Coding theory and cryptography with Sage
a free and open-source mathematics package

David Joyner

S3CM conference, Soria, Spain, July 2010

Sage homepage: http://www.sagemath.org/
Coding theory and Sage

David Joyner

What is Sage?
What is in Sage?
The CLI
The GUI
Python
What is Python?
for loops
XGCD, lambda, Sage examples
Repeated squaring algorithm
Fibonacci numbers
Classes
Coding theory functionality in Sage
General constructions
Coding theory functions
Coding theory bounds
Coding theory not implemented in Sage
Cryptography
Classical cryptography
Algebraic cryptosystems
LFSRs
Blum-Goldwasser
Miscellaneous topics
Guava
Duursma zeta functions
Self-dual codes
What is Sage?

First, let us take a tour of the website ...

http://www.sagemath.org
What is Sage?

As we saw, **Sage** includes are: Maxima, *pynac* (a *Python*-icized *GiNaC*), and *SymPy* (for calculus and other symbolic computation), *Singular* and *GAP* (for algebra), *R* (for statistics), *Pari* (for number theory), *SciPy* (for numerical computation), *libcrypt* for cryptography, and over 60 more.

*Sage* is based on the mainstream programming language *Python*.

*Sage* is headed by the mathematician William Stein, who is at the University of Washington, in Seattle.
What is in **Sage**?

Other packages available in **Sage**:

<table>
<thead>
<tr>
<th>Basic Arithmetic</th>
<th>GMP, NTL, flint</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command Line</td>
<td>IPython</td>
</tr>
<tr>
<td>Graphical Interface</td>
<td><strong>Sage Notebook</strong></td>
</tr>
<tr>
<td>Graphics</td>
<td>jmol, Matplotlib, ...</td>
</tr>
<tr>
<td>Graph theory</td>
<td>NetworkX</td>
</tr>
<tr>
<td>Interpreted programming language</td>
<td><strong>Python</strong></td>
</tr>
<tr>
<td>Networking</td>
<td>Twisted</td>
</tr>
<tr>
<td>Applied Math.</td>
<td>SciPy, GSL, GLPK, etc.</td>
</tr>
<tr>
<td>Source control system</td>
<td>Mercurial</td>
</tr>
<tr>
<td>Symbolic computation, calculus</td>
<td>SymPy, pynac</td>
</tr>
</tbody>
</table>

To be a component of **Sage**, the software must be: free, open source, robust, high quality, and portable.
Some history: **Sage 0.1 to Sage 4.5**

- Nov 2004: William Stein developed Manin, a precursor to **Sage**.
- Feb 2005: **Sage 0.1**. This included **Pari**.
- Oct 2005, **Sage 0.8**: **GAP** and **Singular** included as standard.
- Feb 2006: **Sage Days 1** workshop, UCSD – **Sage 1.0**
- May-July, 2006 (**Sage 1.3.*) GUI Notebook developed by William Stein, Alex Clemsha and Tom Boothby.
- **Sage Days Workshops** at UCLA, UW, Cambridge, Bristol, Austin, France, San Diego, Seattle, MSRI, Barcelona, ... .
- **Sage** won first prize in the Trophees du Libre (November 2007)
- **Sage** Days 23.5 – Kaiserslautern, Germany on “Singular and Sage integration,” ends July 9, 2010.


**Sage** now has a **huge** range of functionality.
The **Sage Command Line**

When you start **Sage** you will get a small **Sage** banner and then the **Sage** command-line prompt `sage:`.

If you are happy to work at the command line, here is an example of what a short **Sage** session could look like:

```
sage: 2^3
8
sage: t = var("t")
sage: integrate(t*sin(t^2),t)
-cos(t^2)/2
sage: plot[\tab]
```

**Tab-completion** helps you select the command you want with less effort.
The Sage Notebook can be tried out for free by anyone with an internet connection and a good browser at http://www.sagenb.org.

- Connect to Sage running locally or elsewhere (via internet).
- Create embedded graphics (in 2- and 3-d).
- Typeset mathematical expressions using \LaTeX{}.
- Add and delete input, re-executing entire block of commands at once.
- Start and interrupt multiple calculations at once.
- The notebook also works with Maxima, Python, R, Singular, \LaTeX{}, html, etc!
The following screenshot illustrates a Notebook worksheet.

The Sage Notebook

The Sage Notebook

The following screenshot illustrates a Notebook worksheet.
Here are the commands used to create the output in the Notebook session in the above screenshot:

```plaintext
a, b, c, d, x, y = var('a, b, c, d, x, y')
show(solve(a*x^2+b*x+c==0,x))
show(solve(a*x^3+b*x+c==0,x))
solve(a*x+b*y==0,c*x+d*y==0,x,y)
```

Worksheets can be saved (as text or as an sws file in Sage worksheet format), downloaded and emailed (for use by someone else), shared (with colleagues or students), or published (if created on a public Sage server).
If you enjoy playing with the Rubik’s cube, there are several programs for solving the Rubik’s cube in Sage:

You can rotate the Rubik’s cube interactively with your mouse.
Open source philosophy

Sage is Free!

- **Sage** is free software. You can check the algorithms yourself in the source code.
- You can legally serve all its functionality over the web (unlike Magma, Maple, Mathematica, and Matlab).
- Everything in **Sage** is 100% GPL-compatible (except jsmath, which is Apache licensed and runs in browser).
- A lot of work has went into "clarifying" licenses on existing math software (... the Singular/oMalloc story).
- Sometimes we reimplement major algorithms from the ground up because of license problems (... the Nauty/NICE story).
- You can change absolutely anything in **Sage** or any of its dependencies and definitely rebuild or publicly redistribute the result.
Why is open source relevant for mathematics? From a recent interview published in the AMS Notices:

I think we need a symbolic standard to make computer manipulations easier to document and verify. And with all due respect to the free market, perhaps we should not be dependent on commercial software here. An open source project could, perhaps, find better answers to the obvious problems such as availability, bugs, backward compatibility, platform independence, standard libraries, etc. One can learn from the success of \TeX and more specialized software like Macaulay2. I do hope that funding agencies are looking into this.

Andrei Okounkov, 2006 Fields Medalist
Open source software is part of the integrated network fabric which connects and enables our command and control system to work effectively, as people’s lives depend on it.

Open source software is all about “playing nice with others.” It is all about “citizenship.” We need more software collaboration in the DoD. My challenge to you: Become a citizen of the OSS community.

Brig. Gen. N. G. Justice, U. S. Army
Elliptic curves

- All standard algorithms
- p-adic L-functions, complex L-functions
- Heegner points
  - Euler system and Iwasawa-theoretic bounds on Shafarevich-Tate groups
- Group structure over finite fields
- Fast point counting modulo p
- Plotting pictures of elliptic curves
Number theory

Extensive collection of number theory functions. However, for factoring of large integers, only select algorithms are implemented.

<table>
<thead>
<tr>
<th>Example</th>
</tr>
</thead>
</table>
| sage: zeta(0.5+14.0*I)  
0.0222411426099936 - 0.103258123266450*I  
sage: zeta(0.5+14.1*I)  
0.00469840018348919 - 0.0270582823742510*I  
sage: zeta(0.5+14.2*I)  
-0.00681621815859797 + 0.0515969909777821*I  
sage: zeta(0.5+14.3*I)  
-0.0119878243107407 + 0.132231368469266*I |
Probably **Sage** is the best software for this area of computational mathematics.

### Example

```
sage: m = ModularForms(Gamma0(389),6)
sage: m.eisenstein_submodule()
Eisenstein subspace of dimension 2 of Modular Forms
space of dimension 163 for Congruence Subgroup
Gamma0(389) of weight 6 over Rational Field
```
Rings

- Weyl character ring and group rings,

- Algebraic rings: All of the standard rings, such as $\mathbb{Z}$, $\mathbb{Q}$, finite fields $GF(p^k)$, and polynomial, power series and Laurent series rings over any other ring in Sage. Three models of $p$-adic numbers.

  The algebraic closure of $\mathbb{Q}$ and its maximal totally real subfield are also implemented, using intervals.

- Numerical: Real and complex numbers of any fixed precision. Rings that model $\mathbb{R}$ and $\mathbb{C}$ with intervals (interval arithmetic).

- Symbolic rings (for calculus, etc).
Number fields

- Absolute, relative, arbitrary towers (built on Pari but offers much more flexibility)
- Class groups, units, norm equations, maximal orders, reduction mod primes
Commutative Algebra

- **Clean, structured, object-oriented multivariate polynomial rings, coordinate rings of varieties, and ideals**
- **Uses Singular as backend when possible for arithmetic speed and certain algorithms**
- **Groebner Basis computations**
Algebraic geometry

- Varieties and Schemes
- Genus 2 curves and their Jacobians (including fast p-adic point counting algorithms of Kedlaya and Harvey)
- Implicit plotting of curves and surfaces
Linear algebra

- Sparse and dense linear algebra over many rings
- Highly optimized in many cases
- In some cases, possibly the fastest money can buy
Algebraic topology

- The Steenrod algebra
- Simplical complexes and their homology
Graph theory

Sage may overall be the best graph theory software money can buy...

(Thanks to Robert Miller, Nathann Cohen, Emily Kirkman, ...)

Graph theory
Sage and graph theory

Sage

sage: graph_dict = {0: [1,4,5], 1: [2,6], 2: [3,7], 3: [4,2], 4: [0,1], 5: [7, 6], 6: [2], 7: [2]}
sage: G = Graph(graph_dict)
sage: G.show(graph_border=True)

Figure: A graph created using Sage.
Sage has excellent functionality in algebraic combinatorics

- Nicolas Thiery: Mupad-combinat $\mapsto$ Sage-combinat
- Symmetric functions, partitions, Lie algebras and root systems, enumeration, crystals, species, etc.
Group theory

- **Sage** includes GAP
- Weyl groups and Coxeter groups,
- **Sage** includes some "native" permutation group functions
- **Sage** includes "native" abelian group functions
- **Sage** includes a matrix group class, abelian group class and a permutation group class
- **Sage** has some native group cohomology functions

**Sage** lacks a free group class (for example).
Applied math

- **Sage** includes sympy
- **Sage** will include GLPK
- **Sage** includes scipy, numpy, and GSL
- **Sage** includes R
  - Sage can solve some ODEs using maxima or sympy.
Statistics

- **Sage** includes R
- **Sage** includes scipy.stats
- **Sage** includes a finance module
What is Sage?
What is in Sage?
The CLI
The GUI

Python
What is Python?
for loops
XGCD, lambda, Sage examples
Repeated squaring algorithm
Fibonacci numbers
Classes

Coding theory functionality in Sage
General constructions
Coding theory functions
Coding theory bounds

Coding theory not implemented in Sage

Cryptography
Classical cryptography
Algebraic cryptosystems
LFSRs
Blum-Goldwasser

Miscellaneous topics
Guava
Duursma zeta functions
Self-dual codes

Coding theory and Sage
David Joyner
Python is a powerful modern interpreted general programming language, which happens to be very well-suited for scientific programming.

- "Python is fast enough for our site and allows us to produce maintainable features in record times, with a minimum of developers," said Cuong Do, Software Architect, YouTube.com.
“Google has made no secret of the fact they use Python a lot for a number of internal projects. Even knowing that, once I was an employee, I was amazed at how much Python code there actually is in the Google source code system."”, said Guido van Rossum, Google, creator of Python.

“Python plays a key role in our production pipeline. Without it a project the size of Star Wars: Episode II would have been very difficult to pull off. From crowd rendering to batch processing to compositing, Python binds all things together,” said Tommy Burnette, Senior Technical Director, Industrial Light & Magic.
Python is...

- Easy for you to define your own data types and methods on it. Symbolic expressions, graphics types, vector spaces, special functions, whatever.
- Very clean language that results in easy to read code.
- Easy to learn:
  - Free: Python Tutorial [http://docs.python.org/tut/](http://docs.python.org/tut/)
- A huge number of libraries: statistics, networking, databases, bioinformatic, physics, video games, 3d graphics, ...
Python is...

- Easy to use any C/C++ libraries from Python.
- Excellent support for string manipulation and file manipulation.
Coding theory and Sage

David Joyner

What is Sage?
What is in Sage?
The CLI
The GUI

Python
What is Python?
for loops
XGCD, lambda, Sage examples
Repeated squaring algorithm
Fibonacci numbers
Classes

Coding theory functionality in Sage
General constructions
Coding theory functions
Coding theory bounds

Coding theory not implemented in Sage

Cryptography
Classical cryptography
Algebraic cryptosystems
LFSRs
Blum-Goldwasser

Miscellaneous topics
Guava
Duursma zeta functions
Self-dual codes

Figure: Python. xkcd.com license:
http://creativecommons.org/licenses/by-nc/2.5/
Python is...

- The **Python** programming language has a specific syntax (form) and semantics (meaning) which enables it to express computations and data manipulations which can be performed by a computer.
- **Python**’s implementation was started in 1989 by Guido van Rossum, while at CWI.
- **Python** is an “interpreted’ language, i.e., **Python** programs are not directly executed by the host CPU but rather executed by a program known as an “interpreter.”
- The source code of a **Python** program is translated or (partially) compiled to a “bytecode” form of a **Python** “process virtual machine” language.
Because **Python** is dynamically typed, **Python** can figure out the type from the command at run-time.

```python
>>> a = 2012
>>> type(a)
<type 'int'>
>>> b = 2.011
>>> type(b)
<type 'float'>
```
Python is an object-oriented language. Objects are data structures consisting of datafields and methods. Here is an example of a method, `sort`, which applies to the object `L` of type `list`.

```python
>>> L = [2, 1, 4, 3]
>>> type(L)
<type 'list'>
>>> L.sort()
>>> L
[1, 2, 3, 4]
```
Some **Python** data types

Python data types are described in http://docs.python.org/library/datatypes.html.

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Syntax example</th>
</tr>
</thead>
<tbody>
<tr>
<td>str</td>
<td>An immutable sequence of Unicode characters</td>
<td>&quot;string&quot;, &quot;&quot;&quot;\python is great&quot;&quot;, '2012'</td>
</tr>
<tr>
<td>list</td>
<td>Mutable, can contain mixed types</td>
<td>[1.0, 'list', True]</td>
</tr>
<tr>
<td>tuple</td>
<td>Immutable, can contain mixed types</td>
<td>(-1.0, 'tuple', False)</td>
</tr>
<tr>
<td>dict</td>
<td>A mutable group of key and value pairs</td>
<td>{'key1': 1.0, 'key2': False}</td>
</tr>
<tr>
<td>int</td>
<td>immutable fixed precision</td>
<td>42</td>
</tr>
<tr>
<td>float</td>
<td>immutable floating point</td>
<td>2.71828</td>
</tr>
<tr>
<td>bool</td>
<td>An immutable Boolean value</td>
<td>True, False</td>
</tr>
</tbody>
</table>
You can create a dictionary “from scratch,” adding entries “manually” and using `pop` to remove items. Otherwise, a dictionary is like a list.

```python
sage: d = {}
sage: d["1"] = 2
sage: d[2010] = "year"
sage: d
{'1': 2, 2010: 'year'}
sage: type(d)
<type 'dict'>
sage: d.pop(2010)
'year'
sage: d
{'1': 2}
```
**Python keywords**

<table>
<thead>
<tr>
<th>Keyword</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>and</td>
<td>boolean operator</td>
</tr>
<tr>
<td>as</td>
<td>used with import and with</td>
</tr>
<tr>
<td>assert</td>
<td>used for debugging</td>
</tr>
<tr>
<td>break</td>
<td>used in a for/while loop</td>
</tr>
<tr>
<td>class</td>
<td>creates a class</td>
</tr>
<tr>
<td>continue</td>
<td>used in for/while loops</td>
</tr>
<tr>
<td>def</td>
<td>defines a function or method</td>
</tr>
<tr>
<td>del</td>
<td>deletes a reference to an object instance</td>
</tr>
<tr>
<td>elif</td>
<td>used in if ... then statements</td>
</tr>
<tr>
<td>else</td>
<td>used in if ... then statements</td>
</tr>
<tr>
<td>except</td>
<td>used in if ... then statements</td>
</tr>
<tr>
<td>exec</td>
<td>executes a system command</td>
</tr>
<tr>
<td>finally</td>
<td>used in if ... then statements</td>
</tr>
<tr>
<td>for</td>
<td>used in a for loop</td>
</tr>
<tr>
<td>from</td>
<td>used in a for loop</td>
</tr>
<tr>
<td>global</td>
<td>this is a (constant) data type</td>
</tr>
<tr>
<td>if</td>
<td>used in if ... then statements</td>
</tr>
</tbody>
</table>
Python keywords

<table>
<thead>
<tr>
<th>Keyword</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>import</td>
<td>loads a file of data or Python commands</td>
</tr>
<tr>
<td>in</td>
<td>boolean operator on a set</td>
</tr>
<tr>
<td>is</td>
<td>boolean operator</td>
</tr>
<tr>
<td>lambda</td>
<td>defines a simple “one-liner” function</td>
</tr>
<tr>
<td>not</td>
<td>boolean operator</td>
</tr>
<tr>
<td>or</td>
<td>boolean operator</td>
</tr>
<tr>
<td>pass</td>
<td>allows and if-then-elif statement to skip a case</td>
</tr>
<tr>
<td>print</td>
<td>prints the value of the argument</td>
</tr>
<tr>
<td>raise</td>
<td>used for error messages</td>
</tr>
<tr>
<td>return</td>
<td>output of a function</td>
</tr>
<tr>
<td>try</td>
<td>allows you to test for an error</td>
</tr>
<tr>
<td>while</td>
<td>used in a while loop</td>
</tr>
<tr>
<td>with</td>
<td>used in try statements</td>
</tr>
<tr>
<td>yield</td>
<td>used for iterators and generators</td>
</tr>
</tbody>
</table>

(Type import keyword; keyword.kwlist for this list within Python.)
The Zen of Python

The Zen of Python, I
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren’t special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.

Type import this to see the rest!
for loops

A for loop:

```python
>>> for n in range(10,14):
...    if not(n%4 == 2):
...        print n
... 11 12 13
>>> [n for n in range(10,20) if not(n%4==2)] # list comprehension
[11, 12, 13, 15, 16, 17, 19]
```

Note the indentation after the “:”. 
Here is a template of a properly documented Python function.

```python
def my_function(my_input1, my_input2 = my_default_value2):
    """
    Your docstring (see next slide).
    """
    command1  # comment 1
    command2  # comment 2
    return output
```

Documenting appropriately for Sage submissions is required.
Here is a docstring of a properly documented Python function. (Add an AUTHOR(s) field if appropriate).

```
"
Description.

   INPUT:
       my_input1 - the type of the 1st input
       my_input2 - the type of the 2nd input

   OUTPUT:
       the type of the output

   EXAMPLES:
       >>> my_function(arg1,arg2)
       <the output>

   REFERENCES:
       [1] <A Wikipedia article describing the algorithm used>, <url>
       [2] <A book on algorithms describing the algorithm used>,
           <page numbers>
"
```
The example below gives an interactive example requiring user input.

```python
>>> def hello():
...    name = raw_input('What is your name?
')
...    print "Hello World! My name is %s"%name
...  
>>> hello()
What is your name? ### This is input
David ### This is input
Hello World! My name is David ### This is output
```
xgcd

def extended_gcd(a, b):
    """
    Implements Euclid’s extended greatest common divisor algorithm (returns (x, y) s.t. a*x+b*y=gcd(a,b)).
    """
    if a%b == 0:
        return (0, 1)
    else:
        (x, y) = extended_gcd(b, a%b)
    return (y, x-y*int(a/b))
Figure: 11th grade. xkcd.com license: http://creativecommons.org/licenses/by-nc/2.5/
The command \texttt{lambda} allows you to create a one-line function which does not have any local variables except those used to define the function.

\begin{verbatim}
>>> f = lambda x,y: x+y
>>> f(1,2)
3
\end{verbatim}
The function below is in Python but uses Sage classes.

```python
def Hexacode():
    """
    This function returns the [6,3,4] hexacode over GF(4).
    It is an extremal (Hermitian) self-dual Type IV code.
    
    EXAMPLES:
    sage: C = Hexacode()
    sage: C.minimum_distance()
    4
    """
    F = GF(4,"z")
    z = F.gen()
    MS = MatrixSpace(F, 3, 6)
    G = MS([[1, 0, 0, 1, z, z ], [0, 1, 0, z, 1, z ], [0, 0, 1, z, z, 1 ]])
    return LinearCode(G)
```
The **Collatz conjecture** (or the $3n + 1$ conjecture, or as the **Syracuse problem**): Start with any integer $n$ greater than 1. If $n$ is even, we halve it ($n/2$), else we “triple it plus one” ($3n + 1$). According to the conjecture, for all positive numbers this process eventually converges to 1.

For example,

$$10 \rightarrow 5 \rightarrow 16 \rightarrow 8 \rightarrow 4 \rightarrow 2 \rightarrow 1.$$ 

**Exercise**: Write a **Python** function to test this conjecture.
Figure: The Collatz Conjecture. xkcd license:
http://creativecommons.org/licenses/by-nc/2.5/
Repeated squaring algorithm

**Example:** Compute $x^{13}$.

Use the “binary decomposition”: $13 = 1 + 2^2 + 2^3$. First compute $x^1$ (0 steps), then $x^4$ (2 steps, namely $x^2 = x \cdot x$ and $x^4 = x^2 \cdot x^2$), and finally $x^8$ (1 more step, namely $x^8 = x^4 \cdot x^4$). Now (3 more steps)

$$x^{13} = x \cdot x \cdot x^4 \cdot x^8.$$  

In general, we can compute $x^n$ in about $O(\log n)$ steps.
### Repeated squaring algorithm

**Python**

```python
def power(x, n):
    ""
    INPUT:
    x - a number
    n - an integer > 0
    OUTPUT:
    x^n
    EXAMPLES:
    >>> power(3,13)
    1594323
    >>> 3**(13)
    1594323
    ""
    if n == 1:
        return x
    if n%2 == 0:
        return power(x, int(n/2))**2
    if n%2 == 1:
        return x*power(x, int((n-1)/2))**2
```
Leonardo of Pisa, known as Fibonacci, who mentioned the \( \{ f_n \}_{n=0}^{\infty} \) in a book he wrote in the 1200’s. The recursion equation

\[
f_n = f_{n-1} + f_{n-2}, \quad n > 1, \quad f_1 = 1, \quad f_0 = 0,
\]

defines the sequence of **Fibonacci numbers**.
Fibonacci numbers

Computes the $f_n$ very slowly (note: the input $n$ requires $O(\log n)$ bits).

```python
def my_fibonacci(n):
    """
    This is really really slow.
    """
    if n==0:
        return 0
    elif n==1:
        return 1
    else:
        return my_fibonacci(n-1)+my_fibonacci(n-2)
```

In fact, the “complexity” of this algorithm to compute $f_n$ is about equal to $f_n$. This is $O(\phi^n)$, where $\phi = \frac{1+\sqrt{5}}{2}$. (Think about the associated binary tree ...)
Fibonacci numbers

The following is left as an exercise.

Lemma

For each $n > 0$, we have $F^n = \begin{pmatrix} f_{n-1} & f_n \\ f_n & f_{n+1} \end{pmatrix}$, where

$$F = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}.$$ 

Thanks to “repeated squaring,” the “complexity” of this algorithm to compute $f_n$ is about equal to $O(\log n)$. 

Thanks to “repeated squaring,” the “complexity” of this algorithm to compute $f_n$ is about equal to $O(\log n)$.
A simple Python class for a prime finite fields, 1

Prime finite fields in Python.

```python
class FF:
    """
    Implements "prime" finite fields.
    """
    EXAMPLES:
    sage: F = FF(5)
    sage: print F
    Finite field with 5 elements
    sage: F
    FF(5)
    """
    def __init__(self, p):
        self.characteristic = p
```

"""
Continued (note the indentation):

```python
def __repr__(self):
    """ Called to compute the "official" string representation of an object. If at all possible, this should look like a valid Python expression that could be used to recreate an object with the same value.

    EXAMPLES:
    sage: F = FF(5)
    sage: F
    FF(5)
    """
    return "FF(%s)"%self.characteristic
```
Continued (note the indentation):

```python
def __str__(self):
    ""
    Called to compute the "informal" string description of an object.
    
    EXAMPLES:
    sage: F = FF(5)
    sage: print F
    Finite field with 5 elements
    ""
    return "Finite field with %s elements"%self.characteristic
```
A simple Python class for a prime finite fields, 4

```python
def char(self):
    """
    Returns the characteristic of the finite field.
    """
    EXAMPLES:
    sage: FF(5).char()
    5
    """
    return self.characteristic

def __eq__(self, other):
    """
    Returns True of self = other and False otherwise.
    """
    EXAMPLES:
    sage: FF(5) == FF(7)
    False
    """
    p = self.char()
    q = other.char()
    return p == q
```
A simple Python class for a prime finite fields, 5

Continued (note the indentation):

```python
def __call__(self, a):
    ""
    Reduces $a \pmod p$, returning an element of the FF (``coercion'').
    EXAMPLES:
    sage: F = FF(5)
    sage: F(12)
    2
    ""
    p = self.characteristic
    return FFEElement(p, a)
```
A simple Python class for a prime finite fields, 6

```python
def __contains__(self, a):
    """
    Tests if a is in the FF.
    """

    EXAMPLES:
    sage: F = FF(5)
    sage: 2 in F
    True
    sage: 6 in F
    False

    """
    p = self.characteristic
    if a>=0 and a<p:
        return True
    else:
        return False
```
A simple Python class for a prime finite fields, 7

```python
class FFElement:
    ""
    A class for elements of a FF.
    ""
    def __init__(self, p, a):
        self.characteristic = p
        self.element = a%p
        self.base_field = FF(p)
```

A new class:
A simple Python class for a prime finite fields, 8

```
def __repr__(self):
    """
    Called to compute the "official" string representation of an object.
    If at all possible, this should look like a valid Python expression
    that could be used to recreate an object with the same value.
    
    EXAMPLES:
    sage: F = FF(5)
    sage: a = F(3)
    sage: a
    FFElement(5.3)
    """
    return "FFElement(%s, %s)"%(self.characteristic, self.element)
```
A simple Python class for a prime finite fields, 9

Continued (note the indentation):

```python
def __str__(self):
    ""
    Called to compute the "informal" string description of an object.

    EXAMPLES:
    sage: F = FF(5)
    sage: a = F(3)
    sage: print a
    Finite field element 3 in Finite field with 5 elements

    ""
    return "Finite field element %s in %s"%(self.element, self.base_field)
```

Continued (note the indentation):
Continued (note the indentation):

```python
    def __add__(self, other):
        """
        Implements +.

        EXAMPLES:
        sage: F = FF(7)
        sage: a = F(102); b = F(-2)
        sage: a; b; print a; print b; a+b
        FFElement(7, 4)
        FFElement(7, 5)
        Finite field element 4 in Finite field with 7 elements
        Finite field element 5 in Finite field with 7 elements
        2
        """
        p = self.characteristic
        return (self.element+other.element)%p
```
Continued (note the indentation):

```python
def __sub__(self, other):
    ""
    Implements -.
    EXAMPLES:
    sage: F = FF(7)
    sage: a = F(102); b = F(-2)
    sage: a; b; print a; print b; a-b
    Finite field element 4 in Finite field with 7 elements
    Finite field element 5 in Finite field with 7 elements
    6
    ""
    p = self.characteristic
    return (self.element-other.element)%p
```
Continued (note the indentation):

```python
def __mul__(self, other):
    """
    Implements multiplication *.
    EXAMPLES:
    sage: F = FF(7)
    sage: a = F(102); b = F(-2)
    sage: a; b; print a; print b; a*b
    FFElement(7, 4)
    FFElement(7, 5)
    Finite field element 4 in Finite field with 7 elements
    Finite field element 5 in Finite field with 7 elements
    6
    """
    p = self.characteristic
    return (self.element*other.element)%p
```
A simple Python class for a prime finite fields, 13

Continued (note the indentation):

```python
def __div__(self, other):
    """
    Implements /. (Assumes other is not = 0.)
    EXAMPLES:
    sage: F = FF(7)
    sage: a = F(102); b = F(-2)
    sage: a; b; print a; print b; a/b
    FFElement(7, 4)
    Finite field element 4 in Finite field with 7 elements
    Finite field element 5 in Finite field with 7 elements
    5
    """
    p = self.characteristic
    a = self.element
    b = other.element
    return (a*b.__pow__(-1))%p
```
Continued (note the indentation):

```python
def __pow__(self, n):
    """
    Implements ^ or **.
    
    EXAMPLES:
    sage: F = FF(7)
    sage: a = F(102); b = F(-2)
    sage: a; b; a**(-1); b^2
    FFElement(7, 4)
    FFElement(7, 5)
    2
    4
    """
    p = self.characteristic
    a = self.element
    n = int(n)
```
Continued (note the indentation):

```python
if a%p == 0 and not(n<0):
    return 0
if p == 2 and n == -1:
    return a%p
if n == 0:
    return 1
if n == 1:
    return a%p
if n>1:
    if n%2 == 0:
        return ((a.__pow__(int(n/2)))**2)%p # repeated squaring
    if n%2 == 1:
        return (a*(a.__pow__(int(n/2)))**2)%p # repeated squaring
if n == -1:
    return (a.__pow__(p-2))%p
if n<-1:
    return ((a.__pow__(-1))**(-n))%p
return 0 # should never happen
```
def inverse(self):
    """
    Implements the inverse.
    EXAMPLES:
    sage: F = FF(7)
    sage: a = F(102); b = F(-2)
    sage: a.inverse(); b.inverse()
    2
    3
    """
    p = self.characteristic
    a = self.element
    if a%p == 0:
        raise ValueError, "Element must be non-zero."
    if p == 2:
        return a%p
    return (a.__pow__(p-2))%p
A Python class for finite fields

The Python class FF for finite fields $GF(p)$, $p$ prime, is given in above. Modify this class as follows.

**Exercise:** Make your own class that implements the class FFVectorSpace and FFVectors.

- The vector space class must be able to take a prime $p$ (for the characteristic) and an integer $n$ (for the dimension) as arguments.
- The vectors class must be able to take a prime $p$, an integer $n$ and a list of length $n$ of integers (for the coordinates of the vector) as arguments.
- Implement $\equiv$, vector addition, subtraction and scalar multiplication.
- Document your code with standard Python docstrings.
Coding theory functionality in Sage

Coding theory and Sage

David Joyner

What is Sage?
What is in Sage?
The CLI
The GUI

Python
What is Python?
for loops
XGCD, lambda, Sage examples
Repeated squaring algorithm
Fibonacci numbers
Classes

Coding theory functionality in Sage
General constructions
Coding theory functions
Coding theory bounds

Coding theory not implemented in Sage

Cryptography
Classical cryptography
Algebraic cryptosystems
LFSRs
Blum-Goldwasser

Miscellaneous topics
Guava
Duursma zeta functions
Self-dual codes
Basic notation and terms

A code is a linear block code over a finite field $\mathbb{F} = GF(q)$, i.e., a subspace of $\mathbb{F}^n$ with a fixed basis. In the exact sequence

$$0 \rightarrow \mathbb{F}^k \xrightarrow{G} \mathbb{F}^n \xrightarrow{H} \mathbb{F}^{n-k} \rightarrow 0,$$

- $G$ represents a generating matrix,
- $H$ represents a check matrix,
- $C = Image(G) = Kernel(H)$ is the code.
Sage contains GAP but not Guava (which can be loaded as an optional package via sage -i).

<table>
<thead>
<tr>
<th>General constructions</th>
<th>LinearCode, LinearCodeFromCheckMatrix, LinearCodeFromVectorSpace, RandomLinearCode</th>
</tr>
</thead>
</table>

Sage contains GAP but not Guava (which can be loaded as an optional package via sage -i).
LinearCode

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); C
Linear code of length 7, dimension 4 over Finite Field of size 2
sage: C.base_ring()
Finite Field of size 2
sage: C.length(); C.dimension(); C.minimum_distance()
7
4
3
sage: C.weight_distribution()
[1, 0, 0, 7, 7, 0, 0, 1]
LinearCodeFromCheckMatrix

```
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 = LinearCodeFromCheckMatrix(G); C
Linear code of length 7, dimension 3 over Finite Field of size 2
sage: C.length(); C.dimension(); C.minimum_distance()
7
3
4
sage: C.weight_distribution()
[1, 0, 0, 0, 7, 0, 0, 0]
```
LinearCodeFromVectorSpace

```python
sage: V = GF(2)^7
sage: S = V.subspace([[1,1,1,0,0,0,0], [1,0,0,1,1,0,0], [0,1,0,1,0,1,0]])
sage: S.dimension()
3
sage: C = LinearCodeFromVectorSpace(S); C
Linear code of length 7, dimension 3 over Finite Field of size 2
sage: C.length(); C.dimension(); C.minimum_distance()
7
3
3
```
Hamming weight, etc.

Hamming metric is the function $d : \mathbb{F}^n \times \mathbb{F}^n \rightarrow \mathbb{R}$,

$$d(v, w) = |\{i \mid v_i \neq w_i\}| = d(v - w, 0).$$

- the weight is $wt(c) = d(c, 0)$
- minimum distance of $C$ is $d(C) = \min_{c \neq 0} wt(c)$.
- weight distribution (or spectrum) of $C$ is $\text{spec}(C) = (A_0, A_1, \ldots, A_n)$, where

$$A_i = |\{c \in C \mid wt(c) = i\}|.$$
Coding theory functions

| coding theory functions       | spectrum, minimum_distance, characteristic_function, binomialMoment, gen_mat, check_mat, support, decode, standard_form, |

Cryptography
- Classical cryptography
- Algebraic cryptosystems
- LFSRs
- Blum-Goldwasser

Miscellaneous topics
- Guava
- Duursma zeta functions
- Self-dual codes
## Coding theory functions

<table>
<thead>
<tr>
<th>coding theory functions</th>
<th>divisor, genus, random_element, redundancy_matrix, weight Enumerator, chinen_polynomial, zeta_polynomial, zeta_function</th>
</tr>
</thead>
</table>
Corresponding **GAP** functions.

Some associated **GAP** functions

- \texttt{AClosestVectorCombinationsMatFFEVecFFECoords} (for \(d(C)\))
- \texttt{DistancesDistributionMatFFEVecFFE} (for \(\text{spec}(C)\))
- \texttt{WeightVecFFE, DistanceVecFFE} (for \(\text{wt}(v), d(v, w)\))
- \texttt{ConwayPolynomial} (uses database of polynomials used to construct \(GF(q)\))
- \texttt{RandomPrimitivePolynomial}
Examples: `gen_mat, check_mat, support`

```python
sage: C = HammingCode(3,GF(2))
sage: C.gen_mat()
[1 0 0 1 0 1 0]
[0 1 0 1 0 1 1]
[0 0 1 1 0 0 1]
[0 0 0 0 1 1 1]
sage: C.check_mat()
[1 0 0 1 1 0 1]
[0 1 0 1 0 1 1]
[0 0 1 1 1 1 0]
sage: C.support()
[0, 3, 4, 7]
```
Examples: `characteristic_polynomial`, `support`

```python
sage: C = HammingCode(3,GF(2))
sage: Cd = C.dual_code()
sage: Cd.support()
[0, 4]
sage: C.support()
[0, 3, 4, 7]
sage: C.characteristic_polynomial()
-2*x + 8
sage: Cd.characteristic_polynomial()
-4/21*x^3 + 8/3*x^2 - 244/21*x + 16
```
The i-th binomial moment of the $[n, k, d]_q$-code $C$ is

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

where $k_S$ is the dimension of the shortened code $C_{J-S}$, where $J = [1, 2, \ldots, n]$. 

Examples: `binomial_moment`

```python
sage: C = HammingCode(3,GF(2))
sage: C.binomial_moment?  # this gives you the docstring
Type: instancemethod
String Form: <bound method LinearCode.binomial_moment of Linear code of length 7, dimension 4 over Finite Field of size 2>
File: .../sage-4.4.rc0/local/lib/python2.6/site-packages/sage/coding/linear_code.py
Definition: C.binomial_moment(self, i)
Docstring:
    Returns the i-th binomial moment of the [n,k,d]_q-code C:

    .. math:: 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.
```

---

**Coding theory and Sage**

David Joyner

What is Sage?

- What is in Sage?
- The CLI
- The GUI

Python

- What is Python?
- for loops
- XGCD, lambda, Sage examples
- Repeated squaring algorithm
- Fibonacci numbers
- Classes

Coding theory functionality in Sage

- General constructions
- Coding theory functions
- Coding theory bounds

Coding theory not implemented in Sage

Cryptography

- Classical cryptography
- Algebraic cryptosystems
- LFSRs
- Blum-Goldwasser

Miscellaneous topics

- Guava
- Duursma zeta functions
- Self-dual codes
Examples: `binomial_moment`

```python
sage: C = HammingCode(3,GF(2))
sage: C.binomial_moment??  # this gives you the source code listing
<nip>
    n = self.length()
    k = self.dimension()
    d = self.minimum_distance()
    F = self.base_ring()
    q = F.order()
<nip>
    sage: [(i,C.binomial_moment(i)) for i in range(8)]
[(0, 0), (1, 0), (2, 0), (3, 0), (4, 35), (5, 63), (6, 49), (7, 15)]
```
Examples: \texttt{standard\_form}

\begin{verbatim}
sage: C = HammingCode(3,GF(2)); C.gen_mat()
[[1 0 0 1 0 1 0]
 [0 1 0 1 0 1 1]
 [0 0 1 1 0 0 1]
 [0 0 0 0 1 1 1]]
sage: Cs, p = C.standard_form()
sage: Cs
Linear code of length 7, dimension 4 over Finite Field of size 2
sage: p; p in SymmetricGroup(7)
(4,5)
True
sage: Cs.gen_mat()
[[1 0 0 1 1 0]
 [0 1 0 0 1 1 1]
 [0 0 1 0 1 0 1]
 [0 0 0 1 0 1 1]]
\end{verbatim}
**Examples: decode**

```python
sage: C = HammingCode(3,GF(2))
sage: C.decode??
<snip>
```

```
File: ...sage-4.4.rc0/local/lib/python2.6/site-packages/sage/coding/linear_code.py
Definition: C.decode(self, right, method='syndrome')
Source:
    def decode(self, right, method="syndrome"):
        r""
        Decodes the received vector ''right'' to an element 'c' in this code.
        Optional methods are "guava", "nearest neighbor" or "syndrome". The
        ''method="guava"'' wraps GUAVA’s "Decodeword". Hamming codes have a
        special decoding algorithm; otherwise, ""syndrome"" decoding is
        used.
        <snip>
        from decoder import decode
        if method="syndrome" or method="nearest neighbor":
            return decode(self,right)
        <snip>
```

(The bit about Hamming codes is not true for Sage.)
**Examples: decode**

```
sage: decode??
Object 'decode' not found.
sage: from sage.coding.decoder import decode
sage: decode??
<snip>
File: ... sage-4.4.rc0/local/lib/python2.6/site-packages/sage/coding/decoder.py
Definition: decode(C, v, method='syndrome')
Source:
def decode(C, v, method="syndrome"):  
    ""
    The vector v represents a received word, so should be in the same ambient space V as C. Returns an element in C which is closest to v in the Hamming metric.
    
    Methods implemented include "nearest neighbor" (essentially a brute force search) and "syndrome".

<snip>
```
Examples: \texttt{decode}

\begin{Verbatim}
\texttt{sage: C = HammingCode(3,GF(2)); V = GF(2)^7}
\texttt{sage: v = V([1,1,0,1,1,0,1])}
\texttt{sage: v in V; v in C}
\texttt{True False}
\texttt{sage: c = C.decode(v); c; c in C}
\texttt{(1, 0, 0, 1, 1, 0, 1) True}
\end{Verbatim}

`decode` is slow and only a few algorithms have been implemented. This used syndrome decoding.
**Definitions**

**Weight enumerator polynomial** -

\[ A_C(x, y) = \sum_{i=0}^{n} A_i x^{n-i} y^i = x^n + A_d x^{n-d} y^d + \cdots + A_n y^n, \]

where

\[ A_i = |\{ c \in C \mid \text{wt}(c) = i \}| = \# \text{ of codewords wt } i. \]
Weight enumerators

Examples:

- \( W_5(x, y) = x^8 + 14x^4y^4 + y^8 \) is the weight enumerator of the Type II \([8, 4, 4]\) code \( C \) constructed by extending the binary \([7, 4, 3]\) Hamming code by a check bit. This is the smallest Type II code.

- \( W_6(x, y) = x^{24} + 759x^{16}y^8 + 2576x^{12}y^{12} + 759x^8y^{16} + y^{24} \) is the weight enumerator of the extended the binary Golay code with parameters \([24, 12, 8]\).
Example: **weight_enumerator, ...**

**Sage** can verify the fact from the previous slide.

```
sage: C = HammingCode(3,GF(2))
sage: Cx = C.extended_code()
sage: Cx.weight_enumerator()
x^8 + 14*x^4*y^4 + y^8
sage: C = ExtendedBinaryGolayCode()
sage: C.weight_enumerator()
x^24 + 759*x^16*y^8 + 2576*x^12*y^12 + 759*x^8*y^16 + y^24
```

More on these later.
Examples: genus, weight_enumerator, ...

The Duursma zeta function is implemented.

```
sage: C = HammingCode(3,GF(2))
sage: C.genus() # n+1-k-d
1
sage: C.weight_enumerator()
x^7 + 7*x^4*y^3 + 7*x^3*y^4 + y^7
sage: C.zeta_function()
(2/5*T^2 + 2/5*T + 1/5)/(2*T^2 - 3*T + 1)
sage: C.zeta_polynomial()
2/5*T^2 + 2/5*T + 1/5
```

More on these later.
Coding constructions

**Guava** has a lot more constructions, but does not have `galois_closure`.
**Examples:** extended_code

**extended_code simply adds a check-bit at the end.**

```sage
sage: C = HammingCode(3,GF(2))
sage: Cx = C.extended_code()
sage: Cx.is_self_orthogonal()
True
sage: Cx.is_self_dual()
True
sage: Cx.divisor()
4
sage: Cx.spectrum()
[1, 0, 0, 0, 14, 0, 0, 0, 1]
```

More on self-dual codes later.
Examples: direct_sum


```
sage: C = HammingCode(3,GF(2))
sage: C1 = HammingCode(3,GF(2))
sage: C2 = C1.extended_code()
sage: C3 = (C2.direct_sum(C2)).direct_sum(C2)
sage: R.<T> = PolynomialRing(CC, "T")
sage: f = C3.zeta_polynomial(); f = R(f); rts = f.roots()
sage: [abs(z[0]*sqrt(2.0)) for z in rts]
```

```
[0.733550688875582, 1.36323230986647, 1.00000000000000,
1.00000000000000, 1.00000000000000, 1.00000000000000,
1.00000000000000, 1.00000000000000, 1.00000000000000,
1.00000000000000, 1.00000000000000, 1.00000000000000,
1.00000000000000, 1.00000000000000, 1.00000000000000]
```
Komichi’s example, continued.

```python
sage: P1 = list_plot([(z[0].real(),z[0].imag()) for z in f.roots()])
sage: pts = lambda t: [cos(t)/sqrt(2),sin(t)/sqrt(2)]
sage: t = var("t")
sage: P2 = parametric_plot(pts(t),(0,2*pi),linestyle="--",rgbcolor=(1,0,0))
sage: show(P1+P2)
```

Figure: Zeros of the Duursma zeta function of Komichi’s code.
Examples: \texttt{galois\_closure}

\texttt{galois\_closure} of a code $C$ defined over $GF(p^k)$ returns the smallest code defined over $GF(p^k)$ closed under the Galois action of $\text{Gal}(GF(p^k)/GF(p))$.

```
sage: C = HammingCode(3,GF(4,'a'))
sage: Cc = C.galois_closure(GF(2))
sage: C; Cc
Linear code of length 21, dimension 18 over Finite Field in a of size 2^2
Linear code of length 21, dimension 20 over Finite Field in a of size 2^2
sage: C.is_subcode(Cc)
True
sage: Cc.is_galois_closed()
True
```
Automorphism group of a code

What is an automorphism of a code?

Let $S_n$ denote the symmetric group on $n$ letters. The (permutation) automorphism group of a code $C$ of length $n$ is simply the group

$$\text{Aut}(C) = \{ \sigma \in S_n \mid (c_1, \ldots, c_n) \in C \implies (c_{\sigma(1)}, \ldots, c_{\sigma(n)}) \in C \}. $$

There are no known methods for computing these groups which are polynomial time in the length $n$ of $C$. 


Automorphism group of a code

If

(a) $C_1, C_2 \subseteq \mathbb{F}^n$ are codes, and

(b) $\exists \sigma \in S_n$ for which $(c_1, \ldots, c_n) \in C_1 \iff (c_{\sigma(1)}, \ldots, c_{\sigma(n)}) \in C_2$,

then $C_1 \cong C_2$ (i.e., $C_1$ and $C_2$ are permutation equivalent).
Examples: permuted_code

```python
sage: C = HammingCode(3,GF(2))
sage: g = SymmetricGroup(7).random_element(); g
(1,2)(3,7,4)
sage: Cg = C.permuted_code(g)
sage: Cg.is_permutation_equivalent(C)
True
sage: G = C.automorphism_group_binary_code(); G
Permutation Group with generators [(3,4)(5,6), (3,5)(4,6), (2,3)(5,7), (1,2)(5,6)]
sage: g = G("(2,3)(5,7)")
sage: Cg = C.permuted_code(g)
sage: C == Cg
True
```
Examples: punctured, shortened

The code $C^L$ obtained from $C$ by puncturing at the positions in $L$ is the code of length $n - |L|$ consisting of codewords of $C$ which have their $i$-th coordinate deleted if $i \in L$ and left alone if $i \notin L$.

```python
sage: C = HammingCode(3,GF(2))
sage: C.punctured([1,2])
Linear code of length 5, dimension 4 over Finite Field of size 2
sage: C.shortened([1,2])
Linear code of length 5, dimension 2 over Finite Field of size 2
```

The subcode $C(L)$ is 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 $C_L$. 
Examples of most of these have been seen already.
Examples: `permuted_code`

```python
sage: C = HammingCode(3,GF(2))
sage: g = SymmetricGroup(7).random_element(); g
(1,6,4,7,3)(2,5)
sage: C.is_permutation_automorphism(g)
False
sage: Cg = C.permuted_code(g)
sage: Cg.is_permutation_equivalent(C)
True
```

All this is expected behavior.
### Coding constructions

**coding theory functions** (group theoretical)

<table>
<thead>
<tr>
<th>module Composition factors, automorphism group Binary code</th>
</tr>
</thead>
</table>

**module_composition_factors** prints the **GAP record of the** **Meataxe composition factors module** in **Meataxe notation**.
Examples: `module_composition_factors`

```python
sage: C = HammingCode(3,GF(2))
sage: Cx = C.extended_code()
sage: G = Cx.automorphism_group_binary_code()
sage: G.order()
1344
sage: Cx.module_composition_factors(G)
[ rec(
    field := GF(2),
    isMTXModule := true,
    dimension := 1,
    generators := [ [ [ Z(2)^0 ] ], [ [ Z(2)^0 ] ], [ [ Z(2)^0 ] ],
                  [ [ Z(2)^0 ] ], [ [ Z(2)^0 ] ], [ [ Z(2)^0 ] ] ],
    <snip>
    IsIrreducible := true ) ]
```

(A lot is omitted for space reasons.)
## Coding theory functions

| coding theory functions (combinatorial) | assmus_mattson_designs |
A block design: 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.

- If every subset of points of size \(t\) is contained in exactly \(\lambda\) blocks the block design is called a \(t-(v, k, \lambda)\) design.

- When \(\lambda = 1\) then the block design is called a \(S(t, k, v)\) Steiner system.
Assmus and Mattson Theorem: Let $A_0, A_1, \ldots, A_n$ be the weights of the codewords in a binary linear $[n, k, d]$ code $C$, and let $A_0^*, A_1^*, \ldots, A_n^*$ 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 $s \leq d - t$.

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

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

```python
sage: C = HammingCode(3,GF(2))
sage: Cx = C.extended_code()
sage: Cx.assmus_mattson_designs(3)
['weights from C: ', [4, 8],
 'designs from C: ', [[3, (8, 4, 1)], [3, (8, 8, 1)]],
 'weights from C*: ', [4],
 'designs from C*: ', [[3, (8, 4, 1)]]
```
<table>
<thead>
<tr>
<th>Special constructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>BinaryGolayCode,</td>
</tr>
<tr>
<td>ExtendedBinaryGolayCode,</td>
</tr>
<tr>
<td>TernaryGolayCode,</td>
</tr>
<tr>
<td>ExtendedTernaryGolayCode,</td>
</tr>
<tr>
<td>CyclicCode, BCHCode,</td>
</tr>
<tr>
<td>CyclicCodeFromCheckPolynomial,</td>
</tr>
<tr>
<td>DuadicCodeEvenPair,</td>
</tr>
<tr>
<td>DuadicCodeOddPair,</td>
</tr>
<tr>
<td>HammingCode,</td>
</tr>
</tbody>
</table>
### Special constructions (cont.)

| QuadraticResidueCodeEvenPair, QuadraticResidueCodeOddPair, QuadraticResidueCode, ExtendedQuadraticResidueCode, ReedSolomonCode, self_dual_codes_binary, ToricCode, WalshCode |

### sage

David Joyner

What is Sage?
What is in Sage?
The CLI
The GUI

Python
What is Python?
for loops
XGCD, lambda, Sage examples
Repeated squaring algorithm
Fibonacci numbers
Classes

Coding theory functionality in Sage
General constructions
Coding theory functions
Coding theory bounds

Coding theory not implemented in Sage

Cryptography
Classical cryptography
Algebraic cryptosystems
LFSRs
Blum-Goldwasser

Miscellaneous topics
Guava
Duursma zeta functions
Self-dual codes
Examples: ReedSolomonCode

ReedSolomonCode - Also called a “generalized Reed-Solomon code”.

The “narrow” RS codes codes are also cyclic codes; they are part of GUAVA but have not been ported over to native Python/Sage (yet).
Examples: ReedSolomonCode

Let $\mathbb{F} = GF(q)$,
let $n$ and $k$ be such that $1 \leq k \leq n \leq q$,
and pick $n$ distinct elements of $\mathbb{F}$, $\{x_1, x_2, \ldots, x_n\}$.

Define the GRS code by

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

This is an $[n, k, n - k + 1]$ code.
Examples: ReedSolomonCode

```sage
sage: C = ReedSolomonCode(6,4,GF(7)); C
Linear code of length 6, dimension 4 over Finite Field of size 7
sage: C.minimum_distance()
3
sage: F.<a> = GF(3^2,"a")
```
```sage
sage: pts = [0,1,a,a^2,2*a,2*a+1]
sage: len(Set(pts)) == 6 # to make sure there are no duplicates
True
sage: C = ReedSolomonCode(6,4,F,pts); C
Linear code of length 6, dimension 4 over Finite Field in a of size 3^2
sage: C.minimum_distance()
3
```
Examples: **ExtendedTernaryGolayCode**

The permutation automorphism group of the extended ternary Golay code is the Mathieu group \( M_{11} \).

```python
sage: C = ExtendedTernaryGolayCode(); C; C.minimum_distance()
Linear code of length 12, dimension 6 over Finite Field of size 3
6
sage: G = C.permutation_automorphism_group(); G
Permutation Group with generators [(5,7)(6,11)(8,9)(10,12), (4,6)(5,10)(7,8)(9,12),
(3,4)(6,8)(9,11)(10,12), (2,3)(5,7)(8,10)(9,12), (1,2)(5,12)(6,11)(7,10)]
```

```python
sage: G.order(); G.is_simple()
7920
True
```

```python
sage: M11 = MathieuGroup(11); G.is_isomorphic(M11)
True
```

(The full “monomial” automorphism group is larger, but **Sage** lacks the functionality to compute that at this point.)
Examples: ToricCode

ToricCodes can be bad or very good.

```sage
sage: C = ToricCode([[-2,-2],[-1,-2],[-1,-1],[-1,0],[0,-1],
   [0,0],[0,1],[1,-1],[1,0]],GF(5))
sage: C
Linear code of length 16, dimension 9 over Finite Field of size 5
sage: C.minimum_distance()
6
```

(Ask Diego Ruano if you have more questions about this family of codes.)
**Examples:** `self_dual_codes_binary`

Sage has a small database of `self_dual_codes_binary`s.

```python
sage: C = self_dual_codes_binary(10)
sage: C.keys()
['10']
sage: C['10'].keys()
['1', '0']
sage: C['10']['0']
{'Comment': 'No Type II of this length.', 'Type': 'I',
 'code': Linear code of length 10, dimension 5 over Finite Field of size 2,
 'order autgp': 3840, 'spectrum': [1, 0, 5, 0, 10, 0, 10, 0, 5, 0, 1]}
sage: C = self_dual_codes_binary(10)
sage: C = C['10']['0']['code']
sage: C
Linear code of length 10, dimension 5 over Finite Field of size 2
sage: C.divisor()
2
```
<table>
<thead>
<tr>
<th>code bounds</th>
</tr>
</thead>
<tbody>
<tr>
<td>best_known_linear_code_www, bounds_minimum_distance, codesize_upper_bound(n,d,q), dimension_upper_bound(n,d,q), gilbert_lower_bound(n,q,d), plotkin_upper_bound(n,q,d), griesmer_upper_bound(n,q,d), elias_upper_bound(n,q,d)</td>
</tr>
</tbody>
</table>
**Code bounds**

<table>
<thead>
<tr>
<th>code bounds</th>
</tr>
</thead>
<tbody>
<tr>
<td>hamming_upper_bound(n, q, d),</td>
</tr>
<tr>
<td>singleton_upper_bound(n, q, d)</td>
</tr>
<tr>
<td>gv_info_rate(n, delta, q),</td>
</tr>
<tr>
<td>gv_bound_asympt(d, q)</td>
</tr>
<tr>
<td>plotkin_bound_asympt(d, q),</td>
</tr>
<tr>
<td>elias_bound_asympt(d, q)</td>
</tr>
<tr>
<td>hamming_bound_asympt(d, q),</td>
</tr>
<tr>
<td>singleton_bound_asympt(d, q)</td>
</tr>
<tr>
<td>mrrw1_bound_asympt(d, q)</td>
</tr>
</tbody>
</table>
Examples: best_known_linear_code_www (interface with codetables.de since A. Brouwer’s online tables have been disabled).

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

INPUT:

- n – integer, the length of the code
- k – integer, the dimension of the code
- F – finite field, whose field order must be in [2, 3, 4, 5, 7, 8, 9]
- verbose – bool (default=False), print verbose message
Examples: best_known_linear_code_www

```python
sage: L = best_known_linear_code_www(72, 36, GF(2)) # requires internet
sage: print L
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
```
(Manin) There exists a continuous decreasing function

\[ \alpha_q : [0, 1] \rightarrow [0, 1], \]

such that

- \( \alpha_q \) is strictly decreasing on \([0, \frac{q-1}{q}]\),
- \( \alpha_q(0) = 1 \),
- if \( \frac{q-1}{q} \leq x \leq 1 \) then \( \alpha_q(x) = 0 \),
- \( \Sigma_q = \{ (\delta, R) \in [0, 1]^2 \mid 0 \leq R \leq \alpha_q(\delta) \} \).
Asymptotic bounds

Not a single value of $\alpha_q(x)$ is known for $0 < x < \frac{q-1}{q}$! It is not known whether or not the maximum value of the bound, $R = \alpha_q(\delta)$, is attained by a sequence of linear codes. It is not known whether or not $\alpha_q(x)$ is differentiable for $0 < x < \frac{q-1}{q}$, nor is it known if $\alpha_q(x)$ is convex on $0 < x < \frac{q-1}{q}$. 
Asymptotic bounds

(\textit{Gilbert-Varshamov}) \textit{We have}

\[ \alpha_q(x) \geq 1 - x \log_q(q - 1) - x \log_q(x) - (1 - x) \log_q(1 - x). \]

\textit{In other words, for each fixed } \epsilon > 0, \textit{there exists an } (n, k, d)\textit{-code } C \textit{ (which may depend on } \epsilon \textit{) with}

\[ R(C) + \delta(C) \geq 1 - \delta(C) \log_q\left(\frac{q - 1}{q}\right) - \delta(C) \log_q(\delta(C)) - (1 - \delta(C)) \log_q(1 - \delta(C)). \]

\textit{The curve } \left(\delta, 1 - \delta \log_q\left(\frac{q - 1}{q}\right) - \delta \log_q(\delta) - (1 - \delta) \log_q(1 - \delta)\right) \textit{is called the \textit{Gilbert-Varshamov curve}.}
Examples: A plot with `gv_bound_asymp`

Sage has excellent plotting functionality.

```python
sage: f = lambda x: gv_bound_asymp(x,2)
sage: P1 = plot(f,0,1/2)
sage: P2 = list_plot([[3/7,4/7]])
sage: P3 = text('Hamm(7,4,3)', (0.4,0.62), rgbcolor=(0,1,0))
sage: P4 = text('$\star$', (4/8,4/8), rgbcolor=(1,0,0))
sage: P5 = text('Hamm^+(8,4,4)', (0.45,0.4), rgbcolor=(0,1,0))
sage: show(P1+P2+P3+P4+P5)
```
Examples: A plot with `gv_bound_asymp`

Figure: Gilbert-Varshamov curve plotted with the $[7, 4, 3]_2$ and extended $[8, 4, 4]_2$ Hamming codes.
Figure: Gilbert-Varshamov curve and MRRW1 curve plotted with some “good” codes.
Asymptotic bounds

Figure: Plot of the Gilbert-Varshamov (dotted), Elias (red), Plotkin (dashed), Singleton (dash-dotted), Hamming (green), and MRRW (blue) curves using Sage.
Coding theory functionality lacking in Sage

Coding theory functionality in Sage
- General constructions
- Coding theory functions
- Coding theory bounds

Coding theory not yet in Sage

Cryptography
- Classical cryptography
- Algebraic cryptosystems
- LFSRs
- Blum-Goldwasser

Miscellaneous topics
- Guava
- Duursma zeta functions
- Self-dual codes
The only fast implementation of \texttt{minimum\_distance} is in the binary case (and due to Robert Miller).

\textbf{Guava} has a fast implementation of \texttt{MinimumDistance} in the ternary case.

\textbf{Sage} needs a fast implementation of \texttt{minimum\_distance} in the non-binary case.
Automorphism groups for non-binary codes

- The only fast implementation of automorphism groups is in the binary case (and also due to Robert Miller).
- **Sage** needs a fast implementation of automorphism groups is in the non-binary case.
AG codes are implemented in *Singular*, but not yet completely implemented in *Sage*.

There is a module `ag_code` in *Sage*’s `coding` directory but it does not work at present and is not imported.

This needs to be fixed! (See also trac ticket # 8997.)
Decoding

- **Sage** has no special decoding algorithms. (Not even for Hamming codes!)
- **Guava** has some but still is very limited.
- **Sage** needs a lot of work in this area!
Gray codes

- **Sage** has nothing on Gray codes
- A lot of **Python** modules exists that could be submitted.
  
  http://boxen.math.washington.edu/home/wdj/research/coding-theory/graycode.sage

- Lack of developers in this area is the main problem.

More on this later.
Sage has nothing on graph-theoretic cycle or cocycle codes.

Python modules do exist that could be submitted.

http://boxen.math.washington.edu/home/wdj/research/coding-theory/cycle-space.sage

Lack of developers in this area is the main problem.

More on this later.
LDPC codes

- **Sage** has nothing on LDPC codes.
- I think there is C code which possibly could be “wrapped”?
- **Guava** has very limited functionality.
Guava homepage: http://sage.math.washington.edu/home/wdj/guava/

Recent contributors: David Joyner (USNA), Cen Tjhal (Univ Plymouth), Robert Miller (Univ Wash.), Tom Boothby (Univ Wash.).

Joe Fields (S. Conn. St. Univ.) is lead maintainer

Figure: Robert Miller
Figure: Cen Tjhal (“CJ”)
Figure: Tom Boothby
Figure: Joe Fields
Guava is not part of Sage, though it can be loaded easily. Finish porting or “wrapping” everything in Guava to Sage.

```python
sage: install_package("gap_packages")
sage: gap.eval('LoadPackage("guava")')
'true'
sage: C = gap("HammingCode(3,GF(2))")
sage: C.MinimumDistance()
3
```
Coders over finite rings

- **Sage** has nothing on ring codes.
- There is Cython code written (mostly) by Cesar A. Garcia-Vazquez.
- Cesar's code can go in with some extra effort (see trac #6452).

```python
sage: M = Matrix(IntegerModRing(12), [[0, 1, 6, -1],[1, 6, 1, 2],[6, 1, 1, 0]])
sage: C = RingCode(M); C
(4, 1728, 2)-code over the Ring of integers modulo 12
sage: c = C.minimum_weight_codeword(); c
(0, 1, 0, 5)
sage: c in C
True
```
Circuit and cocircuit codes from matroids

- **Sage** has nothing on matroids, much less circuit or cocircuit codes.
- There is some **Python** code which could be submitted.
- Lack of developers in this area is the main problem.

More on this later.
Gray codes

Here's an example after attaching the module `graycode.sage`.

```python
sage: graycode_GF(2,GF(2))
[[0, 0], [1, 0], [1, 1], [0, 1]]
sage: graycode_GF(2,GF(3))
[[0, 0], [1, 0], [2, 0], [2, 1], [1, 1], [0, 1], [0, 2], [1, 2], [2, 2]]
sage: graycode_GF(2,GF(4,"a"))
[[0, 0], [a, 0], [a + 1, 0], [1, 0], [1, a], [a + 1, a],
 [a, a], [0, a], [0, a + 1], [a, a + 1], [a + 1, a + 1],
 [1, a + 1], [1, 1], [a + 1, 1], [a, 1], [0, 1]]
```
Cycle and cocycle codes

It is easy to load and run your own Sage modules. You can even access your own docstrings as usual.

```sage
sage: attach "/Users/wdj/sagefiles/cycle-space.sage"
sage: cycle_code?
<snip>
Definition:       cycle_code(G)
Docstring:
    Returns a "circuit code", called here a cycle code, as described by
    Hakimi-Bredeson, IEE Trans Info Thry 14(1968).

    INPUT:
        G - a simple connected graph with n edges.

    OUTPUT:
        a binary code of length n
<snip>
```
Cycle and cocycle codes

```python
sage: G = graphs.HeawoodGraph()
sage: G.girth()
6
sage: C = cycle_code(G); C; C.minimum_distance()
Linear code of length 21, dimension 8 over Finite Field of size 2
6
```

Figure: Heawood graph of girth 6
Cryptography in Sage

Coding theory and Sage

David Joyner

What is Sage?
What is in Sage?
The CLI
The GUI

Python
What is Python?
for loops
XGCD, lambda, Sage examples
Repeated squaring algorithm
Fibonacci numbers
Classes

Coding theory functionality in Sage
General constructions
Coding theory functions
Coding theory bounds

Coding theory not implemented in Sage

Cryptography
Classical cryptography
Algebraic cryptosystems
LFSRs
Blum-Goldwasser

Miscellaneous topics
Guava
Duursma zeta functions
Self-dual codes

Cryptography in Sage
A cryptosystem is an injection

\[ E : KS \rightarrow \text{Hom}_{\text{Set}}(MS, CS), \]

where

- \( KS \) is the key space,
- \( MS \) is the plaintext (or message) space, and
- \( CS \) is the ciphertext space.
Sage's modules on “classical” ciphers was created by David Kohel and Minh van Nyugen.

- Hill, substitution, transposition, shift cipher, affine cipher and Vigenere cryptosystems are implemented.
Affine cryptosystem

Let $A = \{a_0, a_1, a_2, \ldots, a_{n-1}\}$ be an alphabet. Define an injection $f : A \longrightarrow \mathbb{Z}/n\mathbb{Z}$ given by $f(a_i) = i$.

Set $MS = CS = \mathbb{Z}/n\mathbb{Z} \cong A$

**key space:** $KS = \{(a, b) \in \mathbb{Z}/n\mathbb{Z} \times \mathbb{Z}/n\mathbb{Z} \mid \gcd(a, n) = 1\}$. Let $(a, b) \in KS$.

**Encryption:** For $p \in MS$, define $c \in CS$ by $c \equiv ap + b \pmod{n}$

**Decryption:** For $c \in CS$, define $p \in MS$ by $p \equiv a^{-1}(c - b) \pmod{n}$ where $a^{-1}$ is the inverse of $a$ modulo $n$. 
Affine cryptosystem

```python
sage: A = AffineCryptosystem(AlphabeticStrings())
sage: P = A.encoding("Hello to everyone in Spain!!"); P
HELLOTOEVERYONEINSPAIN
sage: a, b = (3, 7)
sage: C = A.enciphering(a, b, P); C
CTOXMXTSTGBXUTFUJAHFU
sage: L = A.brute_force(C)
sage: sorted(L.items())[30:35]
[((3, 4), IFMMPUPFWFSZQOFJOTQBO), ((3, 5), ZWDDGLGWJWJQFWAFKHSAF),
((3, 6), QNUUXCENNHXWNRHYRJRW), ((3, 7), HELLOTOEVERYONEINSPAIN),
((3, 8), YVCCFKVMPFPEVZJEGRZE)]
sage: L = A.brute_force(C, ranking="chisquare")
sage: L[0]
((3, 7), HELLOTOEVERYONEINSPAIN)
```
**Shift cryptosystem**

```python
sage: S = ShiftCryptosystem(AlphabeticStrings())
sage: P = S.encoding("Shift from Mathematica to Sage!"); P
SHIFTFROMMATHEMATICATOSAGE
sage: K = 3
sage: C = S.enciphering(K, P); C
VKLIWIURPPDWKHPDWLFDWRVDJH
sage: S.enciphering(26-K, C)
SHIFTFROMMATHEMATICATOSAGE
sage: S.deciphering(K, C)
SHIFTFROMMATHEMATICATOSAGE
```
Sage’s modules on algebraic cryptosystems was created by Martin Albrecht and Minh van Nyugen.

- **mini-DES,**
  

- **mini-AES,**
  

- **Small Scale Variants of the AES Polynomial System Generator**

- **Multivariate Polynomial Systems**

See also:
RSA is a deterministic public key encryption algorithm which relies on

- the extended Euclidean algorithm, and
- Euler’s theorem in the special case of a modulus which is a product of two primes.
RSA: Key generation

PKC generalities:

- Two keys - a public key and a private key.
- public key - known to everyone, used for encryption.
- private key - Known only to the receiver, ciphertext can only be decrypted using the private key.
- The security of the RSA cryptosystem relies on that belief that it is computationally infeasible to compute the private key from the public key.
RSA: Key generation

Suppose Alice wants to send a message to Bob using RSA. She says, “Bob, I need to tell you something.”

Bob says, “Hang on a second while I generate the keys.”
RSA: Key generation

Bob then

- chooses two distinct prime numbers \( p \) and \( q \) (only Bob knows these),
- computes \( n = pq \) (\( n \) is used for both the public and private keys),
- computes \( \phi(pq) = (p - 1)(q - 1) \) (\( \phi = \) Euler’s function),
- chooses an integer \( e \) such that \( 1 < e < \phi(pq) \) and \( gcd(e, \phi(pq)) \) (\( e \) is the public key exponent),
- determines \( d \) which satisfies \( de \equiv 1 \pmod{\phi(pq)} \) (\( d \) is the private key exponent).

The **public key** consists of \( (n, e) \).

The **private key** consists of \( (n, d) \).
RSA

Alice wants to send a message to Bob. Bob selects $p = 1009$ and $q = 1013$, so $n = pq = 1022117$. Bob computes $\phi(n) = 1020096$. If he selects $e = 123451$, then he can compute $d = 300019$.

Alice wants to send Bob the message $m = 46577$. She encrypts it using $46577^{123451} \pmod{1022117}$, which is the ciphertext $c = 622474$. 
RSA

What is Sage?
What is Python?
for loops
XGCD, lambda, Sage examples
Repeated squaring algorithm
Fibonacci numbers
Classes
Coding theory functionality in Sage
General constructions
Coding theory functions
Coding theory bounds
Coding theory not implemented in Sage
Cryptography
Classical cryptography
Algebraic cryptosystems
LFSRs
Blum-Goldwasser
Miscellaneous topics
Guava
Duursma zeta functions
Self-dual codes

```
sage: p = next_prime(1000); q = next_prime(1010); n = p*q; n
1022117
sage: k = euler_phi(n); e = 123451
sage: k; xgcd(k, e)
1020096
(1, -36308, 300019)
sage: x = xgcd(k, e)[1]; y = xgcd(k, e)[2]
sage: d = y%k
sage: y*e%k; d*e%k
1
1
sage: m = randint(100, k); m
46577
sage: c = power_mod(m,e,n) # faster than m^e%n
622474
sage: power_mod(c,d,n) # so m was correctly decrypted
46577
```
The **discrete logarithm problem** is the following: Let $G$ be a multiplicative abelian group and let $a, b \in G$. Find $x \in \mathbb{Z}$ such that

$$b^x = a,$$

if it exists.
Discrete logs

```python
sage: p = next_prime(10^30)
sage: F = GF(p)
sage: b = F(2); b.multiplicative_order()
500000000000000000000000000028
sage: b = F(3); b.multiplicative_order()
41666666666666666666666666669
sage: b = F(5); b.multiplicative_order()
1000000000000000000000000000056
sage: a = F.random_element(); a
837776537981704766224734890062
sage: time a.log(b)
CPU times: user 0.04 s, sys: 0.01 s, total: 0.06 s
Wall time: 0.22 s
972953394163188347701109599170
```
Alice and Bob want to share a secret key.

- Alice and Bob agree on a finite cyclic group $G$ and a generating element $g \in G$. ($g$ is assumed to be known by all attackers.) Assume $G$ has order $n$.
- Alice picks a random $a$, $1 < a < n$, and sends $g^a$ to Bob.
- Bob picks a random $b$, $1 < b < n$, and sends $g^b$ to Alice.
- Alice computes $(g^b)^a$.
- Bob computes $(g^a)^b$.
- Both Alice and Bob posses a shared secret key, $g^{ab}$.
Discrete logs

```
sage: G = IntegerModRing(101)
sage: g = G.random_element(); g; g.multiplicative_order()
  3
  100
sage: a = randint(1,50); b = randint(1,50)
sage: a; b
  35
  36
sage: ga = g^a; gb = g^b
sage: ga^b; gb^a == gb^a
  36
  True
```
The Elgamal cryptosystem and the Elgamal digital signature system have been implemented as Sage modules, but not yet submitted to Sage.

Let $q$ be a prime power, $\ell > 1$ be an integer, and let $c_1, \ldots, c_\ell$ are given elements of $GF(q)$.

A linear feedback shift register sequence (LFSR) modulo $p$ of length $\ell$ is a sequence $s_0, s_1, s_2, \ldots \in GF(q)$ such that

1. $s_0, s_1, \ldots, s_{\ell-1}$ are given, and
2. $s_n + c_1 s_{n-1} + c_2 s_{n-2} + \ldots + c_\ell s_{n-\ell} = 0$, $n \geq \ell$. 

LFSRs
LFSRs

Terminology:

- **key** - the list of coefficients $[c_1, c_2, \ldots, c_\ell]$
- **fill** - the list of initial values $s_0, s_1, \ldots, s_{\ell-1}$.
- **connection polynomial** - $c(x) = 1 + c_1 x + \ldots c_\ell x^\ell$. 
**LFSRs**

```
sage: F = GF(2); l = F(1); o = F(0)
sage: fill = [o,l]; key = [1,l]; n = 20
sage: c = lfsr_sequence(key, fill, n); c
[0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1]
sage: f = lfsr_connection_polynomial(c); f
x^2 + x + 1
sage: f.isPrimitive()
True
```

Notice that this Fibonacci sequence mod 2 seems to be periodic with period 3 ($= q^{\deg(c(x))} - 1$).
LFSRs

```
sage: F = GF(3); 1 = F(1); o = F(0)
sage: fill = [o,l]; key = [1,l]; n = 20
sage: c = lfsr_sequence(key, fill, n); c
[0, 1, 1, 2, 0, 2, 2, 1, 0, 1, 1, 2, 0, 2, 2, 1, 0, 1, 1, 2]
sage: f = lfsr_connection_polynomial(c); f
2*x^2 + 2*x + 1
sage: f.isPrimitive()
True
```

Notice that this Fibonacci sequence mod 3 seems to be periodic with period 8 (= \( q^{\deg(c(x))} - 1 \)).
Theorem

Let $S = \{s_i\}$ be a LFSR over $GF(p)$. The period of $S$ is at most $p^k - 1$. It’s period is exactly $P = p^k - 1$ if and only if the characteristic polynomial of $A$

$$A = \begin{pmatrix} 0 & 1 & 0 & \ldots & 0 \\ 0 & 0 & 1 & \ldots & 1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \ldots & 0 & 1 \\ -c_\ell & -c_{\ell-1} & \ldots & -c_1 \end{pmatrix},$$

is irreducible and primitive over $GF(p)$. 
BBS streamcipher

Definition

Let $p, q$ be two distinct prime numbers such that $p \equiv 3 \pmod{4}$ and $q \equiv 3 \pmod{4}$. Let $n = pq$ and let $0 < r < n$ be a random number. We define $x_0$, the first “seed” of the Blum-Blum-Shub pseudorandom number generator as

$$x_0 = r^2 \pmod{n}.$$ 

Each proceeding seed can be defined as

$$x_{i+1} = x_i^2 \pmod{n}.$$ 

The streamcipher, $b = b_1 b_2 \ldots b_t$, is created by setting $b_i = x_i \mod 2$. 
BBS streamcipher

sage: from sage.crypto.stream import blum_blum_shub
sage: p0 = next_prime(1015); q0 = next_prime(1100)
sage: blum_blum_shub(length=50, seed=999, p=p0, q=q0)
11111000110010101001001100100001010100101001011010
sage: from sage.crypto.util import carmichael_lambda as carmichael
sage: carmichael(carmichael(p0*q0))
32004

The last output tells us the maximum possible value of period of the BBS sequence.
Blum-Goldwasser

Alice wants to send a message $m$ to Bob.

Bob generates two distinct prime numbers $p$ and $q$ such that $p \equiv 3 \pmod{4}$, $q \equiv 3 \pmod{4}$.

Bob computes $n = pq$.

Using the extended Euclidean algorithm, Bob computes $a, b$ such that $ap + bq = 1$.

The public key is $n$. The private key is $(p, q, a, b)$.
Blum-Goldwasser

Let $x_0$ be a random QR $\pmod{n}$.

- Plaintext: $m = m_1 m_2 \ldots m_t$ - a binary string of length $t$.
- Let $b = b_1 b_2 \ldots b_t$ be the BBS streamcipher of length $t$ associated to $x_0, n$.
- Ciphertext: $c = b \oplus m$, where $\oplus$ indicates the XOR operation.

Alice sends the ciphertext $c$ along with a number $y = x_0^{2^{t+1}} \pmod{n}$. 
BBS streamcipher

```python
from sage.crypto.public_key.blum_goldwasser import BlumGoldwasser
bg = BlumGoldwasser()
p = 499; q = 547
pubkey = bg.public_key(p, q)
prikey = bg.private_key(p, q)
M = "10011100000100001100"
C = bg.encrypt(M, pubkey, seed=159201)
M0 = bg.decrypt(C, prikey)
```

```
10011100000100001100
```

```
139680
```

```
10011100000100001100
```
NTRU has been partially implemented as a Sage module, but not yet submitted to Sage. 


This would be a welcomed addition!
Miscellaneous topics

Guava, Duursma zeta functions, self-dual codes, cool examples.
A brief tour of **Guava**

**homepage:**

http://sage.math.washington.edu/home/wdj/guava/
Basic Guava functions

- MinimumDistance
- MinimumDistanceLeon  (does not call Leon’s C code)
- MinimumDistanceRandom
- CoveringRadius
- WeightDistribution  (for spec(C), should call Leon?)
- DistancesDistribution  (the distribution of the distances of elements of C to a vector w)
Leon’s code.

Leon’s C code for computing automorphism groups of matrices and designs and linear codes is now GPL’d. Good news:

- it’s GPL’d, optimized C code,
- Joe Fields is working on Guava

Drawbacks:

- it has memory leaks and “home-brewed” finite fields (should use Conway polynomials),
- Guava only interfaces a small part of what it does.

Robert Miller and Tom Boothby have tried to fix up Leon’s code.
Leon’s code.

Guava functions interfacing with Leon’s code:

- `IsEquivalent`,
- `CodeIsomorphism`,
- `AutomorphismGroup`,
- `ConstantWeightSubcode`,
- `PermutationDecode` - see below.
“Unrestricted” codes:

- `ElementsCode`, `RandomCode`
- `HadamardCode` *(assumes Guava has associated Hadamard matrix in it database to construct `HadamardMat(...)`)*
- `ConferenceCode`
- `MOLSCode` *(from mutually orthogonal Latin squares)*
- `NordstromRobinsonCode`
- `GreedyCode`, `LexiCode`
General linear code constructions.

From the check/generator matrix or tables:

- GeneratorMatCode
- CheckMatCodeMutable, CheckMatCode
- RandomLinearCode
- OptimalityCode, BestKnownLinearCode

The last command uses tables developed by Cen Tjhal. (Much larger “best known” codes tables are needed.)
Common linear code constructions.

- HammingCode, ReedMullerCode,
- SrivastavaCode, GeneralizedSrivastavaCode
- FerreroDesignCode (uses SONATA)
- (classical) GoppaCode

Figure: Richard Hamming (1915-1998)
The **covering radius** of a linear code $C$ is the smallest number $r$ with the property that each element $v \in \mathbb{F}^n$ there must be a codeword $c \in C$ with $d(c, v) \leq r$.

- **GabidulinCode**
- **EnlargedGabidulinCode**
- **DavydovCode**
- **TombakCode**
- **EnlargedTombakCode**

Much larger covering codes tables are needed.
Golay codes.

- BinaryGolayCode
- ExtendedBinaryGolayCode
- TernaryGolayCode
- ExtendedTernaryGolayCode

Figure: Marcel Golay (1902-1989)
Cyclic codes.

From the check/generator poly, etc:

- GeneratorPolCode, CheckPolCode
- RootsCode, FireCode
- ReedSolomonCode
- BCHCode, AlternantCode
- QRCODE, QQRCodeNC
- CyclicCodes, NrCyclicCodes

Figure: Irving Reed, Gustave Solomon
Evaluation codes

- EvaluationCode
- GeneralizedReedSolomonCode
- GeneralizedReedMullerCode
- ToricCode
- GoppaCodeClassical
- EvaluationBivariateCode, EvaluationBivariateCodeNC
- OnePointAGCode
This code was once best known:

```gap
gap> C := 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 linear $[49,11,1..39]_{25..38}$ toric code over $GF(8)$

min. dist. = 28.

- Diego Ruano and many others have also searched for other “new and good” toric-like codes, finding many more.
- Choosing the polytope carefully, the code can be constructed to have a large automorphism group.
Decoding methods

**Decode** \((C, r)\) uses syndrome decoding or nearest-neighbor except for:

- Hamming codes (the usual trick),
- GRS codes - see below,
- cyclic codes (error-trapping - sometimes), and
- BCH codes (Sugiyama decoding).
Decoding methods

The default algorithm used for generalized Reed-Solomon codes is the interpolation algorithm. Gao’s decoding method for GRS codes is also available as an option.
Decoding codes obtained from evaluating polynomials at lots of points “should be easy”.

Rough idea: codewords are values of polynomial and \# values known is > \text{deg}(\text{polynomials}), so the vector overdetermines the polynomial. If the number of errors is “small” then the polynomial can still be reconstructed....
**Syntax:** Decodeword( C, r ), where C is a GRS code. This does “interpolation decoding”.

**GeneralizedReedSolomonDecoderGao** is a version which uses an algorithm of Gao.

**GeneralizedReedSolomonListDecoder( C, r, tau )** implements Sudan’s list-decoding algorithm for “low rate” GRS codes. It returns the list of all codewords in C which are a distance of at most $\tau$ from $r$. 
Permutation decoding

Here is the basic idea.

- $C$ is a code, $v \in \mathbb{F}^n$ is a received vector, $G = Aut(C)$ is the perm. automorphism group.
- Assume $C$ is in standard form, with check matrix $H$. 
Permutation decoding

The algorithm runs through the elements $g$ of $G = Aut(C)$, checking if $\text{wt}(H(g \cdot v)) < (d - 1)/2$. If it is then the vector $g \cdot v$ is used to decode $v$: $c = g^{-1} \cdot Gm$ is the decoded word, where $m$ is the information digits part of $g \cdot v$.

If no such $g$ exists then “fail” is returned.

- This generalizes “error-trapping” for decoding cyclic codes,

**Guava functions:** $\text{PermutationDecodeNC}( C, v, G )$, $\text{PermutationDecode}( C, v )$
Sage and Guava

What is Sage?
What is in Sage?
The CLI
The GUI
Python
What is Python?
for loops
XGCD, lambda, Sage examples
Repeated squaring algorithm
Fibonacci numbers
Classes
Coding theory functionality in Sage
General constructions
Coding theory functions
Coding theory bounds
Coding theory not implemented in Sage
Cryptography
Classical cryptography
Algebraic cryptosystems
LFSRs
Blum-Goldwasser
Miscellaneous topics
Guava
Duursma zeta functions
Self-dual codes

In **Sage**, bad news:

- most **GUAVA** functions are not wrapped or ported,
- most Leon functions are not wrapped (nor have they been rewritten in Cython)

Lots of work to be done.
In Sage, computing Duursma zeta functions of codes is implemented.

Figure: Tom Hoeholdt talking to Iwan Duursma at the IMA coding theory conference, May 2007.
Duursma zeta functions

C is an \([n, k, d]_q\) code

\(C^\perp\) is an \([n, k^\perp, d^\perp]_q\) code

Motivated by local CFT, Iwan Duursma introduced the **zeta function** \(Z = Z_C\) associated to \(C\):

\[
Z(T) = \frac{P(T)}{(1 - T)(1 - qT)}, \tag{2}
\]

where \(P(T)\) is a polynomial of degree \(n + 2 - d - d^\perp\), called the **zeta polynomial**.
The *genus* of an \([n, k, d]_q\)-code \(C\) is defined by

\[
\gamma(C) = n + 1 - k - d
\]

\(= \text{“distance code is from being MDS”}.

For AG codes, it often is equal to the genus of the associated curve.

Note that if \(C\) is a self-dual code then its genus satisfies

\[
\gamma = n/2 + 1 - d.
\]
Definitions

Weight enumerator polynomial -

\[ A_C(x, y) = \sum_{i=0}^{n} A_i x^{n-i} y^i = x^n + A_d x^{n-d} y^d + \ldots + A_n y^n, \]

where

\[ A_i = |\{ c \in C \mid \text{wt}(c) = i\}| = \# \text{ of codewords wt } i. \]

\[ A_C(x, y) = A_{C^\perp}(x, y) \text{ iff } C \text{ is formally self-dual code} \]

There exist a SD MDS code [10, 5, 6]_{41} (due to J.-L. Kim, Y. Lee).
Definition of the zeta polynomial

A polynomial \( P(T) \) for which

\[
\frac{(xT + (1 - T)y)^n}{(1 - T)(1 - qT)} P(T) = \ldots + \frac{A_C(x, y) - x^n}{q - 1} T^{n - d} + \ldots .
\]

is called a **Duursma zeta polynomial** of \( C \). (The Duusma zeta polynomial \( P = P_C \) exists and is unique.)

The **functional equation** holds:

\[
P^\perp(T) = P\left(\frac{1}{qT}\right) q^g T^{g^\perp},
\]

where \( g = n/2 + 1 - d \) and \( g^\perp = n/2 + 1 - d^\perp \).

The **Riemann hypothesis** is the statement that all zeros of \( P(T) \) lie on the circle \( |T| = 1/\sqrt{q} \).
Let $C$ be a fsd $b$-divisible $[n, k, d]_q$-code. We say $C$ is **Type I** if $q = b = 2$, and $n$ is even. We say $C$ is **Type II** if $q = 2$, $b = 4$, and $8|n$. We say $C$ is **Type III** if $q = b = 3$, and $4|n$. If $q = 4$, $b = 2$, and $n$ is even then $C$ is said to be **Type IV**.

**Lemma (Mallows-Sloane bounds)** If $C$ is SD then

$$d \leq \begin{cases} 2[n/8] + 2, & \text{if } C \text{ is Type I}, \\ 4[n/24] + 4, & \text{if } C \text{ is Type II}, \\ 3[n/12] + 3, & \text{if } C \text{ is Type III}, \\ 2[n/6] + 2, & \text{if } C \text{ is Type IV}. \end{cases}$$
Virtual weight enumerators

**Virtual weight enumerator** - a homogeneous polynomial

\[ F(x, y) = x^n + \sum_{i=1}^{n} f_i x^{n-i} y^i \]

of degree \( n \) with complex coefficients.

If \( F(x, y) = x^n + \sum_{i=d}^{n} f_i x^{n-i} y^i \) with \( f_d \neq 0 \) then we say that the **length** of \( F \) is \( n \) and the **minimum distance** of \( F \) is \( d \).
Virtual weight enumerators

Formally self-dual weight enumerator - Such an $F$ of even degree invariant under $\sigma = \frac{1}{\sqrt{q}} \begin{pmatrix} 1 & 1 \\ q-1 & -1 \end{pmatrix}$

Genus of a FSDWE: $\gamma(F) = \frac{n}{2} + 1 - d$.

A virtual weight enumerator $F$ is formally identified with an object we call a virtual code $C$ subject only to the following condition: we formally extend the definition of $C \mapsto \lambda C$ to all virtual codes by $\lambda C = F$. 
Extremal FSDWEs

**Theorem:** If $F$ is a FSDWE with length $n$ and minimum distance $d$ then

$$d \leq \begin{cases} 
2[n/8] + 2, & \text{if } C \text{ is Type I}, \\
4[n/24] + 4, & \text{if } C \text{ is Type II}, \\
3[n/12] + 3, & \text{if } C \text{ is Type III}, \\
2[n/6] + 2, & \text{if } C \text{ is Type IV}.
\end{cases}$$

A FSDWE $F$ (ie, a virtual SD code) is called **extremal** if the bound in the theorem holds with equality.
The statement of the RH for codes

A code is called *optimal* if its minimum distance is maximal among all linear codes of that length and dimension.

It is known that any two extremal codes (if they exist) have the same weight enumerator polynomial.

Duusma’s conjecture:

The RH holds for $Z(T)$ for all extremal virtual codes.
Sage examples

Example

```python
sage: C = HammingCode(3,GF(2))
sage: C.zeta_function()
(1/5 + 2/5*T + 2/5*T^2)/(1 - 3*T + 2*T^2)
sage: C = ExtendedTernaryGolayCode()
sage: C.zeta_function()
(1/7 + 3/7*T + 3/7*T^2)/(1 - 4*T + 3*T^2)
```

These satisfy the RH.
Consider the $[26, 13, 6]_{13}$ code with weight distribution

$$[1, 0, 0, 0, 0, 39, 0, 455, 0, 1196, 0, 2405, 0, 2405, 0, 1196, 0, 455, 0, 39, 0, 0, 0, 0, 0, 1].$$

This is an optimal formally self-dual code $C$. 
Coding theory and Sage

David Joyner

What is Sage?
What is in Sage?
The CLI
The GUI

Python
What is Python?
for loops
XGCD, lambda, Sage examples
Repeated squaring algorithm
Fibonacci numbers
Classes

Coding theory functionality in Sage
General constructions
Coding theory functions
Coding theory bounds

Coding theory not implemented in Sage

Cryptography
Classical cryptography
Algebraic cryptosystems
LFSRs
Blum-Goldwasser

Miscellaneous topics
Guava
Duursma zeta functions
Self-dual codes

\[ P(T) = \frac{3}{17710} + \frac{6}{8855} T + \frac{611}{22539} T^{2} + \frac{2185}{66303} T^{3} + \frac{3441}{408595} T^{4} + \]
\[ + \frac{17710}{408595} T^{5} + \frac{44499}{1634380} T^{6} + \frac{22539}{51584} T^{7} + \frac{66303}{1040060} T^{8} + \]
\[ + \frac{408595}{260015} T^{9} + \frac{19552}{4999} T^{10} + \frac{408595}{22539} T^{11} + \frac{408595}{44499} T^{12} + \]
\[ + \frac{2185}{168245} T^{13} + \frac{611}{8855} T^{14} + \frac{2185}{8855} T^{15} + \frac{3441}{8855} T^{16}. \]

Using Sage, it can be checked that only 8 of the 12 zeros of this function have absolute value \( \sqrt{2} \).
Duursma has “explicitly” computed all zeta functions of extremal virtual SD codes.

Duursma verified the RH for Type IV codes.

For all low values of the parameters, computations using Sage have shown that the RH holds.
Sage has good functionality for working with Self-dual codes.
Sage includes a database of all self-dual binary codes of length \( \leq 20 \) (and some of length 22). The main function is `self_dual_codes_binary`, which is a list of Python dictionaries.

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

- `code` - a self-dual code \( C \) of length \( n \), dimension \( n/2 \), over \( GF(2) \),
- `order autgp` - order of the permutation autom. group of \( C \),
- `Type` - the type of \( C \) (which can be "I" or "II", in the binary case),
- `spectrum` - the spectrum \([A_0, A_1, \ldots, A_n]\),
- `Comment` - possibly an empty string.
Self-dual codes database

sage: C = self_dual_codes_binary(10)["10"]
sage: C["0"]["code"] == C["0"]["code"].dual_code()
True
sage: C["1"]["code"] == C["1"]["code"].dual_code()
True
sage: len(C.keys()) # number of inequiv sd codes of length 10
2
sage: C = self_dual_codes_binary(12)["12"]
sage: C["0"]["code"] == C["0"]["code"].dual_code()
True
sage: C["1"]["code"] == C["1"]["code"].dual_code()
True
sage: C["2"]["code"] == C["2"]["code"].dual_code()
True

These commands check that some of the database entries (of length 10 and of length 12), are indeed self dual.
Self-orthogonal codes

For classification of doubly even self-orthogonal codes using Sage, see http://www.rlmiller.org/de_codes/.
The number of permutation equivalence classes of all doubly even \([n, k]\)–codes is shown in the table at http://www.rlmiller.org/de_codes/, and the list of codes so far discovered is linked from the list entries.
Self-orthogonal codes

Figure: http://www.rlmiller.org/de_codes/.
Self-orthogonal codes

Each link on that webpage points to a Sage object file, which when loaded. For example

```sage```
```
L = load('24_12_de_codes.sobj')
```

is a list of matrices in standard form.
Cool example (on self-dual codes).

Recall:

- $W_5(x, y) = x^8 + 14x^4y^4 + y^8$ is the weight enumerator of the Type II $[8, 4, 4]$ code $C$ constructed by extending the binary $[7, 4, 3]$ Hamming code by a check bit. This is the smallest Type II code.
- $W_6(x, y) = x^{24} + 759x^{16}y^8 + 2576x^{12}y^{12} + 759x^8y^{16} + y^{24}$ is the weight enumerator of the extended binary Golay code with parameters $[24, 12, 8]$. 
Cool example (on self-dual codes).

Theorem

Assume $C$ is a formally self-dual divisible code of Type II. Then $A_C(x, y)$ is invariant under the group

$$G_{II} = \langle \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}, \begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix} \rangle$$

of order 192. Moreover, $C[x, y]^{G_{II}} = \mathbb{C}[W_5, W_6]$. 

Cool example (on self-dual codes).

Consider the group $G$ generated by

$$
g_1 = \begin{pmatrix} 1/\sqrt{q} & 1/\sqrt{q} \\ (q-1)/\sqrt{q} & -1/\sqrt{q} \end{pmatrix}, \quad g_2 = \begin{pmatrix} i & 0 \\ 0 & 1 \end{pmatrix}, \quad g_3 = \begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix},
$$

with $q = 2$. This group leaves invariant the weight enumerator of any self-dual doubly even binary code, e.g., \textit{ExtendedBinaryGolayCode}. 


Cool example (on self-dual codes).

**Sage** code below calls **GAP** to construct the matrix group.

```python
sage: F = CyclotomicField(8)
sage: z = F.gen()
sage: a = z+1/z
sage: b = z^2
sage: MS = MatrixSpace(F,2,2)
sage: g1 = MS([[1/a,1/a],[1/a,-1/a]])
sage: g2 = MS([[1,0],[0,b]])
sage: g3 = MS([[b,0],[0,1]])
sage: G = MatrixGroup([g1,g2,g3])
sage: G.order()
192
```
Cool example (on self-dual codes).

Sage code below calls Singular for computing the invariants of $G$. We see that the invariants are indeed as predicted.

```
sage: G.invariant_generators()
[x1^8 + 14*x1^4*x2^4 + x2^8,
 x1^24 + 10626/1025*x1^20*x2^4 + 735471/1025*x1^16*x2^8
 + 2704156/1025*x1^12*x2^12 + 735471/1025*x1^8*x2^16
 + 10626/1025*x1^4*x2^20 + x2^24]
```
Cool example (on self-dual codes).

The above result implies that any such weight enumerator must be a polynomial in

\[ x^8 + 14x^4y^4 + y^8 \]

and

\[ 1025x^{24} + 10626x^{20}y^4 + 735471x^{16}y^8 + 2704156x^{12}y^{12} + 735471x^8y^{16} + 10626x^4y^{20} + 1025y^{24}. \]

(Consistent with the previously mentioned result.)
Sage is a community. Please join us!

Have fun with Sage!

The End.