galois

Matt Hostetter

Sep 09, 2021

CONTENTS

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

BASIC USAGE

The main idea of the *galois* package can be summarized as follows. The user creates a "Galois field array class" using `GF = galois.GF(p**m)`. A Galois field array class `GF` is a subclass of `numpy.ndarray` and its constructor `x = GF(array_like)` mimics the call signature of `numpy.array()`. A Galois field array `x` is operated on like any other numpy array, but all arithmetic is performed in $\mathrm{GF}(p^m)$ not $\mathbb{Z}$ or $\mathbb{R}$.

Internally, the Galois field arithmetic is implemented by replacing numpy ufuncs. The new ufuncs are written in python and then just-in-time compiled with numba. The ufuncs can be configured to use either lookup tables (for speed) or explicit calculation (for memory savings). Numba also provides the ability to "target" the JIT-compiled ufuncs for CPUs or GPUs.

In addition to normal array arithmetic, *galois* also supports linear algebra (with `numpy.linalg` functions) and polynomials over Galois fields (with the *galois.Poly* class).

2.1 Class construction

Galois field array classes are created using the *galois.GF()* class factory function.

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

In [2]: import galois

In [3]: GF256 = galois.GF(2**8)

In [4]: print(GF256)
<class 'numpy.ndarray over GF(2^8)'>
```

These classes are subclasses of *galois.GFArray* (which itself subclasses `numpy.ndarray`) and have *galois.GFMeta* as their metaclass.

```
In [5]: issubclass(GF256, np.ndarray)
Out[5]: True

In [6]: issubclass(GF256, galois.GFArray)
Out[6]: True

In [7]: issubclass(type(GF256), galois.GFMeta)
Out[7]: True
```

A Galois field array class contains attributes relating to its Galois field and methods to modify how the field is calculated or displayed. See the attributes and methods in *galois.GFMeta*.

```
# Summarizes some properties of the Galois field
In [8]: 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)

# Access each attribute individually
In [9]: GF256.irreducible_poly
Out[9]: Poly(x^8 + x^4 + x^3 + x^2 + 1, GF(2))
```

The *galois* package even supports arbitrarily-large fields! This is accomplished by using numpy arrays with `dtype=object` and pure-python ufuncs. This comes at a performance penalty compared to smaller fields which use numpy integer dtypes (e.g., `numpy.uint32`) and have compiled ufuncs.

```
In [10]: GF = galois.GF(36893488147419103183); print(GF.properties)
GF(36893488147419103183):
  characteristic: 36893488147419103183
  degree: 1
  order: 36893488147419103183
  irreducible_poly: Poly(x + 36893488147419103180, GF(36893488147419103183))
  is_primitive_poly: True
  primitive_element: GF(3, order=36893488147419103183)

In [11]: GF = galois.GF(2**100); print(GF.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)
```

2.2 Array creation

Galois field arrays can be created from existing numpy arrays.

```
# Represents an existing numpy array
In [12]: array = np.random.randint(0, GF256.order, 10, dtype=int); array
Out[12]: array([209, 156, 155,   1, 102,  46, 144,   2, 140, 117])

# Explicit Galois field array creation (a copy is performed)
In [13]: GF256(array)
Out[13]: GF([209, 156, 155,   1, 102,  46, 144,   2, 140, 117], order=2^8)

# Or view an existing numpy array as a Galois field array (no copy is performed)
```

(continues on next page)

(continued from previous page)

```
In [14]: array.view(GF256)
Out[14]: GF([209, 156, 155,   1, 102,  46, 144,   2, 140, 117], order=2^8)
```

Or they can be created from "array-like" objects. These include strings representing a Galois field element as a polynomial over its prime subfield.

```
# Arrays can be specified as iterables of iterables
In [15]: GF256([[217, 130, 42], [74, 208, 113]])
Out[15]:
GF([[217, 130,  42],
    [ 74, 208, 113]], order=2^8)

# You can mix-and-match polynomial strings and integers
In [16]: GF256(["x^6 + 1", 2, "1", "x^5 + x^4 + x"])
Out[16]: GF([65,  2,  1, 50], order=2^8)

# Single field elements are 0-dimensional arrays
In [17]: GF256("x^6 + x^4 + 1")
Out[17]: GF(81, order=2^8)
```

Galois field arrays also have constructor class methods for convenience. They include:

- *galois.GFArray.Zeros()*, *galois.GFArray.Ones()*, *galois.GFArray.Identity()*, *galois.GFArray.Range()*, *galois.GFArray.Random()*, *galois.GFArray.Elements()*

Galois field elements can either be displayed using their integer representation, polynomial representation, or power representation. Calling *galois.GFMeta.display()* will change the element representation. If called as a context manager, the display mode will only be temporarily changed.

```
In [18]: x = GF256(["y**6 + 1", 0, 2, "1", "y**5 + y**4 + y"]); x
Out[18]: GF([65,  0,  2,  1, 50], order=2^8)

# Set the display mode to represent GF(2^8) field elements as polynomials over GF(2)␣
↪with degree less than 8
In [19]: GF256.display("poly");

In [20]: x
Out[20]: GF([^6 + 1, 0, , 1, ^5 + ^4 + ], order=2^8)

# Temporarily set the display mode to represent GF(2^8) field elements as powers of the␣
↪primitive element
In [21]: with GF256.display("power"):
   ....:     print(x)
   ....:
GF([^191, -∞, , 1, ^194], order=2^8)

# Resets the display mode to the integer representation
In [22]: GF256.display();
```

2.3 Field arithmetic

Galois field arrays are treated like any other numpy array. Array arithmetic is performed using python operators or numpy functions.

In the list below, `GF` is a Galois field array class created by `GF = galois.GF(p**m)`, `x` and `y` are `GF` arrays, and `z` is an integer `numpy.ndarray`. All arithmetic operations follow normal numpy broadcasting rules.

- Addition: `x + y == np.add(x, y)`
- Subtraction: `x - y == np.subtract(x, y)`
- Multiplication: `x * y == np.multiply(x, y)`
- Division: `x / y == x // y == np.divide(x, y)`
- Scalar multiplication: `x * z == np.multiply(x, z)`, e.g. `x * 3 == x + x + x`
- Additive inverse: `-x == np.negative(x)`
- Multiplicative inverse: `GF(1) / x == np.reciprocal(x)`
- Exponentiation: `x ** z == np.power(x, z)`, e.g. `x ** 3 == x * x * x`
- Logarithm: `np.log(x)`, e.g. `GF.primitive_element ** np.log(x) == x`
- **COMING SOON:** Logarithm base `b`: `GF.log(x, b)`, where `b` is any field element
- Matrix multiplication: `A @ B == np.matmul(A, B)`

```
In [23]: x = GF256.Random((2,5)); x
Out[23]:
GF([[ 10, 202,  29,  80, 213],
    [  3, 156, 227, 193,  22]], order=2^8)

In [24]: y = GF256.Random(5); y
Out[24]: GF([146, 252, 204,  39,  84], order=2^8)

# y is broadcast over the last dimension of x
In [25]: x + y
Out[25]:
GF([[152,  54, 209, 119, 129],
    [145,  96,  47, 230,  66]], order=2^8)
```

2.4 Linear algebra

The *galois* package intercepts relevant calls to numpy's linear algebra functions and performs the specified operation in $\mathrm{GF}(p^m)$ not in $\mathbb{R}$. Some of these functions include:

- *np.trace()*
- `np.dot()`, `np.inner()`, `np.outer()`
- *np.linalg.matrix_rank()*, *np.linalg.matrix_power()*
- *np.linalg.det()*, *np.linalg.inv()*, *np.linalg.solve()*

```
In [26]: A = GF256.Random((3,3)); A
Out[26]:
GF([[109, 190, 180],
    [216, 242, 157],
    [153,  32,  59]], order=2^8)

In [27]: b = GF256.Random(3); b
Out[27]: GF([ 25, 249,  75], order=2^8)

In [28]: x = np.linalg.solve(A, b); x
Out[28]: GF([33, 75, 60], order=2^8)

In [29]: np.array_equal(A @ x, b)
Out[29]: True
```

Galois field arrays also contain matrix decomposition routines not included in numpy. These include:

- *galois.GFArray.row_reduce()*, *galois.GFArray.lu_decompose()*, *galois.GFArray.lup_decompose()*

2.5 Numpy ufunc methods

Galois field arrays support numpy ufunc methods. This allows the user to apply a ufunc in a unique way across the target array. The ufunc method signature is `<ufunc>.<method>(*args, **kwargs)`. All arithmetic ufuncs are supported. Below is a list of their ufunc methods.

- `<method>`: `reduce`, `accumulate`, `reduceat`, `outer`, `at`

```
In [30]: X = GF256.Random((2,5)); X
Out[30]:
GF([[ 94, 239,  78, 148, 171],
    [243,  63, 220, 144, 121]], order=2^8)

In [31]: np.multiply.reduce(X, axis=0)
Out[31]: GF([ 70, 102,  69, 116, 169], order=2^8)
```

```
In [32]: x = GF256.Random(5); x
Out[32]: GF([ 39, 132, 188, 181, 136], order=2^8)

In [33]: y = GF256.Random(5); y
Out[33]: GF([223, 250,  26,   8,  58], order=2^8)

In [34]: np.multiply.outer(x, y)
Out[34]:
GF([[197, 234,  33,  37, 181],
    [244,  23, 233,  84, 164],
    [209, 185, 227, 137, 253],
    [184, 192,  41, 193,  10],
    [ 25,  91,  81,  52, 129]], order=2^8)
```

2.6 Numpy functions

Many other relevant numpy functions are supported on Galois field arrays. These include:

- *np.copy()*, *np.concatenate()*, *np.insert()*, np.reshape()

2.7 Polynomial construction

The *galois* package supports polynomials over Galois fields with the *galois.Poly* class. *galois.Poly* does not subclass numpy.ndarray but instead contains a *galois.GFArray* of coefficients as an attribute (implements the "has-a", not "is-a", architecture).

Polynomials can be created by specifying the polynomial coefficients as either a *galois.GFArray* or an "array-like" object with the field keyword argument.

```
In [35]: p = galois.Poly([172, 22, 0, 0, 225], field=GF256); p
Out[35]: Poly(172x^4 + 22x^3 + 225, GF(2^8))

In [36]: coeffs = GF256([33, 17, 0, 225]); coeffs
Out[36]: GF([ 33,  17,   0, 225], order=2^8)

In [37]: p = galois.Poly(coeffs, order="asc"); p
Out[37]: Poly(225x^3 + 17x + 33, GF(2^8))
```

Polynomials over Galois fields can also display the field elements as polynomials over their prime subfields. This can be quite confusing to read, so be warned!

```
In [38]: print(p)
Poly(225x^3 + 17x + 33, GF(2^8))

In [39]: with GF256.display("poly"):
   ....:     print(p)
   ....:
Poly((^7 + ^6 + ^5 + 1)x^3 + (^4 + 1)x + (^5 + 1), GF(2^8))
```

Polynomials can also be created using a number of constructor class methods. They include:

- *galois.Poly.Zero()*, *galois.Poly.One()*, *galois.Poly.Identity()*, *galois.Poly.Random()*, *galois.Poly.Integer()*, *galois.Poly.Degrees()*, *galois.Poly.Roots()*

```
# Construct a polynomial by specifying its roots
In [40]: q = galois.Poly.Roots([155, 37], field=GF256); q
Out[40]: Poly(x^2 + 190x + 71, GF(2^8))

In [41]: q.roots()
Out[41]: GF([ 37, 155], order=2^8)
```

2.8 Polynomial arithmetic

Polynomial arithmetic is performed using python operators.

```
In [42]: p
Out[42]: Poly(225x^3 + 17x + 33, GF(2^8))

In [43]: q
Out[43]: Poly(x^2 + 190x + 71, GF(2^8))

In [44]: p + q
Out[44]: Poly(225x^3 + x^2 + 175x + 102, GF(2^8))

In [45]: divmod(p, q)
Out[45]: (Poly(225x + 57, GF(2^8)), Poly(56x + 104, GF(2^8)))

In [46]: p ** 2
Out[46]: Poly(171x^6 + 28x^2 + 117, GF(2^8))
```

Polynomials over Galois fields can be evaluated at scalars or arrays of field elements.

```
In [47]: p = galois.Poly.Degrees([4, 3, 0], [172, 22, 225], field=GF256); p
Out[47]: Poly(172x^4 + 22x^3 + 225, GF(2^8))

# Evaluate the polynomial at a single value
In [48]: p(1)
Out[48]: GF(91, order=2^8)

In [49]: x = GF256.Random((2,5)); x
Out[49]:
GF([[ 68, 104,  42, 115,  64],
    [ 25,  10, 151, 175, 225]], order=2^8)

# Evaluate the polynomial at an array of values
In [50]: p(x)
Out[50]:
GF([[186, 197,  90,  17,  31],
    [ 89, 154, 144,  22,  88]], order=2^8)
```

Polynomials can also be evaluated in superfields. For example, evaluating a Galois field's irreducible polynomial at one of its elements.

```
# Notice the irreducible polynomial is over GF(2), not GF(2^8)
In [51]: p = GF256.irreducible_poly; p
Out[51]: Poly(x^8 + x^4 + x^3 + x^2 + 1, GF(2))

In [52]: GF256.is_primitive_poly
Out[52]: True

# Notice the primitive element is in GF(2^8)
In [53]: alpha = GF256.primitive_element; alpha
Out[53]: GF(2, order=2^8)
```

(continues on next page)

(continued from previous page)

```
# Since p(x) is a primitive polynomial, alpha is one of its roots
In [54]: p(alpha, field=GF256)
Out[54]: GF(0, order=2^8)
```

CHAPTER

THREE

TUTORIALS

3.1 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.GFArray* with metaclass *galois.GFMeta*. 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.GFArray)
Out[4]: True

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

In [6]: print(GF2.properties)
GF(2):
  characteristic: 2
  degree: 1
  order: 2
  irreducible_poly: Poly(x + 1, GF(2))
  is_primitive_poly: True
  primitive_element: GF(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)'>
```

(continues on next page)

(continued from previous page)

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

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

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

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)
```

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.GFArray)
Out[16]: True

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

In [18]: print(GF7.properties)
GF(7):
  characteristic: 7
  degree: 1
  order: 7
  irreducible_poly: Poly(x + 4, GF(7))
  is_primitive_poly: True
  primitive_element: GF(3, order=7)
```

3.2 Array creation

3.2.1 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([3, 4, 3, 5, 6, 4, 2, 0, 4, 6])

# Create a Galois field array through explicit construction (x_np is copied)
In [2]: x = GF7(x_np); x
Out[2]: GF([3, 4, 3, 5, 6, 4, 2, 0, 4, 6], order=7)
```

3.2.2 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([3, 4, 3, 5, 6, 4, 2, 0, 4, 6], 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([3, 4, 3, 5, 6, 4, 2, 0, 4, 6])

# 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([4, 4, 3, 5, 6, 4, 2, 0, 4, 6])

# Notice x is unchanged due to the copy during the explicit construction
In [6]: x
Out[6]: GF([3, 4, 3, 5, 6, 4, 2, 0, 4, 6], order=7)

# Notice y is changed due to view casting
In [7]: y
Out[7]: GF([4, 4, 3, 5, 6, 4, 2, 0, 4, 6], order=7)
```

3.2.3 Alternate constructors

There are alternate constructors for convenience: *galois.GFArray.Zeros*, *galois.GFArray.Ones*, *galois.GFArray.Range*, *galois.GFArray.Random*, and *galois.GFArray.Elements*.

```
In [8]: GF256.Random((2,5))
Out[8]:
GF([[101,  50,  68,  86, 203],
    [101, 122,  87, 208,  90]], order=2^8)
```

(continues on next page)

(continued from previous page)

```
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,
    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)
```

3.2.4 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, 3, 1, 5, 1, 2, 5, 0, 6], 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, 3, 1, 5, 1, 2, 5, 0, 6], 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([2, 0, 4, 0, 1, 3, 0, 4, 1, 4], order=7)
```

(continues on next page)

(continued from previous page)

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

3.2.5 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-:math`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)

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

# The default mode represents the field elements as integers
In [20]: a
Out[20]: GF([3, 1, 5, 2, 3, 0, 2, 4, 2, 5], order=2^3)

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

# The display mode can be set to "power" mode
In [22]: GF.display("power"); a
Out[22]: GF([^3, 1, ^6, , ^3, -∞, , ^2, , ^6], order=2^3)

# Reset the display mode to the default
In [23]: GF.display(); a
Out[23]: GF([3, 1, 5, 2, 3, 0, 2, 4, 2, 5], order=2^3)
```

The `galois.GFArray.display` method can be called as a context manager.

```
# The original display mode
In [24]: print(a)
GF([3, 1, 5, 2, 3, 0, 2, 4, 2, 5], order=2^3)

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

In [26]: with GF.display("power"):
   ....:     print(a)
   ....:
GF([^3, 1, ^6, , ^3, -∞, , ^2, , ^6], order=2^3)
```

(continues on next page)

(continued from previous page)

```
# Returns to the original display mode
In [27]: print(a)
GF([3, 1, 5, 2, 3, 0, 2, 4, 2, 5], order=2^3)
```

3.3 Galois field array arithmetic

3.3.1 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)'>

# Create a random GF(7) array with 10 elements
In [3]: x = GF7.Random(10); x
Out[3]: GF([6, 0, 0, 4, 2, 4, 4, 6, 5, 3], 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([3, 1, 6, 3, 3, 2, 4, 2, 3, 2], order=7)

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

# Subtraction in the finite field
In [6]: x - y
Out[6]: GF([3, 6, 1, 1, 6, 2, 0, 4, 2, 1], order=7)

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

# Division in the finite field
In [8]: x / y
Out[8]: GF([2, 0, 0, 6, 3, 2, 1, 3, 4, 5], order=7)

In [9]: x // y
Out[9]: GF([2, 0, 0, 6, 3, 2, 1, 3, 4, 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
```

(continues on next page)

(continued from previous page)

```
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],
    [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)
```

3.3.2 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 $mathrmGF(2^8)$, 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([3, 3, 5, 5, 7, 3, 7, 1, 5, 0], order=2^3)

# Calculates a x "2" in the finite field
In [18]: a * GF8(2)
Out[18]: GF([6, 6, 1, 1, 5, 6, 5, 2, 1, 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 $mathrmGF(p)$, multiplication and scalar multiplication are equivalent.

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

In [21]: a = GF7.Random(10); a
Out[21]: GF([1, 3, 1, 0, 2, 6, 2, 0, 4, 4], order=7)

# Calculates a x "2" in the finite field
In [22]: a * GF7(2)
Out[22]: GF([2, 6, 2, 0, 4, 5, 4, 0, 1, 1], order=7)

# Calculates a + a
In [23]: a * 2
Out[23]: GF([2, 6, 2, 0, 4, 5, 4, 0, 1, 1], order=7)
```

3.3.3 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, 4, 6, 6, 1, 5, 1, 1, 2, 3], order=7)

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

3.3.4 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([2, 4, 6, 6, 1, 3, 4, 6, 3, 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([2, 4, 3, 3, 0, 1, 4, 3, 1, 4])

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

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

3.4 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.

3.4.1 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([931024088563052443, 29131431422693511503, 30880901481987118468,
    1552200098612073063 5, 2581797512086648177, 29247506552951728276,
    35368148452854141793, 22129952724144132621, 25452939337262120392,
    18875719434988970283], order=36893488147419103183)
```

(continues on next page)

(continued from previous page)

```
In [6]: b = GF.Random(10); b
Out[6]:
GF([7616305316913037192, 16737301754745749692, 26790891543631911747,
    29298507964573044743, 12824986279638728261, 8193750986730357991,
    13665300401550413334, 5709991981389047295, 29924706991992122589,
    35536844121068655627], order=36893488147419103183)

In [7]: a + b
Out[7]:
GF([8547329405476089635, 8975245030020158012, 20778304878199927032,
    7927020803274672195, 15406783791725376438, 547769392262983084,
    12139960706985451944, 27839944705533179916, 18484158181835139798,
    17519075408638522727], order=36893488147419103183)
```

3.4.2 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.meta_gf.DisplayContext at 0x7f285ae9a940>

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)
```

(continues on next page)

(continued from previous page)

```
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.meta_gf.DisplayContext at 0x7f285aeb2898>
```

CHAPTER

FOUR

PERFORMANCE TESTING

4.1 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.

4.1.1 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); print(GF.properties)
GF(31):
  characteristic: 31
  degree: 1
  order: 31
  irreducible_poly: Poly(x + 28, GF(31))
  is_primitive_poly: True
  primitive_element: GF(3, order=31)
  dtypes: ['uint8', 'uint16', 'uint32', 'int8', 'int16', 'int32', 'int64']
  ufunc_mode: 'jit-lookup'
  ufunc_target: 'cpu'

In [4]: def construct_arrays(GF, N):
   ...:     a = np.random.randint(1, GF.order, N, dtype=int)
   ...:     b = np.random.randint(1, GF.order, N, dtype=int)
   ...:     ga = a.view(GF)
   ...:     gb = b.view(GF)
   ...:     return a, b, ga, gb
   ...:

In [5]: N = int(10e3)

In [6]: a, b, ga, gb = construct_arrays(GF, N)
```

(continues on next page)

(continued from previous page)

```
In [7]: a
Out[7]: array([29, 20, 29, ..., 29, 22, 24])

In [8]: ga
Out[8]: GF([29, 20, 29, ..., 29, 22, 24], order=31)

In [9]: %timeit (a * b) % GF.order
88.2 µs ± 931 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)

In [10]: %timeit ga * gb
67.9 µs ± 425 ns 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 [15]: N = 10

In [16]: a, b, ga, gb = construct_arrays(GF, N)

In [17]: a
Out[17]: array([17, 22,  9, 11,  7, 14, 27, 16, 21, 30])

In [18]: ga
Out[18]: GF([17, 22,  9, 11,  7, 14, 27, 16, 21, 30], order=31)

In [19]: %timeit (a * b) % GF.order
1.32 µs ± 22.5 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)

In [20]: %timeit ga * gb
35.1 µs ± 879 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

This runtime discrepancy can be explained by the time numpy takes to perform the type conversion and view.

```
In [21]: %timeit a.astype(np.uint8).view(GF)
31.2 µs ± 5.53 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

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

```
In [22]: N = int(10e6)

In [23]: a, b, ga, gb = construct_arrays(GF, N)

In [24]: a
Out[24]: array([29,  9, 16, ..., 15, 24,  9])

In [25]: ga
Out[25]: GF([29,  9, 16, ..., 15, 24,  9], order=31)

In [26]: %timeit (a * b) % GF.order
109 ms ± 1.01 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

(continues on next page)

(continued from previous page)

```
In [27]: %timeit ga * gb
55.2 ms ± 1.18 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

4.1.2 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]: prime = galois.next_prime(2**21); prime
Out[3]: 2097169

In [4]: GF = galois.GF(prime); print(GF.properties)
GF(2097169):
  characteristic: 2097169
  degree: 1
  order: 2097169
  irreducible_poly: Poly(x + 2097122, GF(2097169))
  is_primitive_poly: True
  primitive_element: GF(47, order=2097169)
  dtypes: ['uint32', 'int32', 'int64']
  ufunc_mode: 'jit-calculate'
  ufunc_target: 'cpu'

In [5]: def construct_arrays(GF, N):
   ...:     a = np.random.randint(1, GF.order, N, dtype=int)
   ...:     b = np.random.randint(1, GF.order, N, dtype=int)
   ...:     ga = a.view(GF)
   ...:     gb = b.view(GF)
   ...:     return a, b, ga, gb
   ...:

In [6]: N = int(10e3)

In [7]: a, b, ga, gb = construct_arrays(GF, N)

In [8]: a
Out[8]: array([331469, 337477, 453485, ..., 186502, 794636, 535201])

In [9]: ga
Out[9]: GF([331469, 337477, 453485, ..., 186502, 794636, 535201], order=2097169)

In [10]: %timeit (a * b) % GF.order
88.3 µs ± 557 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

(continues on next page)

(continued from previous page)

```
In [11]: %timeit ga * gb
57.2 µs ± 749 ns 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 [12]: N = int(10e6)

In [13]: a, b, ga, gb = construct_arrays(GF, N)

In [14]: a
Out[14]: array([2090232, 2071169, 1463892, ..., 1382279, 1067677, 1901668])

In [15]: ga
Out[15]: GF([2090232, 2071169, 1463892, ..., 1382279, 1067677, 1901668], order=2097169)

In [16]: %timeit (a * b) % GF.order
109 ms ± 781 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)

In [17]: %timeit ga * gb
50.3 ms ± 619 µs per loop (mean ± std. dev. of 7 runs, 10 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_modes\n", "targets = GF.ufunc_targets\n", "targets.remove("cuda") # Can't test with a GPU on my machine\n", "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: add\n", "Target: cpu, Mode: jit-lookup\n", "  104 µs ± 1.57 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)\n", "Target: cpu, Mode: jit-calculate\n", "  71.3 µs ± 2.95 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)\n", "Target: parallel, Mode: jit-lookup\n", "  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-calculate\n", "  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

FIVE

DEVELOPMENT

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

5.1 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
```

5.2 Install for development with min dependencies

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

Listing 1: requirements-min.txt

```
numpy==1.17.3
numba==0.49
```

`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
```

5.3 Lint the package

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

Listing 2: requirements-lint.txt

```
pylint
```

Install the linter dependencies.

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

Run the linter.

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

5.4 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`.

Listing 3: requirements-test.txt

```
pytest
pytest-cov
```

Install the test dependencies.

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

Run the unit tests.

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

5.5 Build the documentation

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

Listing 4: requirements-doc.txt

```
sphinx>=3
recommonmark>=0.5
sphinx_rtd_theme>=0.5
readthedocs-sphinx-ext>=1.1
ipykernel
nbsphinx
pandoc
numpy
```

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

SIX

API REFERENCE V0.0.14

galois	A performant numpy extension for Galois fields.

6.1 galois

A performant numpy extension for Galois fields.

Class Factory Functions

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

6.1.1 galois.GF

galois.**GF**(*order*, *irreducible_poly=None*, *primitive_element=None*, *verify_irreducible=True*, *verify_primitive=True*, *mode='auto'*, *target='cpu'*)

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

The created class will be a subclass of *galois.GFArray* with metaclass *galois.GFMeta*. The *galois.GFArray* inheritance provides the numpy.ndarray functionality. The *galois.GFMeta* 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*, 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. Default is None which uses the Conway polynomial $C_{p,m}$ obtained from *galois.conway_poly()*.
- **primitive_element** (*int*, 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 $\mathrm{GF}(p)$ or its integer representation. The default is None which uses galois.primitive_root(p) for prime fields and galois.primitive_element(irreducible_poly) for extension fields.

- **verify_irreducible** (*bool, optional*) – Indicates whether to verify that the specified irreducible polynomial is in fact irreducible. The default is `True`. For large irreducible polynomials that are already known to be irreducible (and may take a long time to verify), this argument can be set to `False`.
- **verify_primitive** (*bool, optional*) – Indicates whether to verify that the specified primitive element is in fact a generator of the multiplicative group. The default is `True`.
- **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**16` will use the "jit-lookup" mode.
- **target** (*str, optional*) – The `target` keyword argument from `numba.vectorize()`, either `"cpu"`, `"parallel"`, or `"cuda"`.

Returns A new Galois field array class that is a subclass of *galois.GFArray* with *galois.GFMeta* metaclass.

Return type *galois.GFMeta*

Examples

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

```
# Construct a GF(2^m) class
In [5]: GF256 = galois.GF(2**8)

# Notice the irreducible polynomial is primitive
In [6]: 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 [7]: poly = GF256.irreducible_poly
```

Construct a Galois field specifying a specific irreducible polynomial.

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

In [9]: print(GF256_AES.properties)
GF(2^8):
  characteristic: 2
  degree: 8
  order: 256
```

(continues on next page)

(continued from previous page)

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

# Construct a GF(p) class
In [10]: GF571 = galois.GF(571); print(GF571.properties)
GF(571):
  characteristic: 571
  degree: 1
  order: 571
  irreducible_poly: Poly(x + 568, GF(571))
  is_primitive_poly: True
  primitive_element: GF(3, order=571)

# Construct a very large GF(2^m) class
In [11]: 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 [12]: GFp = galois.GF(36893488147419103183); print(GFp.properties)
GF(36893488147419103183):
  characteristic: 36893488147419103183
  degree: 1
  order: 36893488147419103183
  irreducible_poly: Poly(x + 36893488147419103180, GF(36893488147419103183))
  is_primitive_poly: True
  primitive_element: GF(3, order=36893488147419103183)
```

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

Abstract Classes

GFArray(array[, dtype, copy, order, ndmin])	Create an array over GF(p^m).
GFMeta(name, bases, namespace, **kwargs)	Defines a metaclass for all *galois.GFArray* classes.

6.1.2 galois.GFArray

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

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

The *galois.GFArray* 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.GFArray* 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 [74]: GF7 = galois.GF(7)

In [75]: print(GF7)
<class 'numpy.ndarray over GF(7)'>
```

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

```
In [76]: GF7([3,5,0,2,1])
Out[76]: GF([3, 5, 0, 2, 1], order=7)

In [77]: GF7.Random((2,5))
Out[77]:
GF([[3, 1, 3, 5, 3],
    [1, 0, 4, 4, 3]], order=7)
```

galois.GFArray is a subclass of `numpy.ndarray`. The *galois.GFArray* constructor has the same syntax as `numpy.array()`. The returned *galois.GFArray* 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.GFMeta.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.GFArray*

Examples

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

```
# Construct a GF(2^m) class
In [78]: GF256 = galois.GF(2**8); print(GF256)
<class 'numpy.ndarray over GF(2^8)'>

# Construct a GF(p) class
In [79]: GF571 = galois.GF(571); print(GF571)
<class 'numpy.ndarray over GF(571)'>

# Construct a very large GF(2^m) class
In [80]: GF2m = galois.GF(2**100); print(GF2m)
<class 'numpy.ndarray over GF(2^100)'>

# Construct a very large GF(p) class
In [81]: 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 [82]: GF256.dtypes
Out[82]:
[numpy.uint8,
 numpy.uint16,
 numpy.uint32,
 numpy.int16,
 numpy.int32,
 numpy.int64]

In [83]: GF571.dtypes
Out[83]: [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 [84]: GF2m.dtypes
Out[84]: [numpy.object_]

In [85]: GFp.dtypes
Out[85]: [numpy.object_]
```

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

```
In [86]: a = GF256.Random(10); a
Out[86]: GF([208,  73, 242,  95,   1, 183,  62, 249,  12, 136], order=2^8)

In [87]: a.dtype
Out[87]: dtype('uint8')
```

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

```
In [88]: a = GF256.Random(10, dtype=np.uint32); a
Out[88]: GF([224, 161,  92,  78, 221, 214,  48, 232, 169, 148], order=2^8)

In [89]: a.dtype
Out[89]: dtype('uint32')
```

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

```
# Construct a Galois field array from a list
In [90]: l = [142, 27, 92, 253, 103]; l
Out[90]: [142, 27, 92, 253, 103]

In [91]: GF256(l)
Out[91]: GF([142,  27,  92, 253, 103], order=2^8)

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

In [93]: GF256(l)
Out[93]: 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 [94]: a = x_np.view(GF256); a
Out[94]: GF([142,  27,  92, 253, 103], order=2^8)

# Changing `x_np` will change `a`
In [95]: x_np[0] = 0; x_np
Out[95]: array([  0,  27,  92, 253, 103])

In [96]: a
Out[96]: 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.GFMeta.dtypes*.

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

Return type *galois.GFArray*

Examples

```
In [97]: GF = galois.GF(31)

In [98]: GF.Elements()
Out[98]:
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.GFMeta.dtypes*.

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

Return type *galois.GFArray*

Examples

```
In [99]: GF = galois.GF(31)

In [100]: GF.Identity(4)
Out[100]:
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.GFMeta.dtypes*.

Returns A Galois field array of ones.

Return type *galois.GFArray*

Examples

```
In [101]: GF = galois.GF(31)

In [102]: GF.Ones((2,5))
Out[102]:
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.GFMeta.dtypes*.

Returns A Galois field array of random field elements.

Return type *galois.GFArray*

Examples

```
In [103]: GF = galois.GF(31)

In [104]: GF.Random((2,5))
Out[104]:
GF([[ 5, 20, 12, 10,  2],
    [30,  2, 13, 30,  2]], 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.GFMeta.dtypes*.

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

Return type *galois.GFArray*

Examples

```
In [105]: GF = galois.GF(31)

In [106]: GF.Range(10,20)
Out[106]: 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.GFArray*) – 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.GFMeta.dtypes*.

Returns The $m \times n$ Vandermonde matrix.

Return type *galois.GFArray*

Examples

```
In [107]: GF = galois.GF(2**3)

In [108]: a = GF.primitive_element
```

(continues on next page)

(continued from previous page)

```
In [109]: V = GF.Vandermonde(a, 7, 7)

In [110]: 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.GFMeta.dtypes*.

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

Return type *galois.GFArray*

Examples

```
In [111]: GF = galois.GF(2**6)

In [112]: vec = galois.GF2.Random((3,6)); vec
Out[112]:
GF([[1, 0, 1, 0, 0, 0],
    [1, 0, 1, 1, 1, 1],
    [0, 0, 1, 0, 0, 1]], order=2)

In [113]: a = GF.Vector(vec); a
Out[113]: GF([40, 47,  9], order=2^6)

In [114]: with GF.display("poly"):
   .....:     print(a)
   .....:
GF([^5 + ^3, ^5 + ^3 + ^2 +  + 1, ^3 + 1], order=2^6)

In [115]: a.vector()
Out[115]:
GF([[1, 0, 1, 0, 0, 0],
    [1, 0, 1, 1, 1, 1],
```

(continues on next page)

(continued from previous page)

```
    [0, 0, 1, 0, 0, 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.GFMeta.dtypes*.

Returns A Galois field array of zeros.

Return type *galois.GFArray*

Examples

```
In [116]: GF = galois.GF(31)

In [117]: GF.Zeros((2,5))
Out[117]:
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.GFArray* – The lower triangular matrix.
- *galois.GFArray* – The upper triangular matrix.

Examples

```
In [118]: GF = galois.GF(5)

# Not every square matrix has an LU decomposition
In [119]: A = GF([[2, 4, 4, 1], [3, 3, 1, 4], [4, 3, 4, 2], [4, 4, 3, 1]])

In [120]: L, U = A.lu_decompose()

In [121]: L
Out[121]:
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 [122]: U
Out[122]:
GF([[2, 4, 4, 1],
    [0, 2, 0, 0],
    [0, 0, 1, 0],
    [0, 0, 0, 4]], order=5)

# A = L U
In [123]: np.array_equal(A, L @ U)
Out[123]: True
```

`lup_decompose()`

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

Returns

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

Examples

```
In [124]: GF = galois.GF(5)

In [125]: A = GF([[1, 3, 2, 0], [3, 4, 2, 3], [0, 2, 1, 4], [4, 3, 3, 1]])

In [126]: L, U, P = A.lup_decompose()

In [127]: L
Out[127]:
GF([[1, 0, 0, 0],
    [0, 1, 0, 0],
    [3, 0, 1, 0],
    [4, 3, 2, 1]], order=5)

In [128]: U
Out[128]:
GF([[1, 3, 2, 0],
    [0, 2, 1, 4],
    [0, 0, 1, 3],
    [0, 0, 0, 3]], order=5)

In [129]: P
Out[129]:
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 [130]: np.array_equal(P @ A, L @ U)
Out[130]: 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.GFArray*

Examples

```
In [131]: GF = galois.GF(31)

In [132]: A = GF.Random((4,4)); A
Out[132]:
GF([[29, 29,  3, 17],
    [ 5,  8,  9, 13],
    [16, 20, 15,  8],
    [23, 16, 13, 18]], order=31)

In [133]: A.row_reduce()
Out[133]:
GF([[1, 0, 0, 0],
    [0, 1, 0, 0],
    [0, 0, 1, 0],
    [0, 0, 0, 1]], order=31)

In [134]: np.linalg.matrix_rank(A)
Out[134]: 4
```

One column is a linear combination of another.

```
In [135]: GF = galois.GF(31)

In [136]: A = GF.Random((4,4)); A
Out[136]:
GF([[ 7,  0, 11,  4],
    [11, 28, 28,  7],
    [22, 25, 29,  5],
    [10, 14,  3, 23]], order=31)
```

(continues on next page)

(continued from previous page)

```
In [137]: A[:,2] = A[:,1] * GF(17); A
Out[137]:
GF([[ 7,  0,  0,  4],
    [11, 28, 11,  7],
    [22, 25, 22,  5],
    [10, 14, 21, 23]], order=31)

In [138]: A.row_reduce()
Out[138]:
GF([[ 1,  0,  0,  0],
    [ 0,  1, 17,  0],
    [ 0,  0,  0,  1],
    [ 0,  0,  0,  0]], order=31)

In [139]: np.linalg.matrix_rank(A)
Out[139]: 3
```

One row is a linear combination of another.

```
In [140]: GF = galois.GF(31)

In [141]: A = GF.Random((4,4)); A
Out[141]:
GF([[28,  4, 15, 17],
    [ 9,  6,  1,  6],
    [ 2,  2,  7, 19],
    [ 7,  6, 27,  6]], order=31)

In [142]: A[3,:] = A[2,:] * GF(8); A
Out[142]:
GF([[28,  4, 15, 17],
    [ 9,  6,  1,  6],
    [ 2,  2,  7, 19],
    [16, 16, 25, 28]], order=31)

In [143]: A.row_reduce()
Out[143]:
GF([[ 1,  0,  0, 10],
    [ 0,  1,  0, 14],
    [ 0,  0,  1, 18],
    [ 0,  0,  0,  0]], order=31)

In [144]: np.linalg.matrix_rank(A)
Out[144]: 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.GFMeta.dtypes*.

Returns A Galois field array of length-m vectors over $\mathrm{GF}(p)$.

Return type *galois.GFArray*

Examples

```
In [145]: GF = galois.GF(2**6)

In [146]: a = GF.Random(3); a
Out[146]: GF([ 1, 21,  9], order=2^6)

In [147]: vec = a.vector(); vec
Out[147]:
GF([[0, 0, 0, 0, 0, 1],
    [0, 1, 0, 1, 0, 1],
    [0, 0, 1, 0, 0, 1]], order=2)

In [148]: GF.Vector(vec)
Out[148]: GF([ 1, 21,  9], order=2^6)
```

6.1.3 galois.GFMeta

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

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

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

Constructors

—

Methods

compile(mode[, target])	Recompile the just-in-time compiled numba ufuncs with a new calculation mode or target.
display([mode])	Sets the display mode for all Galois field arrays of this type.

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 $\mathrm{GF}(p^m)$.

continues on next page

Table 8 – continued from previous page

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 formmatted 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.
ufunc_target	The numba target for the JIT-compiled ufuncs, either "cpu", "parallel", or "cuda".
ufunc_targets	All supported ufunc targets for this Galois field array class.

compile(*mode*, *target='cpu'*)

Recompile the just-in-time compiled numba ufuncs with a new calculation mode or target.

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.GFMeta.ufunc_modes*.
- **target** (*str, optional*) – The target keyword argument from numba.vectorize, either "cpu", "parallel", or "cuda". The default is "cpu". For extremely large fields the only supported target is "cpu" (which doesn't use numba it uses pure python to calculate the field arithmetic). The list of valid targets for this field is in *galois.GFMeta.ufunc_targets*.

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

In [150]: a = GF.Random(); a
Out[150]: GF(23, order=2^8)

# Change the display mode going forward
In [151]: GF.display("poly"); a
Out[151]: GF(^4 + ^2 +  + 1, order=2^8)

In [152]: GF.display("power"); a
Out[152]: GF(^129, order=2^8)

# Reset to the default display mode
In [153]: GF.display(); a
Out[153]: GF(23, 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 [154]: GF = galois.GF(2**8)

In [155]: 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 [156]: a = GF.Random(); a
Out[156]: GF(221, order=2^8)

In [157]: with GF.display("poly"):
   .....:     print(a)
   .....:
GF(^7 + ^6 + ^4 + ^3 + ^2 + 1, order=2^8)

In [158]: with GF.display("power"):
   .....:     print(a)
   .....:
GF(^204, order=2^8)
```

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

```
In [159]: GF = galois.GF(2**8, irreducible_poly=galois.Poly.Degrees([8,4,3,1,
↪0]))

In [160]: 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 [161]: a = GF.Random(); a
Out[161]: GF(228, order=2^8)

In [162]: with GF.display("poly"):
   .....:     print(a)
   .....:
GF(x^7 + x^6 + x^5 + x^2, order=2^8)

In [163]: with GF.display("power"):
   .....:     print(a)
   .....:
GF(^35, order=2^8)
```

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

In [165]: GF.characteristic
Out[165]: 2

In [166]: a = GF.Random(); a
Out[166]: GF(106, order=2^8)

In [167]: a * GF.characteristic
Out[167]: GF(0, order=2^8)
```

```
In [168]: GF = galois.GF(31)

In [169]: GF.characteristic
Out[169]: 31

In [170]: a = GF.Random(); a
Out[170]: GF(29, order=31)

In [171]: a * GF.characteristic
```

(continues on next page)

(continued from previous page)

```
Out[171]: GF(0, order=31)
```

Type int

property default_ufunc_mode

The default ufunc arithmetic mode for this Galois field.

Examples

```
In [172]: galois.GF(2).default_ufunc_mode
Out[172]: 'jit-calculate'

In [173]: galois.GF(2**8).default_ufunc_mode
Out[173]: 'jit-lookup'

In [174]: galois.GF(31).default_ufunc_mode
Out[174]: 'jit-lookup'

In [175]: galois.GF(2**100).default_ufunc_mode
Out[175]: '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 [176]: galois.GF(2).degree
Out[176]: 1

In [177]: galois.GF(2**8).degree
Out[177]: 8

In [178]: galois.GF(31).degree
Out[178]: 1

# galois.GF(7**5).degree
```

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

In [180]: 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 [181]: a = GF.Random(); a
Out[181]: GF(185, order=2^8)

In [182]: with GF.display("poly"):
   .....:     print(a)
   .....:
GF(^7 + ^5 + ^4 + ^3 + 1, order=2^8)

In [183]: with GF.display("power"):
   .....:     print(a)
   .....:
GF(^60, order=2^8)
```

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

```
In [184]: GF = galois.GF(2**8, irreducible_poly=galois.Poly.Degrees([8,4,3,1,
 ↪0]))

In [185]: 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 [186]: a = GF.Random(); a
Out[186]: GF(131, order=2^8)

In [187]: with GF.display("poly"):
   .....:     print(a)
   .....:
GF(x^7 + x + 1, order=2^8)

In [188]: with GF.display("power"):
   .....:     print(a)
   .....:
GF(^80, order=2^8)
```

Type str

property dtypes

List of valid integer `numpy.dtype` objects that are compatible with this Galois field. Valid data types are signed and unsinged integers that can represent decimal values in $[0, p^m)$.

Examples

```
In [189]: galois.GF(2).dtypes
Out[189]:
[numpy.uint8,
 numpy.uint16,
 numpy.uint32,
 numpy.int8,
 numpy.int16,
 numpy.int32,
 numpy.int64]

In [190]: galois.GF(2**8).dtypes
Out[190]:
[numpy.uint8,
 numpy.uint16,
 numpy.uint32,
 numpy.int16,
 numpy.int32,
 numpy.int64]

In [191]: galois.GF(31).dtypes
Out[191]:
[numpy.uint8,
 numpy.uint16,
 numpy.uint32,
 numpy.int8,
 numpy.int16,
 numpy.int32,
 numpy.int64]

# galois.GF(7**5).dtypes
```

For field's with orders that cannot be represented by `numpy.int64`, the only valid dtype is `numpy.object_`.

```
In [192]: galois.GF(2**100).dtypes
Out[192]: [numpy.object_]

In [193]: galois.GF(36893488147419103183).dtypes
Out[193]: [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 [194]: galois.GF(2).irreducible_poly
Out[194]: Poly(x + 1, GF(2))

In [195]: galois.GF(2**8).irreducible_poly
Out[195]: Poly(x^8 + x^4 + x^3 + x^2 + 1, GF(2))

In [196]: galois.GF(31).irreducible_poly
Out[196]: Poly(x + 28, GF(31))

# galois.GF(7**5).irreducible_poly
```

Type *galois.Poly*

property is_extension_field

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

Examples

```
In [197]: galois.GF(2).is_extension_field
Out[197]: False

In [198]: galois.GF(2**8).is_extension_field
Out[198]: True

In [199]: galois.GF(31).is_extension_field
Out[199]: False

# galois.GF(7**5).is_extension_field
```

Type bool

property is_prime_field

Indicates if the field's order is prime.

Examples

```
In [200]: galois.GF(2).is_prime_field
Out[200]: True

In [201]: galois.GF(2**8).is_prime_field
Out[201]: False

In [202]: galois.GF(31).is_prime_field
Out[202]: True

# galois.GF(7**5).is_prime_field
```

Type bool

property is_primitive_poly

Indicates whether the *irreducible_poly* is a primitive polynomial.

Examples

```
In [203]: GF = galois.GF(2**8)

In [204]: GF.irreducible_poly
Out[204]: Poly(x^8 + x^4 + x^3 + x^2 + 1, GF(2))

In [205]: GF.primitive_element
Out[205]: GF(2, order=2^8)

# The irreducible polynomial is a primitive polynomial is the primitive element␣
↪is a root
In [206]: GF.irreducible_poly(GF.primitive_element, field=GF)
Out[206]: GF(0, order=2^8)

In [207]: GF.is_primitive_poly
Out[207]: True
```

```
# Field used in AES
In [208]: GF = galois.GF(2**8, irreducible_poly=galois.Poly.Degrees([8,4,3,1,
↪0]))

In [209]: GF.irreducible_poly
Out[209]: Poly(x^8 + x^4 + x^3 + x + 1, GF(2))

In [210]: GF.primitive_element
Out[210]: GF(3, order=2^8)

# The irreducible polynomial is a primitive polynomial is the primitive element␣
↪is a root
In [211]: GF.irreducible_poly(GF.primitive_element, field=GF)
Out[211]: GF(6, order=2^8)

In [212]: GF.is_primitive_poly
Out[212]: False
```

Type bool

property name

The Galois field name.

Examples

```
In [213]: galois.GF(2).name
Out[213]: 'GF(2)'
```

(continues on next page)

(continued from previous page)

```
In [214]: galois.GF(2**8).name
Out[214]: 'GF(2^8)'

In [215]: galois.GF(31).name
Out[215]: 'GF(31)'

# galois.GF(7**5).name
```

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 [216]: galois.GF(2).order
Out[216]: 2

In [217]: galois.GF(2**8).order
Out[217]: 256

In [218]: galois.GF(31).order
Out[218]: 31

# galois.GF(7**5).order
```

Type int

property prime_subfield

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

Examples

```
In [219]: print(galois.GF(2).prime_subfield.properties)
GF(2):
  characteristic: 2
  degree: 1
  order: 2
  irreducible_poly: Poly(x + 1, GF(2))
  is_primitive_poly: True
  primitive_element: GF(1, order=2)

In [220]: print(galois.GF(2**8).prime_subfield.properties)
GF(2):
  characteristic: 2
  degree: 1
  order: 2
  irreducible_poly: Poly(x + 1, GF(2))
```

(continues on next page)

(continued from previous page)

```
  is_primitive_poly: True
  primitive_element: GF(1, order=2)

In [221]: print(galois.GF(31).prime_subfield.properties)
GF(31):
  characteristic: 31
  degree: 1
  order: 31
  irreducible_poly: Poly(x + 28, GF(31))
  is_primitive_poly: True
  primitive_element: GF(3, order=31)

# print(galois.GF(7**5).prime_subfield.properties)
```

Type *galois.GFMeta*

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 $\mathrm{GF}(p^m) = \{0, 1, \alpha^1, \alpha^2, \ldots, \alpha^{p^m-2}\}$.

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

Examples

```
In [222]: galois.GF(2).primitive_element
Out[222]: GF(1, order=2)

In [223]: galois.GF(2**8).primitive_element
Out[223]: GF(2, order=2^8)

In [224]: galois.GF(31).primitive_element
Out[224]: GF(3, order=31)

# galois.GF(7**5).primitive_element
```

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 $\mathrm{GF}(p^m) = \{0, 1, \alpha^1, \alpha^2, \ldots, \alpha^{p^m-2}\}$.

Examples

```
In [225]: galois.GF(2).primitive_elements
Out[225]: GF([1], order=2)

In [226]: galois.GF(2**8).primitive_elements
Out[226]:
GF([  2,   4,   6,   9,  13,  14,  16,  18,  19,  20,  22,  24,  25,  27,
```

(continues on next page)

(continued from previous page)

```
     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 [227]: galois.GF(31).primitive_elements
Out[227]: GF([ 3, 11, 12, 13, 17, 21, 22, 24], order=31)

# galois.GF(7**5).primitive_elements
```

Type int

property properties

A formmatted string displaying relevant properties of the Galois field.

Examples

```
In [228]: print(galois.GF(2).properties)
GF(2):
  characteristic: 2
  degree: 1
  order: 2
  irreducible_poly: Poly(x + 1, GF(2))
  is_primitive_poly: True
  primitive_element: GF(1, order=2)

In [229]: print(galois.GF(2**8).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 [230]: print(galois.GF(31).properties)
GF(31):
  characteristic: 31
  degree: 1
  order: 31
  irreducible_poly: Poly(x + 28, GF(31))
  is_primitive_poly: True
  primitive_element: GF(3, order=31)

# print(galois.GF(7**5).properties)
```

Type str

property ufunc_mode

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

Examples

```
In [231]: galois.GF(2).ufunc_mode
Out[231]: 'jit-calculate'

In [232]: galois.GF(2**8).ufunc_mode
Out[232]: 'jit-lookup'

In [233]: galois.GF(31).ufunc_mode
Out[233]: 'jit-lookup'

# galois.GF(7**5).ufunc_mode
```

Type str

property ufunc_modes

All supported ufunc modes for this Galois field array class.

Examples

```
In [234]: galois.GF(2).ufunc_modes
Out[234]: ['jit-calculate']

In [235]: galois.GF(2**8).ufunc_modes
Out[235]: ['jit-lookup', 'jit-calculate']

In [236]: galois.GF(31).ufunc_modes
Out[236]: ['jit-lookup', 'jit-calculate']

In [237]: galois.GF(2**100).ufunc_modes
Out[237]: ['python-calculate']
```

Type list

property ufunc_target

The numba target for the JIT-compiled ufuncs, either "cpu", "parallel", or "cuda".

Examples

```
In [238]: galois.GF(2).ufunc_target
Out[238]: 'cpu'
```

(continues on next page)

(continued from previous page)

```
In [239]: galois.GF(2**8).ufunc_target
Out[239]: 'cpu'

In [240]: galois.GF(31).ufunc_target
Out[240]: 'cpu'

# galois.GF(7**5).ufunc_target
```

Type str

property ufunc_targets

All supported ufunc targets for this Galois field array class.

Examples

```
In [241]: galois.GF(2).ufunc_targets
Out[241]: ['cpu', 'parallel', 'cuda']

In [242]: galois.GF(2**8).ufunc_targets
Out[242]: ['cpu', 'parallel', 'cuda']

In [243]: galois.GF(31).ufunc_targets
Out[243]: ['cpu', 'parallel', 'cuda']

In [244]: galois.GF(2**100).ufunc_targets
Out[244]: ['cpu']
```

Type list

Classes

GF2(array[, dtype, copy, order, ndmin])	A pre-created Galois field array class for $\mathrm{GF}(2)$.
Poly(coeffs[, field, order])	Create a polynomial $f(x)$ over $\mathrm{GF}(p^m)$.

6.1.4 galois.GF2

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

A pre-created Galois field array class for $\mathrm{GF}(2)$.

This class is a subclass of *galois.GFArray* and has metaclass *galois.GFMeta*.

Examples

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

```
In [13]: print(galois.GF2)
<class 'numpy.ndarray over GF(2)'>
```

(continues on next page)

(continued from previous page)

```
In [14]: GF2 = galois.GF(2); print(GF2)
<class 'numpy.ndarray over GF(2)'>

In [15]: GF2 is galois.GF2
Out[15]: True
```

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

```
# View a summary of the field's properties
In [16]: print(galois.GF2.properties)
GF(2):
  characteristic: 2
  degree: 1
  order: 2
  irreducible_poly: Poly(x + 1, GF(2))
  is_primitive_poly: True
  primitive_element: GF(1, order=2)

# Or access each attribute individually
In [17]: galois.GF2.irreducible_poly
Out[17]: Poly(x + 1, GF(2))

In [18]: galois.GF2.is_prime_field
Out[18]: True
```

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

```
# Construct a Galois field array from an iterable
In [19]: galois.GF2([1,0,1,1,0,0,0,1])
Out[19]: GF([1, 0, 1, 1, 0, 0, 0, 1], order=2)

# Or an iterable of iterables
In [20]: galois.GF2([[1,0],[1,1]])
Out[20]:
GF([[1, 0],
    [1, 1]], order=2)

# Or a single integer
In [21]: galois.GF2(1)
Out[21]: 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.GFMeta.dtypes*.

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

Return type *galois.GFArray*

Examples

```
In [22]: GF = galois.GF(31)

In [23]: GF.Elements()
Out[23]:
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.GFMeta.dtypes*.

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

Return type *galois.GFArray*

Examples

```
In [24]: GF = galois.GF(31)

In [25]: GF.Identity(4)
Out[25]:
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.GFMeta.dtypes*.

Returns A Galois field array of ones.

Return type *galois.GFArray*

Examples

```
In [26]: GF = galois.GF(31)

In [27]: GF.Ones((2,5))
Out[27]:
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,)`, repre-

sents 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.GFMeta.dtypes*.

Returns A Galois field array of random field elements.

Return type *galois.GFArray*

Examples

```
In [28]: GF = galois.GF(31)

In [29]: GF.Random((2,5))
Out[29]:
GF([[22, 17, 30,  4, 28],
    [22, 11,  0, 19,  1]], 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.GFMeta.dtypes*.

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

Return type *galois.GFArray*

Examples

```
In [30]: GF = galois.GF(31)

In [31]: GF.Range(10,20)
Out[31]: 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.GFArray*) – 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.GFMeta.dtypes*.

Returns The $m \times n$ Vandermonde matrix.

Return type *galois.GFArray*

Examples

```
In [32]: GF = galois.GF(2**3)

In [33]: a = GF.primitive_element

In [34]: V = GF.Vandermonde(a, 7, 7)

In [35]: 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.GFMeta.dtypes*.

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

Return type *galois.GFArray*

Examples

```
In [36]: GF = galois.GF(2**6)

In [37]: vec = galois.GF2.Random((3,6)); vec
Out[37]:
GF([[1, 1, 0, 0, 1, 0],
    [0, 1, 0, 1, 1, 0],
```

(continues on next page)

(continued from previous page)

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

In [38]: a = GF.Vector(vec); a
Out[38]: GF([50, 22, 52], order=2^6)

In [39]: with GF.display("poly"):
   ....:     print(a)
   ....:
GF([^5 + ^4 + , ^4 + ^2 + , ^5 + ^4 + ^2], order=2^6)

In [40]: a.vector()
Out[40]:
GF([[1, 1, 0, 0, 1, 0],
    [0, 1, 0, 1, 1, 0],
    [1, 1, 0, 1, 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.GFMeta.dtypes*.

Returns A Galois field array of zeros.

Return type *galois.GFArray*

Examples

```
In [41]: GF = galois.GF(31)

In [42]: GF.Zeros((2,5))
Out[42]:
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.GFArray* – The lower triangular matrix.
- *galois.GFArray* – The upper triangular matrix.

Examples

```
In [43]: GF = galois.GF(5)

# Not every square matrix has an LU decomposition
In [44]: A = GF([[2, 4, 4, 1], [3, 3, 1, 4], [4, 3, 4, 2], [4, 4, 3, 1]])

In [45]: L, U = A.lu_decompose()

In [46]: L
Out[46]:
GF([[1, 0, 0, 0],
    [4, 1, 0, 0],
    [2, 0, 1, 0],
    [2, 3, 0, 1]], order=5)

In [47]: U
Out[47]:
GF([[2, 4, 4, 1],
    [0, 2, 0, 0],
    [0, 0, 1, 0],
    [0, 0, 0, 4]], order=5)

# A = L U
In [48]: np.array_equal(A, L @ U)
Out[48]: True
```

lup_decompose()

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

Returns

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

Examples

```
In [49]: GF = galois.GF(5)

In [50]: A = GF([[1, 3, 2, 0], [3, 4, 2, 3], [0, 2, 1, 4], [4, 3, 3, 1]])

In [51]: L, U, P = A.lup_decompose()

In [52]: L
Out[52]:
GF([[1, 0, 0, 0],
    [0, 1, 0, 0],
    [3, 0, 1, 0],
    [4, 3, 2, 1]], order=5)

In [53]: U
Out[53]:
```

(continues on next page)

(continued from previous page)

```
GF([[1, 3, 2, 0],
    [0, 2, 1, 4],
    [0, 0, 1, 3],
    [0, 0, 0, 3]], order=5)

In [54]: P
Out[54]:
GF([[1, 0, 0, 0],
    [0, 0, 1, 0],
    [0, 1, 0, 0],
    [0, 0, 0, 1]], order=5)

# P A = L U
In [55]: np.array_equal(P @ A, L @ U)
Out[55]: 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.GFArray*

Examples

```
In [56]: GF = galois.GF(31)

In [57]: A = GF.Random((4,4)); A
Out[57]:
GF([[12, 15, 15, 13],
    [14, 24, 27,  2],
    [ 6,  5, 14,  3],
    [20, 20, 24, 12]], order=31)

In [58]: A.row_reduce()
Out[58]:
GF([[1, 0, 0, 0],
    [0, 1, 0, 0],
    [0, 0, 1, 0],
    [0, 0, 0, 1]], order=31)
```

(continues on next page)

(continued from previous page)

```
In [59]: np.linalg.matrix_rank(A)
Out[59]: 4
```

One column is a linear combination of another.

```
In [60]: GF = galois.GF(31)

In [61]: A = GF.Random((4,4)); A
Out[61]:
GF([[20,  1,  3,  2],
    [ 4,  3, 18, 13],
    [13,  5,  6, 19],
    [ 6, 21, 19, 14]], order=31)

In [62]: A[:,2] = A[:,1] * GF(17); A
Out[62]:
GF([[20,  1, 17,  2],
    [ 4,  3, 20, 13],
    [13,  5, 23, 19],
    [ 6, 21, 16, 14]], order=31)

In [63]: A.row_reduce()
Out[63]:
GF([[ 1,  0,  0,  0],
    [ 0,  1, 17,  0],
    [ 0,  0,  0,  1],
    [ 0,  0,  0,  0]], order=31)

In [64]: np.linalg.matrix_rank(A)
Out[64]: 3
```

One row is a linear combination of another.

```
In [65]: GF = galois.GF(31)

In [66]: A = GF.Random((4,4)); A
Out[66]:
GF([[17, 18, 24, 14],
    [26, 24,  2,  9],
    [ 9, 12, 13, 24],
    [25,  5,  4,  0]], order=31)

In [67]: A[3,:] = A[2,:] * GF(8); A
Out[67]:
GF([[17, 18, 24, 14],
    [26, 24,  2,  9],
    [ 9, 12, 13, 24],
    [10,  3, 11,  6]], order=31)

In [68]: A.row_reduce()
Out[68]:
GF([[ 1,  0,  0, 17],
```

(continues on next page)

(continued from previous page)

```
    [ 0,  1,  0,  6],
    [ 0,  0,  1,  6],
    [ 0,  0,  0,  0]], order=31)

In [69]: np.linalg.matrix_rank(A)
Out[69]: 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.GFMeta.dtypes*.

Returns A Galois field array of length-m vectors over $\mathrm{GF}(p)$.

Return type *galois.GFArray*

Examples

```
In [70]: GF = galois.GF(2**6)

In [71]: a = GF.Random(3); a
Out[71]: GF([46, 56, 44], order=2^6)

In [72]: vec = a.vector(); vec
Out[72]:
GF([[1, 0, 1, 1, 1, 0],
    [1, 1, 1, 0, 0, 0],
    [1, 0, 1, 1, 0, 0]], order=2)

In [73]: GF.Vector(vec)
Out[73]: GF([46, 56, 44], order=2^6)
```

6.1.5 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*) – List of polynomial coefficients $\{a_d, a_{d-1}, \ldots, a_1, a_0\}$ with type *galois.GFArray*, `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.GFArray`, *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 [245]: galois.Poly([1,0,1,1])
Out[245]: Poly(x^3 + x + 1, GF(2))

In [246]: galois.Poly.Degrees([3,1,0])
Out[246]: Poly(x^3 + x + 1, GF(2))
```

Create a polynomial over $\mathrm{GF}(2^8)$.

```
In [247]: GF = galois.GF(2**8)

In [248]: galois.Poly([124,0,223,0,0,15], field=GF)
Out[248]: Poly(124x^5 + 223x^3 + 15, GF(2^8))

# Alternate way of constructing the same polynomial
In [249]: galois.Poly.Degrees([5,3,0], coeffs=[124,223,15], field=GF)
Out[249]: Poly(124x^5 + 223x^3 + 15, GF(2^8))
```

Polynomial arithmetic using binary operators.

```
In [250]: a = galois.Poly([117,0,63,37], field=GF); a
Out[250]: Poly(117x^3 + 63x + 37, GF(2^8))

In [251]: b = galois.Poly([224,0,21], field=GF); b
Out[251]: Poly(224x^2 + 21, GF(2^8))

In [252]: a + b
Out[252]: Poly(117x^3 + 224x^2 + 63x + 48, GF(2^8))

In [253]: a - b
Out[253]: Poly(117x^3 + 224x^2 + 63x + 48, GF(2^8))

# Compute the quotient of the polynomial division
In [254]: a / b
Out[254]: Poly(202x, GF(2^8))

# True division and floor division are equivalent
In [255]: a / b == a // b
Out[255]: True

# Compute the remainder of the polynomial division
In [256]: a % b
Out[256]: Poly(198x + 37, GF(2^8))
```

(continues on next page)

(continued from previous page)

```
# Compute both the quotient and remainder in one pass
In [257]: divmod(a, b)
Out[257]: (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 $\mathrm{GF}(p^m)$.
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.
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.

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.GFArray, 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 [258]: galois.Poly.Degrees([3,1,0])
Out[258]: Poly(x^3 + x + 1, GF(2))
```

Construct a polynomial over $\mathrm{GF}(2^8)$ by specifying the degrees with non-zero coefficients.

```
In [259]: GF = galois.GF(2**8)

In [260]: galois.Poly.Degrees([3,1,0], coeffs=[251,73,185], field=GF)
Out[260]: 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.GFArray, 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 [261]: galois.Poly.Identity()
Out[261]: Poly(x, GF(2))
```

Construct the identity polynomial over $\mathrm{GF}(2^8)$.

```
In [262]: GF = galois.GF(2**8)

In [263]: galois.Poly.Identity(field=GF)
Out[263]: 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.GFArray, 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 [264]: galois.Poly.Integer(5)
Out[264]: Poly(x^2 + 1, GF(2))
```

Construct a polynomial over $\mathrm{GF}(2^8)$ from its integer representation.

```
In [265]: GF = galois.GF(2**8)

In [266]: galois.Poly.Integer(13*256**3 + 117, field=GF)
Out[266]: 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.GFArray, 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 [267]: galois.Poly.One()
Out[267]: Poly(1, GF(2))
```

Construct the one polynomial over $\mathrm{GF}(2^8)$.

```
In [268]: GF = galois.GF(2**8)

In [269]: galois.Poly.One(field=GF)
Out[269]: 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.GFArray, 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 [270]: galois.Poly.Random(5)
Out[270]: Poly(x^5 + x^3 + x^2 + 1, GF(2))
```

Construct a random degree-5 polynomial over $\mathrm{GF}(2^8)$.

```
In [271]: GF = galois.GF(2**8)

In [272]: galois.Poly.Random(5, field=GF)
Out[272]: Poly(5x^5 + 19x^4 + 151x^3 + 180x^2 + 117x + 14, 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.GFArray, 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 [273]: roots = [0, 0, 1]

In [274]: p = galois.Poly.Roots(roots); p
Out[274]: Poly(x^3 + x^2, GF(2))

In [275]: p(roots)
Out[275]: GF([0, 0, 0], order=2)
```

Construct a polynomial over $\mathrm{GF}(2^8)$ from a list of its roots.

```
In [276]: GF = galois.GF(2**8)

In [277]: roots = [121, 198, 225]

In [278]: p = galois.Poly.Roots(roots, field=GF); p
Out[278]: Poly(x^3 + 94x^2 + 174x + 89, GF(2^8))

In [279]: p(roots)
Out[279]: GF([0, 0, 0], order=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.GFArray, 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 [280]: galois.Poly.Zero()
Out[280]: Poly(0, GF(2))
```

Construct the zero polynomial over $\mathrm{GF}(2^8)$.

```
In [281]: GF = galois.GF(2**8)

In [282]: galois.Poly.Zero(field=GF)
Out[282]: 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 [283]: p = galois.Poly.Random(7); p
Out[283]: Poly(x^7 + x^4 + x^3 + x, GF(2))

In [284]: p.derivative()
Out[284]: 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 [285]: p.derivative(2)
Out[285]: Poly(0, GF(2))
```

Compute the derivatives of a polynomial over $\mathrm{GF}(7)$.

```
In [286]: GF = galois.GF(7)

In [287]: p = galois.Poly.Random(11, field=GF); p
Out[287]: Poly(5x^11 + 6x^10 + 5x^9 + 4x^8 + 6x^7 + x^6 + x^5 + x^4 + 4x^3 + 2x^
↪2 + 2x + 6, GF(7))

In [288]: p.derivative()
Out[288]: Poly(6x^10 + 4x^9 + 3x^8 + 4x^7 + 6x^5 + 5x^4 + 4x^3 + 5x^2 + 4x + 2,␣
↪GF(7))

In [289]: p.derivative(2)
Out[289]: Poly(4x^9 + x^8 + 3x^7 + 2x^4 + 6x^3 + 5x^2 + 3x + 4, GF(7))

In [290]: p.derivative(3)
Out[290]: Poly(x^8 + x^7 + x^3 + 4x^2 + 3x + 3, GF(7))

# k derivatives of a polynomial where k is the Galois field's characteristic␣
↪will always result in 0
In [291]: p.derivative(7)
Out[291]: Poly(0, GF(2))
```

Compute the derivatives of a polynomial over $\mathrm{GF}(2^8)$.

```
In [292]: GF = galois.GF(2**8)

In [293]: p = galois.Poly.Random(7, field=GF); p
Out[293]: Poly(73x^7 + 54x^6 + 241x^5 + 101x^4 + 131x^3 + 5x^2 + 188x + 98,␣
↪GF(2^8))

In [294]: p.derivative()
Out[294]: Poly(73x^6 + 241x^4 + 131x^2 + 188, GF(2^8))
```

(continues on next page)

(continued from previous page)

```
# k derivatives of a polynomial where k is the Galois field's characteristic␣
↪will always result in 0
In [295]: p.derivative(2)
Out[295]: 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.GFArray* – 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 [296]: p = galois.Poly.Roots([0,]*7 + [1,]*13); p
Out[296]: Poly(x^20 + x^19 + x^16 + x^15 + x^12 + x^11 + x^8 + x^7, GF(2))

In [297]: p.roots()
Out[297]: GF([0, 1], order=2)

In [298]: p.roots(multiplicity=True)
Out[298]: (GF([0, 1], order=2), array([ 7, 13]))
```

Find the roots of a polynomial over $\mathrm{GF}(2^8)$.

```
In [299]: GF = galois.GF(2**8)

In [300]: p = galois.Poly.Roots([18,]*7 + [155,]*13 + [227,]*9, field=GF); p
Out[300]: 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 [301]: p.roots()
Out[301]: GF([ 18, 155, 227], order=2^8)

In [302]: p.roots(multiplicity=True)
Out[302]: (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 [303]: GF = galois.GF(7)

In [304]: p = galois.Poly([3, 0, 5, 2], field=GF)

In [305]: p.coeffs
Out[305]: GF([3, 0, 5, 2], order=7)
```

Type *galois.GFArray*

property degree

The degree of the polynomial, i.e. the highest degree with non-zero coefficient.

Examples

```
In [306]: GF = galois.GF(7)

In [307]: p = galois.Poly([3, 0, 5, 2], field=GF)

In [308]: p.degree
Out[308]: 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 [309]: GF = galois.GF(7)

In [310]: p = galois.Poly([3, 0, 5, 2], field=GF)

In [311]: p.degrees
Out[311]: array([3, 2, 1, 0])
```

Type numpy.ndarray

property field

The Galois field array class to which the coefficients belong.

Examples

```
In [312]: a = galois.Poly.Random(5); a
Out[312]: Poly(x^5, GF(2))

In [313]: a.field
Out[313]: <class 'numpy.ndarray over GF(2)'>

In [314]: b = galois.Poly.Random(5, field=galois.GF(2**8)); b
Out[314]: Poly(46x^5 + 56x^4 + 20x^3 + 9x^2 + 203x + 142, GF(2^8))

In [315]: b.field
Out[315]: <class 'numpy.ndarray over GF(2^8)'>
```

Type *galois.GFMeta*

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 [316]: GF = galois.GF(7)

In [317]: p = galois.Poly([3, 0, 5, 2], field=GF)

In [318]: p.integer
Out[318]: 1066

In [319]: p.integer == 3*7**3 + 5*7**1 + 2*7**0
Out[319]: 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 [320]: GF = galois.GF(7)

In [321]: p = galois.Poly([3, 0, 5, 2], field=GF)

In [322]: p.nonzero_coeffs
Out[322]: GF([3, 5, 2], order=7)
```

Type *galois.GFArray*

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 [323]: GF = galois.GF(7)

In [324]: p = galois.Poly([3, 0, 5, 2], field=GF)

In [325]: p.nonzero_degrees
Out[325]: array([3, 1, 0])
```

Type numpy.ndarray

Functions

carmichael(n)	Finds the smallest positive integer m such that $a^m \equiv 1 (\text{mod } n)$ for every integer a in $1 \leq a < n$ that is coprime to n.
conway_poly(p, n)	Returns the degree-n Conway polynomial $C_{p,n}$ over $\text{GF}(p)$.
crt(a, m)	Solves the simultaneous system of congruences for x.
euler_totient(n)	Counts the positive integers (totatives) in $1 \leq k < n$ that are relatively prime to n, i.e. $gcd(n, k) = 1$.
fermat_primality_test(n)	Probabilistic primality test of n.
gcd(a, b)	Finds the integer multiplicands of a and b such that $ax + by = \text{gcd}(a, b)$.
is_cyclic(n)	Determines whether the multiplicative group $\mathbb{Z}_n^\times$ is cyclic.
is_irreducible(poly)	Checks whether the polynomial $f(x)$ over $\text{GF}(p)$ is irreducible.
is_monic(poly)	Determines whether the polynomial is monic, i.e. having leading coefficient equal to 1.
is_prime(n)	Determines if n is prime.
is_primitive(poly)	Checks whether the polynomial $f(x)$ over $\text{GF}(p)$ is primitive.
is_primitive_element(element, irreducible_poly)	Determines if $g(x)$ is a primitive element of the Galois field $\text{GF}(p^m)$ with degree-m irreducible polynomial $f(x)$ over $\text{GF}(p)$.
is_primitive_root(g, n)	Determines if g is a primitive root modulo n.
is_smooth(n, B)	Determines if the positive integer n is B-smooth, i.e. all its prime factors satisfy $p \leq B$.
isqrt(n)	Computes the integer square root of n such that $\text{isqrt}(n)^2 \leq n$.
kth_prime(k)	Returns the k-th prime.
lcm(*integers)	Computes the least common multiple of the integer arguments.
mersenne_exponents([n])	Returns all known Mersenne exponents e for $e \leq n$.
mersenne_primes([n])	Returns all known Mersenne primes p for $p \leq 2^n - 1$.
miller_rabin_primality_test(n[, a, rounds])	Probabilistic primality test of n.
next_prime(x)	Returns the nearest prime p, such that $p > x$.
poly_exp_mod(poly, power, modulus)	Efficiently exponentiates a polynomial $f(x)$ to the power k reducing by modulo $g(x)$, $f^k \text{ mod } g$.
poly_gcd(a, b)	Finds the greatest common divisor of two polynomials $a(x)$ and $b(x)$ over $\text{GF}(q)$.
prev_prime(x)	Returns the nearest prime p, such that $p \leq x$.
prime_factors(n)	Computes the prime factors of the positive integer n.
primes(n)	Returns all primes p for $p \leq n$.
primitive_element(irreducible_poly[, start, ...])	Finds the smallest primitive element $g(x)$ of the Galois field $\text{GF}(p^m)$ with degree-m irreducible polynomial $f(x)$ over $\text{GF}(p)$.
primitive_elements(irreducible_poly[, ...])	Finds all primitive elements $g(x)$ of the Galois field $\text{GF}(p^m)$ with degree-m irreducible polynomial $f(x)$ over $\text{GF}(p)$.
primitive_root(n[, start, stop, reverse])	Finds the smallest primitive root modulo n.

continues on next page

Table 15 – continued from previous page

primitive_roots(n[, start, stop, reverse])	Finds all primitive roots modulo n.
random_prime(bits)	Returns a random prime p with b bits, such that $2^b \leq p < 2^{b+1}$.
totatives(n)	Returns the positive integers (totatives) in $1 \leq k < n$ that are coprime with n, i.e. $gcd(n, k) = 1$.

6.1.6 galois.carmichael

galois.**carmichael**(n)

Finds the smallest positive integer m such that $a^m \equiv 1 (\text{mod } n)$ 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 $a^m \equiv 1 (\text{mod } n)$ 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 [326]: n = 20

In [327]: lambda_ = galois.carmichael(n); lambda_
Out[327]: 4

# Find the totatives that are relatively coprime with n
In [328]: totatives = [i for i in range(n) if math.gcd(i, n) == 1]; totatives
Out[328]: [1, 3, 7, 9, 11, 13, 17, 19]

In [329]: 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
```

(continues on next page)

(continued from previous page)

```
In [330]: galois.euler_totient(13), galois.carmichael(13)
Out[330]: (12, 12)
```

6.1.7 galois.conway_poly

galois.**conway_poly**(p, n)

Returns the degree-n Conway polynomial $C_{p,n}$ over $\mathrm{GF}(p)$.

A Conway polynomial is a an irreducible and primitive polynomial over $\mathrm{GF}(p)$ that provides a standard representation of $\mathrm{GF}(p^n)$ as a splitting field of $C_{p,n}$. Conway polynomials provide compatability between fields and their subfields, and hence are the common way to represent extension fields.

The Conway polynomial $C_{p,n}$ is defined as the lexicographically-minimal monic irreducible polynomial of degree n over $\mathrm{GF}(p)$ that is compatible with all $C_{p,m}$ for m dividing n.

This function uses Frank Luebeck's Conway polynomial database for fast lookup, not construction.

Parameters

- **p** (*int*) – The prime characteristic of the field $\mathrm{GF}(p)$.
- **n** (*int*) – The degree n of the Conway polynomial.

Returns The degree-n Conway polynomial $C_{p,n}$ over $\mathrm{GF}(p)$.

Return type *galois.Poly*

Raises **LookupError** – If the Conway polynomial $C_{p,n}$ is not found in Frank Luebeck's database.

Warning: If the $\mathrm{GF}(p)$ field hasn't previously been created, it will be created in this function since it's needed for the construction of the return polynomial.

Examples

```
In [331]: galois.conway_poly(2, 100)
Out[331]: 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 [332]: galois.conway_poly(7, 13)
Out[332]: Poly(x^13 + 6x^2 + 4, GF(7))
```

6.1.8 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 [333]: a = [0, 3, 4]

In [334]: m = [3, 4, 5]

In [335]: x = galois.crt(a, m); x
Out[335]: 39

In [336]: 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
```

6.1.9 galois.euler_totient

galois.**euler_totient**(n)

Counts the positive integers (totatives) in $1 \leq k < n$ that are relatively prime to n, i.e. $gcd(n, k) = 1$.

Implements the Euler Totient function $\phi(n)$.

Parameters **n** (*int*) – A positive integer.

Returns The number of totatives that are relatively prime to n.

Return type int

References

- https://en.wikipedia.org/wiki/Euler%27s_totient_function
- https://oeis.org/A000010

Examples

```
In [337]: n = 20

In [338]: phi = galois.euler_totient(n); phi
Out[338]: 8

# Find the totatives that are coprime with n
In [339]: totatives = [k for k in range(n) if math.gcd(k, n) == 1]; totatives
Out[339]: [1, 3, 7, 9, 11, 13, 17, 19]

# The number of totatives is phi
In [340]: len(totatives) == phi
Out[340]: True

# For prime n, phi is always n-1
In [341]: galois.euler_totient(13)
Out[341]: 12
```

6.1.10 galois.fermat_primality_test

galois.**fermat_primality_test**(*n*)

Probabilistic primality test of n.

This function implements Fermat's primality test. The test says that for an integer n, select an integer a coprime with n. If $a^{n-1} \equiv 1 (\text{mod } n)$, then n is prime or pseudoprime.

Parameters **n** (*int*) – A positive integer.

Returns `False` if n is known to be composite. `True` if n is prime or pseudoprime.

Return type bool

References

- https://oeis.org/A001262
- https://oeis.org/A001567

Examples

```
# List of some primes
In [342]: primes = [257, 24841, 65497]

In [343]: for prime in primes:
```

(continues on next page)

(continued from previous page)

```
   .....:     is_prime = galois.fermat_primality_test(prime)
   .....:     p, k = galois.prime_factors(prime)
   .....:     print("Prime = {:5d}, Fermat's Prime Test = {}, Prime factors = {}".
↪format(prime, is_prime, list(p)))
   .....:
Prime =   257, Fermat's Prime Test = True, Prime factors = [257]
Prime = 24841, Fermat's Prime Test = True, Prime factors = [24841]
Prime = 65497, Fermat's Prime Test = True, Prime factors = [65497]

# List of some strong pseudoprimes with base 2
In [344]: pseudoprimes = [2047, 29341, 65281]

In [345]: for pseudoprime in pseudoprimes:
   .....:     is_prime = galois.fermat_primality_test(pseudoprime)
   .....:     p, k = galois.prime_factors(pseudoprime)
   .....:
↪print("Pseudoprime = {:5d}, Fermat's Prime Test = {}, Prime factors = {}".
↪format(pseudoprime, is_prime, list(p)))
   .....:
Pseudoprime =  2047, Fermat's Prime Test = True, Prime factors = [23, 89]
Pseudoprime = 29341, Fermat's Prime Test = True, Prime factors = [13, 37, 61]
Pseudoprime = 65281, Fermat's Prime Test = True, Prime factors = [97, 673]
```

6.1.11 galois.gcd

galois.**gcd**(a, b)

Finds the integer multiplicands of a and b such that $ax + by = \gcd(a, b)$.

This implementation uses 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 [346]: a = 2

In [347]: b = 13

In [348]: gcd, x, y = galois.gcd(a, b)

In [349]: gcd, x, y
Out[349]: (1, -6, 1)

In [350]: a*x + b*y == gcd
Out[350]: True
```

6.1.12 galois.is_cyclic

galois.**is_cyclic**(n)

Determines whether the multiplicative group $\mathbb{Z}_n^\times$ is cyclic.

The multiplicative group $\mathbb{Z}_n^\times$ is the set of positive integers $1 \leq a < n$ that are coprime with n. $\mathbb{Z}_n^\times$ being cyclic means that some primitive root (or generator) g can generate the group $\mathbb{Z}_n^\times = \{g, g^2, \ldots, g^k\}$, where k is order of the group. The order of the group is defined by Euler's totient function, $\phi(n) = k$. If $\mathbb{Z}_n^\times$ is cyclic, the number of primitive roots is found by $\phi(k)$ or $\phi(\phi(n))$.

$\mathbb{Z}_n^\times$ 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 $\mathbb{Z}_n^\times$ is cyclic.

Return type bool

References

- https://en.wikipedia.org/wiki/Primitive_root_modulo_n

Examples

The elements of $\mathbb{Z}_n^\times$ are the positive integers less than n that are coprime with n. For example when $n = 14$, then $\mathbb{Z}_{14}^\times = \{1, 3, 5, 9, 11, 13\}$.

```
# n is of type 2*p^k, which is cyclic
In [351]: n = 14

In [352]: galois.is_cyclic(n)
```

(continues on next page)

(continued from previous page)

```
Out[352]: True

# The congruence class coprime with n
In [353]: Znx = set([a for a in range(1, n) if math.gcd(n, a) == 1]); Znx
Out[353]: {1, 3, 5, 9, 11, 13}

# Euler's totient function counts the "totatives", positive integers coprime with n
In [354]: phi = galois.euler_totient(n); phi
Out[354]: 6

In [355]: len(Znx) == phi
Out[355]: True

# The primitive roots are the elements in Znx that multiplicatively generate the␣
↪group
In [356]: 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))
   .....:
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 [357]: roots = galois.primitive_roots(n); roots
Out[357]: [3, 5]

# Euler's totient function phi(phi(n)) counts the primitive roots of n
In [358]: len(roots) == galois.euler_totient(phi)
Out[358]: True
```

A counterexample is $n = 15 = 3 * 5$, which doesn't fit the condition for cyclicness. $\mathbb{Z}_{15}^{\times} = \{1, 2, 4, 7, 8, 11, 13, 14\}$.

```
# n is of type p1^k1 * p2^k2, which is not cyclic
In [359]: n = 15

In [360]: galois.is_cyclic(n)
Out[360]: False

# The congruence class coprime with n
In [361]: Znx = set([a for a in range(1, n) if math.gcd(n, a) == 1]); Znx
Out[361]: {1, 2, 4, 7, 8, 11, 13, 14}

# Euler's totient function counts the "totatives", positive integers coprime with n
In [362]: phi = galois.euler_totient(n); phi
Out[362]: 8
```

(continues on next page)

(continued from previous page)

```
In [363]: len(Znx) == phi
Out[363]: True

# The primitive roots are the elements in Znx that multiplicatively generate the␣
↪group
In [364]: 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

In [365]: roots = galois.primitive_roots(n); roots
Out[365]: []

# Note the max order of any element is 4, not 8, which is Carmichael's lambda␣
↪function
In [366]: galois.carmichael(n)
Out[366]: 4
```

6.1.13 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-n polynomial $f(x)$ over $\mathrm{GF}(p)$ for prime p is irreducible if and only if $f(x) \mid (x^{p^n} - x)$ and $\gcd(f(x),\ x^{p^{m_i}} - x) = 1$ for $1 \leq i \leq k$, where $m_i = n/p_i$ for the k prime divisors p_i of n.

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
- https://en.wikipedia.org/wiki/Factorization_of_polynomials_over_finite_fields

Examples

```
# Conway polynomials are always irreducible (and primitive)
In [367]: f = galois.conway_poly(2, 5); f
Out[367]: Poly(x^5 + x^2 + 1, GF(2))

# f(x) has no roots in GF(2), a requirement of being irreducible
In [368]: f.roots()
Out[368]: GF([], order=2)

In [369]: galois.is_irreducible(f)
Out[369]: True
```

```
In [370]: g = galois.conway_poly(2, 4); g
Out[370]: Poly(x^4 + x + 1, GF(2))

In [371]: h = galois.conway_poly(2, 5); h
Out[371]: Poly(x^5 + x^2 + 1, GF(2))

In [372]: f = g * h; f
Out[372]: 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 [373]: f.roots()
Out[373]: GF([], order=2)

In [374]: galois.is_irreducible(f)
Out[374]: False
```

6.1.14 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 [375]: GF = galois.GF(7)

In [376]: p = galois.Poly([1,0,4,5], field=GF); p
Out[376]: Poly(x^3 + 4x + 5, GF(7))

In [377]: galois.is_monic(p)
Out[377]: True
```

```
In [378]: p = galois.Poly([3,0,4,5], field=GF); p
Out[378]: Poly(3x^3 + 4x + 5, GF(7))

In [379]: galois.is_monic(p)
Out[379]: False
```

6.1.15 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 [380]: galois.is_prime(13)
Out[380]: True

In [381]: galois.is_prime(15)
Out[381]: False
```

The algorithm is also efficient on very large n.

```
In [382]: galois.is_prime(100000000000000000035000061)
Out[382]: True
```

6.1.16 galois.is_primitive

galois.**is_primitive**(*poly*)

Checks whether the polynomial $f(x)$ over $\mathrm{GF}(p)$ is primitive.

A degree-n polynomial $f(x)$ over $\mathrm{GF}(p)$ is *primitive* if $f(x) \mid (x^k - 1)$ for $k = p^n - 1$ and no k less than $p^n - 1$.

Parameters **poly** (galois.Poly) – A polynomial $f(x)$ over $\mathrm{GF}(p)$.

Returns True if the polynomial is primitive.

Return type bool

Examples

All Conway polynomials are primitive.

```
In [383]: f = galois.conway_poly(2, 8); f
Out[383]: Poly(x^8 + x^4 + x^3 + x^2 + 1, GF(2))

In [384]: galois.is_primitive(f)
Out[384]: True

In [385]: f = galois.conway_poly(3, 5); f
Out[385]: Poly(x^5 + 2x + 1, GF(3))

In [386]: galois.is_primitive(f)
Out[386]: True
```

The irreducible polynomial of $\mathrm{GF}(2^8)$ for AES is not primitive.

```
In [387]: f = galois.Poly.Degrees([8,4,3,1,0]); f
Out[387]: Poly(x^8 + x^4 + x^3 + x + 1, GF(2))

In [388]: galois.is_primitive(f)
Out[388]: False
```

6.1.17 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_totient*.

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 [389]: GF = galois.GF(3)

In [390]: f = galois.Poly([1,1,2], field=GF); f
Out[390]: Poly(x^2 + x + 2, GF(3))

In [391]: galois.is_irreducible(f)
Out[391]: True

In [392]: galois.is_primitive(f)
Out[392]: True

In [393]: g = galois.Poly.Identity(GF); g
Out[393]: Poly(x, GF(3))

In [394]: galois.is_primitive_element(g, f)
Out[394]: True
```

```
In [395]: GF = galois.GF(3)

In [396]: f = galois.Poly([1,0,1], field=GF); f
Out[396]: Poly(x^2 + 1, GF(3))

In [397]: galois.is_irreducible(f)
Out[397]: True

In [398]: galois.is_primitive(f)
Out[398]: False

In [399]: g = galois.Poly.Identity(GF); g
Out[399]: Poly(x, GF(3))

In [400]: galois.is_primitive_element(g, f)
Out[400]: False
```

6.1.18 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 [401]: galois.is_primitive_root(2, 7)
Out[401]: False

In [402]: galois.is_primitive_root(3, 7)
Out[402]: True

In [403]: galois.primitive_roots(7)
Out[403]: [3, 5]
```

6.1.19 galois.is_smooth

galois.**is_smooth**(n, B)

Determines if the positive integer n is B-smooth, i.e. all its prime factors satisfy $p \leq B$.

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.

Returns `True` if n is B-smooth.

Return type bool

Examples

```
In [404]: galois.is_smooth(2**10, 2)
Out[404]: True

In [405]: galois.is_smooth(10, 5)
Out[405]: True

In [406]: galois.is_smooth(12, 5)
Out[406]: True

In [407]: galois.is_smooth(60**2, 5)
Out[407]: True
```

6.1.20 galois.isqrt

galois.**isqrt**(n)

Computes the integer square root of n such that $\text{isqrt}(n)^2 \leq n$.

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 such that $\text{isqrt}(n)^2 \leq n$.

Return type int

Examples

```
# Use a large Mersenne prime
In [408]: p = galois.mersenne_primes(2000)[-1]; p
Out[408]:␣
↪1040793219466439908192524032736408553861526224726670480531911235040360805967336029801223944173232

In [409]: sqrt_p = galois.isqrt(p); sqrt_p
Out[409]:␣
↪3226132699481594337650229932669505772017441235628244885631123722785761803162998767122846394796285

In [410]: sqrt_p**2 <= p
Out[410]: True

In [411]: (sqrt_p + 1)**2 <= p
Out[411]: False
```

6.1.21 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 [412]: galois.kth_prime(1)
Out[412]: 2

In [413]: galois.kth_prime(3)
Out[413]: 5
```

(continues on next page)

(continued from previous page)

```
In [414]: galois.kth_prime(1000)
Out[414]: 7919
```

6.1.22 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 [415]: galois.lcm()
Out[415]: 1

In [416]: galois.lcm(2, 4, 14)
Out[416]: 28

In [417]: galois.lcm(3, 0, 9)
Out[417]: 0
```

This function also works on arbitrarily-large integers.

```
In [418]: prime1, prime2 = galois.mersenne_primes(100)[-2:]

In [419]: prime1, prime2
Out[419]: (2305843009213693951, 618970019642690137449562111)

In [420]: lcm = galois.lcm(prime1, prime2); lcm
Out[420]: 1427247692705959880439315947500961989719490561

In [421]: lcm == prime1 * prime2
Out[421]: True
```

6.1.23 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 [422]: e = galois.mersenne_exponents(2000); e
Out[422]: [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 [423]: p = 2**e[-1] - 1; p
Out[423]:␣
↪10407932194664399081925240327364085538615262247266704805319112350403608059673360298012239441732321

In [424]: galois.is_prime(p)
Out[424]: True
```

6.1.24 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 [425]: p = galois.mersenne_primes(2000); p
Out[425]:
[3,
 7,
 31,
 127,
 8191,
 131071,
 524287,
 2147483647,
 2305843009213693951,
 618970019642690137449562111,
 162259276829213363391578010288127,
 170141183460469231731687303715884105727,
⌊
 ↪68647976601306097149819007990813932172694353001433054093944634591855431833976560521225596406614547
 ↪
⌊
 ↪53113799281676709868958820655246862732959311772703192319944413820040355986085224273916250226522927
 ↪
⌊
 ↪10407932194664399081925240327364085538615262247266704805319112350403608059673360298012239441732324
 ↪

In [426]: galois.is_prime(p[-1])
Out[426]: True
```

6.1.25 galois.miller_rabin_primality_test

galois.**miller_rabin_primality_test**(*n*, *a=None*, *rounds=1*)

Probabilistic primality test of n.

This function implements the Miller-Rabin primality test. The test says that for an integer n, select an integer a such that $a < n$. Factor $n - 1$ such that $2^s d = n - 1$. Then, n is composite, if $a^d \not\equiv 1 (\text{mod } n)$ and $a^{2^r d} \not\equiv n - 1 (\text{mod } n)$ for $1 \leq r < s$.

Parameters

- **n** (*int*) – A positive integer.
- **a** (*int, optional*) – Initial composite witness value, $1 \leq a < n$. On subsequent rounds, a will be a different value. The default is a random value.
- **rounds** (*int, optional*) – The number of iterations attempting to detect n as composite. Additional rounds will choose new a. Sufficient rounds have arbitrarily-high probability of detecting a composite.

Returns `False` if n is known to be composite. `True` if n is prime or pseudoprime.

Return type bool

References

- https://math.dartmouth.edu/~carlp/PDF/paper25.pdf
- https://oeis.org/A001262

Examples

```
# List of some primes
In [427]: primes = [257, 24841, 65497]

In [428]: for prime in primes:
   .....:     is_prime = galois.miller_rabin_primality_test(prime)
   .....:     p, k = galois.prime_factors(prime)
   .....:     print("Prime = {:5d}, Miller-Rabin Prime Test = {}, Prime factors = {}".
   ↪format(prime, is_prime, list(p)))
   .....:
Prime =   257, Miller-Rabin Prime Test = True, Prime factors = [257]
Prime = 24841, Miller-Rabin Prime Test = True, Prime factors = [24841]
Prime = 65497, Miller-Rabin Prime Test = True, Prime factors = [65497]

# List of some strong pseudoprimes with base 2
In [429]: pseudoprimes = [2047, 29341, 65281]

# Single round of Miller-Rabin, sometimes fooled by pseudoprimes
In [430]: for pseudoprime in pseudoprimes:
   .....:     is_prime = galois.miller_rabin_primality_test(pseudoprime)
   .....:     p, k = galois.prime_factors(pseudoprime)
   .....:     print("Pseudoprime = {:5d}, Miller-Rabin Prime Test = {}, Prime factors = {}".
   ↪format(pseudoprime, is_prime, list(p)))
   .....:
Pseudoprime =  2047, Miller-Rabin Prime Test = False, Prime factors = [23, 89]
Pseudoprime = 29341, Miller-Rabin Prime Test = False, Prime factors = [13, 37, 61]
Pseudoprime = 65281, Miller-Rabin Prime Test = False, Prime factors = [97, 673]

# 7 rounds of Miller-Rabin, never fooled by pseudoprimes
In [431]: for pseudoprime in pseudoprimes:
   .....:     is_prime = galois.miller_rabin_primality_test(pseudoprime, rounds=7)
   .....:     p, k = galois.prime_factors(pseudoprime)
   .....:     print("Pseudoprime = {:5d}, Miller-Rabin Prime Test = {}, Prime factors = {}".
   ↪format(pseudoprime, is_prime, list(p)))
   .....:
Pseudoprime =  2047, Miller-Rabin Prime Test = False, Prime factors = [23, 89]
Pseudoprime = 29341, Miller-Rabin Prime Test = False, Prime factors = [13, 37, 61]
Pseudoprime = 65281, Miller-Rabin Prime Test = False, Prime factors = [97, 673]
```

6.1.26 galois.next_prime

galois.**next_prime**(x)

Returns the nearest prime p, such that $p > x$.

Parameters **x** (*int*) – A positive integer.

Returns The nearest prime $p > x$.

Return type int

Examples

```
In [432]: galois.next_prime(13)
Out[432]: 17

In [433]: galois.next_prime(15)
Out[433]: 17
```

6.1.27 galois.poly_exp_mod

galois.**poly_exp_mod**(*poly*, *power*, *modulus*)

Efficiently exponentiates a polynomial $f(x)$ to the power k reducing by modulo $g(x)$, f^k mod g.

The algorithm is more efficient than exponentiating first and then reducing modulo $g(x)$. Instead, this algorithm repeatedly squares f, reducing modulo g at each step.

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 [434]: GF = galois.GF(31)

In [435]: f = galois.Poly.Random(10, field=GF); f
Out[435]: Poly(13x^10 + 27x^9 + 14x^8 + 7x^7 + 29x^6 + 23x^4 + 28x^3 + 4x^2 + 27x +␣
↪3, GF(31))

In [436]: g = galois.Poly.Random(7, field=GF); g
Out[436]: Poly(16x^7 + 14x^6 + 17x^5 + 26x^4 + 5x^3 + 20x^2 + 20x + 23, GF(31))

# %timeit f**200 % g
# 1.23 s ± 41.1 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
In [437]: f**200 % g
Out[437]: Poly(4x^6 + 16x^5 + 4x^4 + 3x^3 + 3x^2 + 13x + 19, GF(31))
```

(continues on next page)

(continued from previous page)

```
# %timeit galois.poly_exp_mod(f, 200, g)
# 41.7 ms ± 468 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
In [438]: galois.poly_exp_mod(f, 200, g)
Out[438]: Poly(4x^6 + 16x^5 + 4x^4 + 3x^3 + 3x^2 + 13x + 19, GF(31))
```

6.1.28 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 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 $x(x)$, such that $ax + by = gcd(a, b)$.
- *galois.Poly* – Polynomial $y(x)$, such that $ax + by = gcd(a, b)$.

Examples

```
In [439]: GF = galois.GF(7)

In [440]: a = galois.Poly.Roots([2,2,2,3,6], field=GF); a
Out[440]: Poly(x^5 + 6x^4 + x + 3, GF(7))

# a(x) and b(x) only share the root 2 in common
In [441]: b = galois.Poly.Roots([1,2], field=GF); b
Out[441]: Poly(x^2 + 4x + 2, GF(7))

In [442]: gcd, x, y = galois.poly_gcd(a, b)

# The GCD has only 2 as a root with multiplicity 1
In [443]: gcd.roots(multiplicity=True)
Out[443]: (GF([2], order=7), array([1]))

In [444]: a*x + b*y == gcd
Out[444]: True
```

6.1.29 galois.prev_prime

galois.**prev_prime**(x)

Returns the nearest prime p, such that $p \leq x$.

Parameters **x** (*int*) – A positive integer.

Returns The nearest prime $p \leq x$.

Return type int

Examples

```
In [445]: galois.prev_prime(13)
Out[445]: 13

In [446]: galois.prev_prime(15)
Out[446]: 13
```

6.1.30 galois.prime_factors

galois.**prime_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. Use trial division with a list of primes up to 10^6. If no residual factors, return the discovered prime factors.
3. Use Pollard's Rho algorithm to find a non-trivial factor of the residual. Continue until all are found.

Parameters **n** (*int*) – The positive integer to be factored.

Returns

- *list* – Sorted list of k prime factors $p = [p_1, p_2, \dots, p_{k-1}]$ with $p_1 < p_2 < \cdots < p_{k-1}$.
- *list* – List of corresponding prime powers $e = [e_1, e_2, \dots, e_{k-1}]$.

Examples

```
In [447]: p, e = galois.prime_factors(120)

In [448]: p, e
Out[448]: ([2, 3, 5], [3, 1, 1])

# The product of the prime powers is the factored integer
In [449]: np.multiply.reduce(np.array(p) ** np.array(e))
Out[449]: 120
```

Prime factorization of 1 less than a large prime.

```
In [450]: prime =100000000000000035000061

In [451]: galois.is_prime(prime)
Out[451]: True

In [452]: p, e = galois.prime_factors(prime - 1)

In [453]: p, e
Out[453]: ([2, 3, 5, 17, 19, 112850813, 457237177399], [2, 1, 1, 1, 1, 1, 1])

In [454]: np.multiply.reduce(np.array(p) ** np.array(e))
Out[454]: 2003764205241896700
```

6.1.31 galois.primes

galois.**primes**(n)

Returns all primes p for $p \leq n$.

Parameters **n** (*int*) – A positive integer.

Returns The primes up to and including n.

Return type list

References

- https://oeis.org/A000040

Examples

```
In [455]: galois.primes(19)
Out[455]: [2, 3, 5, 7, 11, 13, 17, 19]
```

6.1.32 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 [456]: GF = galois.GF(3)

In [457]: f = galois.Poly([1,1,2], field=GF); f
Out[457]: Poly(x^2 + x + 2, GF(3))

In [458]: galois.is_irreducible(f)
Out[458]: True

In [459]: galois.is_primitive(f)
Out[459]: True

In [460]: galois.primitive_element(f)
Out[460]: Poly(x, GF(3))
```

```
In [461]: GF = galois.GF(3)

In [462]: f = galois.Poly([1,0,1], field=GF); f
Out[462]: Poly(x^2 + 1, GF(3))

In [463]: galois.is_irreducible(f)
Out[463]: True

In [464]: galois.is_primitive(f)
Out[464]: False

In [465]: galois.primitive_element(f)
Out[465]: Poly(x + 1, GF(3))
```

6.1.33 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_totient`.

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 [466]: GF = galois.GF(3)

In [467]: f = galois.Poly([1,1,2], field=GF); f
Out[467]: Poly(x^2 + x + 2, GF(3))

In [468]: galois.is_irreducible(f)
Out[468]: True

In [469]: galois.is_primitive(f)
Out[469]: True

In [470]: g = galois.primitive_elements(f); g
Out[470]: [Poly(x, GF(3)), Poly(x + 1, GF(3)), Poly(2x, GF(3)), Poly(2x + 2, GF(3))]

In [471]: len(g) == galois.euler_totient(3**2 - 1)
Out[471]: True
```

```
In [472]: GF = galois.GF(3)

In [473]: f = galois.Poly([1,0,1], field=GF); f
Out[473]: Poly(x^2 + 1, GF(3))

In [474]: galois.is_irreducible(f)
Out[474]: True

In [475]: galois.is_primitive(f)
Out[475]: False

In [476]: g = galois.primitive_elements(f); g
Out[476]:
[Poly(x + 1, GF(3)),
 Poly(x + 2, GF(3)),
 Poly(2x + 1, GF(3)),
 Poly(2x + 2, GF(3))]

In [477]: len(g) == galois.euler_totient(3**2 - 1)
Out[477]: True
```

6.1.34 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 $\mathbb{Z}/\mathbf{n}\mathbb{Z}$.

```
In [478]: n = 6

In [479]: root = galois.primitive_root(n); root
Out[479]: 5
```

(continues on next page)

(continued from previous page)

```
# The congruence class coprime with n
In [480]: Znx = set([a for a in range(1, n) if math.gcd(n, a) == 1]); Znx
Out[480]: {1, 5}

# Euler's totient function counts the "totatives", positive integers coprime with n
In [481]: phi = galois.euler_totient(n); phi
Out[481]: 2

In [482]: len(Znx) == phi
Out[482]: True

# The primitive roots are the elements in Znx that multiplicatively generate the␣
↪group
In [483]: 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 $\mathbb{Z}/\mathbf{n}\mathbb{Z}$.

```
In [484]: n = 7

In [485]: root = galois.primitive_root(n); root
Out[485]: 3

# The congruence class coprime with n
In [486]: Znx = set([a for a in range(1, n) if math.gcd(n, a) == 1]); Znx
Out[486]: {1, 2, 3, 4, 5, 6}

# Euler's totient function counts the "totatives", positive integers coprime with n
In [487]: phi = galois.euler_totient(n); phi
Out[487]: 6

In [488]: len(Znx) == phi
Out[488]: True

# The primitive roots are the elements in Znx that multiplicatively generate the␣
↪group
In [489]: 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
```

(continues on next page)

(continued from previous page)

```
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 [490]: n = 1000000000000000035000061

In [491]: galois.primitive_root(n)
Out[491]: 7

In [492]: galois.primitive_root(n, reverse=True)
Out[492]: 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 [493]: n = 8

In [494]: root = galois.primitive_root(n); root

# The congruence class coprime with n
In [495]: Znx = set([a for a in range(1, n) if math.gcd(n, a) == 1]); Znx
Out[495]: {1, 3, 5, 7}

# Euler's totient function counts the "totatives", positive integers coprime with n
In [496]: phi = galois.euler_totient(n); phi
Out[496]: 4

In [497]: len(Znx) == phi
Out[497]: True

# Test all elements for being primitive roots. The powers of a primitive span the␣
↪congruence classes mod n.
In [498]: 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 [499]: galois.carmichael(n)
Out[499]: 2
```

6.1.35 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 [500]: n = 6

In [501]: roots = galois.primitive_roots(n); roots
Out[501]: [5]

# The congruence class coprime with n
In [502]: Znx = set([a for a in range(1, n) if math.gcd(n, a) == 1]); Znx
Out[502]: {1, 5}
```

(continues on next page)

(continued from previous page)

```
# Euler's totient function counts the "totatives", positive integers coprime with n
In [503]: phi = galois.euler_totient(n); phi
Out[503]: 2

In [504]: len(Znx) == phi
Out[504]: True

# Test all elements for being primitive roots. The powers of a primitive span the␣
↪congruence classes mod n.
In [505]: 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 [506]: len(roots) == galois.euler_totient(phi)
Out[506]: 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 [507]: n = 7

In [508]: roots = galois.primitive_roots(n); roots
Out[508]: [3, 5]

# The congruence class coprime with n
In [509]: Znx = set([a for a in range(1, n) if math.gcd(n, a) == 1]); Znx
Out[509]: {1, 2, 3, 4, 5, 6}

# Euler's totient function counts the "totatives", positive integers coprime with n
In [510]: phi = galois.euler_totient(n); phi
Out[510]: 6

In [511]: len(Znx) == phi
Out[511]: True

# Test all elements for being primitive roots. The powers of a primitive span the␣
↪congruence classes mod n.
In [512]: 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
```

(continues on next page)

(continued from previous page)

```
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 [513]: len(roots) == galois.euler_totient(phi)
Out[513]: True
```

The algorithm is also efficient for very large n.

```
In [514]: n = 100000000000000035000061

# Euler's totient function phi(phi(n)) counts the primitive roots of n
In [515]: galois.euler_totient(galois.euler_totient(n))
Out[515]: 2377708957253484157870008

# Only find some of the primitive roots
In [516]: galois.primitive_roots(n, stop=100)
Out[516]: [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 [517]: 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 [518]: n = 8

In [519]: roots = galois.primitive_roots(n); roots
Out[519]: []

# The congruence class coprime with n
In [520]: Znx = set([a for a in range(1, n) if math.gcd(n, a) == 1]); Znx
Out[520]: {1, 3, 5, 7}

# Euler's totient function counts the "totatives", positive integers coprime with n
In [521]: phi = galois.euler_totient(n); phi
Out[521]: 4

In [522]: len(Znx) == phi
Out[522]: True

# Test all elements for being primitive roots. The powers of a primitive span the␣
↪congruence classes mod n.
In [523]: 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))
```

(continues on next page)

(continued from previous page)

```
   .....:
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
```

6.1.36 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 [524]: p = galois.random_prime(1024); p
Out[524]:␣
↪19419442655770645166620047714165582718562753676752452275750219537007025945021759947391236282510995
In [525]: galois.is_prime(p)
Out[525]: True
```

```
$ openssl prime␣
↪23686178792695738220699688608721459202975252407802639235893684447966742357083311612650692787873131
1514D68EDB7C650F1FF713531A1A43255A4BE6D66EE1FDBD96F4EB32757C1B1BAF16A5933E24D45FAD6C6A814F3C8C14F3C
↪(23686178792695738220699688608721459202975252407802639235893684447966742357083311612650692787873
↪is prime
```

6.1.37 galois.totatives

galois.**totatives**(n)

Returns the positive integers (totatives) in $1 \leq k < n$ that are coprime with n, i.e. $gcd(n, k) = 1$.

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 [526]: n = 20

In [527]: totatives = galois.totatives(n); totatives
Out[527]: [1, 3, 7, 9, 11, 13, 17, 19]

In [528]: phi = galois.euler_totient(n); phi
Out[528]: 8

In [529]: len(totatives) == phi
Out[529]: True
```

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

A pre-created Galois field array class for $\mathrm{GF}(2)$.

This class is a subclass of *galois.GFArray* and has metaclass *galois.GFMeta*.

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.GFMeta*.

```
# View a summary of the field's properties
In [4]: print(galois.GF2.properties)
GF(2):
  characteristic: 2
  degree: 1
  order: 2
  irreducible_poly: Poly(x + 1, GF(2))
  is_primitive_poly: True
  primitive_element: GF(1, order=2)

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

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)
```

---

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.GFMeta.dtypes*.

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

Return type *galois.GFArray*

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.GFMeta.dtypes*.

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

Return type *galois.GFArray*

Examples

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

In [2]: GF.Identity(4)
Out[2]:
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.GFMeta.dtypes*.

Returns A Galois field array of ones.

Return type *galois.GFArray*

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.GFMeta.dtypes*.

Returns A Galois field array of random field elements.

Return type *galois.GFArray*

Examples

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

In [2]: GF.Random((2,5))
Out[2]:
GF([[ 2, 18,  3,  8, 23],
    [26, 25, 20, 30,  2]], 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.GFMeta.dtypes*.

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

Return type *galois.GFArray*

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.GFArray*) – 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.GFMeta.dtypes*.

Returns The $m \times n$ Vandermonde matrix.

Return type *galois.GFArray*

Examples

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

In [2]: a = GF.primitive_element

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.GFMeta.dtypes*.

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

Return type *galois.GFArray*

Examples

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

In [2]: vec = galois.GF2.Random((3,6)); vec
```

(continues on next page)

(continued from previous page)

```
Out[2]:
GF([[1, 0, 0, 1, 0, 1],
    [1, 1, 0, 1, 0, 1],
    [0, 0, 1, 0, 0, 1]], order=2)

In [3]: a = GF.Vector(vec); a
Out[3]: GF([37, 53,  9], order=2^6)

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

In [5]: a.vector()
Out[5]:
GF([[1, 0, 0, 1, 0, 1],
    [1, 1, 0, 1, 0, 1],
    [0, 0, 1, 0, 0, 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.GFMeta.dtypes*.

Returns A Galois field array of zeros.

Return type *galois.GFArray*

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.GFArray* – The lower triangular matrix.
- *galois.GFArray* – 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)

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.GFArray* – The lower triangular matrix.
- *galois.GFArray* – The upper triangular matrix.
- *galois.GFArray* – 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)
```

(continues on next page)

(continued from previous page)

```
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
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.GFArray*

Examples

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

In [2]: A = GF.Random((4,4)); A
Out[2]:
GF([[ 3, 17, 21,  2],
    [14,  4, 11,  8],
    [11,  3, 19, 14],
    [ 6, 23, 30,  6]], 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)
```

(continues on next page)

(continued from previous page)

```
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([[20, 24, 11,  4],
    [11, 23,  1, 16],
    [20, 16, 10, 19],
    [ 1,  1, 30, 28]], order=31)

In [7]: A[:,2] = A[:,1] * GF(17); A
Out[7]:
GF([[20, 24,  5,  4],
    [11, 23, 19, 16],
    [20, 16, 24, 19],
    [ 1,  1, 17, 28]], 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([[30,  6, 25,  9],
    [17, 27, 21,  9],
    [24,  6, 12, 13],
    [ 2, 16, 30, 27]], order=31)

In [12]: A[3,:] = A[2,:] * GF(8); A
Out[12]:
GF([[30,  6, 25,  9],
    [17, 27, 21,  9],
    [24,  6, 12, 13],
    [ 6, 17,  3, 11]], order=31)

In [13]: A.row_reduce()
Out[13]:
```

(continues on next page)

(continued from previous page)

```
GF([[ 1,  0,  0, 11],
    [ 0,  1,  0, 21],
    [ 0,  0,  1, 28],
    [ 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.GFMeta.dtypes*.

Returns A Galois field array of length-m vectors over $\mathrm{GF}(p)$.

Return type *galois.GFArray*

Examples

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

In [2]: a = GF.Random(3); a
Out[2]: GF([24, 33, 48], order=2^6)

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

In [4]: GF.Vector(vec)
Out[4]: GF([24, 33, 48], order=2^6)
```

---

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

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

The *galois.GFArray* 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.GFArray* 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, 5, 6, 2, 4],
    [5, 2, 6, 3, 0]], order=7)
```

galois.GFArray is a subclass of numpy.ndarray. The *galois.GFArray* constructor has the same syntax as numpy.array(). The returned *galois.GFArray* 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.GFMeta.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.GFArray*

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([ 26,  95,  80,  14,  28, 167,  62,  15, 194,  99], 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([118, 137,  62, 156,  60,  22, 199, 163, 207, 196], 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)
```

---

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.GFMeta.dtypes*.

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

Return type *galois.GFArray*

Examples

```
In [24]: GF = galois.GF(31)

In [25]: GF.Elements()
Out[25]:
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.GFMeta.dtypes*.

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

Return type *galois.GFArray*

Examples

```
In [26]: GF = galois.GF(31)

In [27]: GF.Identity(4)
Out[27]:
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.GFMeta.dtypes*.

Returns A Galois field array of ones.

Return type *galois.GFArray*

Examples

```
In [28]: GF = galois.GF(31)

In [29]: GF.Ones((2,5))
Out[29]:
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.GFMeta.dtypes*.

Returns A Galois field array of random field elements.

Return type *galois.GFArray*

Examples

```
In [30]: GF = galois.GF(31)

In [31]: GF.Random((2,5))
Out[31]:
GF([[ 3, 18, 23, 24, 18],
    [20,  5,  1,  6, 26]], 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.GFMeta.dtypes*.

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

Return type *galois.GFArray*

Examples

```
In [32]: GF = galois.GF(31)

In [33]: GF.Range(10,20)
Out[33]: 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.GFArray*) – 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.GFMeta.dtypes*.

Returns The $m \times n$ Vandermonde matrix.

Return type *galois.GFArray*

Examples

```
In [34]: GF = galois.GF(2**3)

In [35]: a = GF.primitive_element
```

(continues on next page)

(continued from previous page)

```
In [36]: V = GF.Vandermonde(a, 7, 7)

In [37]: 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.GFMeta.dtypes*.

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

Return type *galois.GFArray*

Examples

```
In [38]: GF = galois.GF(2**6)

In [39]: vec = galois.GF2.Random((3,6)); vec
Out[39]:
GF([[1, 1, 0, 1, 1, 1],
    [0, 0, 1, 0, 0, 0],
    [1, 0, 0, 1, 1, 0]], order=2)

In [40]: a = GF.Vector(vec); a
Out[40]: GF([55,  8, 38], order=2^6)

In [41]: with GF.display("poly"):
   ....:     print(a)
   ....:
GF([^5 + ^4 + ^2 +  + 1, ^3, ^5 + ^2 + ], order=2^6)

In [42]: a.vector()
Out[42]:
GF([[1, 1, 0, 1, 1, 1],
    [0, 0, 1, 0, 0, 0],
```

(continues on next page)

(continued from previous page)

```
    [1, 0, 0, 1, 1, 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.GFMeta.dtypes*.

Returns A Galois field array of zeros.

Return type *galois.GFArray*

Examples

```
In [43]: GF = galois.GF(31)

In [44]: GF.Zeros((2,5))
Out[44]:
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.GFArray* – The lower triangular matrix.
- *galois.GFArray* – The upper triangular matrix.

Examples

```
In [45]: GF = galois.GF(5)

# Not every square matrix has an LU decomposition
In [46]: A = GF([[2, 4, 4, 1], [3, 3, 1, 4], [4, 3, 4, 2], [4, 4, 3, 1]])

In [47]: L, U = A.lu_decompose()

In [48]: L
Out[48]:
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 [49]: U
Out[49]:
GF([[2, 4, 4, 1],
    [0, 2, 0, 0],
    [0, 0, 1, 0],
    [0, 0, 0, 4]], order=5)

# A = L U
In [50]: np.array_equal(A, L @ U)
Out[50]: True
```

---

lup_decompose()

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

Returns

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

Examples

```
In [51]: GF = galois.GF(5)

In [52]: A = GF([[1, 3, 2, 0], [3, 4, 2, 3], [0, 2, 1, 4], [4, 3, 3, 1]])

In [53]: L, U, P = A.lup_decompose()

In [54]: L
Out[54]:
GF([[1, 0, 0, 0],
    [0, 1, 0, 0],
    [3, 0, 1, 0],
    [4, 3, 2, 1]], order=5)

In [55]: U
Out[55]:
GF([[1, 3, 2, 0],
    [0, 2, 1, 4],
    [0, 0, 1, 3],
    [0, 0, 0, 3]], order=5)

In [56]: P
Out[56]:
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 [57]: np.array_equal(P @ A, L @ U)
Out[57]: 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.GFArray*

Examples

```
In [58]: GF = galois.GF(31)

In [59]: A = GF.Random((4,4)); A
Out[59]:
GF([[16,  6, 27, 27],
    [27,  4, 23,  0],
    [23,  2,  5, 19],
    [ 3, 18, 13, 10]], order=31)

In [60]: A.row_reduce()
Out[60]:
GF([[1, 0, 0, 0],
    [0, 1, 0, 0],
    [0, 0, 1, 0],
    [0, 0, 0, 1]], order=31)

In [61]: np.linalg.matrix_rank(A)
Out[61]: 4
```

One column is a linear combination of another.

```
In [62]: GF = galois.GF(31)

In [63]: A = GF.Random((4,4)); A
Out[63]:
GF([[ 7,  4, 27, 28],
    [18,  7, 15, 12],
    [22, 29, 23, 26],
    [ 4,  5, 24,  6]], order=31)
```

(continues on next page)

(continued from previous page)

```
In [64]: A[:,2] = A[:,1] * GF(17); A
Out[64]:
GF([[ 7,  4,  6, 28],
    [18,  7, 26, 12],
    [22, 29, 28, 26],
    [ 4,  5, 23,  6]], order=31)

In [65]: A.row_reduce()
Out[65]:
GF([[ 1,  0,  0,  0],
    [ 0,  1, 17,  0],
    [ 0,  0,  0,  1],
    [ 0,  0,  0,  0]], order=31)

In [66]: np.linalg.matrix_rank(A)
Out[66]: 3
```

One row is a linear combination of another.

```
In [67]: GF = galois.GF(31)

In [68]: A = GF.Random((4,4)); A
Out[68]:
GF([[14, 17, 28, 27],
    [ 8, 27, 30, 28],
    [30, 22,  1,  8],
    [21, 25,  9,  2]], order=31)

In [69]: A[3,:] = A[2,:] * GF(8); A
Out[69]:
GF([[14, 17, 28, 27],
    [ 8, 27, 30, 28],
    [30, 22,  1,  8],
    [23, 21,  8,  2]], order=31)

In [70]: A.row_reduce()
Out[70]:
GF([[ 1,  0,  0,  5],
    [ 0,  1,  0, 19],
    [ 0,  0,  1, 29],
    [ 0,  0,  0,  0]], order=31)

In [71]: np.linalg.matrix_rank(A)
Out[71]: 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.GFMeta.dtypes*.

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

Return type *galois.GFArray*

Examples

```
In [72]: GF = galois.GF(2**6)

In [73]: a = GF.Random(3); a
Out[73]: GF([11, 37, 11], order=2^6)

In [74]: vec = a.vector(); vec
Out[74]:
GF([[0, 0, 1, 0, 1, 1],
    [1, 0, 0, 1, 0, 1],
    [0, 0, 1, 0, 1, 1]], order=2)

In [75]: GF.Vector(vec)
Out[75]: GF([11, 37, 11], order=2^6)
```

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

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

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

compile(*mode*, *target='cpu'*)

Recompile the just-in-time compiled numba ufuncs with a new calculation mode or target.

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.GFMeta.ufunc_modes*.
- **target** (*str, optional*) – The `target` keyword argument from `numba.vectorize`, either "cpu", "parallel", or "cuda". The default is "cpu". For extremely large fields the only supported target is "cpu" (which doesn't use numba it uses pure python to calculate the field arithmetic). The list of valid targets for this field is in *galois.GFMeta.ufunc_targets*.

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(3, order=2^8)

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

In [4]: GF.display("power"); a
Out[4]: GF(^25, order=2^8)

# Reset to the default display mode
In [5]: GF.display(); a
Out[5]: GF(3, 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)
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(254, order=2^8)

In [9]: with GF.display("poly"):
   ...:     print(a)
   ...:
GF(^7 + ^6 + ^5 + ^4 + ^3 + ^2 + , order=2^8)

In [10]: with GF.display("power"):
   ....:     print(a)
   ....:
GF(^88, 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)
```

(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 + 1, GF(2))
  is_primitive_poly: False
  primitive_element: GF(3, order=2^8)

In [13]: a = GF.Random(); a
Out[13]: GF(63, order=2^8)

In [14]: with GF.display("poly"):
   ....:     print(a)
   ....:
GF(x^5 + x^4 + x^3 + x^2 + x + 1, order=2^8)

In [15]: with GF.display("power"):
   ....:     print(a)
   ....:
GF(^142, order=2^8)
```

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(96, 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(28, 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

# galois.GF(7**5).degree
```

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]: print(GF.properties)
GF(2^8):
```

(continues on next page)

(continued from previous page)

```
  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]: a = GF.Random(); a
Out[3]: GF(159, order=2^8)

In [4]: with GF.display("poly"):
   ...:     print(a)
   ...:
GF(^7 + ^4 + ^3 + ^2 +  + 1, order=2^8)

In [5]: with GF.display("power"):
   ...:     print(a)
   ...:
GF(^46, 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]: 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 [8]: a = GF.Random(); a
Out[8]: GF(46, order=2^8)

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

In [10]: with GF.display("power"):
   ....:     print(a)
   ....:
GF(^9, order=2^8)
```

Type str

property dtypes

List of valid integer `numpy.dtype` objects that are compatible with this Galois field. Valid data types are signed and unsinged integers that can represent decimal values in $[0, p^m)$.

Examples

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

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

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

# galois.GF(7**5).dtypes
```

For field's with orders that cannot be represented by `numpy.int64`, the only valid dtype is `numpy.object_`.

```
In [4]: galois.GF(2**100).dtypes
Out[4]: [numpy.object_]

In [5]: galois.GF(36893488147419103183).dtypes
Out[5]: [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))
```

(continues on next page)

(continued from previous page)

```
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))

# galois.GF(7**5).irreducible_poly
```

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

# galois.GF(7**5).is_extension_field
```

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

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

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

# galois.GF(7**5).is_prime_field
```

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)'
```

(continues on next page)

(continued from previous page)

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

# galois.GF(7**5).name
```

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

# galois.GF(7**5).order
```

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
  order: 2
  irreducible_poly: Poly(x + 1, GF(2))
  is_primitive_poly: True
  primitive_element: GF(1, order=2)

In [2]: print(galois.GF(2**8).prime_subfield.properties)
GF(2):
  characteristic: 2
  degree: 1
  order: 2
  irreducible_poly: Poly(x + 1, GF(2))
  is_primitive_poly: True
  primitive_element: GF(1, order=2)
```

(continues on next page)

(continued from previous page)

```
In [3]: print(galois.GF(31).prime_subfield.properties)
GF(31):
  characteristic: 31
  degree: 1
  order: 31
  irreducible_poly: Poly(x + 28, GF(31))
  is_primitive_poly: True
  primitive_element: GF(3, order=31)

# print(galois.GF(7**5).prime_subfield.properties)
```

Type *galois.GFMeta*

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 $\mathrm{GF}(p^m) = \{0, 1, \alpha^1, \alpha^2, \ldots, \alpha^{p^m-2}\}$.

A primitive element is a root of the primitive polynomial (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)

# galois.GF(7**5).primitive_element
```

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 $\mathrm{GF}(p^m) = \{0, 1, \alpha^1, \alpha^2, \ldots, \alpha^{p^m-2}\}$.

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,
```

(continues on next page)

(continued from previous page)

```
    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)

# galois.GF(7**5).primitive_elements
```

Type int

property properties

A formmatted string displaying relevant properties of the Galois field.

Examples

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

In [2]: print(galois.GF(2**8).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]: print(galois.GF(31).properties)
GF(31):
  characteristic: 31
  degree: 1
  order: 31
  irreducible_poly: Poly(x + 28, GF(31))
  is_primitive_poly: True
  primitive_element: GF(3, order=31)

# print(galois.GF(7**5).properties)
```

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'

# galois.GF(7**5).ufunc_mode
```

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

property ufunc_target

The numba target for the JIT-compiled ufuncs, either "cpu", "parallel", or "cuda".

Examples

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

In [2]: galois.GF(2**8).ufunc_target
Out[2]: 'cpu'

In [3]: galois.GF(31).ufunc_target
```

(continues on next page)

(continued from previous page)

```
Out[3]: 'cpu'

# galois.GF(7**5).ufunc_target
```

Type str

property ufunc_targets

All supported ufunc targets for this Galois field array class.

Examples

```
In [1]: galois.GF(2).ufunc_targets
Out[1]: ['cpu', 'parallel', 'cuda']

In [2]: galois.GF(2**8).ufunc_targets
Out[2]: ['cpu', 'parallel', 'cuda']

In [3]: galois.GF(31).ufunc_targets
Out[3]: ['cpu', 'parallel', 'cuda']

In [4]: galois.GF(2**100).ufunc_targets
Out[4]: ['cpu']
```

Type list

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*) – List of polynomial coefficients $\{a_d, a_{d-1}, \ldots, a_1, a_0\}$ with type *galois.GFArray*, 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.GFArray, 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)))
```

---

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.GFArray, 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.GFArray, 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.GFArray, 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.GFArray, 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.GFArray, 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^4, 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(114x^5 + 210x^4 + 153x^3 + 69x^2 + 105x + 45, 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.GFArray, 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))
```

(continues on next page)

(continued from previous page)

```
In [7]: p(roots)
Out[7]: GF([0, 0, 0], order=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.GFArray, 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^6 + x^4 + x + 1, GF(2))

In [2]: p.derivative()
Out[2]: Poly(x^6 + 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 + x^10 + 6x^9 + 2x^8 + 6x^7 + 4x^6 + 4x^5 + x^4 + 6x^3 + x^2␣
↪+ x + 5, GF(7))

In [6]: p.derivative()
Out[6]: Poly(4x^10 + 3x^9 + 5x^8 + 2x^7 + 3x^5 + 6x^4 + 4x^3 + 4x^2 + 2x + 1,␣
↪GF(7))

In [7]: p.derivative(2)
Out[7]: Poly(5x^9 + 6x^8 + 5x^7 + x^4 + 3x^3 + 5x^2 + x + 2, GF(7))

In [8]: p.derivative(3)
Out[8]: Poly(3x^8 + 6x^7 + 4x^3 + 2x^2 + 3x + 1, 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(41x^7 + 17x^6 + 138x^5 + 109x^4 + 175x^3 + 184x^2 + 115x + 121,␣
↪GF(2^8))

In [12]: p.derivative()
Out[12]: Poly(41x^6 + 138x^4 + 175x^2 + 115, GF(2^8))
```

(continues on next page)

(continued from previous page)

```
# 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.GFArray* – 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.GFArray*

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^2, GF(2))

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(31x^5 + 196x^4 + 133x^3 + 41x^2 + 228x + 211, GF(2^8))

In [4]: b.field
Out[4]: <class 'numpy.ndarray over GF(2^8)'>
```

Type *galois.GFMeta*

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.GFArray*

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

galois.**GF**(*order*, *irreducible_poly=None*, *primitive_element=None*, *verify_irreducible=True*, *verify_primitive=True*, *mode='auto'*, *target='cpu'*)

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

The created class will be a subclass of *galois.GFArray* with metaclass *galois.GFMeta*. The *galois.GFArray* inheritance provides the numpy.ndarray functionality. The *galois.GFMeta* 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, 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. Default is `None` which uses the Conway polynomial $C_{p,m}$ obtained from *galois.conway_poly()*.
- **primitive_element** (*int, 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 $\mathrm{GF}(p)$ or its integer representation. The default is `None` which uses `galois.primitive_root(p)` for prime fields and `galois.primitive_element(irreducible_poly)` for extension fields.
- **verify_irreducible** (*bool, optional*) – Indicates whether to verify that the specified irreducible polynomial is in fact irreducible. The default is `True`. For large irreducible polynomials that are already known to be irreducible (and may take a long time to verify), this argument can be set to `False`.
- **verify_primitive** (*bool, optional*) – Indicates whether to verify that the specified primitive element is in fact a generator of the multiplicative group. The default is `True`.
- **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**16` will use the "jit-lookup" mode.
- **target** (*str, optional*) – The `target` keyword argument from `numba.vectorize()`, either `"cpu"`, `"parallel"`, or `"cuda"`.

Returns A new Galois field array class that is a subclass of *galois.GFArray* with *galois.GFMeta* metaclass.

Return type *galois.GFMeta*

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
```

(continues on next page)

(continued from previous page)

```
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))
  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
  irreducible_poly: Poly(x + 568, GF(571))
  is_primitive_poly: True
  primitive_element: GF(3, 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)
```

(continues on next page)

(continued from previous page)

```
GF(36893488147419103183):
  characteristic: 36893488147419103183
  degree: 1
  order: 36893488147419103183
  irreducible_poly: Poly(x + 36893488147419103180, GF(36893488147419103183))
  is_primitive_poly: True
  primitive_element: GF(3, order=36893488147419103183)
```

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

galois.**carmichael**(n)

Finds the smallest positive integer m such that $a^m \equiv 1 (\text{mod } n)$ 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 $a^m \equiv 1 (\text{mod } n)$ 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(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)
```

(continues on next page)

(continued from previous page)

```
# For prime n, phi and lambda are always n-1
In [5]: galois.euler_totient(13), galois.carmichael(13)
Out[5]: (12, 12)
```

---

galois.**conway_poly**(p, n)

Returns the degree-n Conway polynomial $C_{p,n}$ over $\mathrm{GF}(p)$.

A Conway polynomial is a an irreducible and primitive polynomial over $\mathrm{GF}(p)$ that provides a standard representation of $\mathrm{GF}(p^n)$ as a splitting field of $C_{p,n}$. Conway polynomials provide compatability between fields and their subfields, and hence are the common way to represent extension fields.

The Conway polynomial $C_{p,n}$ is defined as the lexicographically-minimal monic irreducible polynomial of degree n over $\mathrm{GF}(p)$ that is compatible with all $C_{p,m}$ for m dividing n.

This function uses Frank Luebeck's Conway polynomial database for fast lookup, not construction.

Parameters

- **p** (*int*) – The prime characteristic of the field $\mathrm{GF}(p)$.
- **n** (*int*) – The degree n of the Conway polynomial.

Returns The degree-n Conway polynomial $C_{p,n}$ over $\mathrm{GF}(p)$.

Return type *galois.Poly*

Raises **LookupError** – If the Conway polynomial $C_{p,n}$ is not found in Frank Luebeck's database.

Warning: If the $\mathrm{GF}(p)$ field hasn't previously been created, it will be created in this function since it's needed for the construction of the return polynomial.

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))
```

---

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.**euler_totient**(n)

Counts the positive integers (totatives) in $1 \leq k < n$ that are relatively prime to n, i.e. $gcd(n, k) = 1$.

Implements the Euler Totient function $\phi(n)$.

Parameters **n** (*int*) – A positive integer.

Returns The number of totatives that are relatively prime 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_totient(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
```

(continues on next page)

(continued from previous page)

```
# For prime n, phi is always n-1
In [5]: galois.euler_totient(13)
Out[5]: 12
```

galois.**fermat_primality_test**(n)

Probabilistic primality test of n.

This function implements Fermat's primality test. The test says that for an integer n, select an integer a coprime with n. If $a^{n-1} \equiv 1 (\text{mod } n)$, then n is prime or pseudoprime.

Parameters **n** (*int*) – A positive integer.

Returns `False` if n is known to be composite. `True` if n is prime or pseudoprime.

Return type bool

References

- https://oeis.org/A001262
- https://oeis.org/A001567

Examples

```
# List of some primes
In [1]: primes = [257, 24841, 65497]

In [2]: for prime in primes:
   ...:     is_prime = galois.fermat_primality_test(prime)
   ...:     p, k = galois.prime_factors(prime)
   ...:     print("Prime = {:5d}, Fermat's Prime Test = {}, Prime factors = {}".
↪format(prime, is_prime, list(p)))
   ...:
Prime =   257, Fermat's Prime Test = True, Prime factors = [257]
Prime = 24841, Fermat's Prime Test = True, Prime factors = [24841]
Prime = 65497, Fermat's Prime Test = True, Prime factors = [65497]

# List of some strong pseudoprimes with base 2
In [3]: pseudoprimes = [2047, 29341, 65281]

In [4]: for pseudoprime in pseudoprimes:
   ...:     is_prime = galois.fermat_primality_test(pseudoprime)
   ...:     p, k = galois.prime_factors(pseudoprime)
   ...:
↪print("Pseudoprime = {:5d}, Fermat's Prime Test = {}, Prime factors = {}".
↪format(pseudoprime, is_prime, list(p)))
   ...:
Pseudoprime =  2047, Fermat's Prime Test = True, Prime factors = [23, 89]
Pseudoprime = 29341, Fermat's Prime Test = True, Prime factors = [13, 37, 61]
Pseudoprime = 65281, Fermat's Prime Test = True, Prime factors = [97, 673]
```

galois.**gcd**(a, b)

Finds the integer multiplicands of a and b such that $ax + by = \gcd(a, b)$.

This implementation uses 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 = 2

In [2]: b = 13

In [3]: gcd, x, y = galois.gcd(a, b)

In [4]: gcd, x, y
Out[4]: (1, -6, 1)

In [5]: a*x + b*y == gcd
Out[5]: True
```

galois.**is_cyclic**(n)

Determines whether the multiplicative group $\mathbb{Z}_n^\times$ is cyclic.

The multiplicative group $\mathbb{Z}_n^\times$ is the set of positive integers $1 \leq a < n$ that are coprime with n. $\mathbb{Z}_n^\times$ being cyclic means that some primitive root (or generator) g can generate the group $\mathbb{Z}_n^\times = \{g, g^2, \ldots, g^k\}$, where k is order of the group. The order of the group is defined by Euler's totient function, $\phi(n) = k$. If $\mathbb{Z}_n^\times$ is cyclic, the number of primitive roots is found by $\phi(k)$ or $\phi(\phi(n))$.

$\mathbb{Z}_n^\times$ 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 $\mathbb{Z}_n^\times$ is cyclic.

Return type bool

References

- https://en.wikipedia.org/wiki/Primitive_root_modulo_n

Examples

The elements of $\mathbb{Z}_n^\times$ are the positive integers less than n that are coprime with n. For example when $n = 14$, then $\mathbb{Z}_{14}^\times = \{1, 3, 5, 9, 11, 13\}$.

```
# 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_totient(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))
   ...:
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_totient(phi)
Out[8]: True
```

A counterexample is $n = 15 = 3 * 5$, which doesn't fit the condition for cyclicness. $\mathbb{Z}_{15}^\times = \{1, 2, 4, 7, 8, 11, 13, 14\}$.

```
# n is of type p1^k1 * p2^k2, which is not cyclic
In [9]: n = 15
```

(continues on next page)

(continued from previous page)

```
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_totient(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

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(n)
Out[16]: 4
```

`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-n polynomial $f(x)$ over $\mathrm{GF}(p)$ for prime p is irreducible if and only if $f(x) \mid (x^{p^n} - x)$ and $\gcd(f(x),\ x^{p^{m_i}} - x) = 1$ for $1 \leq i \leq k$, where $m_i = n/p_i$ for the k prime divisors p_i of n.

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
- 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 requirement 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
```

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
```

---

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_primitive**(*poly*)

Checks whether the polynomial $f(x)$ over $\mathrm{GF}(p)$ is primitive.

A degree-n polynomial $f(x)$ over $\mathrm{GF}(p)$ is *primitive* if $f(x) \mid (x^k - 1)$ for $k = p^n - 1$ and no k less than $p^n - 1$.

Parameters **poly** (galois.Poly) – A polynomial $f(x)$ over $\mathrm{GF}(p)$.

Returns `True` if the polynomial is primitive.

Return type bool

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
```

---

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_totient*.

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))

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
```

(continues on next page)

(continued from previous page)

```
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
```

---

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)
Out[2]: True

In [3]: galois.primitive_roots(7)
Out[3]: [3, 5]
```

---

galois.**is_smooth**(n, B)

Determines if the positive integer n is B-smooth, i.e. all its prime factors satisfy $p \leq B$.

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.

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.**isqrt**(n)

Computes the integer square root of n such that $\text{isqrt}(n)^2 \leq n$.

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 such that $\text{isqrt}(n)^2 \leq n$.

Return type int

Examples

```
# Use a large Mersenne prime
In [1]: p = galois.mersenne_primes(2000)[-1]; p
Out[1]:␣
↪1040793219466439908192524032736408553861526224726670480531911235040360805967336029801223944173232

In [2]: sqrt_p = galois.isqrt(p); sqrt_p
Out[2]:␣
↪3226132699481594337650229932669505772017441235628244885631123722785761803162998767122846394796285

In [3]: sqrt_p**2 <= p
Out[3]: True

In [4]: (sqrt_p + 1)**2 <= p
Out[4]: False
```

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.**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
```

(continues on next page)

(continued from previous page)

```
In [7]: lcm == prime1 * prime2
Out[7]: True
```

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**(*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,
⌊
↪68647976601306097149819007990813932172694353001433054093944634591855431833976560521225596406614547
↪
⌊
↪53113799281676709868958820655246862732959311772703192319944413820040355986085224273916250226522922
↪
⌊
↪10407932194664399081925240327364085538615262247266704805319112350403608059673360298012239441732324

In [2]: galois.is_prime(p[-1])
Out[2]: True
```

galois.**miller_rabin_primality_test**(*n*, *a=None*, *rounds=1*)

Probabilistic primality test of n.

This function implements the Miller-Rabin primality test. The test says that for an integer n, select an integer a such that $a < n$. Factor $n - 1$ such that $2^s d = n - 1$. Then, n is composite, if $a^d \not\equiv 1 (\text{mod } n)$ and $a^{2^r d} \not\equiv n - 1 (\text{mod } n)$ for $1 \leq r < s$.

Parameters

- **n** (*int*) – A positive integer.
- **a** (*int, optional*) – Initial composite witness value, $1 \leq a < n$. On subsequent rounds, a will be a different value. The default is a random value.
- **rounds** (*int, optional*) – The number of iterations attempting to detect n as composite. Additional rounds will choose new a. Sufficient rounds have arbitrarily-high probability of detecting a composite.

Returns `False` if n is known to be composite. `True` if n is prime or pseudoprime.

Return type bool

References

- https://math.dartmouth.edu/~carlp/PDF/paper25.pdf
- https://oeis.org/A001262

Examples

```
# List of some primes
In [1]: primes = [257, 24841, 65497]

In [2]: for prime in primes:
   ...:     is_prime = galois.miller_rabin_primality_test(prime)
   ...:     p, k = galois.prime_factors(prime)
   ...:     print("Prime = {:5d}, Miller-Rabin Prime Test = {}, Prime factors = {}".
↪format(prime, is_prime, list(p)))
   ...:
Prime =   257, Miller-Rabin Prime Test = True, Prime factors = [257]
Prime = 24841, Miller-Rabin Prime Test = True, Prime factors = [24841]
Prime = 65497, Miller-Rabin Prime Test = True, Prime factors = [65497]

# List of some strong pseudoprimes with base 2
In [3]: pseudoprimes = [2047, 29341, 65281]

# Single round of Miller-Rabin, sometimes fooled by pseudoprimes
In [4]: for pseudoprime in pseudoprimes:
   ...:     is_prime = galois.miller_rabin_primality_test(pseudoprime)
   ...:     p, k = galois.prime_factors(pseudoprime)
   ...:
↪print("Pseudoprime = {:5d}, Miller-Rabin Prime Test = {}, Prime factors = {}".
↪format(pseudoprime, is_prime, list(p)))
   ...:
Pseudoprime =  2047, Miller-Rabin Prime Test = False, Prime factors = [23, 89]
Pseudoprime = 29341, Miller-Rabin Prime Test = False, Prime factors = [13, 37, 61]
Pseudoprime = 65281, Miller-Rabin Prime Test = False, Prime factors = [97, 673]

# 7 rounds of Miller-Rabin, never fooled by pseudoprimes
In [5]: for pseudoprime in pseudoprimes:
   ...:     is_prime = galois.miller_rabin_primality_test(pseudoprime, rounds=7)
   ...:     p, k = galois.prime_factors(pseudoprime)
   ...:
↪print("Pseudoprime = {:5d}, Miller-Rabin Prime Test = {}, Prime factors = {}".
↪format(pseudoprime, is_prime, list(p)))
   ...:
Pseudoprime =  2047, Miller-Rabin Prime Test = False, Prime factors = [23, 89]
Pseudoprime = 29341, Miller-Rabin Prime Test = False, Prime factors = [13, 37, 61]
Pseudoprime = 65281, Miller-Rabin Prime Test = False, Prime factors = [97, 673]
```

galois.**next_prime**(x)

Returns the nearest prime p, such that $p > x$.

Parameters **x** (*int*) – A positive integer.

Returns The nearest prime $p > x$.

Return type int

Examples

```
In [1]: galois.next_prime(13)
Out[1]: 17

In [2]: galois.next_prime(15)
Out[2]: 17
```

galois.**poly_exp_mod**(*poly*, *power*, *modulus*)

Efficiently exponentiates a polynomial $f(x)$ to the power k reducing by modulo $g(x)$, f^k mod g.

The algorithm is more efficient than exponentiating first and then reducing modulo $g(x)$. Instead, this algorithm repeatedly squares f, reducing modulo g at each step.

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(7x^10 + 20x^9 + 22x^8 + 25x^7 + 7x^6 + 10x^5 + 17x^4 + 10x^3 + 14x^2 +␣
↪3x + 4, GF(31))

In [3]: g = galois.Poly.Random(7, field=GF); g
Out[3]: Poly(5x^7 + 8x^6 + 12x^5 + 13x^3 + 23x^2 + 16x + 17, GF(31))

# %timeit f**200 % g
# 1.23 s ± 41.1 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
In [4]: f**200 % g
Out[4]: Poly(12x^6 + 19x^5 + 14x^4 + 2x^3 + 20x^2 + 19x + 18, GF(31))

# %timeit galois.poly_exp_mod(f, 200, g)
# 41.7 ms ± 468 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
In [5]: galois.poly_exp_mod(f, 200, g)
Out[5]: Poly(12x^6 + 19x^5 + 14x^4 + 2x^3 + 20x^2 + 19x + 18, GF(31))
```

galois.**poly_gcd**(*a*, *b*)

Finds the greatest common divisor of two polynomials $a(x)$ and $b(x)$ over GF(q).

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 $x(x)$, such that $ax + by = gcd(a, b)$.
- *galois.Poly* – Polynomial $y(x)$, such that $ax + by = gcd(a, b)$.

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_gcd(a, b)

# 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.**prev_prime**(x)

Returns the nearest prime p, such that $p \leq x$.

Parameters **x** (*int*) – A positive integer.

Returns The nearest prime $p \leq x$.

Return type int

Examples

```
In [1]: galois.prev_prime(13)
Out[1]: 13

In [2]: galois.prev_prime(15)
Out[2]: 13
```

galois.**prime_factors**(n)

Computes the prime factors of the positive integer n.

The integer n can be factored into $n = p_1^{e_1} p_2^{e_2} \ldots p_{k-1}^{e_{k-1}}$.

Steps:

1. Test if n is prime. If so, return `[n], [1]`.

2. Use trial division with a list of primes up to 10^6. If no residual factors, return the discovered prime factors.
3. Use Pollard's Rho algorithm to find a non-trivial factor of the residual. Continue until all are found.

Parameters **n** (*int*) – The positive integer to be factored.

Returns

- *list* – Sorted list of k prime factors $p = [p_1, p_2, \ldots, p_{k-1}]$ with $p_1 < p_2 < \cdots < p_{k-1}$.
- *list* – List of corresponding prime powers $e = [e_1, e_2, \ldots, e_{k-1}]$.

Examples

```
In [1]: p, e = galois.prime_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 =100000000000000000350000061

In [5]: galois.is_prime(prime)
Out[5]: True

In [6]: p, e = galois.prime_factors(prime - 1)

In [7]: p, e
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
```

galois.**primes**(n)

Returns all primes p for $p \leq n$.

Parameters **n** (*int*) – A positive integer.

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.**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))
```

(continues on next page)

(continued from previous page)

```
In [8]: galois.is_irreducible(f)
Out[8]: True

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**(*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_totient`.

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_totient(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_totient(3**2 - 1)
Out[12]: True
```

---

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_totient(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_totient(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]: 100000000000000035000054
```

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_totient(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(n)
Out[22]: 2
```

---

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_totient(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_totient(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_totient(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_totient(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_totient(galois.euler_totient(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_totient(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.**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]:␣
↪22298111106087631063095347963854661843222377239683469032547882212359453626536109487506041508496460
```

(continues on next page)

(continued from previous page)

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

```
$ openssl prime␣
↪2368617879269573822069968860872145920297525240780263923589368444796674235708331161265069278787731
1514D68EDB7C650F1FF713531A1A43255A4BE6D66EE1FDBD96F4EB32757C1B1BAF16A5933E24D45FAD6C6A814F3C8C14F3C
↪(2368617879269573822069968860872145920297525240780263923589368444796674235708331161265069278787731
↪is prime
```

galois.**totatives**(n)

Returns the positive integers (totatives) in $1 \leq k < n$ that are coprime with n, i.e. $gcd(n, k) = 1$.

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_totient(n); phi
Out[3]: 8

In [4]: len(totatives) == phi
Out[4]: True
```

np	Documentation of some native numpy functions when called on Galois field arrays.

6.2 np

Documentation of some native numpy functions when called on Galois field arrays.

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.

6.2.1 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([7, 2, 2, 6, 7], order=2^3)

In [3]: b = np.copy(a); b
Out[3]: GF([7, 2, 2, 6, 7], order=2^3)

In [4]: a[0] = 0; a
Out[4]: GF([0, 2, 2, 6, 7], order=2^3)

# b is unmodified
In [5]: b
Out[5]: GF([7, 2, 2, 6, 7], order=2^3)
```

6.2.2 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, 2],
```

(continues on next page)

(continued from previous page)

```
    [2, 4]], order=2^3)

In [3]: B = GF.Random((2,2)); B
Out[3]:
GF([[6, 6],
    [5, 5]], order=2^3)

In [4]: np.concatenate((A,B), axis=0)
Out[4]:
GF([[1, 2],
    [2, 4],
    [6, 6],
    [5, 5]], order=2^3)

In [5]: np.concatenate((A,B), axis=1)
Out[5]:
GF([[1, 2, 6, 6],
    [2, 4, 5, 5]], order=2^3)
```

6.2.3 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([2, 2, 4, 0, 3], order=2^3)

In [3]: np.insert(x, 1, [0,1,2,3])
Out[3]: GF([2, 0, 1, 2, 3, 2, 4, 0, 3], order=2^3)
```

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.

continues on next page

Table 18 – continued from previous page

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(x1, x2)	Computes the matrix multiplication of two Galois field arrays.

6.2.4 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([30,  9, 18,  4, 17, 13, 14, 17,  6, 21], order=31)

In [3]: y = GF.Random(10); y
Out[3]: GF([14,  4, 25, 24, 11, 21, 25,  7, 13,  4], order=31)

In [4]: np.add(x, y)
Out[4]: GF([13, 13, 12, 28, 28,  3,  8, 24, 19, 25], order=31)

In [5]: x + y
Out[5]: GF([13, 13, 12, 28, 28,  3,  8, 24, 19, 25], order=31)
```

6.2.5 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([ 4, 23,  5,  6, 23, 17, 14, 15,  4, 16], order=31)

In [3]: y = GF.Random(10); y
Out[3]: GF([ 1, 30, 28, 10,  4, 14, 29, 29, 28, 15], order=31)

In [4]: np.subtract(x, y)
Out[4]: GF([ 3, 24,  8, 27, 19,  3, 16, 17,  7,  1], order=31)

In [5]: x - y
Out[5]: GF([ 3, 24,  8, 27, 19,  3, 16, 17,  7,  1], order=31)
```

6.2.6 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([20, 13,  3, 25, 11, 19, 10, 17, 25,  9], order=31)

In [3]: y = GF.Random(10); y
Out[3]: GF([11,  8, 29, 14,  0, 18, 25, 14, 19, 13], order=31)

In [4]: np.multiply(x, y)
Out[4]: GF([ 3, 11, 25,  9,  0,  1,  2, 21, 10, 24], order=31)

In [5]: x * y
Out[5]: GF([ 3, 11, 25,  9,  0,  1,  2, 21, 10, 24], 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([ 4, 20, 12,  5, 24, 28, 27, 18, 30, 27], order=31)

In [8]: np.multiply(x, 3)
Out[8]: GF([12, 29,  5, 15, 10, 22, 19, 23, 28, 19], order=31)

In [9]: x * 3
Out[9]: GF([12, 29,  5, 15, 10, 22, 19, 23, 28, 19], order=31)
```

```
In [10]: print(GF.properties)
GF(31):
  characteristic: 31
  degree: 1
  order: 31
  irreducible_poly: Poly(x + 28, GF(31))
  is_primitive_poly: True
  primitive_element: GF(3, 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)
```

6.2.7 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([25, 12,  7,  8, 22, 20,  4, 12, 28, 28], order=31)

In [3]: y = GF.Random(10, low=1); y
Out[3]: GF([21,  1,  8, 13, 20, 24, 15,  2, 13,  3], order=31)

In [4]: z = np.divide(x, y); z
Out[4]: GF([13, 12, 28,  3, 29,  6, 23,  6, 26, 30], order=31)

In [5]: y * z
Out[5]: GF([25, 12,  7,  8, 22, 20,  4, 12, 28, 28], order=31)
```

```
In [6]: np.true_divide(x, y)
Out[6]: GF([13, 12, 28,  3, 29,  6, 23,  6, 26, 30], order=31)

In [7]: x / y
Out[7]: GF([13, 12, 28,  3, 29,  6, 23,  6, 26, 30], order=31)

In [8]: np.floor_divide(x, y)
Out[8]: GF([13, 12, 28,  3, 29,  6, 23,  6, 26, 30], order=31)

In [9]: x // y
Out[9]: GF([13, 12, 28,  3, 29,  6, 23,  6, 26, 30], order=31)
```

6.2.8 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([ 3,  4,  1, 25, 29, 25, 24, 11, 21,  8], order=31)

In [3]: y = np.negative(x); y
Out[3]: GF([28, 27, 30,  6,  2,  6,  7, 20, 10, 23], 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([28, 27, 30,  6,  2,  6,  7, 20, 10, 23], order=31)

In [6]: -1*x
Out[6]: GF([28, 27, 30,  6,  2,  6,  7, 20, 10, 23], order=31)
```

6.2.9 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([19, 29, 14, 17, 22], order=31)

In [3]: y = np.reciprocal(x); y
Out[3]: GF([18, 15, 20, 11, 24], order=31)

In [4]: x * y
Out[4]: GF([1, 1, 1, 1, 1], order=31)
```

```
In [5]: x ** -1
Out[5]: GF([18, 15, 20, 11, 24], order=31)

In [6]: GF(1) / x
Out[6]: GF([18, 15, 20, 11, 24], order=31)

In [7]: GF(1) // x
Out[7]: GF([18, 15, 20, 11, 24], order=31)
```

6.2.10 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([ 4, 22, 19, 29, 25, 14,  0,  9,  8, 25], order=31)
```

(continues on next page)

(continued from previous page)

```
In [3]: np.power(x, 3)
Out[3]: GF([ 2, 15,  8, 23,  1, 16,  0, 16, 16,  1], order=31)

In [4]: x ** 3
Out[4]: GF([ 2, 15,  8, 23,  1, 16,  0, 16, 16,  1], order=31)

In [5]: x * x * x
Out[5]: GF([ 2, 15,  8, 23,  1, 16,  0, 16, 16,  1], order=31)
```

```
In [6]: x = GF.Random(10, low=1); x
Out[6]: GF([16, 30, 23, 19, 30, 22, 15, 20, 16, 17], order=31)

In [7]: y = np.random.randint(-10, 10, 10); y
Out[7]: array([ -5,   8,   2,   9,  -4, -10,  -7,  -3, -10,  -1])

In [8]: np.power(x, y)
Out[8]: GF([ 1,  1,  2, 16,  1, 25, 27, 16,  1, 11], order=31)

In [9]: x ** y
Out[9]: GF([ 1,  1,  2, 16,  1, 25, 27, 16,  1, 11], order=31)
```

6.2.11 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([11, 15, 14,  2, 25, 10, 11,  7, 24,  5], order=31)

In [3]: np.square(x)
Out[3]: GF([28,  8, 10,  4,  5,  7, 28, 18, 18, 25], order=31)

In [4]: x ** 2
Out[4]: GF([28,  8, 10,  4,  5,  7, 28, 18, 18, 25], order=31)

In [5]: x * x
Out[5]: GF([28,  8, 10,  4,  5,  7, 28, 18, 18, 25], order=31)
```

6.2.12 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.GFMeta.primitive_element*. See `galois.GFArray.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([23,  3, 21, 29, 12, 21, 10, 28, 30, 10], order=31)

In [4]: y = np.log(x); y
Out[4]: array([27,  1, 29,  9, 19, 29, 14, 16, 15, 14])

In [5]: alpha ** y
Out[5]: GF([23,  3, 21, 29, 12, 21, 10, 28, 30, 10], order=31)
```

6.2.13 np.matmul

np.**matmul**(*x1*, *x2*)

Computes the matrix multiplication of two Galois field arrays.

References

- https://numpy.org/doc/stable/reference/generated/numpy.log.html

Examples

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

In [2]: x1 = GF.Random((3,4)); x1
Out[2]:
GF([[28, 20, 16, 24],
    [14, 20, 17, 30],
    [ 3, 24,  0,  7]], order=31)

In [3]: x2 = GF.Random((4,5)); x2
```

(continues on next page)

(continued from previous page)

```
Out[3]:
GF([[27, 27, 10, 21, 22],
    [28, 12, 27,  5, 27],
    [ 7, 28, 28, 11, 15],
    [12,  2, 21, 29, 18]], order=31)

In [4]: np.matmul(x1, x2)
Out[4]:
GF([[11,  4,  5, 10, 30],
    [22,  7, 19, 25,  0],
    [ 0, 11, 19, 14,  3]], order=31)

In [5]: x1 @ x2
Out[5]:
GF([[11,  4,  5, 10, 30],
    [22,  7, 19, 25,  0],
    [ 0, 11, 19, 14,  3]], order=31)
```

Linear Algebra

np.trace(x)	Returns the sum along the diagonal of a Galois field array.
np.matmul(x1, x2)	Computes the matrix multiplication of two Galois field arrays.
np.linalg.matrix_rank(x)	Returns the rank of a Galois field matrix.
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.inv(A)	Computes the inverse of the matrix.
np.linalg.solve(x)	Solves the system of linear equations.

6.2.14 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([[ 5, 19,  6, 14, 26, 23],
```

(continues on next page)

(continued from previous page)

```
    [22, 13,  1,  6, 16, 27],
    [11,  3,  5, 12,  4,  9],
    [ 0, 13,  5, 14,  4, 20],
    [ 6, 19,  3, 16, 12, 25]], order=31)

In [3]: np.trace(A)
Out[3]: GF(18, order=31)

In [4]: A[0,0] + A[1,1] + A[2,2] + A[3,3] + A[4,4]
Out[4]: GF(18, order=31)
```

```
In [5]: np.trace(A, offset=1)
Out[5]: GF(30, order=31)

In [6]: A[0,1] + A[1,2] + A[2,3] + A[3,4] + A[4,5]
Out[6]: GF(30, order=31)
```

6.2.15 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([[15,  5,  4, 10],
    [ 0, 24, 24,  4],
```

(continues on next page)

(continued from previous page)

```
    [ 1,  9, 24,  1],
    [ 6, 22,  0, 29]], order=31)

In [6]: A[:,2] = A[:,1] * GF(17); A
Out[6]:
GF([[15,  5, 23, 10],
    [ 0, 24,  5,  4],
    [ 1,  9, 29,  1],
    [ 6, 22,  2, 29]], order=31)

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([[ 1,  8, 20, 18],
    [21,  6, 26, 12],
    [13,  3, 20, 15],
    [17, 15,  6, 29]], order=31)

In [10]: A[3,:] = A[2,:] * GF(8); A
Out[10]:
GF([[ 1,  8, 20, 18],
    [21,  6, 26, 12],
    [13,  3, 20, 15],
    [11, 24,  5, 27]], order=31)

In [11]: np.linalg.matrix_rank(A)
Out[11]: 3
```

6.2.16 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
```

(continues on next page)

(continued from previous page)

```
Out[2]:
GF([[14, 29, 14],
    [15, 16, 10],
    [11, 25, 25]], order=31)

In [3]: np.linalg.matrix_power(A, 3)
Out[3]:
GF([[15,  7,  8],
    [ 7, 17, 14],
    [17, 20, 15]], order=31)

In [4]: A @ A @ A
Out[4]:
GF([[15,  7,  8],
    [ 7, 17, 14],
    [17, 20, 15]], 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([[ 7,  0, 30],
    [ 5,  1, 29],
    [ 3,  7,  1]], order=31)

In [8]: np.linalg.matrix_power(A, -3)
Out[8]:
GF([[19, 29,  8],
    [ 7, 24, 18],
    [17, 25,  1]], order=31)

In [9]: A_inv = np.linalg.inv(A)

In [10]: A_inv @ A_inv @ A_inv
Out[10]:
GF([[19, 29,  8],
    [ 7, 24, 18],
    [17, 25,  1]], order=31)
```

6.2.17 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([[17,  1],
    [25, 29]], order=31)

In [3]: np.linalg.det(A)
Out[3]: GF(3, order=31)

In [4]: A[0,0]*A[1,1] - A[0,1]*A[1,0]
Out[4]: GF(3, order=31)
```

6.2.18 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([[13, 21,  5],
```

(continues on next page)

(continued from previous page)

```
    [ 1, 25, 22],
    [10,  5, 21]], order=31)

In [4]: A_inv = np.linalg.inv(A); A_inv
Out[4]:
GF([[26,  8, 12],
    [23, 13,  6],
    [22, 30, 18]], order=31)

In [5]: A_inv @ A
Out[5]:
GF([[1, 0, 0],
    [0, 1, 0],
    [0, 0, 1]], order=31)
```

6.2.19 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([[19,  0, 13,  6],
    [22,  6, 18,  8],
    [15, 13, 13,  0],
    [23,  0, 16, 27]], order=31)

In [4]: b = GF.Random(4); b
Out[4]: GF([ 9, 19,  3, 26], order=31)

In [5]: x = np.linalg.solve(A, b); x
Out[5]: GF([30, 26,  4, 27], order=31)
```

(continues on next page)

(continued from previous page)

```
In [6]: A @ x
Out[6]: GF([ 9, 19,  3, 26], order=31)
```

```
In [7]: GF = galois.GF(31)

# 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([[11, 25, 18,  5],
    [ 5,  1,  6,  7],
    [ 5, 19, 17,  6],
    [17,  2, 12, 21]], order=31)

In [10]: B = GF.Random((4,2)); B
Out[10]:
GF([[ 8, 12],
    [ 0,  9],
    [28, 17],
    [12, 12]], order=31)

In [11]: X = np.linalg.solve(A, B); X
Out[11]:
GF([[21, 20],
    [28, 21],
    [21, 16],
    [25, 19]], order=31)

In [12]: A @ X
Out[12]:
GF([[ 8, 12],
    [ 0,  9],
    [28, 17],
    [12, 12]], order=31)
```

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([25, 23, 29, 14, 16, 10, 17,  4, 27, 13], order=31)

In [3]: y = GF.Random(10); y
Out[3]: GF([28,  4, 15,  6,  6, 26, 26,  6, 24, 30], order=31)

In [4]: np.add(x, y)
Out[4]: GF([22, 27, 13, 20, 22,  5, 12, 10, 20, 12], order=31)

In [5]: x + y
Out[5]: GF([22, 27, 13, 20, 22,  5, 12, 10, 20, 12], order=31)
```

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([[4, 5],
    [3, 7]], order=2^3)

In [3]: B = GF.Random((2,2)); B
Out[3]:
GF([[1, 0],
    [5, 5]], order=2^3)

In [4]: np.concatenate((A,B), axis=0)
Out[4]:
GF([[4, 5],
    [3, 7],
    [1, 0],
    [5, 5]], order=2^3)

In [5]: np.concatenate((A,B), axis=1)
Out[5]:
GF([[4, 5, 1, 0],
    [3, 7, 5, 5]], order=2^3)
```

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([10, 29, 19, 30, 28, 16,  0, 18,  1,  0], order=31)

In [3]: y = GF.Random(10, low=1); y
Out[3]: GF([30, 30, 27,  1,  6,  9, 30, 25,  1, 19], order=31)

In [4]: z = np.divide(x, y); z
Out[4]: GF([21,  2,  3, 30, 15, 19,  0, 28,  1,  0], order=31)

In [5]: y * z
Out[5]: GF([10, 29, 19, 30, 28, 16,  0, 18,  1,  0], order=31)
```

```
In [6]: np.true_divide(x, y)
Out[6]: GF([21,  2,  3, 30, 15, 19,  0, 28,  1,  0], order=31)

In [7]: x / y
Out[7]: GF([21,  2,  3, 30, 15, 19,  0, 28,  1,  0], order=31)

In [8]: np.floor_divide(x, y)
Out[8]: GF([21,  2,  3, 30, 15, 19,  0, 28,  1,  0], order=31)

In [9]: x // y
Out[9]: GF([21,  2,  3, 30, 15, 19,  0, 28,  1,  0], order=31)
```

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([7, 2, 7, 2, 5], order=2^3)

In [3]: np.insert(x, 1, [0,1,2,3])
Out[3]: GF([7, 0, 1, 2, 3, 2, 7, 2, 5], order=2^3)
```

np.**log**(*x*)

Computes the logarithm (base `GF.primitive_element`) of a Galois field array element-wise.

Calling *np.log()* implicitly uses base *galois.GFMeta.primitive_element*. See `galois.GFArray.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([14,  6, 27, 24,  3, 26,  9,  2,  5,  2], order=31)

In [4]: y = np.log(x); y
Out[4]: array([22, 25,  3, 13,  1,  5,  2, 24, 20, 24])

In [5]: alpha ** y
Out[5]: GF([14,  6, 27, 24,  3, 26,  9,  2,  5,  2], order=31)
```

np.**matmul**(*x1*, *x2*)

Computes the matrix multiplication of two Galois field arrays.

References

- https://numpy.org/doc/stable/reference/generated/numpy.log.html

Examples

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

In [2]: x1 = GF.Random((3,4)); x1
Out[2]:
GF([[19, 16, 14, 25],
    [26,  3, 11, 14],
    [22,  8, 12, 27]], order=31)

In [3]: x2 = GF.Random((4,5)); x2
Out[3]:
GF([[14,  4, 19, 18, 18],
    [13, 13, 18,  1,  2],
    [ 7,  1, 21, 26,  8],
    [21, 21,  4, 11, 13]], order=31)
```

(continues on next page)

(continued from previous page)

```
In [4]: np.matmul(x1, x2)
Out[4]:
GF([[12, 17, 20,  5,  5],
    [30, 14, 29, 12,  0],
    [ 9, 27, 23, 21, 22]], order=31)

In [5]: x1 @ x2
Out[5]:
GF([[12, 17, 20,  5,  5],
    [30, 14, 29, 12,  0],
    [ 9, 27, 23, 21, 22]], order=31)
```

---

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([17, 26,  6, 17, 12,  4, 24,  9, 15, 16], order=31)

In [3]: y = GF.Random(10); y
Out[3]: GF([ 7, 27, 15, 25, 16, 22, 15,  2,  6,  3], order=31)

In [4]: np.multiply(x, y)
Out[4]: GF([26, 20, 28, 22,  6, 26, 19, 18, 28, 17], order=31)

In [5]: x * y
Out[5]: GF([26, 20, 28, 22,  6, 26, 19, 18, 28, 17], 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([ 8,  7, 27,  0, 20,  6,  5, 11, 17,  8], order=31)

In [8]: np.multiply(x, 3)
Out[8]: GF([24, 21, 19,  0, 29, 18, 15,  2, 20, 24], order=31)

In [9]: x * 3
Out[9]: GF([24, 21, 19,  0, 29, 18, 15,  2, 20, 24], order=31)
```

```
In [10]: print(GF.properties)
GF(31):
  characteristic: 31
  degree: 1
  order: 31
  irreducible_poly: Poly(x + 28, GF(31))
  is_primitive_poly: True
  primitive_element: GF(3, 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.**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, 15, 29, 25,  1,  3, 23,  9,  5, 28], order=31)

In [3]: y = np.negative(x); y
Out[3]: GF([13, 16,  2,  6, 30, 28,  8, 22, 26,  3], 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, 16,  2,  6, 30, 28,  8, 22, 26,  3], order=31)

In [6]: -1*x
Out[6]: GF([13, 16,  2,  6, 30, 28,  8, 22, 26,  3], order=31)
```

---

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([25, 22, 29,  1,  1, 29,  6, 14,  4,  8], order=31)

In [3]: np.power(x, 3)
Out[3]: GF([ 1, 15, 23,  1,  1, 23, 30, 16,  2, 16], order=31)

In [4]: x ** 3
Out[4]: GF([ 1, 15, 23,  1,  1, 23, 30, 16,  2, 16], order=31)

In [5]: x * x * x
Out[5]: GF([ 1, 15, 23,  1,  1, 23, 30, 16,  2, 16], order=31)
```

```
In [6]: x = GF.Random(10, low=1); x
Out[6]: GF([27, 21, 14,  6,  7,  9, 12, 14,  3,  6], order=31)

In [7]: y = np.random.randint(-10, 10, 10); y
Out[7]: array([-8, -4,  1, -6,  8, -3, -8,  1,  8, -4])

In [8]: np.power(x, y)
Out[8]: GF([16, 19, 14,  1, 10,  2,  7, 14, 20,  5], order=31)

In [9]: x ** y
Out[9]: GF([16, 19, 14,  1, 10,  2,  7, 14, 20,  5], order=31)
```

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([21,  8,  3, 14, 18], order=31)

In [3]: y = np.reciprocal(x); y
Out[3]: GF([ 3,  4, 21, 20, 19], order=31)
```

(continues on next page)

(continued from previous page)

```
In [4]: x * y
Out[4]: GF([1, 1, 1, 1, 1], order=31)
```

```
In [5]: x ** -1
Out[5]: GF([ 3,  4, 21, 20, 19], order=31)

In [6]: GF(1) / x
Out[6]: GF([ 3,  4, 21, 20, 19], order=31)

In [7]: GF(1) // x
Out[7]: GF([ 3,  4, 21, 20, 19], order=31)
```

---

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([24, 10, 21, 13, 10, 16,  1,  3,  6, 18], order=31)

In [3]: np.square(x)
Out[3]: GF([18,  7,  7, 14,  7,  8,  1,  9,  5, 14], order=31)

In [4]: x ** 2
Out[4]: GF([18,  7,  7, 14,  7,  8,  1,  9,  5, 14], order=31)

In [5]: x * x
Out[5]: GF([18,  7,  7, 14,  7,  8,  1,  9,  5, 14], order=31)
```

---

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
```

(continues on next page)

(continued from previous page)

```
Out[2]: GF([20, 10, 25, 27, 13, 29, 14,  2,  7,  2], order=31)

In [3]: y = GF.Random(10); y
Out[3]: GF([ 5,  4, 29,  0, 13, 20, 10, 24, 11,  0], order=31)

In [4]: np.subtract(x, y)
Out[4]: GF([15,  6, 27, 27,  0,  9,  4,  9, 27,  2], order=31)

In [5]: x - y
Out[5]: GF([15,  6, 27, 27,  0,  9,  4,  9, 27,  2], order=31)
```

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX

N

O

P

R

S

T

U

V

Z