galois

Matt Hostetter

Sep 09, 2021

CONTENTS

1 Installation **3**
- 1.1 Install with pip . . . 3

2 Tutorials **5**
- 2.1 Intro to Galois Fields: Prime Fields . . . 5
- 2.2 Intro to Galois Fields: Extension Fields . . . 11
- 2.3 Constructing Galois field array classes . . . 23
- 2.4 Array creation . . . 24
- 2.5 Galois field array arithmetic . . . 27
- 2.6 Extremely large fields . . . 31

3 Performance Testing **33**
- 3.1 Performance compared with native numpy . . . 33

4 Development **41**
- 4.1 Install for development . . . 41
- 4.2 Install for development with min dependencies . . . 41
- 4.3 Lint the package . . . 41
- 4.4 Run the unit tests . . . 42
- 4.5 Build the documentation . . . 42

5 API Reference v0.0.18 **43**
- 5.1 Galois Fields . . . 43
- 5.2 Polynomials over Galois Fields . . . 99
- 5.3 Number Theory . . . 117
- 5.4 Integer Factorization . . . 131
- 5.5 Primes . . . 141
- 5.6 Forward Error Correcting Codes . . . 148
- 5.7 Linear Sequences . . . 167
- 5.8 Numpy Examples . . . 167

6 Release Notes **189**
- 6.1 v0.0.18 . . . 189
- 6.2 v0.0.17 . . . 190
- 6.3 v0.0.16 . . . 191
- 6.4 v0.0.15 . . . 191
- 6.5 v0.0.14 . . . 192

7 Indices and tables **195**

Index **197**

Ask Jacobi or Gauss publicly to give their opinion, not as to the truth, but as to the importance of these theorems. Later there will be, I hope, some people who will find it to their advantage to decipher all this mess. – Évariste Galois, two days before his death

Fig. 1: Évariste Galois, image credit

CHAPTER

ONE

INSTALLATION

1.1 Install with pip

The latest version of `galois` can be installed from PyPI using `pip`.

```
$ python3 -m pip install galois
```

Note: Fun fact: read here from python core developer Brett Cannon about why it's better to install using `python3 -m pip` rather than `pip3`.

CHAPTER

TWO

TUTORIALS

2.1 Intro to Galois Fields: Prime Fields

A Galois field is a finite field named in honor of Évariste Galois, one of the fathers of group theory. A *field* is a set that is closed under addition, subtraction, multiplication, and division. To be *closed* under an operation means that performing the operation on any two elements of the set will result in a third element from the set. A *finite field* is a field with a finite set.

Galois proved that finite fields exist only when their *order* (or size of the set) is a prime power p^m. Accordingly, finite fields can be broken into two categories: prime fields $\mathrm{GF}(p)$ and extension fields $\mathrm{GF}(p^m)$. This tutorial will focus on prime fields.

2.1.1 Elements

The elements of the Galois field $\mathrm{GF}(p)$ are naturally represented as the integers $\{0, 1, \ldots, p-1\}$.

Using the `galois` package, a Galois field array class is created using the class factory *galois.GF()*.

```
In [1]: GF7 = galois.GF(7); GF7
Out[1]: <class 'numpy.ndarray over GF(7)'>

In [2]: print(GF7.properties)
GF(7):
  characteristic: 7
  degree: 1
  order: 7
```

The elements of the Galois field can be represented as a 1-dimensional array using the *galois.FieldArray.Elements()* method.

```
In [3]: GF7.Elements()
Out[3]: GF([0, 1, 2, 3, 4, 5, 6], order=7)
```

This array should be read as "a Galois field array [0, 1, 2, 3, 4, 5, 6] over the finite field with order 7".

2.1.2 Arithmetic mod p

Addition, subtraction, and multiplication in $\mathrm{GF}(p)$ is equivalent to integer addition, subtraction, and multiplication reduced modulo p. Mathematically speaking, this is the ring of integers mod p, $\mathbb{Z}/p\mathbb{Z}$.

With `galois`, we can represent a single Galois field element using `GF7(int)`. For example, `GF7(3)` to represent the field element 3. We can see that $3 + 5 \equiv 1 \pmod 7$, so accordingly $3 + 5 = 1$ in $\mathrm{GF}(7)$. The same can be shown for subtraction and multiplication.

```
In [4]: GF7(3) + GF7(5)
Out[4]: GF(1, order=7)

In [5]: GF7(3) - GF7(5)
Out[5]: GF(5, order=7)

In [6]: GF7(3) * GF7(5)
Out[6]: GF(1, order=7)
```

The power of `galois`, however, is array arithmetic not scalar arithmetic. Random arrays over $\mathrm{GF}(7)$ can be created using *`galois.FieldArray.Random()`*. Normal binary operators work on Galois field arrays just like numpy arrays.

```
In [7]: x = GF7.Random(10); x
Out[7]: GF([4, 3, 2, 2, 2, 0, 2, 2, 6, 4], order=7)

In [8]: y = GF7.Random(10); y
Out[8]: GF([0, 4, 6, 2, 3, 0, 2, 1, 5, 1], order=7)

In [9]: x + y
Out[9]: GF([4, 0, 1, 4, 5, 0, 4, 3, 4, 5], order=7)

In [10]: x - y
Out[10]: GF([4, 6, 3, 0, 6, 0, 0, 1, 1, 3], order=7)

In [11]: x * y
Out[11]: GF([0, 5, 5, 4, 6, 0, 4, 2, 2, 4], order=7)
```

The `galois` package includes the ability to display the arithmetic tables for a given finite field. The table is only readable for small fields, but nonetheless the capability is provided. Select a few computations at random and convince yourself the answers are correct.

```
In [12]: print(GF7.arithmetic_table("+"))

 x + y  0 | 1 | 2 | 3 | 4 | 5 | 6
-------------------------------
     0  0 | 1 | 2 | 3 | 4 | 5 | 6
-------------------------------
     1  1 | 2 | 3 | 4 | 5 | 6 | 0
-------------------------------
     2  2 | 3 | 4 | 5 | 6 | 0 | 1
-------------------------------
     3  3 | 4 | 5 | 6 | 0 | 1 | 2
-------------------------------
     4  4 | 5 | 6 | 0 | 1 | 2 | 3
-------------------------------
```

(continues on next page)

(continued from previous page)

```
     5  5 | 6 | 0 | 1 | 2 | 3 | 4
-------------------------------
     6  6 | 0 | 1 | 2 | 3 | 4 | 5
```

```
In [13]: print(GF7.arithmetic_table("-"))
```

```
 x - y  0 | 1 | 2 | 3 | 4 | 5 | 6
     0  0 | 6 | 5 | 4 | 3 | 2 | 1
-------------------------------
     1  1 | 0 | 6 | 5 | 4 | 3 | 2
-------------------------------
     2  2 | 1 | 0 | 6 | 5 | 4 | 3
-------------------------------
     3  3 | 2 | 1 | 0 | 6 | 5 | 4
-------------------------------
     4  4 | 3 | 2 | 1 | 0 | 6 | 5
-------------------------------
     5  5 | 4 | 3 | 2 | 1 | 0 | 6
-------------------------------
     6  6 | 5 | 4 | 3 | 2 | 1 | 0
```

```
In [14]: print(GF7.arithmetic_table("*"))
```

```
 x * y  0 | 1 | 2 | 3 | 4 | 5 | 6
     0  0 | 0 | 0 | 0 | 0 | 0 | 0
-------------------------------
     1  0 | 1 | 2 | 3 | 4 | 5 | 6
-------------------------------
     2  0 | 2 | 4 | 6 | 1 | 3 | 5
-------------------------------
     3  0 | 3 | 6 | 2 | 5 | 1 | 4
-------------------------------
     4  0 | 4 | 1 | 5 | 2 | 6 | 3
-------------------------------
     5  0 | 5 | 3 | 1 | 6 | 4 | 2
-------------------------------
     6  0 | 6 | 5 | 4 | 3 | 2 | 1
```

Division in $\mathrm{GF}(p)$ is a little more difficult. Division can't be as simple as taking $x/y \pmod p$ because many integer divisions do not result in integers. The division of $x/y = z$ can be reformulated as the question "what z multiplied by y results in x?". This is an equivalent problem to "what z multiplied by y results in 1?", where z is the multiplicative inverse of y.

To find the multiplicative inverse of y, one can simply perform trial multiplication until the result of 1 is found. For instance, suppose $y = 4$ in $\mathrm{GF}(7)$. We can multiply 4 by every element in the field until the product is 1 and we'll find that $4^{-1} = 2$ in $\mathrm{GF}(7)$, namely $2 * 4 = 1$ in $\mathrm{GF}(7)$.

```
In [15]: y = GF7(4); y
Out[15]: GF(4, order=7)

# Hypothesize each element from GF(7)
In [16]: guesses = GF7.Elements(); guesses
Out[16]: GF([0, 1, 2, 3, 4, 5, 6], order=7)

In [17]: results = y * guesses; results
Out[17]: GF([0, 4, 1, 5, 2, 6, 3], order=7)

In [18]: y_inv = guesses[np.where(results == 1)[0][0]]; y_inv
Out[18]: GF(2, order=7)
```

This algorithm is terribly inefficient for large fields, however. Fortunately, Euclid came up with an efficient algorithm, now called the Extended Eulcidean Algorithm. Given two integers a and b, the Extended Euclidean Algorithm finds the integers x and y such that $xa + yb = \gcd(a, b)$. This algorithm is implemented in *galois.gcd()*.

If a is a field element of $\mathrm{GF}(7)$ and $b = 7$, then $x = a^{-1}$ in $\mathrm{GF}(7)$. Note, the GCD will always be 1 because p is prime.

```
In [19]: galois.egcd(4, 7)
Out[19]: (1, 2, -1)
```

The `galois` package uses the Extended Euclidean Algorithm to compute multiplicative inverses (and division) in prime fields. The inverse of 4 in $\mathrm{GF}(7)$ can be easily computed in the following way.

```
In [20]: y = GF7(4); y
Out[20]: GF(4, order=7)

In [21]: np.reciprocal(y)
Out[21]: GF(2, order=7)

In [22]: y ** -1
Out[22]: GF(2, order=7)
```

With this in mind, the division table for $\mathrm{GF}(7)$ can be calculated. Note that division is not defined for $y = 0$.

```
In [23]: print(GF7.arithmetic_table("/"))
 x / y  1 | 2 | 3 | 4 | 5 | 6
    0   0 | 0 | 0 | 0 | 0 | 0
-------------------------------
    1   1 | 4 | 5 | 2 | 3 | 6
-------------------------------
    2   2 | 1 | 3 | 4 | 6 | 5
-------------------------------
    3   3 | 5 | 1 | 6 | 2 | 4
-------------------------------
    4   4 | 2 | 6 | 1 | 5 | 3
-------------------------------
    5   5 | 6 | 4 | 3 | 1 | 2
-------------------------------
    6   6 | 3 | 2 | 5 | 4 | 1
```

(continues on next page)

(continued from previous page)

2.1.3 Primitive elements

A property of finite fields is that some elements can produce the entire field by their powers. Namely, a *primitive element* g of $\mathrm{GF}(p)$ is an element such that $\mathrm{GF}(p) = \{0, g^0, g^1, \ldots, g^{p-1}\}$. In prime fields $\mathrm{GF}(p)$, the generators or primitive elements of $\mathrm{GF}(p)$ are *primitive roots mod p*.

The integer g is a *primitive root mod p* if every number coprime to p can be represented as a power of g mod p. Namely, every a coprime to p can be represented as $g^k \equiv a \pmod{p}$ for some k. In prime fields, since p is prime, every integer $1 \leq a < p$ is coprime to p. Finding primitive roots mod p is implemented in *galois.primitive_root()* and *galois.primitive_roots()*.

```
In [24]: galois.primitive_root(7)
Out[24]: 3
```

Since 3 is a primitive root mod 7, the claim is that the elements of $\mathrm{GF}(7)$ can be written as $\mathrm{GF}(7) = \{0, 3^0, 3^1, \ldots, 3^6\}$. 0 is a special element. It can technically be represented as $g^{-\infty}$, however that can't be computed on a computer. For the non-zero elements, they can easily be calculated as powers of g. The set $\{3^0, 3^1, \ldots, 3^6\}$ forms a cyclic multiplicative group, namely $\mathrm{GF}(7)^\times$.

```
In [25]: g = GF7(3); g
Out[25]: GF(3, order=7)

In [26]: g ** np.arange(0, GF7.order - 1)
Out[26]: GF([1, 3, 2, 6, 4, 5], order=7)
```

A primitive element of $\mathrm{GF}(p)$ can be accessed through *galois.FieldClass.primitive_element*.

```
In [27]: GF7.primitive_element
Out[27]: GF(3, order=7)
```

The `galois` package allows you to easily display all powers of an element and their equivalent polynomial, vector, and integer representations. Let's ignore the polynomial and vector representations for now; they will become useful for extension fields.

```
In [28]: print(GF7.repr_table())

 Power | Polynomial | Vector | Integer
-------+------------+--------+---------
   0   |     0      |  [0]   |    0
-------+------------+--------+---------
  3^0  |     1      |  [1]   |    1
-------+------------+--------+---------
  3^1  |     3      |  [3]   |    3
-------+------------+--------+---------
  3^2  |     2      |  [2]   |    2
-------+------------+--------+---------
  3^3  |     6      |  [6]   |    6
-------+------------+--------+---------
  3^4  |     4      |  [4]   |    4
-------+------------+--------+---------
```

(continues on next page)

(continued from previous page)

```
  3^5  |     5      |  [5]   |    5
```

There are multiple primitive elements of a given field. In the case of $\mathrm{GF}(7)$, 3 and 5 are primitive elements.

```
In [29]: GF7.primitive_elements
Out[29]: GF([3, 5], order=7)
```

```
In [30]: print(GF7.repr_table(GF7(5)))

 Power | Polynomial | Vector | Integer

   0   |     0      |  [0]   |    0
-------+------------+--------+--------
  5^0  |     1      |  [1]   |    1
-------+------------+--------+--------
  5^1  |     5      |  [5]   |    5
-------+------------+--------+--------
  5^2  |     4      |  [4]   |    4
-------+------------+--------+--------
  5^3  |     6      |  [6]   |    6
-------+------------+--------+--------
  5^4  |     2      |  [2]   |    2
-------+------------+--------+--------
  5^5  |     3      |  [3]   |    3
```

And it can be seen that every other element of $\mathrm{GF}(7)$ is not a generator of the multiplicative group. For instance, 2 does not generate the multiplicative group $\mathrm{GF}(7)^\times$.

```
In [31]: print(GF7.repr_table(GF7(2)))

 Power | Polynomial | Vector | Integer

   0   |     0      |  [0]   |    0
-------+------------+--------+--------
  2^0  |     1      |  [1]   |    1
-------+------------+--------+--------
  2^1  |     2      |  [2]   |    2
-------+------------+--------+--------
  2^2  |     4      |  [4]   |    4
-------+------------+--------+--------
  2^3  |     1      |  [1]   |    1
-------+------------+--------+--------
  2^4  |     2      |  [2]   |    2
-------+------------+--------+--------
  2^5  |     4      |  [4]   |    4
```

2.2 Intro to Galois Fields: Extension Fields

As discussed in the previous tutorial, a finite field is a finite set that is closed under addition, subtraction, multiplication, and division. Galois proved that finite fields exist only when their *order* (or size of the set) is a prime power p^m. When the order is prime, the arithmetic can be *mostly* computed using integer arithmetic mod p. In the case of prime power order, namely extension fields $\mathrm{GF}(p^m)$, the finite field arithmetic is computed using polynomials over $\mathrm{GF}(p)$ with degree less than m.

2.2.1 Elements

The elements of the Galois field $\mathrm{GF}(p^m)$ can be thought of as the integers $\{0, 1, \ldots, p^m - 1\}$, although their arithmetic doesn't obey integer arithmetic. A more common interpretation is to view the elements of $\mathrm{GF}(p^m)$ as polynomials over $\mathrm{GF}(p)$ with degree less than m, for instance $a_{m-1}x^{m-1} + a_{m-2}x^{m-2} + \cdots + a_1x^1 + a_0 \in \mathrm{GF}(p)[x]$.

For example, consider the finite field $\mathrm{GF}(3^2)$. The order of the field is 9, so we know there are 9 elements. The only question is what to call each element and how to represent them.

```
In [1]: GF9 = galois.GF(9); GF9
Out[1]: <class 'numpy.ndarray over GF(3^2)'>

In [2]: print(GF9.properties)
GF(3^2):
  characteristic: 3
  degree: 2
  order: 9
  irreducible_poly: Poly(x^2 + 2x + 2, GF(3))
  is_primitive_poly: True
  primitive_element: GF(3, order=3^2)
```

In `galois`, the default element display mode is the integer representation. This is natural when storing and working with integer numpy arrays. However, there are other representations and at times it may be useful to view the elements in one of those representations.

```
In [3]: GF9.Elements()
Out[3]: GF([0, 1, 2, 3, 4, 5, 6, 7, 8], order=3^2)
```

Below, we will view the representation table again to compare and contrast the different equivalent representations.

```
In [4]: print(GF9.repr_table())

 Power | Polynomial | Vector | Integer
---------------------------------------
   0   |     0      | [0, 0] |    0
---------------------------------------
  ^0   |     1      | [0, 1] |    1
---------------------------------------
  ^1   |            | [1, 0] |    3
---------------------------------------
  ^2   |    + 1     | [1, 1] |    4
---------------------------------------
  ^3   |   2 + 1    | [2, 1] |    7
---------------------------------------
  ^4   |     2      | [0, 2] |    2
```

(continues on next page)

(continued from previous page)

```
 ^5  |     2     | [2, 0] |    6
-----------------------------------
 ^6  |   2 + 2   | [2, 2] |    8
-----------------------------------
 ^7  |    + 2    | [1, 2] |    5
```

As before, there are some elements whose powers generate the field; we'll skip them for now. The main takeaway from this table is the equivalence of the integer representation and the polynomial (or vector) representation. In $\mathrm{GF}(3^2)$, the element $2\alpha + 1$ is a polynomial that can be thought of as $2x + 1$ (we'll explain why α is used later). The conversion between the polynomial and integer representation is performed simply by substituting $x = 3$ into the polynomial $2 * 3 + 1 = 7$, using normal integer arithmetic.

With `galois`, we can represent a single Galois field element using `GF9(int)` or `GF9(string)`.

```
# Create a single field element from its integer representation
In [5]: GF9(7)
Out[5]: GF(7, order=3^2)

# Create a single field element from its polynomial representation
In [6]: GF9("2x + 1")
Out[6]: GF(7, order=3^2)

# Create a single field element from its vector representation
In [7]: GF9.Vector([2,1])
Out[7]: GF(7, order=3^2)
```

In addition to scalars, these conversions work for arrays.

```
In [8]: GF9([4, 8, 7])
Out[8]: GF([4, 8, 7], order=3^2)

In [9]: GF9(["x + 1", "2x + 2", "2x + 1"])
Out[9]: GF([4, 8, 7], order=3^2)

In [10]: GF9.Vector([[1,1], [2,2], [2,1]])
Out[10]: GF([4, 8, 7], order=3^2)
```

Anytime you have a large array, you can easily view its elements in whichever mode is most illustrative.

```
In [11]: x = GF9.Random(10); x
Out[11]: GF([0, 1, 6, 0, 0, 6, 6, 1, 5, 7], order=3^2)

# Temporarily print x using the power representation
In [12]: with GF9.display("power"):
   ....:     print(x)
   ....:
GF([0, 1, ^5, 0, 0, ^5, ^5, 1, ^7, ^3], order=3^2)

# Permanently set the display mode to the polynomial representation
In [13]: GF9.display("poly"); x
Out[13]: GF([0, 1, 2, 0, 0, 2, 2, 1,  + 2, 2 + 1], order=3^2)
```

(continues on next page)

(continued from previous page)

```
# Reset the display mode to the integer representation
In [14]: GF9.display(); x
Out[14]: GF([0, 1, 6, 0, 0, 6, 6, 1, 5, 7], order=3^2)

# Or convert the (10,) array of GF(p^m) elements to a (10,2) array of vectors over GF(p)
In [15]: x.vector()
Out[15]:
GF([[0, 0],
    [0, 1],
    [2, 0],
    [0, 0],
    [0, 0],
    [2, 0],
    [2, 0],
    [0, 1],
    [1, 2],
    [2, 1]], order=3)
```

2.2.2 Arithmetic mod p(x)

In prime fields $\mathrm{GF}(p)$, integer arithmetic (addition, subtraction, and multiplication) was performed and then reduced modulo p. In extension fields $\mathrm{GF}(p^m)$, polynomial arithmetic (addition, subtraction, and multiplication) is performed over $\mathrm{GF}(p)$ and then reduced by a polynomial $p(x)$. This polynomial is called an irreducible polynomial because it cannot be factored over $\mathrm{GF}(p)$ – an analogue of a prime number.

When constructing an extension field, if an explicit irreducible polynomial is not specified, a default is chosen. The default polynomial is a Conway polynomial which is irreducible and *primitive*, see *galois.conway_poly()* for more information.

```
In [16]: p = GF9.irreducible_poly; p
Out[16]: Poly(x^2 + 2x + 2, GF(3))

In [17]: galois.is_irreducible(p)
Out[17]: True

# Explicit polynomial factorization returns itself as a multiplicity-1 factor
In [18]: galois.poly_factors(p)
Out[18]: ([Poly(x^2 + 2x + 2, GF(3))], [1])
```

Polynomial addition and subtract never result in polynomials of larger degree, so it is unnecessary to reduce them modulo $p(x)$. Let's try an example of addition. Suppose two field elements $a = x + 2$ and $b = x + 1$. These polynomials add degree-wise in $\mathrm{GF}(p)$. Relatively easily we can see that $a + b = (1 + 1)x + (2 + 1) = 2x$. But we can use `galois` and *galois.Poly* to confirm this.

```
In [19]: GF3 = galois.GF(3)

# Explicitly create a polynomial over GF(3) to represent a
In [20]: a = galois.Poly([1, 2], field=GF3); a
Out[20]: Poly(x + 2, GF(3))
```

(continues on next page)

(continued from previous page)

```
In [21]: a.integer
Out[21]: 5

# Explicitly create a polynomial over GF(3) to represent b
In [22]: b = galois.Poly([1, 1], field=GF3); b
Out[22]: Poly(x + 1, GF(3))

In [23]: b.integer
Out[23]: 4

In [24]: c = a + b; c
Out[24]: Poly(2x, GF(3))

In [25]: c.integer
Out[25]: 6
```

We can do the equivalent calculation directly in the field $\mathrm{GF}(3^2)$.

```
In [26]: a = GF9("x + 2"); a
Out[26]: GF(5, order=3^2)

In [27]: b = GF9("x + 1"); b
Out[27]: GF(4, order=3^2)

In [28]: c = a + b; c
Out[28]: GF(6, order=3^2)

# Or view the answer in polynomial form
In [29]: with GF9.display("poly"):
   ....:     print(c)
   ....:
GF(2, order=3^2)
```

From here, we can view the entire addition arithmetic table. And we can choose to view the elements in the integer representation or polynomial representation.

```
In [30]: print(GF9.arithmetic_table("+"))

 x + y  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8
-------------------------------------------
     0  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8
-------------------------------------------
     1  1 | 2 | 0 | 4 | 5 | 3 | 7 | 8 | 6
-------------------------------------------
     2  2 | 0 | 1 | 5 | 3 | 4 | 8 | 6 | 7
-------------------------------------------
     3  3 | 4 | 5 | 6 | 7 | 8 | 0 | 1 | 2
-------------------------------------------
     4  4 | 5 | 3 | 7 | 8 | 6 | 1 | 2 | 0
-------------------------------------------
     5  5 | 3 | 4 | 8 | 6 | 7 | 2 | 0 | 1
-------------------------------------------
```

(continues on next page)

(continued from previous page)

```
    6  6 | 7 | 8 | 0 | 1 | 2 | 3 | 4 | 5
--------------------------------------
    7  7 | 8 | 6 | 1 | 2 | 0 | 4 | 5 | 3
--------------------------------------
    8  8 | 6 | 7 | 2 | 0 | 1 | 5 | 3 | 4
```

```
In [31]: print(GF9.arithmetic_table("+", mode="poly"))
```

```
 x + y    0    |   1    |   2    |        | + 1  | + 2  |   2    | 2 + 1 | 2 + 2
-------------------------------------------------------------------------------
     0    0    |   1    |   2    |        | + 1  | + 2  |   2    | 2 + 1 | 2 + 2
-------------------------------------------------------------------------------
     1    1    |   2    |   0    | + 1    | + 2  |      | 2 + 1  | 2 + 2 |   2
-------------------------------------------------------------------------------
     2    2    |   0    |   1    | + 2    |      | + 1  | 2 + 2  |   2   | 2 + 1
-------------------------------------------------------------------------------
          | + 1  | + 2  |   2    | 2 + 1  | 2 + 2  |   0    |   1   |   2
-------------------------------------------------------------------------------
  + 1  + 1 | + 2  |      | 2 + 1  | 2 + 2  |   2    |   1    |   2   |   0
-------------------------------------------------------------------------------
  + 2  + 2 |      | + 1  | 2 + 2  |   2    | 2 + 1  |   2    |   0   |   1
-------------------------------------------------------------------------------
    2    2 | 2 + 1 | 2 + 2 |   0    |   1    |   2    |      | + 1  | + 2
-------------------------------------------------------------------------------
2 + 1 2 + 1 | 2 + 2 |   2    |   1    |   2    |   0    | + 1  | + 2  |
-------------------------------------------------------------------------------
2 + 2 2 + 2 |   2   | 2 + 1  |   2    |   0    |   1    | + 2  |      | + 1
```

Polynomial multiplication, however, often results in products of larger degree than the multiplicands. In this case, the result must be reduced modulo $p(x)$.

Let's use the same example from before with $a = x + 2$ and $b = x + 1$. To compute $c = ab$, we need to multiply the polynomials $c = (x + 2)(x + 1) = x^2 + 2$ in $\mathrm{GF}(3)$. The issue is that $x^2 + 2$ has degree-2 and the elements of $\mathrm{GF}(3^2)$ can have degree at most 1, hence the need to reduce modulo $p(x)$. After remainder division, we see that $c = ab \equiv x \bmod p$.

As before, let's compute this polynomial product explicitly first.

```
# The irreducible polynomial for GF(3^2)
In [32]: p = GF9.irreducible_poly; p
Out[32]: Poly(x^2 + 2x + 2, GF(3))

# Explicitly create a polynomial over GF(3) to represent a
In [33]: a = galois.Poly([1, 2], field=GF3); a
Out[33]: Poly(x + 2, GF(3))

In [34]: a.integer
Out[34]: 5

# Explicitly create a polynomial over GF(3) to represent b
In [35]: b = galois.Poly([1, 1], field=GF3); b
```

(continues on next page)

(continued from previous page)

```
Out[35]: Poly(x + 1, GF(3))

In [36]: b.integer
Out[36]: 4

In [37]: c = (a * b) % p; c
Out[37]: Poly(x, GF(3))

In [38]: c.integer
Out[38]: 3
```

And now we'll compare that direct computation of this finite field multiplication is equivalent.

```
In [39]: a = GF9("x + 2"); a
Out[39]: GF(5, order=3^2)

In [40]: b = GF9("x + 1"); b
Out[40]: GF(4, order=3^2)

In [41]: c = a * b; c
Out[41]: GF(3, order=3^2)

# Or view the answer in polynomial form
In [42]: with GF9.display("poly"):
   ....:     print(c)
   ....:
GF(, order=3^2)
```

Now the entire multiplication table can be shown for completeness.

```
In [43]: print(GF9.arithmetic_table("*", mode="poly"))

  x * y    0    |   1    |   2    |        | + 1  | + 2  |   2    | 2 + 1 | 2 + 2
-------------------------------------------------------------------------------
     0     0    |   0    |   0    |   0    |   0  |   0  |   0    |   0   |   0
-------------------------------------------------------------------------------
     1     0    |   1    |   2    |        | + 1  | + 2  |   2    | 2 + 1 | 2 + 2
-------------------------------------------------------------------------------
     2     0    |   2    |   1    |   2    | 2 + 2| 2 + 1|        | + 2   | + 1
-------------------------------------------------------------------------------
           0    |        |   2    | + 1    | 2 + 1|   1  | 2 + 2  |   2   | + 2
-------------------------------------------------------------------------------
   + 1     0    | + 1    | 2 + 2  | 2 + 1  |   2  |      | + 2    |   2   |   1
-------------------------------------------------------------------------------
   + 2     0    | + 2    | 2 + 1  |   1    |      | 2 + 2|   2    | + 1   |   2
-------------------------------------------------------------------------------
     2     0    |   2    |        | 2 + 2  | + 2  |   2  | + 1    |   1   | 2 + 1
-------------------------------------------------------------------------------
 2 + 1     0    | 2 + 1  | + 2    |   2    |   2  | + 1  |   1    | 2 + 2 |
-------------------------------------------------------------------------------
 2 + 2     0    | 2 + 2  | + 1    | + 2    |   1  |   2  | 2 + 1  |       |   2
```

Division, as in $\text{GF}(p)$, is a little more difficult. Fortunately the Extended Euclidean Algorithm, which was used in prime fields on integers, can be used for extension fields on polynomials. Given two polynomials a and b, the Extended Euclidean Algorithm finds the polynomials x and y such that $xa + yb = \text{gcd}(a, b)$. This algorithm is implemented in *galois.poly_egcd()*.

If a is a field element of $\text{GF}(3^2)$ and $b = p(x)$, the field's irreducible polynomial, then $x = a^{-1}$ in $\text{GF}(3^2)$. Note, the GCD will always be 1 because $p(x)$ is irreducible.

```
In [44]: p = GF9.irreducible_poly; p
Out[44]: Poly(x^2 + 2x + 2, GF(3))

In [45]: a = galois.Poly([1, 2], field=GF3); a
Out[45]: Poly(x + 2, GF(3))

In [46]: gcd, x, y = galois.poly_egcd(a, p); gcd, x, y
Out[46]: (Poly(1, GF(3)), Poly(x, GF(3)), Poly(2, GF(3)))
```

The claim is that $(x + 2)^{-1} = x$ in $\text{GF}(3^2)$ or, equivalently, $(x + 2)(x) \equiv 1 \bmod p(x)$. This can be easily verified with `galois`.

```
In [47]: (a * x) % p
Out[47]: Poly(1, GF(3))
```

`galois` performs all this arithmetic under the hood. With `galois`, performing finite field arithmetic is as simple as invoking the appropriate numpy function or binary operator.

```
In [48]: a = GF9("x + 2"); a
Out[48]: GF(5, order=3^2)

In [49]: np.reciprocal(a)
Out[49]: GF(3, order=3^2)

In [50]: a ** -1
Out[50]: GF(3, order=3^2)

# Or view the answer in polynomial form
In [51]: with GF9.display("poly"):
   ....:     print(a ** -1)
   ....:
GF(, order=3^2)
```

And finally, for completeness, we'll include the division table for $\text{GF}(3^2)$. Note, division is not defined for $y = 0$.

```
In [52]: print(GF9.arithmetic_table("/", mode="poly"))

  x / y    1    |   2    |        | + 1  |  + 2   |   2    | 2 + 1  | 2 + 2
-------------------------------------------------------------------------------
     0     0    |   0    |   0    |   0  |   0    |   0    |   0    |   0
-------------------------------------------------------------------------------
     1     1    |   2    |  + 2   | 2 + 2|        | 2 + 1  |   2    |  + 1
-------------------------------------------------------------------------------
     2     2    |   1    | 2 + 1  |  + 1 |   2    |  + 2   |        | 2 + 2
-------------------------------------------------------------------------------
               |   2    |   1    |  + 2 |  + 1   |   2    | 2 + 2  | 2 + 1
-------------------------------------------------------------------------------
```

(continues on next page)

(continued from previous page)

```
  + 1   + 1  | 2 + 2 |       |   1   | 2 + 1 |   2   |  + 2  |   2
-------------------------------------------------------------------------
  + 2   + 2  | 2 + 1 | 2 + 2 |   2   |   1   |  + 1  |   2   |
-------------------------------------------------------------------------
    2     2  |       |   2   | 2 + 1 | 2 + 2 |   1   |  + 1  |  + 2
-------------------------------------------------------------------------
2 + 1 2 + 1  |  + 2  |  + 1  |       |   2   | 2 + 2 |   1   |   2
-------------------------------------------------------------------------
2 + 2 2 + 2  |  + 1  |   2   |   2   |  + 2  |       | 2 + 1 |   1
```

2.2.3 Primitive elements

A property of finite fields is that some elements can produce the entire field by their powers. Namely, a *primitive element* g of $\mathrm{GF}(p^m)$ is an element such that $\mathrm{GF}(p^m) = \{0, g^0, g^1, \ldots, g^{p^m-1}\}$.

In `galois`, the primitive elements of an extension field can be found by the class attribute *galois.FieldClass.primitive_element* and *galois.FieldClass.primitive_elements*.

```
# Switch to polynomial display mode
In [53]: GF9.display("poly");

In [54]: p = GF9.irreducible_poly; p
Out[54]: Poly(x^2 + 2x + 2, GF(3))

In [55]: GF9.primitive_element
Out[55]: GF(, order=3^2)

In [56]: GF9.primitive_elements
Out[56]: GF([,  + 2, 2, 2 + 1], order=3^2)
```

This means that x, $x+2$, $2x$, and $2x+1$ can all generate the nonzero multiplicative group $\mathrm{GF}(3^2)^\times$. We can examine this by viewing the representation table using different generators.

Here is the representation table using the default generator $g = x$.

```
In [57]: print(GF9.repr_table())

 Power | Polynomial | Vector | Integer
-------------------------------------
   0   |     0      | [0, 0] |    0
-------------------------------------
  ^0   |     1      | [0, 1] |    1
-------------------------------------
  ^1   |            | [1, 0] |    3
-------------------------------------
  ^2   |    + 1     | [1, 1] |    4
-------------------------------------
  ^3   |   2 + 1    | [2, 1] |    7
-------------------------------------
  ^4   |     2      | [0, 2] |    2
-------------------------------------
```

(continues on next page)

(continued from previous page)

```
 ^5  │     2     │ [2, 0] │    6
─────┼───────────┼────────┼──────
 ^6  │   2 + 2   │ [2, 2] │    8
─────┼───────────┼────────┼──────
 ^7  │    + 2    │ [1, 2] │    5
```

And here is the representation table using a different generator $g = 2x + 1$.

```
In [58]: print(GF9.repr_table(GF9("2x + 1")))

  Power    │ Polynomial │ Vector │ Integer
─────────┼────────────┼────────┼─────────
    0      │     0      │ [0, 0] │    0
─────────┼────────────┼────────┼─────────
(2 + 1)^0 │     1      │ [0, 1] │    1
─────────┼────────────┼────────┼─────────
(2 + 1)^1 │   2 + 1    │ [2, 1] │    7
─────────┼────────────┼────────┼─────────
(2 + 1)^2 │   2 + 2    │ [2, 2] │    8
─────────┼────────────┼────────┼─────────
(2 + 1)^3 │            │ [1, 0] │    3
─────────┼────────────┼────────┼─────────
(2 + 1)^4 │     2      │ [0, 2] │    2
─────────┼────────────┼────────┼─────────
(2 + 1)^5 │    + 2     │ [1, 2] │    5
─────────┼────────────┼────────┼─────────
(2 + 1)^6 │    + 1     │ [1, 1] │    4
─────────┼────────────┼────────┼─────────
(2 + 1)^7 │     2      │ [2, 0] │    6
```

All other elements cannot generate the multiplicative subgroup. Another way of putting that is that their multiplicative order is less than $p^m - 1$. For example, the element $e = x + 1$ has $\text{ord}(e) = 4$. This can be seen because $e^4 = 1$.

```
In [59]: print(GF9.repr_table(GF9("x + 1")))

  Power   │ Polynomial │ Vector │ Integer
─────────┼────────────┼────────┼─────────
    0     │     0      │ [0, 0] │    0
─────────┼────────────┼────────┼─────────
( + 1)^0  │     1      │ [0, 1] │    1
─────────┼────────────┼────────┼─────────
( + 1)^1  │    + 1     │ [1, 1] │    4
─────────┼────────────┼────────┼─────────
( + 1)^2  │     2      │ [0, 2] │    2
─────────┼────────────┼────────┼─────────
( + 1)^3  │   2 + 2    │ [2, 2] │    8
─────────┼────────────┼────────┼─────────
( + 1)^4  │     1      │ [0, 1] │    1
─────────┼────────────┼────────┼─────────
( + 1)^5  │    + 1     │ [1, 1] │    4
```

(continues on next page)

(continued from previous page)

```
 ( + 1)^6 │     2      │ [0, 2] │    2
──────────┼────────────┼────────┼────────
 ( + 1)^7 │   2 + 2    │ [2, 2] │    8
```

2.2.4 Primitive polynomials

Some irreducible polynomials have special properties, these are primitive polynomial. A degree-m polynomial is *primitive* over $\mathrm{GF}(p)$ if it has as a root that is a generator of $\mathrm{GF}(p^m)$.

In `galois`, the default choice of irreducible polynomial is a Conway polynomial, which is also a primitive polynomial. Consider the finite field $\mathrm{GF}(2^4)$. The Conway polynomial for $\mathrm{GF}(2^4)$ is $C_{2,4} = x^4 + x + 1$, which is irreducible and primitive.

```
In [60]: GF16 = galois.GF(2**4)

In [61]: print(GF16.properties)
GF(2^4):
  characteristic: 2
  degree: 4
  order: 16
  irreducible_poly: Poly(x^4 + x + 1, GF(2))
  is_primitive_poly: True
  primitive_element: GF(2, order=2^4)
```

Since $p(x) = C_{2,4}$ is primitive, it has the primitive element of $\mathrm{GF}(2^4)$ as a root.

```
In [62]: p = GF16.irreducible_poly; p
Out[62]: Poly(x^4 + x + 1, GF(2))

In [63]: galois.is_irreducible(p)
Out[63]: True

In [64]: galois.is_primitive(p)
Out[64]: True

# Evaluate the irreducible polynomial over GF(2^4) at the primitive element
In [65]: p(GF16.primitive_element, field=GF16)
Out[65]: GF(0, order=2^4)
```

Since the irreducible polynomial is primitive, we write the field elements in polynomial basis with indeterminate α instead of x, where α represents the primitive element of $\mathrm{GF}(p^m)$. For powers of α less than 4, it can be seen that $\alpha = x$, $\alpha^2 = x^2$, and $\alpha^3 = x^3$.

```
In [66]: print(GF16.repr_table())

 Power  │    Polynomial    │    Vector    │ Integer
────────┼──────────────────┼──────────────┼─────────
   0    │        0         │ [0, 0, 0, 0] │    0
────────┼──────────────────┼──────────────┼─────────
  ^0    │        1         │ [0, 0, 0, 1] │    1
```

(continues on next page)

(continued from previous page)

```
 ^1   |                | [0, 0, 1, 0] |    2
------|----------------|--------------|-------
 ^2   |       ^2       | [0, 1, 0, 0] |    4
------|----------------|--------------|-------
 ^3   |       ^3       | [1, 0, 0, 0] |    8
------|----------------|--------------|-------
 ^4   |       + 1      | [0, 0, 1, 1] |    3
------|----------------|--------------|-------
 ^5   |      ^2 +      | [0, 1, 1, 0] |    6
------|----------------|--------------|-------
 ^6   |     ^3 + ^2    | [1, 1, 0, 0] |   12
------|----------------|--------------|-------
 ^7   |    ^3 +  + 1   | [1, 0, 1, 1] |   11
------|----------------|--------------|-------
 ^8   |      ^2 + 1    | [0, 1, 0, 1] |    5
------|----------------|--------------|-------
 ^9   |      ^3 +      | [1, 0, 1, 0] |   10
------|----------------|--------------|-------
 ^10  |    ^2 +  + 1   | [0, 1, 1, 1] |    7
------|----------------|--------------|-------
 ^11  |   ^3 + ^2 +    | [1, 1, 1, 0] |   14
------|----------------|--------------|-------
 ^12  | ^3 + ^2 +  + 1 | [1, 1, 1, 1] |   15
------|----------------|--------------|-------
 ^13  |   ^3 + ^2 + 1  | [1, 1, 0, 1] |   13
------|----------------|--------------|-------
 ^14  |      ^3 + 1    | [1, 0, 0, 1] |    9
```

Extension fields do not need to be constructed from primitive polynomials, however. The polynomial $p(x) = x^4 + x^3 + x^2 + x + 1$ is irreducible, but not primitive. This polynomial can define arithmetic in $\mathrm{GF}(2^4)$. The two fields (the first defined by a primitive polynomial and the second defined by a non-primitive polynomial) are *isomorphic* to one another.

```
In [67]: p = galois.Poly.Degrees([4,3,2,1,0]); p
Out[67]: Poly(x^4 + x^3 + x^2 + x + 1, GF(2))

In [68]: galois.is_irreducible(p)
Out[68]: True

In [69]: galois.is_primitive(p)
Out[69]: False
```

```
In [70]: GF16_v2 = galois.GF(2**4, irreducible_poly=p)

In [71]: print(GF16_v2.properties)
GF(2^4):
  characteristic: 2
  degree: 4
  order: 16
  irreducible_poly: Poly(x^4 + x^3 + x^2 + x + 1, GF(2))
```

(continues on next page)

(continued from previous page)

```
  is_primitive_poly: False
  primitive_element: GF(3, order=2^4)

In [72]: with GF16_v2.display("poly"):
   ....:     print(GF16_v2.primitive_element)
   ....:
GF(x + 1, order=2^4)
```

Notice the primitive element of $\mathrm{GF}(2^4)$ with irreducible polynomial $p(x) = x^4 + x^3 + x^2 + x + 1$ does not have $x + 1$ as root in $\mathrm{GF}(2^4)$.

```
# Evaluate the irreducible polynomial over GF(2^4) at the primitive element
In [73]: p(GF16_v2.primitive_element, field=GF16_v2)
Out[73]: GF(6, order=2^4)
```

As can be seen in the representation table, for powers of α less than 4, $\alpha \neq x$, $\alpha^2 \neq x^2$, and $\alpha^3 \neq x^3$. Therefore the polynomial indeterminate used is x to distinguish it from α, the primitive element.

```
In [74]: print(GF16_v2.repr_table())

   Power    |     Polynomial     |    Vector    | Integer
------------+--------------------+--------------+---------
     0      |         0          | [0, 0, 0, 0] |    0
------------+--------------------+--------------+---------
 (x + 1)^0  |         1          | [0, 0, 0, 1] |    1
------------+--------------------+--------------+---------
 (x + 1)^1  |       x + 1        | [0, 0, 1, 1] |    3
------------+--------------------+--------------+---------
 (x + 1)^2  |      x^2 + 1       | [0, 1, 0, 1] |    5
------------+--------------------+--------------+---------
 (x + 1)^3  | x^3 + x^2 + x + 1  | [1, 1, 1, 1] |   15
------------+--------------------+--------------+---------
 (x + 1)^4  |   x^3 + x^2 + x    | [1, 1, 1, 0] |   14
------------+--------------------+--------------+---------
 (x + 1)^5  |   x^3 + x^2 + 1    | [1, 1, 0, 1] |   13
------------+--------------------+--------------+---------
 (x + 1)^6  |        x^3         | [1, 0, 0, 0] |    8
------------+--------------------+--------------+---------
 (x + 1)^7  |    x^2 + x + 1     | [0, 1, 1, 1] |    7
------------+--------------------+--------------+---------
 (x + 1)^8  |      x^3 + 1       | [1, 0, 0, 1] |    9
------------+--------------------+--------------+---------
 (x + 1)^9  |        x^2         | [0, 1, 0, 0] |    4
------------+--------------------+--------------+---------
 (x + 1)^10 |      x^3 + x^2     | [1, 1, 0, 0] |   12
------------+--------------------+--------------+---------
 (x + 1)^11 |    x^3 + x + 1     | [1, 0, 1, 1] |   11
------------+--------------------+--------------+---------
 (x + 1)^12 |         x          | [0, 0, 1, 0] |    2
------------+--------------------+--------------+---------
 (x + 1)^13 |      x^2 + x       | [0, 1, 1, 0] |    6
------------+--------------------+--------------+---------
```

(continues on next page)

(continued from previous page)

```
(x + 1)^14 │      x^3 + x      │ [1, 0, 1, 0] │    10
```

2.3 Constructing Galois field array classes

The main idea of the `galois` package is that it constructs "Galois field array classes" using `GF = galois.GF(p**m)`. Galois field array classes, e.g. `GF`, are subclasses of `numpy.ndarray` and their constructors `a = GF(array_like)` mimic the `numpy.array()` function. Galois field arrays, e.g. `a`, can be operated on like any other numpy array. For example: `a + b`, `np.reshape(a, new_shape)`, `np.multiply.reduce(a, axis=0)`, etc.

Galois field array classes are subclasses of *galois.FieldArray* with metaclass *galois.FieldClass*. The metaclass provides useful methods and attributes related to the finite field.

The Galois field $\mathrm{GF}(2)$ is already constructed in `galois`. It can be accessed by *galois.GF2*.

```
In [1]: GF2 = galois.GF2

In [2]: print(GF2)
<class 'numpy.ndarray over GF(2)'>

In [3]: issubclass(GF2, np.ndarray)
Out[3]: True

In [4]: issubclass(GF2, galois.FieldArray)
Out[4]: True

In [5]: issubclass(type(GF2), galois.FieldClass)
Out[5]: True

In [6]: print(GF2.properties)
GF(2):
  characteristic: 2
  degree: 1
  order: 2
```

$\mathrm{GF}(2^m)$ fields, where m is a positive integer, can be constructed using the class factory *galois.GF()*.

```
In [7]: GF8 = galois.GF(2**3)

In [8]: print(GF8)
<class 'numpy.ndarray over GF(2^3)'>

In [9]: issubclass(GF8, np.ndarray)
Out[9]: True

In [10]: issubclass(GF8, galois.FieldArray)
Out[10]: True

In [11]: issubclass(type(GF8), galois.FieldClass)
Out[11]: True
```

(continues on next page)

(continued from previous page)

```
In [12]: print(GF8.properties)
GF(2^3):
  characteristic: 2
  degree: 3
  order: 8
  irreducible_poly: Poly(x^3 + x + 1, GF(2))
  is_primitive_poly: True
  primitive_element: GF(2, order=2^3)
```

$\mathrm{GF}(p)$ fields, where p is prime, can be constructed using the class factory *galois.GF()*.

```
In [13]: GF7 = galois.GF(7)

In [14]: print(GF7)
<class 'numpy.ndarray over GF(7)'>

In [15]: issubclass(GF7, np.ndarray)
Out[15]: True

In [16]: issubclass(GF7, galois.FieldArray)
Out[16]: True

In [17]: issubclass(type(GF7), galois.FieldClass)
Out[17]: True

In [18]: print(GF7.properties)
GF(7):
  characteristic: 7
  degree: 1
  order: 7
```

2.4 Array creation

2.4.1 Explicit construction

Galois field arrays can be constructed either explicitly or through numpy view casting. The method of array creation is the same for all Galois fields, but $\mathrm{GF}(7)$ is used as an example here.

```
# Represents an existing numpy array
In [1]: x_np = np.random.randint(0, 7, 10, dtype=int); x_np
Out[1]: array([2, 2, 0, 4, 6, 3, 6, 1, 2, 0])

# Create a Galois field array through explicit construction (x_np is copied)
In [2]: x = GF7(x_np); x
Out[2]: GF([2, 2, 0, 4, 6, 3, 6, 1, 2, 0], order=7)
```

2.4.2 View casting

```
# View cast an existing array to a Galois field array (no copy operation)
In [3]: y = x_np.view(GF7); y
Out[3]: GF([2, 2, 0, 4, 6, 3, 6, 1, 2, 0], order=7)
```

Warning: View casting creates a pointer to the original data and simply interprets it as a new `numpy.ndarray` subclass, namely the Galois field classes. So, if the original array is modified so will the Galois field array.

```
In [4]: x_np
Out[4]: array([2, 2, 0, 4, 6, 3, 6, 1, 2, 0])

# Add 1 (mod 7) to the first element of x_np
In [5]: x_np[0] = (x_np[0] + 1) % 7; x_np
Out[5]: array([3, 2, 0, 4, 6, 3, 6, 1, 2, 0])

# Notice x is unchanged due to the copy during the explicit construction
In [6]: x
Out[6]: GF([2, 2, 0, 4, 6, 3, 6, 1, 2, 0], order=7)

# Notice y is changed due to view casting
In [7]: y
Out[7]: GF([3, 2, 0, 4, 6, 3, 6, 1, 2, 0], order=7)
```

2.4.3 Alternate constructors

There are alternate constructors for convenience: *FieldArray.Zeros()*, *FieldArray.Ones()*, *FieldArray.Range()*, *FieldArray.Random()*, and *FieldArray.Elements()*.

```
In [8]: GF256.Random((2,5))
Out[8]:
GF([[128,  20, 105,  62,  27],
    [221,  24, 117, 171,  56]], order=2^8)

In [9]: GF256.Range(10,20)
Out[9]: GF([10, 11, 12, 13, 14, 15, 16, 17, 18, 19], order=2^8)

In [10]: GF256.Elements()
Out[10]:
GF([  0,   1,   2,   3,   4,   5,   6,   7,   8,   9,  10,  11,  12,  13,
     14,  15,  16,  17,  18,  19,  20,  21,  22,  23,  24,  25,  26,  27,
     28,  29,  30,  31,  32,  33,  34,  35,  36,  37,  38,  39,  40,  41,
     42,  43,  44,  45,  46,  47,  48,  49,  50,  51,  52,  53,  54,  55,
     56,  57,  58,  59,  60,  61,  62,  63,  64,  65,  66,  67,  68,  69,
     70,  71,  72,  73,  74,  75,  76,  77,  78,  79,  80,  81,  82,  83,
     84,  85,  86,  87,  88,  89,  90,  91,  92,  93,  94,  95,  96,  97,
     98,  99, 100, 101, 102, 103, 104, 105, 106, 107, 108, 109, 110, 111,
    112, 113, 114, 115, 116, 117, 118, 119, 120, 121, 122, 123, 124, 125,
    126, 127, 128, 129, 130, 131, 132, 133, 134, 135, 136, 137, 138, 139,
    140, 141, 142, 143, 144, 145, 146, 147, 148, 149, 150, 151, 152, 153,
```

(continues on next page)

(continued from previous page)

```
    154, 155, 156, 157, 158, 159, 160, 161, 162, 163, 164, 165, 166, 167,
    168, 169, 170, 171, 172, 173, 174, 175, 176, 177, 178, 179, 180, 181,
    182, 183, 184, 185, 186, 187, 188, 189, 190, 191, 192, 193, 194, 195,
    196, 197, 198, 199, 200, 201, 202, 203, 204, 205, 206, 207, 208, 209,
    210, 211, 212, 213, 214, 215, 216, 217, 218, 219, 220, 221, 222, 223,
    224, 225, 226, 227, 228, 229, 230, 231, 232, 233, 234, 235, 236, 237,
    238, 239, 240, 241, 242, 243, 244, 245, 246, 247, 248, 249, 250, 251,
    252, 253, 254, 255], order=2^8)
```

2.4.4 Array dtypes

Galois field arrays support all signed and unsigned integer dtypes, presuming the data type can store values in $[0, p^m)$. The default dtype is the smallest valid unsigned dtype.

```
In [11]: GF = galois.GF(7)

In [12]: a = GF.Random(10); a
Out[12]: GF([2, 3, 2, 1, 3, 0, 5, 5, 4, 2], order=7)

In [13]: a.dtype
Out[13]: dtype('uint8')

# Type cast an existing Galois field array to a different dtype
In [14]: a = a.astype(np.int16); a
Out[14]: GF([2, 3, 2, 1, 3, 0, 5, 5, 4, 2], order=7)

In [15]: a.dtype
Out[15]: dtype('int16')
```

A specific dtype can be chosen by providing the `dtype` keyword argument during array creation.

```
# Explicitly create a Galois field array with a specific dtype
In [16]: b = GF.Random(10, dtype=np.int16); b
Out[16]: GF([0, 2, 1, 4, 5, 2, 5, 4, 0, 3], order=7)

In [17]: b.dtype
Out[17]: dtype('int16')
```

2.4.5 Field element display modes

The default representation of a finite field element is the integer representation. That is, for $\mathrm{GF}(p^m)$ the p^m elements are represented as $\{0, 1, \ldots, p^m - 1\}$. For extension fields, the field elements can alternatively be represented as polynomials in $\mathrm{GF}(p)[x]$ with degree less than m. For prime fields, the integer and polynomial representations are equivalent because in the polynomial representation each element is a degree-0 polynomial over $\mathrm{GF}(p)$.

For example, in $\mathrm{GF}(2^3)$ the integer representation of the 8 field elements is $\{0, 1, 2, 3, 4, 5, 6, 7\}$ and the polynomial representation is $\{0,\ 1,\ x,\ x+1,\ x^2,\ x^2+1,\ x^2+x,\ x^2+x+1\}$.

```
In [18]: GF = galois.GF(2**3)
```

(continues on next page)

(continued from previous page)

```
In [19]: a = GF.Random(10)

# The default mode represents the field elements as integers
In [20]: a
Out[20]: GF([0, 2, 7, 2, 2, 0, 1, 1, 0, 0], order=2^3)

# The display mode can be set to "poly" mode
In [21]: GF.display("poly"); a
Out[21]: GF([0, , ^2 +  + 1, , , 0, 1, 1, 0, 0], order=2^3)

# The display mode can be set to "power" mode
In [22]: GF.display("power"); a
Out[22]: GF([0, , ^5, , , 0, 1, 1, 0, 0], order=2^3)

# Reset the display mode to the default
In [23]: GF.display(); a
Out[23]: GF([0, 2, 7, 2, 2, 0, 1, 1, 0, 0], order=2^3)
```

The *FieldClass.display()* method can be called as a context manager.

```
# The original display mode
In [24]: print(a)
GF([0, 2, 7, 2, 2, 0, 1, 1, 0, 0], order=2^3)

# The new display context
In [25]: with GF.display("poly"):
   ....:     print(a)
   ....:
GF([0, , ^2 +  + 1, , , 0, 1, 1, 0, 0], order=2^3)

In [26]: with GF.display("power"):
   ....:     print(a)
   ....:
GF([0, , ^5, , , 0, 1, 1, 0, 0], order=2^3)

# Returns to the original display mode
In [27]: print(a)
GF([0, 2, 7, 2, 2, 0, 1, 1, 0, 0], order=2^3)
```

2.5 Galois field array arithmetic

2.5.1 Addition, subtraction, multiplication, division

A finite field is a set that defines the operations addition, subtraction, multiplication, and division. The field is closed under these operations.

```
In [1]: GF7 = galois.GF(7)

In [2]: print(GF7)
<class 'numpy.ndarray over GF(7)'>
```

(continues on next page)

(continued from previous page)

```
# Create a random GF(7) array with 10 elements
In [3]: x = GF7.Random(10); x
Out[3]: GF([2, 5, 6, 0, 6, 2, 3, 0, 6, 2], order=7)

# Create a random GF(7) array with 10 elements, with the lowest element being 1 (used to␣
↪prevent ZeroDivisionError later on)
In [4]: y = GF7.Random(10, low=1); y
Out[4]: GF([1, 4, 2, 6, 6, 6, 5, 6, 3, 6], order=7)

# Addition in the finite field
In [5]: x + y
Out[5]: GF([3, 2, 1, 6, 5, 1, 1, 6, 2, 1], order=7)

# Subtraction in the finite field
In [6]: x - y
Out[6]: GF([1, 1, 4, 1, 0, 3, 5, 1, 3, 3], order=7)

# Multiplication in the finite field
In [7]: x * y
Out[7]: GF([2, 6, 5, 0, 1, 5, 1, 0, 4, 5], order=7)

# Division in the finite field
In [8]: x / y
Out[8]: GF([2, 3, 3, 0, 1, 5, 2, 0, 2, 5], order=7)

In [9]: x // y
Out[9]: GF([2, 3, 3, 0, 1, 5, 2, 0, 2, 5], order=7)
```

One can easily create the addition, subtraction, multiplication, and division tables for any field. Here is an example using $\mathrm{GF}(7)$.

```
In [10]: X, Y = np.meshgrid(GF7.Elements(), GF7.Elements(), indexing="ij")

In [11]: X + Y
Out[11]:
GF([[0, 1, 2, 3, 4, 5, 6],
    [1, 2, 3, 4, 5, 6, 0],
    [2, 3, 4, 5, 6, 0, 1],
    [3, 4, 5, 6, 0, 1, 2],
    [4, 5, 6, 0, 1, 2, 3],
    [5, 6, 0, 1, 2, 3, 4],
    [6, 0, 1, 2, 3, 4, 5]], order=7)

In [12]: X - Y
Out[12]:
GF([[0, 6, 5, 4, 3, 2, 1],
    [1, 0, 6, 5, 4, 3, 2],
    [2, 1, 0, 6, 5, 4, 3],
    [3, 2, 1, 0, 6, 5, 4],
    [4, 3, 2, 1, 0, 6, 5],
    [5, 4, 3, 2, 1, 0, 6],
```

(continues on next page)

(continued from previous page)

```
    [6, 5, 4, 3, 2, 1, 0]], order=7)

In [13]: X * Y
Out[13]:
GF([[0, 0, 0, 0, 0, 0, 0],
    [0, 1, 2, 3, 4, 5, 6],
    [0, 2, 4, 6, 1, 3, 5],
    [0, 3, 6, 2, 5, 1, 4],
    [0, 4, 1, 5, 2, 6, 3],
    [0, 5, 3, 1, 6, 4, 2],
    [0, 6, 5, 4, 3, 2, 1]], order=7)

In [14]: X, Y = np.meshgrid(GF7.Elements(), GF7.Elements()[1:], indexing="ij")

In [15]: X / Y
Out[15]:
GF([[0, 0, 0, 0, 0, 0],
    [1, 4, 5, 2, 3, 6],
    [2, 1, 3, 4, 6, 5],
    [3, 5, 1, 6, 2, 4],
    [4, 2, 6, 1, 5, 3],
    [5, 6, 4, 3, 1, 2],
    [6, 3, 2, 5, 4, 1]], order=7)
```

2.5.2 Scalar multiplication

A finite field $\mathrm{GF}(p^m)$ is a set that is closed under four operations: addition, subtraction, multiplication, and division. For multiplication, $xy = z$ for $x, y, z \in \mathrm{GF}(p^m)$.

Let's define another notation for scalar multiplication. For $x \cdot r = z$ for $x, z \in \mathrm{GF}(p^m)$ and $r \in \mathbb{Z}$, which represents r additions of x, i.e. $x + \cdots + x = z$. In prime fields $\mathrm{GF}(p)$ multiplication and scalar multiplication are equivalent. However, in extension fields $\mathrm{GF}(p^m)$ they are not.

Warning: In the extension field $\mathrm{GF}(2^3)$, there is a difference between `GF8(6) * GF8(2)` and `GF8(6) * 2`. The former represents the field element "6" multiplied by the field element "2" using finite field multiplication. The latter represents adding the field element "6" two times.

```
In [16]: GF8 = galois.GF(2**3)

In [17]: a = GF8.Random(10); a
Out[17]: GF([4, 3, 5, 1, 7, 5, 5, 0, 2, 0], order=2^3)

# Calculates a x "2" in the finite field
In [18]: a * GF8(2)
Out[18]: GF([3, 6, 1, 2, 5, 1, 1, 0, 4, 0], order=2^3)

# Calculates a + a
In [19]: a * 2
Out[19]: GF([0, 0, 0, 0, 0, 0, 0, 0, 0, 0], order=2^3)
```

In prime fields $\mathrm{GF}(p)$, multiplication and scalar multiplication are equivalent.

```
In [20]: GF7 = galois.GF(7)

In [21]: a = GF7.Random(10); a
Out[21]: GF([0, 5, 1, 2, 0, 5, 6, 5, 2, 2], order=7)

# Calculates a x "2" in the finite field
In [22]: a * GF7(2)
Out[22]: GF([0, 3, 2, 4, 0, 3, 5, 3, 4, 4], order=7)

# Calculates a + a
In [23]: a * 2
Out[23]: GF([0, 3, 2, 4, 0, 3, 5, 3, 4, 4], order=7)
```

2.5.3 Exponentiation

```
In [24]: GF7 = galois.GF(7)

In [25]: print(GF7)
<class 'numpy.ndarray over GF(7)'>

In [26]: x = GF7.Random(10); x
Out[26]: GF([2, 1, 0, 1, 0, 5, 4, 0, 4, 1], order=7)

# Calculates "x" * "x", note 2 is not a field element
In [27]: x ** 2
Out[27]: GF([4, 1, 0, 1, 0, 4, 2, 0, 2, 1], order=7)
```

2.5.4 Logarithm

```
In [28]: GF7 = galois.GF(7)

In [29]: print(GF7)
<class 'numpy.ndarray over GF(7)'>

# The primitive element of the field
In [30]: GF7.primitive_element
Out[30]: GF(3, order=7)

In [31]: x = GF7.Random(10, low=1); x
Out[31]: GF([5, 2, 1, 4, 4, 2, 3, 6, 5, 4], order=7)

# Notice the outputs of log(x) are not field elements, but integers
In [32]: e = np.log(x); e
Out[32]: array([5, 2, 0, 4, 4, 2, 1, 3, 5, 4])

In [33]: GF7.primitive_element**e
Out[33]: GF([5, 2, 1, 4, 4, 2, 3, 6, 5, 4], order=7)
```

(continues on next page)

(continued from previous page)

```
In [34]: np.all(GF7.primitive_element**e == x)
Out[34]: True
```

2.6 Extremely large fields

Arbitrarily-large $\mathrm{GF}(2^m)$, $\mathrm{GF}(p)$, $\mathrm{GF}(p^m)$ fields are supported. Because field elements can't be represented with `numpy.int64`, we use `dtype=object` in the `numpy` arrays. This enables use of native python `int`, which doesn't overflow. It comes at a performance cost though. There are no JIT-compiled arithmetic ufuncs. All the arithmetic is done in pure python. All the same array operations, broadcasting, ufunc methods, etc are supported.

2.6.1 Large GF(p) fields

```
In [1]: prime = 36893488147419103183

In [2]: galois.is_prime(prime)
Out[2]: True

In [3]: GF = galois.GF(prime)

In [4]: print(GF)
<class 'numpy.ndarray over GF(36893488147419103183)'>

In [5]: a = GF.Random(10); a
Out[5]:
GF([9044847532831320892, 15853482708728350832, 19781878178992414494,
    14494851118923906252, 3630731975922491509, 33690577635501475944,
    31052763121380356207, 5863951190644048027, 11848896132760502240,
    34579630831090244537], order=36893488147419103183)

In [6]: b = GF.Random(10); b
Out[6]:
GF([22063817286325419212, 32421375398638500307, 13331892119880457926,
    15240641888093679243, 32037927147526209750, 4826751148850055089,
    8597459099047558856, 21448807778508595067, 20501826987105372954,
    22990730533344240285], order=36893488147419103183)

In [7]: a + b
Out[7]:
GF([31108664819156740104, 11381369959947747956, 33113770298872872420,
    29735493007017585495, 35668659123448701259, 16238406369324278 50,
    27567340730088118 80, 27312758969152643094, 32350723119865875194,
    20676873217015381639], order=36893488147419103183)
```

2.6.2 Large GF(2^m) fields

```
In [8]: GF = galois.GF(2**100)

In [9]: print(GF)
<class 'numpy.ndarray over GF(2^100)'>

In [10]: a = GF([2**8, 2**21, 2**35, 2**98]); a
Out[10]:
GF([256, 2097152, 34359738368, 316912650057057350374175801344],
   order=2^100)

In [11]: b = GF([2**91, 2**40, 2**40, 2**2]); b
Out[11]:
GF([2475880078570760549798248448, 1099511627776, 1099511627776, 4],
   order=2^100)

In [12]: a + b
Out[12]:
GF([2475880078570760549798248704, 1099513724928, 1133871366144,
    316912650057057350374175801348], order=2^100)

# Display elements as polynomials
In [13]: GF.display("poly")
Out[13]: <galois._field._meta_class.DisplayContext at 0x7ff8230bda90>

In [14]: a
Out[14]: GF([^8, ^21, ^35, ^98], order=2^100)

In [15]: b
Out[15]: GF([^91, ^40, ^40, ^2], order=2^100)

In [16]: a + b
Out[16]: GF([^91 + ^8, ^40 + ^21, ^40 + ^35, ^98 + ^2], order=2^100)

In [17]: a * b
Out[17]:
GF([^99, ^61, ^75,
    ^57 + ^56 + ^55 + ^52 + ^48 + ^47 + ^46 + ^45 + ^44 + ^43 + ^41 + ^37 + ^36 + ^35 + ^
↪34 + ^31 + ^30 + ^27 + ^25 + ^24 + ^22 + ^20 + ^19 + ^16 + ^15 + ^11 + ^9 + ^8 + ^6 + ^
↪5 + ^3 + 1],
   order=2^100)

# Reset the display mode
In [18]: GF.display()
Out[18]: <galois._field._meta_class.DisplayContext at 0x7ff822467ef0>
```

CHAPTER

THREE

PERFORMANCE TESTING

3.1 Performance compared with native numpy

To compare the performance of `galois` and native numpy, we'll use a prime field `GF(p)`. This is because it is the simplest field. Namely, addition, subtraction, and multiplication are modulo `p`, which can be simply computed with numpy arrays `(x + y) % p`. For extension fields `GF(p^m)`, the arithmetic is computed using polynomials over `GF(p)` and can't be so tersely expressed in numpy.

3.1.1 Lookup performance

For fields with order less than or equal to `2^20`, `galois` uses lookup tables for efficiency. Here is an example of multiplying two arrays in `GF(31)` using native numpy and `galois` with `ufunc_mode="jit-lookup"`.

```
In [1]: import numpy as np

In [2]: import galois

In [3]: GF = galois.GF(31)

In [4]: GF.ufunc_mode
Out[4]: 'jit-lookup'

In [5]: a = GF.Random(10_000, dtype=int)

In [6]: b = GF.Random(10_000, dtype=int)

In [7]: %timeit a * b
79.7 µs ± 1 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)

In [8]: aa, bb = a.view(np.ndarray), b.view(np.ndarray)

# Equivalent calculation of a * b using native numpy implementation
In [9]: %timeit (aa * bb) % GF.order
96.6 µs ± 2.4 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

The `galois` ufunc runtime has a floor, however. This is due to a requirement to `view` the output array and convert its dtype with `astype()`. For example, for small array sizes numpy is faster than `galois` because it doesn't need to do these conversions.

```
In [4]: a = GF.Random(10, dtype=int)

In [5]: b = GF.Random(10, dtype=int)

In [6]: %timeit a * b
45.1 µs ± 1.82 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)

In [7]: aa, bb = a.view(np.ndarray), b.view(np.ndarray)

# Equivalent calculation of a * b using native numpy implementation
In [8]: %timeit (aa * bb) % GF.order
1.52 µs ± 34.8 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

However, for large N `galois` is strictly faster than numpy.

```
In [10]: a = GF.Random(10_000_000, dtype=int)

In [11]: b = GF.Random(10_000_000, dtype=int)

In [12]: %timeit a * b
59.8 ms ± 1.64 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)

In [13]: aa, bb = a.view(np.ndarray), b.view(np.ndarray)

# Equivalent calculation of a * b using native numpy implementation
In [14]: %timeit (aa * bb) % GF.order
129 ms ± 8.01 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

3.1.2 Calculation performance

For fields with order greater than `2^20`, `galois` will use explicit arithmetic calculation rather than lookup tables. Even in these cases, `galois` is faster than numpy!

Here is an example multiplying two arrays in `GF(2097169)` using numpy and `galois` with `ufunc_mode="jit-calculate"`.

```
In [1]: import numpy as np

In [2]: import galois

In [3]: GF = galois.GF(2097169)

In [4]: GF.ufunc_mode
Out[4]: 'jit-calculate'

In [5]: a = GF.Random(10_000, dtype=int)

In [6]: b = GF.Random(10_000, dtype=int)

In [7]: %timeit a * b
68.2 µs ± 2.09 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

(continues on next page)

(continued from previous page)

```
In [8]: aa, bb = a.view(np.ndarray), b.view(np.ndarray)

# Equivalent calculation of a * b using native numpy implementation
In [9]: %timeit (aa * bb) % GF.order
93.4 µs ± 2.12 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

And again, the runtime comparison with numpy improves with large N because the time of viewing and type converting the output is small compared to the computation time. `galois` achieves better performance than numpy because the multiplication and modulo operations are compiled together into one ufunc rather than two.

```
In [10]: a = GF.Random(10_000_000, dtype=int)

In [11]: b = GF.Random(10_000_000, dtype=int)

In [12]: %timeit a * b
51.2 ms ± 1.08 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)

In [13]: aa, bb = a.view(np.ndarray), b.view(np.ndarray)

# Equivalent calculation of a * b using native numpy implementation
In [14]: %timeit (aa * bb) % GF.order
111 ms ± 1.48 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

3.1.3 Linear algebra performance

Linear algebra over Galois fields is highly optimized. For prime fields `GF(p)`, the performance is comparable to the native numpy implementation (using BLAS/LAPACK).

```
In [1]: import numpy as np

In [2]: import galois

In [3]: GF = galois.GF(31)

In [4]: A = GF.Random((100,100), dtype=int)

In [5]: B = GF.Random((100,100), dtype=int)

In [6]: %timeit A @ B
720 µs ± 5.36 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)

In [7]: AA, BB = A.view(np.ndarray), B.view(np.ndarray)

# Equivalent calculation of A @ B using the native numpy implementation
In [8]: %timeit (AA @ BB) % GF.order
777 µs ± 4.6 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

For extension fields `GF(p^m)`, the performance of `galois` is close to native numpy linear algebra (about 10x slower). However, for extension fields, each multiplication operation is equivalently a convolution (polynomial multiplication) of two `m`-length arrays. So it's not an apples-to-apples comparison.

Below is a comparison of `galois` computing the correct matrix multiplication over `GF(2^8)` and numpy computing

a normal integer matrix multiplication (which is not the correct result!). This comparison is just for a performance reference.

```
In [1]: import numpy as np

In [2]: import galois

In [3]: GF = galois.GF(2**8)

In [4]: A = GF.Random((100,100), dtype=int)

In [5]: B = GF.Random((100,100), dtype=int)

In [6]: %timeit A @ B
7.13 ms ± 114 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)

In [7]: AA, BB = A.view(np.ndarray), B.view(np.ndarray)

# Native numpy matrix multiplication, which doesn't produce the correct result!!
In [8]: %timeit AA @ BB
651 µs ± 12.4 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```


{

"cells": [

{ "cell_type": "markdown", "id": "subjective-appreciation", "metadata": {}, "source": [

"# GF(p) speed tests"

]

}, {

"cell_type": "code", "execution_count": 1, "id": "hidden-costa", "metadata": {}, "outputs": [], "source": [

"import numpy as npn", "import galois"

]

}, {

"cell_type": "code", "execution_count": 2, "id": "uniform-accuracy", "metadata": {}, "outputs": [

{ "name": "stdout", "output_type": "stream", "text": [

"GF(8009):n", " characteristic: 8009n", " degree: 1n", " order: 8009n", " irreducible_poly: Poly(x + 8006, GF(8009))n", " is_primitive_poly: Truen", " primitive_element: GF(3, order=8009)n", " dtypes: ['uint16', 'uint32', 'int16', 'int32', 'int64']n", " ufunc_mode: 'jit-lookup'n", " ufunc_target: 'cpu'n"

]

}

], "source": [

"prime = galois.next_prime(8000)n", "n", "GF = galois.GF(prime)n", "print(GF.properties)"

]

}, {

“cell_type”: “code”, “execution_count”: 3, “id”: “later-convention”, “metadata”: {}, “outputs”: [

{ “name”: “stdout”, “output_type”: “stream”, “text”: [

“[‘jit-lookup’, ‘jit-calculate’]n”, “[‘cpu’, ‘parallel’]n”

]

}

], “source”: [

“modes = GF.ufunc_modesn”, “targets = GF.ufunc_targetsn”, “targets.remove("cuda") # Can’t test with a GPU on my machinen”, “print(modes)n”, “print(targets)”

]

}, {

“cell_type”: “code”, “execution_count”: 4, “id”: “described-placement”, “metadata”: {}, “outputs”: [], “source”: [

“def speed_test(GF, N):n”, ” a = GF.Random(N)n”, ” b = GF.Random(N, low=1)n”, “n”, ” for operation in [np.add, np.multiply]:n”, ” print(f"Operation: {operation.__name__}")n”, ” for target in targets:n”, ” for mode in modes:n”, ” GF.compile(mode, target)n”, ” print(f"Target: {target}, Mode: {mode}", end="\n ")n”, ” %timeit operation(a, b)n”, ” print()n”, “n”, ” for operation in [np.reciprocal, np.log]:n”, ” print(f"Operation: {operation.__name__}")n”, ” for target in targets:n”, ” for mode in modes:n”, ” GF.compile(mode, target)n”, ” print(f"Target: {target}, Mode: {mode}", end="\n ")n”, ” %timeit operation(b)n”, ” print()”

]

}, {

“cell_type”: “markdown”, “id”: “saving-housing”, “metadata”: {}, “source”: [

“## N = 10k”

]

}, {

“cell_type”: “code”, “execution_count”: 5, “id”: “superb-disposal”, “metadata”: {}, “outputs”: [

{ “name”: “stdout”, “output_type”: “stream”, “text”: [

“Operation: addn”, “Target: cpu, Mode: jit-lookupn”, ” 104 µs ± 1.57 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)n”, “Target: cpu, Mode: jit-calculaten”, ” 71.3 µs ± 2.95 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)n”, “Target: parallel, Mode: jit-lookupn”, ” The slowest run took 436.22 times longer than the fastest. This could mean that an intermediate result is being cached.n”, “10.2 ms ± 18.6 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)n”, “Target: parallel, Mode: jit-calculaten”, ” The slowest run took 24.46 times longer than the fastest. This could mean that an intermediate result is being cached.n”, “3.41 ms ± 2.38 ms per loop (mean ± std. dev. of 7 runs, 1000 loops each)n”, “n”, “Operation: multiplyn”, “Target: cpu, Mode: jit-lookupn”, ” 93.1 µs ± 1.33 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)n”, “Target: cpu, Mode: jit-calculaten”, ” 72.1 µs ± 1.8 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)n”, “Target: parallel, Mode: jit-lookupn”, ” 163 µs ±

```
                    19.9 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)n", "Target:
                    parallel, Mode: jit-calculaten", " 2.5 ms ± 680 µs per loop (mean ± std. dev. of
                    7 runs, 10000 loops each)n", "n", "Operation: reciprocaln", "Target: cpu, Mode:
                    jit-lookupn", " 67.3 µs ± 929 ns per loop (mean ± std. dev. of 7 runs, 10000
                    loops each)n", "Target: cpu, Mode: jit-calculaten", " 6.01 ms ± 61.4 µs per loop
                    (mean ± std. dev. of 7 runs, 100 loops each)n", "Target: parallel, Mode: jit-
                    lookupn", " 152 µs ± 15.9 µs per loop (mean ± std. dev. of 7 runs, 10000 loops
                    each)n", "Target: parallel, Mode: jit-calculaten", " 11.1 ms ± 1.05 ms per loop
                    (mean ± std. dev. of 7 runs, 100 loops each)n", "n", "Operation: logn", "Target:
                    cpu, Mode: jit-lookupn", " 75.3 µs ± 242 ns per loop (mean ± std. dev. of 7 runs,
                    10000 loops each)n", "Target: cpu, Mode: jit-calculaten", " 149 ms ± 846 µs per
                    loop (mean ± std. dev. of 7 runs, 10 loops each)n", "Target: parallel, Mode: jit-
                    lookupn", " 175 µs ± 16.4 µs per loop (mean ± std. dev. of 7 runs, 10000 loops
                    each)n", "Target: parallel, Mode: jit-calculaten", " 56.2 ms ± 20.1 ms per loop
                    (mean ± std. dev. of 7 runs, 10 loops each)n", "n"

                ]

            }

        ], "source": [

            "speed_test(GF, 10_000)"

        ]

    }

], "metadata": {

    "kernelspec": { "display_name": "Python 3", "language": "python", "name": "python3"

    }, "language_info": {

        "codemirror_mode": { "name": "ipython", "version": 3

        }, "file_extension": ".py", "mimetype": "text/x-python", "name": "python", "nbconvert_exporter": "python", "pygments_lexer": "ipython3", "version": "3.8.5"

    }

}, "nbformat": 4, "nbformat_minor": 5

}

{

    "cells": [

        { "cell_type": "markdown", "id": "synthetic-appeal", "metadata": {}, "source": [

            "# GF(2^m) speed tests"

        ]

        }, {

            "cell_type": "code", "execution_count": 1, "id": "planned-violence", "metadata": {}, "outputs": [], "source": [

            "import numpy as npn", "import galois"

        ]

        }, {
```

“cell_type”: “code”, “execution_count”: 2, “id”: “distant-letters”, “metadata”: {}, “outputs”: [

{ “name”: “stdout”, “output_type”: “stream”, “text”: [

“GF(2^13):n”, ” characteristic: 2n”, ” degree: 13n”, ” order: 8192n”, ” irreducible_poly: Poly(x^13 + x^4 + x^3 + x + 1, GF(2))n”, ” is_primitive_poly: Truen”, ” primitive_element: GF(2, order=2^13)n”, ” dtypes: [‘uint16’, ‘uint32’, ‘int16’, ‘int32’, ‘int64’]n”, ” ufunc_mode: ‘jit-lookup’n”, ” ufunc_target: ‘cpu’n”

]

}

], “source”: [

“GF = galois.GF(2**13)n”, “print(GF.properties)”

]

}, {

“cell_type”: “code”, “execution_count”: 3, “id”: “further-turning”, “metadata”: {}, “outputs”: [

{ “name”: “stdout”, “output_type”: “stream”, “text”: [

“[‘jit-lookup’, ‘jit-calculate’]n”, “[‘cpu’, ‘parallel’]n”

]

}

], “source”: [

“modes = GF.ufunc_modesn”, “targets = GF.ufunc_targetsn”, “targets.remove("cuda") # Can’t test with a GPU on my machinen”, “print(modes)n”, “print(targets)”

]

}, {

“cell_type”: “code”, “execution_count”: 4, “id”: “strategic-prerequisite”, “metadata”: {}, “outputs”: [], “source”: [

“def speed_test(GF, N):n”, ” a = GF.Random(N)n”, ” b = GF.Random(N, low=1)n”, “n”, ” for operation in [np.add, np.multiply]:n”, ” print(f"Operation: {operation.__name__}")n”, ” for target in targets:n”, ” for mode in modes:n”, ” GF.compile(mode, target)n”, ” print(f"Target: {target}, Mode: {mode}", end="\n ")n”, ” %timeit operation(a, b)n”, ” print()n”, “n”, ” for operation in [np.reciprocal, np.log]:n”, ” print(f"Operation: {operation.__name__}")n”, ” for target in targets:n”, ” for mode in modes:n”, ” GF.compile(mode, target)n”, ” print(f"Target: {target}, Mode: {mode}", end="\n ")n”, ” %timeit operation(b)n”, ” print()”

]

}, {

“cell_type”: “markdown”, “id”: “homeless-infrastructure”, “metadata”: {}, “source”: [

“## N = 10k”

]

}, {

“cell_type”: “code”, “execution_count”: 5, “id”: “forced-cambridge”, “metadata”: {}, “outputs”: [

```
{ “name”: “stdout”, “output_type”: “stream”, “text”: [
    “Operation: addn”, “Target: cpu, Mode: jit-lookupn”, ” 188 µs ± 23.9 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)n”, “Target: cpu, Mode: jit-calculaten”, ” 95.6 µs ± 11.2 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)n”, “Target: parallel, Mode: jit-lookupn”, ” 61.4 ms ± 4.82 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)n”, “Target: parallel, Mode: jit-calculaten”, ” 61.8 ms ± 6.03 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)n”, “n”, “Operation: multiplyn”, “Target: cpu, Mode: jit-lookupn”, ” 148 µs ± 10.5 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)n”, “Target: cpu, Mode: jit-calculaten”, ” 646 µs ± 73.4 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)n”, “Target: parallel, Mode: jit-lookupn”, ” 59.4 ms ± 4.54 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)n”, “Target: parallel, Mode: jit-calculaten”, ” 59.9 ms ± 4.29 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)n”, “n”, “Operation: reciprocaln”, “Target: cpu, Mode: jit-lookupn”, ” 113 µs ± 7.92 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)n”, “Target: cpu, Mode: jit-calculaten”, ” 9.86 ms ± 1.3 ms per loop (mean ± std. dev. of 7 runs, 100 loops each)n”, “Target: parallel, Mode: jit-lookupn”, ” The slowest run took 189.29 times longer than the fastest. This could mean that an intermediate result is being cached.n”, “4.88 ms ± 6.96 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)n”, “Target: parallel, Mode: jit-calculaten”, ” 59.4 ms ± 4.46 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)n”, “n”, “Operation: logn”, “Target: cpu, Mode: jit-lookupn”, ” 138 µs ± 19 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)n”, “Target: cpu, Mode: jit-calculaten”, ” 63.1 ms ± 3.53 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)n”, “Target: parallel, Mode: jit-lookupn”, ” 58.2 ms ± 1.78 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)n”, “Target: parallel, Mode: jit-calculaten”, ” 69.4 ms ± 2.95 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)n”, “n”
  ]
 }
], “source”: [
  “speed_test(GF, 10_000)”
 ]
}
], “metadata”: {
 “kernelspec”: { “display_name”: “Python 3”, “language”: “python”, “name”: “python3”
 }, “language_info”: {
  “codemirror_mode”: { “name”: “ipython”, “version”: 3
  }, “file_extension”: “.py”, “mimetype”: “text/x-python”, “name”: “python”, “nbconvert_exporter”: “python”, “pygments_lexer”: “ipython3”, “version”: “3.8.5”
 }
}, “nbformat”: 4, “nbformat_minor”: 5
}
```

CHAPTER

FOUR

DEVELOPMENT

For users who would like to actively develop with `galois`, these sections may prove helpful.

4.1 Install for development

The the latest code from `master` can be checked out and installed locally in an "editable" fashion.

```
$ git clone https://github.com/mhostetter/galois.git
$ python3 -m pip install -e galois
```

4.2 Install for development with min dependencies

The package dependencies have minimum supported version. They are stored in `requirements-min.txt`.

`pip` installing `galois` will install the latest versions of the dependencies. If you'd like to test against the oldest supported dependencies, you can do the following:

```
$ git clone https://github.com/mhostetter/galois.git

# First install the minimum version of the dependencies
$ python3 -m pip install -r galois/requirements-min.txt

# Then, installing the package won't upgrade the dependencies since their versions are␣
↪satisfactory
$ python3 -m pip install -e galois
```

4.3 Lint the package

Linting is done with pylint. The linting dependencies are stored in `requirements-lint.txt`.

Install the linter dependencies.

```
$ python3 -m pip install -r requirements-lint.txt
```

Run the linter.

```
$ python3 -m pylint --rcfile=setup.cfg galois/
```

4.4 Run the unit tests

Unit testing is done through pytest. The tests themselves are stored in `tests/`. We test against test vectors, stored in `tests/data/`. generated using SageMath. See the `scripts/generate_test_vectors.py` script. The testing dependencies are stored in `requirements-test.txt`.

Install the test dependencies.

```
$ python3 -m pip install -r requirements-test.txt
```

Run the unit tests.

```
$ python3 -m pytest tests/
```

4.5 Build the documentation

The documentation is generated with Sphinx. The dependencies are stored in `requirements-doc.txt`.

Install the documentation dependencies.

```
$ python3 -m pip install -r requirements-doc.txt
```

Build the HTML documentation. The index page will be located at `docs/build/index.html`.

```
$ sphinx-build -b html -v docs/build/
```

CHAPTER

FIVE

API REFERENCE V0.0.18

5.1 Galois Fields

This section contains classes and functions for creating Galois field arrays.

5.1.1 Galois field class creation

Class factory functions

GF(order[, irreducible_poly, ...])	Factory function to construct a Galois field array class for $\mathrm{GF}(p^m)$.
Field(order[, irreducible_poly, ...])	Alias of *galois.GF()*.

galois.GF

class galois.**GF**(*order*, *irreducible_poly=None*, *primitive_element=None*, *verify=True*, *mode='auto'*)

Factory function to construct a Galois field array class for $\mathrm{GF}(p^m)$.

The created class will be a subclass of *galois.FieldArray* and instance of *galois.FieldClass*. The *galois.FieldArray* inheritance provides the numpy.ndarray functionality. The *galois.FieldClass* metaclass provides a variety of class attributes and methods relating to the finite field.

Parameters

- **order** (*int*) – The order p^m of the field $\mathrm{GF}(p^m)$. The order must be a prime power.
- **irreducible_poly** (*int*, *tuple*, *list*, *numpy.ndarray*, galois.Poly, *optional*) – Optionally specify an irreducible polynomial of degree m over $\mathrm{GF}(p)$ that will define the Galois field arithmetic. An integer may be provided, which is the integer representation of the irreducible polynomial. A tuple, list, or ndarray may be provided, which represents the polynomial coefficients in degree-descending order. The default is None which uses the Conway polynomial $C_{p,m}$ obtained from *galois.conway_poly()*.
- **primitive_element** (*int*, *tuple*, *list*, *numpy.ndarray*, galois.Poly, *optional*) – Optionally specify a primitive element of the field $\mathrm{GF}(p^m)$. A primitive element is a generator of the multiplicative group of the field. For prime fields $\mathrm{GF}(p)$, the primitive element must be an integer and is a primitive root modulo p. For extension fields $\mathrm{GF}(p^m)$, the primitive element is a polynomial of degree less than m over $\mathrm{GF}(p)$. An integer may be provided, which is the integer representation of the polynomial. A tuple, list, or ndarray may be provided, which represents the polynomial coefficients in degree-descending order. The default is None which uses galois.primitive_root(p)

for prime fields and `galois.primitive_element(irreducible_poly)` for extension fields.

- **verify** (*bool, optional*) – Indicates whether to verify that the specified irreducible polynomial is in fact irreducible and that the specified primitive element is in fact a generator of the multiplicative group. The default is `True`. For large fields and irreducible polynomials that are already known to be irreducible (and may take a long time to verify), this argument can be set to `False`. If the default irreducible polynomial and primitive element are used, no verification is performed because the defaults are already guaranteed to be irreducible and a multiplicative generator, respectively.
- **mode** (*str, optional*) – The type of field computation, either `"auto"`, `"jit-lookup"`, or `"jit-calculate"`. The default is `"auto"`. The "jit-lookup" mode will use Zech log, log, and anti-log lookup tables for efficient calculation. The "jit-calculate" mode will not store any lookup tables, but instead perform field arithmetic on the fly. The "jit-calculate" mode is designed for large fields that cannot or should not store lookup tables in RAM. Generally, "jit-calculate" mode will be slower than "jit-lookup". The "auto" mode will determine whether to use "jit-lookup" or "jit-calculate" based on the field's size. In "auto" mode, field's with `order <= 2**20` will use the "jit-lookup" mode.

Returns A new Galois field array class that is a subclass of *galois.FieldArray* and instance of *galois.FieldClass*.

Return type *galois.FieldClass*

Examples

Construct a Galois field array class with default irreducible polynomial and primitive element.

```
# Construct a GF(2^m) class
In [1]: GF256 = galois.GF(2**8)

# Notice the irreducible polynomial is primitive
In [2]: print(GF256.properties)
GF(2^8):
  characteristic: 2
  degree: 8
  order: 256
  irreducible_poly: Poly(x^8 + x^4 + x^3 + x^2 + 1, GF(2))
  is_primitive_poly: True
  primitive_element: GF(2, order=2^8)

In [3]: poly = GF256.irreducible_poly
```

Construct a Galois field specifying a specific irreducible polynomial.

```
# Field used in AES
In [4]: GF256_AES = galois.GF(2**8, irreducible_poly=galois.Poly.Degrees([8,4,3,1,
 ↪0]))

In [5]: print(GF256_AES.properties)
GF(2^8):
  characteristic: 2
  degree: 8
  order: 256
  irreducible_poly: Poly(x^8 + x^4 + x^3 + x + 1, GF(2))
```

(continues on next page)

(continued from previous page)

```
  is_primitive_poly: False
  primitive_element: GF(3, order=2^8)

# Construct a GF(p) class
In [6]: GF571 = galois.GF(571); print(GF571.properties)
GF(571):
  characteristic: 571
  degree: 1
  order: 571

# Construct a very large GF(2^m) class
In [7]: GF2m = galois.GF(2**100); print(GF2m.properties)
GF(2^100):
  characteristic: 2
  degree: 100
  order: 1267650600228229401496703205376
  irreducible_poly: Poly(x^100 + x^57 + x^56 + x^55 + x^52 + x^48 + x^47 + x^46 + x^
↪45 + x^44 + x^43 + x^41 + x^37 + x^36 + x^35 + x^34 + x^31 + x^30 + x^27 + x^25 +␣
↪x^24 + x^22 + x^20 + x^19 + x^16 + x^15 + x^11 + x^9 + x^8 + x^6 + x^5 + x^3 + 1,␣
↪GF(2))
  is_primitive_poly: True
  primitive_element: GF(2, order=2^100)

# Construct a very large GF(p) class
In [8]: GFp = galois.GF(36893488147419103183); print(GFp.properties)
GF(36893488147419103183):
  characteristic: 36893488147419103183
  degree: 1
  order: 36893488147419103183
```

See *galois.FieldArray* for more examples of what Galois field arrays can do.

galois.Field

class galois.**Field**(*order*, *irreducible_poly=None*, *primitive_element=None*, *verify=True*, *mode='auto'*)

Alias of *galois.GF()*.

Abstract base classes

FieldArray(array[, dtype, copy, order, ndmin])	Creates an array over $\mathrm{GF}(p^m)$.
FieldClass(name, bases, namespace, **kwargs)	Defines a metaclass for all *galois.FieldArray* classes.

galois.FieldArray

class galois.**FieldArray**(*array*, *dtype=None*, *copy=True*, *order='K'*, *ndmin=0*)

Creates an array over $\mathrm{GF}(p^m)$.

The *galois.FieldArray* class is a parent class for all Galois field array classes. Any Galois field $\mathrm{GF}(p^m)$ with prime characteristic p and positive integer m, can be constructed by calling the class factory `galois.GF(p**m)`.

Warning: This is an abstract base class for all Galois field array classes. *galois.FieldArray* cannot be instantiated directly. Instead, Galois field array classes are created using *galois.GF()*.

For example, one can create the $\mathrm{GF}(7)$ field array class as follows.

```
In [1]: GF7 = galois.GF(7)

In [2]: print(GF7)
<class 'numpy.ndarray over GF(7)'>
```

This subclass can then be used to instantiate arrays over $\mathrm{GF}(7)$.

```
In [3]: GF7([3,5,0,2,1])
Out[3]: GF([3, 5, 0, 2, 1], order=7)

In [4]: GF7.Random((2,5))
Out[4]:
GF([[1, 4, 0, 1, 0],
    [3, 2, 5, 1, 3]], order=7)
```

galois.FieldArray is a subclass of `numpy.ndarray`. The *galois.FieldArray* constructor has the same syntax as `numpy.array()`. The returned *galois.FieldArray* object is an array that can be acted upon like any other numpy array.

Parameters

- **array** (*array_like*) – The input array to be converted to a Galois field array. The input array is copied, so the original array is unmodified by changes to the Galois field array. Valid input array types are `numpy.ndarray`, `list` or `tuple` of int or str, `int`, or `str`.
- **dtype** (*numpy.dtype, optional*) – The `numpy.dtype` of the array elements. The default is `None` which represents the smallest valid dtype for this class, i.e. the first element in *galois.FieldClass.dtypes*.
- **copy** (*bool, optional*) – The `copy` keyword argument from `numpy.array()`. The default is `True` which makes a copy of the input object is it's an array.
- **order** ({"K", "A", "C", "F"}, optional) – The `order` keyword argument from `numpy.array()`. Valid values are `"K"` (default), `"A"`, `"C"`, or `"F"`.
- **ndmin** (*int, optional*) – The `ndmin` keyword argument from `numpy.array()`. The minimum number of dimensions of the output. The default is 0.

Returns The copied input array as a $\mathrm{GF}(p^m)$ field array.

Return type *galois.FieldArray*

Examples

Construct various kinds of Galois fields using *galois.GF*.

```
# Construct a GF(2^m) class
In [5]: GF256 = galois.GF(2**8); print(GF256)
<class 'numpy.ndarray over GF(2^8)'>

# Construct a GF(p) class
In [6]: GF571 = galois.GF(571); print(GF571)
<class 'numpy.ndarray over GF(571)'>

# Construct a very large GF(2^m) class
In [7]: GF2m = galois.GF(2**100); print(GF2m)
<class 'numpy.ndarray over GF(2^100)'>

# Construct a very large GF(p) class
In [8]: GFp = galois.GF(36893488147419103183); print(GFp)
<class 'numpy.ndarray over GF(36893488147419103183)'>
```

Depending on the field's order (size), only certain `dtype` values will be supported.

```
In [9]: GF256.dtypes
Out[9]:
[numpy.uint8,
 numpy.uint16,
 numpy.uint32,
 numpy.int16,
 numpy.int32,
 numpy.int64]

In [10]: GF571.dtypes
Out[10]: [numpy.uint16, numpy.uint32, numpy.int16, numpy.int32, numpy.int64]
```

Very large fields, which can't be represented using `np.int64`, can only be represented as `dtype=np.object_`.

```
In [11]: GF2m.dtypes
Out[11]: [numpy.object_]

In [12]: GFp.dtypes
Out[12]: [numpy.object_]
```

Newly-created arrays will use the smallest, valid dtype.

```
In [13]: a = GF256.Random(10); a
Out[13]: GF([203, 158, 140,  71,  89, 147, 248, 185,  34, 179], order=2^8)

In [14]: a.dtype
Out[14]: dtype('uint8')
```

This can be explicitly set by specifying the `dtype` keyword argument.

```
In [15]: a = GF256.Random(10, dtype=np.uint32); a
Out[15]: GF([177, 179, 177, 217, 156,  66, 157, 250, 168, 215], order=2^8)

In [16]: a.dtype
Out[16]: dtype('uint32')
```

Arrays can also be created explicitly by converting an "array-like" object.

```
# Construct a Galois field array from a list
In [17]: l = [142, 27, 92, 253, 103]; l
Out[17]: [142, 27, 92, 253, 103]

In [18]: GF256(l)
Out[18]: GF([142,  27,  92, 253, 103], order=2^8)

# Construct a Galois field array from an existing numpy array
In [19]: x_np = np.array(l, dtype=np.int64); x_np
Out[19]: array([142,  27,  92, 253, 103])

In [20]: GF256(l)
Out[20]: GF([142,  27,  92, 253, 103], order=2^8)
```

Arrays can also be created by "view casting" from an existing numpy array. This avoids a copy operation, which is especially useful for large data already brought into memory.

```
In [21]: a = x_np.view(GF256); a
Out[21]: GF([142,  27,  92, 253, 103], order=2^8)

# Changing `x_np` will change `a`
In [22]: x_np[0] = 0; x_np
Out[22]: array([  0,  27,  92, 253, 103])

In [23]: a
Out[23]: GF([  0,  27,  92, 253, 103], order=2^8)
```

Constructors

Elements([dtype])	Creates a Galois field array of the field's elements $\{0, \ldots, p^m - 1\}$.
Identity(size[, dtype])	Creates an $n \times n$ Galois field identity matrix.
Ones(shape[, dtype])	Creates a Galois field array with all ones.
Random([shape, low, high, dtype])	Creates a Galois field array with random field elements.
Range(start, stop[, step, dtype])	Creates a Galois field array with a range of field elements.
Vandermonde(a, m, n[, dtype])	Creates a $m \times n$ Vandermonde matrix of $a \in \mathrm{GF}(p^m)$.
Vector(array[, dtype])	Creates a Galois field array over $\mathrm{GF}(p^m)$ from length-m vectors over the prime subfield $\mathrm{GF}(p)$.
Zeros(shape[, dtype])	Creates a Galois field array with all zeros.

Methods

lu_decompose()	Decomposes the input array into the product of lower and upper triangular matrices.
lup_decompose()	Decomposes the input array into the product of lower and upper triangular matrices using partial pivoting.
row_reduce([ncols])	Performs Gaussian elimination on the matrix to achieve reduced row echelon form.
vector([dtype])	Converts the Galois field array over $\mathrm{GF}(p^m)$ to length-m vectors over the prime subfield $\mathrm{GF}(p)$.

classmethod Elements(*dtype=None*)

Creates a Galois field array of the field's elements $\{0, \ldots, p^m - 1\}$.

Parameters **dtype** (*numpy.dtype, optional*) – The `numpy.dtype` of the array elements. The default is `None` which represents the smallest valid dtype for this class, i.e. the first element in *galois.FieldClass.dtypes*.

Returns A Galois field array of all the field's elements.

Return type *galois.FieldArray*

Examples

```
In [1]: GF = galois.GF(31)

In [2]: GF.Elements()
Out[2]:
GF([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
    17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30], order=31)
```

classmethod Identity(*size*, *dtype=None*)

Creates an $n \times n$ Galois field identity matrix.

Parameters

- **size** (*int*) – The size n along one axis of the matrix. The resulting array has shape `(size,size)`.
- **dtype** (*numpy.dtype, optional*) – The `numpy.dtype` of the array elements. The default is `None` which represents the smallest valid dtype for this class, i.e. the first element in *galois.FieldClass.dtypes*.

Returns A Galois field identity matrix of shape `(size, size)`.

Return type *galois.FieldArray*

Examples

```
In [1]: GF = galois.GF(31)

In [2]: GF.Identity(4)
Out[2]:
GF([[1, 0, 0, 0],
```

(continues on next page)

(continued from previous page)

```
    [0, 1, 0, 0],
    [0, 0, 1, 0],
    [0, 0, 0, 1]], order=31)
```

classmethod **Ones**(*shape*, *dtype=None*)

Creates a Galois field array with all ones.

Parameters

- **shape** (*tuple*) – A numpy-compliant `shape` tuple, see `numpy.ndarray.shape`. An empty tuple `()` represents a scalar. A single integer or 1-tuple, e.g. `N` or `(N,)`, represents the size of a 1-dim array. An n-tuple, e.g. `(M,N)`, represents an n-dim array with each element indicating the size in each dimension.
- **dtype** (*numpy.dtype, optional*) – The `numpy.dtype` of the array elements. The default is `None` which represents the smallest valid dtype for this class, i.e. the first element in *galois.FieldClass.dtypes*.

Returns A Galois field array of ones.

Return type *galois.FieldArray*

Examples

```
In [1]: GF = galois.GF(31)

In [2]: GF.Ones((2,5))
Out[2]:
GF([[1, 1, 1, 1, 1],
    [1, 1, 1, 1, 1]], order=31)
```

classmethod **Random**(*shape=()*, *low=0*, *high=None*, *dtype=None*)

Creates a Galois field array with random field elements.

Parameters

- **shape** (*tuple*) – A numpy-compliant `shape` tuple, see `numpy.ndarray.shape`. An empty tuple `()` represents a scalar. A single integer or 1-tuple, e.g. `N` or `(N,)`, represents the size of a 1-dim array. An n-tuple, e.g. `(M,N)`, represents an n-dim array with each element indicating the size in each dimension.
- **low** (*int, optional*) – The lowest value (inclusive) of a random field element. The default is 0.
- **high** (*int, optional*) – The highest value (exclusive) of a random field element. The default is `None` which represents the field's order p^m.
- **dtype** (*numpy.dtype, optional*) – The `numpy.dtype` of the array elements. The default is `None` which represents the smallest valid dtype for this class, i.e. the first element in *galois.FieldClass.dtypes*.

Returns A Galois field array of random field elements.

Return type *galois.FieldArray*

Examples

```
In [1]: GF = galois.GF(31)

In [2]: GF.Random((2,5))
Out[2]:
GF([[14, 21, 28, 15, 21],
    [26,  3, 21,  0, 20]], order=31)
```

classmethod Range(*start*, *stop*, *step=1*, *dtype=None*)

Creates a Galois field array with a range of field elements.

Parameters

- **start** (*int*) – The starting value (inclusive).
- **stop** (*int*) – The stopping value (exclusive).
- **step** (*int, optional*) – The space between values. The default is 1.
- **dtype** (*numpy.dtype, optional*) – The numpy.dtype of the array elements. The default is None which represents the smallest valid dtype for this class, i.e. the first element in *galois.FieldClass.dtypes*.

Returns A Galois field array of a range of field elements.

Return type *galois.FieldArray*

Examples

```
In [1]: GF = galois.GF(31)

In [2]: GF.Range(10,20)
Out[2]: GF([10, 11, 12, 13, 14, 15, 16, 17, 18, 19], order=31)
```

classmethod Vandermonde(*a*, *m*, *n*, *dtype=None*)

Creates a $m \times n$ Vandermonde matrix of $a \in \mathrm{GF}(p^m)$.

Parameters

- **a** (*int, galois.FieldArray*) – An element of $\mathrm{GF}(p^m)$.
- **m** (*int*) – The number of rows in the Vandermonde matrix.
- **n** (*int*) – The number of columns in the Vandermonde matrix.
- **dtype** (*numpy.dtype, optional*) – The numpy.dtype of the array elements. The default is None which represents the smallest valid dtype for this class, i.e. the first element in *galois.FieldClass.dtypes*.

Returns The $m \times n$ Vandermonde matrix.

Return type *galois.FieldArray*

Examples

```
In [1]: GF = galois.GF(2**3)

In [2]: a = GF.primitive_element
```

(continues on next page)

(continued from previous page)

```
In [3]: V = GF.Vandermonde(a, 7, 7)

In [4]: with GF.display("power"):
   ...:     print(V)
   ...:
GF([[  1,   1,   1,   1,   1,   1,   1],
    [  1,   , ^2, ^3, ^4, ^5, ^6],
    [  1, ^2, ^4, ^6,   , ^3, ^5],
    [  1, ^3, ^6, ^2, ^5,   , ^4],
    [  1, ^4,   , ^5, ^2, ^6, ^3],
    [  1, ^5, ^3,   , ^6, ^4, ^2],
    [  1, ^6, ^5, ^4, ^3, ^2,   ]], order=2^3)
```

classmethod Vector(*array*, *dtype=None*)

Creates a Galois field array over $\mathrm{GF}(p^m)$ from length-m vectors over the prime subfield $\mathrm{GF}(p)$.

Parameters

- **array** (*array_like*) – The input array with field elements in $\mathrm{GF}(p)$ to be converted to a Galois field array in $\mathrm{GF}(p^m)$. The last dimension of the input array must be m. An input array with shape `(n1, n2, m)` has output shape `(n1, n2)`.
- **dtype** (*numpy.dtype, optional*) – The `numpy.dtype` of the array elements. The default is `None` which represents the smallest valid dtype for this class, i.e. the first element in *galois.FieldClass.dtypes*.

Returns A Galois field array over $\mathrm{GF}(p^m)$.

Return type *galois.FieldArray*

Examples

```
In [1]: GF = galois.GF(2**6)

In [2]: vec = galois.GF2.Random((3,6)); vec
Out[2]:
GF([[1, 1, 0, 0, 0, 0],
    [1, 0, 0, 1, 1, 1],
    [0, 1, 1, 0, 0, 0]], order=2)

In [3]: a = GF.Vector(vec); a
Out[3]: GF([48, 39, 24], order=2^6)

In [4]: with GF.display("poly"):
   ...:     print(a)
   ...:
GF([^5 + ^4, ^5 + ^2 +  + 1, ^4 + ^3], order=2^6)

In [5]: a.vector()
Out[5]:
GF([[1, 1, 0, 0, 0, 0],
    [1, 0, 0, 1, 1, 1],
```

(continues on next page)

(continued from previous page)

```
    [0, 1, 1, 0, 0, 0]], order=2)
```

classmethod Zeros(*shape*, *dtype=None*)

Creates a Galois field array with all zeros.

Parameters

- **shape** (*tuple*) – A numpy-compliant `shape` tuple, see `numpy.ndarray.shape`. An empty tuple `()` represents a scalar. A single integer or 1-tuple, e.g. `N` or `(N,)`, represents the size of a 1-dim array. An n-tuple, e.g. `(M,N)`, represents an n-dim array with each element indicating the size in each dimension.
- **dtype** (*numpy.dtype, optional*) – The `numpy.dtype` of the array elements. The default is `None` which represents the smallest valid dtype for this class, i.e. the first element in *galois.FieldClass.dtypes*.

Returns A Galois field array of zeros.

Return type *galois.FieldArray*

Examples

```
In [1]: GF = galois.GF(31)

In [2]: GF.Zeros((2,5))
Out[2]:
GF([[0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0]], order=31)
```

lu_decompose()

Decomposes the input array into the product of lower and upper triangular matrices.

Returns

- *galois.FieldArray* – The lower triangular matrix.
- *galois.FieldArray* – The upper triangular matrix.

Examples

```
In [1]: GF = galois.GF(5)

# Not every square matrix has an LU decomposition
In [2]: A = GF([[2, 4, 4, 1], [3, 3, 1, 4], [4, 3, 4, 2], [4, 4, 3, 1]])

In [3]: L, U = A.lu_decompose()

In [4]: L
Out[4]:
GF([[1, 0, 0, 0],
    [4, 1, 0, 0],
    [2, 0, 1, 0],
    [2, 3, 0, 1]], order=5)
```

(continues on next page)

(continued from previous page)

```
In [5]: U
Out[5]:
GF([[2, 4, 4, 1],
    [0, 2, 0, 0],
    [0, 0, 1, 0],
    [0, 0, 0, 4]], order=5)

# A = L U
In [6]: np.array_equal(A, L @ U)
Out[6]: True
```

`lup_decompose()`

Decomposes the input array into the product of lower and upper triangular matrices using partial pivoting.

Returns

- *galois.FieldArray* – The lower triangular matrix.
- *galois.FieldArray* – The upper triangular matrix.
- *galois.FieldArray* – The permutation matrix.

Examples

```
In [1]: GF = galois.GF(5)

In [2]: A = GF([[1, 3, 2, 0], [3, 4, 2, 3], [0, 2, 1, 4], [4, 3, 3, 1]])

In [3]: L, U, P = A.lup_decompose()

In [4]: L
Out[4]:
GF([[1, 0, 0, 0],
    [0, 1, 0, 0],
    [3, 0, 1, 0],
    [4, 3, 2, 1]], order=5)

In [5]: U
Out[5]:
GF([[1, 3, 2, 0],
    [0, 2, 1, 4],
    [0, 0, 1, 3],
    [0, 0, 0, 3]], order=5)

In [6]: P
Out[6]:
GF([[1, 0, 0, 0],
    [0, 0, 1, 0],
    [0, 1, 0, 0],
    [0, 0, 0, 1]], order=5)

# P A = L U
```

(continues on next page)

(continued from previous page)

```
In [7]: np.array_equal(P @ A, L @ U)
Out[7]: True
```

---

row_reduce(*ncols=None*)

Performs Gaussian elimination on the matrix to achieve reduced row echelon form.

Row reduction operations

1. Swap the position of any two rows.
2. Multiply a row by a non-zero scalar.
3. Add one row to a scalar multiple of another row.

Parameters **ncols** (*int, optional*) – The number of columns to perform Gaussian elimination over. The default is `None` which represents the number of columns of the input array.

Returns The reduced row echelon form of the input array.

Return type *galois.FieldArray*

Examples

```
In [1]: GF = galois.GF(31)

In [2]: A = GF.Random((4,4)); A
Out[2]:
GF([[11, 10, 10,  1],
    [ 5, 20, 25, 24],
    [ 6,  9, 10, 18],
    [30,  3, 22,  5]], order=31)

In [3]: A.row_reduce()
Out[3]:
GF([[1, 0, 0, 0],
    [0, 1, 0, 0],
    [0, 0, 1, 0],
    [0, 0, 0, 1]], order=31)

In [4]: np.linalg.matrix_rank(A)
Out[4]: 4
```

One column is a linear combination of another.

```
In [5]: GF = galois.GF(31)

In [6]: A = GF.Random((4,4)); A
Out[6]:
GF([[ 1, 23, 27, 14],
    [21,  7, 24,  5],
    [12, 15, 17, 20],
    [10, 23,  6, 16]], order=31)
```

(continues on next page)

(continued from previous page)

```
In [7]: A[:,2] = A[:,1] * GF(17); A
Out[7]:
GF([[ 1, 23, 19, 14],
    [21,  7, 26,  5],
    [12, 15,  7, 20],
    [10, 23, 19, 16]], order=31)

In [8]: A.row_reduce()
Out[8]:
GF([[ 1,  0,  0,  0],
    [ 0,  1, 17,  0],
    [ 0,  0,  0,  1],
    [ 0,  0,  0,  0]], order=31)

In [9]: np.linalg.matrix_rank(A)
Out[9]: 3
```

One row is a linear combination of another.

```
In [10]: GF = galois.GF(31)

In [11]: A = GF.Random((4,4)); A
Out[11]:
GF([[11, 20,  2,  4],
    [ 5, 15,  1,  8],
    [17,  4, 28, 13],
    [ 5,  6,  6,  7]], order=31)

In [12]: A[3,:] = A[2,:] * GF(8); A
Out[12]:
GF([[11, 20,  2,  4],
    [ 5, 15,  1,  8],
    [17,  4, 28, 13],
    [12,  1,  7, 11]], order=31)

In [13]: A.row_reduce()
Out[13]:
GF([[ 1,  0,  0, 21],
    [ 0,  1,  0, 25],
    [ 0,  0,  1, 24],
    [ 0,  0,  0,  0]], order=31)

In [14]: np.linalg.matrix_rank(A)
Out[14]: 3
```

---

vector(*dtype=None*)

Converts the Galois field array over $\mathrm{GF}(p^m)$ to length-m vectors over the prime subfield $\mathrm{GF}(p)$.

For an input array with shape `(n1, n2)`, the output shape is `(n1, n2, m)`.

Parameters **dtype** (*numpy.dtype, optional*) – The `numpy.dtype` of the array elements. The default is `None` which represents the smallest valid dtype for this class, i.e. the first element in *galois.FieldClass.dtypes*.

Returns A Galois field array of length-m vectors over GF(p).

Return type *galois.FieldArray*

Examples

```
In [1]: GF = galois.GF(2**6)

In [2]: a = GF.Random(3); a
Out[2]: GF([16, 60, 22], order=2^6)

In [3]: vec = a.vector(); vec
Out[3]:
GF([[0, 1, 0, 0, 0, 0],
    [1, 1, 1, 1, 0, 0],
    [0, 1, 0, 1, 1, 0]], order=2)

In [4]: GF.Vector(vec)
Out[4]: GF([16, 60, 22], order=2^6)
```

galois.FieldClass

class galois.**FieldClass**(*name*, *bases*, *namespace*, ***kwargs*)

Defines a metaclass for all *galois.FieldArray* classes.

This metaclass gives *galois.FieldArray* classes returned from *galois.GF()* class methods and properties relating to its Galois field.

Constructors

—

Methods

arithmetic_table(operation[, mode])	Generates the specified arithmetic table for the Galois field.
compile(mode)	Recompile the just-in-time compiled numba ufuncs for a new calculation mode.
display([mode])	Sets the display mode for all Galois field arrays of this type.
repr_table([primitive_element])	Generates an element representation table comparing the power, polynomial, vector, and integer representations.

Attributes

characteristic	The prime characteristic p of the Galois field $\mathrm{GF}(p^m)$.
default_ufunc_mode	The default ufunc arithmetic mode for this Galois field.
degree	The prime characteristic's degree m of the Galois field $\mathrm{GF}(p^m)$.
display_mode	The representation of Galois field elements, either `"int"`, `"poly"`, or `"power"`.
dtypes	List of valid integer `numpy.dtype` objects that are compatible with this Galois field.
irreducible_poly	The irreducible polynomial $f(x)$ of the Galois field $\mathrm{GF}(p^m)$.
is_extension_field	Indicates if the field's order is a prime power.
is_prime_field	Indicates if the field's order is prime.
is_primitive_poly	Indicates whether the *irreducible_poly* is a primitive polynomial.
name	The Galois field name.
order	The order p^m of the Galois field $\mathrm{GF}(p^m)$.
prime_subfield	The prime subfield $\mathrm{GF}(p)$ of the extension field $\mathrm{GF}(p^m)$.
primitive_element	A primitive element α of the Galois field $\mathrm{GF}(p^m)$.
primitive_elements	All primitive elements α of the Galois field $\mathrm{GF}(p^m)$.
properties	A formatted string displaying relevant properties of the Galois field.
ufunc_mode	The mode for ufunc compilation, either `"jit-lookup"`, `"jit-calculate"`, `"python-calculate"`.
ufunc_modes	All supported ufunc modes for this Galois field array class.

arithmetic_table(*operation*, *mode='int'*)

Generates the specified arithmetic table for the Galois field.

Parameters

- **operation** (*str*) – Either `"+"`, `"-"`, `"*"`, or `"/"`.
- **mode** (*str, optional*) – The display mode to represent the field elements, either `"int"` (default), `"poly"`, or `"power"`.

Returns A UTF-8 formatted arithmetic table.

Return type str

Examples

```
In [1]: GF = galois.GF(3**2)

In [2]: print(GF.arithmetic_table("+"))

 x + y  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8
```

(continues on next page)

(continued from previous page)

```
    0  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8
--------------------------------------------
    1  1 | 2 | 0 | 4 | 5 | 3 | 7 | 8 | 6
--------------------------------------------
    2  2 | 0 | 1 | 5 | 3 | 4 | 8 | 6 | 7
--------------------------------------------
    3  3 | 4 | 5 | 6 | 7 | 8 | 0 | 1 | 2
--------------------------------------------
    4  4 | 5 | 3 | 7 | 8 | 6 | 1 | 2 | 0
--------------------------------------------
    5  5 | 3 | 4 | 8 | 6 | 7 | 2 | 0 | 1
--------------------------------------------
    6  6 | 7 | 8 | 0 | 1 | 2 | 3 | 4 | 5
--------------------------------------------
    7  7 | 8 | 6 | 1 | 2 | 0 | 4 | 5 | 3
--------------------------------------------
    8  8 | 6 | 7 | 2 | 0 | 1 | 5 | 3 | 4
```

```
In [3]: GF = galois.GF(3**2)

In [4]: print(GF.arithmetic_table("+", mode="poly"))

  x + y    0     |   1     |   2     |         | + 1   | + 2   |   2     | 2 + 1 | 2 +␣
↪2

     0     0     |   1     |   2     |         | + 1   | + 2   |   2     | 2 + 1 | 2 +␣
↪2
-------------------------------------------------------------------------------------
     1     1     |   2     |   0     |   + 1   | + 2   |         | 2 + 1 | 2 + 2 |   2␣
↪
-------------------------------------------------------------------------------------
     2     2     |   0     |   1     |   + 2   |         | + 1   | 2 + 2 |   2   | 2 +␣
↪1
-------------------------------------------------------------------------------------
                 |   + 1   |   + 2   |   2     | 2 + 1 | 2 + 2 |   0   |   1   |   2 ␣
↪
-------------------------------------------------------------------------------------
   + 1   + 1     |   + 2   |         | 2 + 1 | 2 + 2 |   2   |   1   |   2   |   0 ␣
↪
-------------------------------------------------------------------------------------
   + 2   + 2     |         |   + 1   | 2 + 2 |   2   | 2 + 1 |   2   |   0   |   1 ␣
↪
-------------------------------------------------------------------------------------
     2     2     | 2 + 1 | 2 + 2 |   0     |   1     |   2     |         | + 1   | + 2␣
↪
-------------------------------------------------------------------------------------
 2 + 1   2 + 1   | 2 + 2 |   2   |   1     |   2     |   0     | + 1   | + 2   |     ␣
↪
-------------------------------------------------------------------------------------
 2 + 2   2 + 2   |   2   | 2 + 1 |   2     |   0     |   1     | + 2   |         | + 1␣
↪
```

(continues on next page)

(continued from previous page)

compile(*mode*)

Recompile the just-in-time compiled numba ufuncs for a new calculation mode.

Parameters **mode** (*str*) – The method of field computation, either "jit-lookup", "jit-calculate", "python-calculate". The "jit-lookup" mode will use Zech log, log, and anti-log lookup tables for speed. The "jit-calculate" mode will not store any lookup tables, but perform field arithmetic on the fly. The "jit-calculate" mode is designed for large fields that cannot store lookup tables in RAM. Generally, "jit-calculate" is slower than "jit-lookup". The "python-calculate" mode is reserved for extremely large fields. In this mode the ufuncs are not JIT-compiled, but are pur python functions operating on python ints. The list of valid modes for this field is in *galois.FieldClass.ufunc_modes*.

display(*mode='int'*)

Sets the display mode for all Galois field arrays of this type.

The display mode can be set to either the integer representation, polynomial representation, or power representation. This function updates *display_mode*.

For the power representation, *np.log()* is computed on each element. So for large fields without lookup tables, this may take longer than desired.

Parameters **mode** (*str, optional*) – The field element display mode, either "int" (default), "poly", or "power".

Examples

Change the display mode by calling the *display()* method.

```
In [1]: GF = galois.GF(2**8)

In [2]: a = GF.Random(); a
Out[2]: GF(57, order=2^8)

# Change the display mode going forward
In [3]: GF.display("poly"); a
Out[3]: GF(^5 + ^4 + ^3 + 1, order=2^8)

In [4]: GF.display("power"); a
Out[4]: GF(^154, order=2^8)

# Reset to the default display mode
In [5]: GF.display(); a
Out[5]: GF(57, order=2^8)
```

The *display()* method can also be used as a context manager, as shown below.

For the polynomial representation, when the primitive element is $x \in \mathrm{GF}(p)[x]$ the polynomial indeterminate used is .

```
In [6]: GF = galois.GF(2**8)

In [7]: print(GF.properties)
```

(continues on next page)

(continued from previous page)

```
GF(2^8):
  characteristic: 2
  degree: 8
  order: 256
  irreducible_poly: Poly(x^8 + x^4 + x^3 + x^2 + 1, GF(2))
  is_primitive_poly: True
  primitive_element: GF(2, order=2^8)

In [8]: a = GF.Random(); a
Out[8]: GF(2, order=2^8)

In [9]: with GF.display("poly"):
   ...:     print(a)
   ...:
GF(, order=2^8)

In [10]: with GF.display("power"):
   ....:     print(a)
   ....:
GF(, order=2^8)
```

But when the primitive element is not $x \in \mathrm{GF}(p)[x]$, the polynomial indeterminate used is `x`.

```
In [11]: GF = galois.GF(2**8, irreducible_poly=galois.Poly.Degrees([8,4,3,1,0]))

In [12]: print(GF.properties)
GF(2^8):
  characteristic: 2
  degree: 8
  order: 256
  irreducible_poly: Poly(x^8 + x^4 + x^3 + x + 1, GF(2))
  is_primitive_poly: False
  primitive_element: GF(3, order=2^8)

In [13]: a = GF.Random(); a
Out[13]: GF(244, order=2^8)

In [14]: with GF.display("poly"):
   ....:     print(a)
   ....:
GF(x^7 + x^6 + x^5 + x^4 + x^2, order=2^8)

In [15]: with GF.display("power"):
   ....:     print(a)
   ....:
GF(^197, order=2^8)
```

---

repr_table(*primitive_element=None*)

Generates an element representation table comparing the power, polynomial, vector, and integer representations.

Parameters **primitive_element** (galois.FieldArray, *optional*) – The primitive ele-

ment to use for the power representation. The default is `None` which uses the field's default primitive element, *galois.FieldClass.primitive_element*.

Returns A UTF-8 formatted table comparing the power, polynomial, vector, and integer representations of each field element.

Return type str

Examples

```
In [1]: GF = galois.GF(2**4)

In [2]: print(GF.repr_table())
```

Power	Polynomial	Vector	Integer
0	0	[0, 0, 0, 0]	0
^0	1	[0, 0, 0, 1]	1
^1		[0, 0, 1, 0]	2
^2	^2	[0, 1, 0, 0]	4
^3	^3	[1, 0, 0, 0]	8
^4	^ + 1	[0, 0, 1, 1]	3
^5	^2 ^	[0, 1, 1, 0]	6
^6	^3 + ^2	[1, 1, 0, 0]	12
^7	^3 ^ ^ + 1	[1, 0, 1, 1]	11
^8	^2 + 1	[0, 1, 0, 1]	5
^9	^3 ^	[1, 0, 1, 0]	10
^10	^2 ^ ^ + 1	[0, 1, 1, 1]	7
^11	^3 + ^2 ^	[1, 1, 1, 0]	14
^12	^3 + ^2 ^ ^ + 1	[1, 1, 1, 1]	15
^13	^3 + ^2 + 1	[1, 1, 0, 1]	13
^14	^3 + 1	[1, 0, 0, 1]	9

```
In [3]: alpha = GF.primitive_elements[-1]

In [4]: print(GF.repr_table(alpha))
```

Power	Polynomial	Vector	Integer

(continues on next page)

(continued from previous page)

```
        0          |         0          | [0, 0, 0, 0] |     0
-------------------+--------------------+--------------+-------
 (^3 + ^2 + )^0    |         1          | [0, 0, 0, 1] |     1
-------------------+--------------------+--------------+-------
 (^3 + ^2 + )^1    |    ^3 + ^2 +       | [1, 1, 1, 0] |    14
-------------------+--------------------+--------------+-------
 (^3 + ^2 + )^2    |     ^3 +  + 1      | [1, 0, 1, 1] |    11
-------------------+--------------------+--------------+-------
 (^3 + ^2 + )^3    |         ^3         | [1, 0, 0, 0] |     8
-------------------+--------------------+--------------+-------
 (^3 + ^2 + )^4    |       ^3 + 1       | [1, 0, 0, 1] |     9
-------------------+--------------------+--------------+-------
 (^3 + ^2 + )^5    |     ^2 +  + 1      | [0, 1, 1, 1] |     7
-------------------+--------------------+--------------+-------
 (^3 + ^2 + )^6    |      ^3 + ^2       | [1, 1, 0, 0] |    12
-------------------+--------------------+--------------+-------
 (^3 + ^2 + )^7    |         ^2         | [0, 1, 0, 0] |     4
-------------------+--------------------+--------------+-------
 (^3 + ^2 + )^8    |    ^3 + ^2 + 1     | [1, 1, 0, 1] |    13
-------------------+--------------------+--------------+-------
 (^3 + ^2 + )^9    |       ^3 +         | [1, 0, 1, 0] |    10
-------------------+--------------------+--------------+-------
 (^3 + ^2 + )^10   |       ^2 +         | [0, 1, 1, 0] |     6
-------------------+--------------------+--------------+-------
 (^3 + ^2 + )^11   |                    | [0, 0, 1, 0] |     2
-------------------+--------------------+--------------+-------
 (^3 + ^2 + )^12   | ^3 + ^2 +  + 1     | [1, 1, 1, 1] |    15
-------------------+--------------------+--------------+-------
 (^3 + ^2 + )^13   |       ^2 + 1       | [0, 1, 0, 1] |     5
-------------------+--------------------+--------------+-------
 (^3 + ^2 + )^14   |         + 1        | [0, 0, 1, 1] |     3
```

property characteristic

The prime characteristic p of the Galois field $\mathrm{GF}(p^m)$. Adding p copies of any element will always result in 0.

Examples

```
In [1]: GF = galois.GF(2**8)

In [2]: GF.characteristic
Out[2]: 2

In [3]: a = GF.Random(); a
Out[3]: GF(144, order=2^8)

In [4]: a * GF.characteristic
Out[4]: GF(0, order=2^8)
```

```
In [5]: GF = galois.GF(31)

In [6]: GF.characteristic
Out[6]: 31

In [7]: a = GF.Random(); a
Out[7]: GF(7, order=31)

In [8]: a * GF.characteristic
Out[8]: GF(0, order=31)
```

Type int

property default_ufunc_mode

The default ufunc arithmetic mode for this Galois field.

Examples

```
In [1]: galois.GF(2).default_ufunc_mode
Out[1]: 'jit-calculate'

In [2]: galois.GF(2**8).default_ufunc_mode
Out[2]: 'jit-lookup'

In [3]: galois.GF(31).default_ufunc_mode
Out[3]: 'jit-lookup'

In [4]: galois.GF(2**100).default_ufunc_mode
Out[4]: 'python-calculate'
```

Type str

property degree

The prime characteristic's degree m of the Galois field $\mathrm{GF}(p^m)$. The degree is a positive integer.

Examples

```
In [1]: galois.GF(2).degree
Out[1]: 1

In [2]: galois.GF(2**8).degree
Out[2]: 8

In [3]: galois.GF(31).degree
Out[3]: 1

In [4]: galois.GF(7**5).degree
Out[4]: 5
```

Type int

property display_mode

The representation of Galois field elements, either `"int"`, `"poly"`, or `"power"`. This can be changed with *display()*.

Examples

For the polynomial representation, when the primitive element is $x \in \mathrm{GF}(p)[x]$ the polynomial indeterminate used is .

```
In [1]: GF = galois.GF(2**8)

In [2]: a = GF.Random()

In [3]: print(GF.display_mode, a)
int GF(240, order=2^8)

In [4]: with GF.display("poly"):
   ...:     print(GF.display_mode, a)
   ...:
poly GF(^7 + ^6 + ^5 + ^4, order=2^8)

# The display mode is reset after exiting the context manager
In [5]: print(GF.display_mode, a)
int GF(240, order=2^8)
```

But when the primitive element is not $x \in \mathrm{GF}(p)[x]$, the polynomial indeterminate used is `x`.

```
In [6]: GF = galois.GF(2**8, irreducible_poly=galois.Poly.Degrees([8,4,3,1,0]))

In [7]: a = GF.Random()

In [8]: print(GF.display_mode, a)
int GF(61, order=2^8)

In [9]: with GF.display("poly"):
   ...:     print(GF.display_mode, a)
   ...:
poly GF(x^5 + x^4 + x^3 + x^2 + 1, order=2^8)

# The display mode is reset after exiting the context manager
In [10]: print(GF.display_mode, a)
int GF(61, order=2^8)
```

The power representation displays elements as powers of α the prrimitive element, see *FieldClass.primitive_element*.

```
In [11]: with GF.display("power"):
   ....:     print(GF.display_mode, a)
   ....:
power GF(^147, order=2^8)

# The display mode is reset after exiting the context manager
```

(continues on next page)

(continued from previous page)

```
In [12]: print(GF.display_mode, a)
int GF(61, order=2^8)
```

Type str

property dtypes

List of valid integer `numpy.dtype` objects that are compatible with this Galois field.

Examples

```
In [1]: GF = galois.GF(2); GF.dtypes
Out[1]:
[numpy.uint8,
 numpy.uint16,
 numpy.uint32,
 numpy.int8,
 numpy.int16,
 numpy.int32,
 numpy.int64]

In [2]: GF = galois.GF(2**8); GF.dtypes
Out[2]:
[numpy.uint8,
 numpy.uint16,
 numpy.uint32,
 numpy.int16,
 numpy.int32,
 numpy.int64]

In [3]: GF = galois.GF(31); GF.dtypes
Out[3]:
[numpy.uint8,
 numpy.uint16,
 numpy.uint32,
 numpy.int8,
 numpy.int16,
 numpy.int32,
 numpy.int64]

In [4]: GF = galois.GF(7**5); GF.dtypes
Out[4]: [numpy.uint16, numpy.uint32, numpy.int16, numpy.int32, numpy.int64]
```

For Galois fields that cannot be represented by `numpy.int64`, the only valid dtype is `numpy.object_`.

```
In [5]: GF = galois.GF(2**100); GF.dtypes
Out[5]: [numpy.object_]

In [6]: GF = galois.GF(36893488147419103183); GF.dtypes
Out[6]: [numpy.object_]
```

Type list

property irreducible_poly

The irreducible polynomial $f(x)$ of the Galois field $\mathrm{GF}(p^m)$. The irreducible polynomial is of degree m over $\mathrm{GF}(p)$.

Examples

```
In [1]: galois.GF(2).irreducible_poly
Out[1]: Poly(x + 1, GF(2))

In [2]: galois.GF(2**8).irreducible_poly
Out[2]: Poly(x^8 + x^4 + x^3 + x^2 + 1, GF(2))

In [3]: galois.GF(31).irreducible_poly
Out[3]: Poly(x + 28, GF(31))

In [4]: galois.GF(7**5).irreducible_poly
Out[4]: Poly(x^5 + x + 4, GF(7))
```

Type *galois.Poly*

property is_extension_field

Indicates if the field's order is a prime power.

Examples

```
In [1]: galois.GF(2).is_extension_field
Out[1]: False

In [2]: galois.GF(2**8).is_extension_field
Out[2]: True

In [3]: galois.GF(31).is_extension_field
Out[3]: False

In [4]: galois.GF(7**5).is_extension_field
Out[4]: True
```

Type bool

property is_prime_field

Indicates if the field's order is prime.

Examples

```
In [1]: galois.GF(2).is_prime_field
Out[1]: True
```

(continues on next page)

(continued from previous page)

```
In [2]: galois.GF(2**8).is_prime_field
Out[2]: False

In [3]: galois.GF(31).is_prime_field
Out[3]: True

In [4]: galois.GF(7**5).is_prime_field
Out[4]: False
```

Type bool

property is_primitive_poly

Indicates whether the *irreducible_poly* is a primitive polynomial.

Examples

```
In [1]: GF = galois.GF(2**8)

In [2]: GF.irreducible_poly
Out[2]: Poly(x^8 + x^4 + x^3 + x^2 + 1, GF(2))

In [3]: GF.primitive_element
Out[3]: GF(2, order=2^8)

# The irreducible polynomial is a primitive polynomial is the primitive element␣
↪is a root
In [4]: GF.irreducible_poly(GF.primitive_element, field=GF)
Out[4]: GF(0, order=2^8)

In [5]: GF.is_primitive_poly
Out[5]: True
```

```
# Field used in AES
In [6]: GF = galois.GF(2**8, irreducible_poly=galois.Poly.Degrees([8,4,3,1,0]))

In [7]: GF.irreducible_poly
Out[7]: Poly(x^8 + x^4 + x^3 + x + 1, GF(2))

In [8]: GF.primitive_element
Out[8]: GF(3, order=2^8)

# The irreducible polynomial is a primitive polynomial is the primitive element␣
↪is a root
In [9]: GF.irreducible_poly(GF.primitive_element, field=GF)
Out[9]: GF(6, order=2^8)

In [10]: GF.is_primitive_poly
Out[10]: False
```

Type bool

property name
The Galois field name.

Examples

```
In [1]: galois.GF(2).name
Out[1]: 'GF(2)'

In [2]: galois.GF(2**8).name
Out[2]: 'GF(2^8)'

In [3]: galois.GF(31).name
Out[3]: 'GF(31)'

In [4]: galois.GF(7**5).name
Out[4]: 'GF(7^5)'
```

Type str

property order
The order p^m of the Galois field $\mathrm{GF}(p^m)$. The order of the field is also equal to the field's size.

Examples

```
In [1]: galois.GF(2).order
Out[1]: 2

In [2]: galois.GF(2**8).order
Out[2]: 256

In [3]: galois.GF(31).order
Out[3]: 31

In [4]: galois.GF(7**5).order
Out[4]: 16807
```

Type int

property prime_subfield
The prime subfield $\mathrm{GF}(p)$ of the extension field $\mathrm{GF}(p^m)$.

Examples

```
In [1]: print(galois.GF(2).prime_subfield.properties)
GF(2):
  characteristic: 2
  degree: 1
```

(continues on next page)

(continued from previous page)

```
  order: 2

In [2]: print(galois.GF(2**8).prime_subfield.properties)
GF(2):
  characteristic: 2
  degree: 1
  order: 2

In [3]: print(galois.GF(31).prime_subfield.properties)
GF(31):
  characteristic: 31
  degree: 1
  order: 31

In [4]: print(galois.GF(7**5).prime_subfield.properties)
GF(7):
  characteristic: 7
  degree: 1
  order: 7
```

Type *galois.FieldClass*

property primitive_element

A primitive element α of the Galois field $\mathrm{GF}(p^m)$. A primitive element is a multiplicative generator of the field, such that $\mathrm{GF}(p^m) = \{0, 1, \alpha, \alpha^2, \ldots, \alpha^{p^m-2}\}$.

A primitive element is a root of the primitive polynomial $f(x)$, such that $f(\alpha) = 0$ over $\mathrm{GF}(p^m)$.

Examples

```
In [1]: galois.GF(2).primitive_element
Out[1]: GF(1, order=2)

In [2]: galois.GF(2**8).primitive_element
Out[2]: GF(2, order=2^8)

In [3]: galois.GF(31).primitive_element
Out[3]: GF(3, order=31)

In [4]: galois.GF(7**5).primitive_element
Out[4]: GF(7, order=7^5)
```

Type int

property primitive_elements

All primitive elements α of the Galois field $\mathrm{GF}(p^m)$. A primitive element is a multiplicative generator of the field, such that $\mathrm{GF}(p^m) = \{0, 1, \alpha, \alpha^2, \ldots, \alpha^{p^m-2}\}$.

Examples

```
In [1]: galois.GF(2).primitive_elements
Out[1]: GF([1], order=2)

In [2]: galois.GF(2**8).primitive_elements
Out[2]:
GF([  2,   4,   6,   9,  13,  14,  16,  18,  19,  20,  22,  24,  25,  27,
     29,  30,  31,  34,  35,  40,  42,  43,  48,  49,  50,  52,  57,  60,
     63,  65,  66,  67,  71,  72,  73,  74,  75,  76,  81,  82,  83,  84,
     88,  90,  91,  92,  93,  95,  98,  99, 104, 105, 109, 111, 112, 113,
    118, 119, 121, 122, 123, 126, 128, 129, 131, 133, 135, 136, 137, 140,
    141, 142, 144, 148, 149, 151, 154, 155, 157, 158, 159, 162, 163, 164,
    165, 170, 171, 175, 176, 177, 178, 183, 187, 188, 189, 192, 194, 198,
    199, 200, 201, 202, 203, 204, 209, 210, 211, 212, 213, 216, 218, 222,
    224, 225, 227, 229, 232, 234, 236, 238, 240, 243, 246, 247, 248, 249,
    250, 254], order=2^8)

In [3]: galois.GF(31).primitive_elements
Out[3]: GF([ 3, 11, 12, 13, 17, 21, 22, 24], order=31)

In [4]: galois.GF(7**5).primitive_elements
Out[4]: GF([    7,     8,    14, ..., 16797, 16798, 16803], order=7^5)
```

Type int

property properties

A formatted string displaying relevant properties of the Galois field.

Examples

```
In [1]: GF = galois.GF(2); print(GF.properties)
GF(2):
  characteristic: 2
  degree: 1
  order: 2

In [2]: GF = galois.GF(2**8); print(GF.properties)
GF(2^8):
  characteristic: 2
  degree: 8
  order: 256
  irreducible_poly: Poly(x^8 + x^4 + x^3 + x^2 + 1, GF(2))
  is_primitive_poly: True
  primitive_element: GF(2, order=2^8)

In [3]: GF = galois.GF(31); print(GF.properties)
GF(31):
  characteristic: 31
  degree: 1
  order: 31

In [4]: GF = galois.GF(7**5); print(GF.properties)
```

(continues on next page)

(continued from previous page)

```
GF(7^5):
  characteristic: 7
  degree: 5
  order: 16807
  irreducible_poly: Poly(x^5 + x + 4, GF(7))
  is_primitive_poly: True
  primitive_element: GF(7, order=7^5)
```

Type str

property ufunc_mode

The mode for ufunc compilation, either "jit-lookup", "jit-calculate", "python-calculate".

Examples

```
In [1]: galois.GF(2).ufunc_mode
Out[1]: 'jit-calculate'

In [2]: galois.GF(2**8).ufunc_mode
Out[2]: 'jit-lookup'

In [3]: galois.GF(31).ufunc_mode
Out[3]: 'jit-lookup'

In [4]: galois.GF(7**5).ufunc_mode
Out[4]: 'jit-lookup'
```

Type str

property ufunc_modes

All supported ufunc modes for this Galois field array class.

Examples

```
In [1]: galois.GF(2).ufunc_modes
Out[1]: ['jit-calculate']

In [2]: galois.GF(2**8).ufunc_modes
Out[2]: ['jit-lookup', 'jit-calculate']

In [3]: galois.GF(31).ufunc_modes
Out[3]: ['jit-lookup', 'jit-calculate']

In [4]: galois.GF(2**100).ufunc_modes
Out[4]: ['python-calculate']
```

Type list

Pre-made Galois field classes

GF2(array[, dtype, copy, order, ndmin])	Creates an array over $\mathrm{GF}(2)$.

galois.GF2

class galois.**GF2**(*array*, *dtype=None*, *copy=True*, *order='K'*, *ndmin=0*)

Creates an array over $\mathrm{GF}(2)$.

This class is a subclass of *galois.FieldArray* and instance of *galois.FieldClass*.

Parameters

- **array** (*array_like*) – The input array to be converted to a Galois field array. The input array is copied, so the original array is unmodified by changes to the Galois field array. Valid input array types are `numpy.ndarray`, `list` or `tuple` of int or str, `int`, or `str`.
- **dtype** (*numpy.dtype, optional*) – The `numpy.dtype` of the array elements. The default is `None` which represents the smallest valid dtype for this class, i.e. the first element in *galois.FieldClass.dtypes*.
- **copy** (*bool, optional*) – The `copy` keyword argument from `numpy.array()`. The default is `True` which makes a copy of the input object is it's an array.
- **order** ({"K", "A", "C", "F"}, optional) – The `order` keyword argument from `numpy.array()`. Valid values are `"K"` (default), `"A"`, `"C"`, or `"F"`.
- **ndmin** (*int, optional*) – The `ndmin` keyword argument from `numpy.array()`. The minimum number of dimensions of the output. The default is 0.

Examples

This class is equivalent (and, in fact, identical) to the class returned from the Galois field array class constructor.

```
In [1]: print(galois.GF2)
<class 'numpy.ndarray over GF(2)'>

In [2]: GF2 = galois.GF(2); print(GF2)
<class 'numpy.ndarray over GF(2)'>

In [3]: GF2 is galois.GF2
Out[3]: True
```

The Galois field properties can be viewed by class attributes, see *galois.FieldClass*.

```
# View a summary of the field's properties
In [4]: print(galois.GF2.properties)
GF(2):
  characteristic: 2
  degree: 1
  order: 2

# Or access each attribute individually
In [5]: galois.GF2.irreducible_poly
Out[5]: Poly(x + 1, GF(2))
```

(continues on next page)

(continued from previous page)

```
In [6]: galois.GF2.is_prime_field
Out[6]: True
```

The class's constructor mimics the call signature of `numpy.array()`.

```
# Construct a Galois field array from an iterable
In [7]: galois.GF2([1,0,1,1,0,0,0,1])
Out[7]: GF([1, 0, 1, 1, 0, 0, 0, 1], order=2)

# Or an iterable of iterables
In [8]: galois.GF2([[1,0],[1,1]])
Out[8]:
GF([[1, 0],
    [1, 1]], order=2)

# Or a single integer
In [9]: galois.GF2(1)
Out[9]: GF(1, order=2)
```

Constructors

Elements([dtype])	Creates a Galois field array of the field's elements $\{0, \ldots, p^m - 1\}$.
Identity(size[, dtype])	Creates an $n \times n$ Galois field identity matrix.
Ones(shape[, dtype])	Creates a Galois field array with all ones.
Random([shape, low, high, dtype])	Creates a Galois field array with random field elements.
Range(start, stop[, step, dtype])	Creates a Galois field array with a range of field elements.
Vandermonde(a, m, n[, dtype])	Creates a $m \times n$ Vandermonde matrix of $a \in \mathrm{GF}(p^m)$.
Vector(array[, dtype])	Creates a Galois field array over $\mathrm{GF}(p^m)$ from length-m vectors over the prime subfield $\mathrm{GF}(p)$.
Zeros(shape[, dtype])	Creates a Galois field array with all zeros.

Methods

`lu_decompose`()	Decomposes the input array into the product of lower and upper triangular matrices.
`lup_decompose`()	Decomposes the input array into the product of lower and upper triangular matrices using partial pivoting.
`row_reduce`([ncols])	Performs Gaussian elimination on the matrix to achieve reduced row echelon form.
`vector`([dtype])	Converts the Galois field array over $\mathrm{GF}(p^m)$ to length-m vectors over the prime subfield $\mathrm{GF}(p)$.

classmethod Elements(*dtype=None*)

Creates a Galois field array of the field's elements $\{0, \ldots, p^m - 1\}$.

Parameters **dtype** (*numpy.dtype, optional*) – The numpy.dtype of the array elements. The default is None which represents the smallest valid dtype for this class, i.e. the first element in *galois.FieldClass.dtypes*.

Returns A Galois field array of all the field's elements.

Return type *galois.FieldArray*

Examples

```
In [10]: GF = galois.GF(31)

In [11]: GF.Elements()
Out[11]:
GF([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
    17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30], order=31)
```

classmethod Identity(*size*, *dtype=None*)

Creates an $n \times n$ Galois field identity matrix.

Parameters

- **size** (*int*) – The size n along one axis of the matrix. The resulting array has shape (size,size).
- **dtype** (*numpy.dtype, optional*) – The numpy.dtype of the array elements. The default is None which represents the smallest valid dtype for this class, i.e. the first element in *galois.FieldClass.dtypes*.

Returns A Galois field identity matrix of shape (size, size).

Return type *galois.FieldArray*

Examples

```
In [12]: GF = galois.GF(31)

In [13]: GF.Identity(4)
Out[13]:
GF([[1, 0, 0, 0],
    [0, 1, 0, 0],
    [0, 0, 1, 0],
    [0, 0, 0, 1]], order=31)
```

classmethod Ones(*shape*, *dtype=None*)

Creates a Galois field array with all ones.

Parameters

- **shape** (*tuple*) – A numpy-compliant shape tuple, see numpy.ndarray.shape. An empty tuple () represents a scalar. A single integer or 1-tuple, e.g. N or (N,), represents the size of a 1-dim array. An n-tuple, e.g. (M,N), represents an n-dim array with each element indicating the size in each dimension.

- **dtype** (*numpy.dtype, optional*) – The numpy.dtype of the array elements. The default is None which represents the smallest valid dtype for this class, i.e. the first element in *galois.FieldClass.dtypes*.

Returns A Galois field array of ones.

Return type *galois.FieldArray*

Examples

```
In [14]: GF = galois.GF(31)

In [15]: GF.Ones((2,5))
Out[15]:
GF([[1, 1, 1, 1, 1],
    [1, 1, 1, 1, 1]], order=31)
```

classmethod Random(*shape=()*, *low=0*, *high=None*, *dtype=None*)

Creates a Galois field array with random field elements.

Parameters

- **shape** (*tuple*) – A numpy-compliant shape tuple, see numpy.ndarray.shape. An empty tuple () represents a scalar. A single integer or 1-tuple, e.g. N or (N,), represents the size of a 1-dim array. An n-tuple, e.g. (M,N), represents an n-dim array with each element indicating the size in each dimension.
- **low** (*int, optional*) – The lowest value (inclusive) of a random field element. The default is 0.
- **high** (*int, optional*) – The highest value (exclusive) of a random field element. The default is None which represents the field's order p^m.
- **dtype** (*numpy.dtype, optional*) – The numpy.dtype of the array elements. The default is None which represents the smallest valid dtype for this class, i.e. the first element in *galois.FieldClass.dtypes*.

Returns A Galois field array of random field elements.

Return type *galois.FieldArray*

Examples

```
In [16]: GF = galois.GF(31)

In [17]: GF.Random((2,5))
Out[17]:
GF([[ 2,  2, 20, 20, 10],
    [12, 22, 23, 20, 30]], order=31)
```

classmethod Range(*start*, *stop*, *step=1*, *dtype=None*)

Creates a Galois field array with a range of field elements.

Parameters

- **start** (*int*) – The starting value (inclusive).

- **stop** (*int*) – The stopping value (exclusive).
- **step** (*int, optional*) – The space between values. The default is 1.
- **dtype** (*numpy.dtype, optional*) – The `numpy.dtype` of the array elements. The default is `None` which represents the smallest valid dtype for this class, i.e. the first element in *galois.FieldClass.dtypes*.

Returns A Galois field array of a range of field elements.

Return type *galois.FieldArray*

Examples

```
In [18]: GF = galois.GF(31)

In [19]: GF.Range(10,20)
Out[19]: GF([10, 11, 12, 13, 14, 15, 16, 17, 18, 19], order=31)
```

classmethod Vandermonde(*a*, *m*, *n*, *dtype=None*)

Creates a $m \times n$ Vandermonde matrix of $a \in \mathrm{GF}(p^m)$.

Parameters

- **a** (*int, galois.FieldArray*) – An element of $\mathrm{GF}(p^m)$.
- **m** (*int*) – The number of rows in the Vandermonde matrix.
- **n** (*int*) – The number of columns in the Vandermonde matrix.
- **dtype** (*numpy.dtype, optional*) – The `numpy.dtype` of the array elements. The default is `None` which represents the smallest valid dtype for this class, i.e. the first element in *galois.FieldClass.dtypes*.

Returns The $m \times n$ Vandermonde matrix.

Return type *galois.FieldArray*

Examples

```
In [20]: GF = galois.GF(2**3)

In [21]: a = GF.primitive_element

In [22]: V = GF.Vandermonde(a, 7, 7)

In [23]: with GF.display("power"):
   ....:     print(V)
   ....:
GF([[  1,   1,   1,   1,   1,   1,   1],
    [  1,   , ^2, ^3, ^4, ^5, ^6],
    [  1, ^2, ^4, ^6,   , ^3, ^5],
    [  1, ^3, ^6, ^2, ^5,   , ^4],
    [  1, ^4,   , ^5, ^2, ^6, ^3],
    [  1, ^5, ^3,   , ^6, ^4, ^2],
    [  1, ^6, ^5, ^4, ^3, ^2,   ]], order=2^3)
```

classmethod Vector(*array*, *dtype=None*)

Creates a Galois field array over $\mathrm{GF}(p^m)$ from length-m vectors over the prime subfield $\mathrm{GF}(p)$.

Parameters

- **array** (*array_like*) – The input array with field elements in $\mathrm{GF}(p)$ to be converted to a Galois field array in $\mathrm{GF}(p^m)$. The last dimension of the input array must be m. An input array with shape `(n1, n2, m)` has output shape `(n1, n2)`.
- **dtype** (*numpy.dtype, optional*) – The `numpy.dtype` of the array elements. The default is `None` which represents the smallest valid dtype for this class, i.e. the first element in *galois.FieldClass.dtypes*.

Returns A Galois field array over $\mathrm{GF}(p^m)$.

Return type *galois.FieldArray*

Examples

```
In [24]: GF = galois.GF(2**6)

In [25]: vec = galois.GF2.Random((3,6)); vec
Out[25]:
GF([[1, 0, 0, 1, 1, 1],
    [0, 1, 1, 0, 1, 1],
    [0, 1, 0, 0, 1, 1]], order=2)

In [26]: a = GF.Vector(vec); a
Out[26]: GF([39, 27, 19], order=2^6)

In [27]: with GF.display("poly"):
   ....:     print(a)
   ....:
GF([^5 + ^2 +  + 1, ^4 + ^3 +  + 1, ^4 +  + 1], order=2^6)

In [28]: a.vector()
Out[28]:
GF([[1, 0, 0, 1, 1, 1],
    [0, 1, 1, 0, 1, 1],
    [0, 1, 0, 0, 1, 1]], order=2)
```

classmethod Zeros(*shape*, *dtype=None*)

Creates a Galois field array with all zeros.

Parameters

- **shape** (*tuple*) – A numpy-compliant `shape` tuple, see `numpy.ndarray.shape`. An empty tuple `()` represents a scalar. A single integer or 1-tuple, e.g. `N` or `(N,)`, represents the size of a 1-dim array. An n-tuple, e.g. `(M,N)`, represents an n-dim array with each element indicating the size in each dimension.
- **dtype** (*numpy.dtype, optional*) – The `numpy.dtype` of the array elements. The default is `None` which represents the smallest valid dtype for this class, i.e. the first element in *galois.FieldClass.dtypes*.

Returns A Galois field array of zeros.

Return type *galois.FieldArray*

Examples

```
In [29]: GF = galois.GF(31)

In [30]: GF.Zeros((2,5))
Out[30]:
GF([[0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0]], order=31)
```

5.1.2 Prime field functions

Primitive roots

primitive_root(n[, start, stop, reverse])	Finds the smallest primitive root modulo n.
primitive_roots(n[, start, stop, reverse])	Finds all primitive roots modulo n.
is_primitive_root(g, n)	Determines if g is a primitive root modulo n.

galois.primitive_root

galois.**primitive_root**(*n*, *start=1*, *stop=None*, *reverse=False*)

Finds the smallest primitive root modulo n.

g is a primitive root if the totatives of n, the positive integers $1 \leq a < n$ that are coprime with n, can be generated by powers of g.

Alternatively said, g is a primitive root modulo n if and only if g is a generator of the multiplicative group of integers modulo n, $\mathbb{Z}_n^\times$. That is, $\mathbb{Z}_n^\times = \{g, g^2, \ldots, g^k\}$, where k is order of the group. The order of the group $\mathbb{Z}_n^\times$ is defined by Euler's totient function, $\phi(n) = k$. If $\mathbb{Z}_n^\times$ is cyclic, the number of primitive roots modulo n is given by $\phi(k)$ or $\phi(\phi(n))$.

See *galois.is_cyclic*.

Parameters

- **n** (*int*) – A positive integer.
- **start** (*int, optional*) – Starting value (inclusive) in the search for a primitive root. The default is 1. The resulting primitive root, if found, will be start $\leq g <$ stop.
- **stop** (*int, optional*) – Stopping value (exclusive) in the search for a primitive root. The default is None which corresponds to n. The resulting primitive root, if found, will be start $\leq g <$ stop.
- **reverse** (*bool, optional*) – Search for a primitive root in reverse order, i.e. find the largest primitive root first. Default is False.

Returns The smallest primitive root modulo n. Returns None if no primitive roots exist.

Return type int

References

- V. Shoup. Searching for primitive roots in finite fields. https://www.ams.org/journals/mcom/1992-58-197/S0025-5718-1992-1106981-9/S0025-5718-1992-1106981-9.pdf
- L. K. Hua. On the least primitive root of a prime. https://www.ams.org/journals/bull/1942-48-10/S0002-9904-1942-07767-6/S0002-9904-1942-07767-6.pdf
- https://en.wikipedia.org/wiki/Finite_field#Roots_of_unity
- https://en.wikipedia.org/wiki/Primitive_root_modulo_n
- http://www.numbertheory.org/courses/MP313/lectures/lecture7/page1.html

---

Examples

Here is an example with one primitive root, $n = 6 = 2 * 3^1$, which fits the definition of cyclicness, see *galois.is_cyclic*. Because $n = 6$ is not prime, the primitive root isn't a multiplicative generator of $\mathbb{Z}/n\mathbb{Z}$.

```
In [1]: n = 6

In [2]: root = galois.primitive_root(n); root
Out[2]: 5

# The congruence class coprime with n
In [3]: Znx = set([a for a in range(1, n) if math.gcd(n, a) == 1]); Znx
Out[3]: {1, 5}

# Euler's totient function counts the "totatives", positive integers coprime with n
In [4]: phi = galois.euler_phi(n); phi
Out[4]: 2

In [5]: len(Znx) == phi
Out[5]: True

# The primitive roots are the elements in Znx that multiplicatively generate the␣
↪group
In [6]: for a in Znx:
   ...:     span = set([pow(a, i, n) for i in range(1, phi + 1)])
   ...:     primitive_root = span == Znx
   ...:     print("Element: {}, Span: {:<6}, Primitive root: {}".format(a,␣
↪str(span), primitive_root))
   ...:
Element: 1, Span: {1}   , Primitive root: False
Element: 5, Span: {1, 5}, Primitive root: True
```

Here is an example with two primitive roots, $n = 7 = 7^1$, which fits the definition of cyclicness, see *galois.is_cyclic*. Since $n = 7$ is prime, the primitive root is a multiplicative generator of $\mathbb{Z}/n\mathbb{Z}$.

```
In [7]: n = 7

In [8]: root = galois.primitive_root(n); root
Out[8]: 3

# The congruence class coprime with n
```

(continues on next page)

(continued from previous page)

```
In [9]: Znx = set([a for a in range(1, n) if math.gcd(n, a) == 1]); Znx
Out[9]: {1, 2, 3, 4, 5, 6}

# Euler's totient function counts the "totatives", positive integers coprime with n
In [10]: phi = galois.euler_phi(n); phi
Out[10]: 6

In [11]: len(Znx) == phi
Out[11]: True

# The primitive roots are the elements in Znx that multiplicatively generate the␣
↪group
In [12]: for a in Znx:
   ....:     span = set([pow(a, i, n) for i in range(1, phi + 1)])
   ....:     primitive_root = span == Znx
   ....:     print("Element: {}, Span: {:<18}, Primitive root: {}".format(a,␣
↪str(span), primitive_root))
   ....:
Element: 1, Span: {1}               , Primitive root: False
Element: 2, Span: {1, 2, 4}         , Primitive root: False
Element: 3, Span: {1, 2, 3, 4, 5, 6}, Primitive root: True
Element: 4, Span: {1, 2, 4}         , Primitive root: False
Element: 5, Span: {1, 2, 3, 4, 5, 6}, Primitive root: True
Element: 6, Span: {1, 6}            , Primitive root: False
```

The algorithm is also efficient for very large n.

```
In [13]: n = 1000000000000000035000061

In [14]: galois.primitive_root(n)
Out[14]: 7

In [15]: galois.primitive_root(n, reverse=True)
Out[15]: 1000000000000000035000054
```

Here is a counterexample with no primitive roots, $n = 8 = 2^3$, which does not fit the definition of cyclicness, see *galois.is_cyclic*.

```
In [16]: n = 8

In [17]: root = galois.primitive_root(n); root

# The congruence class coprime with n
In [18]: Znx = set([a for a in range(1, n) if math.gcd(n, a) == 1]); Znx
Out[18]: {1, 3, 5, 7}

# Euler's totient function counts the "totatives", positive integers coprime with n
In [19]: phi = galois.euler_phi(n); phi
Out[19]: 4

In [20]: len(Znx) == phi
Out[20]: True
```

(continues on next page)

(continued from previous page)

```
# Test all elements for being primitive roots. The powers of a primitive span the␣
↪congruence classes mod n.
In [21]: for a in Znx:
   ....:     span = set([pow(a, i, n) for i in range(1, phi + 1)])
   ....:     primitive_root = span == Znx
   ....:     print("Element: {}, Span: {:<6}, Primitive root: {}".format(a,␣
↪str(span), primitive_root))
   ....:
Element: 1, Span: {1}   , Primitive root: False
Element: 3, Span: {1, 3}, Primitive root: False
Element: 5, Span: {1, 5}, Primitive root: False
Element: 7, Span: {1, 7}, Primitive root: False

# Note the max order of any element is 2, not 4, which is Carmichael's lambda␣
↪function
In [22]: galois.carmichael_lambda(n)
Out[22]: 2
```

galois.primitive_roots

galois.**primitive_roots**(*n*, *start=1*, *stop=None*, *reverse=False*)

Finds all primitive roots modulo n.

g is a primitive root if the totatives of n, the positive integers $1 \leq a < n$ that are coprime with n, can be generated by powers of g.

Alternatively said, g is a primitive root modulo n if and only if g is a generator of the multiplicative group of integers modulo n, $\mathbb{Z}_n^\times$. That is, $\mathbb{Z}_n^\times = \{g, g^2, \ldots, g^k\}$, where k is order of the group. The order of the group $\mathbb{Z}_n^\times$ is defined by Euler's totient function, $\phi(n) = k$. If $\mathbb{Z}_n^\times$ is cyclic, the number of primitive roots modulo n is given by $\phi(k)$ or $\phi(\phi(n))$.

See *galois.is_cyclic*.

Parameters

- **n** (*int*) – A positive integer.
- **start** (*int, optional*) – Starting value (inclusive) in the search for a primitive root. The default is 1. The resulting primitive roots, if found, will be start $\leq x <$ stop.
- **stop** (*int, optional*) – Stopping value (exclusive) in the search for a primitive root. The default is `None` which corresponds to `n`. The resulting primitive roots, if found, will be start $\leq x <$ stop.
- **reverse** (*bool, optional*) – List all primitive roots in descending order, i.e. largest to smallest. Default is `False`.

Returns All the positive primitive n-th roots of unity, x.

Return type list

References

- V. Shoup. Searching for primitive roots in finite fields. https://www.ams.org/journals/mcom/1992-58-197/S0025-5718-1992-1106981-9/S0025-5718-1992-1106981-9.pdf
- https://en.wikipedia.org/wiki/Finite_field#Roots_of_unity
- https://en.wikipedia.org/wiki/Primitive_root_modulo_n
- http://www.numbertheory.org/courses/MP313/lectures/lecture7/page1.html

---

Examples

Here is an example with one primitive root, $n = 6 = 2 * 3^1$, which fits the definition of cyclicness, see *galois.is_cyclic*. Because $n = 6$ is not prime, the primitive root isn't a multiplicative generator of $\mathbb{Z}/\mathbf{n}\mathbb{Z}$.

```
In [1]: n = 6

In [2]: roots = galois.primitive_roots(n); roots
Out[2]: [5]

# The congruence class coprime with n
In [3]: Znx = set([a for a in range(1, n) if math.gcd(n, a) == 1]); Znx
Out[3]: {1, 5}

# Euler's totient function counts the "totatives", positive integers coprime with n
In [4]: phi = galois.euler_phi(n); phi
Out[4]: 2

In [5]: len(Znx) == phi
Out[5]: True

# Test all elements for being primitive roots. The powers of a primitive span the␣
↪congruence classes mod n.
In [6]: for a in Znx:
   ...:     span = set([pow(a, i, n) for i in range(1, phi + 1)])
   ...:     primitive_root = span == Znx
   ...:     print("Element: {}, Span: {:<6}, Primitive root: {}".format(a,␣
↪str(span), primitive_root))
   ...:
Element: 1, Span: {1}   , Primitive root: False
Element: 5, Span: {1, 5}, Primitive root: True

# Euler's totient function phi(phi(n)) counts the primitive roots of n
In [7]: len(roots) == galois.euler_phi(phi)
Out[7]: True
```

Here is an example with two primitive roots, $n = 7 = 7^1$, which fits the definition of cyclicness, see *galois.is_cyclic*. Since $n = 7$ is prime, the primitive root is a multiplicative generator of $\mathbb{Z}/\mathbf{n}\mathbb{Z}$.

```
In [8]: n = 7

In [9]: roots = galois.primitive_roots(n); roots
Out[9]: [3, 5]
```

(continues on next page)

(continued from previous page)

```
# The congruence class coprime with n
In [10]: Znx = set([a for a in range(1, n) if math.gcd(n, a) == 1]); Znx
Out[10]: {1, 2, 3, 4, 5, 6}

# Euler's totient function counts the "totatives", positive integers coprime with n
In [11]: phi = galois.euler_phi(n); phi
Out[11]: 6

In [12]: len(Znx) == phi
Out[12]: True

# Test all elements for being primitive roots. The powers of a primitive span the␣
↪congruence classes mod n.
In [13]: for a in Znx:
   ....:     span = set([pow(a, i, n) for i in range(1, phi + 1)])
   ....:     primitive_root = span == Znx
   ....:     print("Element: {}, Span: {:<18}, Primitive root: {}".format(a,␣
↪str(span), primitive_root))
   ....:
Element: 1, Span: {1}                , Primitive root: False
Element: 2, Span: {1, 2, 4}          , Primitive root: False
Element: 3, Span: {1, 2, 3, 4, 5, 6}, Primitive root: True
Element: 4, Span: {1, 2, 4}          , Primitive root: False
Element: 5, Span: {1, 2, 3, 4, 5, 6}, Primitive root: True
Element: 6, Span: {1, 6}             , Primitive root: False

# Euler's totient function phi(phi(n)) counts the primitive roots of n
In [14]: len(roots) == galois.euler_phi(phi)
Out[14]: True
```

The algorithm is also efficient for very large n.

```
In [15]: n = 10000000000000000035000061

# Euler's totient function phi(phi(n)) counts the primitive roots of n
In [16]: galois.euler_phi(galois.euler_phi(n))
Out[16]: 237770895725348415787008

# Only find some of the primitive roots
In [17]: galois.primitive_roots(n, stop=100)
Out[17]: [7, 21, 28, 34, 35, 38, 39, 43, 47, 52, 58, 65, 67, 73, 88, 97, 98]

# The generator can also be used in a for loop
In [18]: for r in galois.primitive_roots(n, stop=100):
   ....:     print(r, end=" ")
   ....:
7 21 28 34 35 38 39 43 47 52 58 65 67 73 88 97 98
```

Here is a counterexample with no primitive roots, $n = 8 = 2^3$, which does not fit the definition of cyclicness, see *galois.is_cyclic*.

```
In [19]: n = 8

In [20]: roots = galois.primitive_roots(n); roots
Out[20]: []

# The congruence class coprime with n
In [21]: Znx = set([a for a in range(1, n) if math.gcd(n, a) == 1]); Znx
Out[21]: {1, 3, 5, 7}

# Euler's totient function counts the "totatives", positive integers coprime with n
In [22]: phi = galois.euler_phi(n); phi
Out[22]: 4

In [23]: len(Znx) == phi
Out[23]: True

# Test all elements for being primitive roots. The powers of a primitive span the␣
↪congruence classes mod n.
In [24]: for a in Znx:
   ....:     span = set([pow(a, i, n) for i in range(1, phi + 1)])
   ....:     primitive_root = span == Znx
   ....:     print("Element: {}, Span: {:<6}, Primitive root: {}".format(a,␣
↪str(span), primitive_root))
   ....:
Element: 1, Span: {1}   , Primitive root: False
Element: 3, Span: {1, 3}, Primitive root: False
Element: 5, Span: {1, 5}, Primitive root: False
Element: 7, Span: {1, 7}, Primitive root: False
```

galois.is_primitive_root

galois.**is_primitive_root**(g, n)

Determines if g is a primitive root modulo n.

g is a primitive root if the totatives of n, the positive integers $1 \leq a < n$ that are coprime with n, can be generated by powers of g.

Parameters

- **g** (*int*) – A positive integer that may be a primitive root modulo n.
- **n** (*int*) – A positive integer.

Returns `True` if g is a primitive root modulo n.

Return type bool

Examples

```
In [1]: galois.is_primitive_root(2, 7)
Out[1]: False

In [2]: galois.is_primitive_root(3, 7)
```

(continues on next page)

(continued from previous page)

```
Out[2]: True

In [3]: galois.primitive_roots(7)
Out[3]: [3, 5]
```

5.1.3 Extension field functions

Irreducible polynomials

irreducible_poly(characteristic, degree[, ...])	Returns a monic irreducible polynomial $f(x)$ over $\mathrm{GF}(p)$ with degree m.
irreducible_polys(characteristic, degree)	Returns all monic irreducible polynomials $f(x)$ over $\mathrm{GF}(p)$ with degree m.
is_irreducible(poly)	Checks whether the polynomial $f(x)$ over $\mathrm{GF}(p)$ is irreducible.

galois.irreducible_poly

galois.**irreducible_poly**(*characteristic*, *degree*, *method='min'*)

Returns a monic irreducible polynomial $f(x)$ over $\mathrm{GF}(p)$ with degree m.

If $f(x)$ is an irreducible polynomial over $\mathrm{GF}(p)$ and $a \in \mathrm{GF}(p)\backslash\{0\}$, then $a \cdot f(x)$ is also irreducible.

In addition to other applications, $f(x)$ produces the field extension $\mathrm{GF}(p^m)$ of $\mathrm{GF}(p)$.

Parameters

- **characteristic** (*int*) – The prime characteristic p of the field $\mathrm{GF}(p)$ that the polynomial is over.
- **degree** (*int*) – The degree m of the desired irreducible polynomial.
- **method** (*str, optional*) – The search method for finding the irreducible polynomial.
  - "min" (default): Returns the lexicographically-minimal monic irreducible polynomial.
  - "max": Returns the lexicographically-maximal monic irreducible polynomial.
  - "random": Returns a randomly generated degree-m monic irreducible polynomial.

Returns The degree-m monic irreducible polynomial over $\mathrm{GF}(p)$.

Return type *galois.Poly*

Examples

```
# The lexicographically-minimal irreducible polynomial
In [1]: p = galois.irreducible_poly(7, 5); p
Out[1]: Poly(x^5 + x + 3, GF(7))

In [2]: galois.is_irreducible(p)
Out[2]: True
```

(continues on next page)

(continued from previous page)

```
# The lexicographically-maximal irreducible polynomial
In [3]: p = galois.irreducible_poly(7, 5, method="max"); p
Out[3]: Poly(x^5 + 6x^4 + 6x^3 + 6x^2 + 6x + 6, GF(7))

In [4]: galois.is_irreducible(p)
Out[4]: True

# A random irreducible polynomial
In [5]: p = galois.irreducible_poly(7, 5, method="random"); p
Out[5]: Poly(x^5 + 4x^4 + 3x + 1, GF(7))

In [6]: galois.is_irreducible(p)
Out[6]: True
```

Products with non-zero constants are also irreducible.

```
In [7]: GF = galois.GF(7)

In [8]: p = galois.irreducible_poly(7, 5); p
Out[8]: Poly(x^5 + x + 3, GF(7))

In [9]: galois.is_irreducible(p)
Out[9]: True

In [10]: galois.is_irreducible(p * GF(3))
Out[10]: True
```

---

galois.irreducible_polys

galois.**irreducible_polys**(*characteristic*, *degree*)

Returns all monic irreducible polynomials $f(x)$ over $\mathrm{GF}(p)$ with degree m.

If $f(x)$ is an irreducible polynomial over $\mathrm{GF}(p)$ and $a \in \mathrm{GF}(p) \backslash \{0\}$, then $a \cdot f(x)$ is also irreducible.

In addition to other applications, $f(x)$ produces the field extension $\mathrm{GF}(p^m)$ of $\mathrm{GF}(p)$.

Parameters

- **characteristic** (*int*) – The prime characteristic p of the field $\mathrm{GF}(p)$ that the polynomial is over.
- **degree** (*int*) – The degree m of the desired irreducible polynomial.

Returns All degree-m monic irreducible polynomials over $\mathrm{GF}(p)$.

Return type list

Examples

```
In [1]: galois.irreducible_polys(2, 5)
Out[1]:
[Poly(x^5 + x^2 + 1, GF(2)),
 Poly(x^5 + x^3 + 1, GF(2)),
```

(continues on next page)

(continued from previous page)

```
Poly(x^5 + x^3 + x^2 + x + 1, GF(2)),
Poly(x^5 + x^4 + x^2 + x + 1, GF(2)),
Poly(x^5 + x^4 + x^3 + x + 1, GF(2)),
Poly(x^5 + x^4 + x^3 + x^2 + 1, GF(2))]
```

---

galois.is_irreducible

galois.**is_irreducible**(*poly*)

Checks whether the polynomial $f(x)$ over $\mathrm{GF}(p)$ is irreducible.

A polynomial $f(x) \in \mathrm{GF}(p)[x]$ is *reducible* over $\mathrm{GF}(p)$ if it can be represented as $f(x) = g(x)h(x)$ for some $g(x), h(x) \in \mathrm{GF}(p)[x]$ of strictly lower degree. If $f(x)$ is not reducible, it is said to be *irreducible*. Since Galois fields are not algebraically closed, such irreducible polynomials exist.

This function implements Rabin's irreducibility test. It says a degree-m polynomial $f(x)$ over $\mathrm{GF}(p)$ for prime p is irreducible if and only if $f(x) \mid (x^{p^m} - x)$ and $\gcd(f(x),\ x^{p^{m_i}} - x) = 1$ for $1 \leq i \leq k$, where $m_i = m/p_i$ for the k prime divisors p_i of m.

Parameters **poly** (galois.Poly) – A polynomial $f(x)$ over $\mathrm{GF}(p)$.

Returns `True` if the polynomial is irreducible.

Return type bool

References

- M. O. Rabin. Probabilistic algorithms in finite fields. SIAM Journal on Computing (1980), 273–280. https://apps.dtic.mil/sti/pdfs/ADA078416.pdf
- S. Gao and D. Panarino. Tests and constructions of irreducible polynomials over finite fields. https://www.math.clemson.edu/~sgao/papers/GP97a.pdf
- Section 4.5.1 from https://cacr.uwaterloo.ca/hac/about/chap4.pdf
- https://en.wikipedia.org/wiki/Factorization_of_polynomials_over_finite_fields

---

Examples

```
# Conway polynomials are always irreducible (and primitive)
In [1]: f = galois.conway_poly(2, 5); f
Out[1]: Poly(x^5 + x^2 + 1, GF(2))

# f(x) has no roots in GF(2), a necessary but not sufficient condition of being␣
↪irreducible
In [2]: f.roots()
Out[2]: GF([], order=2)

In [3]: galois.is_irreducible(f)
Out[3]: True
```

```
In [4]: g = galois.conway_poly(2, 4); g
Out[4]: Poly(x^4 + x + 1, GF(2))

In [5]: h = galois.conway_poly(2, 5); h
Out[5]: Poly(x^5 + x^2 + 1, GF(2))

In [6]: f = g * h; f
Out[6]: Poly(x^9 + x^5 + x^4 + x^3 + x^2 + x + 1, GF(2))

# Even though f(x) has no roots in GF(2), it is still reducible
In [7]: f.roots()
Out[7]: GF([], order=2)

In [8]: galois.is_irreducible(f)
Out[8]: False
```

Primitive polynomials

primitive_poly(characteristic, degree[, method])	Returns a monic primitive polynomial $f(x)$ over $\mathrm{GF}(p)$ with degree m.
primitive_polys(characteristic, degree)	Returns all monic primitive polynomials $f(x)$ over $\mathrm{GF}(p)$ with degree m.
conway_poly(characteristic, degree)	Returns the Conway polynomial $C_{p,m}(x)$ over $\mathrm{GF}(p)$ with degree m.
matlab_primitive_poly(characteristic, degree)	Returns Matlab's default primitive polynomial $f(x)$ over $\mathrm{GF}(p)$ with degree m.
is_primitive(poly)	Checks whether the polynomial $f(x)$ over $\mathrm{GF}(p)$ is primitive.

galois.primitive_poly

galois.**primitive_poly**(*characteristic*, *degree*, *method='min'*)

Returns a monic primitive polynomial $f(x)$ over $\mathrm{GF}(p)$ with degree m.

In addition to other applications, $f(x)$ produces the field extension $\mathrm{GF}(p^m)$ of $\mathrm{GF}(p)$. Since $f(x)$ is primitive, x is a primitive element α of $\mathrm{GF}(p^m)$ such that $\mathrm{GF}(p^m) = \{0, 1, \alpha, \alpha^2, \ldots, \alpha^{p^m-2}\}$.

Parameters

- **characteristic** (*int*) – The prime characteristic p of the field $\mathrm{GF}(p)$ that the polynomial is over.
- **degree** (*int*) – The degree m of the desired primitive polynomial.
- **method** (*str, optional*) – The search method for finding the primitive polynomial.
  - "min" (default): Returns the lexicographically-minimal monic primitive polynomial.
  - "max": Returns the lexicographically-maximal monic primitive polynomial.
  - "random": Returns a randomly generated degree-m monic primitive polynomial.

Returns The degree-m monic primitive polynomial over $\mathrm{GF}(p)$.

Return type *galois.Poly*

Examples

Notice *primitive_poly()* returns the lexicographically-minimal primitive polynomial, where *conway_poly()* returns the lexicographically-minimal primitive polynomial that is *consistent* with smaller Conway polynomials.

```
In [1]: galois.primitive_poly(7, 10)
Out[1]: Poly(x^10 + 5x^2 + x + 5, GF(7))

In [2]: galois.conway_poly(7, 10)
Out[2]: Poly(x^10 + x^6 + x^5 + 4x^4 + x^3 + 2x^2 + 3x + 3, GF(7))
```

galois.primitive_polys

galois.**primitive_polys**(*characteristic*, *degree*)

Returns all monic primitive polynomials $f(x)$ over $\mathrm{GF}(p)$ with degree m.

In addition to other applications, $f(x)$ produces the field extension $\mathrm{GF}(p^m)$ of $\mathrm{GF}(p)$. Since $f(x)$ is primitive, x is a primitive element α of $\mathrm{GF}(p^m)$ such that $\mathrm{GF}(p^m) = \{0, 1, \alpha, \alpha^2, \ldots, \alpha^{p^m-2}\}$.

Parameters

- **characteristic** (*int*) – The prime characteristic p of the field $\mathrm{GF}(p)$ that the polynomial is over.
- **degree** (*int*) – The degree m of the desired primitive polynomial.

Returns All degree-m monic primitive polynomials over $\mathrm{GF}(p)$.

Return type list

Examples

```
In [1]: galois.primitive_polys(2, 5)
Out[1]:
[Poly(x^5 + x^2 + 1, GF(2)),
 Poly(x^5 + x^3 + 1, GF(2)),
 Poly(x^5 + x^3 + x^2 + x + 1, GF(2)),
 Poly(x^5 + x^4 + x^2 + x + 1, GF(2)),
 Poly(x^5 + x^4 + x^3 + x + 1, GF(2)),
 Poly(x^5 + x^4 + x^3 + x^2 + 1, GF(2))]
```

galois.conway_poly

galois.**conway_poly**(*characteristic*, *degree*)

Returns the Conway polynomial $C_{p,m}(x)$ over $\mathrm{GF}(p)$ with degree m.

A Conway polynomial is a an irreducible and primitive polynomial over $\mathrm{GF}(p)$ that provides a standard representation of $\mathrm{GF}(p^m)$ as a splitting field of $C_{p,m}(x)$. Conway polynomials provide compatability between fields and their subfields, and hence are the common way to represent extension fields.

The Conway polynomial $C_{p,m}(x)$ is defined as the lexicographically-minimal monic primitive polynomial of degree m over $\mathrm{GF}(p)$ that is compatible with all $C_{p,n}(x)$ for n dividing m.

This function uses Frank Luebeck's Conway polynomial database for fast lookup, not construction.

Parameters

- **characteristic** (*int*) – The prime characteristic p of the field $\mathrm{GF}(p)$.
- **degree** (*int*) – The degree m of the Conway polynomial and degree of the extension field $\mathrm{GF}(p^m)$.

Returns The degree-m Conway polynomial $C_{p,m}(x)$ over $\mathrm{GF}(p)$.

Return type *galois.Poly*

Raises **LookupError** – If the Conway polynomial $C_{p,m}(x)$ is not found in Frank Luebeck's database.

Examples

```
In [1]: galois.conway_poly(2, 100)
Out[1]: Poly(x^100 + x^57 + x^56 + x^55 + x^52 + x^48 + x^47 + x^46 + x^45 + x^44 +␣
↪x^43 + x^41 + x^37 + x^36 + x^35 + x^34 + x^31 + x^30 + x^27 + x^25 + x^24 + x^22␣
↪+ x^20 + x^19 + x^16 + x^15 + x^11 + x^9 + x^8 + x^6 + x^5 + x^3 + 1, GF(2))

In [2]: galois.conway_poly(7, 13)
Out[2]: Poly(x^13 + 6x^2 + 4, GF(7))
```

Notice *primitive_poly()* returns the lexicographically-minimal primitive polynomial, where *conway_poly()* returns the lexicographically-minimal primitive polynomial that is *consistent* with smaller Conway polynomials.

```
In [3]: galois.primitive_poly(7, 10)
Out[3]: Poly(x^10 + 5x^2 + x + 5, GF(7))

In [4]: galois.conway_poly(7, 10)
Out[4]: Poly(x^10 + x^6 + x^5 + 4x^4 + x^3 + 2x^2 + 3x + 3, GF(7))
```

galois.matlab_primitive_poly

galois.**matlab_primitive_poly**(*characteristic*, *degree*)

Returns Matlab's default primitive polynomial $f(x)$ over $\mathrm{GF}(p)$ with degree m.

This function returns the same result as Matlab's `gfprimdf(m, p)`. Matlab uses the lexicographically-minimal primitive polynomial (equivalent to `galois.primitive_poly(p, m)`) as the default... *mostly*. There are three notable exceptions:

1. $\mathrm{GF}(2^7)$ uses $x^7 + x^3 + 1$, not $x^7 + x + 1$.
2. $\mathrm{GF}(2^{14})$ uses $x^{14} + x^{10} + x^6 + x + 1$, not $x^{14} + x^5 + x^3 + x + 1$.
3. $\mathrm{GF}(2^{16})$ uses $x^{16} + x^{12} + x^3 + x + 1$, not $x^{16} + x^5 + x^3 + x^2 + 1$.

Warning: This has been tested for all the $\mathrm{GF}(2^m)$ fields for $2 \leq m \leq 16$ (Matlab doesn't support larger than 16). And it has been spot-checked for $\mathrm{GF}(p^m)$. There may exist other exceptions. Please submit a GitHub issue if you discover one.

Parameters

- **characteristic** (*int*) – The prime characteristic p of the field $\mathrm{GF}(p)$ that the polynomial is over.
- **degree** (*int*) – The degree m of the desired polynomial that produces the field extension $\mathrm{GF}(p^m)$ of $\mathrm{GF}(p)$.

Returns Matlab's default degree-m primitive polynomial over $\mathrm{GF}(p)$.

Return type *galois.Poly*

Examples

```
In [1]: galois.primitive_poly(2, 6)
Out[1]: Poly(x^6 + x + 1, GF(2))

In [2]: galois.matlab_primitive_poly(2, 6)
Out[2]: Poly(x^6 + x + 1, GF(2))
```

```
In [3]: galois.primitive_poly(2, 7)
Out[3]: Poly(x^7 + x + 1, GF(2))

In [4]: galois.matlab_primitive_poly(2, 7)
Out[4]: Poly(x^7 + x^3 + 1, GF(2))
```

galois.is_primitive

galois.**is_primitive**(*poly*)

Checks whether the polynomial $f(x)$ over $\mathrm{GF}(p)$ is primitive.

A degree-m polynomial $f(x)$ over $\mathrm{GF}(p)$ is *primitive* if it is irreducible and $f(x) \mid (x^k - 1)$ for $k = p^m - 1$ and no k less than $p^m - 1$.

Parameters **poly** (galois.Poly) – A degree-m polynomial $f(x)$ over $\mathrm{GF}(p)$.

Returns True if the polynomial is primitive.

Return type bool

References

- Algorithm 4.77 from https://cacr.uwaterloo.ca/hac/about/chap4.pdf

Examples

All Conway polynomials are primitive.

```
In [1]: f = galois.conway_poly(2, 8); f
Out[1]: Poly(x^8 + x^4 + x^3 + x^2 + 1, GF(2))

In [2]: galois.is_primitive(f)
Out[2]: True

In [3]: f = galois.conway_poly(3, 5); f
Out[3]: Poly(x^5 + 2x + 1, GF(3))

In [4]: galois.is_primitive(f)
Out[4]: True
```

The irreducible polynomial of $\mathrm{GF}(2^8)$ for AES is not primitive.

```
In [5]: f = galois.Poly.Degrees([8,4,3,1,0]); f
Out[5]: Poly(x^8 + x^4 + x^3 + x + 1, GF(2))

In [6]: galois.is_primitive(f)
Out[6]: False
```

Primitive elements

primitive_element(irreducible_poly[, start, ...])	Finds the smallest primitive element $g(x)$ of the Galois field $\mathrm{GF}(p^m)$ with degree-m irreducible polynomial $f(x)$ over $\mathrm{GF}(p)$.
primitive_elements(irreducible_poly[, ...])	Finds all primitive elements $g(x)$ of the Galois field $\mathrm{GF}(p^m)$ with degree-m irreducible polynomial $f(x)$ over $\mathrm{GF}(p)$.

continues on next page

Table 14 – continued from previous page

is_primitive_element(element, irreducible_poly)	Determines if $g(x)$ is a primitive element of the Galois field $\mathrm{GF}(p^m)$ with degree-m irreducible polynomial $f(x)$ over $\mathrm{GF}(p)$.

galois.primitive_element

galois.**primitive_element**(*irreducible_poly*, *start=None*, *stop=None*, *reverse=False*)

Finds the smallest primitive element $g(x)$ of the Galois field $\mathrm{GF}(p^m)$ with degree-m irreducible polynomial $f(x)$ over $\mathrm{GF}(p)$.

Parameters

- **irreducible_poly** (*galois.Poly*) – The degree-m irreducible polynomial $f(x)$ over $\mathrm{GF}(p)$ that defines the extension field $\mathrm{GF}(p^m)$.
- **start** (*int, optional*) – Starting value (inclusive, integer representation of the polynomial) in the search for a primitive element $g(x)$ of $\mathrm{GF}(p^m)$. The default is `None` which represents p, which corresponds to $g(x) = x$ over $\mathrm{GF}(p)$.
- **stop** (*int, optional*) – Stopping value (exclusive, integer representation of the polynomial) in the search for a primitive element $g(x)$ of $\mathrm{GF}(p^m)$. The default is `None` which represents p^m, which corresponds to $g(x) = x^m$ over $\mathrm{GF}(p)$.
- **reverse** (*bool, optional*) – Search for a primitive element in reverse order, i.e. find the largest primitive element first. Default is `False`.

Returns A primitive element of $\mathrm{GF}(p^m)$ with irreducible polynomial $f(x)$. The primitive element $g(x)$ is a polynomial over $\mathrm{GF}(p)$ with degree less than m.

Return type *galois.Poly*

Examples

```
In [1]: GF = galois.GF(3)

In [2]: f = galois.Poly([1,1,2], field=GF); f
Out[2]: Poly(x^2 + x + 2, GF(3))

In [3]: galois.is_irreducible(f)
Out[3]: True

In [4]: galois.is_primitive(f)
Out[4]: True

In [5]: galois.primitive_element(f)
Out[5]: Poly(x, GF(3))
```

```
In [6]: GF = galois.GF(3)

In [7]: f = galois.Poly([1,0,1], field=GF); f
Out[7]: Poly(x^2 + 1, GF(3))

In [8]: galois.is_irreducible(f)
Out[8]: True
```

(continues on next page)

(continued from previous page)

```
In [9]: galois.is_primitive(f)
Out[9]: False

In [10]: galois.primitive_element(f)
Out[10]: Poly(x + 1, GF(3))
```

galois.primitive_elements

galois.**primitive_elements**(*irreducible_poly*, *start=None*, *stop=None*, *reverse=False*)

Finds all primitive elements $g(x)$ of the Galois field $\mathrm{GF}(p^m)$ with degree-m irreducible polynomial $f(x)$ over $\mathrm{GF}(p)$.

The number of primitive elements of $\mathrm{GF}(p^m)$ is $\phi(p^m - 1)$, where $\phi(n)$ is the Euler totient function. See :obj:galois.euler_phi`.

Parameters

- **irreducible_poly** (galois.Poly) – The degree-m irreducible polynomial $f(x)$ over $\mathrm{GF}(p)$ that defines the extension field $\mathrm{GF}(p^m)$.
- **start** (*int, optional*) – Starting value (inclusive, integer representation of the polynomial) in the search for primitive elements $g(x)$ of $\mathrm{GF}(p^m)$. The default is `None` which represents p, which corresponds to $g(x) = x$ over $\mathrm{GF}(p)$.
- **stop** (*int, optional*) – Stopping value (exclusive, integer representation of the polynomial) in the search for primitive elements $g(x)$ of $\mathrm{GF}(p^m)$. The default is `None` which represents p^m, which corresponds to $g(x) = x^m$ over $\mathrm{GF}(p)$.
- **reverse** (*bool, optional*) – Search for primitive elements in reverse order, i.e. largest to smallest. Default is `False`.

Returns List of all primitive elements of $\mathrm{GF}(p^m)$ with irreducible polynomial $f(x)$. Each primitive element $g(x)$ is a polynomial over $\mathrm{GF}(p)$ with degree less than m.

Return type list

Examples

```
In [1]: GF = galois.GF(3)

In [2]: f = galois.Poly([1,1,2], field=GF); f
Out[2]: Poly(x^2 + x + 2, GF(3))

In [3]: galois.is_irreducible(f)
Out[3]: True

In [4]: galois.is_primitive(f)
Out[4]: True

In [5]: g = galois.primitive_elements(f); g
Out[5]: [Poly(x, GF(3)), Poly(x + 1, GF(3)), Poly(2x, GF(3)), Poly(2x + 2, GF(3))]
```

(continues on next page)

(continued from previous page)

```
In [6]: len(g) == galois.euler_phi(3**2 - 1)
Out[6]: True
```

```
In [7]: GF = galois.GF(3)

In [8]: f = galois.Poly([1,0,1], field=GF); f
Out[8]: Poly(x^2 + 1, GF(3))

In [9]: galois.is_irreducible(f)
Out[9]: True

In [10]: galois.is_primitive(f)
Out[10]: False

In [11]: g = galois.primitive_elements(f); g
Out[11]:
[Poly(x + 1, GF(3)),
 Poly(x + 2, GF(3)),
 Poly(2x + 1, GF(3)),
 Poly(2x + 2, GF(3))]

In [12]: len(g) == galois.euler_phi(3**2 - 1)
Out[12]: True
```

galois.is_primitive_element

galois.**is_primitive_element**(*element*, *irreducible_poly*)

Determines if $g(x)$ is a primitive element of the Galois field $\mathrm{GF}(p^m)$ with degree-m irreducible polynomial $f(x)$ over $\mathrm{GF}(p)$.

The number of primitive elements of $\mathrm{GF}(p^m)$ is $\phi(p^m-1)$, where $\phi(n)$ is the Euler totient function, see *galois.euler_phi*.

Parameters

- **element** (galois.Poly) – An element $g(x)$ of $\mathrm{GF}(p^m)$ as a polynomial over $\mathrm{GF}(p)$ with degree less than m.
- **irreducible_poly** (galois.Poly) – The degree-m irreducible polynomial $f(x)$ over $\mathrm{GF}(p)$ that defines the extension field $\mathrm{GF}(p^m)$.

Returns `True` if $g(x)$ is a primitive element of $\mathrm{GF}(p^m)$ with irreducible polynomial $f(x)$.

Return type bool

Examples

```
In [1]: GF = galois.GF(3)

In [2]: f = galois.Poly([1,1,2], field=GF); f
Out[2]: Poly(x^2 + x + 2, GF(3))
```

(continues on next page)

(continued from previous page)

```
In [3]: galois.is_irreducible(f)
Out[3]: True

In [4]: galois.is_primitive(f)
Out[4]: True

In [5]: g = galois.Poly.Identity(GF); g
Out[5]: Poly(x, GF(3))

In [6]: galois.is_primitive_element(g, f)
Out[6]: True
```

```
In [7]: GF = galois.GF(3)

In [8]: f = galois.Poly([1,0,1], field=GF); f
Out[8]: Poly(x^2 + 1, GF(3))

In [9]: galois.is_irreducible(f)
Out[9]: True

In [10]: galois.is_primitive(f)
Out[10]: False

In [11]: g = galois.Poly.Identity(GF); g
Out[11]: Poly(x, GF(3))

In [12]: galois.is_primitive_element(g, f)
Out[12]: False
```

Minimal polynomials

minimal_poly(element)	Computes the minimal polynomial $m_e(x) \in \mathrm{GF}(p)[x]$ of a Galois field element $e \in \mathrm{GF}(p^m)$.

galois.minimal_poly

galois.**minimal_poly**(*element*)

Computes the minimal polynomial $m_e(x) \in \mathrm{GF}(p)[x]$ of a Galois field element $e \in \mathrm{GF}(p^m)$.

The *minimal polynomial* of a Galois field element $e \in \mathrm{GF}(p^m)$ is the polynomial of minimal degree over $\mathrm{GF}(p)$ for which e is a root when evaluated in $\mathrm{GF}(p^m)$. Namely, $m_e(x) \in \mathrm{GF}(p)[x] \in \mathrm{GF}(p^m)[x]$ and $m_e(e) = 0$ over $\mathrm{GF}(p^m)$.

Parameters **element** (galois.FieldArray) – Any element e of the Galois field $\mathrm{GF}(p^m)$. This must be a 0-dim array.

Returns The minimal polynomial $m_e(x)$ over $\mathrm{GF}(p)$ of the element e.

Return type *galois.Poly*

Examples

```
In [1]: GF = galois.GF(2**4)

In [2]: e = GF.primitive_element; e
Out[2]: GF(2, order=2^4)

In [3]: m_e = galois.minimal_poly(e); m_e
Out[3]: Poly(x^4 + x + 1, GF(2))

# Evaluate m_e(e) in GF(2^4)
In [4]: m_e(e, field=GF)
Out[4]: GF(0, order=2^4)
```

For a given element e, the minimal polynomials of e and all its conjugates are the same.

```
# The conjugates of e
In [5]: conjugates = np.unique(e**(2**np.arange(0, 4))); conjugates
Out[5]: GF([2, 3, 4, 5], order=2^4)

In [6]: for conjugate in conjugates:
   ...:     print(galois.minimal_poly(conjugate))
   ...:
Poly(x^4 + x + 1, GF(2))
Poly(x^4 + x + 1, GF(2))
Poly(x^4 + x + 1, GF(2))
Poly(x^4 + x + 1, GF(2))
```

Not all elements of $\mathrm{GF}(2^4)$ have minimal polynomials with degree-4.

```
In [7]: e = GF.primitive_element**5; e
Out[7]: GF(6, order=2^4)

# The conjugates of e
In [8]: conjugates = np.unique(e**(2**np.arange(0, 4))); conjugates
Out[8]: GF([6, 7], order=2^4)

In [9]: for conjugate in conjugates:
   ...:     print(galois.minimal_poly(conjugate))
   ...:
Poly(x^2 + x + 1, GF(2))
Poly(x^2 + x + 1, GF(2))
```

In prime fields, the minimal polynomial of e is simply $m_e(x) = x - e$.

```
In [10]: GF = galois.GF(7)

In [11]: e = GF(3); e
Out[11]: GF(3, order=7)

In [12]: m_e = galois.minimal_poly(e); m_e
Out[12]: Poly(x + 4, GF(7))
```

(continues on next page)

(continued from previous page)

```
In [13]: m_e(e)
Out[13]: GF(0, order=7)
```

5.2 Polynomials over Galois Fields

This section contains classes and functions for creating polynomials over Galois fields.

5.2.1 Polynomial classes

Poly(coeffs[, field, order])	Create a polynomial $f(x)$ over $\mathrm{GF}(p^m)$.

galois.Poly

class galois.**Poly**(*coeffs*, *field=None*, *order='desc'*)

Create a polynomial $f(x)$ over $\mathrm{GF}(p^m)$.

The polynomial $f(x) = a_d x^d + a_{d-1} x^{d-1} + \cdots + a_1 x + a_0$ has coefficients $\{a_d, a_{d-1}, \ldots, a_1, a_0\}$ in $\mathrm{GF}(p^m)$.

Parameters

- **coeffs** (*array_like*) – The polynomial coefficients $\{a_d, a_{d-1}, \ldots, a_1, a_0\}$ with type *galois.FieldArray*, numpy.ndarray, list, or tuple. The first element is the highest-degree element if `order="desc"` or the first element is the 0-th degree element if `order="asc"`.
- **field** (galois.FieldClass, *optional*) – The field $\mathrm{GF}(p^m)$ the polynomial is over. The default is `None` which represents *galois.GF2*. If `coeffs` is a Galois field array, then that field is used and the `field` argument is ignored.
- **order** (*str*, *optional*) – The interpretation of the coefficient degrees, either `"desc"` (default) or `"asc"`. For `"desc"`, the first element of `coeffs` is the highest degree coefficient x^d) and the last element is the 0-th degree element x^0.

Returns The polynomial $f(x)$.

Return type *galois.Poly*

Examples

Create a polynomial over $\mathrm{GF}(2)$.

```
In [1]: galois.Poly([1,0,1,1])
Out[1]: Poly(x^3 + x + 1, GF(2))

In [2]: galois.Poly.Degrees([3,1,0])
Out[2]: Poly(x^3 + x + 1, GF(2))
```

Create a polynomial over $\mathrm{GF}(2^8)$.

```
In [3]: GF = galois.GF(2**8)

In [4]: galois.Poly([124,0,223,0,0,15], field=GF)
Out[4]: Poly(124x^5 + 223x^3 + 15, GF(2^8))

# Alternate way of constructing the same polynomial
In [5]: galois.Poly.Degrees([5,3,0], coeffs=[124,223,15], field=GF)
Out[5]: Poly(124x^5 + 223x^3 + 15, GF(2^8))
```

Polynomial arithmetic using binary operators.

```
In [6]: a = galois.Poly([117,0,63,37], field=GF); a
Out[6]: Poly(117x^3 + 63x + 37, GF(2^8))

In [7]: b = galois.Poly([224,0,21], field=GF); b
Out[7]: Poly(224x^2 + 21, GF(2^8))

In [8]: a + b
Out[8]: Poly(117x^3 + 224x^2 + 63x + 48, GF(2^8))

In [9]: a - b
Out[9]: Poly(117x^3 + 224x^2 + 63x + 48, GF(2^8))

# Compute the quotient of the polynomial division
In [10]: a / b
Out[10]: Poly(202x, GF(2^8))

# True division and floor division are equivalent
In [11]: a / b == a // b
Out[11]: True

# Compute the remainder of the polynomial division
In [12]: a % b
Out[12]: Poly(198x + 37, GF(2^8))

# Compute both the quotient and remainder in one pass
In [13]: divmod(a, b)
Out[13]: (Poly(202x, GF(2^8)), Poly(198x + 37, GF(2^8)))
```

Constructors

Degrees(degrees[, coeffs, field])	Constructs a polynomial over $\mathrm{GF}(p^m)$ from its non-zero degrees.
Identity([field])	Constructs the identity polynomial $f(x) = x$ over $\mathrm{GF}(p^m)$.
Integer(integer[, field])	Constructs a polynomial over $\mathrm{GF}(p^m)$ from its integer representation.
One([field])	Constructs the one polynomial $f(x) = 1$ over $\mathrm{GF}(p^m)$.

continues on next page

Table 17 – continued from previous page

Random(degree[, field])	Constructs a random polynomial over $\mathrm{GF}(p^m)$ with degree d.
Roots(roots[, multiplicities, field])	Constructs a monic polynomial in $\mathrm{GF}(p^m)[x]$ from its roots.
String(string[, field])	Constructs a polynomial over $\mathrm{GF}(p^m)$ from its string representation.
Zero([field])	Constructs the zero polynomial $f(x) = 0$ over $\mathrm{GF}(p^m)$.

Methods

derivative([k])	Computes the k-th formal derivative $\frac{d^k}{dx^k} f(x)$ of the polynomial $f(x)$.
roots([multiplicity])	Calculates the roots r of the polynomial $f(x)$, such that $f(r) = 0$.

Attributes

coeffs	The coefficients of the polynomial in degree-descending order.
degree	The degree of the polynomial, i.e. the highest degree with non-zero coefficient.
degrees	An array of the polynomial degrees in degree-descending order.
field	The Galois field array class to which the coefficients belong.
integer	The integer representation of the polynomial.
nonzero_coeffs	The non-zero coefficients of the polynomial in degree-descending order.
nonzero_degrees	An array of the polynomial degrees that have non-zero coefficients, in degree-descending order.
string	The string representation of the polynomial, without specifying the Galois field.

classmethod Degrees(*degrees*, *coeffs=None*, *field=None*)

Constructs a polynomial over $\mathrm{GF}(p^m)$ from its non-zero degrees.

Parameters

- **degrees** (*list*) – List of polynomial degrees with non-zero coefficients.
- **coeffs** (*array_like, optional*) – List of corresponding non-zero coefficients. The default is `None` which corresponds to all one coefficients, i.e. `[1,]*len(degrees)`.
- **field** (*galois.FieldClass, optional*) – The field $\mathrm{GF}(p^m)$ the polynomial is over. The default is`None` which represents *galois.GF2*.

Returns The polynomial $f(x)$.

Return type *galois.Poly*

Examples

Construct a polynomial over $\mathrm{GF}(2)$ by specifying the degrees with non-zero coefficients.

```
In [1]: galois.Poly.Degrees([3,1,0])
Out[1]: Poly(x^3 + x + 1, GF(2))
```

Construct a polynomial over $\mathrm{GF}(2^8)$ by specifying the degrees with non-zero coefficients.

```
In [2]: GF = galois.GF(2**8)

In [3]: galois.Poly.Degrees([3,1,0], coeffs=[251,73,185], field=GF)
Out[3]: Poly(251x^3 + 73x + 185, GF(2^8))
```

classmethod Identity(*field=<class 'numpy.ndarray over GF(2)'>*)

Constructs the identity polynomial $f(x) = x$ over $\mathrm{GF}(p^m)$.

Parameters **field** (*galois.FieldClass, optional*) – The field $\mathrm{GF}(p^m)$ the polynomial is over. The default is *galois.GF2*.

Returns The polynomial $f(x)$.

Return type *galois.Poly*

Examples

Construct the identity polynomial over $\mathrm{GF}(2)$.

```
In [1]: galois.Poly.Identity()
Out[1]: Poly(x, GF(2))
```

Construct the identity polynomial over $\mathrm{GF}(2^8)$.

```
In [2]: GF = galois.GF(2**8)

In [3]: galois.Poly.Identity(field=GF)
Out[3]: Poly(x, GF(2^8))
```

classmethod Integer(*integer*, *field=<class 'numpy.ndarray over GF(2)'>*)

Constructs a polynomial over $\mathrm{GF}(p^m)$ from its integer representation.

The integer value i represents the polynomial $f(x) = a_d x^d + a_{d-1} x^{d-1} + \cdots + a_1 x + a_0$ over field $\mathrm{GF}(p^m)$ if $i = a_d (p^m)^d + a_{d-1} (p^m)^{d-1} + \cdots + a_1 (p^m) + a_0$ using integer arithmetic, not finite field arithmetic.

Parameters

- **integer** (*int*) – The integer representation of the polynomial $f(x)$.
- **field** (*galois.FieldClass, optional*) – The field $\mathrm{GF}(p^m)$ the polynomial is over. The default is *galois.GF2*.

Returns The polynomial $f(x)$.

Return type *galois.Poly*

Examples

Construct a polynomial over $\mathrm{GF}(2)$ from its integer representation.

```
In [1]: galois.Poly.Integer(5)
Out[1]: Poly(x^2 + 1, GF(2))
```

Construct a polynomial over $\mathrm{GF}(2^8)$ from its integer representation.

```
In [2]: GF = galois.GF(2**8)

In [3]: galois.Poly.Integer(13*256**3 + 117, field=GF)
Out[3]: Poly(13x^3 + 117, GF(2^8))
```

---

classmethod One(*field=<class 'numpy.ndarray over GF(2)'>*)

Constructs the one polynomial $f(x) = 1$ over $\mathrm{GF}(p^m)$.

Parameters **field** (*galois.FieldClass, optional*) – The field $\mathrm{GF}(p^m)$ the polynomial is over. The default is *galois.GF2*.

Returns The polynomial $f(x)$.

Return type *galois.Poly*

Examples

Construct the one polynomial over $\mathrm{GF}(2)$.

```
In [1]: galois.Poly.One()
Out[1]: Poly(1, GF(2))
```

Construct the one polynomial over $\mathrm{GF}(2^8)$.

```
In [2]: GF = galois.GF(2**8)

In [3]: galois.Poly.One(field=GF)
Out[3]: Poly(1, GF(2^8))
```

---

classmethod Random(*degree*, *field=<class 'numpy.ndarray over GF(2)'>*)

Constructs a random polynomial over $\mathrm{GF}(p^m)$ with degree d.

Parameters

- **degree** (*int*) – The degree of the polynomial.
- **field** (*galois.FieldClass, optional*) – The field $\mathrm{GF}(p^m)$ the polynomial is over. The default is *galois.GF2*.

Returns The polynomial $f(x)$.

Return type *galois.Poly*

Examples

Construct a random degree-5 polynomial over $\mathrm{GF}(2)$.

```
In [1]: galois.Poly.Random(5)
Out[1]: Poly(x^5 + x^3, GF(2))
```

Construct a random degree-5 polynomial over $\mathrm{GF}(2^8)$.

```
In [2]: GF = galois.GF(2**8)

In [3]: galois.Poly.Random(5, field=GF)
Out[3]: Poly(56x^5 + 207x^4 + 92x^3 + x^2 + 168x + 110, GF(2^8))
```

---

classmethod Roots(*roots*, *multiplicities=None*, *field=None*)

Constructs a monic polynomial in $\mathrm{GF}(p^m)[x]$ from its roots.

The polynomial $f(x)$ with d roots $\{r_0, r_1, \ldots, r_{d-1}\}$ is:

$$f(x) = (x - r_0)(x - r_1)\ldots(x - r_{d-1})$$
$$f(x) = a_d x^d + a_{d-1} x^{d-1} + \cdots + a_1 x + a_0$$

Parameters

- **roots** (*array_like*) – List of roots in $\mathrm{GF}(p^m)$ of the desired polynomial.
- **multiplicities** (*array_like, optional*) – List of multiplicity of each root. The default is `None` which corresponds to all ones.
- **field** (*galois.FieldClass, optional*) – The field $\mathrm{GF}(p^m)$ the polynomial is over. The default is`None` which represents *galois.GF2*.

Returns The polynomial $f(x)$.

Return type *galois.Poly*

Examples

Construct a polynomial over $\mathrm{GF}(2)$ from a list of its roots.

```
In [1]: roots = [0, 0, 1]

In [2]: p = galois.Poly.Roots(roots); p
Out[2]: Poly(x^3 + x^2, GF(2))

In [3]: p(roots)
Out[3]: GF([0, 0, 0], order=2)
```

Construct a polynomial over $\mathrm{GF}(2^8)$ from a list of its roots.

```
In [4]: GF = galois.GF(2**8)

In [5]: roots = [121, 198, 225]

In [6]: p = galois.Poly.Roots(roots, field=GF); p
Out[6]: Poly(x^3 + 94x^2 + 174x + 89, GF(2^8))

In [7]: p(roots)
Out[7]: GF([0, 0, 0], order=2^8)
```

---

classmethod String(*string*, *field=<class 'numpy.ndarray over GF(2)'>*)

Constructs a polynomial over $\mathrm{GF}(p^m)$ from its string representation.

Parameters

- **string** (*str*) – The string representation of the polynomial $f(x)$.
- **field** (*galois.FieldClass, optional*) – The field $\mathrm{GF}(p^m)$ the polynomial is over. The default is *galois.GF2*.

Returns The polynomial $f(x)$.

Return type *galois.Poly*

Examples

Construct a polynomial over $\mathrm{GF}(2)$ from its string representation.

```
In [1]: galois.Poly.String("x^2 + 1")
Out[1]: Poly(x^2 + 1, GF(2))
```

Construct a polynomial over $\mathrm{GF}(2^8)$ from its string representation.

```
In [2]: GF = galois.GF(2**8)

In [3]: galois.Poly.String("13x^3 + 117", field=GF)
Out[3]: Poly(13x^3 + 117, GF(2^8))
```

---

classmethod Zero(*field=<class 'numpy.ndarray over GF(2)'>*)

Constructs the zero polynomial $f(x) = 0$ over $\mathrm{GF}(p^m)$.

Parameters **field** (*galois.FieldClass, optional*) – The field $\mathrm{GF}(p^m)$ the polynomial is over. The default is *galois.GF2*.

Returns The polynomial $f(x)$.

Return type *galois.Poly*

Examples

Construct the zero polynomial over $\mathrm{GF}(2)$.

```
In [1]: galois.Poly.Zero()
Out[1]: Poly(0, GF(2))
```

Construct the zero polynomial over $\mathrm{GF}(2^8)$.

```
In [2]: GF = galois.GF(2**8)

In [3]: galois.Poly.Zero(field=GF)
Out[3]: Poly(0, GF(2^8))
```

---

derivative(*k=1*)

Computes the k-th formal derivative $\frac{d^k}{dx^k} f(x)$ of the polynomial $f(x)$.

For the polynomial

$$f(x) = a_d x^d + a_{d-1} x^{d-1} + \cdots + a_1 x + a_0$$

the first formal derivative is defined as

$$p'(x) = (d) \cdot a_d x^{d-1} + (d-1) \cdot a_{d-1} x^{d-2} + \cdots + (2) \cdot a_2 x + a_1$$

where $\cdot$ represents scalar multiplication (repeated addition), not finite field multiplication, e.g. $3 \cdot a = a + a + a$.

Parameters **k** (*int, optional*) – The number of derivatives to compute. 1 corresponds to $p'(x)$, 2 corresponds to $p''(x)$, etc. The default is 1.

Returns The k-th formal derivative of the polynomial $f(x)$.

Return type *galois.Poly*

References

- https://en.wikipedia.org/wiki/Formal_derivative

Examples

Compute the derivatives of a polynomial over $\mathrm{GF}(2)$.

```
In [1]: p = galois.Poly.Random(7); p
Out[1]: Poly(x^7 + x^3 + x^2 + x, GF(2))

In [2]: p.derivative()
Out[2]: Poly(x^6 + x^2 + 1, GF(2))

# k derivatives of a polynomial where k is the Galois field's characteristic␣
↪will always result in 0
In [3]: p.derivative(2)
Out[3]: Poly(0, GF(2))
```

Compute the derivatives of a polynomial over $\mathrm{GF}(7)$.

```
In [4]: GF = galois.GF(7)

In [5]: p = galois.Poly.Random(11, field=GF); p
Out[5]: Poly(x^11 + 2x^10 + 2x^9 + 5x^7 + x^5 + 5x^4 + 4x^3 + x^2 + 4x + 2,␣
↪GF(7))

In [6]: p.derivative()
Out[6]: Poly(4x^10 + 6x^9 + 4x^8 + 5x^4 + 6x^3 + 5x^2 + 2x + 4, GF(7))

In [7]: p.derivative(2)
Out[7]: Poly(5x^9 + 5x^8 + 4x^7 + 6x^3 + 4x^2 + 3x + 2, GF(7))

In [8]: p.derivative(3)
Out[8]: Poly(3x^8 + 5x^7 + 4x^2 + x + 3, GF(7))

# k derivatives of a polynomial where k is the Galois field's characteristic␣
↪will always result in 0
In [9]: p.derivative(7)
Out[9]: Poly(0, GF(2))
```

Compute the derivatives of a polynomial over $\mathrm{GF}(2^8)$.

```
In [10]: GF = galois.GF(2**8)

In [11]: p = galois.Poly.Random(7, field=GF); p
Out[11]: Poly(15x^7 + 52x^6 + 31x^5 + 21x^4 + 55x^3 + 240x^2 + 141x + 125, GF(2^
         ↪8))

In [12]: p.derivative()
Out[12]: Poly(15x^6 + 31x^4 + 55x^2 + 141, GF(2^8))

# k derivatives of a polynomial where k is the Galois field's characteristic␣
         ↪will always result in 0
In [13]: p.derivative(2)
Out[13]: Poly(0, GF(2^8))
```

---

roots(*multiplicity=False*)

Calculates the roots r of the polynomial $f(x)$, such that $f(r) = 0$.

This implementation uses Chien's search to find the roots $\{r_0, r_1, \ldots, r_{k-1}\}$ of the degree-d polynomial

$$f(x) = a_d x^d + a_{d-1} x^{d-1} + \cdots + a_1 x + a_0,$$

where $k \leq d$. Then, $f(x)$ can be factored as

$$f(x) = (x - r_0)^{m_0} (x - r_1)^{m_1} \ldots (x - r_{k-1})^{m_{k-1}},$$

where m_i is the multiplicity of root r_i and

$$\sum_{i=0}^{k-1} m_i = d.$$

The Galois field elements can be represented as $\mathrm{GF}(p^m) = \{0, 1, \alpha, \alpha^2, \ldots, \alpha^{p^m - 2}\}$, where α is a primitive element of $\mathrm{GF}(p^m)$.

0 is a root of $f(x)$ if:

$$a_0 = 0$$

1 is a root of $f(x)$ if:

$$\sum_{j=0}^{d} a_j = 0$$

The remaining elements of $\mathrm{GF}(p^m)$ are powers of α. The following equations calculate $p(\alpha^i)$, where α^i is a root of $f(x)$ if $p(\alpha^i) = 0$.

$$\begin{aligned}
p(\alpha^i) &= a_d(\alpha^i)^d + a_{d-1}(\alpha^i)^{d-1} + \cdots + a_1(\alpha^i) + a_0 \\
p(\alpha^i) &\triangleq \lambda_{i,d} + \lambda_{i,d-1} + \cdots + \lambda_{i,1} + \lambda_{i,0} \\
p(\alpha^i) &= \sum_{j=0}^{d} \lambda_{i,j}
\end{aligned}$$

The next power of α can be easily calculated from the previous calculation.

$$
\begin{aligned}
p(\alpha^{i+1}) &= a_d(\alpha^{i+1})^d + a_{d-1}(\alpha^{i+1})^{d-1} + \cdots + a_1(\alpha^{i+1}) + a_0 \\
p(\alpha^{i+1}) &= a_d(\alpha^{i})^d\alpha^d + a_{d-1}(\alpha^{i})^{d-1}\alpha^{d-1} + \cdots + a_1(\alpha^{i})\alpha + a_0 \\
p(\alpha^{i+1}) &= \lambda_{i,d}\alpha^d + \lambda_{i,d-1}\alpha^{d-1} + \cdots + \lambda_{i,1}\alpha + \lambda_{i,0} \\
p(\alpha^{i+1}) &= \sum_{j=0}^{d} \lambda_{i,j}\alpha^j
\end{aligned}
$$

Parameters **multiplicity** (*bool, optional*) – Optionally return the multiplicity of each root. The default is `False`, which only returns the unique roots.

Returns

- *galois.FieldArray* – Galois field array of roots of $f(x)$.
- *np.ndarray* – The multiplicity of each root. Only returned if `multiplicity=True`.

References

- https://en.wikipedia.org/wiki/Chien_search

Examples

Find the roots of a polynomial over $\mathrm{GF}(2)$.

```
In [1]: p = galois.Poly.Roots([0,]*7 + [1,]*13); p
Out[1]: Poly(x^20 + x^19 + x^16 + x^15 + x^12 + x^11 + x^8 + x^7, GF(2))

In [2]: p.roots()
Out[2]: GF([0, 1], order=2)

In [3]: p.roots(multiplicity=True)
Out[3]: (GF([0, 1], order=2), array([ 7, 13]))
```

Find the roots of a polynomial over $\mathrm{GF}(2^8)$.

```
In [4]: GF = galois.GF(2**8)

In [5]: p = galois.Poly.Roots([18,]*7 + [155,]*13 + [227,]*9, field=GF); p
Out[5]: Poly(x^29 + 106x^28 + 27x^27 + 155x^26 + 230x^25 + 38x^24 + 78x^23 + 8x^
↪22 + 46x^21 + 210x^20 + 248x^19 + 214x^18 + 172x^17 + 152x^16 + 82x^15 + 237x^
↪14 + 172x^13 + 230x^12 + 141x^11 + 63x^10 + 103x^9 + 167x^8 + 199x^7 + 127x^6␣
↪+ 254x^5 + 95x^4 + 93x^3 + 3x^2 + 4x + 208, GF(2^8))

In [6]: p.roots()
Out[6]: GF([ 18, 155, 227], order=2^8)

In [7]: p.roots(multiplicity=True)
Out[7]: (GF([ 18, 155, 227], order=2^8), array([ 7, 13,  9]))
```

property coeffs

The coefficients of the polynomial in degree-descending order. The entries of *galois.Poly.degrees* are paired with *galois.Poly.coeffs*.

Examples

```
In [1]: GF = galois.GF(7)

In [2]: p = galois.Poly([3, 0, 5, 2], field=GF)

In [3]: p.coeffs
Out[3]: GF([3, 0, 5, 2], order=7)
```

Type *galois.FieldArray*

property degree

The degree of the polynomial, i.e. the highest degree with non-zero coefficient.

Examples

```
In [1]: GF = galois.GF(7)

In [2]: p = galois.Poly([3, 0, 5, 2], field=GF)

In [3]: p.degree
Out[3]: 3
```

Type int

property degrees

An array of the polynomial degrees in degree-descending order. The entries of *galois.Poly.degrees* are paired with *galois.Poly.coeffs*.

Examples

```
In [1]: GF = galois.GF(7)

In [2]: p = galois.Poly([3, 0, 5, 2], field=GF)

In [3]: p.degrees
Out[3]: array([3, 2, 1, 0])
```

Type numpy.ndarray

property field

The Galois field array class to which the coefficients belong.

Examples

```
In [1]: a = galois.Poly.Random(5); a
Out[1]: Poly(x^5 + x^4 + x^3, GF(2))
```

(continues on next page)

(continued from previous page)

```
In [2]: a.field
Out[2]: <class 'numpy.ndarray over GF(2)'>

In [3]: b = galois.Poly.Random(5, field=galois.GF(2**8)); b
Out[3]: Poly(157x^5 + 215x^4 + 118x^3 + 180x^2 + 221x + 20, GF(2^8))

In [4]: b.field
Out[4]: <class 'numpy.ndarray over GF(2^8)'>
```

Type *galois.FieldClass*

property integer

The integer representation of the polynomial. For polynomial $f(x) = a_d x^d + a_{d-1} x^{d-1} + \cdots + a_1 x + a_0$ with elements in $a_k \in \mathrm{GF}(p^m)$, the integer representation is $i = a_d(p^m)^d + a_{d-1}(p^m)^{d-1} + \cdots + a_1(p^m) + a_0$ (using integer arithmetic, not finite field arithmetic).

Examples

```
In [1]: GF = galois.GF(7)

In [2]: p = galois.Poly([3, 0, 5, 2], field=GF)

In [3]: p.integer
Out[3]: 1066

In [4]: p.integer == 3*7**3 + 5*7**1 + 2*7**0
Out[4]: True
```

Type int

property nonzero_coeffs

The non-zero coefficients of the polynomial in degree-descending order. The entries of *galois.Poly.nonzero_degrees* are paired with *galois.Poly.nonzero_coeffs*.

Examples

```
In [1]: GF = galois.GF(7)

In [2]: p = galois.Poly([3, 0, 5, 2], field=GF)

In [3]: p.nonzero_coeffs
Out[3]: GF([3, 5, 2], order=7)
```

Type *galois.FieldArray*

property nonzero_degrees

An array of the polynomial degrees that have non-zero coefficients, in degree-descending order. The entries of *galois.Poly.nonzero_degrees* are paired with *galois.Poly.nonzero_coeffs*.

Examples

```
In [1]: GF = galois.GF(7)

In [2]: p = galois.Poly([3, 0, 5, 2], field=GF)

In [3]: p.nonzero_degrees
Out[3]: array([3, 1, 0])
```

Type numpy.ndarray

property string

The string representation of the polynomial, without specifying the Galois field.

Examples

```
In [1]: GF = galois.GF(7)

In [2]: p = galois.Poly([3, 0, 5, 2], field=GF); p
Out[2]: Poly(3x^3 + 5x + 2, GF(7))

In [3]: p.string
Out[3]: '3x^3 + 5x + 2'
```

Type str

5.2.2 Polynomial functions

poly_gcd(a, b)	Finds the greatest common divisor of two polynomials $a(x)$ and $b(x)$ over GF(q).
poly_egcd(a, b)	Finds the polynomial multiplicands of $a(x)$ and $b(x)$ such that $a(x)s(x) + b(x)t(x) = \gcd(a(x), b(x))$.
poly_pow(poly, power, modulus)	Efficiently exponentiates a polynomial $f(x)$ to the power k reducing by modulo $g(x)$, $f(x)^k$ mod $g(x)$.
poly_factors(poly)	Factors the polynomial $f(x)$ into a product of n irreducible factors $f(x) = g_0(x)^{k_0} g_1(x)^{k_1} \dots g_{n-1}(x)^{k_{n-1}}$ with $k_0 \leq k_1 \leq \dots \leq k_{n-1}$.

galois.poly_gcd

galois.**poly_gcd**(a, b)

Finds the greatest common divisor of two polynomials $a(x)$ and $b(x)$ over $\mathrm{GF}(q)$.

This implementation uses the Euclidean Algorithm.

Parameters

- **a** (galois.Poly) – A polynomial $a(x)$ over $\mathrm{GF}(q)$.
- **b** (galois.Poly) – A polynomial $b(x)$ over $\mathrm{GF}(q)$.

Returns Polynomial greatest common divisor of $a(x)$ and $b(x)$.

Return type *galois.Poly*

Examples

```
In [1]: GF = galois.GF(7)

In [2]: a = galois.Poly.Roots([2,2,2,3,6], field=GF); a
Out[2]: Poly(x^5 + 6x^4 + x + 3, GF(7))

# a(x) and b(x) only share the root 2 in common
In [3]: b = galois.Poly.Roots([1,2], field=GF); b
Out[3]: Poly(x^2 + 4x + 2, GF(7))

In [4]: gcd = galois.poly_gcd(a, b); gcd
Out[4]: Poly(x + 5, GF(7))

# The GCD has only 2 as a root with multiplicity 1
In [5]: gcd.roots(multiplicity=True)
Out[5]: (GF([2], order=7), array([1]))
```

galois.poly_egcd

galois.**poly_egcd**(a, b)

Finds the polynomial multiplicands of $a(x)$ and $b(x)$ such that $a(x)s(x) + b(x)t(x) = \gcd(a(x), b(x))$.

This implementation uses the Extended Euclidean Algorithm.

Parameters

- **a** (galois.Poly) – A polynomial $a(x)$ over $\mathrm{GF}(q)$.
- **b** (galois.Poly) – A polynomial $b(x)$ over $\mathrm{GF}(q)$.

Returns

- *galois.Poly* – Polynomial greatest common divisor of $a(x)$ and $b(x)$.
- *galois.Poly* – Polynomial $s(x)$, such that $a(x)s(x) + b(x)t(x) = \gcd(a(x), b(x))$.
- *galois.Poly* – Polynomial $t(x)$, such that $a(x)s(x) + b(x)t(x) = \gcd(a(x), b(x))$.

Examples

```
In [1]: GF = galois.GF(7)

In [2]: a = galois.Poly.Roots([2,2,2,3,6], field=GF); a
Out[2]: Poly(x^5 + 6x^4 + x + 3, GF(7))

# a(x) and b(x) only share the root 2 in common
In [3]: b = galois.Poly.Roots([1,2], field=GF); b
Out[3]: Poly(x^2 + 4x + 2, GF(7))

In [4]: gcd, x, y = galois.poly_egcd(a, b); gcd, x, y
Out[4]: (Poly(x + 5, GF(7)), Poly(5, GF(7)), Poly(2x^3 + 4x^2 + x + 2, GF(7)))

# The GCD has only 2 as a root with multiplicity 1
In [5]: gcd.roots(multiplicity=True)
Out[5]: (GF([2], order=7), array([1]))

In [6]: a*x + b*y == gcd
Out[6]: True
```

galois.poly_pow

galois.**poly_pow**(*poly*, *power*, *modulus*)

Efficiently exponentiates a polynomial $f(x)$ to the power k reducing by modulo $g(x)$, $f(x)^k$ mod $g(x)$.

The algorithm is more efficient than exponentiating first and then reducing modulo $g(x)$. Instead, this algorithm repeatedly squares $f(x)$, reducing modulo $g(x)$ at each step. This is the polynomial equivalent of *galois.pow()*.

Parameters

- **poly** (galois.Poly) – The polynomial to be exponentiated $f(x)$.
- **power** (*int*) – The non-negative exponent k.
- **modulus** (galois.Poly) – The reducing polynomial $g(x)$.

Returns The resulting polynomial $h(x) = f^k$ mod g.

Return type *galois.Poly*

Examples

```
In [1]: GF = galois.GF(31)

In [2]: f = galois.Poly.Random(10, field=GF); f
Out[2]: Poly(16x^10 + 29x^9 + 13x^8 + 24x^7 + 17x^6 + 17x^5 + 22x^4 + 15x^3 + 2x^2␣
↪+ 5x + 29, GF(31))

In [3]: g = galois.Poly.Random(7, field=GF); g
Out[3]: Poly(18x^7 + 17x^6 + 28x^5 + 8x^4 + 28x^3 + 8x^2 + 16x + 30, GF(31))

# %timeit f**200 % g
# 1.23 s ± 41.1 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

(continues on next page)

(continued from previous page)

```
In [4]: f**200 % g
Out[4]: Poly(9x^6 + 19x^5 + 20x^4 + 11x^3 + 30x^2 + 24x + 10, GF(31))

# %timeit galois.poly_pow(f, 200, g)
# 41.7 ms ± 468 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
In [5]: galois.poly_pow(f, 200, g)
Out[5]: Poly(9x^6 + 19x^5 + 20x^4 + 11x^3 + 30x^2 + 24x + 10, GF(31))
```

galois.poly_factors

galois.**poly_factors**(*poly*)

Factors the polynomial $f(x)$ into a product of n irreducible factors $f(x) = g_0(x)^{k_0} g_1(x)^{k_1} \dots g_{n-1}(x)^{k_{n-1}}$ with $k_0 \leq k_1 \leq \cdots \leq k_{n-1}$.

This function implements the Square-Free Factorization algorithm.

Parameters **poly** (galois.Poly) – The polynomial $f(x)$ over $\mathrm{GF}(p^m)$ to be factored.

Returns

- *list* – The list of n polynomial factors $\{g_0(x), g_1(x), \dots, g_{n-1}(x)\}$.
- *list* – The list of n polynomial multiplicities $\{k_0, k_1, \dots, k_{n-1}\}$.

References

- D. Hachenberger, D. Jungnickel. Topics in Galois Fields. Algorithm 6.1.7.

Examples

```
In [1]: GF = galois.GF2

# Ensure the factors are irreducible by using Conway polynomials
In [2]: g0, g1, g2 = galois.conway_poly(2, 3), galois.conway_poly(2, 4), galois.
↪conway_poly(2, 5)

In [3]: g0, g1, g2
Out[3]:
(Poly(x^3 + x + 1, GF(2)),
 Poly(x^4 + x + 1, GF(2)),
 Poly(x^5 + x^2 + 1, GF(2)))

In [4]: k0, k1, k2 = 2, 3, 4

# Construct the composite polynomial
In [5]: f = g0**k0 * g1**k1 * g2**k2

In [6]: galois.poly_factors(f)
Out[6]:
([Poly(x^3 + x + 1, GF(2)),
  Poly(x^4 + x + 1, GF(2)),
```

(continues on next page)

(continued from previous page)

```
 Poly(x^5 + x^2 + 1, GF(2))],
[2, 3, 4])
```

```
In [7]: GF = galois.GF(3)

# Ensure the factors are irreducible by using Conway polynomials
In [8]: g0, g1, g2 = galois.conway_poly(3, 3), galois.conway_poly(3, 4), galois.
↪conway_poly(3, 5)

In [9]: g0, g1, g2
Out[9]:
(Poly(x^3 + 2x + 1, GF(3)),
 Poly(x^4 + 2x^3 + 2, GF(3)),
 Poly(x^5 + 2x + 1, GF(3)))

In [10]: k0, k1, k2 = 3, 4, 6

# Construct the composite polynomial
In [11]: f = g0**k0 * g1**k1 * g2**k2

In [12]: galois.poly_factors(f)
Out[12]:
([Poly(x^3 + 2x + 1, GF(3)),
  Poly(x^4 + 2x^3 + 2, GF(3)),
  Poly(x^5 + 2x + 1, GF(3))],
 [3, 4, 6])
```

5.2.3 Special polynomial creation

irreducible_poly(characteristic, degree[, ...])	Returns a monic irreducible polynomial $f(x)$ over $\mathrm{GF}(p)$ with degree m.
irreducible_polys(characteristic, degree)	Returns all monic irreducible polynomials $f(x)$ over $\mathrm{GF}(p)$ with degree m.
primitive_poly(characteristic, degree[, method])	Returns a monic primitive polynomial $f(x)$ over $\mathrm{GF}(p)$ with degree m.
primitive_polys(characteristic, degree)	Returns all monic primitive polynomials $f(x)$ over $\mathrm{GF}(p)$ with degree m.
conway_poly(characteristic, degree)	Returns the Conway polynomial $C_{p,m}(x)$ over $\mathrm{GF}(p)$ with degree m.
matlab_primitive_poly(characteristic, degree)	Returns Matlab's default primitive polynomial $f(x)$ over $\mathrm{GF}(p)$ with degree m.
minimal_poly(element)	Computes the minimal polynomial $m_e(x) \in \mathrm{GF}(p)[x]$ of a Galois field element $e \in \mathrm{GF}(p^m)$.

5.2.4 Polynomial tests

is_monic(poly)	Determines whether the polynomial is monic, i.e. having leading coefficient equal to 1.
is_irreducible(poly)	Checks whether the polynomial $f(x)$ over $\mathrm{GF}(p)$ is irreducible.
is_primitive(poly)	Checks whether the polynomial $f(x)$ over $\mathrm{GF}(p)$ is primitive.

galois.is_monic

galois.**is_monic**(*poly*)

Determines whether the polynomial is monic, i.e. having leading coefficient equal to 1.

Parameters **poly** (galois.Poly) – A polynomial over a Galois field.

Returns `True` if the polynomial is monic.

Return type bool

Examples

```
In [1]: GF = galois.GF(7)

In [2]: p = galois.Poly([1,0,4,5], field=GF); p
Out[2]: Poly(x^3 + 4x + 5, GF(7))

In [3]: galois.is_monic(p)
Out[3]: True
```

```
In [4]: p = galois.Poly([3,0,4,5], field=GF); p
Out[4]: Poly(3x^3 + 4x + 5, GF(7))

In [5]: galois.is_monic(p)
Out[5]: False
```

5.3 Number Theory

This section contains functions for performing modular arithmetic and other number theoretic routines.

5.3.1 Divisibility

gcd(a, b)	Finds the greatest common divisor of the integers a and b.
egcd(a, b)	Finds the integer multiplicands of a and b such that $ax + by = \gcd(a, b)$.
lcm(*integers)	Computes the least common multiple of the integer arguments.
euler_phi(n)	Counts the positive integers (totatives) in $1 \leq k < n$ that are coprime to n.
totatives(n)	Returns the positive integers (totatives) in $1 \leq k < n$ that are coprime to n.

galois.gcd

galois.**gcd**(a, b)

Finds the greatest common divisor of the integers a and b.

Parameters

- **a** (*int*) – Any integer.
- **b** (*int*) – Any integer.

Returns Greatest common divisor of a and b.

Return type int

Examples

```
In [1]: a = 12

In [2]: b = 28

In [3]: galois.gcd(a, b)
Out[3]: 4
```

galois.egcd

galois.**egcd**(a, b)

Finds the integer multiplicands of a and b such that $ax + by = \gcd(a, b)$.

This function implements the Extended Euclidean Algorithm.

Parameters

- **a** (*int*) – Any integer.
- **b** (*int*) – Any integer.

Returns

- *int* – Greatest common divisor of a and b.
- *int* – Integer x, such that $ax + by = \gcd(a, b)$.
- *int* – Integer y, such that $ax + by = \gcd(a, b)$.

References

- T. Moon, "Error Correction Coding", Section 5.2.2: The Euclidean Algorithm and Euclidean Domains, p. 181
- https://en.wikipedia.org/wiki/Euclidean_algorithm#Extended_Euclidean_algorithm

Examples

```
In [1]: a = 12

In [2]: b = 28

In [3]: gcd, x, y = galois.egcd(a, b)

In [4]: gcd, x, y
Out[4]: (4, -2, 1)

In [5]: a*x + b*y == gcd
Out[5]: True
```

galois.lcm

galois.**lcm**(**integers*)

Computes the least common multiple of the integer arguments.

Note: This function is included for Python versions before 3.9. For Python 3.9 and later, this function calls `math.lcm()` from the standard library.

Returns The least common multiple of the integer arguments. If any argument is 0, the LCM is 0. If no arguments are provided, 1 is returned.

Return type int

Examples

```
In [1]: galois.lcm()
Out[1]: 1

In [2]: galois.lcm(2, 4, 14)
Out[2]: 28

In [3]: galois.lcm(3, 0, 9)
Out[3]: 0
```

This function also works on arbitrarily-large integers.

```
In [4]: prime1, prime2 = galois.mersenne_primes(100)[-2:]

In [5]: prime1, prime2
Out[5]: (2305843009213693951, 618970019642690137449562111)

In [6]: lcm = galois.lcm(prime1, prime2); lcm
Out[6]: 1427247692705959880439315947500961989719490561

In [7]: lcm == prime1 * prime2
Out[7]: True
```

galois.euler_phi

galois.**euler_phi**(n)

Counts the positive integers (totatives) in $1 \leq k < n$ that are coprime to n.

This function implements the Euler totient function

$$\phi(n) = n \prod_{p \mid n} \left(1 - \frac{1}{p}\right) = \prod_{i=1}^{k} p_i^{e_i - 1}(p_i - 1)$$

for prime p and the prime factorization $n = p_1^{e_1} \dots p_k^{e_k}$.

Parameters **n** (*int*) – A positive integer.

Returns The number of totatives that are coprime to n.

Return type int

References

- https://en.wikipedia.org/wiki/Euler%27s_totient_function
- https://oeis.org/A000010

Examples

```
In [1]: n = 20

In [2]: phi = galois.euler_phi(n); phi
Out[2]: 8

# Find the totatives that are coprime with n
In [3]: totatives = [k for k in range(n) if math.gcd(k, n) == 1]; totatives
Out[3]: [1, 3, 7, 9, 11, 13, 17, 19]

# The number of totatives is phi
In [4]: len(totatives) == phi
Out[4]: True

# For prime n, phi is always n-1
In [5]: galois.euler_phi(13)
Out[5]: 12
```

galois.totatives

galois.**totatives**(n)

Returns the positive integers (totatives) in $1 \leq k < n$ that are coprime to n.

The totatives of n form the multiplicative group $\mathbb{Z}_n^\times$.

Parameters **n** (*int*) – A positive integer.

Returns The totatives of n.

Return type list

References

- https://en.wikipedia.org/wiki/Totative
- https://oeis.org/A000010

Examples

```
In [1]: n = 20

In [2]: totatives = galois.totatives(n); totatives
Out[2]: [1, 3, 7, 9, 11, 13, 17, 19]

In [3]: phi = galois.euler_phi(n); phi
Out[3]: 8

In [4]: len(totatives) == phi
Out[4]: True
```

are_coprime(*integers)	Determines if the integer arguments are pairwise coprime.

galois.are_coprime

galois.**are_coprime**(**integers*)

Determines if the integer arguments are pairwise coprime.

Parameters ***integers** (*tuple of int*) – Each argument must be an integer.

Returns `True` if the integer arguments are pairwise coprime.

Return type bool

Examples

```
In [1]: galois.are_coprime(3, 4, 5)
Out[1]: True

In [2]: galois.are_coprime(3, 7, 9, 11)
Out[2]: False
```

5.3.2 Congruences

pow(base, exp, mod)	Efficiently exponentiates an integer $a^k \pmod m$.
crt(a, m)	Solves the simultaneous system of congruences for x.
primitive_root(n[, start, stop, reverse])	Finds the smallest primitive root modulo n.
primitive_roots(n[, start, stop, reverse])	Finds all primitive roots modulo n.
carmichael_lambda(n)	Finds the smallest positive integer m such that $a^m \equiv 1 \pmod n$ for every integer a in $1 \leq a < n$ that is coprime to n.
legendre_symbol(a, p)	Computes the Legendre symbol $(\frac{a}{p})$.
jacobi_symbol(a, n)	Computes the Jacobi symbol $(\frac{a}{n})$.
kronecker_symbol(a, n)	Computes the Kronecker symbol $(\frac{a}{n})$.

galois.pow

galois.**pow**(*base*, *exp*, *mod*)

Efficiently exponentiates an integer $a^k \pmod m$.

The algorithm is more efficient than exponentiating first and then reducing modulo m. This is the integer equivalent of *galois.poly_pow()*.

Note: This function is an alias of *pow()* in the standard library.

Parameters

- **base** (*int*) – The integer base a.

- **exp** (*int*) – The integer exponent k.
- **mod** (*int*) – The integer modulus m.

Returns The modular exponentiation $a^k \pmod{m}$.

Return type int

Examples

```
In [1]: galois.pow(3, 5, 7)
Out[1]: 5

In [2]: (3**5) % 7
Out[2]: 5
```

galois.crt

galois.**crt**(a, m)

Solves the simultaneous system of congruences for x.

This function implements the Chinese Remainder Theorem.

$$
\begin{aligned}
x &\equiv a_1 \pmod{m_1} \\
x &\equiv a_2 \pmod{m_2} \\
x &\equiv \ldots \\
x &\equiv a_n \pmod{m_n}
\end{aligned}
$$

Parameters

- **a** (*array_like*) – The integer remainders a_i.
- **m** (*array_like*) – The integer modulii m_i.

Returns The simultaneous solution x to the system of congruences.

Return type int

Examples

```
In [1]: a = [0, 3, 4]

In [2]: m = [3, 4, 5]

In [3]: x = galois.crt(a, m); x
Out[3]: 39

In [4]: for i in range(len(a)):
   ...:     ai = x % m[i]
   ...:     print(f"{x} = {ai} (mod {m[i]}), Valid congruence: {ai == a[i]}")
   ...:
39 = 0 (mod 3), Valid congruence: True
39 = 3 (mod 4), Valid congruence: True
39 = 4 (mod 5), Valid congruence: True
```

galois.carmichael_lambda

galois.**carmichael_lambda**(n)

Finds the smallest positive integer m such that $a^m \equiv 1 \pmod{n}$ for every integer a in $1 \leq a < n$ that is coprime to n.

Implements the Carmichael function $\lambda(n)$.

Parameters **n** (*int*) – A positive integer.

Returns The smallest positive integer m such that $a^m \equiv 1 \pmod{n}$ for every a in $1 \leq a < n$ that is coprime to n.

Return type int

References

- https://en.wikipedia.org/wiki/Carmichael_function
- https://oeis.org/A002322

Examples

```
In [1]: n = 20

In [2]: lambda_ = galois.carmichael_lambda(n); lambda_
Out[2]: 4

# Find the totatives that are relatively coprime with n
In [3]: totatives = [i for i in range(n) if math.gcd(i, n) == 1]; totatives
Out[3]: [1, 3, 7, 9, 11, 13, 17, 19]

In [4]: for a in totatives:
   ...:     result = pow(a, lambda_, n)
   ...:     print("{:2d}^{} = {} (mod {})".format(a, lambda_, result, n))
   ...:
 1^4 = 1 (mod 20)
 3^4 = 1 (mod 20)
 7^4 = 1 (mod 20)
 9^4 = 1 (mod 20)
11^4 = 1 (mod 20)
13^4 = 1 (mod 20)
17^4 = 1 (mod 20)
19^4 = 1 (mod 20)

# For prime n, phi and lambda are always n-1
In [5]: galois.euler_phi(13), galois.carmichael_lambda(13)
Out[5]: (12, 12)
```

galois.legendre_symbol

galois.**legendre_symbol**(a, p)

Computes the Legendre symbol $(\frac{a}{p})$.

The Legendre symbol is useful for determining if a is a quadratic residue modulo p, namely $a \in Q_p$. A quadratic residue a modulo p satisfies $x^2 \equiv a \pmod{p}$ for some x.

$$\left(\frac{a}{p}\right) = \begin{cases} 0, & p \mid a \\ 1, & a \in Q_p \\ -1, & a \in \overline{Q}_p \end{cases}$$

Parameters

- **a** (*int*) – An integer.
- **p** (*int*) – An odd prime $p \geq 3$.

Returns The Legendre symbol $(\frac{a}{p})$ with value in $\{0, 1, -1\}$.

Return type int

References

- Algorithm 2.149 from https://cacr.uwaterloo.ca/hac/about/chap2.pdf

Examples

The quadratic residues modulo 7 are $Q_7 = \{1, 2, 4\}$. The quadratic non-residues modulo 7 are $\overline{Q}_7 = \{3, 5, 6\}$.

```
In [1]: [pow(x, 2, 7) for x in range(7)]
Out[1]: [0, 1, 4, 2, 2, 4, 1]

In [2]: for a in range(7):
   ...:     print(f"({a} / 7) = {galois.legendre_symbol(a, 7)}")
   ...:
(0 / 7) = 0
(1 / 7) = 1
(2 / 7) = 1
(3 / 7) = -1
(4 / 7) = 1
(5 / 7) = -1
(6 / 7) = -1
```

galois.jacobi_symbol

galois.**jacobi_symbol**(a, n)

Computes the Jacobi symbol $(\frac{a}{n})$.

The Jacobi symbol extends the Legendre symbol for odd $n \geq 3$. Unlike the Legendre symbol, $(\frac{a}{n}) = 1$ does not imply a is a quadratic residue modulo n. However, all $a \in Q_n$ have $(\frac{a}{n}) = 1$.

Parameters

- **a** (*int*) – An integer.

- **n** (*int*) – An odd integer $n \geq 3$.

Returns The Jacobi symbol $(\frac{a}{n})$ with value in $\{0, 1, -1\}$.

Return type int

References

- Algorithm 2.149 from https://cacr.uwaterloo.ca/hac/about/chap2.pdf

---

Examples

The quadratic residues modulo 9 are $Q_9 = \{1, 4, 7\}$ and these all satisfy $(\frac{a}{9}) = 1$. The quadratic non-residues modulo 9 are $\overline{Q}_9 = \{2, 3, 5, 6, 8\}$, but notice $\{2, 5, 8\}$ also satisfy $(\frac{a}{9}) = 1$. The set of integers $\{3, 6\}$ not coprime to n satisfies $(\frac{a}{9}) = 0$.

```
In [1]: [pow(x, 2, 9) for x in range(9)]
Out[1]: [0, 1, 4, 0, 7, 7, 0, 4, 1]

In [2]: for a in range(9):
   ...:     print(f"({a} / 9) = {galois.jacobi_symbol(a, 9)}")
   ...:
(0 / 9) = 0
(1 / 9) = 1
(2 / 9) = 1
(3 / 9) = 0
(4 / 9) = 1
(5 / 9) = 1
(6 / 9) = 0
(7 / 9) = 1
(8 / 9) = 1
```

---

galois.kronecker_symbol

galois.**kronecker_symbol**(a, n)

Computes the Kronecker symbol $(\frac{a}{n})$.

The Kronecker symbol extends the Jacobi symbol for all n.

Parameters

- **a** (*int*) – An integer.
- **n** (*int*) – An integer.

Returns The Kronecker symbol $(\frac{a}{n})$ with value in $\{0, -1, 1\}$.

Return type int

References

- Algorithm 2.149 from https://cacr.uwaterloo.ca/hac/about/chap2.pdf

is_primitive_root(g, n)	Determines if g is a primitive root modulo n.
is_cyclic(n)	Determines whether the multiplicative group $(\mathbb{Z}/n\mathbb{Z})^\times$ is cyclic.

galois.is_cyclic

galois.**is_cyclic**(n)

Determines whether the multiplicative group $(\mathbb{Z}/n\mathbb{Z})^\times$ is cyclic.

The multiplicative group $(\mathbb{Z}/n\mathbb{Z})^\times$ is the set of positive integers $1 \leq a < n$ that are coprime with n. $(\mathbb{Z}/n\mathbb{Z})^\times$ being cyclic means that some primitive root of n, or generator, g can generate the group $\{g^0, g^1, g^2, \ldots, g^{\phi(n)-1}\}$, where $\phi(n)$ is Euler's totient function and calculates the order of the group. If $(\mathbb{Z}/n\mathbb{Z})^\times$ is cyclic, the number of primitive roots is found by $\phi(\phi(n))$.

$(\mathbb{Z}/n\mathbb{Z})^\times$ is *cyclic* if and only if n is 2, 4, p^k, or $2p^k$, where p is an odd prime and k is a positive integer.

Parameters **n** (*int*) – A positive integer.

Returns `True` if the multiplicative group $(\mathbb{Z}/n\mathbb{Z})^\times$ is cyclic.

Return type bool

Examples

The elements of $(\mathbb{Z}/n\mathbb{Z})^\times$ are the positive integers less than n that are coprime with n. For example, $(\mathbb{Z}/14\mathbb{Z})^\times = \{1, 3, 5, 9, 11, 13\}$.

```
# n is of type 2*p^k, which is cyclic
In [1]: n = 14

In [2]: galois.is_cyclic(n)
Out[2]: True

# The congruence class coprime with n
In [3]: Znx = set([a for a in range(1, n) if math.gcd(n, a) == 1]); Znx
Out[3]: {1, 3, 5, 9, 11, 13}

# Euler's totient function counts the "totatives", positive integers coprime with n
In [4]: phi = galois.euler_phi(n); phi
Out[4]: 6

In [5]: len(Znx) == phi
Out[5]: True

# The primitive roots are the elements in Znx that multiplicatively generate the␣
↪group
In [6]: for a in Znx:
   ...:     span = set([pow(a, i, n) for i in range(1, phi + 1)])
   ...:     primitive_root = span == Znx
   ...:     print("Element: {:2d}, Span: {:<20}, Primitive root: {}".format(a,␣
↪str(span), primitive_root))
```

(continues on next page)

(continued from previous page)

```
   ...:
Element:  1, Span: {1}                 , Primitive root: False
Element:  3, Span: {1, 3, 5, 9, 11, 13}, Primitive root: True
Element:  5, Span: {1, 3, 5, 9, 11, 13}, Primitive root: True
Element:  9, Span: {9, 11, 1}          , Primitive root: False
Element: 11, Span: {9, 11, 1}          , Primitive root: False
Element: 13, Span: {1, 13}             , Primitive root: False

In [7]: roots = galois.primitive_roots(n); roots
Out[7]: [3, 5]

# Euler's totient function phi(phi(n)) counts the primitive roots of n
In [8]: len(roots) == galois.euler_phi(phi)
Out[8]: True
```

A counterexample is $n = 15 = 3 * 5$, which doesn't fit the condition for cyclicness. $(\mathbb{Z}/15\mathbb{Z})^\times = \{1, 2, 4, 7, 8, 11, 13, 14\}$.

```
# n is of type p1^k1 * p2^k2, which is not cyclic
In [9]: n = 15

In [10]: galois.is_cyclic(n)
Out[10]: False

# The congruence class coprime with n
In [11]: Znx = set([a for a in range(1, n) if math.gcd(n, a) == 1]); Znx
Out[11]: {1, 2, 4, 7, 8, 11, 13, 14}

# Euler's totient function counts the "totatives", positive integers coprime with n
In [12]: phi = galois.euler_phi(n); phi
Out[12]: 8

In [13]: len(Znx) == phi
Out[13]: True

# The primitive roots are the elements in Znx that multiplicatively generate the␣
↪group
In [14]: for a in Znx:
   ....:     span = set([pow(a, i, n) for i in range(1, phi + 1)])
   ....:     primitive_root = span == Znx
   ....:     print("Element: {:2d}, Span: {:<13}, Primitive root: {}".format(a,␣
↪str(span), primitive_root))
   ....:
Element:  1, Span: {1}             , Primitive root: False
Element:  2, Span: {8, 1, 2, 4}    , Primitive root: False
Element:  4, Span: {1, 4}          , Primitive root: False
Element:  7, Span: {1, 4, 13, 7}   , Primitive root: False
Element:  8, Span: {8, 1, 2, 4}    , Primitive root: False
Element: 11, Span: {1, 11}         , Primitive root: False
Element: 13, Span: {1, 4, 13, 7}   , Primitive root: False
Element: 14, Span: {1, 14}         , Primitive root: False
```

(continues on next page)

(continued from previous page)

```
In [15]: roots = galois.primitive_roots(n); roots
Out[15]: []

# Note the max order of any element is 4, not 8, which is Carmichael's lambda␣
↪function
In [16]: galois.carmichael_lambda(n)
Out[16]: 4
```

5.3.3 Integer arithmetic

isqrt(n)	Computes $x = \lfloor \sqrt{n} \rfloor$ such that $x^2 \leq n < (x+1)^2$.
iroot(n, k)	Computes $x = \lfloor (n)^{\frac{1}{k}} \rfloor$ such that $x^k \leq n < (x+1)^k$.
ilog(n, b)	Computes $x = \lfloor \log_b(n) \rfloor$ such that $b^x \leq n < b^{x+1}$.

galois.isqrt

galois.**isqrt**(*n*)

Computes $x = \lfloor \sqrt{n} \rfloor$ such that $x^2 \leq n < (x+1)^2$.

Note: This function is included for Python versions before 3.8. For Python 3.8 and later, this function calls `math.isqrt()` from the standard library.

Parameters **n** (*int*) – A non-negative integer.

Returns The integer square root of n.

Return type int

Examples

```
In [1]: galois.isqrt(27**2 - 1)
Out[1]: 26

In [2]: galois.isqrt(27**2)
Out[2]: 27

In [3]: galois.isqrt(27**2 + 1)
Out[3]: 27
```

galois.iroot

galois.**iroot**(n, k)

Computes $x = \lfloor (n)^{\frac{1}{k}} \rfloor$ such that $x^k \leq n < (x+1)^k$.

Parameters

- **n** (*int*) – A non-negative integer.
- **k** (*int*) – The root k, must be at least 2.

Returns The integer k-th root of n.

Return type int

Examples

```
In [1]: galois.iroot(27**5 - 1, 5)
Out[1]: 26

In [2]: galois.iroot(27**5, 5)
Out[2]: 27

In [3]: galois.iroot(27**5 + 1, 5)
Out[3]: 27
```

galois.ilog

galois.**ilog**(n, b)

Computes $x = \lfloor \log_b(n) \rfloor$ such that $b^x \leq n < b^{x+1}$.

Parameters

- **n** (*int*) – A positive integer.
- **b** (*int*) – The logarithm base b.

Returns The integer logarithm base b of n.

Return type int

Examples

```
In [1]: galois.ilog(27**5 - 1, 27)
Out[1]: 4

In [2]: galois.ilog(27**5, 27)
Out[2]: 5

In [3]: galois.ilog(27**5 + 1, 27)
Out[3]: 5
```

5.3.4 Discrete logarithms

log_naive(beta, alpha, modulus)	Computes the discrete logarithm $x = \log_\alpha(\beta) \pmod m$.

galois.log_naive

galois.**log_naive**(*beta*, *alpha*, *modulus*)

Computes the discrete logarithm $x = \log_\alpha(\beta) \pmod m$.

This function implements the naive algorithm. It is included for testing and reference.

Parameters

- **beta** (*int*) – The integer β to compute the logarithm of.
- **alpha** (*int*) – The base α.
- **modulus** (*int*) – The modulus m.

Examples

```
In [1]: N = 17

In [2]: galois.totatives(N)
Out[2]: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16]

In [3]: galois.primitive_roots(N)
Out[3]: [3, 5, 6, 7, 10, 11, 12, 14]

In [4]: x = galois.log_naive(3, 7, N); x
Out[4]: 3

In [5]: 7**x % N
Out[5]: 3
```

```
In [6]: N = 18

In [7]: galois.totatives(N)
Out[7]: [1, 5, 7, 11, 13, 17]

In [8]: galois.primitive_roots(N)
Out[8]: [5, 11]

In [9]: x = galois.log_naive(11, 5, N); x
Out[9]: 5

In [10]: 5**x % N
Out[10]: 11
```

5.4 Integer Factorization

This section contains functions for factoring integers and analyzing their properties.

5.4.1 Prime factorization

factors(n)	Computes the prime factors of the positive integer n.

galois.factors

galois.**factors**(n)

Computes the prime factors of the positive integer n.

The integer n can be factored into $n = p_1^{e_1} p_2^{e_2} \dots p_{k-1}^{e_{k-1}}$.

Steps:

1. Test if n is prime. If so, return `[n], [1]`.
2. Test if n is a perfect power, such that $n = x^k$. If so, prime factor x and multiply its exponents by k.
3. Use trial division with a list of primes up to 10^6. If no residual factors, return the discovered prime factors.
4. Use Pollard's Rho algorithm to find a non-trivial factor of the residual. Continue until all are found.

Parameters **n** (*int*) – Any positive integer.

Returns

- *list* – Sorted list of k prime factors $p = [p_1, p_2, \dots, p_{k-1}]$ with $p_1 < p_2 < \dots < p_{k-1}$.
- *list* – List of corresponding prime powers $e = [e_1, e_2, \dots, e_{k-1}]$.

Examples

```
In [1]: p, e = galois.factors(120)

In [2]: p, e
Out[2]: ([2, 3, 5], [3, 1, 1])

# The product of the prime powers is the factored integer
In [3]: np.multiply.reduce(np.array(p) ** np.array(e))
Out[3]: 120
```

Prime factorization of 1 less than a large prime.

```
In [4]: prime =100000000000000035000061

In [5]: galois.is_prime(prime)
Out[5]: True

In [6]: p, e = galois.factors(prime - 1)

In [7]: p, e
```

(continues on next page)

(continued from previous page)

```
Out[7]: ([2, 3, 5, 17, 19, 112850813, 457237177399], [2, 1, 1, 1, 1, 1, 1])

In [8]: np.multiply.reduce(np.array(p) ** np.array(e))
Out[8]: 2003764205241896700
```

5.4.2 Composite factorization

divisors(n)	Computes all positive integer divisors d of the integer n such that $d \mid n$.
divisor_sigma(n[, k])	Returns the sum of k-th powers of the positive divisors of n.

galois.divisors

galois.**divisors**(n)

Computes all positive integer divisors d of the integer n such that $d \mid n$.

Parameters **n** (*int*) – Any integer.

Returns Sorted list of integer divisors d.

Return type list

Examples

```
In [1]: galois.divisors(0)
Out[1]: []

In [2]: galois.divisors(1)
Out[2]: [1]

In [3]: galois.divisors(24)
Out[3]: [1, 2, 3, 4, 6, 8, 12, 24]

In [4]: galois.divisors(-24)
Out[4]: [1, 2, 3, 4, 6, 8, 12, 24]
```

galois.divisor_sigma

galois.**divisor_sigma**(n, *k=1*)

Returns the sum of k-th powers of the positive divisors of n.

This function implements the $\sigma_k(n)$ function. It is defined as:

$$\sigma_k(n) = \sum_{d \mid n} d^k$$

Parameters

- **n** (*int*) – Any integer.
- **k** (*int, optional*) – The degree of the positive divisors. The default is 1 which corresponds to $\sigma_1(n)$ which is the sum of positive divisors.

Returns The sum of divisors function $\sigma_k(n)$.

Return type int

Examples

```
In [1]: galois.divisors(9)
Out[1]: [1, 3, 9]

In [2]: galois.divisor_sigma(9, k=0)
Out[2]: 3

In [3]: galois.divisor_sigma(9, k=1)
Out[3]: 13

In [4]: galois.divisor_sigma(9, k=2)
Out[4]: 91
```

5.4.3 Specific factorization algorithms

perfect_power(n)	Returns the integer base $m > 1$ and exponent $e > 1$ of $n = m^e$ if n is a perfect power.
trial_division(n[, B])	Finds all the prime factors $p_i^{e_i}$ of n for $p_i \leq B$.
pollard_p1(n, B[, B2])	Attempts to find a non-trivial factor of n if it has a prime factor p such that $p - 1$ is B-smooth.
pollard_rho(n[, c])	Attempts to find a non-trivial factor of n using cycle detection.

galois.perfect_power

galois.**perfect_power**(*n*)

Returns the integer base $m > 1$ and exponent $e > 1$ of $n = m^e$ if n is a perfect power.

Parameters **n** (*int*) – A positive integer $n > 1$.

Returns None is n is not a perfect power. Otherwise, (c, e) such that $n = c^e$. c may be composite.

Return type None, tuple

Examples

```
# Primes are not perfect powers
In [1]: galois.perfect_power(5)

# Products of primes are not perfect powers
In [2]: galois.perfect_power(6)
```

(continues on next page)

(continued from previous page)

```
# Products of prime powers were the GCD of the exponents is 1 are not perfect powers
In [3]: galois.perfect_power(36*125)

In [4]: galois.perfect_power(36)
Out[4]: (6, 2)

In [5]: galois.perfect_power(125)
Out[5]: (5, 3)
```

galois.trial_division

galois.**trial_division**(*n*, *B=None*)

Finds all the prime factors $p_i^{e_i}$ of n for $p_i \leq B$.

The trial division factorization will find all prime factors $p_i \leq B$ such that n factors as $n = p_1^{e_1} \dots p_k^{e_k} n_r$ where n_r is a residual factor (which may be composite).

Parameters

- **n** (*int*) – A positive integer.
- **B** (*int, optional*) – The max divisor in the trial division. The default is `None` which corresponds to $B = \sqrt{n}$. If $B > \sqrt{n}$, the algorithm will only search up to $\sqrt{n}$, since a factor of n cannot be larger than $\sqrt{n}$.

Returns

- *list* – The discovered prime factors $\{p_1, \dots, p_k\}$.
- *list* – The corresponding prime exponents $\{e_1, \dots, e_k\}$.
- *int* – The residual factor n_r.

Examples

```
In [1]: n = 2**4 * 17**3 * 113 * 15013

In [2]: galois.trial_division(n)
Out[2]: ([2, 17, 113, 15013], [4, 3, 1, 1], 1)

In [3]: galois.trial_division(n, B=500)
Out[3]: ([2, 17, 113], [4, 3, 1], 15013)

In [4]: galois.trial_division(n, B=100)
Out[4]: ([2, 17], [4, 3], 1696469)
```

galois.pollard_p1

galois.**pollard_p1**(*n*, *B*, *B2=None*)

Attempts to find a non-trivial factor of n if it has a prime factor p such that $p-1$ is B-smooth.

For a given odd composite n with a prime factor p, Pollard's $p-1$ algorithm can discover a non-trivial factor of n if $p-1$ is B-smooth. Specifically, the prime factorization must satisfy $p-1 = p_1^{e_1} \dots p_k^{e_k}$ with each $p_i \leq B$.

A extension of Pollard's $p-1$ algorithm allows a prime factor p to be B-smooth with the exception of one prime factor $B < p_{k+1} \leq B_2$. In this case, the prime factorization is $p-1 = p_1^{e_1} \dots p_k^{e_k} p_{k+1}$. Often B_2 is chosen such that $B_2 \gg B$.

Parameters

- **n** (*int*) – An odd composite integer $n > 2$ that is not a prime power.
- **B** (*int*) – The smoothness bound $B > 2$.
- **B2** (*int, optional*) – The smoothness bound B_2 for the optional second step of the algorithm. The default is `None`, which will not perform the second step.

Returns A non-trivial factor of n, if found. `None` if not found.

Return type None, int

References

- Section 3.2.3 from https://cacr.uwaterloo.ca/hac/about/chap3.pdf

---

Examples

Here, $n = pq$ where $p-1$ is 1039-smooth and $q-1$ is 17-smooth.

```
In [1]: p, q = 1458757, 1326001

In [2]: galois.factors(p - 1)
Out[2]: ([2, 3, 13, 1039], [2, 3, 1, 1])

In [3]: galois.factors(q - 1)
Out[3]: ([2, 3, 5, 13, 17], [4, 1, 3, 1, 1])
```

Searching with $B = 15$ will not recover a prime factor.

```
In [4]: galois.pollard_p1(p*q, 15)
```

Searching with $B = 17$ will recover the prime factor q.

```
In [5]: galois.pollard_p1(p*q, 17)
Out[5]: 1326001
```

Searching $B = 15$ will not recover a prime factor in the first step, but will find q in the second step because $p_{k+1} = 17$ satisfies $15 < 17 \leq 100$.

```
In [6]: galois.pollard_p1(p*q, 15, B2=100)
Out[6]: 1326001
```

Pollard's $p-1$ algorithm may return a composite factor.

```
In [7]: n = 2133861346249

In [8]: galois.factors(n)
Out[8]: ([37, 41, 5471, 257107], [1, 1, 1, 1])

In [9]: galois.pollard_p1(n, 10)
Out[9]: 1517

In [10]: 37*41
Out[10]: 1517
```

---

galois.pollard_rho

galois.**pollard_rho**(*n*, *c=1*)

Attempts to find a non-trivial factor of n using cycle detection.

Pollard's ρ algorithm seeks to find a non-trivial factor of n by finding a cycle in a sequence of integers $x_0, x_1, \ldots$ defined by $x_i = f(x_{i-1}) = x_{i-1}^2 + 1 \bmod p$ where p is an unknown small prime factor of n. This happens when $x_m \equiv x_{2m} \pmod{p}$. Because p is unknown, this is accomplished by computing the sequence modulo n and looking for $\gcd(x_m - x_{2m}, n) > 1$.

Parameters

- **n** (*int*) – An odd composite integer $n > 2$ that is not a prime power.
- **c** (*int, optional*) – The constant offset in the function $f(x) = x^2 + c \bmod n$. The default is 1. A requirement of the algorithm is that $c \notin \{0, -2\}$.

Returns A non-trivial factor m of n, if found. `None` if not found.

Return type None, int

References

- Section 3.2.2 from https://cacr.uwaterloo.ca/hac/about/chap3.pdf

---

Examples

Pollard's ρ is especially good at finding small factors.

```
In [1]: n = 503**7 * 10007 * 1000003

In [2]: galois.pollard_rho(n)
Out[2]: 503
```

It is also efficient for finding relatively small factors.

```
In [3]: n = 1182640843 * 1716279751

In [4]: galois.pollard_rho(n)
Out[4]: 1716279751
```

5.4.4 Integer tests

is_prime(n)	Determines if n is prime.
is_prime_power(n)	Determines if n is a prime power $n = p^k$ for prime p and $k \geq 1$.
is_perfect_power(n)	Determines if n is a perfect power $n = x^k$ for $x > 0$ and $k \geq 2$.
is_composite(n)	Determines if n is composite.
is_square_free(n)	Determines if n is square-free, such that $n = p_1 p_2 \cdots p_k$.
is_smooth(n, B)	Determines if the positive integer n is B-smooth.
is_powersmooth(n, B)	Determines if the positive integer n is B-powersmooth.

galois.is_prime

galois.**is_prime**(n)

Determines if n is prime.

This algorithm will first run Fermat's primality test to check n for compositeness, see *galois.fermat_primality_test()*. If it determines n is composite, the function will quickly return. If Fermat's primality test returns `True`, then n could be prime or pseudoprime. If so, then the algorithm will run seven rounds of Miller-Rabin's primality test, see *galois.miller_rabin_primality_test()*. With this many rounds, a result of `True` should have high probability of n being a true prime, not a pseudoprime.

Parameters **n** (*int*) – A positive integer.

Returns `True` if the integer n is prime.

Return type bool

Examples

```
In [1]: galois.is_prime(13)
Out[1]: True

In [2]: galois.is_prime(15)
Out[2]: False
```

The algorithm is also efficient on very large n.

```
In [3]: galois.is_prime(100000000000000000035000061)
Out[3]: True
```

galois.is_prime_power

galois.**is_prime_power**(*n*)

Determines if n is a prime power $n = p^k$ for prime p and $k \geq 1$.

There is some controversy over whether 1 is a prime power p^0. Since 1 is the 0-th power of all primes, it is often regarded not as a prime power. This function returns `False` for 1.

Parameters **n** (*int*) – A positive integer.

Returns `True` if the integer n is a prime power.

Return type bool

Examples

```
In [1]: galois.is_prime_power(8)
Out[1]: True

In [2]: galois.is_prime_power(6)
Out[2]: False
```

galois.is_perfect_power

galois.**is_perfect_power**(*n*)

Determines if n is a perfect power $n = x^k$ for $x > 0$ and $k \geq 2$.

Parameters **n** (*int*) – A positive integer.

Returns `True` if the integer n is a perfect power.

Return type bool

Examples

```
In [1]: galois.is_perfect_power(8)
Out[1]: True

In [2]: galois.is_perfect_power(16)
Out[2]: True

In [3]: galois.is_perfect_power(20)
Out[3]: False
```

galois.is_composite

galois.**is_composite**(n)

Determines if n is composite.

Parameters **n** (*int*) – A positive integer.

Returns `True` if the integer n is composite.

Return type bool

Examples

```
In [1]: galois.is_composite(13)
Out[1]: False

In [2]: galois.is_composite(15)
Out[2]: True
```

galois.is_square_free

galois.**is_square_free**(n)

Determines if n is square-free, such that $n = p_1 p_2 \dots p_k$.

A square-free integer n is divisible by no perfect squares. As a consequence, the prime factorization of a square-free integer n is

$$n = \prod_{i=1}^{k} p_i^{e_i} = \prod_{i=1}^{k} p_i.$$

Parameters **n** (*int*) – A positive integer.

Returns `True` if the integer n is square-free.

Return type bool

Examples

```
In [1]: galois.is_square_free(10)
Out[1]: True

In [2]: galois.is_square_free(16)
Out[2]: False
```

galois.is_smooth

galois.**is_smooth**(n, B)

Determines if the positive integer n is B-smooth.

An integer n with prime factorization $n = p_1^{e_1} \dots p_k^{e_k}$ is B-smooth if $p_k \leq B$. The 2-smooth numbers are the powers of 2. The 5-smooth numbers are known as *regular numbers*. The 7-smooth numbers are known as *humble numbers* or *highly composite numbers*.

Parameters

- **n** (*int*) – A positive integer.
- **B** (*int*) – The smoothness bound $B \geq 2$.

Returns `True` if n is B-smooth.

Return type bool

Examples

```
In [1]: galois.is_smooth(2**10, 2)
Out[1]: True

In [2]: galois.is_smooth(10, 5)
Out[2]: True

In [3]: galois.is_smooth(12, 5)
Out[3]: True

In [4]: galois.is_smooth(60**2, 5)
Out[4]: True
```

galois.is_powersmooth

galois.**is_powersmooth**(n, B)

Determines if the positive integer n is B-powersmooth.

An integer n with prime factorization $n = p_1^{e_1} \dots p_k^{e_k}$ is B-powersmooth if $p_i^{e_i} \leq B$ for $1 \leq i \leq k$.

Parameters

- **n** (*int*) – A positive integer.
- **B** (*int*) – The smoothness bound $B \geq 2$.

Returns `True` if n is B-powersmooth.

Return type bool

Examples

Comparison of B-smooth and B-powersmooth. Necessarily, any n that is B-powersmooth must be B-smooth.

```
In [1]: galois.is_smooth(2**4 * 3**2 * 5, 5)
Out[1]: True
```

(continues on next page)

(continued from previous page)

```
In [2]: galois.is_powersmooth(2**4 * 3**2 * 5, 5)
Out[2]: False
```

5.5 Primes

This section contains functions for generating primes and analyzing primality.

5.5.1 Prime number generation

primes(n)	Returns all primes p for $p \leq n$.
kth_prime(k)	Returns the k-th prime.
prev_prime(n)	Returns the nearest prime p, such that $p \leq n$.
next_prime(n)	Returns the nearest prime p, such that $p > n$.
random_prime(bits)	Returns a random prime p with b bits, such that $2^b \leq p < 2^{b+1}$.
mersenne_exponents([n])	Returns all known Mersenne exponents e for $e \leq n$.
mersenne_primes([n])	Returns all known Mersenne primes p for $p \leq 2^n - 1$.

galois.primes

galois.**primes**(n)

Returns all primes p for $p \leq n$.

Parameters **n** (*int*) – A positive integer n $ge2$.

Returns The primes up to and including n.

Return type list

References

- https://oeis.org/A000040

Examples

```
In [1]: galois.primes(19)
Out[1]: [2, 3, 5, 7, 11, 13, 17, 19]
```

galois.kth_prime

galois.**kth_prime**(k)

Returns the k-th prime.

Parameters **k** (*int*) – The prime index, where $k = \{1, 2, 3, 4, \dots\}$ for primes $p = \{2, 3, 5, 7, \dots\}$.

Returns The k-th prime.

Return type int

Examples

```
In [1]: galois.kth_prime(1)
Out[1]: 2

In [2]: galois.kth_prime(3)
Out[2]: 5

In [3]: galois.kth_prime(1000)
Out[3]: 7919
```

galois.prev_prime

galois.**prev_prime**(n)

Returns the nearest prime p, such that $p \leq n$.

Parameters **n** (*int*) – A positive integer.

Returns The nearest prime $p \leq n$.

Return type int

Examples

```
In [1]: galois.prev_prime(13)
Out[1]: 13

In [2]: galois.prev_prime(15)
Out[2]: 13
```

galois.next_prime

galois.**next_prime**(n)

Returns the nearest prime p, such that $p > n$.

Parameters **n** (*int*) – A positive integer.

Returns The nearest prime $p > n$.

Return type int

Examples

```
In [1]: galois.next_prime(13)
Out[1]: 17

In [2]: galois.next_prime(15)
Out[2]: 17
```

galois.random_prime

galois.**random_prime**(*bits*)

Returns a random prime p with b bits, such that $2^b \leq p < 2^{b+1}$.

This function randomly generates integers with b bits and uses the primality tests in *galois.is_prime()* to determine if p is prime.

Parameters **bits** (*int*) – The number of bits in the prime p.

Returns A random prime in $2^b \leq p < 2^{b+1}$.

Return type int

References

- https://en.wikipedia.org/wiki/Prime_number_theorem

Examples

Generate a random 1024-bit prime.

```
In [1]: p = galois.random_prime(1024); p
Out[1]:␣
↪28053687248676594568870438001520820940347363671398796472378500116931458441447003475566439883343311
↪
In [2]: galois.is_prime(p)
Out[2]: True
```

```
$ openssl prime␣
↪23686178792695738220699688608721459202975252407802639235893684447966742357083311612650692787877311
1514D68EDB7C650F1FF713531A1A43255A4BE6D66EE1FDBD96F4EB32757C1B1BAF16A5933E24D45FAD6C6A814F3C8C14F3C
↪(23686178792695738220699688608721459202975252407802639235893684447966742357083311612650692787877311
↪is prime
```

galois.mersenne_exponents

galois.**mersenne_exponents**(*n=None*)

Returns all known Mersenne exponents e for $e \leq n$.

A Mersenne exponent e is an exponent of 2 such that $2^e - 1$ is prime.

Parameters **n** (*int, optional*) – The max exponent of 2. The default is None which returns all known Mersenne exponents.

Returns The list of Mersenne exponents e for $e \leq n$.

Return type list

References

- https://oeis.org/A000043

Examples

```
# List all Mersenne exponents for Mersenne primes up to 2000 bits
In [1]: e = galois.mersenne_exponents(2000); e
Out[1]: [2, 3, 5, 7, 13, 17, 19, 31, 61, 89, 107, 127, 521, 607, 1279]

# Select one Merseene exponent and compute its Mersenne prime
In [2]: p = 2**e[-1] - 1; p
Out[2]:␣
↪10407932194664399081925240327364085538615262247266704805319112350403608059673360298012239441732321

In [3]: galois.is_prime(p)
Out[3]: True
```

galois.mersenne_primes

galois.**mersenne_primes**(*n=None*)

Returns all known Mersenne primes p for $p \leq 2^n - 1$.

Mersenne primes are primes that are one less than a power of 2.

Parameters **n** (*int, optional*) – The max power of 2. The default is None which returns all known Mersenne exponents.

Returns The list of known Mersenne primes p for $p \leq 2^n - 1$.

Return type list

References

- https://oeis.org/A000668

Examples

```
# List all Mersenne primes up to 2000 bits
In [1]: p = galois.mersenne_primes(2000); p
Out[1]:
[3,
 7,
 31,
 127,
 8191,
 131071,
 524287,
 2147483647,
 2305843009213693951,
 618970019642690137449562111,
 162259276829213363391578010288127,
 170141183460469231731687303715884105727,
⌴
↪686479766013060971498190079908139321726943530014330540939446345918554318339765605212255964066145
↪
⌴
↪531137992816767098689588206552468627329593117727031923199444138200403559860852242739162502265229
↪
⌴
↪104079321946643990819252403273640855386152622472667048053191123504036080596733602980122394417323

In [2]: galois.is_prime(p[-1])
Out[2]: True
```

5.5.2 Primality tests

is_prime(n)	Determines if n is prime.
is_prime_power(n)	Determines if n is a prime power $n = p^k$ for prime p and $k \geq 1$.
is_perfect_power(n)	Determines if n is a perfect power $n = x^k$ for $x > 0$ and $k \geq 2$.
is_composite(n)	Determines if n is composite.
is_square_free(n)	Determines if n is square-free, such that $n = p_1 p_2 \dots p_k$.
is_smooth(n, B)	Determines if the positive integer n is B-smooth.
is_powersmooth(n, B)	Determines if the positive integer n is B-powersmooth.

5.5.3 Specific primality tests

fermat_primality_test(n[, a, rounds])	Determines if n is composite using Fermat's primality test.
miller_rabin_primality_test(n[, a, rounds])	Determines if n is composite using the Miller-Rabin primality test.

galois.fermat_primality_test

galois.**fermat_primality_test**(*n*, *a=None*, *rounds=1*)

Determines if n is composite using Fermat's primality test.

Fermat's theorem says that for prime p and $1 \leq a \leq p-1$, the congruence $a^{p-1} \equiv 1 \pmod{p}$ holds. Fermat's primality test of n computes $a^{n-1} \bmod n$ for some $1 \leq a \leq n-1$. If a is such that $a^{p-1} \not\equiv 1 \pmod{p}$, then a is said to be a *Fermat witness* to the compositeness of n. If n is composite and $a^{p-1} \equiv 1 \pmod{p}$, then a is said to be a *Fermat liar* to the primality of n.

Since $a = \{1, n-1\}$ are Fermat liars for all composite n, it is common to reduce the range of possible a to $2 \leq a \leq n-2$.

Parameters

- **n** (*int*) – An odd integer $n \geq 3$.
- **a** (*int, optional*) – An integer in $2 \leq a \leq n-2$. The default is `None` which selects a random a.
- **rounds** (*int, optional*) – The number of iterations attempting to detect n as composite. Additional rounds will choose a new a. The default is 1.

Returns `False` if n is shown to be composite. `True` if n is probable prime.

Return type bool

References

- Section 4.2.1 from https://cacr.uwaterloo.ca/hac/about/chap4.pdf

Examples

Fermat's primality test will never mark a true prime as composite.

```
In [1]: primes = [257, 24841, 65497]

In [2]: [galois.is_prime(p) for p in primes]
Out[2]: [True, True, True]

In [3]: [galois.fermat_primality_test(p) for p in primes]
Out[3]: [True, True, True]
```

However, Fermat's primality test may mark a composite as probable prime. Here are pseudoprimes base 2 from A001567.

```
# List of some Fermat pseudoprimes to base 2
In [4]: pseudoprimes = [2047, 29341, 65281]

In [5]: [galois.is_prime(p) for p in pseudoprimes]
Out[5]: [False, False, False]

# The pseudoprimes base 2 satisfy 2^(p-1) = 1 (mod p)
In [6]: [galois.fermat_primality_test(p, a=2) for p in pseudoprimes]
Out[6]: [True, True, True]

# But they may not satisfy a^(p-1) = 1 (mod p) for other a
In [7]: [galois.fermat_primality_test(p) for p in pseudoprimes]
Out[7]: [False, True, False]
```

And the pseudoprimes base 3 from A005935.

```
# List of some Fermat pseudoprimes to base 3
In [8]: pseudoprimes = [2465, 7381, 16531]

In [9]: [galois.is_prime(p) for p in pseudoprimes]
Out[9]: [False, False, False]

# The pseudoprimes base 3 satisfy 3^(p-1) = 1 (mod p)
In [10]: [galois.fermat_primality_test(p, a=3) for p in pseudoprimes]
Out[10]: [True, True, True]

# But they may not satisfy a^(p-1) = 1 (mod p) for other a
In [11]: [galois.fermat_primality_test(p) for p in pseudoprimes]
Out[11]: [False, False, False]
```

galois.miller_rabin_primality_test

galois.**miller_rabin_primality_test**(*n*, *a=2*, *rounds=1*)

Determines if n is composite using the Miller-Rabin primality test.

The Miller-Rabin primality test is based on the fact that for odd n with factorization $n = 2^s r$ for odd r and integer a such that $\gcd(a, n) = 1$, then either $a^r \equiv 1 \pmod{n}$ or $a^{2^j r} \equiv -1 \pmod{n}$ for some j in $0 \leq j \leq s - 1$.

In the Miller-Rabin primality test, if $a^r \not\equiv 1 \pmod{n}$ and $a^{2^j r} \not\equiv -1 \pmod{n}$ for all j in $0 \leq j \leq s - 1$, then a is called a *strong witness* to the compositeness of n. If not, namely $a^r \equiv 1 \pmod{n}$ or $a^{2^j r} \equiv -1 \pmod{n}$ for any j in $0 \leq j \leq s - 1$, then a is called a *strong liar* to the primality of n and n is called a *strong pseudoprime to the base a*.

Since $a = \{1, n - 1\}$ are strong liars for all composite n, it is common to reduce the range of possible a to $2 \leq a \leq n - 2$.

For composite odd n, the probability that the Miller-Rabin test declares it a probable prime is less than $(\frac{1}{4})^t$, where t is the number of rounds, and is often much lower.

Parameters

- **n** (*int*) – An odd integer $n \geq 3$.
- **a** (*int, optional*) – An integer in $2 \leq a \leq n - 2$. The default is 2.

- **rounds** (*int, optional*) – The number of iterations attempting to detect n as composite. Additional rounds will choose consecutive primes for a.

Returns `False` if n is shown to be composite. `True` if n is probable prime.

Return type bool

References

- Section 4.2.3 from https://cacr.uwaterloo.ca/hac/about/chap4.pdf
- https://math.dartmouth.edu/~carlp/PDF/paper25.pdf

Examples

The Miller-Rabin primality test will never mark a true prime as composite.

```
In [1]: primes = [257, 24841, 65497]

In [2]: [galois.is_prime(p) for p in primes]
Out[2]: [True, True, True]

In [3]: [galois.miller_rabin_primality_test(p) for p in primes]
Out[3]: [True, True, True]
```

However, a composite n may have strong liars. 91 has $\{9, 10, 12, 16, 17, 22, 29, 38, 53, 62, 69, 74, 75, 79, 81, 82\}$ as strong liars.

```
In [4]: strong_liars = [9,10,12,16,17,22,29,38,53,62,69,74,75,79,81,82]

In [5]: witnesses = [a for a in range(2, 90) if a not in strong_liars]

# All strong liars falsely assert that 91 is prime
In [6]: [galois.miller_rabin_primality_test(91, a=a) for a in strong_liars] ==␣
↪[True,]*len(strong_liars)
Out[6]: True

# All other a are witnesses to the compositeness of 91
In [7]: [galois.miller_rabin_primality_test(91, a=a) for a in witnesses] == [False,
↪]*len(witnesses)
Out[7]: True
```

5.6 Forward Error Correcting Codes

This section contains classes and functions for constructing forward error correction codes.

5.6.1 FEC classes

BCH(n, k[, primitive_poly, ...])	Constructs a primitive, narrow-sense binary BCH(n, k) code.
ReedSolomon(n, k[, c, primitive_poly, ...])	Constructs a RS(n, k) code.

galois.BCH

class galois.**BCH**(*n*, *k*, *primitive_poly=None*, *primitive_element=None*, *systematic=True*)

Constructs a primitive, narrow-sense binary BCH(n, k) code.

A BCH(n, k) code is a $[n, k, d]_2$ linear block code.

To create the shortened BCH$(n - s, k - s)$ code, construct the full-sized BCH(n, k) code and then pass $k - s$ bits into *encode()* and $n - s$ bits into *decode()*. Shortened codes are only applicable for systematic codes.

Parameters

- **n** (*int*) – The codeword size n, must be $n = 2^m - 1$.
- **k** (*int*) – The message size k.
- **primitive_poly** (galois.Poly, *optional*) – Optionally specify the primitive polynomial that defines the extension field GF(2^m). The default is None which uses Matlab's default, see *galois.matlab_primitive_poly()*. Matlab tends to use the lexicographically-minimal primitive polynomial as a default instead of the Conway polynomial.
- **primitive_element** (*int*, galois.Poly, *optional*) – Optionally specify the primitive element α whose powers are roots of the generator polynomial $g(x)$. The default is None which uses the lexicographically-minimal primitive element in GF(2^m), i.e. galois.primitive_element(2, m).
- **systematic** (*bool*, *optional*) – Optionally specify if the encoding should be systematic, meaning the codeword is the message with parity appended. The default is True.

Examples

```
In [1]: galois.bch_valid_codes(15)
Out[1]: [(15, 11, 1), (15, 7, 2), (15, 5, 3)]

In [2]: bch = galois.BCH(15, 7)

In [3]: m = galois.GF2.Random(bch.k); m
Out[3]: GF([0, 1, 0, 0, 1, 0, 0], order=2)

In [4]: c = bch.encode(m); c
Out[4]: GF([0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0], order=2)

# Corrupt the first bit in the codeword
In [5]: c[0] ^= 1

In [6]: dec_m = bch.decode(c); dec_m
Out[6]: GF([0, 1, 0, 0, 1, 0, 0], order=2)

In [7]: np.array_equal(dec_m, m)
```

(continues on next page)

(continued from previous page)

```
Out[7]: True

# Instruct the decoder to return the number of corrected bit errors
In [8]: dec_m, N = bch.decode(c, errors=True); dec_m, N
Out[8]: (GF([0, 1, 0, 0, 1, 0, 0], order=2), 1)

In [9]: np.array_equal(dec_m, m)
Out[9]: True
```

Constructors

—

Methods

decode(codeword[, errors])	Decodes the BCH codeword $\mathbf{c}$ into the message $\mathbf{m}$.
detect(codeword)	Detects if errors are present in the BCH codeword $\mathbf{c}$.
encode(message[, parity_only])	Encodes the message $\mathbf{m}$ into the BCH codeword $\mathbf{c}$.

Attributes

G	The generator matrix $\mathbf{G}$ with shape (k, n).
H	The parity-check matrix $\mathbf{H}$ with shape $(2t, n)$.
d	The design distance d of the $[n, k, d]_2$ code.
field	The Galois field $\mathrm{GF}(2^m)$ that defines the BCH code.
generator_poly	The generator polynomial $g(x)$ whose roots are *BCH.roots*.
is_narrow_sense	Indicates if the BCH code is narrow sense, meaning the roots of the generator polynomial are consecutive powers of α starting at 1, i.e. $\alpha, \alpha^2, \ldots, \alpha^{2t-1}$.
is_primitive	Indicates if the BCH code is primitive, meaning $n = 2^m - 1$.
k	The message size k of the $[n, k, d]_2$ code
n	The codeword size n of the $[n, k, d]_2$ code
roots	The $2t$ roots of the generator polynomial.
systematic	Indicates if the code is configured to return codewords in systematic form.
t	The error-correcting capability of the code.

decode(*codeword*, *errors=False*)

Decodes the BCH codeword $\mathbf{c}$ into the message $\mathbf{m}$.

The codeword vector $\mathbf{c}$ is defined as $\mathbf{c} = [c_{n-1}, \ldots, c_1, c_0] \in \mathrm{GF}(2)^n$, which corresponds to the codeword polynomial $c(x) = c_{n-1}x^{n-1} + \cdots + c_1 x + c_0$. The message vector $\mathbf{m}$ is defined as $\mathbf{m} = [m_{k-1}, \ldots, m_1, m_0] \in \mathrm{GF}(2)^k$, which corresponds to the message polynomial $m(x) = m_{k-1}x^{k-1} + \cdots + m_1 x + m_0$.

In decoding, the syndrome vector s is computed by $\mathbf{s} = \mathbf{c}\mathbf{H}^T$, where $\mathbf{H}$ is the parity-check matrix. The equivalent polynomial operation is $s(x) = c(x) \bmod g(x)$. A syndrome of zeros indicates the received codeword is a valid codeword and there are no errors. If the syndrome is non-zero, the decoder will find an error-locator polynomial $\sigma(x)$ and the corresponding error locations and values.

For the shortened BCH$(n-s, k-s)$ code (only applicable for systematic codes), pass $n-s$ bits into *decode()* to return the $k-s$-bit message.

Parameters

- **codeword** (*numpy.ndarray*, galois.FieldArray) – The codeword as either a n-length vector or (N, n) matrix, where N is the number of codewords. For systematic codes, codeword lengths less than n may be provided for shortened codewords.
- **errors** (*bool*, *optional*) – Optionally specify whether to return the nubmer of corrected errors.

Returns

- *numpy.ndarray, galois.FieldArray* – The decoded message as either a k-length vector or (N, k) matrix.
- *int, np.ndarray* – Optional return argument of the number of corrected bit errors as either a scalar or n-length vector. Valid number of corrections are in $[0, t]$. If a codeword has too many errors and cannot be corrected, -1 will be returned.

Examples

Decode a single codeword.

```
In [1]: bch = galois.BCH(15, 7)

In [2]: m = galois.GF2.Random(bch.k); m
Out[2]: GF([0, 1, 1, 1, 0, 1, 1], order=2)

In [3]: c = bch.encode(m); c
Out[3]: GF([0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1], order=2)

# Corrupt the first bit in the codeword
In [4]: c[0] ^= 1

In [5]: dec_m = bch.decode(c); dec_m
Out[5]: GF([0, 1, 1, 1, 0, 1, 1], order=2)

In [6]: np.array_equal(dec_m, m)
Out[6]: True

# Instruct the decoder to return the number of corrected bit errors
In [7]: dec_m, N = bch.decode(c, errors=True); dec_m, N
Out[7]: (GF([0, 1, 1, 1, 0, 1, 1], order=2), 1)

In [8]: np.array_equal(dec_m, m)
Out[8]: True
```

Decode a single, shortened codeword.

```
In [9]: m = galois.GF2.Random(bch.k - 3); m
Out[9]: GF([0, 0, 0, 1], order=2)

In [10]: c = bch.encode(m); c
Out[10]: GF([0, 0, 0, 1, 1, 1, 0, 1, 0, 0, 0, 1], order=2)

# Corrupt the first bit in the codeword
In [11]: c[0] ^= 1

In [12]: dec_m = bch.decode(c); dec_m
Out[12]: GF([0, 0, 0, 1], order=2)

In [13]: np.array_equal(dec_m, m)
Out[13]: True
```

Decode a matrix of codewords.

```
In [14]: m = galois.GF2.Random((5, bch.k)); m
Out[14]:
GF([[1, 1, 1, 0, 1, 0, 1],
    [1, 1, 0, 0, 0, 0, 1],
    [0, 0, 1, 1, 1, 0, 1],
    [1, 0, 1, 0, 0, 1, 0],
    [1, 1, 1, 0, 0, 0, 1]], order=2)

In [15]: c = bch.encode(m); c
Out[15]:
GF([[1, 1, 1, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 1],
    [1, 1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 1, 1, 0, 1],
    [0, 0, 1, 1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0],
    [1, 0, 1, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 1],
    [1, 1, 1, 0, 0, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1]], order=2)

# Corrupt the first bit in each codeword
In [16]: c[:,0] ^= 1

In [17]: dec_m = bch.decode(c); dec_m
Out[17]:
GF([[1, 1, 1, 0, 1, 0, 1],
    [1, 1, 0, 0, 0, 0, 1],
    [0, 0, 1, 1, 1, 0, 1],
    [1, 0, 1, 0, 0, 1, 0],
    [1, 1, 1, 0, 0, 0, 1]], order=2)

In [18]: np.array_equal(dec_m, m)
Out[18]: True

# Instruct the decoder to return the number of corrected bit errors
In [19]: dec_m, N = bch.decode(c, errors=True); dec_m, N
Out[19]:
(GF([[1, 1, 1, 0, 1, 0, 1],
     [1, 1, 0, 0, 0, 0, 1],
     [0, 0, 1, 1, 1, 0, 1],
```

(continues on next page)

(continued from previous page)

```
    [1, 0, 1, 0, 0, 1, 0],
    [1, 1, 1, 0, 0, 0, 1]], order=2),
 array([1, 1, 1, 1, 1]))

In [20]: np.array_equal(dec_m, m)
Out[20]: True
```

detect(*codeword*)

Detects if errors are present in the BCH codeword $\mathbf{c}$.

The $[n, k, d]_2$ BCH code has $d_{min} \geq d$ minimum distance. It can detect up to $d_{min} - 1$ errors.

Parameters **codeword** (*numpy.ndarray*, galois.FieldArray) – The codeword as either a n-length vector or (N, n) matrix, where N is the number of codewords. For systematic codes, codeword lengths less than n may be provided for shortened codewords.

Returns A boolean scalar or array indicating if errors were detected in the corresponding codeword `True` or not `False`.

Return type bool, numpy.ndarray

Examples

Detect errors in a valid codeword.

```
In [1]: bch = galois.BCH(15, 7)

# The minimum distance of the code
In [2]: bch.d
Out[2]: 5

In [3]: m = galois.GF2.Random(bch.k); m
Out[3]: GF([0, 1, 0, 0, 1, 0, 1], order=2)

In [4]: c = bch.encode(m); c
Out[4]: GF([0, 1, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 1, 1], order=2)

In [5]: bch.detect(c)
Out[5]: False
```

Detect $d_{min} - 1$ errors in a received codeword.

```
# Corrupt the first `d - 1` bits in the codeword
In [6]: c[0:bch.d - 1] ^= 1

In [7]: bch.detect(c)
Out[7]: True
```

encode(*message*, *parity_only=False*)

Encodes the message $\mathbf{m}$ into the BCH codeword $\mathbf{c}$.

The message vector $\mathbf{m}$ is defined as $\mathbf{m} = [m_{k-1}, \ldots, m_1, m_0] \in \mathrm{GF}(2)^k$, which corresponds to the message polynomial $m(x) = m_{k-1}x^{k-1} + \cdots + m_1 x + m_0$. The codeword vector $\mathbf{c}$ is defined as $\mathbf{c} =$

$[c_{n-1}, \ldots, c_1, c_0] \in \mathrm{GF}(2)^n$, which corresponds to the codeword polynomial $c(x) = c_{n-1}x^{n-1} + \cdots + c_1 x + c_0$.

The codeword vector is computed from the message vector by $\mathbf{c} = \mathbf{m}\mathbf{G}$, where $\mathbf{G}$ is the generator matrix. The equivalent polynomial operation is $c(x) = m(x)g(x)$. For systematic codes, $\mathbf{G} = [\mathbf{I} \mid \mathbf{P}]$ such that $\mathbf{c} = [\mathbf{m} \mid \mathbf{p}]$. And in polynomial form, $p(x) = -(m(x)x^{n-k} \bmod g(x))$ with $c(x) = m(x)x^{n-k} + p(x)$. For systematic and non-systematic codes, each codeword is a multiple of the generator polynomial, i.e. $g(x) \mid c(x)$.

For the shortened BCH$(n-s, k-s)$ code (only applicable for systematic codes), pass $k-s$ bits into *encode()* to return the $n-s$-bit codeword.

Parameters

- **message** (*numpy.ndarray*, *galois.FieldArray*) – The message as either a k-length vector or (N, k) matrix, where N is the number of messages. For systematic codes, message lengths less than k may be provided to produce shortened codewords.
- **parity_only** (*bool*, *optional*) – Optionally specify whether to return only the parity bits. This only applies to systematic codes. The default is `False`.

Returns The codeword as either a n-length vector or (N, n) matrix. The return type matches the message type. If `parity_only=True`, the parity bits are returned as either a $n-k$-length vector or $(N, n-k)$ matrix.

Return type numpy.ndarray, *galois.FieldArray*

Examples

Encode a single codeword.

```
In [1]: bch = galois.BCH(15, 7)

In [2]: m = galois.GF2.Random(bch.k); m
Out[2]: GF([0, 0, 0, 0, 0, 0, 0], order=2)

In [3]: c = bch.encode(m); c
Out[3]: GF([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], order=2)

In [4]: p = bch.encode(m, parity_only=True); p
Out[4]: GF([0, 0, 0, 0, 0, 0, 0, 0], order=2)
```

Encode a single, shortened codeword.

```
In [5]: m = galois.GF2.Random(bch.k - 3); m
Out[5]: GF([0, 0, 0, 1], order=2)

In [6]: c = bch.encode(m); c
Out[6]: GF([0, 0, 0, 1, 1, 1, 0, 1, 0, 0, 0, 1], order=2)
```

Encode a matrix of codewords.

```
In [7]: m = galois.GF2.Random((5, bch.k)); m
Out[7]:
GF([[1, 1, 1, 1, 0, 1, 0],
    [1, 1, 0, 0, 0, 0, 0],
    [0, 1, 0, 0, 0, 0, 1],
```

(continues on next page)

(continued from previous page)

```
    [0, 0, 1, 0, 0, 1, 0],
    [0, 0, 0, 1, 0, 0, 0]], order=2)

In [8]: c = bch.encode(m); c
Out[8]:
GF([[1, 1, 1, 1, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0],
    [1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 1, 0, 0],
    [0, 1, 0, 0, 0, 0, 1, 1, 0, 1, 0, 0, 1, 0, 1],
    [0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1],
    [0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1]], order=2)

In [9]: p = bch.encode(m, parity_only=True); p
Out[9]:
GF([[1, 1, 0, 0, 1, 0, 0, 0],
    [1, 0, 0, 1, 1, 1, 0, 0],
    [1, 0, 1, 0, 0, 1, 0, 1],
    [0, 1, 0, 0, 1, 0, 0, 1],
    [0, 0, 0, 1, 1, 1, 0, 1]], order=2)
```

property G

The generator matrix $\mathbf{G}$ with shape (k, n).

Type *galois.GF2*

property H

The parity-check matrix $\mathbf{H}$ with shape $(2t, n)$.

Type *galois.FieldArray*

property d

The design distance d of the $[n, k, d]_2$ code. The minimum distance of a BCH code may be greater than the design distance, $d_{min} \geq d$.

Type int

property field

The Galois field $\mathrm{GF}(2^m)$ that defines the BCH code.

Type *galois.FieldClass*

property generator_poly

The generator polynomial $g(x)$ whose roots are *BCH.roots*.

Type *galois.Poly*

property is_narrow_sense

Indicates if the BCH code is narrow sense, meaning the roots of the generator polynomial are consecutive powers of α starting at 1, i.e. $\alpha, \alpha^2, \ldots, \alpha^{2t-1}$.

Type bool

property is_primitive

Indicates if the BCH code is primitive, meaning $n = 2^m - 1$.

Type bool

property k

The message size k of the $[n, k, d]_2$ code

Type int

property n
The codeword size n of the $[n, k, d]_2$ code

Type int

property roots
The $2t$ roots of the generator polynomial. These are consecutive powers of α.

Type *galois.FieldArray*

property systematic
Indicates if the code is configured to return codewords in systematic form.

Type bool

property t
The error-correcting capability of the code. The code can correct t bit errors in a codeword.

Type int

galois.ReedSolomon

class galois.**ReedSolomon**(*n*, *k*, *c=1*, *primitive_poly=None*, *primitive_element=None*, *systematic=True*)
Constructs a RS(n, k) code.

A RS(n, k) code is a $[n, k, d]_q$ linear block code.

To create the shortened RS$(n-s, k-s)$ code, construct the full-sized RS(n, k) code and then pass $k-s$ symbols into *encode()* and $n-s$ symbols into *decode()*. Shortened codes are only applicable for systematic codes.

Parameters

- **n** (*int*) – The codeword size n, must be $n = q - 1$.
- **k** (*int*) – The message size k. The error-correcting capability t is defined by $n - k = 2t$.
- **c** (*int, optional*) – The first consecutive power of α. The default is 1.
- **primitive_poly** (galois.Poly, *optional*) – Optionally specify the primitive polynomial that defines the extension field GF(q). The default is `None` which uses Matlab's default, see *galois.matlab_primitive_poly()*. Matlab tends to use the lexicographically-minimal primitive polynomial as a default instead of the Conway polynomial.
- **primitive_element** (*int*, galois.Poly, *optional*) – Optionally specify the primitive element α of GF(q) whose powers are roots of the generator polynomial $g(x)$. The default is `None` which uses the lexicographically-minimal primitive element in GF(q), i.e. `galois.primitive_element(p, m)`.
- **systematic** (*bool, optional*) – Optionally specify if the encoding should be systematic, meaning the codeword is the message with parity appended. The default is `True`.

Examples

```
In [1]: rs = galois.ReedSolomon(15, 9)

In [2]: GF = rs.field

In [3]: m = GF.Random(rs.k); m
Out[3]: GF([ 7,  6, 15,  1,  6, 10, 14,  4,  1], order=2^4)
```

(continues on next page)

(continued from previous page)

```
In [4]: c = rs.encode(m); c
Out[4]:
GF([ 7,  6, 15,  1,  6, 10, 14,  4,  1,  8,  6,  5, 11,  2,  5],
   order=2^4)

# Corrupt the first symbol in the codeword
In [5]: c[0] ^= 13

In [6]: dec_m = rs.decode(c); dec_m
Out[6]: GF([ 7,  6, 15,  1,  6, 10, 14,  4,  1], order=2^4)

In [7]: np.array_equal(dec_m, m)
Out[7]: True

# Instruct the decoder to return the number of corrected symbol errors
In [8]: dec_m, N = rs.decode(c, errors=True); dec_m, N
Out[8]: (GF([ 7,  6, 15,  1,  6, 10, 14,  4,  1], order=2^4), 1)

In [9]: np.array_equal(dec_m, m)
Out[9]: True
```

Constructors

—

Methods

decode(codeword[, errors])	Decodes the Reed-Solomon codeword $\mathbf{c}$ into the message $\mathbf{m}$.
detect(codeword)	Detects if errors are present in the Reed-Solomon codeword $\mathbf{c}$.
encode(message[, parity_only])	Encodes the message $\mathbf{m}$ into the Reed-Solomon codeword $\mathbf{c}$.

Attributes

G	The generator matrix $\mathbf{G}$ with shape (k, n).
H	The parity-check matrix $\mathbf{H}$ with shape $(2t, n)$.
c	The degree of the first consecutive root.
d	The design distance d of the $[n, k, d]_q$ code.
field	The Galois field $\mathrm{GF}(q)$ that defines the Reed-Solomon code.
generator_poly	The generator polynomial $g(x)$ whose roots are *ReedSolomon.roots*.

continues on next page

Table 42 – continued from previous page

is_narrow_sense	Indicates if the Reed-Solomon code is narrow sense, meaning the roots of the generator polynomial are consecutive powers of α starting at 1, i.e. $\alpha, \alpha^2, \ldots, \alpha^{2t-1}$.
k	The message size k of the $[n, k, d]_q$ code.
n	The codeword size n of the $[n, k, d]_q$ code.
roots	The roots of the generator polynomial.
systematic	Indicates if the code is configured to return codewords in systematic form.
t	The error-correcting capability of the code.

decode(*codeword*, *errors=False*)

Decodes the Reed-Solomon codeword $\mathbf{c}$ into the message $\mathbf{m}$.

The codeword vector $\mathbf{c}$ is defined as $\mathbf{c} = [c_{n-1}, \ldots, c_1, c_0] \in \mathrm{GF}(q)^n$, which corresponds to the codeword polynomial $c(x) = c_{n-1}x^{n-1} + \cdots + c_1 x + c_0$. The message vector $\mathbf{m}$ is defined as $\mathbf{m} = [m_{k-1}, \ldots, m_1, m_0] \in \mathrm{GF}(q)^k$, which corresponds to the message polynomial $m(x) = m_{k-1}x^{k-1} + \cdots + m_1 x + m_0$.

In decoding, the syndrome vector s is computed by $\mathbf{s} = \mathbf{c}\mathbf{H}^T$, where $\mathbf{H}$ is the parity-check matrix. The equivalent polynomial operation is $s(x) = c(x) \bmod g(x)$. A syndrome of zeros indicates the received codeword is a valid codeword and there are no errors. If the syndrome is non-zero, the decoder will find an error-locator polynomial $\sigma(x)$ and the corresponding error locations and values.

For the shortened RS$(n - s, k - s)$ code (only applicable for systematic codes), pass $n - s$ symbols into *decode()* to return the $k - s$-symbol message.

Parameters

- **codeword** (*numpy.ndarray*, *galois.FieldArray*) – The codeword as either a n-length vector or (N, n) matrix, where N is the number of codewords. For systematic codes, codeword lengths less than n may be provided for shortened codewords.
- **errors** (*bool*, *optional*) – Optionally specify whether to return the nubmer of corrected errors.

Returns

- *numpy.ndarray, galois.FieldArray* – The decoded message as either a k-length vector or (N, k) matrix.
- *int, np.ndarray* – Optional return argument of the number of corrected symbol errors as either a scalar or n-length vector. Valid number of corrections are in $[0, t]$. If a codeword has too many errors and cannot be corrected, -1 will be returned.

Examples

Decode a single codeword.

```
In [1]: rs = galois.ReedSolomon(15, 9)

In [2]: GF = rs.field

In [3]: m = GF.Random(rs.k); m
Out[3]: GF([ 4, 12,  5,  5,  5,  2,  1,  4, 14], order=2^4)
```

(continues on next page)

(continued from previous page)

```
In [4]: c = rs.encode(m); c
Out[4]:
GF([ 4, 12,  5,  5,  5,  2,  1,  4, 14, 10,  4, 13,  0,  4, 15],
   order=2^4)

# Corrupt the first symbol in the codeword
In [5]: c[0] += GF(13)

In [6]: dec_m = rs.decode(c); dec_m
Out[6]: GF([ 4, 12,  5,  5,  5,  2,  1,  4, 14], order=2^4)

In [7]: np.array_equal(dec_m, m)
Out[7]: True

# Instruct the decoder to return the number of corrected symbol errors
In [8]: dec_m, N = rs.decode(c, errors=True); dec_m, N
Out[8]: (GF([ 4, 12,  5,  5,  5,  2,  1,  4, 14], order=2^4), 1)

In [9]: np.array_equal(dec_m, m)
Out[9]: True
```

Decode a single, shortened codeword.

```
In [10]: m = GF.Random(rs.k - 4); m
Out[10]: GF([ 0, 14,  0,  4,  8], order=2^4)

In [11]: c = rs.encode(m); c
Out[11]: GF([ 0, 14,  0,  4,  8,  7, 14,  2, 15,  3, 10], order=2^4)

# Corrupt the first symbol in the codeword
In [12]: c[0] += GF(13)

In [13]: dec_m = rs.decode(c); dec_m
Out[13]: GF([ 0, 14,  0,  4,  8], order=2^4)

In [14]: np.array_equal(dec_m, m)
Out[14]: True
```

Decode a matrix of codewords.

```
In [15]: m = GF.Random((5, rs.k)); m
Out[15]:
GF([[ 6,  3,  6,  8,  0,  3,  1, 12,  9],
    [14,  1,  1,  6,  8,  9,  9,  3,  6],
    [ 1, 15,  3,  7,  1,  6,  1,  2,  1],
    [15, 10, 15,  4,  4, 15,  7,  4, 14],
    [ 2,  6, 15, 11,  5, 12,  3,  2,  5]], order=2^4)

In [16]: c = rs.encode(m); c
Out[16]:
GF([[ 6,  3,  6,  8,  0,  3,  1, 12,  9,  0, 11,  2,  3,  7,  6],
    [14,  1,  1,  6,  8,  9,  9,  3,  6, 10,  3, 15, 10,  9,  6],
```

(continues on next page)

(continued from previous page)

```
    [ 1, 15,  3,  7,  1,  6,  1,  2,  1,  1, 10,  0, 13,  7,  6],
    [15, 10, 15,  4,  4, 15,  7,  4, 14, 14, 15,  2,  9,  4, 14],
    [ 2,  6, 15, 11,  5, 12,  3,  2,  5, 11, 11, 11, 10, 15,  4]],
   order=2^4)

# Corrupt the first symbol in each codeword
In [17]: c[:,0] += GF(13)

In [18]: dec_m = rs.decode(c); dec_m
Out[18]:
GF([[ 6,  3,  6,  8,  0,  3,  1, 12,  9],
    [14,  1,  1,  6,  8,  9,  9,  3,  6],
    [ 1, 15,  3,  7,  1,  6,  1,  2,  1],
    [15, 10, 15,  4,  4, 15,  7,  4, 14],
    [ 2,  6, 15, 11,  5, 12,  3,  2,  5]], order=2^4)

In [19]: np.array_equal(dec_m, m)
Out[19]: True

# Instruct the decoder to return the number of corrected symbol errors
In [20]: dec_m, N = rs.decode(c, errors=True); dec_m, N
Out[20]:
(GF([[ 6,  3,  6,  8,  0,  3,  1, 12,  9],
     [14,  1,  1,  6,  8,  9,  9,  3,  6],
     [ 1, 15,  3,  7,  1,  6,  1,  2,  1],
     [15, 10, 15,  4,  4, 15,  7,  4, 14],
     [ 2,  6, 15, 11,  5, 12,  3,  2,  5]], order=2^4),
 array([1, 1, 1, 1, 1]))

In [21]: np.array_equal(dec_m, m)
Out[21]: True
```

---

detect(*codeword*)

Detects if errors are present in the Reed-Solomon codeword $\mathbf{c}$.

The $[n, k, d]_q$ Reed-Solomon code has $d_{min} = d$ minimum distance. It can detect up to $d_{min} - 1$ errors.

Parameters **codeword** (*numpy.ndarray*, galois.FieldArray) – The codeword as either a n-length vector or (N, n) matrix, where N is the number of codewords. For systematic codes, codeword lengths less than n may be provided for shortened codewords.

Returns A boolean scalar or array indicating if errors were detected in the corresponding codeword `True` or not `False`.

Return type bool, numpy.ndarray

Examples

Detect errors in a valid codeword.

```
In [1]: rs = galois.ReedSolomon(15, 9)

In [2]: GF = rs.field
```

(continues on next page)

(continued from previous page)

```
# The minimum distance of the code
In [3]: rs.d
Out[3]: 7

In [4]: m = GF.Random(rs.k); m
Out[4]: GF([13,  1,  5,  1, 10,  8, 13, 11,  8], order=2^4)

In [5]: c = rs.encode(m); c
Out[5]:
GF([13,  1,  5,  1, 10,  8, 13, 11,  8,  2,  7, 15,  1,  3, 15],
   order=2^4)

In [6]: rs.detect(c)
Out[6]: False
```

Detect $d_{min} - 1$ errors in a received codeword.

```
# Corrupt the first `d - 1` symbols in the codeword
In [7]: c[0:rs.d - 1] += GF(13)

In [8]: rs.detect(c)
Out[8]: True
```

---

encode(*message*, *parity_only=False*)

Encodes the message $\mathbf{m}$ into the Reed-Solomon codeword $\mathbf{c}$.

The message vector $\mathbf{m}$ is defined as $\mathbf{m} = [m_{k-1}, \ldots, m_1, m_0] \in \mathrm{GF}(q)^k$, which corresponds to the message polynomial $m(x) = m_{k-1}x^{k-1} + \cdots + m_1 x + m_0$. The codeword vector $\mathbf{c}$ is defined as $\mathbf{c} = [c_{n-1}, \ldots, c_1, c_0] \in \mathrm{GF}(q)^n$, which corresponds to the codeword polynomial $c(x) = c_{n-1}x^{n-1} + \cdots + c_1 x + c_0$.

The codeword vector is computed from the message vector by $\mathbf{c} = \mathbf{m}\mathbf{G}$, where $\mathbf{G}$ is the generator matrix. The equivalent polynomial operation is $c(x) = m(x)g(x)$. For systematic codes, $\mathbf{G} = [\mathbf{I} \mid \mathbf{P}]$ such that $\mathbf{c} = [\mathbf{m} \mid \mathbf{p}]$. And in polynomial form, $p(x) = -(m(x)x^{n-k} \bmod g(x))$ with $c(x) = m(x)x^{n-k} + p(x)$. For systematic and non-systematic codes, each codeword is a multiple of the generator polynomial, i.e. $g(x) \mid c(x)$.

For the shortened RS$(n - s, k - s)$ code (only applicable for systematic codes), pass $k - s$ symbols into *encode()* to return the $n - s$-symbol codeword.

Parameters

- **message** (*numpy.ndarray*, *galois.FieldArray*) – The message as either a k-length vector or (N, k) matrix, where N is the number of messages. For systematic codes, message lengths less than k may be provided to produce shortened codewords.
- **parity_only** (*bool*, *optional*) – Optionally specify whether to return only the parity symbols. This only applies to systematic codes. The default is `False`.

Returns The codeword as either a n-length vector or (N, n) matrix. The return type matches the message type. If `parity_only=True`, the parity symbols are returned as either a $n-k$-length vector or $(N, n - k)$ matrix.

Return type numpy.ndarray, *galois.FieldArray*

Examples

Encode a single codeword.

```
In [1]: rs = galois.ReedSolomon(15, 9)

In [2]: GF = rs.field

In [3]: m = GF.Random(rs.k); m
Out[3]: GF([ 0, 10, 14, 10,  3,  4, 14,  3, 13], order=2^4)

In [4]: c = rs.encode(m); c
Out[4]:
GF([ 0, 10, 14, 10,  3,  4, 14,  3, 13,  3, 13,  8,  4,  5, 10],
   order=2^4)

In [5]: p = rs.encode(m, parity_only=True); p
Out[5]: GF([ 3, 13,  8,  4,  5, 10], order=2^4)
```

Encode a single, shortened codeword.

```
In [6]: m = GF.Random(rs.k - 4); m
Out[6]: GF([ 7,  1,  7,  7, 11], order=2^4)

In [7]: c = rs.encode(m); c
Out[7]: GF([ 7,  1,  7,  7, 11, 13, 10,  7,  2,  1, 15], order=2^4)
```

Encode a matrix of codewords.

```
In [8]: m = GF.Random((5, rs.k)); m
Out[8]:
GF([[ 5, 13,  7, 13,  7,  1,  5,  0, 10],
    [ 6, 14,  2, 15, 13, 14,  6,  0, 14],
    [ 2,  7,  2, 13, 13,  6, 14,  5, 12],
    [10,  7, 10,  3,  2,  0,  0,  4,  3],
    [ 9,  4,  7, 12,  9,  5,  8,  4, 11]], order=2^4)

In [9]: c = rs.encode(m); c
Out[9]:
GF([[ 5, 13,  7, 13,  7,  1,  5,  0, 10,  6,  3,  6, 15,  4, 13],
    [ 6, 14,  2, 15, 13, 14,  6,  0, 14, 12,  7,  1, 13, 13, 12],
    [ 2,  7,  2, 13, 13,  6, 14,  5, 12, 15, 12, 12,  3,  3,  1],
    [10,  7, 10,  3,  2,  0,  0,  4,  3,  2,  3, 10,  6, 15, 14],
    [ 9,  4,  7, 12,  9,  5,  8,  4, 11, 11, 14, 14,  4,  2,  4]],
   order=2^4)

In [10]: p = rs.encode(m, parity_only=True); p
Out[10]:
GF([[ 6,  3,  6, 15,  4, 13],
    [12,  7,  1, 13, 13, 12],
    [15, 12, 12,  3,  3,  1],
    [ 2,  3, 10,  6, 15, 14],
    [11, 14, 14,  4,  2,  4]], order=2^4)
```

property G
The generator matrix $\mathbf{G}$ with shape (k, n).

Type *galois.FieldArray*

property H
The parity-check matrix $\mathbf{H}$ with shape $(2t, n)$.

Type *galois.FieldArray*

property c
The degree of the first consecutive root.

Type int

property d
The design distance d of the $[n, k, d]_q$ code. The minimum distance of a Reed-Solomon code is exactly equal to the design distance, $d_{min} = d$.

Type int

property field
The Galois field $\mathrm{GF}(q)$ that defines the Reed-Solomon code.

Type *galois.FieldClass*

property generator_poly
The generator polynomial $g(x)$ whose roots are `ReedSolomon.roots`.

Type *galois.Poly*

property is_narrow_sense
Indicates if the Reed-Solomon code is narrow sense, meaning the roots of the generator polynomial are consecutive powers of α starting at 1, i.e. $\alpha, \alpha^2, \ldots, \alpha^{2t-1}$.

Type bool

property k
The message size k of the $[n, k, d]_q$ code.

Type int

property n
The codeword size n of the $[n, k, d]_q$ code.

Type int

property roots
The roots of the generator polynomial. These are consecutive powers of α.

Type *galois.FieldArray*

property systematic
Indicates if the code is configured to return codewords in systematic form.

Type bool

property t
The error-correcting capability of the code. The code can correct t symbol errors in a codeword.

Type int

5.6.2 Linear block code functions

generator_to_parity_check_matrix(G)	Converts the generator matrix $\mathbf{G}$ of a linear $[n, k]$ code into its parity-check matrix $\mathbf{H}$.
parity_check_to_generator_matrix(H)	Converts the parity-check matrix $\mathbf{H}$ of a linear $[n, k]$ code into its generator matrix $\mathbf{G}$.

galois.generator_to_parity_check_matrix

galois.**generator_to_parity_check_matrix**(*G*)

Converts the generator matrix $\mathbf{G}$ of a linear $[n, k]$ code into its parity-check matrix $\mathbf{H}$.

The generator and parity-check matrices satisfy the equations $\mathbf{G}\mathbf{H}^T = \mathbf{0}$.

Parameters **G** (*galois.FieldArray*) – The (k, n) generator matrix $\mathbf{G}$ in systematic form $\mathbf{G} = [\mathbf{I}_{k,k} \mid \mathbf{P}_{k,n-k}]$.

Returns The $(n-k, n)$ parity-check matrix $\mathbf{H} = [-\mathbf{P}_{k,n-k}^T \mid \mathbf{I}_{n-k,n-k}]$.

Return type *galois.FieldArray*

galois.parity_check_to_generator_matrix

galois.**parity_check_to_generator_matrix**(*H*)

Converts the parity-check matrix $\mathbf{H}$ of a linear $[n, k]$ code into its generator matrix $\mathbf{G}$.

The generator and parity-check matrices satisfy the equations $\mathbf{G}\mathbf{H}^T = \mathbf{0}$.

Parameters **H** (*galois.FieldArray*) – The $(n-k, n)$ parity-check matrix $\mathbf{G}$ in systematic form $\mathbf{H} = [-\mathbf{P}_{k,n-k}^T \mid \mathbf{I}_{n-k,n-k}]$.

Returns The (k, n) generator matrix $\mathbf{G} = [\mathbf{I}_{k,k} \mid \mathbf{P}_{k,n-k}]$.

Return type *galois.FieldArray*

5.6.3 Cyclic code functions

bch_valid_codes(n[, t_min])	Returns a list of (n, k, t) tuples of valid primitive binary BCH codes.
poly_to_generator_matrix(n, generator_poly)	Converts the generator polynomial $g(x)$ into the generator matrix $\mathbf{G}$ for an $[n, k]$ cyclic code.
roots_to_parity_check_matrix(n, roots)	Converts the generator polynomial roots into the parity-check matrix $\mathbf{H}$ for an $[n, k]$ cyclic code.

galois.bch_valid_codes

galois.**bch_valid_codes**(*n*, *t_min=1*)

Returns a list of (n, k, t) tuples of valid primitive binary BCH codes.

A BCH code with parameters (n, k, t) is represented as a $[n, k, d]_2$ linear block code with $d = 2t + 1$.

Parameters

- **n** (*int*) – The codeword size n, must be $n = 2^m - 1$.
- **t_min** (*int, optional*) – The minimum error-correcting capability. The default is 1.

Returns A list of (n, k, t) tuples of valid primitive BCH codes.

Return type list

Examples

```
In [1]: galois.bch_valid_codes(31)
Out[1]: [(31, 26, 1), (31, 21, 2), (31, 16, 3), (31, 11, 5), (31, 6, 7)]

In [2]: galois.bch_valid_codes(31, t_min=3)
Out[2]: [(31, 16, 3), (31, 11, 5), (31, 6, 7)]
```

galois.poly_to_generator_matrix

galois.**poly_to_generator_matrix**(*n*, *generator_poly*, *systematic=True*)

Converts the generator polynomial $g(x)$ into the generator matrix $\mathbf{G}$ for an $[n, k]$ cyclic code.

Parameters

- **n** (*int*) – The codeword size n.
- **generator_poly** (galois.Poly) – The generator polynomial $g(x)$.
- **systematic** (*bool, optional*) – Optionally specify if the encoding should be systematic, meaning the codeword is the message with parity appended. The default is `True`.

Returns The (k, n) generator matrix $\mathbf{G}$, such that given a message $\mathbf{m}$, a codeword is defined by $\mathbf{c} = \mathbf{m}\mathbf{G}$.

Return type *galois.FieldArray*

Examples

Compute the generator matrix for the $\text{Hamming}(7, 4)$ code.

```
In [1]: g = galois.primitive_poly(2, 3); g
Out[1]: Poly(x^3 + x + 1, GF(2))

In [2]: galois.poly_to_generator_matrix(7, g, systematic=False)
Out[2]:
GF([[1, 0, 1, 1, 0, 0, 0],
    [0, 1, 0, 1, 1, 0, 0],
    [0, 0, 1, 0, 1, 1, 0],
    [0, 0, 0, 1, 0, 1, 1]], order=2)
```

(continues on next page)

(continued from previous page)

```
In [3]: galois.poly_to_generator_matrix(7, g, systematic=True)
Out[3]:
GF([[1, 0, 0, 0, 1, 0, 1],
    [0, 1, 0, 0, 1, 1, 1],
    [0, 0, 1, 0, 1, 1, 0],
    [0, 0, 0, 1, 0, 1, 1]], order=2)
```

galois.roots_to_parity_check_matrix

galois.**roots_to_parity_check_matrix**(*n*, *roots*)

Converts the generator polynomial roots into the parity-check matrix $\mathbf{H}$ for an $[n, k]$ cyclic code.

Parameters

- **n** (*int*) – The codeword size n.
- **roots** (galois.FieldArray) – The $2t$ roots of the generator polynomial $g(x)$.

Returns The $(2t, n)$ parity-check matrix $\mathbf{H}$, such that given a codeword $\mathbf{c}$, the syndrome is defined by $\mathbf{s} = \mathbf{c}\mathbf{H}^T$.

Return type *galois.FieldArray*

Examples

Compute the parity-check for the $\mathrm{RS}(15, 9)$ code.

```
In [1]: GF = galois.GF(2**4)

In [2]: alpha = GF.primitive_element

In [3]: t = 3

In [4]: roots = alpha**np.arange(1, 2*t + 1); roots
Out[4]: GF([ 2,  4,  8,  3,  6, 12], order=2^4)

In [5]: g = galois.Poly.Roots(roots); g
Out[5]: Poly(x^6 + 7x^5 + 9x^4 + 3x^3 + 12x^2 + 10x + 12, GF(2^4))

In [6]: galois.roots_to_parity_check_matrix(15, roots)
Out[6]:
GF([[ 9, 13, 15, 14,  7, 10,  5, 11, 12,  6,  3,  8,  4,  2,  1],
    [13, 14, 10, 11,  6,  8,  2,  9, 15,  7,  5, 12,  3,  4,  1],
    [15, 10, 12,  8,  1, 15, 10, 12,  8,  1, 15, 10, 12,  8,  1],
    [14, 11,  8,  9,  7, 12,  4, 13, 10,  6,  2, 15,  5,  3,  1],
    [ 7,  6,  1,  7,  6,  1,  7,  6,  1,  7,  6,  1,  7,  6,  1],
    [10,  8, 15, 12,  1, 10,  8, 15, 12,  1, 10,  8, 15, 12,  1]],
   order=2^4)
```

5.7 Linear Sequences

This section contains classes and functions for creating and analyzing linear sequences.

5.7.1 Sequence analysis functions

berlekamp_massey(sequence)	Finds the minimum-degree polynomial $c(x)$ that produces the sequence in $\mathrm{GF}(p^m)$.

galois.berlekamp_massey

galois.**berlekamp_massey**(*sequence*)

Finds the minimum-degree polynomial $c(x)$ that produces the sequence in $\mathrm{GF}(p^m)$.

This function implements the Berlekamp-Massey algorithm.

Parameters **sequence** (galois.FieldArray) – A sequence of Galois field elements in $\mathrm{GF}(p^m)$.

Returns The minimum-degree polynomial $c(x) \in \mathrm{GF}(p^m)(x)$ that produces the input sequence.

Return type *galois.Poly*

Examples

TODO: Add an LFSR example once they're added.

5.8 Numpy Examples

This section contains examples of some numpy functions when called on Galois field arrays. Many more functions are supported, just not explicitly documented here.

5.8.1 General

np.copy(a)	Returns a copy of a given Galois field array.
np.concatenate(arrays[, axis])	Concatenates the input arrays along the given axis.
np.insert(array, object, values[, axis])	Inserts values along the given axis.

np.copy

np.**copy**(*a*)

Returns a copy of a given Galois field array.

See: https://numpy.org/doc/stable/reference/generated/numpy.copy.html

Examples

```
In [1]: GF = galois.GF(2**3)

In [2]: a = GF.Random(5, low=1); a
Out[2]: GF([1, 6, 5, 4, 2], order=2^3)

In [3]: b = np.copy(a); b
Out[3]: GF([1, 6, 5, 4, 2], order=2^3)

In [4]: a[0] = 0; a
Out[4]: GF([0, 6, 5, 4, 2], order=2^3)

# b is unmodified
In [5]: b
Out[5]: GF([1, 6, 5, 4, 2], order=2^3)
```

np.concatenate

np.**concatenate**(*arrays*, *axis=0*)

Concatenates the input arrays along the given axis.

See: https://numpy.org/doc/stable/reference/generated/numpy.concatenate.html

Examples

```
In [1]: GF = galois.GF(2**3)

In [2]: A = GF.Random((2,2)); A
Out[2]:
GF([[1, 1],
    [4, 4]], order=2^3)

In [3]: B = GF.Random((2,2)); B
Out[3]:
GF([[0, 6],
    [6, 4]], order=2^3)

In [4]: np.concatenate((A,B), axis=0)
Out[4]:
GF([[1, 1],
    [4, 4],
    [0, 6],
    [6, 4]], order=2^3)

In [5]: np.concatenate((A,B), axis=1)
Out[5]:
GF([[1, 1, 0, 6],
    [4, 4, 6, 4]], order=2^3)
```

np.insert

np.**insert**(*array*, *object*, *values*, *axis=None*)
Inserts values along the given axis.

See: https://numpy.org/doc/stable/reference/generated/numpy.insert.html

Examples

```
In [1]: GF = galois.GF(2**3)

In [2]: x = GF.Random(5); x
Out[2]: GF([5, 6, 2, 3, 7], order=2^3)

In [3]: np.insert(x, 1, [0,1,2,3])
Out[3]: GF([5, 0, 1, 2, 3, 6, 2, 3, 7], order=2^3)
```

5.8.2 Arithmetic

np.add(x, y)	Adds two Galois field arrays element-wise.
np.subtract(x, y)	Subtracts two Galois field arrays element-wise.
np.multiply(x, y)	Multiplies two Galois field arrays element-wise.
np.divide(x, y)	Divides two Galois field arrays element-wise.
np.negative(x)	Returns the element-wise additive inverse of a Galois field array.
np.reciprocal(x)	Returns the element-wise multiplicative inverse of a Galois field array.
np.power(x, y)	Exponentiates a Galois field array element-wise.
np.square(x)	Squares a Galois field array element-wise.
np.log(x)	Computes the logarithm (base `GF.primitive_element`) of a Galois field array element-wise.
np.matmul(a, b)	Returns the matrix multiplication of two Galois field arrays.

np.add

np.**add**(*x*, *y*)
Adds two Galois field arrays element-wise.

References

- https://numpy.org/doc/stable/reference/generated/numpy.add.html

Examples

```
In [1]: GF = galois.GF(31)

In [2]: x = GF.Random(10); x
Out[2]: GF([18, 28, 28, 20, 15, 19, 11, 16,  2,  5], order=31)

In [3]: y = GF.Random(10); y
Out[3]: GF([26, 22, 17,  2, 23,  5, 12,  0, 13, 10], order=31)

In [4]: np.add(x, y)
Out[4]: GF([13, 19, 14, 22,  7, 24, 23, 16, 15, 15], order=31)

In [5]: x + y
Out[5]: GF([13, 19, 14, 22,  7, 24, 23, 16, 15, 15], order=31)
```

np.subtract

np.**subtract**(*x*, *y*)

Subtracts two Galois field arrays element-wise.

References

- https://numpy.org/doc/stable/reference/generated/numpy.subtract.html

Examples

```
In [1]: GF = galois.GF(31)

In [2]: x = GF.Random(10); x
Out[2]: GF([ 7,  6, 18,  9,  7,  5,  6,  5,  8,  6], order=31)

In [3]: y = GF.Random(10); y
Out[3]: GF([24,  7,  1, 13,  9,  8, 12, 18,  7,  2], order=31)

In [4]: np.subtract(x, y)
Out[4]: GF([14, 30, 17, 27, 29, 28, 25, 18,  1,  4], order=31)

In [5]: x - y
Out[5]: GF([14, 30, 17, 27, 29, 28, 25, 18,  1,  4], order=31)
```

np.multiply

np.**multiply**(*x*, *y*)
Multiplies two Galois field arrays element-wise.

References

- https://numpy.org/doc/stable/reference/generated/numpy.multiply.html

---

Examples

Multiplying two Galois field arrays results in field multiplication.

```
In [1]: GF = galois.GF(31)

In [2]: x = GF.Random(10); x
Out[2]: GF([23,  8, 21,  1,  4, 16, 22,  7,  2, 24], order=31)

In [3]: y = GF.Random(10); y
Out[3]: GF([ 6, 11, 22,  8,  2, 25, 28, 19, 26, 25], order=31)

In [4]: np.multiply(x, y)
Out[4]: GF([14, 26, 28,  8,  8, 28, 27,  9, 21, 11], order=31)

In [5]: x * y
Out[5]: GF([14, 26, 28,  8,  8, 28, 27,  9, 21, 11], order=31)
```

Multiplying a Galois field array with an integer results in scalar multiplication.

```
In [6]: GF = galois.GF(31)

In [7]: x = GF.Random(10); x
Out[7]: GF([18, 13, 19, 28, 11, 13, 28, 29, 30, 15], order=31)

In [8]: np.multiply(x, 3)
Out[8]: GF([23,  8, 26, 22,  2,  8, 22, 25, 28, 14], order=31)

In [9]: x * 3
Out[9]: GF([23,  8, 26, 22,  2,  8, 22, 25, 28, 14], order=31)
```

```
In [10]: print(GF.properties)
GF(31):
  characteristic: 31
  degree: 1
  order: 31

# Adding `characteristic` copies of any element always results in zero
In [11]: x * GF.characteristic
Out[11]: GF([0, 0, 0, 0, 0, 0, 0, 0, 0, 0], order=31)
```

---

np.divide

np.**divide**(*x*, *y*)
Divides two Galois field arrays element-wise.

References

- https://numpy.org/doc/stable/reference/generated/numpy.divide.html

Examples

```
In [1]: GF = galois.GF(31)

In [2]: x = GF.Random(10); x
Out[2]: GF([12, 15, 18, 26, 13, 27, 20, 14, 26,  5], order=31)

In [3]: y = GF.Random(10, low=1); y
Out[3]: GF([26, 19, 11, 22, 11, 13,  6, 23, 15,  7], order=31)

In [4]: z = np.divide(x, y); z
Out[4]: GF([10, 22, 27,  4,  4, 14, 24,  6, 10, 14], order=31)

In [5]: y * z
Out[5]: GF([12, 15, 18, 26, 13, 27, 20, 14, 26,  5], order=31)
```

```
In [6]: np.true_divide(x, y)
Out[6]: GF([10, 22, 27,  4,  4, 14, 24,  6, 10, 14], order=31)

In [7]: x / y
Out[7]: GF([10, 22, 27,  4,  4, 14, 24,  6, 10, 14], order=31)

In [8]: np.floor_divide(x, y)
Out[8]: GF([10, 22, 27,  4,  4, 14, 24,  6, 10, 14], order=31)

In [9]: x // y
Out[9]: GF([10, 22, 27,  4,  4, 14, 24,  6, 10, 14], order=31)
```

np.negative

np.**negative**(*x*)
Returns the element-wise additive inverse of a Galois field array.

References

- https://numpy.org/doc/stable/reference/generated/numpy.negative.html

Examples

```
In [1]: GF = galois.GF(31)

In [2]: x = GF.Random(10); x
Out[2]: GF([18, 20, 22, 22, 24,  5, 23, 27, 23, 21], order=31)

In [3]: y = np.negative(x); y
Out[3]: GF([13, 11,  9,  9,  7, 26,  8,  4,  8, 10], order=31)

In [4]: x + y
Out[4]: GF([0, 0, 0, 0, 0, 0, 0, 0, 0, 0], order=31)
```

```
In [5]: -x
Out[5]: GF([13, 11,  9,  9,  7, 26,  8,  4,  8, 10], order=31)

In [6]: -1*x
Out[6]: GF([13, 11,  9,  9,  7, 26,  8,  4,  8, 10], order=31)
```

np.reciprocal

np.**reciprocal**(*x*)
Returns the element-wise multiplicative inverse of a Galois field array.

References

- https://numpy.org/doc/stable/reference/generated/numpy.reciprocal.html

Examples

```
In [1]: GF = galois.GF(31)

In [2]: x = GF.Random(5, low=1); x
Out[2]: GF([ 7, 22, 14, 30, 13], order=31)

In [3]: y = np.reciprocal(x); y
Out[3]: GF([ 9, 24, 20, 30, 12], order=31)

In [4]: x * y
Out[4]: GF([1, 1, 1, 1, 1], order=31)
```

```
In [5]: x ** -1
Out[5]: GF([ 9, 24, 20, 30, 12], order=31)
```

(continues on next page)

(continued from previous page)

```
In [6]: GF(1) / x
Out[6]: GF([ 9, 24, 20, 30, 12], order=31)

In [7]: GF(1) // x
Out[7]: GF([ 9, 24, 20, 30, 12], order=31)
```

np.power

np.**power**(*x*, *y*)

Exponentiates a Galois field array element-wise.

References

- https://numpy.org/doc/stable/reference/generated/numpy.power.html

Examples

```
In [1]: GF = galois.GF(31)

In [2]: x = GF.Random(10); x
Out[2]: GF([ 2, 20, 10, 10,  6, 19, 10, 26, 27, 10], order=31)

In [3]: np.power(x, 3)
Out[3]: GF([ 8,  2,  8,  8, 30,  8,  8, 30, 29,  8], order=31)

In [4]: x ** 3
Out[4]: GF([ 8,  2,  8,  8, 30,  8,  8, 30, 29,  8], order=31)

In [5]: x * x * x
Out[5]: GF([ 8,  2,  8,  8, 30,  8,  8, 30, 29,  8], order=31)
```

```
In [6]: x = GF.Random(10, low=1); x
Out[6]: GF([ 2, 29, 25,  4,  5,  8, 27, 11, 18, 13], order=31)

In [7]: y = np.random.randint(-10, 10, 10); y
Out[7]: array([ 1,  0, -9,  6, -1,  8,  0,  3, -3,  5])

In [8]: np.power(x, y)
Out[8]: GF([ 2,  1,  1,  4, 25, 16,  1, 29,  8,  6], order=31)

In [9]: x ** y
Out[9]: GF([ 2,  1,  1,  4, 25, 16,  1, 29,  8,  6], order=31)
```

np.square

np.**square**(*x*)
Squares a Galois field array element-wise.

References

- https://numpy.org/doc/stable/reference/generated/numpy.square.html

Examples

```
In [1]: GF = galois.GF(31)

In [2]: x = GF.Random(10); x
Out[2]: GF([19, 24, 16, 27, 25,  9, 17, 24, 16, 30], order=31)

In [3]: np.square(x)
Out[3]: GF([20, 18,  8, 16,  5, 19, 10, 18,  8,  1], order=31)

In [4]: x ** 2
Out[4]: GF([20, 18,  8, 16,  5, 19, 10, 18,  8,  1], order=31)

In [5]: x * x
Out[5]: GF([20, 18,  8, 16,  5, 19, 10, 18,  8,  1], order=31)
```

np.log

np.**log**(*x*)
Computes the logarithm (base `GF.primitive_element`) of a Galois field array element-wise.

Calling *np.log()* implicitly uses base *galois.FieldClass.primitive_element*. See `galois.FieldArray.log()` for logarithm with arbitrary base.

References

- https://numpy.org/doc/stable/reference/generated/numpy.log.html

Examples

```
In [1]: GF = galois.GF(31)

In [2]: alpha = GF.primitive_element; alpha
Out[2]: GF(3, order=31)

In [3]: x = GF.Random(10, low=1); x
Out[3]: GF([29, 24, 18, 28, 26, 19,  4,  8, 16, 22], order=31)

In [4]: y = np.log(x); y
```

(continues on next page)

(continued from previous page)

```
Out[4]: array([ 9, 13, 26, 16,  5,  4, 18, 12,  6, 17])

In [5]: alpha ** y
Out[5]: GF([29, 24, 18, 28, 26, 19,  4,  8, 16, 22], order=31)
```

np.matmul

np.**matmul**(*a*, *b*)

Returns the matrix multiplication of two Galois field arrays.

References

- https://numpy.org/doc/stable/reference/generated/numpy.matmul.html

Examples

```
In [1]: GF = galois.GF(31)

In [2]: A = GF.Random((3,3)); A
Out[2]:
GF([[29,  7, 19],
    [23,  4, 28],
    [18, 18, 23]], order=31)

In [3]: B = GF.Random((3,3)); B
Out[3]:
GF([[ 7,  6,  5],
    [30, 29, 17],
    [21, 23, 23]], order=31)

In [4]: np.matmul(A, B)
Out[4]:
GF([[ 6,  8, 19],
    [ 1, 30, 21],
    [ 2, 12, 26]], order=31)

In [5]: A @ B
Out[5]:
GF([[ 6,  8, 19],
    [ 1, 30, 21],
    [ 2, 12, 26]], order=31)
```

5.8.3 Advanced Arithmetic

np.convolve(a, b)	Convolves the input arrays.

np.convolve

np.**convolve**(*a*, *b*)

Convolves the input arrays.

See: https://numpy.org/doc/stable/reference/generated/numpy.convolve.html

Examples

```
In [1]: GF = galois.GF(31)

In [2]: a = GF.Random(10)

In [3]: b = GF.Random(10)

In [4]: np.convolve(a, b)
Out[4]:
GF([ 6, 18,  4, 18,  0,  1,  4, 10,  2, 17, 28, 23, 30,  1, 12, 13, 26,
    29, 22], order=31)

# Equivalent implementation with native numpy
In [5]: np.convolve(a.view(np.ndarray).astype(int), b.view(np.ndarray).astype(int))
 ↪% 31
Out[5]:
array([ 6, 18,  4, 18,  0,  1,  4, 10,  2, 17, 28, 23, 30,  1, 12, 13, 26,
       29, 22])
```

```
In [6]: GF = galois.GF(2**8)

In [7]: a = GF.Random(10)

In [8]: b = GF.Random(10)

In [9]: np.convolve(a, b)
Out[9]:
GF([249, 237,  92, 247, 152,  48, 231, 211, 234, 135, 216,  84, 234,  47,
    134, 131,  42,   2,  27], order=2^8)
```

5.8.4 Linear Algebra

np.dot(a, b)	Returns the dot product of two Galois field arrays.
np.vdot(a, b)	Returns the dot product of two Galois field vectors.
np.inner(a, b)	Returns the inner product of two Galois field arrays.
np.outer(a, b)	Returns the outer product of two Galois field arrays.
np.matmul(a, b)	Returns the matrix multiplication of two Galois field arrays.
np.linalg.matrix_power(x)	Raises a square Galois field matrix to an integer power.
np.linalg.det(A)	Computes the determinant of the matrix.
np.linalg.matrix_rank(x)	Returns the rank of a Galois field matrix.
np.trace(x)	Returns the sum along the diagonal of a Galois field array.
np.linalg.solve(x)	Solves the system of linear equations.
np.linalg.inv(A)	Computes the inverse of the matrix.

np.dot

np.**dot**(*a*, *b*)

Returns the dot product of two Galois field arrays.

References

- https://numpy.org/doc/stable/reference/generated/numpy.dot.html

Examples

```
In [1]: GF = galois.GF(31)

In [2]: a = GF.Random(3); a
Out[2]: GF([ 7, 30, 25], order=31)

In [3]: b = GF.Random(3); b
Out[3]: GF([13,  3,  0], order=31)

In [4]: np.dot(a, b)
Out[4]: GF(26, order=31)
```

```
In [5]: A = GF.Random((3,3)); A
Out[5]:
GF([[22, 17,  2],
    [ 3,  1, 29],
    [11, 30, 27]], order=31)

In [6]: B = GF.Random((3,3)); B
Out[6]:
GF([[14, 15, 26],
    [22, 21, 18],
    [27, 28,  1]], order=31)
```

(continues on next page)

(continued from previous page)

```
In [7]: np.dot(A, B)
Out[7]:
GF([[23, 30, 12],
    [10, 10,  1],
    [24,  1, 16]], order=31)
```

np.vdot

np.**vdot**(*a*, *b*)

Returns the dot product of two Galois field vectors.

References

- https://numpy.org/doc/stable/reference/generated/numpy.vdot.html

Examples

```
In [1]: GF = galois.GF(31)

In [2]: a = GF.Random(3); a
Out[2]: GF([18,  9, 14], order=31)

In [3]: b = GF.Random(3); b
Out[3]: GF([13, 11, 13], order=31)

In [4]: np.vdot(a, b)
Out[4]: GF(19, order=31)
```

```
In [5]: A = GF.Random((3,3)); A
Out[5]:
GF([[27,  3, 15],
    [ 4, 19,  4],
    [14,  0, 24]], order=31)

In [6]: B = GF.Random((3,3)); B
Out[6]:
GF([[20, 29, 22],
    [30,  4, 21],
    [16, 15, 21]], order=31)

In [7]: np.vdot(A, B)
Out[7]: GF(12, order=31)
```

np.inner

np.**inner**(*a*, *b*)

Returns the inner product of two Galois field arrays.

References

- https://numpy.org/doc/stable/reference/generated/numpy.inner.html

Examples

```
In [1]: GF = galois.GF(31)

In [2]: a = GF.Random(3); a
Out[2]: GF([24,  3, 26], order=31)

In [3]: b = GF.Random(3); b
Out[3]: GF([ 5, 17,  8], order=31)

In [4]: np.inner(a, b)
Out[4]: GF(7, order=31)
```

```
In [5]: A = GF.Random((3,3)); A
Out[5]:
GF([[28, 20, 14],
    [17, 19, 25],
    [ 7,  7,  4]], order=31)

In [6]: B = GF.Random((3,3)); B
Out[6]:
GF([[ 5, 27, 21],
    [ 7,  5, 24],
    [ 1,  4,  9]], order=31)

In [7]: np.inner(A, B)
Out[7]:
GF([[13, 12, 17],
    [ 7,  8,  8],
    [29, 25,  9]], order=31)
```

np.outer

np.**outer**(*a*, *b*)

Returns the outer product of two Galois field arrays.

References

- https://numpy.org/doc/stable/reference/generated/numpy.outer.html

Examples

```
In [1]: GF = galois.GF(31)

In [2]: a = GF.Random(3); a
Out[2]: GF([20,  9, 16], order=31)

In [3]: b = GF.Random(3); b
Out[3]: GF([17,  5,  2], order=31)

In [4]: np.outer(a, b)
Out[4]:
GF([[30,  7,  9],
    [29, 14, 18],
    [24, 18,  1]], order=31)
```

np.linalg.matrix_power

np.linalg.**matrix_power**(*x*)

Raises a square Galois field matrix to an integer power.

References

- https://numpy.org/doc/stable/reference/generated/numpy.linalg.matrix_power.html

Examples

```
In [1]: GF = galois.GF(31)

In [2]: A = GF.Random((3,3)); A
Out[2]:
GF([[12, 26,  0],
    [18,  6,  7],
    [ 0, 28,  4]], order=31)

In [3]: np.linalg.matrix_power(A, 3)
Out[3]:
GF([[20,  8,  5],
    [27, 14,  3],
    [21, 12, 18]], order=31)

In [4]: A @ A @ A
Out[4]:
GF([[20,  8,  5],
```

(continues on next page)

(continued from previous page)

```
    [27, 14,  3],
    [21, 12, 18]], order=31)
```

```
In [5]: GF = galois.GF(31)

# Ensure A is full rank and invertible
In [6]: while True:
   ...:     A = GF.Random((3,3))
   ...:     if np.linalg.matrix_rank(A) == 3:
   ...:         break
   ...:

In [7]: A
Out[7]:
GF([[30, 18, 20],
    [ 6, 15, 27],
    [ 3,  6, 21]], order=31)

In [8]: np.linalg.matrix_power(A, -3)
Out[8]:
GF([[10, 26,  6],
    [17, 12, 19],
    [12, 27, 16]], order=31)

In [9]: A_inv = np.linalg.inv(A)

In [10]: A_inv @ A_inv @ A_inv
Out[10]:
GF([[10, 26,  6],
    [17, 12, 19],
    [12, 27, 16]], order=31)
```

np.linalg.det

np.linalg.**det**(*A*)

Computes the determinant of the matrix.

References

- https://numpy.org/doc/stable/reference/generated/numpy.linalg.det.html

Examples

```
In [1]: GF = galois.GF(31)

In [2]: A = GF.Random((2,2)); A
Out[2]:
GF([[23, 13],
```

(continues on next page)

(continued from previous page)

```
    [29,  7]], order=31)

In [3]: np.linalg.det(A)
Out[3]: GF(1, order=31)

In [4]: A[0,0]*A[1,1] - A[0,1]*A[1,0]
Out[4]: GF(1, order=31)
```

np.linalg.matrix_rank

np.linalg.**matrix_rank**(*x*)

Returns the rank of a Galois field matrix.

References

- https://numpy.org/doc/stable/reference/generated/numpy.linalg.matrix_rank.html

Examples

```
In [1]: GF = galois.GF(31)

In [2]: A = GF.Identity(4); A
Out[2]:
GF([[1, 0, 0, 0],
    [0, 1, 0, 0],
    [0, 0, 1, 0],
    [0, 0, 0, 1]], order=31)

In [3]: np.linalg.matrix_rank(A)
Out[3]: 4
```

One column is a linear combination of another.

```
In [4]: GF = galois.GF(31)

In [5]: A = GF.Random((4,4)); A
Out[5]:
GF([[ 0,  4,  5, 22],
    [ 5, 25,  4, 16],
    [ 3,  8, 13, 12],
    [13, 23,  7, 29]], order=31)

In [6]: A[:,2] = A[:,1] * GF(17); A
Out[6]:
GF([[ 0,  4,  6, 22],
    [ 5, 25, 22, 16],
    [ 3,  8, 12, 12],
    [13, 23, 19, 29]], order=31)
```

(continues on next page)

(continued from previous page)

```
In [7]: np.linalg.matrix_rank(A)
Out[7]: 3
```

One row is a linear combination of another.

```
In [8]: GF = galois.GF(31)

In [9]: A = GF.Random((4,4)); A
Out[9]:
GF([[14, 29, 21, 17],
    [26, 27,  1,  4],
    [ 6, 18, 19, 22],
    [18, 19, 10,  6]], order=31)

In [10]: A[3,:] = A[2,:] * GF(8); A
Out[10]:
GF([[14, 29, 21, 17],
    [26, 27,  1,  4],
    [ 6, 18, 19, 22],
    [17, 20, 28, 21]], order=31)

In [11]: np.linalg.matrix_rank(A)
Out[11]: 3
```

np.trace

np.**trace**(*x*)

Returns the sum along the diagonal of a Galois field array.

References

- https://numpy.org/doc/stable/reference/generated/numpy.trace.html

Examples

```
In [1]: GF = galois.GF(31)

In [2]: A = GF.Random((5,6)); A
Out[2]:
GF([[26,  6,  4,  4, 15,  7],
    [22, 26,  6, 12, 29, 11],
    [ 8, 29,  7, 29,  1,  7],
    [ 5, 26, 15,  1, 26,  1],
    [ 0, 19,  8, 25, 17, 10]], order=31)

In [3]: np.trace(A)
Out[3]: GF(15, order=31)
```

(continues on next page)

(continued from previous page)

```
In [4]: A[0,0] + A[1,1] + A[2,2] + A[3,3] + A[4,4]
Out[4]: GF(15, order=31)
```

```
In [5]: np.trace(A, offset=1)
Out[5]: GF(15, order=31)

In [6]: A[0,1] + A[1,2] + A[2,3] + A[3,4] + A[4,5]
Out[6]: GF(15, order=31)
```

---

np.linalg.solve

np.linalg.**solve**(*x*)

Solves the system of linear equations.

References

- https://numpy.org/doc/stable/reference/generated/numpy.linalg.solve.html

---

Examples

```
In [1]: GF = galois.GF(31)

# Ensure A is full rank and invertible
In [2]: while True:
   ...:     A = GF.Random((4,4))
   ...:     if np.linalg.matrix_rank(A) == 4:
   ...:         break
   ...:

In [3]: A
Out[3]:
GF([[28, 16, 20,  1],
    [21, 24, 11, 28],
    [11, 16, 23, 30],
    [24, 29,  9, 12]], order=31)

In [4]: b = GF.Random(4); b
Out[4]: GF([17, 18,  1, 27], order=31)

In [5]: x = np.linalg.solve(A, b); x
Out[5]: GF([25, 30,  3, 17], order=31)

In [6]: A @ x
Out[6]: GF([17, 18,  1, 27], order=31)
```

```
In [7]: GF = galois.GF(31)
```

(continues on next page)

(continued from previous page)

```
# Ensure A is full rank and invertible
In [8]: while True:
   ...:     A = GF.Random((4,4))
   ...:     if np.linalg.matrix_rank(A) == 4:
   ...:         break
   ...:

In [9]: A
Out[9]:
GF([[17,  8,  9, 24],
    [12, 29, 11, 12],
    [21, 18,  8, 19],
    [20, 16,  3, 11]], order=31)

In [10]: B = GF.Random((4,2)); B
Out[10]:
GF([[20, 21],
    [ 9, 10],
    [ 5,  9],
    [28,  8]], order=31)

In [11]: X = np.linalg.solve(A, B); X
Out[11]:
GF([[30,  8],
    [10,  3],
    [12, 27],
    [26, 28]], order=31)

In [12]: A @ X
Out[12]:
GF([[20, 21],
    [ 9, 10],
    [ 5,  9],
    [28,  8]], order=31)
```

np.linalg.inv

np.linalg.**inv**(*A*)
Computes the inverse of the matrix.

References

- https://numpy.org/doc/stable/reference/generated/numpy.linalg.inv.html

Examples

```
In [1]: GF = galois.GF(31)

# Ensure A is full rank and invertible
In [2]: while True:
   ...:     A = GF.Random((3,3))
   ...:     if np.linalg.matrix_rank(A) == 3:
   ...:         break
   ...:

In [3]: A
Out[3]:
GF([[ 9, 28, 15],
    [20,  6,  3],
    [17, 14, 22]], order=31)

In [4]: A_inv = np.linalg.inv(A); A_inv
Out[4]:
GF([[ 1,  1,  2],
    [16, 19,  2],
    [13, 28, 24]], order=31)

In [5]: A_inv @ A
Out[5]:
GF([[1, 0, 0],
    [0, 1, 0],
    [0, 0, 1]], order=31)
```

CHAPTER

SIX

RELEASE NOTES

6.1 v0.0.18

6.1.1 Breaking Changes

- Make API more consistent with software like Matlab and Wolfram:
  - Rename `galois.prime_factors()` to `galois.factors()`.
  - Rename `galois.gcd()` to `galois.egcd()` and add `galois.gcd()` for conventional GCD.
  - Rename `galois.poly_gcd()` to `galois.poly_egcd()` and add `galois.poly_gcd()` for conventional GCD.
  - Rename `galois.euler_totient()` to `galois.euler_phi()`.
  - Rename `galois.carmichael()` to `galois.carmichael_lambda()`.
  - Rename `galois.is_prime_fermat()` to `galois.fermat_primality_test()`.
  - Rename `galois.is_prime_miller_rabin()` to `galois.miller_rabin_primality_test()`.
- Rename polynomial search `method` keyword argument values from `["smallest", "largest", "random"]` to `["min", "max", "random"]`.

6.1.2 Changes

- Clean up `galois` API and `dir()` so only public classes and functions are displayed.
- Speed-up `galois.is_primitive()` test and search for primitive polynomials in `galois.primitive_poly()`.
- Speed-up `galois.is_smooth()`.
- Add Reed-Solomon codes in `galois.ReedSolomon`.
- Add shortened BCH and Reed-Solomon codes.
- Add error detection for BCH and Reed-Solomon with the `detect()` method.
- Add general cyclic linear block code functions.
- Add Matlab default primitive polynomial with `galois.matlab_primitive_poly()`.
- Add number theoretic functions:
  - Add `galois.legendre_symbol()`, `galois.jacobi_symbol()`, `galois.kronecker_symbol()`.
  - Add `galois.divisors()`, `galois.divisor_sigma()`.

  - Add `galois.is_composite()`, `galois.is_prime_power()`, `galois.is_perfect_power()`, `galois.is_square_free()`, `galois.is_powersmooth()`.
  - Add `galois.are_coprime()`.
- Clean up integer factorization algorithms and add some to public API:
  - Add `galois.perfect_power()`, `galois.trial_division()`, `galois.pollard_p1()`, `galois.pollard_rho()`.
- Clean up API reference structure and hierarchy.
- Fix minor bugs in BCH codes.

6.1.3 Contributors

- Matt Hostetter (@mhostetter)

6.2 v0.0.17

6.2.1 Breaking Changes

- Rename `FieldMeta` to `FieldClass`.
- Remove `target` keyword from `FieldClass.compile()` until there is better support for GPUs.
- Consolidate `verify_irreducible` and `verify_primitive` keyword arguments into `verify` for the `galois.GF()` class factory function.
- Remove group arrays until there is more complete support.

6.2.2 Changes

- Speed-up Galois field class creation time.
- Speed-up JIT compilation time by caching functions.
- Speed-up `Poly.roots()` by JIT compiling it.
- Add BCH codes with `galois.BCH`.
- Add ability to generate irreducible polynomials with `irreducible_poly()` and `irreducible_polys()`.
- Add ability to generate primitive polynomials with `primitive_poly()` and `primitive_polys()`.
- Add computation of the minimal polynomial of an element of an extension field with `minimal_poly()`.
- Add display of arithmetic tables with `FieldClass.arithmetic_table()`.
- Add display of field element representation table with `FieldClass.repr_table()`.
- Add Berlekamp-Massey algorithm in `berlekamp_massey()`.
- Enable ipython tab-completion of Galois field classes.
- Cleanup API reference page.
- Add introduction to Galois fields tutorials.
- Fix bug in `is_primitive()` where some reducible polynomials were marked irreducible.

- Fix bug in integer<–>polynomial conversions for large binary polynomials.
- Fix bug in "power" display mode of 0.
- Other minor bug fixes.

6.2.3 Contributors

- Dominik Wernberger (@Werni2A)
- Matt Hostetter (@mhostetter)

6.3 v0.0.16

6.3.1 Changes

- Add `Field()` alias of `GF()` class factory.
- Add finite groups modulo `n` with `Group()` class factory.
- Add `is_group()`, `is_field()`, `is_prime_field()`, `is_extension_field()`.
- Add polynomial constructor `Poly.String()`.
- Add polynomial factorization in `poly_factors()`.
- Add `np.vdot()` support.
- Fix PyPI packaging issue from v0.0.15.
- Fix bug in creation of 0-degree polynomials.
- Fix bug in `poly_gcd()` not returning monic GCD polynomials.

6.3.2 Contributors

- Matt Hostetter (@mhostetter)

6.4 v0.0.15

6.4.1 Breaking Changes

- Rename `poly_exp_mod()` to `poly_pow()` to mimic the native `pow()` function.
- Rename `fermat_primality_test()` to `is_prime_fermat()`.
- Rename `miller_rabin_primality_test()` to `is_prime_miller_rabin()`.

6.4.2 Changes

- Massive linear algebra speed-ups. (See #88)
- Massive polynomial speed-ups. (See #88)
- Various Galois field performance enhancements. (See #92)
- Support `np.convolve()` for two Galois field arrays.
- Allow polynomial arithmetic with Galois field scalars (of the same field). (See #99), e.g.

```
>>> GF = galois.GF(3)

>>> p = galois.Poly([1,2,0], field=GF)
Poly(x^2 + 2x, GF(3))

>>> p * GF(2)
Poly(2x^2 + x, GF(3))
```

- Allow creation of 0-degree polynomials from integers. (See #99), e.g.

```
>>> p = galois.Poly(1)
Poly(1, GF(2))
```

- Add the four Oakley fields from RFC 2409.
- Speed-up unit tests.
- Restructure API reference.

6.4.3 Contributors

- Matt Hostetter (@mhostetter)

6.5 v0.0.14

6.5.1 Breaking Changes

- Rename `GFArray.Eye()` to `GFArray.Identity()`.
- Rename `chinese_remainder_theorem()` to `crt()`.

6.5.2 Changes

- Lots of performance improvements.
- Additional linear algebra support.
- Various bug fixes.

6.5.3 Contributors

- Baalateja Kataru (@BK-Modding)
- Matt Hostetter (@mhostetter)

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

A

add() (*in module np*), 169
are_coprime() (*in module galois*), 121
arithmetic_table() (*galois.FieldClass method*), 58

B

BCH (*class in galois*), 149
bch_valid_codes() (*in module galois*), 165
berlekamp_massey() (*in module galois*), 167

C

c (*galois.ReedSolomon property*), 163
carmichael_lambda() (*in module galois*), 123
characteristic (*galois.FieldClass property*), 63
coeffs (*galois.Poly property*), 108
compile() (*galois.FieldClass method*), 60
concatenate() (*in module np*), 168
convolve() (*in module np*), 177
conway_poly() (*in module galois*), 91
copy() (*in module np*), 167
crt() (*in module galois*), 122

D

d (*galois.BCH property*), 155
d (*galois.ReedSolomon property*), 163
decode() (*galois.BCH method*), 150
decode() (*galois.ReedSolomon method*), 158
default_ufunc_mode (*galois.FieldClass property*), 64
degree (*galois.FieldClass property*), 64
degree (*galois.Poly property*), 109
degrees (*galois.Poly property*), 109
Degrees() (*galois.Poly class method*), 101
derivative() (*galois.Poly method*), 105
det() (*in module np.linalg*), 182
detect() (*galois.BCH method*), 153
detect() (*galois.ReedSolomon method*), 160
display() (*galois.FieldClass method*), 60
display_mode (*galois.FieldClass property*), 65
divide() (*in module np*), 172
divisor_sigma() (*in module galois*), 132
divisors() (*in module galois*), 132
dot() (*in module np*), 178
dtypes (*galois.FieldClass property*), 66

E

egcd() (*in module galois*), 118
Elements() (*galois.FieldArray class method*), 49
Elements() (*galois.GF2 class method*), 74
encode() (*galois.BCH method*), 153
encode() (*galois.ReedSolomon method*), 161
euler_phi() (*in module galois*), 119

F

factors() (*in module galois*), 131
fermat_primality_test() (*in module galois*), 146
Field (*class in galois*), 45
field (*galois.BCH property*), 155
field (*galois.Poly property*), 109
field (*galois.ReedSolomon property*), 163
FieldArray (*class in galois*), 46
FieldClass (*class in galois*), 57

G

G (*galois.BCH property*), 155
G (*galois.ReedSolomon property*), 162
gcd() (*in module galois*), 117
generator_poly (*galois.BCH property*), 155
generator_poly (*galois.ReedSolomon property*), 163
generator_to_parity_check_matrix() (*in module galois*), 164
GF (*class in galois*), 43
GF2 (*class in galois*), 73

H

H (*galois.BCH property*), 155
H (*galois.ReedSolomon property*), 163

I

Identity() (*galois.FieldArray class method*), 49
Identity() (*galois.GF2 class method*), 75
Identity() (*galois.Poly class method*), 102
ilog() (*in module galois*), 129

inner() (*in module np*), 180
insert() (*in module np*), 169
integer (*galois.Poly property*), 110
Integer() (*galois.Poly class method*), 102
inv() (*in module np.linalg*), 186
iroot() (*in module galois*), 129
irreducible_poly (*galois.FieldClass property*), 67
irreducible_poly() (*in module galois*), 86
irreducible_polys() (*in module galois*), 87
is_composite() (*in module galois*), 139
is_cyclic() (*in module galois*), 126
is_extension_field (*galois.FieldClass property*), 67
is_irreducible() (*in module galois*), 88
is_monic() (*in module galois*), 116
is_narrow_sense (*galois.BCH property*), 155
is_narrow_sense (*galois.ReedSolomon property*), 163
is_perfect_power() (*in module galois*), 138
is_powersmooth() (*in module galois*), 140
is_prime() (*in module galois*), 137
is_prime_field (*galois.FieldClass property*), 67
is_prime_power() (*in module galois*), 138
is_primitive (*galois.BCH property*), 155
is_primitive() (*in module galois*), 93
is_primitive_element() (*in module galois*), 96
is_primitive_poly (*galois.FieldClass property*), 68
is_primitive_root() (*in module galois*), 85
is_smooth() (*in module galois*), 140
is_square_free() (*in module galois*), 139
isqrt() (*in module galois*), 128

J

jacobi_symbol() (*in module galois*), 124

K

k (*galois.BCH property*), 155
k (*galois.ReedSolomon property*), 163
kronecker_symbol() (*in module galois*), 125
kth_prime() (*in module galois*), 142

L

lcm() (*in module galois*), 118
legendre_symbol() (*in module galois*), 124
log() (*in module np*), 175
log_naive() (*in module galois*), 130
lu_decompose() (*galois.FieldArray method*), 53
lup_decompose() (*galois.FieldArray method*), 54

M

matlab_primitive_poly() (*in module galois*), 92
matmul() (*in module np*), 176
matrix_power() (*in module np.linalg*), 181
matrix_rank() (*in module np.linalg*), 183
mersenne_exponents() (*in module galois*), 144
mersenne_primes() (*in module galois*), 144
miller_rabin_primality_test() (*in module galois*), 147
minimal_poly() (*in module galois*), 97
multiply() (*in module np*), 171

N

n (*galois.BCH property*), 156
n (*galois.ReedSolomon property*), 163
name (*galois.FieldClass property*), 69
negative() (*in module np*), 172
next_prime() (*in module galois*), 142
nonzero_coeffs (*galois.Poly property*), 110
nonzero_degrees (*galois.Poly property*), 110

O

One() (*galois.Poly class method*), 103
Ones() (*galois.FieldArray class method*), 50
Ones() (*galois.GF2 class method*), 75
order (*galois.FieldClass property*), 69
outer() (*in module np*), 180

P

parity_check_to_generator_matrix() (*in module galois*), 164
perfect_power() (*in module galois*), 133
pollard_p1() (*in module galois*), 135
pollard_rho() (*in module galois*), 136
Poly (*class in galois*), 99
poly_egcd() (*in module galois*), 112
poly_factors() (*in module galois*), 114
poly_gcd() (*in module galois*), 112
poly_pow() (*in module galois*), 113
poly_to_generator_matrix() (*in module galois*), 165
pow() (*in module galois*), 121
power() (*in module np*), 174
prev_prime() (*in module galois*), 142
prime_subfield (*galois.FieldClass property*), 69
primes() (*in module galois*), 141
primitive_element (*galois.FieldClass property*), 70
primitive_element() (*in module galois*), 94
primitive_elements (*galois.FieldClass property*), 70
primitive_elements() (*in module galois*), 95
primitive_poly() (*in module galois*), 89
primitive_polys() (*in module galois*), 90
primitive_root() (*in module galois*), 79
primitive_roots() (*in module galois*), 82
properties (*galois.FieldClass property*), 71

R

Random() (*galois.FieldArray class method*), 50
Random() (*galois.GF2 class method*), 76
Random() (*galois.Poly class method*), 103

random_prime() (*in module galois*), 143
Range() (*galois.FieldArray class method*), 51
Range() (*galois.GF2 class method*), 76
reciprocal() (*in module np*), 173
ReedSolomon (*class in galois*), 156
repr_table() (*galois.FieldClass method*), 61
roots (*galois.BCH property*), 156
roots (*galois.ReedSolomon property*), 163
Roots() (*galois.Poly class method*), 104
roots() (*galois.Poly method*), 107
roots_to_parity_check_matrix() (*in module galois*), 166
row_reduce() (*galois.FieldArray method*), 55

S

solve() (*in module np.linalg*), 185
square() (*in module np*), 175
string (*galois.Poly property*), 111
String() (*galois.Poly class method*), 104
subtract() (*in module np*), 170
systematic (*galois.BCH property*), 156
systematic (*galois.ReedSolomon property*), 163

T

t (*galois.BCH property*), 156
t (*galois.ReedSolomon property*), 163
totatives() (*in module galois*), 120
trace() (*in module np*), 184
trial_division() (*in module galois*), 134

U

ufunc_mode (*galois.FieldClass property*), 72
ufunc_modes (*galois.FieldClass property*), 72

V

Vandermonde() (*galois.FieldArray class method*), 51
Vandermonde() (*galois.GF2 class method*), 77
vdot() (*in module np*), 179
Vector() (*galois.FieldArray class method*), 52
vector() (*galois.FieldArray method*), 56
Vector() (*galois.GF2 class method*), 77

Z

Zero() (*galois.Poly class method*), 105
Zeros() (*galois.FieldArray class method*), 53
Zeros() (*galois.GF2 class method*), 78