,4,algorithm=None)
123361
sage: codes.bounds.codesize_upper_bound(11,3,4,algorithm="gap")  # optional - gap_
↪package_guava
123361
sage: codes.bounds.codesize_upper_bound(11,3,4,algorithm="LP")
109226
```

sage.coding.code_bounds.**dimension_upper_bound**(*n*, *d*, *q*, *algorithm=None*)

Return an upper bound for the dimension of a linear code.

Return an upper bound $B(n, d) = B_q(n, d)$ for the dimension of a linear code of length n, minimum distance d over a field of size q.

Parameter `algorithm` has the same meaning as in *codesize_upper_bound()*

EXAMPLES:

```
sage: codes.bounds.dimension_upper_bound(10,3,2)
6
sage: codes.bounds.dimension_upper_bound(30,15,4)
13
sage: codes.bounds.dimension_upper_bound(30,15,4,algorithm="LP")
12
```

sage.coding.code_bounds.**elias_bound_asymp**(*delta*, *q*)

The asymptotic Elias bound for the information rate.

This only makes sense when $0 < \delta < 1 - 1/q$.

EXAMPLES:

```
sage: codes.bounds.elias_bound_asymp(1/4,2)
0.39912396330...
```

sage.coding.code_bounds.**elias_upper_bound**(*n*, *q*, *d*, *algorithm=None*)

Return the Elias upper bound.

Return the Elias upper bound for number of elements in the largest code of minimum distance d in $\mathbf{F}_q^n$, cf. [HP2003]. If `algorithm="gap"`, it wraps GAP's `UpperBoundElias`.

EXAMPLES:

```
sage: codes.bounds.elias_upper_bound(10,2,3)
232
sage: codes.bounds.elias_upper_bound(10,2,3,algorithm="gap")  # optional - gap_
↪package_guava
232
```

sage.coding.code_bounds.**entropy**(*x*, *q=2*)

Compute the entropy at x on the q-ary symmetric channel.

INPUT:

- `x` – real number in the interval $[0, 1]$.
- `q` – (default: 2) integer greater than 1. This is the base of the logarithm.

EXAMPLES:

```
sage: codes.bounds.entropy(0, 2)
0
sage: codes.bounds.entropy(1/5,4).factor()     # optional - sage.symbolic
1/10*(log(3) - 4*log(4/5) - log(1/5))/log(2)
sage: codes.bounds.entropy(1, 3)               # optional - sage.symbolic
log(2)/log(3)
```

Check that values not within the limits are properly handled:

```
sage: codes.bounds.entropy(1.1, 2)
Traceback (most recent call last):
...
ValueError: The entropy function is defined only for x in the interval [0, 1]
sage: codes.bounds.entropy(1, 1)
Traceback (most recent call last):
...
ValueError: The value q must be an integer greater than 1
```

sage.coding.code_bounds.**entropy_inverse**(*x*, *q=2*)

Find the inverse of the q-ary entropy function at the point `x`.

INPUT:

- `x` – real number in the interval $[0, 1]$.
- `q` - (default: 2) integer greater than 1. This is the base of the logarithm.

OUTPUT:

Real number in the interval $[0, 1 - 1/q]$. The function has multiple values if we include the entire interval $[0, 1]$; hence only the values in the above interval is returned.

EXAMPLES:

```
sage: from sage.coding.code_bounds import entropy_inverse
sage: entropy_inverse(0.1)
0.012986862055...
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*)

Return the Gilbert-Varshamov lower bound.

Return the Gilbert-Varshamov lower bound for number of elements in a largest code of minimum distance d in $\mathbf{F}_q^n$. See Wikipedia article Gilbert-Varshamov_bound

EXAMPLES:

```
sage: codes.bounds.gilbert_lower_bound(10,2,3)
128/7
```

sage.coding.code_bounds.**griesmer_upper_bound**(*n*, *q*, *d*, *algorithm=None*)

Return the Griesmer upper bound.

Return the Griesmer upper bound for the number of elements in a largest linear code of minimum distance d in $\mathbf{F}_q^n$, cf. [HP2003]. If the method is "gap", it wraps GAP's `UpperBoundGriesmer`.

The bound states:

$$`n \geq \sum_{i=0}^{k-1} \lceil d/q^i \rceil.`$$

EXAMPLES:

The bound is reached for the ternary Golay codes:

```
sage: codes.bounds.griesmer_upper_bound(12,3,6)
729
sage: codes.bounds.griesmer_upper_bound(11,3,5)
729
```

```
sage: codes.bounds.griesmer_upper_bound(10,2,3)
128
sage: codes.bounds.griesmer_upper_bound(10,2,3,algorithm="gap")  # optional - gap_
↪package_guava
128
```

sage.coding.code_bounds.**gv_bound_asymp**(*delta*, *q*)

The asymptotic Gilbert-Varshamov bound for the information rate, R.

EXAMPLES:

```
sage: RDF(codes.bounds.gv_bound_asymp(1/4,2))
0.18872187554086...
sage: f = lambda x: codes.bounds.gv_bound_asymp(x,2)
sage: plot(f,0,1)
Graphics object consisting of 1 graphics primitive
```

sage.coding.code_bounds.**gv_info_rate**(*n*, *delta*, *q*)

The Gilbert-Varshamov lower bound for information rate.

The Gilbert-Varshamov lower bound for information rate of a q-ary code of length n and minimum distance $n\delta$.

EXAMPLES:

```
sage: RDF(codes.bounds.gv_info_rate(100,1/4,3))  # abs tol 1e-15
0.36704992608261894
```

sage.coding.code_bounds.**hamming_bound_asymp**(*delta*, *q*)

The asymptotic Hamming bound for the information rate.

EXAMPLES:

```
sage: RDF(codes.bounds.hamming_bound_asymp(1/4,2))
0.456435556800...
sage: f = lambda x: codes.bounds.hamming_bound_asymp(x,2)
sage: plot(f,0,1)
Graphics object consisting of 1 graphics primitive
```

sage.coding.code_bounds.**hamming_upper_bound**(*n*, *q*, *d*)

Return the Hamming upper bound.

Return the Hamming upper bound for number of elements in the largest code of length n and minimum distance d over alphabet of size q.

The Hamming bound (also known as the sphere packing bound) returns an upper bound on the size of a code of length n, minimum distance d, over an alphabet of size q. The Hamming bound is obtained by dividing the contents of the entire Hamming space q^n by the contents of a ball with radius $floor((d-1)/2)$. As all these balls are disjoint, they can never contain more than the whole vector space.

$$M \leq \frac{q^n}{V(n,e)},$$

where M is the maximum number of codewords and $V(n,e)$ is equal to the contents of a ball of radius e. This bound is useful for small values of d. Codes for which equality holds are called perfect. See e.g. [HP2003].

EXAMPLES:

```
sage: codes.bounds.hamming_upper_bound(10,2,3)
93
```

sage.coding.code_bounds.**mrrw1_bound_asymp**(*delta*, *q*)

The first asymptotic McEliese-Rumsey-Rodemich-Welsh bound.

This only makes sense when $0 < \delta < 1 - 1/q$.

EXAMPLES:

```
sage: codes.bounds.mrrw1_bound_asymp(1/4,2)   # abs tol 4e-16
0.3545789026652697
```

`sage.coding.code_bounds.`**`plotkin_bound_asymp`**(*delta*, *q*)

The asymptotic Plotkin bound for the information rate.

This only makes sense when $0 < \delta < 1 - 1/q$.

EXAMPLES:

```
sage: codes.bounds.plotkin_bound_asymp(1/4,2)
1/2
```

`sage.coding.code_bounds.`**`plotkin_upper_bound`**(*n*, *q*, *d*, *algorithm=None*)

Return the Plotkin upper bound.

Return the Plotkin upper bound for the number of elements in a largest code of minimum distance d in $\mathbf{F}_q^n$. More precisely this is a generalization of Plotkin's result for $q = 2$ to bigger q due to Berlekamp.

The `algorithm="gap"` option wraps Guava's `UpperBoundPlotkin`.

EXAMPLES:

```
sage: codes.bounds.plotkin_upper_bound(10,2,3)
192
sage: codes.bounds.plotkin_upper_bound(10,2,3,algorithm="gap")  # optional - gap_
↪package_guava
192
```

`sage.coding.code_bounds.`**`singleton_bound_asymp`**(*delta*, *q*)

The asymptotic Singleton bound for the information rate.

EXAMPLES:

```
sage: codes.bounds.singleton_bound_asymp(1/4,2)
3/4
sage: f = lambda x: codes.bounds.singleton_bound_asymp(x,2)
sage: plot(f,0,1)
Graphics object consisting of 1 graphics primitive
```

`sage.coding.code_bounds.`**`singleton_upper_bound`**(*n*, *q*, *d*)

Return the Singleton upper bound.

Return the Singleton upper bound for number of elements in a largest code of minimum distance d in $\mathbf{F}_q^n$.

This bound is based on the shortening of codes. By shortening an (n, M, d) code $d-1$ times, an $(n-d+1, M, 1)$ code results, with $M \leq q^n - d + 1$. Thus

$$M \leq q^{n-d+1}.$$

Codes that meet this bound are called maximum distance separable (MDS).

EXAMPLES:

```
sage: codes.bounds.singleton_upper_bound(10,2,3)
256
```

`sage.coding.code_bounds.`**`volume_hamming`**(*n*, *q*, *r*)

Return the number of elements in a Hamming ball.

Return the number of elements in a Hamming ball of radius r in $\mathbf{F}_q^n$.

EXAMPLES:

```
sage: codes.bounds.volume_hamming(10,2,3)
176
```

15.2 Delsarte (or linear programming) bounds

This module provides LP upper bounds for the parameters of codes, introduced in by P. Delsarte in [De1973].

The exact LP solver PPL is used by default, ensuring that no rounding/overflow problems occur.

AUTHORS:

- Dmitrii V. (Dima) Pasechnik (2012-10): initial implementation
- Dmitrii V. (Dima) Pasechnik (2015, 2021): minor fixes
- Charalampos Kokkalis (2021): Eberlein polynomials, general Q matrix codes

`sage.coding.delsarte_bounds.`**`delsarte_bound_Q_matrix`**(*q*, *d*, *return_data=False*, *solver='PPL'*, *isinteger=False*)

Delsarte bound on a code with Q matrix `q` and lower bound on min. dist. `d`.

Find the Delsarte bound on a code with Q matrix `q` and lower bound on minimal distance `d`.

INPUT:

- `q` – the Q matrix
- `d` – the (lower bound on) minimal distance of the code
- `return_data` – if `True`, return a triple `(W,LP,bound)`, where `W` is a weights vector, and `LP` the Delsarte upper bound LP; both of them are Sage LP data. `W` need not be a weight distribution of a code.
- `solver` – the LP/ILP solver to be used. Defaults to `PPL`. It is arbitrary precision, thus there will be no rounding errors. With other solvers (see `MixedIntegerLinearProgram` for the list), you are on your own!
- `isinteger` – if `True`, uses an integer programming solver (ILP), rather that an LP solver. Can be very slow if set to `True`.

EXAMPLES:

The bound on dimension of linear $\mathbf{F}_2$-codes of length 10 and minimal distance 6:

```
sage: q_matrix = Matrix([[codes.bounds.krawtchouk(10,2,i,j) for i in range(11)]
....:                     for j in range(11)])
sage: codes.bounds.delsarte_bound_Q_matrix(q_matrix, 6)
2

sage: a,p,val = codes.bounds.delsarte_bound_Q_matrix(q_matrix, 6, return_data=True)
sage: [j for i,j in p.get_values(a).items()]
[1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1]
```

`sage.coding.delsarte_bounds.`**`delsarte_bound_additive_hamming_space`**(*n*, *d*, *q*, *d_star=1*, *q_base=0*, *return_data=False*, *solver='PPL'*, *isinteger=False*)

Find a modified Delsarte bound on additive codes in Hamming space H_q^n of minimal distance d

Find the Delsarte LP bound on F_{q_base}-dimension of additive codes in Hamming space H_q^n of minimal distance d with minimal distance of the dual code at least d_star. If q_base is set to non-zero, then q is a power of q_base, and the code is, formally, linear over F_{q_base}. Otherwise it is assumed that q_base==q.

INPUT:

- n – the code length
- d – the (lower bound on) minimal distance of the code
- q – the size of the alphabet
- d_star – the (lower bound on) minimal distance of the dual code; only makes sense for additive codes.
- q_base – if 0, the code is assumed to be linear. Otherwise, q=q_base^m and the code is linear over F_{q_base}.
- return_data – if True, return a triple (W,LP,bound), where W is a weights vector, and LP the Delsarte bound LP; both of them are Sage LP data. W need not be a weight distribution of a code, or, if isinteger==False, even have integer entries.
- solver – the LP/ILP solver to be used. Defaults to 'PPL'. It is arbitrary precision, thus there will be no rounding errors. With other solvers (see MixedIntegerLinearProgram for the list), you are on your own!
- isinteger – if True, uses an integer programming solver (ILP), rather that an LP solver. Can be very slow if set to True.

EXAMPLES:

The bound on dimension of linear $\mathbf{F}_2$-codes of length 11 and minimal distance 6:

```
sage: codes.bounds.delsarte_bound_additive_hamming_space(11, 6, 2)
3
sage: a,p,val = codes.bounds.delsarte_bound_additive_hamming_space(\
                    11, 6, 2, return_data=True)
sage: [j for i,j in p.get_values(a).items()]
[1, 0, 0, 0, 0, 0, 5, 2, 0, 0, 0, 0]
```

The bound on the dimension of linear $\mathbf{F}_4$-codes of length 11 and minimal distance 3:

```
sage: codes.bounds.delsarte_bound_additive_hamming_space(11,3,4)
8
```

The bound on the $\mathbf{F}_2$-dimension of additive $\mathbf{F}_4$-codes of length 11 and minimal distance 3:

```
sage: codes.bounds.delsarte_bound_additive_hamming_space(11,3,4,q_base=2)
16
```

Such a d_star is not possible:

```
sage: codes.bounds.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_constant_weight_code**(*n*, *d*, *w*, *return_data=False*, *solver='PPL'*, *isinteger=False*)

Find the Delsarte bound on a constant weight code.

Find the Delsarte bound on a constant weight code of weight w, length n, lower bound on minimal distance d.

INPUT:

- n – the code length
- d – the (lower bound on) minimal distance of the code
- w – the weight of the code
- return_data – if True, return a triple (W,LP,bound), where W is a weights vector, and LP the Delsarte upper bound LP; both of them are Sage LP data. W need not be a weight distribution of a code.
- solver – the LP/ILP solver to be used. Defaults to PPL. It is arbitrary precision, thus there will be no rounding errors. With other solvers (see MixedIntegerLinearProgram for the list), you are on your own!
- isinteger – if True, uses an integer programming solver (ILP), rather that an LP solver. Can be very slow if set to True.

EXAMPLES:

The bound on the size of codes of length 17, weight 3, and minimal distance 4:

```
sage: codes.bounds.delsarte_bound_constant_weight_code(17, 4, 3)
45
sage: a, p, val = codes.bounds.delsarte_bound_constant_weight_code(17, 4, 3, return_
↪data=True)
sage: [j for i,j in p.get_values(a).items()]
[21, 70/3]
```

The stricter bound (using ILP) on codes of length 17, weight 3, and minimal distance 4:

```
sage: codes.bounds.delsarte_bound_constant_weight_code(17, 4, 3, isinteger=True)
43
```

sage.coding.delsarte_bounds.**delsarte_bound_hamming_space**(*n*, *d*, *q*, *return_data=False*, *solver='PPL'*, *isinteger=False*)

Find the Delsarte bound on codes in H_q^n of minimal distance d

Find the Delsarte bound [De1973] on the size of codes in the Hamming space H_q^n of minimal distance d.

INPUT:

- n – the code length
- d – the (lower bound on) minimal distance of the code
- q – the size of the alphabet
- return_data – if True, return a triple (W,LP,bound), where W is a weights vector, and LP the Delsarte upper bound LP; both of them are Sage LP data. W need not be a weight distribution of a code.
- solver – the LP/ILP solver to be used. Defaults to PPL. It is arbitrary precision, thus there will be no rounding errors. With other solvers (see MixedIntegerLinearProgram for the list), you are on your own!
- isinteger – if True, uses an integer programming solver (ILP), rather that an LP solver. Can be very slow if set to True.

EXAMPLES:

The bound on the size of the $\mathbf{F}_2$-codes of length 11 and minimal distance 6:

```
sage: codes.bounds.delsarte_bound_hamming_space(11, 6, 2)
12
sage: a, p, val = codes.bounds.delsarte_bound_hamming_space(11, 6, 2, return_
↪data=True)
sage: [j for i,j in p.get_values(a).items()]
[1, 0, 0, 0, 0, 0, 11, 0, 0, 0, 0, 0]
```

The bound on the size of the $\mathbf{F}_2$-codes of length 24 and minimal distance 8, i.e. parameters of the extended binary Golay code:

```
sage: a,p,x = codes.bounds.delsarte_bound_hamming_space(24,8,2,return_data=True)
sage: x
4096
sage: [j for i,j in p.get_values(a).items()]
[1, 0, 0, 0, 0, 0, 0, 0, 759, 0, 0, 0, 2576, 0, 0, 0, 759, 0, 0, 0, 0, 0, 0, 0, 1]
```

The bound on the size of $\mathbf{F}_4$-codes of length 11 and minimal distance 3:

```
sage: codes.bounds.delsarte_bound_hamming_space(11,3,4)
327680/3
```

An improvement of a known upper bound (150) from https://www.win.tue.nl/~aeb/codes/binary-1.html

```
sage: a,p,x = codes.bounds.delsarte_bound_hamming_space(23,10,2,return_data=True,
↪isinteger=True); x # long time
148
sage: [j for i,j in p.get_values(a).items()]                                    #␣
↪long time
[1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 95, 0, 2, 0, 36, 0, 14, 0, 0, 0, 0, 0, 0, 0]
```

Note that a usual LP, without integer variables, won't do the trick

```
sage: codes.bounds.delsarte_bound_hamming_space(23,10,2).n(20)
151.86
```

Such an input is invalid:

```
sage: codes.bounds.delsarte_bound_hamming_space(11,3,-4)
Solver exception: PPL : There is no feasible solution
False
```

`sage.coding.delsarte_bounds.`**`eberlein`**(*n*, *w*, *k*, *u*, *check=True*)

Compute $E_k^{w,n}(x)$, the Eberlein polynomial.

See Wikipedia article Eberlein_polynomials.

It is defined as:

$$E_k^{w,n}(u) = \sum_{j=0}^{k} (-1)^j \binom{u}{j} \binom{w-u}{k-j} \binom{n-w-u}{k-j},$$

INPUT:

- `w, k, x` – arbitrary numbers
- `n` – a nonnegative integer

- check – check the input for correctness. True by default. Otherwise, pass it as it is. Use check=False at your own risk.

EXAMPLES:

```
sage: codes.bounds.eberlein(24,10,2,6)
-9
```

sage.coding.delsarte_bounds.**krawtchouk**(*n*, *q*, *l*, *x*, *check=True*)

Compute $K^{n,q}_l(x)$, the Krawtchouk (a.k.a. Kravchuk) polynomial.

See Wikipedia article Kravchuk_polynomials.

It is defined by the generating function

$$(1+(q-1)z)^{n-x}(1-z)^x = \sum_l K^{n,q}_l(x)z^l$$

and is equal to

$$K^{n,q}_l(x) = \sum_{j=0}^{l}(-1)^j(q-1)^{(l-j)}\binom{x}{j}\binom{n-x}{l-j}.$$

INPUT:

- n, q, x – arbitrary numbers
- l – a nonnegative integer
- check – check the input for correctness. True by default. Otherwise, pass it as it is. Use check=False at your own risk.

See also:

Symbolic Krawtchouk polynomials $\tilde{K}_l(x; n, p)$ which are related by

$$(-q)^l K^{n,q^{-1}}_l(x) = \tilde{K}_l(x; n, 1-q).$$

EXAMPLES:

```
sage: codes.bounds.krawtchouk(24,2,5,4)
2224
sage: codes.bounds.krawtchouk(12300,4,5,6)
567785569973042442072
```

CHAPTER

SIXTEEN

DATABASES FOR CODING THEORY

16.1 Access functions to online databases for coding theory

sage.coding.databases.**best_linear_code_in_codetables_dot_de**(*n*, *k*, *F*, *verbose=False*)

Return the best linear code and its construction as per the web database http://www.codetables.de/

INPUT:

- n - Integer, the length of the code
- k - Integer, the dimension of the code
- F - Finite field, of order 2, 3, 4, 5, 7, 8, or 9
- verbose - Bool (default: False)

OUTPUT:

- An unparsed text explaining the construction of the code.

EXAMPLES:

```
sage: L = codes.databases.best_linear_code_in_codetables_dot_de(72, 36, GF(2))     #␣
↪optional - internet
sage: print(L)                                                                      #␣
↪optional - internet
Construction of a linear code
[72,36,15] over GF(2):
[1]:  [73, 36, 16] Cyclic Linear Code over GF(2)
     CyclicCode of length 73 with generating polynomial x^37 + x^36 + x^34 +
x^33 + x^32 + x^27 + x^25 + x^24 + x^22 + x^21 + x^19 + x^18 + x^15 + x^11 +
x^10 + x^8 + x^7 + x^5 + x^3 + 1
[2]:  [72, 36, 15] Linear Code over GF(2)
     Puncturing of [1] at 1

last modified: 2002-03-20
```

This function raises an IOError if an error occurs downloading data or parsing it. It raises a ValueError if the q input is invalid.

AUTHORS:

- Steven Sivek (2005-11-14)
- David Joyner (2008-03)

sage.coding.databases.**best_linear_code_in_guava**(n, k, F)

Return the linear code of length n, dimension k over field F with the maximal minimum distance which is known to the GAP package GUAVA.

The function uses the tables described in *bounds_on_minimum_distance_in_guava()* to construct this code. This requires the optional GAP package GUAVA.

INPUT:

- n – the length of the code to look up
- k – the dimension of the code to look up
- F – the base field of the code to look up

OUTPUT:

A *LinearCode* which is a best linear code of the given parameters known to GUAVA.

EXAMPLES:

```
sage: codes.databases.best_linear_code_in_guava(10,5,GF(2))    # long time;␣
↪optional - gap_package_guava
[10, 5] linear code over GF(2)
sage: libgap.LoadPackage('guava')                              # long time;␣
↪optional - gap_package_guava
...
sage: libgap.BestKnownLinearCode(10,5,libgap.GF(2))            # long time;␣
↪optional - gap_package_guava
a linear [10,5,4]2..4 shortened code
```

This means that the best possible binary linear code of length 10 and dimension 5 is a code with minimum distance 4 and covering radius s somewhere between 2 and 4. Use bounds_on_minimum_distance_in_guava(10,5, GF(2)) for further details.

sage.coding.databases.**bounds_on_minimum_distance_in_guava**(n, k, F)

Compute a lower and upper bound on the greatest minimum distance of a $[n, k]$ linear code over the field F.

This function requires the optional GAP package GUAVA.

The function returns a GAP record with the two bounds and an explanation for each bound. The method Display can be used to show the explanations.

The values for the lower and upper bound are obtained from a table constructed by Cen Tjhai for GUAVA, derived from the table of Brouwer. See http://www.codetables.de/ for the most recent data. These tables contain lower and upper bounds for $q = 2$ (when n <= 257), $q = 3$ (when n <= 243), $q = 4$ (n <= 256). (Current as of 11 May 2006.) For codes over other fields and for larger word lengths, trivial bounds are used.

INPUT:

- n – the length of the code to look up
- k – the dimension of the code to look up
- F – the base field of the code to look up

OUTPUT:

- A GAP record object. See below for an example.

EXAMPLES:

```
sage: gap_rec = codes.databases.bounds_on_minimum_distance_in_guava(10,5,GF(2))  #␣
↪optional - gap_package_guava
sage: gap_rec.Display()                                                          #␣
↪optional - gap_package_guava
rec(
  construction := [ <Operation "ShortenedCode">,
    [ [ <Operation "UUVCode">,
      [ [ <Operation "DualCode">,
      [ [ <Operation "RepetitionCode">, [ 8, 2 ] ] ] ],
      [ <Operation "UUVCode">, [ [ <Operation "DualCode">,
        [ [ <Operation "RepetitionCode">, [ 4, 2 ] ] ] ] ],
        [ <Operation "RepetitionCode">, [ 4, 2 ] ] ] ] ] ] ],
    [ 1, 2, 3, 4, 5, 6 ] ] ],
  k := 5,
  lowerBound := 4,
  lowerBoundExplanation := ...
  n := 10,
  q := 2,
  references := rec(
       ),
  upperBound := 4,
  upperBoundExplanation := ... )
```

sage.coding.databases.**self_orthogonal_binary_codes**(*n*, *k*, *b=2*, *parent=None*, *BC=None*, *equal=False*, *in_test=None*)

Returns a Python iterator which generates a complete set of representatives of all permutation equivalence classes of self-orthogonal binary linear codes of length in `[1..n]` and dimension in `[1..k]`.

INPUT:

- `n` – Integer, maximal length
- `k` – Integer, maximal dimension
- `b` – Integer, requires that the generators all have weight divisible by `b` (if `b=2`, all self-orthogonal codes are generated, and if `b=4`, all doubly even codes are generated). Must be an even positive integer.
- `parent`- - Used in recursion (default: `None`)
- `BC` – Used in recursion (default: `None`)
- `equal` – If `True`, generates only [n, k] codes (default: `False`)
- `in_test` – Used in recursion (default: `None`)

EXAMPLES:

Generate all self-orthogonal codes of length up to 7 and dimension up to 3:

```
sage: for B in codes.databases.self_orthogonal_binary_codes(7,3):
....:     print(B)
[2, 1] linear code over GF(2)
[4, 2] linear code over GF(2)
[6, 3] linear code over GF(2)
[4, 1] linear code over GF(2)
[6, 2] linear code over GF(2)
[6, 2] linear code over GF(2)
```

(continues on next page)

(continued from previous page)

```
[7, 3] linear code over GF(2)
[6, 1] linear code over GF(2)
```

Generate all doubly-even codes of length up to 7 and dimension up to 3:

```
sage: for B in codes.databases.self_orthogonal_binary_codes(7,3,4):
....:    print(B); print(B.generator_matrix())
[4, 1] linear code over GF(2)
[1 1 1 1]
[6, 2] linear code over GF(2)
[1 1 1 1 0 0]
[0 1 0 1 1 1]
[7, 3] linear code over GF(2)
[1 0 1 1 0 1 0]
[0 1 0 1 1 1 0]
[0 0 1 0 1 1 1]
```

Generate all doubly-even codes of length up to 7 and dimension up to 2:

```
sage: for B in codes.databases.self_orthogonal_binary_codes(7,2,4):
....:    print(B); print(B.generator_matrix())
[4, 1] linear code over GF(2)
[1 1 1 1]
[6, 2] linear code over GF(2)
[1 1 1 1 0 0]
[0 1 0 1 1 1]
```

Generate all self-orthogonal codes of length equal to 8 and dimension equal to 4:

```
sage: for B in codes.databases.self_orthogonal_binary_codes(8, 4, equal=True):
....:     print(B); print(B.generator_matrix())
[8, 4] linear code over GF(2)
[1 0 0 1 0 0 0 0]
[0 1 0 0 1 0 0 0]
[0 0 1 0 0 1 0 0]
[0 0 0 0 0 0 1 1]
[8, 4] linear code over GF(2)
[1 0 0 1 1 0 1 0]
[0 1 0 1 1 1 0 0]
[0 0 1 0 1 1 1 0]
[0 0 0 1 0 1 1 1]
```

Since all the codes will be self-orthogonal, b must be divisible by 2:

```
sage: list(codes.databases.self_orthogonal_binary_codes(8, 4, 1, equal=True))
Traceback (most recent call last):
...
ValueError: b (1) must be a positive even integer.
```

16.2 Database of two-weight codes

This module stores a database of two-weight codes.

$q = 2$	$n = 68$	$k = 8$	$w_1 = 32$	$w_2 = 40$	Shared by Eric Chen [ChenDB].
$q = 2$	$n = 85$	$k = 8$	$w_1 = 40$	$w_2 = 48$	Shared by Eric Chen [ChenDB].
$q = 2$	$n = 70$	$k = 9$	$w_1 = 32$	$w_2 = 40$	Found by Axel Kohnert [Koh2007] and shared by Alfred Wassermann.
$q = 2$	$n = 73$	$k = 9$	$w_1 = 32$	$w_2 = 40$	Shared by Eric Chen [ChenDB].
$q = 2$	$n = 219$	$k = 9$	$w_1 = 96$	$w_2 = 112$	Shared by Eric Chen [ChenDB].
$q = 2$	$n = 198$	$k = 10$	$w_1 = 96$	$w_2 = 112$	Shared by Eric Chen [ChenDB].
$q = 3$	$n = 15$	$k = 4$	$w_1 = 9$	$w_2 = 12$	Shared by Eric Chen [ChenDB].
$q = 3$	$n = 55$	$k = 5$	$w_1 = 36$	$w_2 = 45$	Shared by Eric Chen [ChenDB].
$q = 3$	$n = 56$	$k = 6$	$w_1 = 36$	$w_2 = 45$	Shared by Eric Chen [ChenDB].
$q = 3$	$n = 84$	$k = 6$	$w_1 = 54$	$w_2 = 63$	Shared by Eric Chen [ChenDB].
$q = 3$	$n = 98$	$k = 6$	$w_1 = 63$	$w_2 = 72$	Shared by Eric Chen [ChenDB].
$q = 3$	$n = 126$	$k = 6$	$w_1 = 81$	$w_2 = 90$	Shared by Eric Chen [ChenDB].
$q = 3$	$n = 140$	$k = 6$	$w_1 = 90$	$w_2 = 99$	Found by Axel Kohnert [Koh2007] and shared by Alfred Wassermann.
$q = 3$	$n = 154$	$k = 6$	$w_1 = 99$	$w_2 = 108$	Shared by Eric Chen [ChenDB].
$q = 3$	$n = 168$	$k = 6$	$w_1 = 108$	$w_2 = 117$	From [Di2000]
$q = 4$	$n = 34$	$k = 4$	$w_1 = 24$	$w_2 = 28$	Shared by Eric Chen [ChenDB].
$q = 4$	$n = 121$	$k = 5$	$w_1 = 88$	$w_2 = 96$	From [Di2000]
$q = 4$	$n = 132$	$k = 5$	$w_1 = 96$	$w_2 = 104$	From [Di2000]
$q = 4$	$n = 143$	$k = 5$	$w_1 = 104$	$w_2 = 112$	From [Di2000]
$q = 5$	$n = 39$	$k = 4$	$w_1 = 30$	$w_2 = 35$	From Bouyukliev and Simonis ([BS2003], Theorem 4.1)
$q = 5$	$n = 52$	$k = 4$	$w_1 = 40$	$w_2 = 45$	Shared by Eric Chen [ChenDB].
$q = 5$	$n = 65$	$k = 4$	$w_1 = 50$	$w_2 = 55$	Shared by Eric Chen [ChenDB].

REFERENCE:

- [BS2003]

- [ChenDB]
- [Koh2007]
- [Di2000]

CHAPTER

SEVENTEEN

MISCELLANEOUS MODULES

There is at least one module in Sage for source coding in communications theory:

17.1 Huffman encoding

This module implements functionalities relating to Huffman encoding and decoding.

AUTHOR:

- Nathann Cohen (2010-05): initial version.

17.1.1 Classes and functions

class sage.coding.source_coding.huffman.**Huffman**(*source*)

Bases: 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.

In either case, the alphabet must consist of at least two symbols.

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 sorted(h1.encoding_table().items()):
....:     print("'{}' : {}".format(letter, code))
' ' : 00
'T' : 11100
'a' : 0111
'c' : 1010
'e' : 100
'f' : 1011
'h' : 1100
'n' : 1101
'o' : 11101
'r' : 010
's' : 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")
sage: sorted(ft.items())
[(' ', 5),
 ('T', 1),
 ('a', 2),
 ('c', 2),
 ('e', 4),
 ('f', 2),
 ('h', 2),
 ('n', 2),
 ('o', 1),
 ('r', 3),
 ('s', 1),
 ('w', 1),
 ('y', 1)]

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

↪'11100110010001010000111011101101010000111110111111100001110010110101001101101011000010110100110
↪'
```

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)
' eierhffcoeft TfewrnwrTrsc'
```

This does not look like our original string.

Instead of using frequency, we can assign weights to each alphabetic symbol:

```
sage: from sage.coding.source_coding.huffman import Huffman
sage: T = {"a":45, "b":13, "c":12, "d":16, "e":9, "f":5}
sage: H = Huffman(T)
sage: L = ["deaf", "bead", "fab", "bee"]
sage: E = []
sage: for e in L:
....:     E.append(H.encode(e))
....:     print(E[-1])
111110101100
10111010111
11000101
10111011101
sage: D = []
sage: for e in E:
....:     D.append(H.decode(e))
....:     print(D[-1])
deaf
bead
fab
bee
sage: D == L
True
```

decode(*string*)

Decode the given string using the current encoding table.

INPUT:

- `string` – a string of Huffman encodings.

OUTPUT:

- The Huffman decoding of `string`.

EXAMPLES:

This is how a string is encoded and then decoded:

```
sage: from sage.coding.source_coding.huffman import Huffman
sage: str = "Sage is my most favorite general purpose computer algebra system"
sage: h = Huffman(str)
sage: encoded = h.encode(str); encoded

↪'1100001101000101010110000111110100111001110100110110111101111011100111101000010110111010000000
↪'
```

(continues on next page)

(continued from previous page)

```
sage: h.decode(encoded)
'Sage is my most favorite general purpose computer algebra system'
```

encode(*string*)

Encode the given string based on the current encoding table.

INPUT:

- `string` – a string of symbols over an alphabet.

OUTPUT:

- A Huffman encoding of `string`.

EXAMPLES:

This is how a string is encoded and then decoded:

```
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

↪'11000011010001010101100001111101001110011101001101101111011110111001111010000101101110100000
↪'
sage: h.decode(encoded)
'Sage is my most favorite general purpose computer algebra system'
```

encoding_table()

Returns the current encoding table.

INPUT:

- None.

OUTPUT:

- A dictionary associating an alphabetic symbol to a Huffman encoding.

EXAMPLES:

```
sage: from sage.coding.source_coding.huffman import Huffman
sage: str = "Sage is my most favorite general purpose computer algebra system"
sage: h = Huffman(str)
sage: T = sorted(h.encoding_table().items())
sage: for symbol, code in T:
....:     print("{} {}".format(symbol, code))
  101
S 110000
a 1101
b 110001
c 110010
e 010
f 110011
g 0001
i 10000
l 10001
```

(continues on next page)

(continued from previous page)

```
m 0011
n 00000
o 0110
p 0010
r 1110
s 1111
t 0111
u 10010
v 00001
y 10011
```

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                                                       #␣
↪optional - sage.graphs
Digraph on 39 vertices
sage: T.show(figsize=[20,20])                                               #␣
↪optional - sage.graphs sage.plot
```

sage.coding.source_coding.huffman.**frequency_table**(*string*)

Return the frequency table corresponding to the given string.

INPUT:

- `string` – a string of symbols over some alphabet.

OUTPUT:

- A table of frequency of each unique symbol in `string`. If `string` is an empty string, return an empty table.

EXAMPLES:

The frequency table of a non-empty string:

```
sage: from sage.coding.source_coding.huffman import frequency_table
sage: str = "Stop counting my characters!"
sage: T = sorted(frequency_table(str).items())
sage: for symbol, code in T:
....:     print("{} {}".format(symbol, code))
  3
! 1
S 1
```

(continues on next page)

(continued from previous page)

```
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("")
defaultdict(<... 'int'>, {})
```

CHAPTER

EIGHTEEN

INDICES AND TABLES

- Index
- Module Index
- Search Page

PYTHON MODULE INDEX

C

INDEX

A

B

C

D

E

F

G

M

Q

R

S

T

U

V

W

Z