galois

Matt Hostetter

Sep 08, 2021

CONTENTS

A performant NumPy extension for Galois fields

- Supports all Galois fields $\mathrm{GF}(p^m)$, even arbitrarily-large fields!
- **Faster** than native NumPy! `GF(x) * GF(y)` is faster than `(x * y) % p` for $\mathrm{GF}(p)$
- Seamless integration with NumPy – normal NumPy functions work on Galois field arrays
- Linear algebra on Galois field matrices using normal `np.linalg` functions
- Functions to generate irreducible, primitive, and Conway polynomials
- Polynomials over Galois fields with *`galois.Poly`*
- Forward error correction codes with *`galois.BCH`* and *`galois.ReedSolomon`*
- Fibonacci and Galois linear feedback shift registers with *`galois.LFSR`*, both binary and p-ary
- Various number theoretic functions
- Integer factorization and accompanying algorithms
- Prime number generation and primality testing

CHAPTER ONE

INSTALLATION

1.1 Install with pip

The latest released 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`.

1.2 Install for development

The latest code from `master` can be checked out and installed locally in an "editable" fashion. The "editable" install allows local changes to the `galois/` folder to be seen system-wide upon running `import galois`.

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

Also, feel free to fork *galois* on GitHub, clone your fork, make changes, and contribute back with a pull request!

1.3 Install for development with min dependencies

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

Listing 1: requirements-min.txt

```
numpy==1.17.3
numba==0.53
```

`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
```

(continues on next page)

(continued from previous page)

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

CHAPTER TWO

VERSIONING

This project uses semantic versioning. Releases are versioned `major.minor.patch`. Major releases introduce API-changing features. Minor releases add features and are backwards compatible with other releases in `major.x.x`. Patch releases fix bugs in a minor release and are backwards compatible with other releases in `major.minor.x`.

Releases before `1.0.0` are alpha and beta releases. Alpha releases are `0.0.alpha`. There is no API compatibility guarantee for them. They can be thought of as `0.0.alpha-major`. Beta releases are `0.beta.x` and are API compatible. They can be thought of as `0.beta-major.beta-minor`.

CHAPTER

THREE

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).

3.1 Galois field arrays

3.1.1 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.FieldArray* (which itself subclasses `numpy.ndarray`) and have *galois.FieldClass* as their metaclass.

```
In [5]: isinstance(GF256, galois.FieldClass)
Out[5]: True

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

In [7]: issubclass(GF256, np.ndarray)
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.FieldClass*.

```
# Summarizes some properties of the Galois field
In [8]: print(GF256.properties)
GF(2^8):
  characteristic: 2
  degree: 8
  order: 256
  irreducible_poly: x^8 + x^4 + x^3 + x^2 + 1
  is_primitive_poly: True
  primitive_element: x

# 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: x + 36893488147419103180
  is_primitive_poly: True
  primitive_element: 3

In [11]: GF = galois.GF(2**100); print(GF.properties)
GF(2^100):
  characteristic: 2
  degree: 100
  order: 1267650600228229401496703205376
  irreducible_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
  is_primitive_poly: True
  primitive_element: x
```

3.1.2 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([ 91, 203, 196, 251, 118, 153,  41,  28, 133, 152])

# Explicit Galois field array creation -- a copy is performed
In [13]: GF256(array)
Out[13]: GF([ 91, 203, 196, 251, 118, 153,  41,  28, 133, 152], 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([ 91, 203, 196, 251, 118, 153,  41,  28, 133, 152], 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.FieldArray.Zeros()*, *galois.FieldArray.Ones()*, *galois.FieldArray.Identity()*, *galois.FieldArray.Range()*, *galois.FieldArray.Random()*, *galois.FieldArray.Elements()*

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

```
In [18]: a = GF256(["x^6 + 1", 0, 2, "1", "x^5 + x^4 + x"]); a
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]: a
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(a)
   ....:
GF([^191, 0, , 1, ^194], order=2^8)

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

3.1.3 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([[155, 190, 192,  29,   9],
    [206, 231, 185, 155,  57]], order=2^8)

In [24]: y = GF256.Random(5); y
Out[24]: GF([239, 188, 234,   9, 152], order=2^8)

# y is broadcast over the last dimension of x
In [25]: x + y
Out[25]:
GF([[116,   2,  42,  20, 145],
    [ 33,  91,  83, 146, 161]], order=2^8)
```

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

```
In [26]: A = GF256.Random((3,3)); A
Out[26]:
GF([[188,  62, 183],
    [ 96, 107,  41],
    [ 67, 147, 175]], order=2^8)
```

(continues on next page)

(continued from previous page)

```
# Ensure A is invertible
In [27]: while np.linalg.matrix_rank(A) < 3:
   ....:     A = GF256.Random((3,3)); A
   ....:

In [28]: b = GF256.Random(3); b
Out[28]: GF([ 79, 115, 168], order=2^8)

In [29]: x = np.linalg.solve(A, b); x
Out[29]: GF([ 49, 244, 236], order=2^8)

In [30]: np.array_equal(A @ x, b)
Out[30]: True
```

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

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

3.1.5 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 [31]: X = GF256.Random((2,5)); X
Out[31]:
GF([[110, 134, 198, 125,  58],
    [202, 139, 187, 117, 143]], order=2^8)

In [32]: np.multiply.reduce(X, axis=0)
Out[32]: GF([ 16, 231, 131,  58,  39], order=2^8)
```

```
In [33]: x = GF256.Random(5); x
Out[33]: GF([ 63, 177, 128, 129, 190], order=2^8)

In [34]: y = GF256.Random(5); y
Out[34]: GF([130, 204,  15, 193,   3], order=2^8)

In [35]: np.multiply.outer(x, y)
Out[35]:
GF([[136, 148,  88, 178,  65],
    [201, 242, 209,  92, 206],
    [ 14, 218, 211,  20, 157],
    [140,  22, 220, 213, 158],
    [  4, 130, 132, 103, 223]], order=2^8)
```

3.2 Polynomials over Galois fields

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.FieldArray* 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.FieldArray* or an "array-like" object with the `field` keyword argument.

```
In [36]: p = galois.Poly([172, 22, 0, 0, 225], field=GF256); p
Out[36]: Poly(172x^4 + 22x^3 + 225, GF(2^8))

In [37]: coeffs = GF256([33, 17, 0, 225]); coeffs
Out[37]: GF([ 33,  17,   0, 225], order=2^8)

In [38]: p = galois.Poly(coeffs, order="asc"); p
Out[38]: Poly(225x^3 + 17x + 33, GF(2^8))
```

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

```
In [39]: print(p)
Poly(225x^3 + 17x + 33, GF(2^8))

In [40]: 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.String()*, *galois.Poly.Degrees()*, *galois.Poly.Roots()*

```
# Construct a polynomial by specifying its roots
In [41]: q = galois.Poly.Roots([155, 37], field=GF256); q
Out[41]: Poly(x^2 + 190x + 71, GF(2^8))

In [42]: q.roots()
Out[42]: GF([ 37, 155], order=2^8)
```

Polynomial arithmetic is performed using Python operators.

```
In [43]: p
Out[43]: Poly(225x^3 + 17x + 33, GF(2^8))

In [44]: q
Out[44]: Poly(x^2 + 190x + 71, GF(2^8))

In [45]: p + q
Out[45]: Poly(225x^3 + x^2 + 175x + 102, GF(2^8))

In [46]: divmod(p, q)
Out[46]: (Poly(225x + 57, GF(2^8)), Poly(56x + 104, GF(2^8)))
```

(continues on next page)

(continued from previous page)

```
In [47]: p ** 2
Out[47]: Poly(171x^6 + 28x^2 + 117, GF(2^8))
```

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

```
In [48]: p = galois.Poly.Degrees([4, 3, 0], [172, 22, 225], field=GF256); p
Out[48]: Poly(172x^4 + 22x^3 + 225, GF(2^8))

# Evaluate the polynomial at a single value
In [49]: p(1)
Out[49]: GF(91, order=2^8)

In [50]: x = GF256.Random((2,5)); x
Out[50]:
GF([[230,   2,  35, 152,  67],
    [  3, 211, 214,  43, 231]], order=2^8)

# Evaluate the polynomial at an array of values
In [51]: p(x)
Out[51]:
GF([[161,  67,  64,  90, 108],
    [141, 129, 242,   1,  63]], 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 [52]: p = GF256.irreducible_poly; p
Out[52]: Poly(x^8 + x^4 + x^3 + x^2 + 1, GF(2))

In [53]: GF256.is_primitive_poly
Out[53]: True

# Notice the primitive element is in GF(2^8)
In [54]: alpha = GF256.primitive_element; alpha
Out[54]: GF(2, order=2^8)

# Since p(x) is a primitive polynomial, alpha is one of its roots
In [55]: p(alpha, field=GF256)
Out[55]: GF(0, order=2^8)
```

3.3 Forward error correction codes

To demonstrate the FEC code API, here is an example using BCH codes. Other FEC codes have a similar API.

```
In [56]: import numpy as np

In [57]: import galois
```

(continues on next page)

(continued from previous page)

```
In [58]: bch = galois.BCH(15, 7); bch
Out[58]: <BCH Code: [15, 7, 5] over GF(2)>

In [59]: bch.generator_poly
Out[59]: Poly(x^8 + x^7 + x^6 + x^4 + 1, GF(2))

# The error-correcting capability
In [60]: bch.t
Out[60]: 2
```

A message can be either a 1-D vector or a 2-D matrix of messages. Shortened codes are also supported. See the docs for more details.

```
# Create a matrix of two messages
In [61]: M = galois.GF2.Random((2, bch.k)); M
Out[61]:
GF([[1, 0, 1, 1, 0, 0, 0],
    [1, 0, 0, 0, 0, 1, 1]], order=2)
```

Encoding the message(s) is performed with *galois.BCH.encode()*.

```
In [62]: C = bch.encode(M); C
Out[62]:
GF([[1, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 1, 1, 1, 1],
    [1, 0, 0, 0, 0, 1, 1, 0, 1, 0, 0, 1, 0, 1, 0]], order=2)
```

Decoding the codeword(s) is performed with *galois.BCH.decode()*.

```
# Corrupt the first bit in each codeword
In [63]: C[:,0] ^= 1; C
Out[63]:
GF([[0, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 1, 1, 1, 1],
    [0, 0, 0, 0, 0, 1, 1, 0, 1, 0, 0, 1, 0, 1, 0]], order=2)

In [64]: bch.decode(C)
Out[64]:
GF([[1, 0, 1, 1, 0, 0, 0],
    [1, 0, 0, 0, 0, 1, 1]], order=2)
```

CHAPTER

FOUR

TUTORIALS

4.1 Intro to Galois Fields: Prime Fields

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

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

4.1.1 Elements

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

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

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

In [2]: print(GF7.properties)
GF(7):
  characteristic: 7
  degree: 1
  order: 7
  irreducible_poly: x + 4
  is_primitive_poly: True
  primitive_element: 3
```

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

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

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

4.1.2 Arithmetic mod p

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

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

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

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

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

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

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

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

In [9]: x + y
Out[9]: GF([1, 5, 1, 3, 3, 6, 5, 2, 0, 3], order=7)

In [10]: x - y
Out[10]: GF([2, 3, 1, 1, 5, 4, 6, 2, 5, 5], order=7)

In [11]: x * y
Out[11]: GF([1, 4, 0, 2, 3, 5, 6, 0, 6, 3], order=7)
```

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

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

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

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

(continues on next page)

(continued from previous page)

```
    5  5 │ 6 │ 0 │ 1 │ 2 │ 3 │ 4

    6  6 │ 0 │ 1 │ 2 │ 3 │ 4 │ 5

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

 x - y  0 │ 1 │ 2 │ 3 │ 4 │ 5 │ 6

    0  0 │ 6 │ 5 │ 4 │ 3 │ 2 │ 1

    1  1 │ 0 │ 6 │ 5 │ 4 │ 3 │ 2

    2  2 │ 1 │ 0 │ 6 │ 5 │ 4 │ 3

    3  3 │ 2 │ 1 │ 0 │ 6 │ 5 │ 4

    4  4 │ 3 │ 2 │ 1 │ 0 │ 6 │ 5

    5  5 │ 4 │ 3 │ 2 │ 1 │ 0 │ 6

    6  6 │ 5 │ 4 │ 3 │ 2 │ 1 │ 0

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

 x * y  0 │ 1 │ 2 │ 3 │ 4 │ 5 │ 6

    0  0 │ 0 │ 0 │ 0 │ 0 │ 0 │ 0

    1  0 │ 1 │ 2 │ 3 │ 4 │ 5 │ 6

    2  0 │ 2 │ 4 │ 6 │ 1 │ 3 │ 5

    3  0 │ 3 │ 6 │ 2 │ 5 │ 1 │ 4

    4  0 │ 4 │ 1 │ 5 │ 2 │ 6 │ 3

    5  0 │ 5 │ 3 │ 1 │ 6 │ 4 │ 2

    6  0 │ 6 │ 5 │ 4 │ 3 │ 2 │ 1
```

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

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

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

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

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

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

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

If $a = 4$ is a field element of $\mathrm{GF}(7)$ and $b = 7$, the prime characteristic, then $x = a^{-1}$ in $\mathrm{GF}(7)$. Note, the GCD will always be 1 because b is prime.

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

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

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

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

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

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

```
In [23]: print(GF7.arithmetic_table("/"))

 x / y   1 | 2 | 3 | 4 | 5 | 6

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

(continues on next page)

(continued from previous page)

```
```

4.1.3 Primitive elements

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

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

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

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

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

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

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

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

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

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

 Power | Polynomial | Vector | Integer

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

(continues on next page)

(continued from previous page)

```
 3^5 │      5      │  [5]  │    5
```

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

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

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

 Power │ Polynomial │ Vector │ Integer

   0   │     0      │  [0]   │    0
───────────────────────────────────────
  5^0  │     1      │  [1]   │    1
───────────────────────────────────────
  5^1  │     5      │  [5]   │    5
───────────────────────────────────────
  5^2  │     4      │  [4]   │    4
───────────────────────────────────────
  5^3  │     6      │  [6]   │    6
───────────────────────────────────────
  5^4  │     2      │  [2]   │    2
───────────────────────────────────────
  5^5  │     3      │  [3]   │    3
```

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

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

 Power │ Polynomial │ Vector │ Integer

   0   │     0      │  [0]   │    0
───────────────────────────────────────
  2^0  │     1      │  [1]   │    1
───────────────────────────────────────
  2^1  │     2      │  [2]   │    2
───────────────────────────────────────
  2^2  │     4      │  [4]   │    4
───────────────────────────────────────
  2^3  │     1      │  [1]   │    1
───────────────────────────────────────
  2^4  │     2      │  [2]   │    2
───────────────────────────────────────
  2^5  │     4      │  [4]   │    4
```

4.2 Intro to Galois Fields: Extension Fields

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

4.2.1 Elements

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

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

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

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

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

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

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

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

 Power │ Polynomial │ Vector │ Integer

   0   │     0      │ [0, 0] │    0
──────────────────────────────────────
  x^0  │     1      │ [0, 1] │    1
──────────────────────────────────────
  x^1  │     x      │ [1, 0] │    3
──────────────────────────────────────
  x^2  │   x + 1    │ [1, 1] │    4
──────────────────────────────────────
  x^3  │   2x + 1   │ [2, 1] │    7
──────────────────────────────────────
  x^4  │     2      │ [0, 2] │    2
```

(continues on next page)

(continued from previous page)

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

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

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

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

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

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

In addition to scalars, these conversions work for arrays.

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

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

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

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

```
In [11]: x = GF9.Elements(); x
Out[11]: GF([0, 1, 2, 3, 4, 5, 6, 7, 8], order=3^2)

# Temporarily print x using the power representation
In [12]: with GF9.display("power"):
   ....:     print(x)
   ....:
GF([0, 1, ^4, , ^2, ^7, ^5, ^3, ^6], order=3^2)

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

(continues on next page)

(continued from previous page)

```
# Reset the display mode to the integer representation
In [14]: GF9.display(); x
Out[14]: GF([0, 1, 2, 3, 4, 5, 6, 7, 8], order=3^2)

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

4.2.2 Arithmetic mod p(x)

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

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

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

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

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

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

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

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

In [21]: a.integer
```

(continues on next page)

(continued from previous page)

```
Out[21]: 5

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

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

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

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

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

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

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

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

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

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

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

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

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

(continues on next page)

(continued from previous page)

```
───────────────────────────────────
     7   7 │ 8 │ 6 │ 1 │ 2 │ 0 │ 4 │ 5 │ 3
───────────────────────────────────
     8   8 │ 6 │ 7 │ 2 │ 0 │ 1 │ 5 │ 3 │ 4


In [31]: with GF9.display("poly"):
   ....:     print(GF9.arithmetic_table("+"))
   ....:

  x + y    0    │    1    │    2    │         │   + 1   │   + 2   │    2    │  2 + 1  │  2 + 2

      0    0    │    1    │    2    │         │   + 1   │   + 2   │    2    │  2 + 1  │  2 + 2
────────────────────────────────────────────────────────────────────────────────
      1    1    │    2    │    0    │   + 1   │   + 2   │         │  2 + 1  │  2 + 2  │    2
────────────────────────────────────────────────────────────────────────────────
      2    2    │    0    │    1    │   + 2   │         │   + 1   │  2 + 2  │    2    │  2 + 1
────────────────────────────────────────────────────────────────────────────────
             │   + 1   │   + 2   │    2    │  2 + 1  │  2 + 2  │    0    │    1    │    2
────────────────────────────────────────────────────────────────────────────────
   + 1   + 1 │   + 2   │         │  2 + 1  │  2 + 2  │    2    │    1    │    2    │    0
────────────────────────────────────────────────────────────────────────────────
   + 2   + 2 │         │   + 1   │  2 + 2  │    2    │  2 + 1  │    2    │    0    │    1
────────────────────────────────────────────────────────────────────────────────
     2     2 │  2 + 1  │  2 + 2  │    0    │    1    │    2    │         │   + 1   │   + 2
────────────────────────────────────────────────────────────────────────────────
  2 + 1 2 + 1 │  2 + 2  │    2    │    1    │    2    │    0    │   + 1   │   + 2   │
────────────────────────────────────────────────────────────────────────────────
  2 + 2 2 + 2 │    2    │  2 + 1  │    2    │    0    │    1    │   + 2   │         │   + 1
```

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

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

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

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

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

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

# Explicitly create a polynomial over GF(3) to represent b
```

(continues on next page)

(continued from previous page)

```
In [35]: b = galois.Poly([1, 1], field=GF3); b
Out[35]: Poly(x + 1, GF(3))

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

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

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

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

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

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

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

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

Now the entire multiplication table can be shown for completeness.

```
In [43]: with GF9.display("poly"):
   ....:     print(GF9.arithmetic_table("*"))
   ....:

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

     0     0    |   0    |   0    |   0    |   0    |   0    |   0    |   0    |   0
────────────────────────────────────────────────────────────────────────────────
     1     0    |   1    |   2    |        |  + 1   |  + 2   |   2    | 2 + 1  | 2 + 2
────────────────────────────────────────────────────────────────────────────────
     2     0    |   2    |   1    |   2    | 2 + 2  | 2 + 1  |        |  + 2   |  + 1
────────────────────────────────────────────────────────────────────────────────
           0    |        |   2    |  + 1   | 2 + 1  |   1    | 2 + 2  |   2    |  + 2
────────────────────────────────────────────────────────────────────────────────
   + 1     0    |  + 1   | 2 + 2  | 2 + 1  |   2    |        |  + 2   |   2    |   1
────────────────────────────────────────────────────────────────────────────────
   + 2     0    |  + 2   | 2 + 1  |   1    |        | 2 + 2  |   2    |  + 1   |   2
────────────────────────────────────────────────────────────────────────────────
     2     0    |   2    |        | 2 + 2  |  + 2   |   2    |  + 1   |   1    | 2 + 1
────────────────────────────────────────────────────────────────────────────────
 2 + 1     0    | 2 + 1  |  + 2   |   2    |   2    |  + 1   |   1    | 2 + 2  |
```

(continues on next page)

(continued from previous page)

```
─────────────────────────────────────────────────────────────────────────────────
 2 + 2     0     │ 2 + 2 │  + 1  │  + 2  │   1    │   2    │ 2 + 1 │       │   2
```

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

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

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

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

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

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

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

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

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

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

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

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

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

```
In [52]: with GF9.display("poly"):
   ....:     print(GF9.arithmetic_table("/"))
   ....:

  x / y    1    │   2    │       │  + 1  │  + 2  │   2    │ 2 + 1 │ 2 + 2
```

(continues on next page)

(continued from previous page)

```
  0     0   │   0    │   0    │   0    │   0    │   0    │   0    │   0
─────────────────────────────────────────────────────────────────────
  1     1   │   2    │  + 2  │ 2 + 2 │        │ 2 + 1 │   2    │  + 1
─────────────────────────────────────────────────────────────────────
  2     2   │   1    │ 2 + 1 │  + 1  │   2    │  + 2  │        │ 2 + 2
─────────────────────────────────────────────────────────────────────
            │   2    │   1    │  + 2  │  + 1  │   2    │ 2 + 2 │ 2 + 1
─────────────────────────────────────────────────────────────────────
 + 1   + 1  │ 2 + 2 │        │   1    │ 2 + 1 │   2    │  + 2  │   2
─────────────────────────────────────────────────────────────────────
 + 2   + 2  │ 2 + 1 │ 2 + 2 │   2    │   1    │  + 1  │   2    │
─────────────────────────────────────────────────────────────────────
  2     2   │        │   2    │ 2 + 1 │ 2 + 2 │   1    │  + 1  │  + 2
─────────────────────────────────────────────────────────────────────
2 + 1 2 + 1 │  + 2  │  + 1  │        │   2    │ 2 + 2 │   1    │   2
─────────────────────────────────────────────────────────────────────
2 + 2 2 + 2 │  + 1  │   2    │   2    │  + 2  │        │ 2 + 1 │   1
```

4.2.3 Primitive elements

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

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

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

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

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

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

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

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

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

 Power │ Polynomial │ Vector │ Integer

   0   │     0      │ [0, 0] │    0
────────────────────────────────────
  x^0  │     1      │ [0, 1] │    1
────────────────────────────────────
```

(continues on next page)

(continued from previous page)

```
  x^1     │     x       │ [1, 0] │    3
 ───────────────────────────────────────
  x^2     │   x + 1     │ [1, 1] │    4
 ───────────────────────────────────────
  x^3     │  2x + 1     │ [2, 1] │    7
 ───────────────────────────────────────
  x^4     │     2       │ [0, 2] │    2
 ───────────────────────────────────────
  x^5     │    2x       │ [2, 0] │    6
 ───────────────────────────────────────
  x^6     │  2x + 2     │ [2, 2] │    8
 ───────────────────────────────────────
  x^7     │   x + 2     │ [1, 2] │    5
```

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

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

   Power      │ Polynomial │ Vector │ Integer

     0        │     0      │ [0, 0] │    0
 ─────────────────────────────────────────────
 (2x + 1)^0   │     1      │ [0, 1] │    1
 ─────────────────────────────────────────────
 (2x + 1)^1   │   2x + 1   │ [2, 1] │    7
 ─────────────────────────────────────────────
 (2x + 1)^2   │   2x + 2   │ [2, 2] │    8
 ─────────────────────────────────────────────
 (2x + 1)^3   │     x      │ [1, 0] │    3
 ─────────────────────────────────────────────
 (2x + 1)^4   │     2      │ [0, 2] │    2
 ─────────────────────────────────────────────
 (2x + 1)^5   │   x + 2    │ [1, 2] │    5
 ─────────────────────────────────────────────
 (2x + 1)^6   │   x + 1    │ [1, 1] │    4
 ─────────────────────────────────────────────
 (2x + 1)^7   │    2x      │ [2, 0] │    6
```

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

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

   Power     │ Polynomial │ Vector │ Integer

     0       │     0      │ [0, 0] │    0
 ────────────────────────────────────────────
 (x + 1)^0   │     1      │ [0, 1] │    1
 ────────────────────────────────────────────
 (x + 1)^1   │   x + 1    │ [1, 1] │    4
```

(continues on next page)

(continued from previous page)

```
 ─────────────────────────────────────────────────────
  (x + 1)^2 │      2       │ [0, 2] │     2
 ─────────────────────────────────────────────────────
  (x + 1)^3 │   2x + 2     │ [2, 2] │     8
 ─────────────────────────────────────────────────────
  (x + 1)^4 │      1       │ [0, 1] │     1
 ─────────────────────────────────────────────────────
  (x + 1)^5 │    x + 1     │ [1, 1] │     4
 ─────────────────────────────────────────────────────
  (x + 1)^6 │      2       │ [0, 2] │     2
 ─────────────────────────────────────────────────────
  (x + 1)^7 │   2x + 2     │ [2, 2] │     8
```

4.2.4 Primitive polynomials

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

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

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

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

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

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

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

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

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

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

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

 Power │     Polynomial      │     Vector     │ Integer

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

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

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

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

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

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

(continues on next page)

(continued from previous page)

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

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

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

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

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

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

   Power     |     Polynomial      |    Vector    | Integer

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

(continues on next page)

(continued from previous page)

```
(x + 1)^11 │    x^3 + x + 1     │ [1, 0, 1, 1] │     11
──────────────────────────────────────────────────────
(x + 1)^12 │         x          │ [0, 0, 1, 0] │     2
──────────────────────────────────────────────────────
(x + 1)^13 │      x^2 + x       │ [0, 1, 1, 0] │     6
──────────────────────────────────────────────────────
(x + 1)^14 │      x^3 + x       │ [1, 0, 1, 0] │     10
```

4.3 Constructing Galois field array classes

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

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

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

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

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

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

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

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

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

$\mathrm{GF}(2^m)$ 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.FieldArray)
Out[10]: True

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

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

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

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

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

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

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

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

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

4.4 Array creation

4.4.1 Explicit construction

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

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

# Represents an existing numpy array
In [2]: x_np = np.random.randint(0, 7, 10, dtype=int); x_np
Out[2]: array([3, 0, 2, 2, 0, 1, 3, 0, 6, 5])

# Create a Galois field array through explicit construction (x_np is copied)
In [3]: x = GF7(x_np); x
Out[3]: GF([3, 0, 2, 2, 0, 1, 3, 0, 6, 5], order=7)
```

4.4.2 View casting

```
# View cast an existing array to a Galois field array (no copy operation)
In [4]: y = x_np.view(GF7); y
Out[4]: GF([3, 0, 2, 2, 0, 1, 3, 0, 6, 5], 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 [5]: x_np
Out[5]: array([3, 0, 2, 2, 0, 1, 3, 0, 6, 5])

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

# Notice x is unchanged due to the copy during the explicit construction
In [7]: x
Out[7]: GF([3, 0, 2, 2, 0, 1, 3, 0, 6, 5], order=7)

# Notice y is changed due to view casting
In [8]: y
Out[8]: GF([4, 0, 2, 2, 0, 1, 3, 0, 6, 5], order=7)
```

4.4.3 Alternate constructors

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

```
In [9]: GF256.Random((2,5))
Out[9]:
GF([[ 18, 119,   2, 204, 223],
    [255, 149, 183, 234, 227]], order=2^8)

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

In [11]: GF256.Elements()
Out[11]:
GF([  0,   1,   2,   3,   4,   5,   6,   7,   8,   9,  10,  11,  12,  13,
     14,  15,  16,  17,  18,  19,  20,  21,  22,  23,  24,  25,  26,  27,
     28,  29,  30,  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)
```

4.4.4 Array dtypes

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

```
In [12]: GF = galois.GF(7)

In [13]: a = GF.Random(10); a
Out[13]: GF([1, 2, 4, 5, 0, 6, 5, 2, 3, 5], order=7)

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

# Type cast an existing Galois field array to a different dtype
In [15]: a = a.astype(np.int16); a
Out[15]: GF([1, 2, 4, 5, 0, 6, 5, 2, 3, 5], order=7)
```

(continues on next page)

(continued from previous page)

```
In [16]: a.dtype
Out[16]: 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 [17]: b = GF.Random(10, dtype=np.int16); b
Out[17]: GF([0, 1, 6, 4, 2, 2, 3, 3, 0, 6], order=7)

In [18]: b.dtype
Out[18]: dtype('int16')
```

4.4.5 Field element display modes

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

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

```
In [19]: GF = galois.GF(2**3)

In [20]: a = GF.Random(10)

# The default mode represents the field elements as integers
In [21]: a
Out[21]: GF([4, 1, 3, 1, 0, 2, 6, 3, 7, 5], order=2^3)

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

# The display mode can be set to "power" mode
In [23]: GF.display("power"); a
Out[23]: GF([^2, 1, ^3, 1, 0, , ^4, ^3, ^5, ^6], order=2^3)

# Reset the display mode to the default
In [24]: GF.display(); a
Out[24]: GF([4, 1, 3, 1, 0, 2, 6, 3, 7, 5], order=2^3)
```

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

```
# The original display mode
In [25]: print(a)
GF([4, 1, 3, 1, 0, 2, 6, 3, 7, 5], order=2^3)

# The new display context
In [26]: with GF.display("poly"):
```

(continues on next page)

(continued from previous page)

```
   ....:     print(a)
   ....:
GF([^2, 1,  + 1, 1, 0, , ^2 + ,  + 1, ^2 +  + 1, ^2 + 1],
   order=2^3)

In [27]: with GF.display("power"):
   ....:     print(a)
   ....:
GF([^2, 1, ^3, 1, 0, , ^4, ^3, ^5, ^6], order=2^3)

# Returns to the original display mode
In [28]: print(a)
GF([4, 1, 3, 1, 0, 2, 6, 3, 7, 5], order=2^3)
```

4.5 Galois field array arithmetic

4.5.1 Addition, subtraction, multiplication, division

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

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

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

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

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

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

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

# Multiplication in the finite field
In [7]: x * y
Out[7]: GF([3, 3, 0, 0, 5, 6, 0, 2, 5, 5], order=7)

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

(continues on next page)

(continued from previous page)

```
In [9]: x // y
Out[9]: GF([3, 6, 0, 0, 6, 5, 0, 2, 3, 5], order=7)
```

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

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

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

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

4.5.2 Scalar multiplication

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

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

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

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

In [17]: a = GF8.Random(10); a
Out[17]: GF([0, 6, 1, 7, 0, 2, 0, 6, 6, 2], order=2^3)

# Calculates a x "2" in the finite field
In [18]: a * GF8(2)
Out[18]: GF([0, 7, 2, 5, 0, 4, 0, 7, 7, 4], order=2^3)

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

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

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

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

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

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

4.5.3 Exponentiation

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

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

In [26]: x = GF7.Random(10); x
Out[26]: GF([4, 5, 2, 4, 5, 2, 2, 4, 3, 0], order=7)

# Calculates "x" * "x", note 2 is not a field element
```

(continues on next page)

(continued from previous page)

```
In [27]: x ** 2
Out[27]: GF([2, 4, 4, 2, 4, 4, 4, 2, 2, 0], order=7)
```

4.5.4 Logarithm

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

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

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

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

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

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

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

4.6 Extremely large fields

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

4.6.1 Large GF(p) fields

```
In [1]: prime = 36893488147419103183

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

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

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

In [5]: a = GF.Random(10); a
```

(continues on next page)

(continued from previous page)

```
Out[5]:
GF([33537829449962885674, 16042529009753609664, 10866226205519732295,
    14092577331152141395, 16927998116002922971, 36148846129010931589,
    2103285211293963847, 4017471091364150894, 28782985382329694155,
    2147107529303104376], order=36893488147419103183)

In [6]: b = GF.Random(10); b
Out[6]:
GF([33599818908932537718, 429288030605114487, 9442671844213946301,
    20134906419511163876, 14450089583391726852, 11286927895869630241,
    12277145855857962698, 22106858698299515242, 30669871520546998723,
    4092672160866226338], order=36893488147419103183)

In [7]: a + b
Out[7]:
GF([30244160211476320209, 16471817040358724151, 20308898049733678596,
    34227483750663305271, 31378087699394649823, 10542285877461458647,
    14380431067151926545, 26124329789663666136, 22559368755457589695,
    6239779690169330714], order=36893488147419103183)
```

4.6.2 Large GF(2^m) fields

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

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

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

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

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

# Display elements as polynomials
In [13]: GF.display("poly")
Out[13]: <galois._fields._class.DisplayContext at 0x7f5c76d56080>

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

In [15]: b
```

(continues on next page)

(continued from previous page)

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

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

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

# Reset the display mode
In [18]: GF.display()
Out[18]: <galois._fields._class.DisplayContext at 0x7f5c76d56908>
```

CHAPTER

FIVE

PERFORMANCE TESTING

5.1 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.

5.1.1 Lookup performance

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

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

In [2]: import galois

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

5.1.2 Calculation performance

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

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

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

In [2]: import galois

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

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

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

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

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

(continues on next page)

(continued from previous page)

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

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

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

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

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

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

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

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

5.1.3 Linear algebra performance

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

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

In [2]: import galois

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

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

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

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

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

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

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

Below is a comparison of galois computing the correct matrix multiplication over GF(2^8) and NumPy computing a normal integer matrix multiplication (which is not the correct result!). This comparison is just for a performance reference.

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

In [2]: import galois

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

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

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

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

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

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

5.2 Benchmarking

The *galois* package comes with benchmarking tests. They are contained in the benchmarks/ folder. They are pytest tests using the pytest-benchmark extension.

5.2.1 Running a benchmark test

To run a benchmark, invoke pytest on the benchmarks/ folder or a specific test set (e.g., benchmarks/test_field_arithmetic.py). It is also advised to pass extra arguments to format the display --benchmark-columns=min,max,mean,stddev,median and --benchmark-sort=name.

```
$ pytest benchmarks/test_field_arithmetic.py --benchmark-columns=min,max,mean,stddev,
↪median --benchmark-sort=name
===========================================================================================␣
↪test session starts␣
↪==========================================================================================
platform linux -- Python 3.8.10, pytest-6.0.1, py-1.9.0, pluggy-0.13.1
benchmark: 3.4.1 (defaults: timer=time.perf_counter disable_gc=False min_rounds=5 min_
↪time=0.000005 max_time=1.0 calibration_precision=10 warmup=False warmup_
↪iterations=100000)
rootdir: /home/matt/repos/galois, configfile: setup.cfg
plugins: benchmark-3.4.1, extra-durations-0.1.3, anyio-2.1.0
collected 56 items

benchmarks/test_field_arithmetic.py ...................................................
↪...                                                                                      ␣
↪             [100%]
```

(continues on next page)

(continued from previous page)

```
============================================================================== sum␣
↪of all tests durations␣
↪===============================================================================
25.80s

----------------------- benchmark "GF(2) Array Arithmetic: shape=(100_000,), ufunc_mode=
↪'jit-calculate'": 8 tests -------------------------
Name (time in us)                      Min                   Max                   Mean  ␣
↪             StdDev               Median
-----------------------------------------------------------------------------------------
↪--------------------------------------------------
test_add                            54.9031 (1.04)       238.1941 (1.19)        62.2950␣
↪(1.06)      14.3986 (1.30)       58.4391 (1.03)
test_additive_inverse               52.8959 (1.0)        237.0442 (1.18)        58.6939␣
↪(1.0)       11.1021 (1.0)        56.5751 (1.0)
test_divide                        207.9280 (3.93)       636.3150 (3.17)       234.0875␣
↪(3.99)      38.3417 (3.45)      220.6110 (3.90)
test_multiplicative_inverse        188.9290 (3.57)       652.8331 (3.25)       229.7088␣
↪(3.91)      57.9788 (5.22)      201.3633 (3.56)
test_multiply                       54.6200 (1.03)       228.7410 (1.14)        61.9206␣
↪(1.05)      12.5660 (1.13)       58.8121 (1.04)
test_power                         229.3210 (4.34)       284.5279 (1.42)       246.6029␣
↪(4.20)      22.5143 (2.03)      238.7560 (4.22)
test_scalar_multiply             1,561.2941 (29.52)    3,148.2361 (15.67)    2,058.2764␣
↪(35.07)    361.3528 (32.55)   1,969.9985 (34.82)
test_subtract                       54.5362 (1.03)       200.8791 (1.0)         62.6200␣
↪(1.07)      13.5863 (1.22)       59.0470 (1.04)
-----------------------------------------------------------------------------------------
↪--------------------------------------------------

---------------------- benchmark "GF(257) Array Arithmetic: shape=(100_000,), ufunc_mode=
↪'jit-calculate'": 8 tests -----------------------
Name (time in us)                      Min                   Max                   Mean  ␣
↪             StdDev               Median
-----------------------------------------------------------------------------------------
↪--------------------------------------------------
test_add                           161.1579 (1.22)       583.2699 (1.20)       197.0225␣
↪(1.18)      52.4775 (1.15)      175.6141 (1.21)
test_additive_inverse              132.4308 (1.0)        546.5120 (1.12)       166.7050␣
↪(1.0)       46.9263 (1.03)      144.6060 (1.0)
test_divide                      6,306.7721 (47.62)    7,277.9991 (14.94)    6,658.3270␣
↪(39.94)    374.7916 (8.21)    6,597.4400 (45.62)
test_multiplicative_inverse      6,299.4179 (47.57)    6,477.2971 (13.29)    6,392.0670␣
↪(38.34)     75.0566 (1.64)    6,367.5269 (44.03)
test_multiply                      337.8210 (2.55)       487.2240 (1.0)        362.2038␣
↪(2.17)      45.6575 (1.0)       344.7749 (2.38)
test_power                       5,026.9060 (37.96)    5,523.7100 (11.34)    5,273.6145␣
↪(31.63)    204.7190 (4.48)    5,208.3279 (36.02)
test_scalar_multiply             1,033.5341 (7.80)     2,472.3031 (5.07)     1,223.1746␣
↪(7.34)     205.6682 (4.50)    1,157.2575 (8.00)
test_subtract                      172.0500 (1.30)       655.0739 (1.34)       211.9266␣
↪(1.27)      59.9908 (1.31)      188.1695 (1.30)
```

(continues on next page)

(continued from previous page)

```
--------------------------------------------------------------------------------------------
↪---------------------------------------------------

------------------------ benchmark "GF(257) Array Arithmetic: shape=(100_000,), ufunc_
↪mode='jit-lookup'": 8 tests ------------------------
Name (time in us)                         Min                    Max                   Mean ␣
↪             StdDev                 Median
--------------------------------------------------------------------------------------------
↪---------------------------------------------------
test_add                             175.6609 (1.23)       217.3430 (1.22)       182.2442␣
↪(1.20)       12.6704 (1.09)       177.3606 (1.23)
test_additive_inverse                142.2670 (1.0)        178.2598 (1.0)        152.1027␣
↪(1.0)        13.3374 (1.15)       144.3694 (1.0)
test_divide                          414.2381 (2.91)       543.3671 (3.05)       473.2308␣
↪(3.11)       50.1936 (4.32)       475.6054 (3.29)
test_multiplicative_inverse          336.2088 (2.36)       459.0931 (2.58)       363.2939␣
↪(2.39)       40.4252 (3.48)       342.7650 (2.37)
test_multiply                        319.1240 (2.24)       414.7580 (2.33)       342.5294␣
↪(2.25)       33.0754 (2.85)       322.0430 (2.23)
test_power                           446.4111 (3.14)       505.2469 (2.83)       461.2746␣
↪(3.03)       24.9575 (2.15)       449.4530 (3.11)
test_scalar_multiply               1,037.2710 (7.29)     2,068.3082 (11.60)    1,197.2376␣
↪(7.87)      185.2372 (15.94)    1,137.1800 (7.88)
test_subtract                        176.0179 (1.24)       212.1262 (1.19)       181.7260␣
↪(1.19)       11.6174 (1.0)        177.3450 (1.23)
--------------------------------------------------------------------------------------------
↪---------------------------------------------------

------------------------- benchmark "GF(2^8) Array Arithmetic: shape=(100_000,), ufunc_
↪mode='jit-calculate'": 8 tests --------------------------
Name (time in us)                          Min                        Max                  ␣
↪Mean                StdDev                 Median
--------------------------------------------------------------------------------------------
↪---------------------------------------------------------
test_add                                 62.4850 (1.03)         232.4961 (1.0)          73.
↪3393 (1.0)          19.2285 (1.0)          67.0920 (1.01)
test_additive_inverse                    60.8349 (1.0)          303.9362 (1.31)         75.
↪7256 (1.03)         23.9701 (1.25)         66.5609 (1.0)
test_divide                          31,357.7619 (515.46)    33,928.1799 (145.93)   32,024.
↪4656 (436.66)    1,071.7328 (55.74)    31,594.5051 (474.67)
test_multiplicative_inverse          16,130.9310 (265.16)    22,589.3550 (97.16)    18,422.
↪6860 (251.20)    2,467.6099 (128.33)   17,968.8521 (269.96)
test_multiply                         1,726.0939 (28.37)      2,139.3239 (9.20)      1,832.
↪1673 (24.98)       172.9170 (8.99)      1,769.2349 (26.58)
test_power                           15,418.5530 (253.45)    19,151.2981 (82.37)    17,459.
↪3566 (238.06)    1,456.6692 (75.76)    17,972.3101 (270.01)
test_scalar_multiply                  1,424.5091 (23.42)      3,249.2320 (13.98)     1,838.
↪0753 (25.06)       336.6038 (17.51)     1,740.8130 (26.15)
test_subtract                            66.2010 (1.09)         324.3710 (1.40)         85.
↪7436 (1.17)         29.3386 (1.53)         74.9400 (1.13)
--------------------------------------------------------------------------------------------
↪---------------------------------------------------------
```

(continues on next page)

(continued from previous page)

```
------------------------ benchmark "GF(2^8) Array Arithmetic: shape=(100_000,), ufunc_
↪mode='jit-lookup'": 8 tests ------------------------
Name (time in us)                         Min                    Max                   Mean ␣
↪             StdDev                Median
------------------------------------------------------------------------------------------
↪---------------------------------------------------
test_add                           66.6410 (1.09)       307.5062 (1.25)         92.5952␣
↪(1.25)       31.7356 (1.53)        78.6111 (1.19)
test_additive_inverse              61.1460 (1.0)        288.4439 (1.17)         73.9261␣
↪(1.0)        21.2221 (1.03)        66.1870 (1.0)
test_divide                       381.8041 (6.24)       642.9779 (2.61)        486.1582␣
↪(6.58)      102.6931 (4.96)       477.1240 (7.21)
test_multiplicative_inverse       308.8908 (5.05)       450.9010 (1.83)        343.5373␣
↪(4.65)       58.9238 (2.85)       309.4219 (4.67)
test_multiply                     310.6489 (5.08)       510.5541 (2.07)        392.5511␣
↪(5.31)       74.9355 (3.62)       372.2322 (5.62)
test_power                        454.8710 (7.44)       614.1132 (2.49)        502.7098␣
↪(6.80)       63.6973 (3.08)       480.4349 (7.26)
test_scalar_multiply            1,337.2621 (21.87)    3,347.7580 (13.58)    1,646.4154␣
↪(22.27)     351.6510 (17.00)    1,511.9230 (22.84)
test_subtract                      63.4668 (1.04)       246.5090 (1.0)          76.8375␣
↪(1.04)       20.6870 (1.0)         69.5318 (1.05)
------------------------------------------------------------------------------------------
↪---------------------------------------------------

-------------------------- benchmark "GF(3^5) Array Arithmetic: shape=(1_000,), ufunc_
↪mode='jit-calculate'": 8 tests ---------------------------
Name (time in us)                          Min                     Max                  ␣
↪Mean                    StdDev                  Median
------------------------------------------------------------------------------------------
↪---------------------------------------------------------
test_add                            600.3019 (1.34)        875.1580 (1.42)         669.
↪5663 (1.40)         80.0338 (2.23)          650.1831 (1.43)
test_additive_inverse               447.7361 (1.0)         618.4638 (1.0)          478.
↪0377 (1.0)          51.4350 (1.43)          453.8295 (1.0)
test_divide                      20,522.7621 (45.84)    26,730.4152 (43.22)    23,591.
↪0309 (49.35)     2,306.3176 (64.17)     23,143.9110 (51.00)
test_multiplicative_inverse      18,970.5859 (42.37)    29,393.6098 (47.53)    22,934.
↪1439 (47.98)     4,165.3422 (115.89)    23,078.4880 (50.85)
test_multiply                     1,429.8242 (3.19)      1,609.7031 (2.60)       1,518.
↪5410 (3.18)         83.9082 (2.33)        1,542.0800 (3.40)
test_power                       17,494.5770 (39.07)    20,862.5218 (33.73)    19,221.
↪0962 (40.21)     1,306.0528 (36.34)     19,433.4229 (42.82)
test_scalar_multiply              1,319.3421 (2.95)      3,319.4488 (5.37)       1,828.
↪1196 (3.82)        359.4200 (10.00)       1,704.8109 (3.76)
test_subtract                       594.8299 (1.33)        726.0030 (1.17)         614.
↪7950 (1.29)         35.9430 (1.0)           602.5580 (1.33)
------------------------------------------------------------------------------------------
↪---------------------------------------------------------

-------------------- benchmark "GF(3^5) Array Arithmetic: shape=(1_000,), ufunc_mode=
↪'jit-lookup'": 8 tests --------------------
```

(continues on next page)

(continued from previous page)

```
Name (time in us)                   Min                 Max                Mean              ␣
↪    StdDev              Median
-----------------------------------------------------------------------------------------
↪---------------------------------------
test_add                        48.6970 (1.09)      75.5680 (1.07)      57.5135 (1.18)      ␣
↪   10.6115 (1.33)      56.1608 (1.23)
test_additive_inverse           45.3601 (1.02)      72.2490 (1.03)      52.5008 (1.07)      ␣
↪   11.3766 (1.42)      47.2669 (1.04)
test_divide                     46.1771 (1.04)      70.9470 (1.01)      51.1555 (1.05)      ␣
↪    9.0236 (1.13)      47.0870 (1.03)
test_multiplicative_inverse     49.7589 (1.12)      78.2839 (1.11)      54.3773 (1.11)      ␣
↪    9.2918 (1.16)      50.3049 (1.10)
test_multiply                   44.4769 (1.0)       70.4550 (1.0)       48.9275 (1.0)       ␣
↪    7.9862 (1.0)       45.5696 (1.0)
test_power                      74.0550 (1.67)     148.8561 (2.11)     102.2906 (2.09)      ␣
↪   26.8338 (3.36)     100.3731 (2.20)
test_scalar_multiply            68.6261 (1.54)     294.9270 (4.19)      82.6070 (1.69)      ␣
↪   23.0046 (2.88)      74.2599 (1.63)
test_subtract                   53.4819 (1.20)     106.8390 (1.52)      79.2563 (1.62)      ␣
↪   24.2412 (3.04)      83.1240 (1.82)
-----------------------------------------------------------------------------------------
↪---------------------------------------

Legend:
Outliers: 1 Standard Deviation from Mean; 1.5 IQR (InterQuartile Range) from 1st␣
↪Quartile and 3rd Quartile.
OPS: Operations Per Second, computed as 1 / Mean
=======================================================================================␣
↪56 passed in 26.60s␣
↪=========================================================================================
```

5.2.2 Comparing with previous benchmarks

If you'd like to compare the performance impacts of a branch, for instance, check out `master` and run `pytest` with the `--benchmark-save` option. This will save a file in `.benchmarks/0001_master.json`.

```
$ git checkout master
$ pytest benchmarks/test_field_arithmetic.py --benchmark-columns=min,max,mean,stddev,
↪median --benchmark-sort=name --benchmark-save=master
```

```
$ git checkout branch
$ pytest benchmarks/test_field_arithmetic.py --benchmark-columns=min,max,mean,stddev,
↪median --benchmark-sort=name --benchmark-compare=0001_master
Comparing against benchmarks from: Linux-CPython-3.8-64bit/0001_master.on
=======================================================================================␣
↪test session starts␣
↪=========================================================================================
platform linux -- Python 3.8.10, pytest-6.0.1, py-1.9.0, pluggy-0.13.1
benchmark: 3.4.1 (defaults: timer=time.perf_counter disable_gc=False min_rounds=5 min_
↪time=0.000005 max_time=1.0 calibration_precision=10 warmup=False warmup_
↪iterations=100000)
```

(continues on next page)

(continued from previous page)

```
rootdir: /home/matt/repos/galois, configfile: setup.cfg
plugins: benchmark-3.4.1, extra-durations-0.1.3, anyio-2.1.0
collected 56 items

benchmarks/test_field_arithmetic.py .....................................................
↪...                                                                                  ␣
↪              [100%]

==================================================================================== sum␣
↪of all tests durations␣
↪=====================================================================================
23.72s

----------------------------- benchmark "GF(2) Array Arithmetic: shape=(100_000,), ufunc_
↪mode='jit-calculate'": 16 tests -----------------------------
Name (time in us)                                  Min                    Max               ␣
↪     Mean              StdDev               Median
------------------------------------------------------------------------------------------
↪-------------------------------------------------------------
test_add (0001_master)                            53.6300 (1.04)       570.4910 (3.85)       ␣
↪  74.1800 (1.31)      29.7015 (4.18)        66.1639 (1.23)
test_add (NOW)                                    53.8658 (1.05)       148.1280 (1.0)        ␣
↪  56.6440 (1.0)        7.1116 (1.0)         55.1331 (1.03)
test_additive_inverse (0001_master)               51.5240 (1.0)        200.4388 (1.35)       ␣
↪  62.2888 (1.10)      15.7254 (2.21)        55.8060 (1.04)
test_additive_inverse (NOW)                       51.7170 (1.00)       176.7289 (1.19)       ␣
↪  57.3724 (1.01)      11.7825 (1.66)        53.6300 (1.0)
test_divide (0001_master)                        206.0279 (4.00)       823.5171 (5.56)       ␣
↪ 265.5769 (4.69)      68.8627 (9.68)       239.2294 (4.46)
test_divide (NOW)                                205.6009 (3.99)       698.4309 (4.72)       ␣
↪ 225.6514 (3.98)      35.2674 (4.96)       212.8950 (3.97)
test_multiplicative_inverse (0001_master)        186.9558 (3.63)       566.1491 (3.82)       ␣
↪ 220.2184 (3.89)      49.3615 (6.94)       198.8420 (3.71)
test_multiplicative_inverse (NOW)                187.0971 (3.63)       464.8890 (3.14)       ␣
↪ 203.0407 (3.58)      27.2545 (3.83)       193.4364 (3.61)
test_multiply (0001_master)                       56.0421 (1.09)       248.2592 (1.68)       ␣
↪  81.5853 (1.44)      31.2515 (4.39)        68.5630 (1.28)
test_multiply (NOW)                               53.5559 (1.04)       283.5400 (1.91)       ␣
↪  59.3049 (1.05)      13.7564 (1.93)        54.8789 (1.02)
test_power (0001_master)                         234.1510 (4.54)       330.9071 (2.23)       ␣
↪ 272.5068 (4.81)      40.0108 (5.63)       274.3008 (5.11)
test_power (NOW)                                 230.1259 (4.47)       405.7041 (2.74)       ␣
↪ 277.7140 (4.90)      74.4280 (10.47)      236.8400 (4.42)
test_scalar_multiply (0001_master)             1,345.2091 (26.11)    2,872.6910 (19.39)     ␣
↪1,627.9802 (28.74)    253.5061 (35.65)    1,575.4123 (29.38)
test_scalar_multiply (NOW)                     1,277.7401 (24.80)    2,155.3310 (14.55)     ␣
↪1,377.6186 (24.32)    142.8746 (20.09)    1,313.8375 (24.50)
test_subtract (0001_master)                       60.0331 (1.17)       300.8121 (2.03)       ␣
↪  83.1536 (1.47)      28.1426 (3.96)        69.4490 (1.29)
test_subtract (NOW)                               53.4630 (1.04)       185.5700 (1.25)       ␣
↪  59.3924 (1.05)      12.0366 (1.69)        54.9241 (1.02)
------------------------------------------------------------------------------------------
↪-------------------------------------------------------------
```

(continues on next page)

(continued from previous page)

```
---------------------------- benchmark "GF(257) Array Arithmetic: shape=(100_000,),␣
↪ufunc_mode='jit-calculate'": 16 tests ----------------------------
Name (time in us)                                   Min                    Max                ␣
↪       Mean              StdDev                 Median
------------------------------------------------------------------------------------------
↪--------------------------------------------------------------
test_add (0001_master)                             175.3040 (1.37)       635.8509 (1.89)     ␣
↪ 197.5639 (1.37)       40.8794 (3.55)       186.2240 (1.37)
test_add (NOW)                                     158.3470 (1.23)       636.1029 (1.89)     ␣
↪ 176.8827 (1.22)       32.4656 (2.82)       167.7470 (1.23)
test_additive_inverse (0001_master)                139.4011 (1.09)       504.7920 (1.50)     ␣
↪ 170.0383 (1.18)       38.9894 (3.38)       153.7615 (1.13)
test_additive_inverse (NOW)                        128.2750 (1.0)        481.9639 (1.43)     ␣
↪ 144.6905 (1.0)        27.1122 (2.35)       136.4225 (1.0)
test_divide (0001_master)                        6,233.1830 (48.59)    6,527.6541 (19.42)    ␣
↪6,394.2735 (44.19)     119.4256 (10.36)     6,390.6419 (46.84)
test_divide (NOW)                                5,915.1719 (46.11)    6,276.5831 (18.68)    ␣
↪6,073.0814 (41.97)     174.3328 (15.13)     6,044.0000 (44.30)
test_multiplicative_inverse (0001_master)        6,131.1359 (47.80)    6,474.0989 (19.26)    ␣
↪6,265.0689 (43.30)     157.5507 (13.67)     6,174.8459 (45.26)
test_multiplicative_inverse (NOW)                5,846.0971 (45.57)    7,729.8910 (23.00)    ␣
↪6,541.4660 (45.21)     748.3626 (64.94)     6,168.4800 (45.22)
test_multiply (0001_master)                        294.6239 (2.30)       336.0880 (1.0)      ␣
↪ 300.5223 (2.08)       11.5243 (1.0)        296.6555 (2.17)
test_multiply (NOW)                                313.1870 (2.44)       502.4392 (1.49)     ␣
↪ 340.8181 (2.36)       54.2113 (4.70)       315.9265 (2.32)
test_power (0001_master)                         5,487.4821 (42.78)    5,796.6399 (17.25)    ␣
↪5,624.7412 (38.87)     126.0851 (10.94)     5,565.2100 (40.79)
test_power (NOW)                                 4,765.4652 (37.15)    5,108.4792 (15.20)    ␣
↪4,907.8213 (33.92)     126.2470 (10.95)     4,901.8471 (35.93)
test_scalar_multiply (0001_master)               1,030.8721 (8.04)     2,567.3539 (7.64)     ␣
↪1,164.7107 (8.05)      198.8079 (17.25)     1,100.3308 (8.07)
test_scalar_multiply (NOW)                       1,032.1550 (8.05)     2,005.4830 (5.97)     ␣
↪1,144.9685 (7.91)      132.6795 (11.51)     1,092.9224 (8.01)
test_subtract (0001_master)                        160.0061 (1.25)       377.1251 (1.12)     ␣
↪ 177.7399 (1.23)       31.2034 (2.71)       165.8800 (1.22)
test_subtract (NOW)                                164.6511 (1.28)       705.7940 (2.10)     ␣
↪ 177.2582 (1.23)       31.8202 (2.76)       170.3722 (1.25)
------------------------------------------------------------------------------------------
↪--------------------------------------------------------------

---------------------------- benchmark "GF(257) Array Arithmetic: shape=(100_000,),␣
↪ufunc_mode='jit-lookup'": 16 tests ------------------------------
Name (time in us)                                   Min                    Max                ␣
↪       Mean              StdDev                 Median
------------------------------------------------------------------------------------------
↪--------------------------------------------------------------
test_add (0001_master)                             177.8340 (1.28)       227.8131 (1.34)     ␣
↪ 184.8953 (1.29)       15.3610 (1.59)       179.1745 (1.28)
test_add (NOW)                                     188.0738 (1.36)       321.0511 (1.88)     ␣
↪ 226.4540 (1.58)       50.5466 (5.24)       195.8210 (1.40)
```

(continues on next page)

(continued from previous page)

```
test_additive_inverse (0001_master)          139.9061 (1.01)      170.4299 (1.0)      ␣
↪ 149.0880 (1.04)      10.9319 (1.13)      143.6260 (1.03)
test_additive_inverse (NOW)                  138.5838 (1.0)       171.5040 (1.01)     ␣
↪ 143.1082 (1.0)        9.6383 (1.0)       139.7599 (1.0)
test_divide (0001_master)                    384.4770 (2.77)      472.9840 (2.78)     ␣
↪ 407.4521 (2.85)      31.3674 (3.25)      393.6000 (2.82)
test_divide (NOW)                            368.6091 (2.66)      469.5239 (2.75)     ␣
↪ 397.4139 (2.78)      33.2011 (3.44)      392.4301 (2.81)
test_multiplicative_inverse (0001_master)    335.2840 (2.42)      371.9509 (2.18)     ␣
↪ 345.5276 (2.41)      13.9113 (1.44)      337.5154 (2.41)
test_multiplicative_inverse (NOW)            338.9800 (2.45)      375.3309 (2.20)     ␣
↪ 344.6998 (2.41)      11.1648 (1.16)      339.7759 (2.43)
test_multiply (0001_master)                  314.1060 (2.27)      373.7470 (2.19)     ␣
↪ 322.5519 (2.25)      18.2331 (1.89)      315.9530 (2.26)
test_multiply (NOW)                          283.0350 (2.04)      342.2331 (2.01)     ␣
↪ 294.4223 (2.06)      18.5191 (1.92)      284.2465 (2.03)
test_power (0001_master)                     461.0980 (3.33)      516.3020 (3.03)     ␣
↪ 492.7848 (3.44)      28.5527 (2.96)      509.1929 (3.64)
test_power (NOW)                             446.6961 (3.22)      513.4281 (3.01)     ␣
↪ 471.9436 (3.30)      24.9776 (2.59)      468.0229 (3.35)
test_scalar_multiply (0001_master)         1,034.2200 (7.46)    2,384.6650 (13.99)   ␣
↪1,175.8929 (8.22)     188.4351 (19.55)    1,120.0898 (8.01)
test_scalar_multiply (NOW)                 1,047.1060 (7.56)    2,285.6970 (13.41)   ␣
↪1,148.9137 (8.03)     135.4746 (14.06)    1,106.0899 (7.91)
test_subtract (0001_master)                  175.6351 (1.27)      217.9069 (1.28)     ␣
↪ 186.5705 (1.30)      16.5222 (1.71)      178.6321 (1.28)
test_subtract (NOW)                          177.5022 (1.28)      222.6690 (1.31)     ␣
↪ 188.9046 (1.32)      17.0226 (1.77)      179.9939 (1.29)
-----------------------------------------------------------------------------------------
↪-------------------------------------------------------------

------------------------------- benchmark "GF(2^8) Array Arithmetic: shape=(100_000,),␣
↪ufunc_mode='jit-calculate'": 16 tests -------------------------------
Name (time in us)                                   Min                    Max             ␣
↪          Mean                   StdDev                  Median
-----------------------------------------------------------------------------------------
↪-------------------------------------------------------------------
test_add (0001_master)                             65.4031 (1.10)       303.5041 (1.36)     ␣
↪       85.0358 (1.27)       26.2409 (2.35)        76.7412 (1.23)
test_add (NOW)                                     63.1949 (1.06)       376.9179 (1.69)     ␣
↪       73.6567 (1.10)       18.6649 (1.67)        67.6350 (1.08)
test_additive_inverse (0001_master)                59.5278 (1.0)        223.1461 (1.0)      ␣
↪       68.2673 (1.02)       17.9042 (1.60)        62.5600 (1.0)
test_additive_inverse (NOW)                        60.6440 (1.02)       261.6630 (1.17)     ␣
↪       67.1257 (1.0)        11.1651 (1.0)         64.8301 (1.04)
test_divide (0001_master)                      20,575.8919 (345.65)   22,262.2380 (99.77)   ␣
↪ 21,375.1204 (318.43)     682.3102 (61.11)    21,379.7691 (341.75)
test_divide (NOW)                              20,864.8040 (350.51)   21,292.5780 (95.42)   ␣
↪ 21,095.4332 (314.27)     182.5212 (16.35)    21,152.4100 (338.11)
test_multiplicative_inverse (0001_master)      14,893.7618 (250.20)   18,845.2560 (84.45)   ␣
↪ 16,354.3480 (243.64)   1,507.6655 (135.03)   16,245.4580 (259.68)
test_multiplicative_inverse (NOW)              15,077.0759 (253.28)   15,525.0910 (69.57)   ␣
↪ 15,313.5644 (228.13)     166.0180 (14.87)    15,339.7880 (245.20)
```

(continues on next page)

(continued from previous page)

```
test_multiply (0001_master)                       1,781.0490 (29.92)     1,911.8502 (8.57)    ␣
↪  1,850.5634 (27.57)       48.3193 (4.33)       1,857.6779 (29.69)
test_multiply (NOW)                               1,514.2660 (25.44)     1,614.7611 (7.24)    ␣
↪  1,565.8663 (23.33)       36.6022 (3.28)       1,559.7660 (24.93)
test_power (0001_master)                         17,853.8628 (299.92)   20,269.1101 (90.83)   ␣
↪ 19,207.0828 (286.14)     893.4667 (80.02)     19,430.7449 (310.59)
test_power (NOW)                                 14,477.4460 (243.20)   16,110.0640 (72.20)   ␣
↪ 14,925.3466 (222.35)     685.6721 (61.41)     14,588.3670 (233.19)
test_scalar_multiply (0001_master)                1,414.1060 (23.76)     3,601.1150 (16.14)   ␣
↪  1,645.1560 (24.51)      297.7188 (26.67)      1,549.7539 (24.77)
test_scalar_multiply (NOW)                        1,418.7060 (23.83)     3,286.4269 (14.73)   ␣
↪  1,593.0689 (23.73)      246.1296 (22.04)      1,503.2839 (24.03)
test_subtract (0001_master)                          61.8761 (1.04)        226.9482 (1.02)    ␣
↪     78.5680 (1.17)        20.9293 (1.87)          72.5901 (1.16)
test_subtract (NOW)                                  62.4610 (1.05)        247.6051 (1.11)    ␣
↪     72.8939 (1.09)        18.7591 (1.68)          66.7050 (1.07)
------------------------------------------------------------------------------------------
↪-------------------------------------------------------------------


----------------------------- benchmark "GF(2^8) Array Arithmetic: shape=(100_000,),␣
↪ufunc_mode='jit-lookup'": 16 tests -----------------------------
Name (time in us)                                      Min                    Max        ␣
↪     Mean                 StdDev                 Median
------------------------------------------------------------------------------------------
↪-------------------------------------------------------------
test_add (0001_master)                              62.1150 (1.04)      242.7939 (1.33)      ␣
↪  73.7056 (1.08)       19.0953 (1.73)        67.1479 (1.05)
test_add (NOW)                                      62.2938 (1.04)      183.1381 (1.0)       ␣
↪  68.0219 (1.0)        11.0402 (1.0)         65.1181 (1.02)
test_additive_inverse (0001_master)                 63.4750 (1.06)      306.6689 (1.67)      ␣
↪  83.7360 (1.23)       27.5077 (2.49)        74.2881 (1.16)
test_additive_inverse (NOW)                         59.8810 (1.0)       240.5860 (1.31)      ␣
↪  68.7484 (1.01)       17.0596 (1.55)        63.7915 (1.0)
test_divide (0001_master)                          376.1090 (6.28)      427.3648 (2.33)      ␣
↪ 404.0765 (5.94)       20.4260 (1.85)       410.8525 (6.44)
test_divide (NOW)                                  368.1141 (6.15)      473.6630 (2.59)      ␣
↪ 423.8058 (6.23)       39.0911 (3.54)       431.1745 (6.76)
test_multiplicative_inverse (0001_master)          370.6738 (6.19)      658.4080 (3.60)      ␣
↪ 465.4766 (6.84)      108.3985 (9.82)       448.2989 (7.03)
test_multiplicative_inverse (NOW)                  309.5521 (5.17)      342.8990 (1.87)      ␣
↪ 317.9003 (4.67)       12.0781 (1.09)       311.4310 (4.88)
test_multiply (0001_master)                        344.9500 (5.76)      527.0049 (2.88)      ␣
↪ 376.0530 (5.53)       67.0493 (6.07)       348.1212 (5.46)
test_multiply (NOW)                                293.8330 (4.91)      365.5059 (2.00)      ␣
↪ 313.4176 (4.61)       25.9667 (2.35)       303.4060 (4.76)
test_power (0001_master)                           512.7750 (8.56)      614.1500 (3.35)      ␣
↪ 546.0020 (8.03)       42.9520 (3.89)       523.0750 (8.20)
test_power (NOW)                                   480.9520 (8.03)      541.6819 (2.96)      ␣
↪ 496.3040 (7.30)       25.6325 (2.32)       484.7571 (7.60)
test_scalar_multiply (0001_master)               1,343.6000 (22.44)   3,204.0619 (17.50)    ␣
↪1,808.8479 (26.59)     342.6193 (31.03)     1,742.1505 (27.31)
test_scalar_multiply (NOW)                       1,273.2670 (21.26)   2,594.8270 (14.17)    ␣
↪1,382.8612 (20.33)     174.7877 (15.83)     1,314.1120 (20.60)
```

(continues on next page)

(continued from previous page)

```
test_subtract (0001_master)                     62.6270 (1.05)      264.0169 (1.44)       ␣
↪  77.2011 (1.13)      20.9450 (1.90)       69.2611 (1.09)
test_subtract (NOW)                             62.4580 (1.04)      225.9151 (1.23)       ␣
↪  69.4670 (1.02)      12.8329 (1.16)       65.2981 (1.02)
------------------------------------------------------------------------------------------
↪-------------------------------------------------------------

-------------------------------- benchmark "GF(3^5) Array Arithmetic: shape=(1_000,),␣
↪ufunc_mode='jit-calculate'": 16 tests --------------------------------
Name (time in us)                                    Min                    Max           ␣
↪         Mean                    StdDev                  Median
------------------------------------------------------------------------------------------
↪-------------------------------------------------------------------
test_add (0001_master)                          577.8901 (1.29)       669.9329 (1.15)     ␣
↪     596.5873 (1.22)       29.3569 (1.20)        581.5226 (1.28)
test_add (NOW)                                  565.7980 (1.27)       654.5840 (1.12)     ␣
↪     580.1125 (1.19)       26.6488 (1.09)        569.4189 (1.25)
test_additive_inverse (0001_master)             528.4690 (1.18)     1,180.2840 (2.02)     ␣
↪     679.2012 (1.39)      198.8098 (8.13)        596.6179 (1.31)
test_additive_inverse (NOW)                     447.2181 (1.0)        583.9660 (1.0)      ␣
↪     488.9684 (1.0)        50.0549 (2.05)        454.5329 (1.0)
test_divide (0001_master)                    23,370.1861 (52.26)   27,859.3549 (47.71)    ␣
↪ 25,382.3458 (51.91)    1,936.6428 (79.20)    25,000.1471 (55.00)
test_divide (NOW)                            26,724.1660 (59.76)   29,631.6862 (50.74)    ␣
↪ 28,053.3344 (57.37)    1,366.7746 (55.90)    27,384.5338 (60.25)
test_multiplicative_inverse (0001_master)    19,703.0620 (44.06)   24,010.1409 (41.12)    ␣
↪ 21,643.3670 (44.26)    1,981.0911 (81.02)    21,131.4450 (46.49)
test_multiplicative_inverse (NOW)            18,754.1139 (41.94)   24,443.9621 (41.86)    ␣
↪ 21,822.6110 (44.63)    2,057.5147 (84.15)    21,855.1469 (48.08)
test_multiply (0001_master)                   1,425.0348 (3.19)     1,521.4889 (2.61)     ␣
↪  1,475.4896 (3.02)       45.4442 (1.86)      1,496.0892 (3.29)
test_multiply (NOW)                           1,683.6938 (3.76)     2,381.5830 (4.08)     ␣
↪  2,060.1566 (4.21)      281.4423 (11.51)     2,146.3181 (4.72)
test_power (0001_master)                     23,206.8000 (51.89)   27,687.5449 (47.41)    ␣
↪ 25,244.8478 (51.63)    2,196.7235 (89.84)    24,164.1460 (53.16)
test_power (NOW)                             17,747.6569 (39.68)   22,459.7279 (38.46)    ␣
↪ 19,107.0794 (39.08)    1,926.5370 (78.79)    18,604.5689 (40.93)
test_scalar_multiply (0001_master)            1,225.6859 (2.74)     4,148.2730 (7.10)     ␣
↪  1,776.8746 (3.63)      573.3293 (23.45)     1,577.6126 (3.47)
test_scalar_multiply (NOW)                    1,220.4021 (2.73)     2,658.5320 (4.55)     ␣
↪  1,471.6397 (3.01)      248.3855 (10.16)     1,377.4155 (3.03)
test_subtract (0001_master)                     584.9910 (1.31)       672.4251 (1.15)     ␣
↪     595.7514 (1.22)       24.4517 (1.0)         587.2779 (1.29)
test_subtract (NOW)                             571.2900 (1.28)       636.8279 (1.09)     ␣
↪     591.2687 (1.21)       26.5326 (1.09)        575.3681 (1.27)
------------------------------------------------------------------------------------------
↪-------------------------------------------------------------------

------------------------ benchmark "GF(3^5) Array Arithmetic: shape=(1_000,), ufunc_mode=
↪'jit-lookup'": 16 tests -------------------------
Name (time in us)                              Min                  Max                 ␣
↪Mean                StdDev              Median
```

(continues on next page)

(continued from previous page)

```
---------------------------------------------------------------------------------------------
↪------------------------------------------------
test_add (0001_master)                     49.7401 (1.17)      76.4418 (1.15)     57.
↪4466 (1.24)     10.9692 (1.50)     53.2519 (1.22)
test_add (NOW)                             47.1489 (1.11)      72.2532 (1.09)     54.
↪1565 (1.16)     10.4557 (1.43)     49.3161 (1.13)
test_additive_inverse (0001_master)        43.5489 (1.03)     108.0472 (1.62)     66.
↪0670 (1.42)     29.6528 (4.04)     46.5869 (1.06)
test_additive_inverse (NOW)                42.7819 (1.01)      69.3691 (1.04)     48.
↪8280 (1.05)     10.3019 (1.40)     44.3211 (1.01)
test_divide (0001_master)                  45.0709 (1.06)      86.1811 (1.30)     51.
↪6534 (1.11)     14.1129 (1.92)     45.8754 (1.05)
test_divide (NOW)                          44.6450 (1.05)      68.3721 (1.03)     48.
↪9857 (1.05)      8.0531 (1.10)     45.6730 (1.04)
test_multiplicative_inverse (0001_master)  42.4671 (1.0)       66.5220 (1.0)      46.
↪4964 (1.0)       7.3332 (1.0)      43.8021 (1.0)
test_multiplicative_inverse (NOW)          44.0120 (1.04)      68.5172 (1.03)     48.
↪4346 (1.04)      7.5601 (1.03)     44.8425 (1.02)
test_multiply (0001_master)                44.4860 (1.05)      67.9540 (1.02)     48.
↪5520 (1.04)      7.6291 (1.04)     45.4651 (1.04)
test_multiply (NOW)                        43.8211 (1.03)      67.5640 (1.02)     48.
↪4292 (1.04)      7.3999 (1.01)     44.8920 (1.02)
test_power (0001_master)                   65.6650 (1.55)      97.5579 (1.47)     73.
↪2096 (1.57)     12.2555 (1.67)     68.3501 (1.56)
test_power (NOW)                           65.0741 (1.53)      95.6031 (1.44)     71.
↪8517 (1.55)     11.9027 (1.62)     66.6430 (1.52)
test_scalar_multiply (0001_master)         68.4580 (1.61)     300.1180 (4.51)     78.
↪9913 (1.70)     19.5450 (2.67)     73.0332 (1.67)
test_scalar_multiply (NOW)                 69.1570 (1.63)     256.4839 (3.86)     77.
↪5875 (1.67)     13.4846 (1.84)     74.0800 (1.69)
test_subtract (0001_master)                49.5000 (1.17)      76.2481 (1.15)     61.
↪0748 (1.31)     11.9175 (1.63)     56.3171 (1.29)
test_subtract (NOW)                        47.5000 (1.12)      88.6670 (1.33)     65.
↪7632 (1.41)     16.4101 (2.24)     64.0899 (1.46)
---------------------------------------------------------------------------------------------
↪------------------------------------------------

Legend:
Outliers: 1 Standard Deviation from Mean; 1.5 IQR (InterQuartile Range) from 1st␣
↪Quartile and 3rd Quartile.
OPS: Operations Per Second, computed as 1 / Mean
=========================================================================================␣
↪56 passed in 24.48s␣
↪==========================================================================================
```

Or if you have two saved benchmarks, you can simply compare them without re-running the tests.

```
$ pytest-benchmark compare 0001_master 0002_master
```

CHAPTER

SIX

DEVELOPMENT

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

6.1 Lint the package

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

Listing 1: 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/
```

6.2 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 2: requirements-test.txt

```
pytest
pytest-cov
pytest-benchmark
```

Install the test dependencies.

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

Run the unit tests.

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

6.3 Build the documentation

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

Listing 3: requirements-doc.txt

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

SEVEN

API REFERENCE V0.0.21

7.1 Galois Fields

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

7.1.1 Galois field class creation

Class factory functions

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

galois.GF

galois.**GF**(*order*, *irreducible_poly=None*, *primitive_element=None*, *verify=True*, *compile=None*, *display=None*)

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

Parameters

- **order** (*int*) – The order p^m of the field $\mathrm{GF}(p^m)$. The order must be a prime power.
- **irreducible_poly** (*int*, *str*, *tuple*, *list*, *numpy.ndarray*, galois.Poly, *optional*) – Optionally specify an irreducible polynomial of degree m over $\mathrm{GF}(p)$ that will define the Galois field arithmetic.
  - None (default): Uses the Conway polynomial $C_{p,m}$, see *galois.conway_poly()*.
  - int: The integer representation of the irreducible polynomial.
  - str: The irreducible polynomial expressed as a string, e.g. "x^2 + 1".
  - tuple, list, numpy.ndarray: The irreducible polynomial coefficients in degree-descending order.
  - *galois.Poly*: The irreducible polynomial as a polynomial object.
- **primitive_element** (*int*, *str*, *tuple*, *list*, *numpy.ndarray*, galois.Poly, *optional*) – Optionally specify a primitive element of the field $\mathrm{GF}(p^m)$. This value is used when building the log/anti-log lookup tables and when computing *np.log()*. 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 $\mathrm{GF}(p)$.

For prime fields:

- None (default): Uses the minimal primitive root modulo p, see *galois.primitive_root()*.
- int: A primitive root modulo p.

For extension fields:

- None (default): Uses the lexicographically-minimal primitive element, see *galois.primitive_element()*.
- int: The integer representation of the primitive element.
- str: The primitive element expressed as a string, e.g. "x + 1".
- tuple, list, numpy.ndarray: The primitive element's polynomial coefficients in degree-descending order.
- *galois.Poly*: The primitive element as a polynomial object.

- **verify** (*bool*, *optional*) – Indicates whether to verify that the specified irreducible polynomial is in fact irreducible and whether the specified primitive element is in fact a generator of the multiplicative group. The default is True. For large fields and irreducible polynomials that are already known to be irreducible (which may take a long time to verify), this argument can be set to False. If the default irreducible polynomial and primitive element are used, no verification is performed because the defaults are guaranteed to be irreducible and a multiplicative generator, respectively.
- **compile** (*str*, *optional*) – The ufunc calculation mode. This can be modified after class construction with the *galois.FieldClass.compile()* method.
  - None (default): For newly-created classes, None corresponds to "auto". For Galois field array classes of this type that were previously created, None does not modify the current ufunc compilation mode.
  - "auto": Selects "jit-lookup" for fields with order less than 2^{20}, "jit-calculate" for larger fields, and "python-calculate" for fields whose elements cannot be represented with numpy.int64.
  - "jit-lookup": JIT compiles arithmetic ufuncs to use Zech log, log, and anti-log lookup tables for efficient computation. In the few cases where explicit calculation is faster than table lookup, explicit calculation is used.
  - "jit-calculate": JIT compiles arithmetic ufuncs to use explicit calculation. The "jit-calculate" mode is designed for large fields that cannot or should not store lookup tables in RAM. Generally, the "jit-calculate" mode is slower than "jit-lookup".
  - "python-calculate": Uses pure-python ufuncs with explicit calculation. This is reserved for fields whose elements cannot be represented with numpy.int64 and instead use numpy.object_ with python int (which has arbitrary precision).
- **display** (*str*, *optional*) – The field element display representation. This can be modified after class consstruction with the *galois.FieldClass.display()* method.
  - None (default): For newly-created classes, None corresponds to the integer representation ("int"). For Galois field array classes of this type that were previously created, None does not modify the current display mode.
  - "int": The element displayed as the integer representation of the polynomial. For example, $2x^2+x+2$ is an element of $\mathrm{GF}(3^3)$ and is equivalent to the integer $23 = 2 \cdot 3^2 + 3 + 2$.

- "poly": The element as a polynomial over $\mathrm{GF}(p)$ of degree less than m. For example, $2x^2 + x + 2$ is an element of $\mathrm{GF}(3^3)$.
- "power": The element as a power of the primitive element, see `galois.FieldClass.primitive_element`. For example, $2x^2 + x + 2 = \alpha^5$ in $\mathrm{GF}(3^3)$ with irreducible polynomial $x^3 + 2x + 1$ and primitive element $\alpha = x$.

Returns A Galois field array class for $\mathrm{GF}(p^m)$. If this class has already been created, a reference to that class is returned.

Return type *galois.FieldClass*

Notes

The created class is a subclass of `galois.FieldArray` and an instance of `galois.FieldClass`. The `galois.FieldArray` inheritance provides the `numpy.ndarray` functionality and some additional methods on Galois field arrays, such as `galois.FieldArray.row_reduce()`. The `galois.FieldClass` metaclass provides a variety of class attributes and methods relating to the finite field, such as the `galois.FieldClass.display()` method to change the field element display representation.

Galois field array classes of the same type (order, irreducible polynomial, and primitive element) are singletons. So, calling this class factory with arguments that correspond to the same class will return the same class object.

Examples

Construct various Galois field array class for $\mathrm{GF}(2)$, $\mathrm{GF}(2^m)$, $\mathrm{GF}(p)$, and $\mathrm{GF}(p^m)$ with the default irreducible polynomials and primitive elements. For the extension fields, notice the irreducible polynomials are primitive and x is a primitive element.

```
# Construct a GF(2) class
In [1]: GF2 = galois.GF(2); print(GF2.properties)
GF(2):
  characteristic: 2
  degree: 1
  order: 2
  irreducible_poly: x + 1
  is_primitive_poly: True
  primitive_element: 1

# Construct a GF(2^m) class
In [2]: GF256 = galois.GF(2**8); print(GF256.properties)
GF(2^8):
  characteristic: 2
  degree: 8
  order: 256
  irreducible_poly: x^8 + x^4 + x^3 + x^2 + 1
  is_primitive_poly: True
  primitive_element: x

# Construct a GF(p) class
In [3]: GF3 = galois.GF(3); print(GF3.properties)
GF(3):
  characteristic: 3
  degree: 1
```

(continues on next page)

(continued from previous page)

```
  order: 3
  irreducible_poly: x + 1
  is_primitive_poly: True
  primitive_element: 2

# Construct a GF(p^m) class
In [4]: GF243 = galois.GF(3**5); print(GF243.properties)
GF(3^5):
  characteristic: 3
  degree: 5
  order: 243
  irreducible_poly: x^5 + 2x + 1
  is_primitive_poly: True
  primitive_element: x
```

Or construct a Galois field array class and specify the irreducible polynomial. Here is an example using the $\mathrm{GF}(2^8)$ field from AES. Notice the irreducible polynomial is not primitive and x is not a primitive element.

```
In [5]: poly = galois.Poly.Degrees([8,4,3,1,0]); poly
Out[5]: Poly(x^8 + x^4 + x^3 + x + 1, GF(2))

In [6]: GF256_AES = galois.GF(2**8, irreducible_poly=poly)

In [7]: print(GF256_AES.properties)
GF(2^8):
  characteristic: 2
  degree: 8
  order: 256
  irreducible_poly: x^8 + x^4 + x^3 + x + 1
  is_primitive_poly: False
  primitive_element: x + 1
```

Very large fields are also supported but they use `numpy.object_` dtypes with python `int` and, therefore, do not have compiled ufuncs.

```
# Construct a very large GF(2^m) class
In [8]: GF2m = galois.GF(2**100); print(GF2m.properties)
GF(2^100):
  characteristic: 2
  degree: 100
  order: 1267650600228229401496703205376
  irreducible_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
  is_primitive_poly: True
  primitive_element: x

In [9]: GF2m.dtypes, GF2m.ufunc_mode
Out[9]: ([numpy.object_], 'python-calculate')

# Construct a very large GF(p) class
In [10]: 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: x + 36893488147419103180
  is_primitive_poly: True
  primitive_element: 3

In [11]: GFp.dtypes, GFp.ufunc_mode
Out[11]: ([numpy.object_], 'python-calculate')
```

The default display mode for field elements is the integer representation. This can be modified by using the `display` keyword argument. It can also be changed after class construction by calling the *galois.FieldClass.display()* method.

```
In [12]: GF = galois.GF(2**8)

In [13]: GF.Random()
Out[13]: GF(158, order=2^8)

In [14]: GF = galois.GF(2**8, display="poly")

In [15]: GF.Random()
Out[15]: GF(^6 + ^5 + ^4 +  + 1, order=2^8)
```

Galois field array classes of the same type (order, irreducible polynomial, and primitive element) are singletons. So, calling this class factory with arguments that correspond to the same class will return the same field class object.

```
In [16]: poly1 = galois.Poly([1, 0, 0, 0, 1, 1, 0, 1, 1])

In [17]: poly2 = poly1.integer

In [18]: galois.GF(2**8, irreducible_poly=poly1) is galois.GF(2**8, irreducible_
↪poly=poly2)
Out[18]: True
```

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

galois.Field

galois.**Field**(*order*, *irreducible_poly=None*, *primitive_element=None*, *verify=True*, *compile=None*, *display=None*)

Alias of `galois.GF()`.

Abstract base classes

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

galois.FieldArray

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

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

Important: `galois.FieldArray` is an abstract base class for all Galois field array classes and cannot be instantiated directly. Instead, `galois.FieldArray` subclasses are created using the class factory `galois.GF()`.

This class is included in the API to allow the user to test if an array is a Galois field array subclass.

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

In [2]: issubclass(GF, galois.FieldArray)
Out[2]: True

In [3]: x = GF([1,2,3]); x
Out[3]: GF([1, 2, 3], order=7)

In [4]: isinstance(x, galois.FieldArray)
Out[4]: True
```

Notes

`galois.FieldArray` is an abstract base class and cannot be instantiated directly. Instead, the user creates a `galois.FieldArray` subclass for the field $\mathrm{GF}(p^m)$ by calling the class factory `galois.GF()`, e.g. `GF = galois.GF(p**m)`. In this case, `GF` is a subclass of `galois.FieldArray` and an instance of `galois.FieldClass`, a metaclass that defines special methods and attributes related to the Galois field.

`galois.FieldArray`, and `GF`, is a subclass of `numpy.ndarray` and its constructor `x = GF(array_like)` has the same syntax as `numpy.array()`. The returned `galois.FieldArray` instance `x` is a `numpy.ndarray` that is acted upon like any other numpy array, except all arithmetic is performed in $\mathrm{GF}(p^m)$ not in $\mathbb{Z}$ or $\mathbb{R}$.

Examples

Construct the Galois field class for $GF(2^8)$ using the class factory *galois.GF()* and then display some relevant properties of the field. See *galois.FieldClass* for a complete list of Galois field array class methods and attributes.

```
In [5]: GF256 = galois.GF(2**8)

In [6]: GF256
Out[6]: <class 'numpy.ndarray over GF(2^8)'>

In [7]: print(GF256.properties)
GF(2^8):
  characteristic: 2
  degree: 8
  order: 256
  irreducible_poly: x^8 + x^4 + x^3 + x^2 + 1
  is_primitive_poly: True
  primitive_element: x
```

Depending on the field's order, only certain numpy dtypes are supported. See *galois.FieldClass.dtypes* for more details.

```
In [8]: GF256.dtypes
Out[8]:
[numpy.uint8,
 numpy.uint16,
 numpy.uint32,
 numpy.int16,
 numpy.int32,
 numpy.int64]
```

Galois field arrays can be created from existing numpy arrays.

```
In [9]: x = np.array([155, 232, 162, 159,  63,  29, 247, 141,  75, 189], dtype=int)

# Explicit Galois field array creation -- a copy is performed
In [10]: GF256(x)
Out[10]: GF([155, 232, 162, 159,  63,  29, 247, 141,  75, 189], order=2^8)

# Or view an existing numpy array as a Galois field array -- no copy is performed
In [11]: x.view(GF256)
Out[11]: GF([155, 232, 162, 159,  63,  29, 247, 141,  75, 189], order=2^8)
```

Galois field arrays can also be created explicitly by converting an "array-like" object.

```
# A scalar GF(2^8) element from its integer representation
In [12]: GF256(37)
Out[12]: GF(37, order=2^8)

# A scalar GF(2^8) element from its polynomial representation
In [13]: GF256("x^5 + x^2 + 1")
Out[13]: GF(37, order=2^8)
```

(continues on next page)

(continued from previous page)

```
# A GF(2^8) array from a list of elements in their integer representation
In [14]: GF256([[142, 27], [92, 253]])
Out[14]:
GF([[142,  27],
    [ 92, 253]], order=2^8)

# A GF(2^8) array from a list of elements in their integer and polynomial␣
↪representations
In [15]: GF256([[142, "x^5 + x^2 + 1"], [92, 253]])
Out[15]:
GF([[142,  37],
    [ 92, 253]], order=2^8)
```

There's also an alternate constructor *Vector()* (and accompanying *vector()* method) to convert an array of coefficients over $\mathrm{GF}(p)$ with last dimension m into Galois field elements in $\mathrm{GF}(p^m)$.

```
# A scalar GF(2^8) element from its vector representation
In [16]: GF256.Vector([0, 0, 1, 0, 0, 1, 0, 1])
Out[16]: GF(37, order=2^8)

# A GF(2^8) array from a list of elements in their vector representation
In [17]: GF256.Vector([[[1, 0, 0, 0, 1, 1, 1, 0], [0, 0, 0, 1, 1, 0, 1, 1]], [[0, 1,
↪ 0, 1, 1, 1, 0, 0], [1, 1, 1, 1, 1, 1, 0, 1]]])
Out[17]:
GF([[142,  27],
    [ 92, 253]], order=2^8)
```

Newly-created arrays will use the smallest unsigned dtype, unless otherwise specified.

```
In [18]: a = GF256([66, 166, 27, 182, 125]); a
Out[18]: GF([ 66, 166,  27, 182, 125], order=2^8)

In [19]: a.dtype
Out[19]: dtype('uint8')

In [20]: b = GF256([66, 166, 27, 182, 125], dtype=np.int64); b
Out[20]: GF([ 66, 166,  27, 182, 125], order=2^8)

In [21]: b.dtype
Out[21]: dtype('int64')
```

Constructors

__init__(array[, dtype, copy, order, ndmin])	Creates an array over $\mathrm{GF}(p^m)$.
Elements([dtype])	Creates a 1-D Galois field array of the field's elements $\{0, \dots, p^m - 1\}$.
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.

continues on next page

Table 3 – continued from previous page

Range(start, stop[, step, dtype])	Creates a 1-D Galois field array with a range of field elements.
Vandermonde(a, m, n[, dtype])	Creates an $m \times n$ 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 $\mathrm{GF}(p)$.

Special Methods

__add__(other)	Adds two Galois field arrays element-wise.
__divmod__(other)	Divides two Galois field arrays element-wise and returns the quotient and remainder.
__floordiv__(other)	Divides two Galois field arrays element-wise.
__mod__(other)	Divides two Galois field arrays element-wise and returns the remainder.
__mul__(other)	Multiplies two Galois field arrays element-wise.
__pow__(other)	Exponentiates a Galois field array element-wise.
__sub__(other)	Subtracts two Galois field arrays element-wise.
__truediv__(other)	Divides two Galois field arrays element-wise.

classmethod Elements(*dtype=None*)

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

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

Returns A 1-D Galois field array of all the field's elements.

Return type *galois.FieldArray*

Examples

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

In [2]: GF.Elements()
Out[2]:
GF([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15],
   order=2^4)
```

As usual, Galois field elements can be displayed in either the "integer" (default), "polynomial", or "power" representation. This can be changed by calling *galois.FieldClass.display()*.

```
# Permanently set the display mode to "poly"
In [3]: GF.display("poly");

In [4]: GF.Elements()
Out[4]:
GF([0, 1, , + 1, ^2, ^2 + 1, ^2 + , ^2 + + 1, ^3, ^3 + 1,
    ^3 + , ^3 + + 1, ^3 + ^2, ^3 + ^2 + 1, ^3 + ^2 + ,
    ^3 + ^2 + + 1], order=2^4)

# Temporarily set the display mode to "power"
In [5]: with GF.display("power"):
   ...:     print(GF.Elements())
   ...:
GF([0, 1, , ^4, ^2, ^8, ^5, ^10, ^3, ^14, ^9, ^7, ^6, ^13,
    ^11, ^12], order=2^4)

# Reset the display mode to "int"
In [6]: GF.display();
```

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 unsigned dtype for this class, i.e. the first element in *galois.FieldClass.dtypes*.

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

Return type *galois.FieldArray*

Examples

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

In [2]: GF.Identity(4)
Out[2]:
GF([[1, 0, 0, 0],
    [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** (*int*, *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-D array. A 2-tuple, e.g. `(M,N)`, represents a 2-D 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 unsigned dtype for this class, i.e. the first element in *galois.FieldClass.dtypes*.

Returns A Galois field array of ones.

Return type *galois.FieldArray*

Examples

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

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

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

Creates a Galois field array with random field elements.

Parameters

- **shape** (*int*, *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-D array. A 2-tuple, e.g. `(M,N)`, represents a 2-D array with each element indicating the size in each dimension.
- **low** (*int*, *optional*) – The lowest value (inclusive) of a random field element in its integer representation. The default is 0.
- **high** (*int*, *optional*) – The highest value (exclusive) of a random field element in its integer representation. 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 unsigned dtype for this class, i.e. the first element in *galois.FieldClass.dtypes*.

Returns A Galois field array of random field elements.

Return type *galois.FieldArray*

Examples

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

In [2]: GF.Random((2,5))
Out[2]:
GF([[ 5,  2, 11, 15,  1],
    [11, 11, 12,  1,  5]], order=31)
```

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

Creates a 1-D Galois field array with a range of field elements.

Parameters

- **start** (*int*) – The starting Galois field value (inclusive) in its integer representation.
- **stop** (*int*) – The stopping Galois field value (exclusive) in its integer representation.
- **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 unsigned dtype for this class, i.e. the first element in `galois.FieldClass.dtypes`.

Returns A 1-D Galois field array of a range of field elements.

Return type *galois.FieldArray*

Examples

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

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

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

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

Parameters

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

Returns The $m \times n$ Vandermonde matrix.

Return type *galois.FieldArray*

Examples

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

In [2]: a = GF.primitive_element

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 $\mathrm{GF}(p)$.

This function is the inverse operation of the `vector()` method.

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)`. By convention, the vectors are ordered from highest degree to 0-th degree.
- **dtype** (*numpy.dtype, optional*) – The `numpy.dtype` of the array elements. The default is `None` which represents the smallest unsigned dtype for this class, i.e. the first element in `galois.FieldClass.dtypes`.

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

Return type *galois.FieldArray*

Examples

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

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

In [3]: a = GF.Vector(vec); a
Out[3]: GF([13, 36, 36], order=2^6)

In [4]: with GF.display("poly"):
   ...:     print(a)
   ...:
```

(continues on next page)

(continued from previous page)

```
GF([^3 + ^2 + 1, ^5 + ^2, ^5 + ^2], order=2^6)

In [5]: a.vector()
Out[5]:
GF([[0, 0, 1, 1, 0, 1],
    [1, 0, 0, 1, 0, 0],
    [1, 0, 0, 1, 0, 0]], order=2)
```

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

Creates a Galois field array with all zeros.

Parameters

- **shape** (*int, 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-D array. A 2-tuple, e.g. `(M,N)`, represents a 2-D 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 unsigned dtype for this class, i.e. the first element in `galois.FieldClass.dtypes`.

Returns A Galois field array of zeros.

Return type *galois.FieldArray*

Examples

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

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

__add__(*other*)

Adds two Galois field arrays element-wise.

Broadcasting rules apply. Both arrays must be over the same Galois field.

Parameters **other** (`galois.FieldArray`) – The other Galois field array.

Returns The Galois field array `self + other`.

Return type *galois.FieldArray*

Examples

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

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

In [3]: b = GF.Random(5); b
Out[3]: GF([3, 4, 3, 2, 6], order=7)

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

__divmod__(*other*)

Divides two Galois field arrays element-wise and returns the quotient and remainder.

Broadcasting rules apply. Both arrays must be over the same Galois field. In Galois fields, true division and floor division are equivalent. In Galois fields, the remainder is always zero.

Parameters **other** (`galois.FieldArray`) – The other Galois field array.

Returns

- *galois.FieldArray* – The Galois field array `self // other`.
- *galois.FieldArray* – The Galois field array `self % other`.

Examples

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

In [2]: a = GF.Random((2,5)); a
Out[2]:
GF([[5, 6, 1, 4, 0],
    [2, 0, 1, 0, 4]], order=7)

In [3]: b = GF.Random(5, low=1); b
Out[3]: GF([1, 3, 3, 5, 3], order=7)

In [4]: q, r = divmod(a, b)

In [5]: q, r
Out[5]:
(GF([[5, 2, 5, 5, 0],
     [2, 0, 5, 0, 6]], order=7),
 GF([[0, 0, 0, 0, 0],
     [0, 0, 0, 0, 0]], order=7))

In [6]: b*q + r
Out[6]:
```

(continues on next page)

(continued from previous page)

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

__floordiv__(*other*)

Divides two Galois field arrays element-wise.

Broadcasting rules apply. Both arrays must be over the same Galois field. In Galois fields, true division and floor division are equivalent.

Parameters **other** (`galois.FieldArray`) – The other Galois field array.

Returns The Galois field array `self // other`.

Return type *galois.FieldArray*

Examples

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

In [2]: a = GF.Random((2,5)); a
Out[2]:
GF([[3, 5, 2, 1, 1],
    [3, 1, 6, 6, 5]], order=7)

In [3]: b = GF.Random(5, low=1); b
Out[3]: GF([2, 1, 6, 3, 3], order=7)

In [4]: a // b
Out[4]:
GF([[5, 5, 5, 5, 5],
    [5, 1, 1, 2, 4]], order=7)
```

__init__(*array*, *dtype=None*, *copy=True*, *order='K'*, *ndmin=0*)

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

Parameters

- **array** (*int*, *str*, *tuple*, *list*, *numpy.ndarray*, `galois.FieldArray`) – The input array-like object to be converted to a Galois field array. See the examples section for demonstations of array creation using each input type. See see *galois.FieldClass.display()* and *galois.FieldClass.display_mode* for a description of the "integer" and "polynomial" representation of Galois field elements.
  - `int`: A single integer, which is the "integer representation" of a Galois field element, creates a 0-D array.
  - `str`: A single string, which is the "polynomial representation" of a Galois field element, creates a 0-D array.
  - `tuple`, `list`: A list or tuple (or nested lists/tuples) of ints or strings (which can be mix-and-matched) creates an array of

  Galois field elements from their integer or polynomial representations. * `numpy.ndarray`, *galois.FieldArray*: An array of ints creates a copy of the array over this specific field.

- **dtype** (*numpy.dtype, optional*) – The numpy.dtype of the array elements. The default is None which represents the smallest unsigned dtype for this class, i.e. the first element in *galois.FieldClass.dtypes*.
- **copy** (*bool, optional*) – The copy keyword argument from numpy.array(). The default is True which makes a copy of the input array.
- **order** (*str, 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 An array over $\mathrm{GF}(p^m)$.

Return type *galois.FieldArray*

__mod__(*other*)

Divides two Galois field arrays element-wise and returns the remainder.

Broadcasting rules apply. Both arrays must be over the same Galois field. In Galois fields, true division and floor division are equivalent. In Galois fields, the remainder is always zero.

Parameters **other** (galois.FieldArray) – The other Galois field array.

Returns The Galois field array self % other.

Return type *galois.FieldArray*

Examples

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

In [2]: a = GF.Random((2,5)); a
Out[2]:
GF([[4, 0, 0, 6, 4],
    [3, 6, 2, 2, 0]], order=7)

In [3]: b = GF.Random(5, low=1); b
Out[3]: GF([1, 5, 6, 4, 3], order=7)

In [4]: a % b
Out[4]:
GF([[0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0]], order=7)
```

__mul__(*other*)

Multiplies two Galois field arrays element-wise.

Broadcasting rules apply. Both arrays must be over the same Galois field.

> **Warning:** When both multiplicands are *galois.FieldArray*, that indicates a Galois field multiplication. When one multiplicand is an integer or integer numpy.ndarray, that indicates a scalar multiplication (repeated addition). Galois field multiplication and scalar multiplication are equivalent in prime fields, but not in extension fields.

Parameters **other** (*numpy.ndarray*, galois.FieldArray) – A numpy.ndarray of integers for scalar multiplication or a *galois.FieldArray* of Galois field elements for finite field multiplication.

Returns The Galois field array `self * other`.

Return type *galois.FieldArray*

Examples

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

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

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

In [4]: a * b
Out[4]:
GF([[0, 0, 0, 2, 4],
    [0, 0, 1, 5, 1]], order=7)
```

When both multiplicands are Galois field elements, that indicates a Galois field multiplication.

```
In [5]: GF = galois.GF(2**4, display="poly")

In [6]: a = GF(7); a
Out[6]: GF(^2 +  + 1, order=2^4)

In [7]: b = GF(2); b
Out[7]: GF(, order=2^4)

In [8]: a * b
Out[8]: GF(^3 + ^2 + , order=2^4)
```

When one multiplicand is an integer, that indicates a scalar multiplication (repeated addition).

```
In [9]: a * 2
Out[9]: GF(0, order=2^4)

In [10]: a + a
Out[10]: GF(0, order=2^4)
```

__pow__(*other*)

Exponentiates a Galois field array element-wise.

Broadcasting rules apply. The first array must be a Galois field array and the second must be an integer or integer array.

Parameters **other** (*int*, *numpy.ndarray*) – The exponent(s) as an integer or integer array.

Returns The Galois field array `self ** other`.

Return type *galois.FieldArray*

Examples

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

In [2]: a = GF.Random((2,5)); a
Out[2]:
GF([[2, 4, 6, 1, 5],
    [1, 6, 1, 5, 2]], order=7)

In [3]: b = np.random.randint(0, 10, 5); b
Out[3]: array([1, 0, 8, 8, 9])

In [4]: a ** b
Out[4]:
GF([[2, 1, 1, 1, 6],
    [1, 1, 1, 4, 1]], order=7)
```

`__sub__`(*other*)

Subtracts two Galois field arrays element-wise.

Broadcasting rules apply. Both arrays must be over the same Galois field.

Parameters **`other`** (`galois.FieldArray`) – The other Galois field array.

Returns The Galois field array `self - other`.

Return type *galois.FieldArray*

Examples

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

In [2]: a = GF.Random((2,5)); a
Out[2]:
GF([[2, 4, 5, 6, 0],
    [0, 1, 1, 4, 6]], order=7)

In [3]: b = GF.Random(5); b
Out[3]: GF([1, 5, 3, 0, 6], order=7)

In [4]: a - b
Out[4]:
GF([[1, 6, 2, 6, 1],
    [6, 3, 5, 4, 0]], order=7)
```

`__truediv__`(*other*)

Divides two Galois field arrays element-wise.

Broadcasting rules apply. Both arrays must be over the same Galois field. In Galois fields, true division and floor division are equivalent.

Parameters **`other`** (`galois.FieldArray`) – The other Galois field array.

Returns The Galois field array `self / other`.

Return type *galois.FieldArray*

Examples

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

In [2]: a = GF.Random((2,5)); a
Out[2]:
GF([[3, 0, 4, 4, 4],
    [6, 4, 5, 5, 6]], order=7)

In [3]: b = GF.Random(5, low=1); b
Out[3]: GF([4, 5, 3, 5, 2], order=7)

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

`lu_decompose()`

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

Returns

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

Examples

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

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

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

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

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

(continues on next page)

(continued from previous page)

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

lup_decompose()

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

Returns

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

Examples

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

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

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

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

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

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

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

row_reduce(*ncols=None*)

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

Row reduction operations

1. Swap the position of any two rows.

2. Multiply a row by a non-zero scalar.
3. Add one row to a scalar multiple of another row.

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

Returns The reduced row echelon form of the input array.

Return type *galois.FieldArray*

Examples

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

In [2]: A = GF.Random((4,4)); A
Out[2]:
GF([[ 4, 28,  7, 21],
    [12, 26, 11, 21],
    [12, 26, 17, 18],
    [30, 27, 25, 17]], order=31)

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

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

One column is a linear combination of another.

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

In [6]: A = GF.Random((4,4)); A
Out[6]:
GF([[13, 20, 16,  0],
    [ 3, 21, 18, 19],
    [ 6,  4,  4, 22],
    [11,  3, 12, 14]], order=31)

In [7]: A[:,2] = A[:,1] * GF(17); A
Out[7]:
GF([[13, 20, 30,  0],
    [ 3, 21, 16, 19],
    [ 6,  4,  6, 22],
    [11,  3, 20, 14]], order=31)

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

(continues on next page)

(continued from previous page)

```
    [ 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([[ 6, 18, 12,  4],
    [12, 25,  4,  9],
    [20, 10,  8, 21],
    [11, 14, 16, 30]], order=31)

In [12]: A[3,:] = A[2,:] * GF(8); A
Out[12]:
GF([[ 6, 18, 12,  4],
    [12, 25,  4,  9],
    [20, 10,  8, 21],
    [ 5, 18,  2, 13]], order=31)

In [13]: A.row_reduce()
Out[13]:
GF([[ 1,  0,  0,  9],
    [ 0,  1,  0,  6],
    [ 0,  0,  1, 23],
    [ 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 $\mathrm{GF}(p)$.

This function is the inverse operation of the `Vector()` constructor. For an array with shape `(n1, n2)`, the output shape is `(n1, n2, m)`. By convention, the vectors are ordered from highest degree to 0-th degree.

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

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

Return type *galois.FieldArray*

Examples

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

In [2]: a = GF.Random(3); a
Out[2]: GF([25, 13, 25], order=2^6)

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

In [4]: vec = a.vector(); vec
Out[4]:
GF([[0, 1, 1, 0, 0, 1],
    [0, 0, 1, 1, 0, 1],
    [0, 1, 1, 0, 0, 1]], order=2)

In [5]: GF.Vector(vec)
Out[5]: GF([25, 13, 25], order=2^6)
```

galois.FieldClass

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

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

Important: *galois.FieldClass* is a metaclass for *galois.FieldArray* subclasses created with the class factory *galois.GF()* and should not be instantiated directly. This metaclass gives *galois.FieldArray* subclasses methods and attributes related to their Galois fields.

This class is included in the API to allow the user to test if a class is a Galois field array class.

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

In [2]: isinstance(GF, galois.FieldClass)
Out[2]: True
```

Constructors

__init__(name, bases, namespace, **kwargs)	

Methods

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

Attributes

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

`__init__`(*name*, *bases*, *namespace*, ***kwargs*)

`arithmetic_table`(*operation*, *x=None*, *y=None*)

Generates the specified arithmetic table for the Galois field.

Parameters

- **`operation`** (*`str`*) – The arithmetic operation, either `"+"`, `"-"`, `"*"`, or `"/"`.

- **x** (galois.FieldArray, *optional*) – Optionally specify the x values for the arithmetic table. The default is None which represents $\{0, \ldots, p^m - 1\}$.
- **y** (galois.FieldArray, *optional*) – Optionally specify the y values for the arithmetic table. The default is None which represents $\{0, \ldots, p^m - 1\}$ for addition, subtraction, and multiplication and $\{1, \ldots, p^m - 1\}$ for division.

Returns A UTF-8 formatted arithmetic table.

Return type str

Examples

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

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

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

     0  0 │ 1 │ 2 │ 3 │ 4 │ 5 │ 6 │ 7 │ 8
──────────────────────────────────
     1  1 │ 2 │ 0 │ 4 │ 5 │ 3 │ 7 │ 8 │ 6
──────────────────────────────────
     2  2 │ 0 │ 1 │ 5 │ 3 │ 4 │ 8 │ 6 │ 7
──────────────────────────────────
     3  3 │ 4 │ 5 │ 6 │ 7 │ 8 │ 0 │ 1 │ 2
──────────────────────────────────
     4  4 │ 5 │ 3 │ 7 │ 8 │ 6 │ 1 │ 2 │ 0
──────────────────────────────────
     5  5 │ 3 │ 4 │ 8 │ 6 │ 7 │ 2 │ 0 │ 1
──────────────────────────────────
     6  6 │ 7 │ 8 │ 0 │ 1 │ 2 │ 3 │ 4 │ 5
──────────────────────────────────
     7  7 │ 8 │ 6 │ 1 │ 2 │ 0 │ 4 │ 5 │ 3
──────────────────────────────────
     8  8 │ 6 │ 7 │ 2 │ 0 │ 1 │ 5 │ 3 │ 4
```

```
In [3]: GF.display("poly");

In [4]: print(GF.arithmetic_table("+"))

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

    0      0    │   1    │   2    │        │  + 1   │  + 2   │   2    │ 2 + 1 │ 2 +␣
↪2
────────────────────────────────────────────────────────────────────────────────
    1      1    │   2    │   0    │  + 1   │  + 2   │        │ 2 + 1  │ 2 + 2 │   2␣
↪
────────────────────────────────────────────────────────────────────────────────
    2      2    │   0    │   1    │  + 2   │        │  + 1   │ 2 + 2  │   2   │ 2 +␣
↪1
```

(continues on next page)

(continued from previous page)

```
──────────────────────────────────────────────────────────────────────────────
              │  + 1  │  + 2  │   2   │ 2 + 1 │ 2 + 2 │   0    │   1    │   2 ␣
↪
──────────────────────────────────────────────────────────────────────────────
  + 1   + 1   │  + 2  │       │ 2 + 1 │ 2 + 2 │   2   │   1    │   2    │   0 ␣
↪
──────────────────────────────────────────────────────────────────────────────
  + 2   + 2   │       │  + 1  │ 2 + 2 │   2   │ 2 + 1 │   2    │   0    │   1 ␣
↪
──────────────────────────────────────────────────────────────────────────────
    2     2   │ 2 + 1 │ 2 + 2 │   0    │   1    │   2    │        │  + 1  │  + 2␣
↪
──────────────────────────────────────────────────────────────────────────────
 2 + 1  2 + 1 │ 2 + 2 │   2   │   1    │   2    │   0    │  + 1  │  + 2  │     ␣
↪
──────────────────────────────────────────────────────────────────────────────
 2 + 2  2 + 2 │   2   │ 2 + 1 │   2    │   0    │   1    │  + 2  │       │  + 1␣
↪
```

```
In [5]: GF.display("power");

In [6]: print(GF.arithmetic_table("+"))

 x + y   0  │  1  │     │ ^2 │ ^3 │ ^4 │ ^5 │ ^6 │ ^7

     0   0  │  1  │     │ ^2 │ ^3 │ ^4 │ ^5 │ ^6 │ ^7
────────────────────────────────────────────────────
     1   1  │ ^4 │ ^2 │ ^7 │ ^6 │  0  │ ^3 │ ^5 │
────────────────────────────────────────────────────
          │ ^2 │ ^5 │ ^3 │  1  │ ^7 │  0  │ ^4 │ ^6
────────────────────────────────────────────────────
   ^2   ^2 │ ^7 │ ^3 │ ^6 │ ^4 │     │  1  │  0  │ ^5
────────────────────────────────────────────────────
   ^3   ^3 │ ^6 │  1  │ ^4 │ ^7 │ ^5 │ ^2 │     │  0
────────────────────────────────────────────────────
   ^4   ^4 │  0  │ ^7 │     │ ^5 │  1  │ ^6 │ ^3 │ ^2
────────────────────────────────────────────────────
   ^5   ^5 │ ^3 │  0  │  1  │ ^2 │ ^6 │     │ ^7 │ ^4
────────────────────────────────────────────────────
   ^6   ^6 │ ^5 │ ^4 │  0  │     │ ^3 │ ^7 │ ^2 │  1
────────────────────────────────────────────────────
   ^7   ^7 │     │ ^6 │ ^5 │  0  │ ^2 │ ^4 │  1  │ ^3
```

```
In [7]: GF.display("poly");

In [8]: x = GF.Random(5); x
Out[8]: GF([2 + 1,  + 2, 2, 0, ], order=3^2)

In [9]: y = GF.Random(3); y
```

(continues on next page)

(continued from previous page)

```
Out[9]: GF([2 + 1,  + 1, 2 + 2], order=3^2)

In [10]: print(GF.arithmetic_table("+", x=x, y=y))

  x + y   2 + 1 │  + 1  │ 2 + 2

 2 + 1   + 2  │   2   │
────────────────────────────────
   + 2     0   │   2   │   1
────────────────────────────────
     2    + 1  │   1   │  + 2
────────────────────────────────
       0  2 + 1 │  + 1  │ 2 + 2
────────────────────────────────
          1    │ 2 + 1 │   2


In [11]: GF.display();
```

compile(*mode*)

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

This function updates *ufunc_mode*.

Parameters mode (*str*) – The ufunc calculation mode.

- "auto": Selects "jit-lookup" for fields with order less than 2^{20}, "jit-calculate" for larger fields, and "python-calculate" for fields whose elements cannot be represented with numpy.int64.
- "jit-lookup": JIT compiles arithmetic ufuncs to use Zech log, log, and anti-log lookup tables for efficient computation. In the few cases where explicit calculation is faster than table lookup, explicit calculation is used.
- "jit-calculate": JIT compiles arithmetic ufuncs to use explicit calculation. The "jit-calculate" mode is designed for large fields that cannot or should not store lookup tables in RAM. Generally, the "jit-calculate" mode is slower than "jit-lookup".
- "python-calculate": Uses pure-python ufuncs with explicit calculation. This is reserved for fields whose elements cannot be represented with numpy.int64 and instead use numpy.object_ with python int (which has arbitrary precision).

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*.

> **Warning:** For the power representation, *np.log()* is computed on each element. So for large fields without lookup tables, displaying arrays in the power representation may take longer than expected.

Parameters mode (*str, optional*) – The field element representation.

- "int" (default): The element displayed as the integer representation of the polynomial. For example, $2x^2 + x + 2$ is an element of $\mathrm{GF}(3^3)$ and is equivalent to the integer $23 = 2 \cdot 3^2 + 3 + 2$.

- "poly": The element as a polynomial over $\mathrm{GF}(p)$ of degree less than m. For example, $2x^2 + x + 2$ is an element of $\mathrm{GF}(3^3)$.
- "power": The element as a power of the primitive element, see *FieldClass.primitive_element*. For example, $2x^2 + x + 2 = \alpha^5$ in $\mathrm{GF}(3^3)$ with irreducible polynomial $x^3 + 2x + 1$ and primitive element $\alpha = x$.

Examples

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

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

In [2]: print(GF.properties)
GF(3^3):
  characteristic: 3
  degree: 3
  order: 27
  irreducible_poly: x^3 + 2x + 1
  is_primitive_poly: True
  primitive_element: x

In [3]: a = GF(23); a
Out[3]: GF(23, order=3^3)

# Permanently set the display mode to the polynomial representation
In [4]: GF.display("poly"); a
Out[4]: GF(2^2 +  + 2, order=3^3)

# Permanently set the display mode to the power representation
In [5]: GF.display("power"); a
Out[5]: GF(^5, order=3^3)

# Permanently reset the default display mode to the integer representation
In [6]: GF.display(); a
Out[6]: GF(23, order=3^3)
```

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

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

```
In [7]: GF = galois.GF(2**8)

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

In [9]: a = GF.Random()
```

(continues on next page)

(continued from previous page)

```
In [10]: print(GF.display_mode, a)
int GF(236, order=2^8)

In [11]: with GF.display("poly"):
   ....:     print(GF.display_mode, a)
   ....:
poly GF(^7 + ^6 + ^5 + ^3 + ^2, order=2^8)

In [12]: with GF.display("power"):
   ....:     print(GF.display_mode, a)
   ....:
power GF(^122, order=2^8)

# The display mode is reset after exiting the context manager
In [13]: print(GF.display_mode, a)
int GF(236, order=2^8)
```

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

```
In [14]: GF = galois.GF(2**8, irreducible_poly=galois.Poly.Degrees([8,4,3,1,0]))

In [15]: print(GF.properties)
GF(2^8):
  characteristic: 2
  degree: 8
  order: 256
  irreducible_poly: x^8 + x^4 + x^3 + x + 1
  is_primitive_poly: False
  primitive_element: x + 1

In [16]: a = GF.Random()

In [17]: print(GF.display_mode, a)
int GF(62, order=2^8)

In [18]: with GF.display("poly"):
   ....:     print(GF.display_mode, a)
   ....:
poly GF(x^5 + x^4 + x^3 + x^2 + x, order=2^8)

In [19]: with GF.display("power"):
   ....:     print(GF.display_mode, a)
   ....:
power GF(^218, order=2^8)

# The display mode is reset after exiting the context manager
In [20]: print(GF.display_mode, a)
int GF(62, order=2^8)
```

repr_table(*primitive_element=None*, *sort='power'*)

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

Parameters

- **primitive_element** (`galois.FieldArray`, *optional*) – The primitive element to use for the power representation. The default is `None` which uses the field's default primitive element, *primitive_element*.
- **sort** (*str, optional*) – The sorting method for the table, either `"power"` (default), `"poly"`, `"vector"`, or `"int"`. Sorting by "power" will order the rows of the table by ascending powers of the primitive element. Sorting by any of the others will order the rows in lexicographically-increasing polynomial/vector order, which is equivalent to ascending order of the integer representation.

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

Return type str

Examples

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

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

Generate a representation table for $\mathrm{GF}(2^4)$. Since $x^4 + x + 1$ is a primitive polynomial, x is a primitive element of the field. Notice, $\mathrm{ord}(x) = 15$.

```
In [3]: print(GF.repr_table())

 Power │    Polynomial     │    Vector    │ Integer

   0   │         0         │ [0, 0, 0, 0] │    0
──────────────────────────────────────────────────
  x^0  │         1         │ [0, 0, 0, 1] │    1
──────────────────────────────────────────────────
  x^1  │         x         │ [0, 0, 1, 0] │    2
──────────────────────────────────────────────────
  x^2  │        x^2        │ [0, 1, 0, 0] │    4
──────────────────────────────────────────────────
  x^3  │        x^3        │ [1, 0, 0, 0] │    8
──────────────────────────────────────────────────
  x^4  │       x + 1       │ [0, 0, 1, 1] │    3
──────────────────────────────────────────────────
  x^5  │      x^2 + x      │ [0, 1, 1, 0] │    6
──────────────────────────────────────────────────
  x^6  │     x^3 + x^2     │ [1, 1, 0, 0] │   12
──────────────────────────────────────────────────
  x^7  │    x^3 + x + 1    │ [1, 0, 1, 1] │   11
```

(continues on next page)

(continued from previous page)

```
─────────────────────────────────────────────────
 x^8  │      x^2 + 1      │ [0, 1, 0, 1] │    5
─────────────────────────────────────────────────
 x^9  │      x^3 + x      │ [1, 0, 1, 0] │    10
─────────────────────────────────────────────────
 x^10 │    x^2 + x + 1    │ [0, 1, 1, 1] │    7
─────────────────────────────────────────────────
 x^11 │   x^3 + x^2 + x   │ [1, 1, 1, 0] │    14
─────────────────────────────────────────────────
 x^12 │ x^3 + x^2 + x + 1 │ [1, 1, 1, 1] │    15
─────────────────────────────────────────────────
 x^13 │   x^3 + x^2 + 1   │ [1, 1, 0, 1] │    13
─────────────────────────────────────────────────
 x^14 │      x^3 + 1      │ [1, 0, 0, 1] │    9
```

Generate a representation table for $\mathrm{GF}(2^4)$ using a different primitive element x^3+x^2+x. Notice, $\mathrm{ord}(x^3+x^2+x) = 15$.

```
In [4]: alpha = GF.primitive_elements[-1]

In [5]: print(GF.repr_table(alpha))
       Power         │    Polynomial     │    Vector    │ Integer
         0           │         0         │ [0, 0, 0, 0] │    0
──────────────────────────────────────────────────────────────────
 (x^3 + x^2 + x)^0   │         1         │ [0, 0, 0, 1] │    1
──────────────────────────────────────────────────────────────────
 (x^3 + x^2 + x)^1   │   x^3 + x^2 + x   │ [1, 1, 1, 0] │    14
──────────────────────────────────────────────────────────────────
 (x^3 + x^2 + x)^2   │    x^3 + x + 1    │ [1, 0, 1, 1] │    11
──────────────────────────────────────────────────────────────────
 (x^3 + x^2 + x)^3   │        x^3        │ [1, 0, 0, 0] │    8
──────────────────────────────────────────────────────────────────
 (x^3 + x^2 + x)^4   │      x^3 + 1      │ [1, 0, 0, 1] │    9
──────────────────────────────────────────────────────────────────
 (x^3 + x^2 + x)^5   │    x^2 + x + 1    │ [0, 1, 1, 1] │    7
──────────────────────────────────────────────────────────────────
 (x^3 + x^2 + x)^6   │     x^3 + x^2     │ [1, 1, 0, 0] │    12
──────────────────────────────────────────────────────────────────
 (x^3 + x^2 + x)^7   │        x^2        │ [0, 1, 0, 0] │    4
──────────────────────────────────────────────────────────────────
 (x^3 + x^2 + x)^8   │   x^3 + x^2 + 1   │ [1, 1, 0, 1] │    13
──────────────────────────────────────────────────────────────────
 (x^3 + x^2 + x)^9   │      x^3 + x      │ [1, 0, 1, 0] │    10
──────────────────────────────────────────────────────────────────
 (x^3 + x^2 + x)^10  │      x^2 + x      │ [0, 1, 1, 0] │    6
──────────────────────────────────────────────────────────────────
 (x^3 + x^2 + x)^11  │         x         │ [0, 0, 1, 0] │    2
──────────────────────────────────────────────────────────────────
 (x^3 + x^2 + x)^12  │ x^3 + x^2 + x + 1 │ [1, 1, 1, 1] │    15
```

(continues on next page)

(continued from previous page)

```
 (x^3 + x^2 + x)^13 │       x^2 + 1       │ [0, 1, 0, 1] │    5
 (x^3 + x^2 + x)^14 │        x + 1        │ [0, 0, 1, 1] │    3
```

Generate a representation table for $\mathrm{GF}(2^4)$ using a non-primitive element $x^3 + x^2$. Notice, $\mathrm{ord}(x^3 + x^2) = 5 \neq 15$.

```
In [6]: beta = GF("x^3 + x^2")

In [7]: print(GF.repr_table(beta))
     Power       │     Polynomial      │    Vector    │ Integer
       0         │          0          │ [0, 0, 0, 0] │    0
 (x^3 + x^2)^0   │          1          │ [0, 0, 0, 1] │    1
 (x^3 + x^2)^1   │      x^3 + x^2      │ [1, 1, 0, 0] │    12
 (x^3 + x^2)^2   │ x^3 + x^2 + x + 1   │ [1, 1, 1, 1] │    15
 (x^3 + x^2)^3   │         x^3         │ [1, 0, 0, 0] │    8
 (x^3 + x^2)^4   │       x^3 + x       │ [1, 0, 1, 0] │    10
 (x^3 + x^2)^5   │          1          │ [0, 0, 0, 1] │    1
 (x^3 + x^2)^6   │      x^3 + x^2      │ [1, 1, 0, 0] │    12
 (x^3 + x^2)^7   │ x^3 + x^2 + x + 1   │ [1, 1, 1, 1] │    15
 (x^3 + x^2)^8   │         x^3         │ [1, 0, 0, 0] │    8
 (x^3 + x^2)^9   │       x^3 + x       │ [1, 0, 1, 0] │    10
 (x^3 + x^2)^10  │          1          │ [0, 0, 0, 1] │    1
 (x^3 + x^2)^11  │      x^3 + x^2      │ [1, 1, 0, 0] │    12
 (x^3 + x^2)^12  │ x^3 + x^2 + x + 1   │ [1, 1, 1, 1] │    15
 (x^3 + x^2)^13  │         x^3         │ [1, 0, 0, 0] │    8
 (x^3 + x^2)^14  │       x^3 + x       │ [1, 0, 1, 0] │    10
```

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, display="poly")

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

In [3]: a = GF.Random(low=1); a
Out[3]: GF(^4 + ^3 + ^2 + , 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(low=1); a
Out[7]: GF(2, order=31)

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

Type int

property default_ufunc_mode

The default ufunc arithmetic mode for this Galois field.

Examples

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

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

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

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

Type str

property degree

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

Examples

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

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

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

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

Type int

property display_mode

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

Examples

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

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

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

In [3]: a = GF.Random()

In [4]: print(GF.display_mode, a)
int GF(185, order=2^8)

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

In [6]: with GF.display("power"):
   ...:     print(GF.display_mode, a)
   ...:
power GF(^60, order=2^8)

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

(continues on next page)

(continued from previous page)

```
In [7]: print(GF.display_mode, a)
int GF(185, order=2^8)
```

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

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

In [9]: print(GF.properties)
GF(2^8):
  characteristic: 2
  degree: 8
  order: 256
  irreducible_poly: x^8 + x^4 + x^3 + x + 1
  is_primitive_poly: False
  primitive_element: x + 1

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

In [11]: print(GF.display_mode, a)
int GF(249, order=2^8)

In [12]: with GF.display("poly"):
   ....:     print(GF.display_mode, a)
   ....:
poly GF(x^7 + x^6 + x^5 + x^4 + x^3 + 1, order=2^8)

In [13]: with GF.display("power"):
   ....:     print(GF.display_mode, a)
   ....:
power GF(^99, order=2^8)

# The display mode is reset after exiting the context manager
In [14]: print(GF.display_mode, a)
int GF(249, order=2^8)
```

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

```
In [15]: with GF.display("power"):
   ....:     print(GF.display_mode, a)
   ....:
power GF(^99, order=2^8)

# The display mode is reset after exiting the context manager
In [16]: print(GF.display_mode, a)
int GF(249, order=2^8)
```

Type str

property dtypes

List of valid integer `numpy.dtype` values that are compatible with this Galois field. Creating an array with an unsupported dtype will throw a `TypeError` exception.

Examples

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

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

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

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

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

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

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

Type list

property irreducible_poly

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

Examples

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

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

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

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

Type *galois.Poly*

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

Examples

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

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

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

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

Type bool

property is_prime_field
Indicates if the field's order is prime.

Examples

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

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

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

(continues on next page)

(continued from previous page)

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

Type bool

property is_primitive_poly

Indicates whether the *irreducible_poly* is a primitive polynomial. If so, x is a primitive element of the Galois field.

Examples

```
In [1]: GF = galois.GF(2**8, display="poly")

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(, order=2^8)

# The irreducible polynomial is a primitive polynomial if 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
```

Here is an example using the $\mathrm{GF}(2^8)$ field from AES, which does not use a primitive polynomial.

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

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(x + 1, order=2^8)

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

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

Type bool

property name

The Galois field name.

Examples

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

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

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

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

Type str

property order

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

Examples

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

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

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

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

Type int

property prime_subfield

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

Examples

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

In [2]: print(galois.GF(2**8).prime_subfield.properties)
```

(continues on next page)

(continued from previous page)

```
GF(2):
  characteristic: 2
  degree: 1
  order: 2
  irreducible_poly: x + 1
  is_primitive_poly: True
  primitive_element: 1

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

In [4]: print(galois.GF(7**5).prime_subfield.properties)
GF(7):
  characteristic: 7
  degree: 1
  order: 7
  irreducible_poly: x + 4
  is_primitive_poly: True
  primitive_element: 3
```

Type *galois.FieldClass*

property primitive_element

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

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

Examples

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

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

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

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

Type *galois.FieldArray*

property primitive_elements

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

Examples

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

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

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

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

Type *galois.FieldArray*

property properties

A formatted string displaying relevant properties of the Galois field.

Examples

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

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

(continues on next page)

(continued from previous page)

```
  primitive_element: x

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

In [4]: GF = galois.GF(7**5); print(GF.properties)
GF(7^5):
  characteristic: 7
  degree: 5
  order: 16807
  irreducible_poly: x^5 + x + 4
  is_primitive_poly: True
  primitive_element: x
```

Type str

property ufunc_mode

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

Examples

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

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

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

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

Type str

property ufunc_modes

All supported ufunc modes for this Galois field array class.

Examples

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

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

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

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

Type list

Pre-made Galois field classes

GF2(array[, dtype, copy, order, ndmin])	An array over GF(2).

galois.GF2

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

An array over GF(2).

This class is a pre-generated `galois.FieldArray` subclass generated with `galois.GF(2)` and is included in the API for convenience. See `galois.FieldArray` and `galois.FieldClass` for more complete documentation and examples.

Examples

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

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

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

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

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

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

(continues on next page)

(continued from previous page)

```
  irreducible_poly: x + 1
  is_primitive_poly: True
  primitive_element: 1

# 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 1-D Galois field array of the field's elements $\{0, \dots, p^m - 1\}$.

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

Returns A 1-D Galois field array of all the field's elements.

Return type *galois.FieldArray*

Examples

```
In [10]: GF = galois.GF(2**4)

In [11]: GF.Elements()
Out[11]:
GF([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15],
   order=2^4)
```

As usual, Galois field elements can be displayed in either the "integer" (default), "polynomial", or "power" representation. This can be changed by calling *galois.FieldClass.display()*.

```
# Permanently set the display mode to "poly"
In [12]: GF.display("poly");
```

(continues on next page)

(continued from previous page)

```
In [13]: GF.Elements()
Out[13]:
GF([0, 1, ,  + 1, ^2, ^2 + 1, ^2 + , ^2 +  + 1, ^3, ^3 + 1,
    ^3 + , ^3 +  + 1, ^3 + ^2, ^3 + ^2 + 1, ^3 + ^2 + ,
    ^3 + ^2 +  + 1], order=2^4)

# Temporarily set the display mode to "power"
In [14]: with GF.display("power"):
   ....:     print(GF.Elements())
   ....:
GF([0, 1, , ^4, ^2, ^8, ^5, ^10, ^3, ^14, ^9, ^7, ^6, ^13,
    ^11, ^12], order=2^4)

# Reset the display mode to "int"
In [15]: GF.display();
```

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 unsigned dtype for this class, i.e. the first element in `galois.FieldClass.dtypes`.

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

Return type *galois.FieldArray*

Examples

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

In [17]: GF.Identity(4)
Out[17]:
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** (*int, 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-D array. A 2-tuple, e.g. `(M,N)`, represents a 2-D 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 unsigned dtype for this class, i.e. the first element in `galois.FieldClass.dtypes`.

Returns A Galois field array of ones.

Return type *galois.FieldArray*

Examples

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

In [19]: GF.Ones((2,5))
Out[19]:
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** (*int, 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-D array. A 2-tuple, e.g. `(M,N)`, represents a 2-D array with each element indicating the size in each dimension.
- **low** (*int, optional*) – The lowest value (inclusive) of a random field element in its integer representation. The default is 0.
- **high** (*int, optional*) – The highest value (exclusive) of a random field element in its integer representation. 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 unsigned dtype for this class, i.e. the first element in `galois.FieldClass.dtypes`.

Returns A Galois field array of random field elements.

Return type *galois.FieldArray*

Examples

```
In [20]: GF = galois.GF(31)

In [21]: GF.Random((2,5))
Out[21]:
GF([[17, 29, 20, 28,  6],
    [14, 18, 11, 21, 26]], order=31)
```

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

Creates a 1-D Galois field array with a range of field elements.

Parameters

- **start** (*int*) – The starting Galois field value (inclusive) in its integer representation.
- **stop** (*int*) – The stopping Galois field value (exclusive) in its integer representation.
- **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 unsigned dtype for this class, i.e. the first element in *galois.FieldClass.dtypes*.

Returns A 1-D Galois field array of a range of field elements.

Return type *galois.FieldArray*

Examples

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

In [23]: GF.Range(10,20)
Out[23]: GF([10, 11, 12, 13, 14, 15, 16, 17, 18, 19], order=31)
```

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

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

Parameters

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

Returns The $m \times n$ Vandermonde matrix.

Return type *galois.FieldArray*

Examples

```
In [24]: GF = galois.GF(2**3)

In [25]: a = GF.primitive_element

In [26]: V = GF.Vandermonde(a, 7, 7)

In [27]: 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 $\mathrm{GF}(p)$.

This function is the inverse operation of the `vector()` method.

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)`. By convention, the vectors are ordered from highest degree to 0-th degree.
- **dtype** (*numpy.dtype, optional*) – The `numpy.dtype` of the array elements. The default is `None` which represents the smallest unsigned dtype for this class, i.e. the first element in `galois.FieldClass.dtypes`.

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

Return type *galois.FieldArray*

Examples

```
In [28]: GF = galois.GF(2**6)

In [29]: vec = galois.GF2.Random((3,6)); vec
Out[29]:
GF([[1, 1, 0, 1, 1, 0],
    [1, 0, 1, 0, 1, 1],
    [1, 1, 0, 0, 0, 0]], order=2)

In [30]: a = GF.Vector(vec); a
Out[30]: GF([54, 43, 48], order=2^6)

In [31]: with GF.display("poly"):
   ....:     print(a)
   ....:
GF([^5 + ^4 + ^2 + , ^5 + ^3 +  + 1, ^5 + ^4], order=2^6)

In [32]: a.vector()
Out[32]:
GF([[1, 1, 0, 1, 1, 0],
    [1, 0, 1, 0, 1, 1],
    [1, 1, 0, 0, 0, 0]], order=2)
```

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

Creates a Galois field array with all zeros.

Parameters

- **shape** (*int, 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-D array. A 2-tuple, e.g. `(M,N)`, represents a 2-D 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 unsigned dtype for this class, i.e. the first element in `galois.FieldClass.dtypes`.

Returns A Galois field array of zeros.

Return type *galois.FieldArray*

Examples

```
In [33]: GF = galois.GF(31)

In [34]: GF.Zeros((2,5))
Out[34]:
GF([[0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0]], order=31)
```

`__init__`(*array*, *dtype=None*, *copy=True*, *order='K'*, *ndmin=0*)
Creates an array over $\mathrm{GF}(p^m)$.

Parameters

- **array** (*int*, *str*, *tuple*, *list*, *numpy.ndarray*, `galois.FieldArray`) – The input array-like object to be converted to a Galois field array. See the examples section for demonstations of array creation using each input type. See see *galois.FieldClass.display()* and *galois.FieldClass.display_mode* for a description of the "integer" and "polynomial" representation of Galois field elements.
  - `int`: A single integer, which is the "integer representation" of a Galois field element, creates a 0-D array.
  - `str`: A single string, which is the "polynomial representation" of a Galois field element, creates a 0-D array.
  - `tuple`, `list`: A list or tuple (or nested lists/tuples) of ints or strings (which can be mix-and-matched) creates an array of

  Galois field elements from their integer or polynomial representations. * `numpy.ndarray`, *galois.FieldArray*: An array of ints creates a copy of the array over this specific field.
- **dtype** (*numpy.dtype*, *optional*) – The `numpy.dtype` of the array elements. The default is `None` which represents the smallest unsigned dtype for this class, i.e. the first element in *galois.FieldClass.dtypes*.
- **copy** (*bool*, *optional*) – The `copy` keyword argument from `numpy.array()`. The default is `True` which makes a copy of the input array.
- **order** (*str*, *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 An array over $\mathrm{GF}(p^m)$.

Return type *galois.FieldArray*

7.1.2 Prime field functions

Primitive roots

`primitive_root`(n[, start, stop, reverse])	Finds the smallest primitive root modulo n.
`primitive_roots`(n[, start, stop, reverse])	Finds all primitive roots modulo n.
`is_primitive_root`(g, n)	Determines if g is a primitive root modulo n.

galois.primitive_root

galois.**primitive_root**(*n*, *start=1*, *stop=None*, *reverse=False*)

Finds the smallest primitive root modulo n.

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

Notes

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, $(\mathbb{Z}/n\mathbb{Z})^\times = \{g^0, g^1, g^2, \dots, g^{\phi(n)-1}\}$ where $\phi(n)$ is order of the group. If $(\mathbb{Z}/n\mathbb{Z})^\times$ is cyclic, the number of primitive roots modulo n is given by $\phi(\phi(n))$.

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
- http://www.numbertheory.org/courses/MP313/lectures/lecture7/page1.html

Examples

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

```
# n is of type 2*p^k, which is cyclic
In [1]: n = 14

In [2]: galois.is_cyclic(n)
Out[2]: True

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

# Euler's totient function counts the "totatives", positive integers coprime with n
In [4]: phi = galois.euler_phi(n); phi
Out[4]: 6

In [5]: len(Znx) == phi
Out[5]: True

# The primitive roots are the elements in Znx that multiplicatively generate the␣
↪group
In [6]: for a in Znx:
   ...:     span = set([pow(a, i, n) for i in range(1, phi + 1)])
   ...:     primitive_root = span == Znx
   ...:     print("Element: {:2d}, Span: {:<20}, Primitive root: {}".format(a,␣
↪str(span), primitive_root))
   ...:
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

# Find the smallest primitive root
In [7]: galois.primitive_root(n)
Out[7]: 3

# Find all primitive roots
In [8]: roots = galois.primitive_roots(n); roots
Out[8]: [3, 5]

# Euler's totient function ((n)) counts the primitive roots of n
In [9]: len(roots) == galois.euler_phi(phi)
Out[9]: True
```

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

```
# n is of type p1^k1 * p2^k2, which is not cyclic
In [10]: n = 15
```

(continues on next page)

(continued from previous page)

```
In [11]: galois.is_cyclic(n)
Out[11]: False

# The congruence class coprime with n
In [12]: Znx = set([a for a in range(1, n) if math.gcd(n, a) == 1]); Znx
Out[12]: {1, 2, 4, 7, 8, 11, 13, 14}

# Euler's totient function counts the "totatives", positive integers coprime with n
In [13]: phi = galois.euler_phi(n); phi
Out[13]: 8

In [14]: len(Znx) == phi
Out[14]: True

# The primitive roots are the elements in Znx that multiplicatively generate the␣
↪group
In [15]: 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

# Find the smallest primitive root
In [16]: galois.primitive_root(n)

# Find all primitive roots
In [17]: roots = galois.primitive_roots(n); roots
Out[17]: []

# Note the max order of any element is 4, not 8, which is Carmichael's lambda␣
↪function
In [18]: galois.carmichael_lambda(n)
Out[18]: 4
```

The algorithm is also efficient for very large n.

```
In [19]: n = 1000000000000000035000061

In [20]: galois.primitive_root(n)
Out[20]: 7
```

galois.primitive_roots

galois.**primitive_roots**(n, *start=1*, *stop=None*, *reverse=False*)
Finds all primitive roots modulo n.

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

Notes

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, $(\mathbb{Z}/n\mathbb{Z})^\times = \{g^0, g^1, g^2, \dots, g^{\phi(n)-1}\}$ where $\phi(n)$ is order of the group. If $(\mathbb{Z}/n\mathbb{Z})^\times$ is cyclic, the number of primitive roots modulo n is given by $\phi(\phi(n))$.

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
- http://www.numbertheory.org/courses/MP313/lectures/lecture7/page1.html

Examples

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

```
# n is of type 2*p^k, which is cyclic
In [1]: n = 14

In [2]: galois.is_cyclic(n)
Out[2]: True

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

# Euler's totient function counts the "totatives", positive integers coprime with n
In [4]: phi = galois.euler_phi(n); phi
Out[4]: 6
```

(continues on next page)

(continued from previous page)

```
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

# Find the smallest primitive root
In [7]: galois.primitive_root(n)
Out[7]: 3

# Find all primitive roots
In [8]: roots = galois.primitive_roots(n); roots
Out[8]: [3, 5]

# Euler's totient function ((n)) counts the primitive roots of n
In [9]: len(roots) == galois.euler_phi(phi)
Out[9]: True
```

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

```
# n is of type p1^k1 * p2^k2, which is not cyclic
In [10]: n = 15

In [11]: galois.is_cyclic(n)
Out[11]: False

# The congruence class coprime with n
In [12]: Znx = set([a for a in range(1, n) if math.gcd(n, a) == 1]); Znx
Out[12]: {1, 2, 4, 7, 8, 11, 13, 14}

# Euler's totient function counts the "totatives", positive integers coprime with n
In [13]: phi = galois.euler_phi(n); phi
Out[13]: 8

In [14]: len(Znx) == phi
Out[14]: True

# The primitive roots are the elements in Znx that multiplicatively generate the␣
↪group
```

(continues on next page)

(continued from previous page)

```
In [15]: 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

# Find the smallest primitive root
In [16]: galois.primitive_root(n)

# Find all primitive roots
In [17]: roots = galois.primitive_roots(n); roots
Out[17]: []

# Note the max order of any element is 4, not 8, which is Carmichael's lambda␣
↪function
In [18]: galois.carmichael_lambda(n)
Out[18]: 4
```

galois.is_primitive_root

`galois.is_primitive_root(g, n)`

Determines if g is a primitive root modulo n.

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

Notes

g is a primitive root if the totatives of n, the positive integers $1 \le a < n$ 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, $(\mathbb{Z}/n\mathbb{Z})^\times = \{g^0, g^1, g^2, \dots, g^{\phi(n)-1}\}$ where $\phi(n)$ is order of the group. If $(\mathbb{Z}/n\mathbb{Z})^\times$ is cyclic, the number of primitive roots modulo n is given by $\phi(\phi(n))$.

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]
```

7.1.3 Extension field functions

Irreducible polynomials

irreducible_poly(order, degree[, method])	Returns a monic irreducible polynomial $f(x)$ over $\mathrm{GF}(q)$ with degree m.
irreducible_polys(order, degree)	Returns all monic irreducible polynomials $f(x)$ over $\mathrm{GF}(q)$ with degree m.
is_irreducible(poly)	Determines whether the polynomial $f(x)$ over $\mathrm{GF}(p^m)$ is irreducible.

galois.irreducible_poly

`galois.irreducible_poly`(*order*, *degree*, *method='min'*)

Returns a monic irreducible polynomial $f(x)$ over $\mathrm{GF}(q)$ with degree m.

Parameters

- **order** (*int*) – The prime power order q of the field $\mathrm{GF}(q)$ that the polynomial is over.
- **degree** (*int*) – The degree m of the desired irreducible polynomial.
- **method** (*str, optional*) – The search method for finding the irreducible polynomial.
  - `"min"` (default): Returns the lexicographically-minimal monic irreducible polynomial.
  - `"max"`: Returns the lexicographically-maximal monic irreducible polynomial.
  - `"random"`: Returns a randomly generated degree-m monic irreducible polynomial.

Returns The degree-m monic irreducible polynomial over $\mathrm{GF}(q)$.

Return type *galois.Poly*

Notes

If $f(x)$ is an irreducible polynomial over $\mathrm{GF}(q)$ and $a \in \mathrm{GF}(q) \backslash \{0\}$, then $a \cdot f(x)$ is also irreducible. In addition to other applications, $f(x)$ produces the field extension $\mathrm{GF}(q^m)$ of $\mathrm{GF}(q)$.

Examples

The lexicographically-minimal, monic irreducible polynomial over $\mathrm{GF}(7)$ with degree 5.

```
In [1]: p = galois.irreducible_poly(7, 5); p
Out[1]: Poly(x^5 + x + 3, GF(7))

In [2]: galois.is_irreducible(p)
Out[2]: True
```

Irreducible polynomials scaled by non-zero field elements are also irreducible.

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

In [4]: galois.is_irreducible(p * GF(3))
Out[4]: True
```

A random, monic irreducible polynomial over $\mathrm{GF}(7^2)$ with degree 3.

```
In [5]: p = galois.irreducible_poly(7**2, 3, method="random"); p
Out[5]: Poly(x^3 + 38x^2 + 31x + 13, GF(7^2))

In [6]: galois.is_irreducible(p)
Out[6]: True
```

galois.irreducible_polys

`galois.irreducible_polys`(*order*, *degree*)

Returns all monic irreducible polynomials $f(x)$ over $\mathrm{GF}(q)$ with degree m.

Parameters

- **order** (*int*) – The prime power order q of the field $\mathrm{GF}(q)$ that the polynomial is over.
- **degree** (*int*) – The degree m of the desired irreducible polynomial.

Returns All degree-m monic irreducible polynomials over $\mathrm{GF}(q)$.

Return type list

Notes

If $f(x)$ is an irreducible polynomial over $\mathrm{GF}(q)$ and $a \in \mathrm{GF}(q) \backslash \{0\}$, then $a \cdot f(x)$ is also irreducible. In addition to other applications, $f(x)$ produces the field extension $\mathrm{GF}(q^m)$ of $\mathrm{GF}(q)$.

Examples

All monic irreducible polynomials over $\mathrm{GF}(2)$ with degree 5.

```
In [1]: galois.irreducible_polys(2, 5)
Out[1]:
[Poly(x^5 + x^2 + 1, GF(2)),
 Poly(x^5 + x^3 + 1, GF(2)),
 Poly(x^5 + x^3 + x^2 + x + 1, GF(2)),
 Poly(x^5 + x^4 + x^2 + x + 1, GF(2)),
 Poly(x^5 + x^4 + x^3 + x + 1, GF(2)),
 Poly(x^5 + x^4 + x^3 + x^2 + 1, GF(2))]
```

All monic irreducible polynomials over $\mathrm{GF}(3^2)$ with degree 2.

```
In [2]: galois.irreducible_polys(3**2, 2)
Out[2]:
[Poly(x^2 + 3, GF(3^2)),
 Poly(x^2 + 5, GF(3^2)),
 Poly(x^2 + 6, GF(3^2)),
 Poly(x^2 + 7, GF(3^2)),
 Poly(x^2 + x + 3, GF(3^2)),
 Poly(x^2 + x + 4, GF(3^2)),
 Poly(x^2 + x + 7, GF(3^2)),
 Poly(x^2 + x + 8, GF(3^2)),
 Poly(x^2 + 2x + 3, GF(3^2)),
 Poly(x^2 + 2x + 4, GF(3^2)),
 Poly(x^2 + 2x + 7, GF(3^2)),
 Poly(x^2 + 2x + 8, GF(3^2)),
 Poly(x^2 + 3x + 1, GF(3^2)),
 Poly(x^2 + 3x + 2, GF(3^2)),
 Poly(x^2 + 3x + 6, GF(3^2)),
 Poly(x^2 + 3x + 7, GF(3^2)),
 Poly(x^2 + 4x + 4, GF(3^2)),
 Poly(x^2 + 4x + 5, GF(3^2)),
 Poly(x^2 + 4x + 6, GF(3^2)),
 Poly(x^2 + 4x + 8, GF(3^2)),
 Poly(x^2 + 5x + 1, GF(3^2)),
 Poly(x^2 + 5x + 2, GF(3^2)),
 Poly(x^2 + 5x + 3, GF(3^2)),
 Poly(x^2 + 5x + 5, GF(3^2)),
 Poly(x^2 + 6x + 1, GF(3^2)),
 Poly(x^2 + 6x + 2, GF(3^2)),
 Poly(x^2 + 6x + 6, GF(3^2)),
 Poly(x^2 + 6x + 7, GF(3^2)),
 Poly(x^2 + 7x + 1, GF(3^2)),
 Poly(x^2 + 7x + 2, GF(3^2)),
 Poly(x^2 + 7x + 3, GF(3^2)),
```

(continues on next page)

(continued from previous page)

```
Poly(x^2 + 7x + 5, GF(3^2)),
Poly(x^2 + 8x + 4, GF(3^2)),
Poly(x^2 + 8x + 5, GF(3^2)),
Poly(x^2 + 8x + 6, GF(3^2)),
Poly(x^2 + 8x + 8, GF(3^2))]
```

galois.is_irreducible

galois.**is_irreducible**(*poly*)

Determines whether the polynomial $f(x)$ over $\mathrm{GF}(p^m)$ is irreducible.

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

Returns `True` if the polynomial is irreducible.

Return type bool

Notes

A polynomial $f(x) \in \mathrm{GF}(p^m)[x]$ is *reducible* over $\mathrm{GF}(p^m)$ if it can be represented as $f(x) = g(x)h(x)$ for some $g(x), h(x) \in \mathrm{GF}(p^m)[x]$ of strictly lower degree. If $f(x)$ is not reducible, it is said to be *irreducible*. Since Galois fields are not algebraically closed, such irreducible polynomials exist.

This function implements Rabin's irreducibility test. It says a degree-m polynomial $f(x)$ over $\mathrm{GF}(q)$ for prime power q is irreducible if and only if $f(x) \mid (x^{q^m} - x)$ and $\mathrm{gcd}(f(x),\ x^{q^{m_i}} - x) = 1$ for $1 \le i \le k$, where $m_i = m/p_i$ for the k prime divisors p_i of m.

References

- M. O. Rabin. Probabilistic algorithms in finite fields. SIAM Journal on Computing (1980), 273–280. https://apps.dtic.mil/sti/pdfs/ADA078416.pdf
- S. Gao and D. Panarino. Tests and constructions of irreducible polynomials over finite fields. https://www.math.clemson.edu/~sgao/papers/GP97a.pdf
- Section 4.5.1 from https://cacr.uwaterloo.ca/hac/about/chap4.pdf
- https://en.wikipedia.org/wiki/Factorization_of_polynomials_over_finite_fields

Examples

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

# f(x) has no roots in GF(2), a necessary but not sufficient condition of being
↪irreducible
In [2]: f.roots()
Out[2]: GF([], order=2)

In [3]: galois.is_irreducible(f)
Out[3]: True
```

```
In [4]: g = galois.irreducible_poly(2**4, 2, method="random"); g
Out[4]: Poly(x^2 + 2x + 2, GF(2^4))

In [5]: h = galois.irreducible_poly(2**4, 3, method="random"); h
Out[5]: Poly(x^3 + 4x^2 + 3x + 1, GF(2^4))

In [6]: f = g * h; f
Out[6]: Poly(x^5 + 6x^4 + 9x^3 + 15x^2 + 4x + 2, GF(2^4))

In [7]: galois.is_irreducible(f)
Out[7]: False
```

Primitive polynomials

primitive_poly(order, degree[, method])	Returns a monic primitive polynomial $f(x)$ over $\mathrm{GF}(q)$ with degree m.
primitive_polys(order, degree)	Returns all monic primitive polynomials $f(x)$ over $\mathrm{GF}(q)$ with degree m.
conway_poly(characteristic, degree)	Returns the Conway polynomial $C_{p,m}(x)$ over $\mathrm{GF}(p)$ with degree m.
matlab_primitive_poly(characteristic, degree)	Returns Matlab's default primitive polynomial $f(x)$ over $\mathrm{GF}(p)$ with degree m.
is_primitive(poly)	Determines whether the polynomial $f(x)$ over $\mathrm{GF}(q)$ is primitive.

galois.primitive_poly

galois.**primitive_poly**(*order*, *degree*, *method='min'*)

Returns a monic primitive polynomial $f(x)$ over $\mathrm{GF}(q)$ with degree m.

Parameters

- **order** (*int*) – The prime power order q of the field $\mathrm{GF}(q)$ that the polynomial is over.
- **degree** (*int*) – The degree m of the desired primitive polynomial.
- **method** (*str, optional*) – The search method for finding the primitive polynomial.
  - "min" (default): Returns the lexicographically-minimal monic primitive polynomial.
  - "max": Returns the lexicographically-maximal monic primitive polynomial.
  - "random": Returns a randomly generated degree-m monic primitive polynomial.

Returns The degree-m monic primitive polynomial over $\mathrm{GF}(q)$.

Return type *galois.Poly*

Notes

In addition to other applications, $f(x)$ produces the field extension $\mathrm{GF}(q^m)$ of $\mathrm{GF}(q)$. Since $f(x)$ is primitive, x is a primitive element α of $\mathrm{GF}(q^m)$ such that $\mathrm{GF}(q^m) = \{0, 1, \alpha, \alpha^2, \dots, \alpha^{q^m-2}\}$.

Examples

Notice *galois.primitive_poly()* returns the lexicographically-minimal primitive polynomial, whereas *galois.conway_poly()* returns the lexicographically-minimal primitive polynomial that is *consistent* with smaller Conway polynomials, which is not *necessarily* the same.

```
In [1]: galois.primitive_poly(2, 4)
Out[1]: Poly(x^4 + x + 1, GF(2))

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

```
In [3]: galois.primitive_poly(7, 10)
Out[3]: Poly(x^10 + 5x^2 + x + 5, GF(7))

In [4]: galois.conway_poly(7, 10)
Out[4]: Poly(x^10 + x^6 + x^5 + 4x^4 + x^3 + 2x^2 + 3x + 3, GF(7))
```

galois.primitive_polys

`galois.primitive_polys(order, degree)`

Returns all monic primitive polynomials $f(x)$ over $\mathrm{GF}(q)$ with degree m.

Parameters

- **order** (*int*) – The prime order q of the field $\mathrm{GF}(q)$ that the polynomial is over.
- **degree** (*int*) – The degree m of the desired primitive polynomial.

Returns All degree-m monic primitive polynomials over $\mathrm{GF}(q)$.

Return type list

Notes

In addition to other applications, $f(x)$ produces the field extension $\mathrm{GF}(q^m)$ of $\mathrm{GF}(q)$. Since $f(x)$ is primitive, x is a primitive element α of $\mathrm{GF}(q^m)$ such that $\mathrm{GF}(q^m) = \{0, 1, \alpha, \alpha^2, \dots, \alpha^{q^m-2}\}$.

Examples

All monic primitive polynomials over $\mathrm{GF}(2)$ with degree 5.

```
In [1]: galois.primitive_polys(2, 5)
Out[1]:
[Poly(x^5 + x^2 + 1, GF(2)),
 Poly(x^5 + x^3 + 1, GF(2)),
 Poly(x^5 + x^3 + x^2 + x + 1, GF(2)),
```

(continues on next page)

(continued from previous page)

```
 Poly(x^5 + x^4 + x^2 + x + 1, GF(2)),
 Poly(x^5 + x^4 + x^3 + x + 1, GF(2)),
 Poly(x^5 + x^4 + x^3 + x^2 + 1, GF(2))]
```

All monic primitive polynomials over $\mathrm{GF}(3^2)$ with degree 2.

```
In [2]: galois.primitive_polys(3**2, 2)
Out[2]:
[Poly(x^2 + x + 3, GF(3^2)),
 Poly(x^2 + x + 7, GF(3^2)),
 Poly(x^2 + 2x + 3, GF(3^2)),
 Poly(x^2 + 2x + 7, GF(3^2)),
 Poly(x^2 + 3x + 6, GF(3^2)),
 Poly(x^2 + 3x + 7, GF(3^2)),
 Poly(x^2 + 4x + 5, GF(3^2)),
 Poly(x^2 + 4x + 6, GF(3^2)),
 Poly(x^2 + 5x + 3, GF(3^2)),
 Poly(x^2 + 5x + 5, GF(3^2)),
 Poly(x^2 + 6x + 6, GF(3^2)),
 Poly(x^2 + 6x + 7, GF(3^2)),
 Poly(x^2 + 7x + 3, GF(3^2)),
 Poly(x^2 + 7x + 5, GF(3^2)),
 Poly(x^2 + 8x + 5, GF(3^2)),
 Poly(x^2 + 8x + 6, GF(3^2))]
```

galois.conway_poly

galois.**conway_poly**(*characteristic*, *degree*)

Returns the Conway polynomial $C_{p,m}(x)$ over $\mathrm{GF}(p)$ with degree m.

Parameters

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

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

Return type *galois.Poly*

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

Notes

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

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

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

Examples

Notice *galois.primitive_poly()* returns the lexicographically-minimal primitive polynomial, where *galois.conway_poly()* returns the lexicographically-minimal primitive polynomial that is *consistent* with smaller Conway polynomials, which is not *necessarily* the same.

```
In [1]: galois.primitive_poly(2, 4)
Out[1]: Poly(x^4 + x + 1, GF(2))

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

```
In [3]: galois.primitive_poly(7, 10)
Out[3]: Poly(x^10 + 5x^2 + x + 5, GF(7))

In [4]: galois.conway_poly(7, 10)
Out[4]: Poly(x^10 + x^6 + x^5 + 4x^4 + x^3 + 2x^2 + 3x + 3, GF(7))
```

galois.matlab_primitive_poly

`galois.matlab_primitive_poly`(*characteristic*, *degree*)

Returns Matlab's default primitive polynomial $f(x)$ over $\mathrm{GF}(p)$ with degree m.

Parameters

- **characteristic** (*int*) – The prime characteristic p of the field $\mathrm{GF}(p)$ that the polynomial is over.
- **degree** (*int*) – The degree m of the desired primitive polynomial.

Returns Matlab's default degree-m primitive polynomial over $\mathrm{GF}(p)$.

Return type *galois.Poly*

Notes

This function returns the same result as Matlab's `gfprimdf(m, p)`. Matlab uses the primitive polynomial with minimum terms (equivalent to `galois.primitive_poly(p, m, method="min-terms")`) as the default… *mostly*. There are three notable exceptions:

1. $\mathrm{GF}(2^7)$ uses $x^7 + x^3 + 1$, not $x^7 + x + 1$.
2. $\mathrm{GF}(2^{14})$ uses $x^{14} + x^{10} + x^6 + x + 1$, not $x^{14} + x^5 + x^3 + x + 1$.
3. $\mathrm{GF}(2^{16})$ uses $x^{16} + x^{12} + x^3 + x + 1$, not $x^{16} + x^5 + x^3 + x^2 + 1$.

References

- S. Lin and D. Costello. Error Control Coding. Table 2.7.

> **Warning:** This has been tested for all the $\mathrm{GF}(2^m)$ fields for $2 \leq m \leq 16$ (Matlab doesn't support larger than 16). And it has been spot-checked for $\mathrm{GF}(p^m)$. There may exist other exceptions. Please submit a GitHub issue if you discover one.

Examples

```
In [1]: galois.primitive_poly(2, 6)
Out[1]: Poly(x^6 + x + 1, GF(2))

In [2]: galois.matlab_primitive_poly(2, 6)
Out[2]: Poly(x^6 + x + 1, GF(2))
```

```
In [3]: galois.primitive_poly(2, 7)
Out[3]: Poly(x^7 + x + 1, GF(2))

In [4]: galois.matlab_primitive_poly(2, 7)
Out[4]: Poly(x^7 + x^3 + 1, GF(2))
```

galois.is_primitive

`galois.is_primitive(poly)`

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

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

Parameters **poly** (`galois.Poly`) – A degree-m polynomial $f(x)$ over $\mathrm{GF}(q)$.

Returns `True` if the polynomial is primitive.

Return type bool

References

- Algorithm 4.77 from https://cacr.uwaterloo.ca/hac/about/chap4.pdf

Examples

All Conway polynomials are primitive.

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

In [2]: galois.is_primitive(f)
Out[2]: True
```

(continues on next page)

(continued from previous page)

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

In [4]: galois.is_primitive(f)
Out[4]: True
```

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

```
In [5]: f = galois.Poly.Degrees([8,4,3,1,0]); f
Out[5]: Poly(x^8 + x^4 + x^3 + x + 1, GF(2))

In [6]: galois.is_primitive(f)
Out[6]: False
```

Primitive elements

primitive_element(irreducible_poly[, start, ...])	Finds the smallest primitive element $g(x)$ of the Galois field $\mathrm{GF}(p^m)$ with degree-m irreducible polynomial $f(x)$ over $\mathrm{GF}(p)$.
primitive_elements(irreducible_poly[, ...])	Finds all primitive elements $g(x)$ of the Galois field $\mathrm{GF}(p^m)$ with degree-m irreducible polynomial $f(x)$ over $\mathrm{GF}(p)$.
is_primitive_element(element, irreducible_poly)	Determines if $g(x)$ is a primitive element of the Galois field $\mathrm{GF}(p^m)$ with degree-m irreducible polynomial $f(x)$ over $\mathrm{GF}(p)$.

galois.primitive_element

galois.**primitive_element**(*irreducible_poly*, *start=None*, *stop=None*, *reverse=False*)

Finds the smallest primitive element $g(x)$ of the Galois field $\mathrm{GF}(p^m)$ with degree-m irreducible polynomial $f(x)$ over $\mathrm{GF}(p)$.

Parameters

- **irreducible_poly** (galois.Poly) – The degree-m irreducible polynomial $f(x)$ over $\mathrm{GF}(p)$ that defines the extension field $\mathrm{GF}(p^m)$.
- **start** (*int, optional*) – Starting value (inclusive, integer representation of the polynomial) in the search for a primitive element $g(x)$ of $\mathrm{GF}(p^m)$. The default is `None` which represents p, which corresponds to $g(x) = x$ over $\mathrm{GF}(p)$.
- **stop** (*int, optional*) – Stopping value (exclusive, integer representation of the polynomial) in the search for a primitive element $g(x)$ of $\mathrm{GF}(p^m)$. The default is `None` which represents p^m, which corresponds to $g(x) = x^m$ over $\mathrm{GF}(p)$.
- **reverse** (*bool, optional*) – Search for a primitive element in reverse order, i.e. find the largest primitive element first. Default is `False`.

Returns A primitive element of $\mathrm{GF}(p^m)$ with irreducible polynomial $f(x)$. The primitive element $g(x)$ is a polynomial over $\mathrm{GF}(p)$ with degree less than m.

Return type *galois.Poly*

Examples

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

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

In [3]: galois.is_irreducible(f)
Out[3]: True

In [4]: galois.is_primitive(f)
Out[4]: True

In [5]: galois.primitive_element(f)
Out[5]: Poly(x, GF(3))
```

```
In [6]: GF = galois.GF(3)

In [7]: f = galois.Poly([1,0,1], field=GF); f
Out[7]: Poly(x^2 + 1, GF(3))

In [8]: galois.is_irreducible(f)
Out[8]: True

In [9]: galois.is_primitive(f)
Out[9]: False

In [10]: galois.primitive_element(f)
Out[10]: Poly(x + 1, GF(3))
```

galois.primitive_elements

galois.**primitive_elements**(*irreducible_poly*, *start=None*, *stop=None*, *reverse=False*)

Finds all primitive elements $g(x)$ of the Galois field $\mathrm{GF}(p^m)$ with degree-m irreducible polynomial $f(x)$ over $\mathrm{GF}(p)$.

The number of primitive elements of $\mathrm{GF}(p^m)$ is $\phi(p^m - 1)$, where $\phi(n)$ is the Euler totient function. See :obj:galois.euler_phi`.

Parameters

- **irreducible_poly** (`galois.Poly`) – The degree-m irreducible polynomial $f(x)$ over $\mathrm{GF}(p)$ that defines the extension field $\mathrm{GF}(p^m)$.
- **start** (*int, optional*) – Starting value (inclusive, integer representation of the polynomial) in the search for primitive elements $g(x)$ of $\mathrm{GF}(p^m)$. The default is `None` which represents p, which corresponds to $g(x) = x$ over $\mathrm{GF}(p)$.
- **stop** (*int, optional*) – Stopping value (exclusive, integer representation of the polynomial) in the search for primitive elements $g(x)$ of $\mathrm{GF}(p^m)$. The default is `None` which represents p^m, which corresponds to $g(x) = x^m$ over $\mathrm{GF}(p)$.
- **reverse** (*bool, optional*) – Search for primitive elements in reverse order, i.e. largest to smallest. Default is `False`.

Returns List of all primitive elements of $\mathrm{GF}(p^m)$ with irreducible polynomial $f(x)$. Each primitive element $g(x)$ is a polynomial over $\mathrm{GF}(p)$ with degree less than m.

Return type list

Examples

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

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

In [3]: galois.is_irreducible(f)
Out[3]: True

In [4]: galois.is_primitive(f)
Out[4]: True

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

In [6]: len(g) == galois.euler_phi(3**2 - 1)
Out[6]: True
```

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

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

In [9]: galois.is_irreducible(f)
Out[9]: True

In [10]: galois.is_primitive(f)
Out[10]: False

In [11]: g = galois.primitive_elements(f); g
Out[11]:
[Poly(x + 1, GF(3)),
 Poly(x + 2, GF(3)),
 Poly(2x + 1, GF(3)),
 Poly(2x + 2, GF(3))]

In [12]: len(g) == galois.euler_phi(3**2 - 1)
Out[12]: True
```

galois.is_primitive_element

`galois.is_primitive_element(element, irreducible_poly)`

Determines if $g(x)$ is a primitive element of the Galois field $\mathrm{GF}(p^m)$ with degree-m irreducible polynomial $f(x)$ over $\mathrm{GF}(p)$.

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

Notes

The number of primitive elements of $\mathrm{GF}(p^m)$ is $\phi(p^m-1)$, where $\phi(n)$ is the Euler totient function, see `galois.euler_phi()`.

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
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
```

(continues on next page)

(continued from previous page)

```
Out[11]: Poly(x, GF(3))

In [12]: galois.is_primitive_element(g, f)
Out[12]: False
```

Minimal polynomials

minimal_poly(element)	Computes the minimal polynomial $m_e(x) \in \mathrm{GF}(p)[x]$ of a Galois field element $e \in \mathrm{GF}(p^m)$.

galois.minimal_poly

galois.**minimal_poly**(*element*)

Computes the minimal polynomial $m_e(x) \in \mathrm{GF}(p)[x]$ of a Galois field element $e \in \mathrm{GF}(p^m)$.

The *minimal polynomial* of a Galois field element $e \in \mathrm{GF}(p^m)$ is the polynomial of minimal degree over $\mathrm{GF}(p)$ for which e is a root when evaluated in $\mathrm{GF}(p^m)$. Namely, $m_e(x) \in \mathrm{GF}(p)[x] \in \mathrm{GF}(p^m)[x]$ and $m_e(e) = 0$ over $\mathrm{GF}(p^m)$.

Parameters **element** (galois.FieldArray) – Any element e of the Galois field $\mathrm{GF}(p^m)$. This must be a 0-D array.

Returns The minimal polynomial $m_e(x)$ over $\mathrm{GF}(p)$ of the element e.

Return type *galois.Poly*

Examples

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

In [2]: e = GF.primitive_element; e
Out[2]: GF(2, order=2^4)

In [3]: m_e = galois.minimal_poly(e); m_e
Out[3]: Poly(x^4 + x + 1, GF(2))

# Evaluate m_e(e) in GF(2^4)
In [4]: m_e(e, field=GF)
Out[4]: GF(0, order=2^4)
```

For a given element e, the minimal polynomials of e and all its conjugates are the same.

```
# The conjugates of e
In [5]: conjugates = np.unique(e**(2**np.arange(0, 4))); conjugates
Out[5]: GF([2, 3, 4, 5], order=2^4)

In [6]: for conjugate in conjugates:
   ...:     print(galois.minimal_poly(conjugate))
   ...:
Poly(x^4 + x + 1, GF(2))
```

(continues on next page)

(continued from previous page)

```
Poly(x^4 + x + 1, GF(2))
Poly(x^4 + x + 1, GF(2))
Poly(x^4 + x + 1, GF(2))
```

Not all elements of $\mathrm{GF}(2^4)$ have minimal polynomials with degree-4.

```
In [7]: e = GF.primitive_element**5; e
Out[7]: GF(6, order=2^4)

# The conjugates of e
In [8]: conjugates = np.unique(e**(2**np.arange(0, 4))); conjugates
Out[8]: GF([6, 7], order=2^4)

In [9]: for conjugate in conjugates:
   ...:     print(galois.minimal_poly(conjugate))
   ...:
Poly(x^2 + x + 1, GF(2))
Poly(x^2 + x + 1, GF(2))
```

In prime fields, the minimal polynomial of e is simply $m_e(x) = x - e$.

```
In [10]: GF = galois.GF(7)

In [11]: e = GF(3); e
Out[11]: GF(3, order=7)

In [12]: m_e = galois.minimal_poly(e); m_e
Out[12]: Poly(x + 4, GF(7))

In [13]: m_e(e)
Out[13]: GF(0, order=7)
```

7.2 Polynomials over Galois Fields

This section contains classes and functions for creating polynomials over Galois fields.

7.2.1 Polynomial classes

Poly(coeffs[, field, order])	Create a polynomial $f(x)$ over $\mathrm{GF}(p^m)$.

galois.Poly

class galois.**Poly**(*coeffs*, *field=None*, *order='desc'*)

Create a polynomial $f(x)$ over $\mathrm{GF}(p^m)$.

The polynomial $f(x) = a_d x^d + a_{d-1} x^{d-1} + \dots + a_1 x + a_0$ has coefficients $\{a_d, a_{d-1}, \dots, a_1, a_0\}$ in $\mathrm{GF}(p^m)$.

Parameters

- **coeffs** (*tuple, list, numpy.ndarray,* galois.FieldArray) – The polynomial coefficients $\{a_d, a_{d-1}, \dots, a_1, a_0\}$ with type *galois.FieldArray*. Alternatively, an iterable tuple, list, or numpy.ndarray may be provided and the Galois field domain is taken from the field keyword argument.
- **field** (galois.FieldClass, *optional*) – The Galois field $\mathrm{GF}(p^m)$ the polynomial is over.
  - None (default): If the coefficients are a *galois.FieldArray*, they won't be modified. If the coefficients are not explicitly in a Galois field, they are assumed to be from $\mathrm{GF}(2)$ and are converted using galois.GF2(coeffs).
  - *galois.FieldClass*: The coefficients are explicitly converted to this Galois field field(coeffs).
- **order** (*str, optional*) – The interpretation of the coefficient degrees.
  - "desc" (default): The first element of coeffs is the highest degree coefficient, i.e. $\{a_d, a_{d-1}, \dots, a_1, a_0\}$.
  - "asc": The first element of coeffs is the lowest degree coefficient, i.e. $\{a_0, a_1, \dots, a_{d-1}, a_d\}$.

Returns The polynomial $f(x)$.

Return type *galois.Poly*

Examples

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

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

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

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

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

In [4]: galois.Poly([124,0,223,0,0,15], field=GF)
Out[4]: Poly(124x^5 + 223x^3 + 15, GF(2^8))

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

Polynomial arithmetic using binary operators.

```
In [6]: a = galois.Poly([117,0,63,37], field=GF); a
Out[6]: Poly(117x^3 + 63x + 37, GF(2^8))

In [7]: b = galois.Poly([224,0,21], field=GF); b
Out[7]: Poly(224x^2 + 21, GF(2^8))

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

In [9]: a - b
Out[9]: Poly(117x^3 + 224x^2 + 63x + 48, GF(2^8))

# Compute the quotient of the polynomial division
In [10]: a / b
Out[10]: Poly(202x, GF(2^8))

# True division and floor division are equivalent
In [11]: a / b == a // b
Out[11]: True

# Compute the remainder of the polynomial division
In [12]: a % b
Out[12]: Poly(198x + 37, GF(2^8))

# Compute both the quotient and remainder in one pass
In [13]: divmod(a, b)
Out[13]: (Poly(202x, GF(2^8)), Poly(198x + 37, GF(2^8)))
```

Constructors

__init__()	
Degrees(degrees[, coeffs, field])	Constructs a polynomial over $\mathrm{GF}(p^m)$ from its non-zero degrees.
Identity([field])	Constructs the 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 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 over $\mathrm{GF}(p^m)$ from its roots.
String(string[, field])	Constructs a polynomial over $\mathrm{GF}(p^m)$ from its string representation.
Zero([field])	Constructs the 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 $f(x)$.
reverse()	Returns the d-th reversal $x^d f(\frac{1}{x})$ of the polynomial $f(x)$ with degree d.
roots([multiplicity])	Calculates the roots r of the polynomial $f(x)$, such that $f(r) = 0$.

Special Methods

__add__(other)	Adds two polynomials.
__divmod__(other)	Divides two polynomials and returns the quotient and remainder.
__floordiv__(other)	Divides two polynomials and returns the quotient.
__mod__(other)	Divides two polynomials and returns the remainder.
__mul__(other)	Multiplies two polynomials.
__pow__(other)	Exponentiates the polynomial to an integer power.
__sub__(other)	Subtracts two polynomials.
__truediv__(other)	Divides two polynomials and returns the quotient.

Attributes

coeffs	The coefficients of the polynomial in degree-descending order.
degree	The degree of the polynomial, i.e. the highest degree with non-zero coefficient.
degrees	An array of the polynomial degrees in degree-descending order.
field	The Galois field array class to which the coefficients belong.
integer	The integer representation of the polynomial.
nonzero_coeffs	The non-zero coefficients of the polynomial in degree-descending order.
nonzero_degrees	An array of the polynomial degrees that have non-zero coefficients, in degree-descending order.
string	The string representation of the polynomial, without specifying the Galois field.

classmethod Degrees(*degrees*, *coeffs=None*, *field=None*)

Constructs a polynomial over $\mathrm{GF}(p^m)$ from its non-zero degrees.

Parameters

- **degrees** (*tuple*, *list*, *numpy.ndarray*) – The polynomial degrees with non-zero coefficients.
- **coeffs** (*tuple*, *list*, *numpy.ndarray*, `galois.FieldArray`, *optional*) – The corresponding non-zero polynomial coefficients with type *galois.FieldArray*. Alternatively, an iterable `tuple`, `list`, or `numpy.ndarray` may be provided and the Galois field

domain is taken from the `field` keyword argument. The default is `None` which corresponds to all ones.

- **`field`** (`galois.FieldClass, optional`) – The Galois field $\mathrm{GF}(p^m)$ the polynomial is over.
  - `None` (default): If the coefficients are a *`galois.FieldArray`*, they won't be modified. If the coefficients are not explicitly in a Galois field, they are assumed to be from $\mathrm{GF}(2)$ and are converted using `galois.GF2(coeffs)`.
  - *`galois.FieldClass`*: The coefficients are explicitly converted to this Galois field `field(coeffs)`.

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 polynomial $f(x) = x$ over $\mathrm{GF}(p^m)$.

Parameters **`field`** (`galois.FieldClass, optional`) – The Galois field $\mathrm{GF}(p^m)$ the polynomial is over. The default is *`galois.GF2`*.

Returns The polynomial $f(x) = x$.

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.

Parameters

- **integer** (*int*) – The integer representation of the polynomial $f(x)$.
- **field** (galois.FieldClass, *optional*) – The Galois field $\mathrm{GF}(p^m)$ the polynomial is over. The default is *galois.GF2*.

Returns The polynomial $f(x)$.

Return type *galois.Poly*

Notes

The integer value i represents the polynomial $f(x) = a_d x^d + a_{d-1} x^{d-1} + \dots + a_1 x + a_0$ over the field $\mathrm{GF}(p^m)$ if $i = a_d(p^m)^d + a_{d-1}(p^m)^{d-1} + \dots + a_1(p^m) + a_0$ using integer arithmetic, not finite field arithmetic.

Said differently, if the polynomial coefficients $\{a_d, a_{d-1}, \dots, a_1, a_0\}$ are considered as the "digits" of a radix-p^m value, the polynomial's integer representation is the decimal value (radix-10).

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 polynomial $f(x) = 1$ over $\mathrm{GF}(p^m)$.

Parameters **field** (galois.FieldClass, *optional*) – The Galois field $\mathrm{GF}(p^m)$ the polynomial is over. The default is *galois.GF2*.

Returns The polynomial $f(x) = 1$.

Return type *galois.Poly*

Examples

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

```
In [1]: galois.Poly.One()
Out[1]: Poly(1, GF(2))
```

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

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

In [3]: galois.Poly.One(field=GF)
Out[3]: Poly(1, GF(2^8))
```

classmethod Random(*degree*, *field=<class 'numpy.ndarray over GF(2)'>*)

Constructs a random polynomial over $\mathrm{GF}(p^m)$ with degree d.

Parameters

- **degree** (*int*) – The degree of the polynomial.
- **field** (`galois.FieldClass`, *optional*) – The Galois field $\mathrm{GF}(p^m)$ the polynomial is over. The default is *galois.GF2*.

Returns The polynomial $f(x)$.

Return type *galois.Poly*

Examples

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

```
In [1]: galois.Poly.Random(5)
Out[1]: Poly(x^5 + x^3, GF(2))
```

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

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

In [3]: galois.Poly.Random(5, field=GF)
Out[3]: Poly(151x^5 + 70x^4 + 89x^3 + 180x^2 + 92x + 137, GF(2^8))
```

classmethod Roots(*roots*, *multiplicities=None*, *field=None*)

Constructs a monic polynomial over $\mathrm{GF}(p^m)$ from its roots.

Parameters

- **roots** (*tuple*, *list*, *numpy.ndarray*, `galois.FieldArray`) – The roots of the desired polynomial with type *galois.FieldArray*. Alternatively, an iterable `tuple`, `list`, or `numpy.ndarray` may be provided and the Galois field domain is taken from the `field` keyword argument.
- **multiplicities** (*tuple*, *list*, *numpy.ndarray*, *optional*) – The corresponding root multiplicities. The default is `None` which corresponds to all ones, i.e. `[1,]*len(roots)`.
- **field** (`galois.FieldClass`, *optional*) – The Galois field $\mathrm{GF}(p^m)$ the polynomial is over.
  - `None` (default): If the roots are a *galois.FieldArray*, they won't be modified. If the roots are not explicitly in a Galois field, they are assumed to be from $\mathrm{GF}(2)$ and are converted using `galois.GF2(roots)`.
  - *galois.FieldClass*: The roots are explicitly converted to this Galois field `field(roots)`.

Returns The polynomial $f(x)$.

Return type *galois.Poly*

Notes

The polynomial $f(x)$ with k roots $\{r_1, r_2, \dots, r_k\}$ with multiplicities $\{m_1, m_2, \dots, m_k\}$ is

$$f(x) = (x - r_1)^{m_1}(x - r_2)^{m_2} \dots (x - r_k)^{m_k}$$
$$f(x) = a_d x^d + a_{d-1}x^{d-1} + \dots + a_1 x + a_0$$

with degree $d = \sum_{i=1}^{k} m_i$.

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))

# Evaluate the polynomial at its roots
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 with specific multiplicities.

```
In [4]: GF = galois.GF(2**8)

In [5]: roots = [121, 198, 225]

In [6]: multiplicities = [1, 2, 1]

In [7]: p = galois.Poly.Roots(roots, multiplicities=multiplicities, field=GF); p
Out[7]: Poly(x^4 + 152x^3 + 85x^2 + 223x + 147, GF(2^8))

# Evaluate the polynomial at its roots
In [8]: p(roots)
Out[8]: GF([0, 0, 0], order=2^8)
```

classmethod String(*string*, *field=<class 'numpy.ndarray over GF(2)'>*)

Constructs a polynomial over $\mathrm{GF}(p^m)$ from its string representation.

Parameters

- **string** (*str*) – The string representation of the polynomial $f(x)$.
- **field** (`galois.FieldClass`, *optional*) – The Galois field $\mathrm{GF}(p^m)$ the polynomial is over. The default is *galois.GF2*.

Returns The polynomial $f(x)$.

Return type *galois.Poly*

Notes

The string parsing rules include:

- Either `^` or `**` may be used for indicating the polynomial degrees. For example, `"13x^3 + 117"` or `"13x**3 + 117"`.
- Multiplication operators `*` may be used between coefficients and the polynomial indeterminate `x`, but are not required. For example, `"13x^3 + 117"` or `"13*x^3 + 117"`.
- Polynomial coefficients of 1 may be specified or omitted. For example, `"x^3 + 117"` or `"1*x^3 + 117"`.
- The polynomial indeterminate can be any single character, but must be consistent. For example, `"13x^3 + 117"` or `"13y^3 + 117"`.
- Spaces are not required between terms. For example, `"13x^3 + 117"` or `"13x^3+117"`.
- Any combination of the above rules is acceptable.

Examples

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

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

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

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

In [3]: galois.Poly.String("13x^3 + 117", field=GF)
Out[3]: Poly(13x^3 + 117, GF(2^8))
```

classmethod Zero(*field=<class 'numpy.ndarray over GF(2)'>*)

Constructs the polynomial $f(x) = 0$ over $\mathrm{GF}(p^m)$.

Parameters **field** (`galois.FieldClass`, *optional*) – The Galois field $\mathrm{GF}(p^m)$ the polynomial is over. The default is *galois.GF2*.

Returns The polynomial $f(x) = 0$.

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))
```

__add__(*other*)

Adds two polynomials.

Parameters **other** (galois.Poly) – The polynomial $b(x)$.

Returns The polynomial $c(x) = a(x) + b(x)$.

Return type *galois.Poly*

Examples

```
In [1]: a = galois.Poly.Random(5); a
Out[1]: Poly(x^5 + x^2, GF(2))

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

In [3]: a + b
Out[3]: Poly(x^5 + x^3 + x + 1, GF(2))
```

__divmod__(*other*)

Divides two polynomials and returns the quotient and remainder.

Parameters **other** (galois.Poly) – The polynomial $b(x)$.

Returns

- *galois.Poly* – The quotient polynomial $q(x)$ such that $a(x) = b(x)q(x) + r(x)$.
- *galois.Poly* – The remainder polynomial $r(x)$ such that $a(x) = b(x)q(x) + r(x)$.

Examples

```
In [1]: a = galois.Poly.Random(5); a
Out[1]: Poly(x^5 + x + 1, GF(2))

In [2]: b = galois.Poly.Random(3); b
Out[2]: Poly(x^3 + x^2 + x, GF(2))

In [3]: q, r = divmod(a, b)

In [4]: q, r
Out[4]: (Poly(x^2 + x, GF(2)), Poly(x^2 + x + 1, GF(2)))

In [5]: b*q + r
Out[5]: Poly(x^5 + x + 1, GF(2))
```

__floordiv__(*other*)

Divides two polynomials and returns the quotient.

True division and floor division are equivalent.

Parameters **other** (galois.Poly) – The polynomial $b(x)$.

Returns The quotient polynomial $q(x)$ such that $a(x) = b(x)q(x) + r(x)$.

Return type *galois.Poly*

Examples

```
In [1]: a = galois.Poly.Random(5); a
Out[1]: Poly(x^5 + x^4 + x^3 + x + 1, GF(2))

In [2]: b = galois.Poly.Random(3); b
Out[2]: Poly(x^3 + x^2 + x, GF(2))

In [3]: divmod(a, b)
Out[3]: (Poly(x^2, GF(2)), Poly(x + 1, GF(2)))

In [4]: a // b
Out[4]: Poly(x^2, GF(2))
```

`__init__`()

`__mod__`(*other*)

Divides two polynomials and returns the remainder.

Parameters **`other`** (`galois.Poly`) – The polynomial $b(x)$.

Returns The remainder polynomial $r(x)$ such that $a(x) = b(x)q(x) + r(x)$.

Return type *galois.Poly*

Examples

```
In [1]: a = galois.Poly.Random(5); a
Out[1]: Poly(x^5 + x^3 + x^2 + x + 1, GF(2))

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

In [3]: divmod(a, b)
Out[3]: (Poly(x^2, GF(2)), Poly(x + 1, GF(2)))

In [4]: a % b
Out[4]: Poly(x + 1, GF(2))
```

`__mul__`(*other*)

Multiplies two polynomials.

Parameters **`other`** (`galois.Poly`) – The polynomial $b(x)$.

Returns The polynomial $c(x) = a(x)b(x)$.

Return type *galois.Poly*

Examples

```
In [1]: a = galois.Poly.Random(5); a
Out[1]: Poly(x^5 + x^4 + x^2 + x + 1, GF(2))

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

In [3]: a * b
Out[3]: Poly(x^8 + x^7 + x^3 + x^2 + x + 1, GF(2))
```

__pow__(*other*)

Exponentiates the polynomial to an integer power.

Parameters **other** (*int*) – The non-negative integer exponent.

Returns The polynomial $a(x) **b$.

Return type *galois.Poly*

Examples

```
In [1]: a = galois.Poly.Random(5); a
Out[1]: Poly(x^5 + x^3 + x^2 + x + 1, GF(2))

In [2]: a**3
Out[2]: Poly(x^15 + x^13 + x^12 + x^10 + x^8 + x^7 + x^5 + x + 1, GF(2))

In [3]: a * a * a
Out[3]: Poly(x^15 + x^13 + x^12 + x^10 + x^8 + x^7 + x^5 + x + 1, GF(2))
```

__sub__(*other*)

Subtracts two polynomials.

Parameters **other** (`galois.Poly`) – The polynomial $b(x)$.

Returns The polynomial $c(x) = a(x) - b(x)$.

Return type *galois.Poly*

Examples

```
In [1]: a = galois.Poly.Random(5); a
Out[1]: Poly(x^5 + x^4 + x^3 + 1, GF(2))

In [2]: b = galois.Poly.Random(3); b
Out[2]: Poly(x^3, GF(2))

In [3]: a - b
Out[3]: Poly(x^5 + x^4 + 1, GF(2))
```

__truediv__(*other*)

Divides two polynomials and returns the quotient.

True division and floor division are equivalent.

Parameters **other** (galois.Poly) – The polynomial $b(x)$.

Returns The quotient polynomial $q(x)$ such that $a(x) = b(x)q(x) + r(x)$.

Return type *galois.Poly*

Examples

```
In [1]: a = galois.Poly.Random(5); a
Out[1]: Poly(x^5 + x^2, GF(2))

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

In [3]: divmod(a, b)
Out[3]: (Poly(x^2 + x + 1, GF(2)), Poly(x^2 + x + 1, GF(2)))

In [4]: a / b
Out[4]: Poly(x^2 + x + 1, GF(2))
```

derivative(*k=1*)

Computes the k-th formal derivative $\frac{d^k}{dx^k} f(x)$ of the polynomial $f(x)$.

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*

Notes

For the polynomial

$$f(x) = a_d x^d + a_{d-1} x^{d-1} + \dots + a_1 x + a_0$$

the first formal derivative is defined as

$$f'(x) = (d) \cdot a_d x^{d-1} + (d-1) \cdot a_{d-1} x^{d-2} + \dots + (2) \cdot a_2 x + a_1$$

where $\cdot$ represents scalar multiplication (repeated addition), not finite field multiplication. For example, $3 \cdot a = a + a + a$.

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, GF(2))

In [2]: p.derivative()
Out[2]: Poly(x^6, 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(2x^11 + x^10 + x^9 + 4x^7 + 6x^6 + x^5 + 3x^3 + 4x^2 + 5x, GF(7))

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

In [7]: p.derivative(2)
Out[7]: Poly(3x^9 + 6x^8 + 2x^7 + 5x^4 + 6x^3 + 4x + 1, GF(7))

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

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(85x^7 + 123x^6 + 167x^5 + 195x^4 + 118x^3 + 60x^2 + 66x + 82,␣
↪GF(2^8))

In [12]: p.derivative()
Out[12]: Poly(85x^6 + 167x^4 + 118x^2 + 66, GF(2^8))

# k derivatives of a polynomial where k is the Galois field's characteristic␣
↪will always result in 0
In [13]: p.derivative(2)
Out[13]: Poly(0, GF(2^8))
```

reverse()

Returns the d-th reversal $x^d f(\frac{1}{x})$ of the polynomial $f(x)$ with degree d.

Returns The n-th reversal $x^n f(\frac{1}{x})$.

Return type *galois.Poly*

Notes

For a polynomial $f(x) = a_d x^d + a_{d-1} x^{d-1} + \dots + a_1 x + a_0$ with degree d, the d-th reversal is equivalent to reversing the coefficients.

$$\text{rev}_d f(x) = x^d f(x^{-1}) = a_0 x^d + a_1 x^{d-1} + \dots + a_{d-1} x + a_d$$

Examples

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

In [2]: f = galois.Poly([5, 0, 3, 4], field=GF); f
Out[2]: Poly(5x^3 + 3x + 4, GF(7))

In [3]: f.reverse()
Out[3]: Poly(4x^3 + 3x^2 + 5, GF(7))
```

roots(*multiplicity=False*)

Calculates the roots r of the polynomial $f(x)$, such that $f(r) = 0$.

Parameters multiplicity (*bool, optional*) – Optionally return the multiplicity of each root. The default is `False` which only returns the unique roots.

Returns

- *galois.FieldArray* – Galois field array of roots of $f(x)$. The roots are ordered in increasing order.
- *np.ndarray* – The multiplicity of each root, only returned if `multiplicity=True`.

Notes

This implementation uses Chien's search to find the roots $\{r_1, r_2, \dots, r_k\}$ of the degree-d polynomial

$$f(x) = a_d x^d + a_{d-1} x^{d-1} + \dots + a_1 x + a_0,$$

where $k \le d$. Then, $f(x)$ can be factored as

$$f(x) = (x - r_1)^{m_1} (x - r_2)^{m_2} \dots (x - r_k)^{m_k},$$

where m_i is the multiplicity of root r_i and $d = \sum_{i=1}^{k} m_i$.

The Galois field elements can be represented as $\mathrm{GF}(p^m) = \{0, 1, \alpha, \alpha^2, \dots, \alpha^{p^m-2}\}$, 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 $f(\alpha^i)$, where α^i is a root of $f(x)$ if $f(\alpha^i) = 0$.

$$f(\alpha^i) = a_d(\alpha^i)^d + a_{d-1}(\alpha^i)^{d-1} + \dots + a_1(\alpha^i) + a_0$$

$$f(\alpha^i) \triangleq \lambda_{i,d} + \lambda_{i,d-1} + \dots + \lambda_{i,1} + \lambda_{i,0}$$

$$f(\alpha^i) = \sum_{j=0}^{d} \lambda_{i,j}$$

The next power of α can be easily calculated from the previous calculation.

$$
\begin{aligned}
f(\alpha^{i+1}) &= a_d(\alpha^{i+1})^d + a_{d-1}(\alpha^{i+1})^{d-1} + \cdots + a_1(\alpha^{i+1}) + a_0 \\
f(\alpha^{i+1}) &= a_d(\alpha^i)^d\alpha^d + a_{d-1}(\alpha^i)^{d-1}\alpha^{d-1} + \cdots + a_1(\alpha^i)\alpha + a_0 \\
f(\alpha^{i+1}) &= \lambda_{i,d}\alpha^d + \lambda_{i,d-1}\alpha^{d-1} + \cdots + \lambda_{i,1}\alpha + \lambda_{i,0} \\
f(\alpha^{i+1}) &= \sum_{j=0}^{d} \lambda_{i,j}\alpha^j
\end{aligned}
$$

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 *degrees* are paired with *coeffs*.

Examples

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

In [2]: p = galois.Poly([3, 0, 5, 2], field=GF); p
Out[2]: Poly(3x^3 + 5x + 2, GF(7))

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

Type *galois.FieldArray*

property degree

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

Examples

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

In [2]: p = galois.Poly([3, 0, 5, 2], field=GF); p
Out[2]: Poly(3x^3 + 5x + 2, GF(7))

In [3]: p.degree
Out[3]: 3
```

Type int

property degrees

An array of the polynomial degrees in degree-descending order. The entries of `degrees` are paired with `coeffs`.

Examples

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

In [2]: p = galois.Poly([3, 0, 5, 2], field=GF); p
Out[2]: Poly(3x^3 + 5x + 2, GF(7))

In [3]: p.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 + 1, GF(2))

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

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

In [4]: b = galois.Poly.Random(5, field=GF); b
Out[4]: Poly(96x^5 + 6x^4 + 141x^3 + 179x^2 + 30x + 129, GF(2^8))

In [5]: b.field
Out[5]: <class 'numpy.ndarray over GF(2^8)'>
```

Type *galois.FieldClass*

property integer

The integer representation of the polynomial. For the polynomial $f(x) = a_d x^d + a_{d-1} x^{d-1} + \dots + a_1 x + a_0$ over the field $\mathrm{GF}(p^m)$, the integer representation is $i = a_d (p^m)^d + a_{d-1} (p^m)^{d-1} + \dots + a_1 (p^m) + a_0$ using integer arithmetic, not finite field arithmetic.

Said differently, if the polynomial coefficients $\{a_d, a_{d-1}, \dots, a_1, a_0\}$ are considered as the "digits" of a radix-p^m value, the polynomial's integer representation is the decimal value (radix-10).

Examples

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

In [2]: p = galois.Poly([3, 0, 5, 2], field=GF); p
Out[2]: Poly(3x^3 + 5x + 2, GF(7))

In [3]: p.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 `nonzero_degrees` are paired with `nonzero_coeffs`.

Examples

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

In [2]: p = galois.Poly([3, 0, 5, 2], field=GF); p
Out[2]: Poly(3x^3 + 5x + 2, GF(7))

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

Type *galois.FieldArray*

property nonzero_degrees

An array of the polynomial degrees that have non-zero coefficients, in degree-descending order. The entries of `nonzero_degrees` are paired with `nonzero_coeffs`.

Examples

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

In [2]: p = galois.Poly([3, 0, 5, 2], field=GF); p
Out[2]: Poly(3x^3 + 5x + 2, GF(7))

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

Type numpy.ndarray

property string

The string representation of the polynomial, without specifying the Galois field.

Examples

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

In [2]: p = galois.Poly([3, 0, 5, 2], field=GF); p
Out[2]: Poly(3x^3 + 5x + 2, GF(7))

In [3]: p.string
Out[3]: '3x^3 + 5x + 2'
```

Type str

7.2.2 Special polynomials

Irreducible polynomials

irreducible_poly(order, degree[, method])	Returns a monic irreducible polynomial $f(x)$ over $\mathrm{GF}(q)$ with degree m.
irreducible_polys(order, degree)	Returns all monic irreducible polynomials $f(x)$ over $\mathrm{GF}(q)$ with degree m.

Primitive polynomials

primitive_poly(order, degree[, method])	Returns a monic primitive polynomial $f(x)$ over $\mathrm{GF}(q)$ with degree m.
primitive_polys(order, degree)	Returns all monic primitive polynomials $f(x)$ over $\mathrm{GF}(q)$ with degree m.
conway_poly(characteristic, degree)	Returns the Conway polynomial $C_{p,m}(x)$ over $\mathrm{GF}(p)$ with degree m.
matlab_primitive_poly(characteristic, degree)	Returns Matlab's default primitive polynomial $f(x)$ over $\mathrm{GF}(p)$ with degree m.

Minimal polynomials

minimal_poly(element)	Computes the minimal polynomial $m_e(x) \in \mathrm{GF}(p)[x]$ of a Galois field element $e \in \mathrm{GF}(p^m)$.

7.2.3 Polynomial functions

Divisibility

gcd(a, b)	Finds the greatest common divisor of a and b.
egcd(a, b)	Finds the multiplicands of a and b such that $as + bt = \mathrm{gcd}(a, b)$.
lcm(*values)	Computes the least common multiple of the arguments.
prod(*values)	Computes the product of the arguments.
are_coprime(*values)	Determines if the arguments are pairwise coprime.

galois.gcd

`galois.gcd`(a, b)

Finds the greatest common divisor of a and b.

Parameters

- **a** (*int*, `galois.Poly`) – The first integer or polynomial argument.
- **b** (*int*, `galois.Poly`) – The second integer or polynomial argument.

Returns Greatest common divisor of a and b.

Return type int, *galois.Poly*

Notes

This function implements the Euclidean Algorithm.

References

- Algorithm 2.104 from https://cacr.uwaterloo.ca/hac/about/chap2.pdf
- Algorithm 2.218 from https://cacr.uwaterloo.ca/hac/about/chap2.pdf

Examples

Compute the GCD of two integers.

```
In [1]: galois.gcd(12, 16)
Out[1]: 4
```

Compute the GCD of two polynomials.

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

In [3]: p1 = galois.irreducible_poly(7, 1); p1
Out[3]: Poly(x, GF(7))

In [4]: p2 = galois.irreducible_poly(7, 2); p2
Out[4]: Poly(x^2 + 1, GF(7))

In [5]: p3 = galois.irreducible_poly(7, 3); p3
Out[5]: Poly(x^3 + 2, GF(7))

In [6]: a = p1**2 * p2; a
Out[6]: Poly(x^4 + x^2, GF(7))

In [7]: b = p1 * p3; b
Out[7]: Poly(x^4 + 2x, GF(7))

In [8]: gcd = galois.gcd(a, b); gcd
Out[8]: Poly(x, GF(7))
```

galois.egcd

`galois.egcd(a, b)`

Finds the multiplicands of a and b such that $as + bt = \gcd(a, b)$.

Parameters

- **a** (*int*, `galois.Poly`) – The first integer or polynomial argument.
- **b** (*int*, `galois.Poly`) – The second integer or polynomial argument.

Returns

- *int, galois.Poly* – Greatest common divisor of a and b.
- *int, galois.Poly* – The multiplicand s of a, such that $as + bt = \gcd(a, b)$.
- *int, galois.Poly* – The multiplicand t of b, such that $as + bt = \gcd(a, b)$.

Notes

This function implements the Extended Euclidean Algorithm.

References

- Algorithm 2.107 from https://cacr.uwaterloo.ca/hac/about/chap2.pdf
- Algorithm 2.221 from https://cacr.uwaterloo.ca/hac/about/chap2.pdf
- T. Moon, "Error Correction Coding", Section 5.2.2: The Euclidean Algorithm and Euclidean Domains, p. 181

Examples

Compute the extended GCD of two integers.

```
In [1]: a, b = 12, 16

In [2]: gcd, s, t = galois.egcd(a, b)

In [3]: gcd, s, t
Out[3]: (4, -1, 1)

In [4]: a*s + b*t == gcd
Out[4]: True
```

Compute the extended GCD of two polynomials.

```
In [5]: GF = galois.GF(7)

In [6]: p1 = galois.irreducible_poly(7, 1); p1
Out[6]: Poly(x, GF(7))

In [7]: p2 = galois.irreducible_poly(7, 2); p2
Out[7]: Poly(x^2 + 1, GF(7))

In [8]: p3 = galois.irreducible_poly(7, 3); p3
Out[8]: Poly(x^3 + 2, GF(7))

In [9]: a = p1**2 * p2; a
Out[9]: Poly(x^4 + x^2, GF(7))

In [10]: b = p1 * p3; b
Out[10]: Poly(x^4 + 2x, GF(7))

In [11]: gcd, s, t = galois.egcd(a, b)
```

(continues on next page)

(continued from previous page)

```
In [12]: gcd, s, t
Out[12]: (Poly(x, GF(7)), Poly(2x^2 + 4x + 1, GF(7)), Poly(5x^2 + 3x + 4, GF(7)))

In [13]: a*s + b*t == gcd
Out[13]: True
```

galois.lcm

`galois.lcm(*values)`

Computes the least common multiple of the arguments.

Parameters `*values` (*int*, `galois.Poly`) – Each argument must be an integer or polynomial.

Returns The least common multiple of the arguments.

Return type int, *galois.Poly*

Examples

Compute the LCM of three integers.

```
In [1]: galois.lcm(2, 4, 14)
Out[1]: 28
```

Compute the LCM of three polynomials.

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

In [3]: p1 = galois.irreducible_poly(7, 1); p1
Out[3]: Poly(x, GF(7))

In [4]: p2 = galois.irreducible_poly(7, 2); p2
Out[4]: Poly(x^2 + 1, GF(7))

In [5]: p3 = galois.irreducible_poly(7, 3); p3
Out[5]: Poly(x^3 + 2, GF(7))

In [6]: a = p1**2 * p2; a
Out[6]: Poly(x^4 + x^2, GF(7))

In [7]: b = p1 * p3; b
Out[7]: Poly(x^4 + 2x, GF(7))

In [8]: c = p2 * p3; c
Out[8]: Poly(x^5 + x^3 + 2x^2 + 2, GF(7))

In [9]: galois.lcm(a, b, c)
Out[9]: Poly(x^7 + x^5 + 2x^4 + 2x^2, GF(7))

In [10]: p1**2 * p2 * p3
Out[10]: Poly(x^7 + x^5 + 2x^4 + 2x^2, GF(7))
```

galois.prod

`galois.prod(*values)`

Computes the product of the arguments.

Parameters ***values** (*int*, galois.Poly) – Each argument must be an integer or polynomial.

Returns The product of the arguments.

Return type int, *galois.Poly*

Examples

Compute the product of three integers.

```
In [1]: galois.prod(2, 4, 14)
Out[1]: 112
```

Compute the product of three polynomials.

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

In [3]: a = galois.Poly.Random(2, field=GF)

In [4]: b = galois.Poly.Random(3, field=GF)

In [5]: c = galois.Poly.Random(4, field=GF)

In [6]: galois.prod(a, b, c)
Out[6]: Poly(4x^9 + 6x^8 + 3x^7 + x^5 + 5x^4 + 6x^3 + x^2 + 6x, GF(7))

In [7]: a * b * c
Out[7]: Poly(4x^9 + 6x^8 + 3x^7 + x^5 + 5x^4 + 6x^3 + x^2 + 6x, GF(7))
```

galois.are_coprime

`galois.are_coprime(*values)`

Determines if the arguments are pairwise coprime.

Parameters ***values** (*int*, galois.Poly) – Each argument must be an integer or polynomial.

Returns `True` if the arguments are pairwise coprime.

Return type bool

Notes

A set of integers or polynomials are pairwise coprime if their LCM is equal to their product.

Examples

Determine if a set of integers are pairwise coprime.

```
In [1]: galois.are_coprime(3, 4, 5)
Out[1]: True

In [2]: galois.are_coprime(3, 7, 9, 11)
Out[2]: False
```

Determine if a set of polynomials are pairwise coprime.

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

In [4]: p1 = galois.irreducible_poly(7, 1); p1
Out[4]: Poly(x, GF(7))

In [5]: p2 = galois.irreducible_poly(7, 2); p2
Out[5]: Poly(x^2 + 1, GF(7))

In [6]: p3 = galois.irreducible_poly(7, 3); p3
Out[6]: Poly(x^3 + 2, GF(7))

In [7]: galois.are_coprime(p1, p2, p3)
Out[7]: True

In [8]: galois.are_coprime(p1*p2, p2, p3)
Out[8]: False
```

Congruences

pow(base, exponent, modulus)	Efficiently performs modular exponentiation.
crt(remainders, moduli)	Solves the simultaneous system of congruences for x.

galois.pow

`galois.pow`(*base*, *exponent*, *modulus*)

Efficiently performs modular exponentiation.

Parameters

- **base** (*int*, `galois.Poly`) – The integer or polynomial base a.
- **exponent** (*int*) – The non-negative integer exponent k.
- **modulus** (*int*, `galois.Poly`) – The integer or polynomial modulus m.

Returns The modular exponentiation $a^k \bmod m$.

Return type int, *galois.Poly*

Notes

This function implements the Square-and-Multiply Algorithm. The algorithm is more efficient than exponentiating first and then reducing modulo m, especially for very large exponents. Instead, this algorithm repeatedly squares a, reducing modulo m at each step.

References

- Algorithm 2.143 from https://cacr.uwaterloo.ca/hac/about/chap2.pdf
- Algorithm 2.227 from https://cacr.uwaterloo.ca/hac/about/chap2.pdf

Examples

Compute the modular exponentiation of an integer.

```
In [1]: galois.pow(3, 100, 7)
Out[1]: 4

In [2]: 3**100 % 7
Out[2]: 4
```

Compute the modular exponentiation of a polynomial.

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

In [4]: a = galois.Poly.Random(3, field=GF)

In [5]: m = galois.Poly.Random(10, field=GF)

In [6]: galois.pow(a, 100, m)
Out[6]: Poly(6x^9 + 3x^8 + 2x^7 + 3x^6 + x^5 + 5x^4 + 5x^3 + 6x^2 + 4, GF(7))

In [7]: a**100 % m
Out[7]: Poly(6x^9 + 3x^8 + 2x^7 + 3x^6 + x^5 + 5x^4 + 5x^3 + 6x^2 + 4, GF(7))
```

galois.crt

`galois.crt`(*remainders*, *moduli*)

Solves the simultaneous system of congruences for x.

Parameters

- **remainders** (*tuple, list*) – The integer or polynomial remainders a_i.
- **moduli** (*tuple, list*) – The integer or polynomial moduli m_i.

Returns The simultaneous solution x to the system of congruences.

Return type int

Notes

This function implements the Chinese Remainder Theorem.

$$\begin{aligned} x &\equiv a_1 \pmod{m_1} \\ x &\equiv a_2 \pmod{m_2} \\ x &\equiv \ldots \\ x &\equiv a_n \pmod{m_n} \end{aligned}$$

References

- Section 14.5 from https://cacr.uwaterloo.ca/hac/about/chap14.pdf

Examples

Solve a system of integer congruences.

```
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
```

Solve a system of polynomial congruences.

```
In [5]: GF = galois.GF(7)

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

In [7]: m = [galois.irreducible_poly(7, 2), galois.irreducible_poly(7, 3), galois.
↪irreducible_poly(7, 4)]; m
Out[7]: [Poly(x^2 + 1, GF(7)), Poly(x^3 + 2, GF(7)), Poly(x^4 + x + 1, GF(7))]

In [8]: a = [x_truth % mi for mi in m]; a
Out[8]:
[Poly(5x + 3, GF(7)),
 Poly(2x^2 + 4x + 6, GF(7)),
 Poly(2x^3 + x^2 + x + 5, GF(7))]

In [9]: x = galois.crt(a, m); x
Out[9]: Poly(4x^8 + 6x^7 + 3x^6 + 4x^3 + x + 2, GF(7))
```

(continues on next page)

(continued from previous page)

```
In [10]: for i in range(len(a)):
   ....:     ai = x % m[i]
   ....:     print(f"{x} = {ai} (mod {m[i]}), Valid congruence: {ai == a[i]}")
   ....:
Poly(4x^8 + 6x^7 + 3x^6 + 4x^3 + x + 2, GF(7)) = Poly(5x + 3, GF(7)) (mod Poly(x^2␣
↪+ 1, GF(7))), Valid congruence: True
Poly(4x^8 + 6x^7 + 3x^6 + 4x^3 + x + 2, GF(7)) = Poly(2x^2 + 4x + 6, GF(7)) (mod␣
↪Poly(x^3 + 2, GF(7))), Valid congruence: True
Poly(4x^8 + 6x^7 + 3x^6 + 4x^3 + x + 2, GF(7)) = Poly(2x^3 + x^2 + x + 5, GF(7))␣
↪(mod Poly(x^4 + x + 1, GF(7))), Valid congruence: True
```

Polynomial factorization

factors(value)	Computes the prime factors of a positive integer or the irreducible factors of a non-constant, monic polynomial.
square_free_factorization(poly)	Factors the monic polynomial $f(x)$ into a product of square-free polynomials.
distinct_degree_factorization(poly)	Factors the monic, square-free polynomial $f(x)$ into a product of polynomials whose irreducible factors all have the same degree.
equal_degree_factorization(poly, degree)	Factors the monic, square-free polynomial $f(x)$ of degree rd into a product of r irreducible factors with degree d.

galois.factors

galois.**factors**(*value*)

Computes the prime factors of a positive integer or the irreducible factors of a non-constant, monic polynomial.

Parameters **value** (*int*, galois.Poly) – A positive integer n or a non-constant, monic polynomial $f(x)$.

Returns

- *list* – Sorted list of prime factors $\{p_1, p_2, \dots, p_k\}$ of n with $p_1 < p_2 < \dots < p_k$ or irreducible factors $\{g_1(x), g_2(x), \dots, g_k(x)\}$ of $f(x)$ sorted in lexicographically-increasing order.
- *list* – List of corresponding multiplicities $\{e_1, e_2, \dots, e_k\}$.

Notes

Integer Factorization

This function factors a positive integer n into its k prime factors such that $n = p_1^{e_1} p_2^{e_2} \dots p_k^{e_k}$.

Steps:

1. Test if n is prime. If so, return `[n], [1]`.
2. Test if n is a perfect power, such that $n = x^k$. If so, prime factor x and multiply the exponents by k.
3. Use trial division with a list of primes up to 10^6. If no residual factors, return the discovered prime factors.
4. Use Pollard's Rho algorithm to find a non-trivial factor of the residual. Continue until all are found.

Polynomial Factorization

This function factors a monic polynomial $f(x)$ into its k irreducible factors such that $f(x) = g_1(x)^{e_1} g_2(x)^{e_2} \dots g_k(x)^{e_k}$.

Steps:

1. Apply the Square-Free Factorization algorithm to factor the monic polynomial into square-free polynomials.
2. Apply the Distinct-Degree Factorization algorithm to factor each square-free polynomial into a product of factors with the same degree.
3. Apply the Equal-Degree Factorization algorithm to factor the product of factors of equal degree into their irreducible factors.

References

- D. Hachenberger, D. Jungnickel. Topics in Galois Fields. Algorithm 6.1.7.
- Section 2.1 from https://people.csail.mit.edu/dmoshkov/courses/codes/poly-factorization.pdf

Examples

Factor a positive integer.

```
In [1]: galois.factors(120)
Out[1]: ([2, 3, 5], [3, 1, 1])
```

Factor a polynomial over GF(3).

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

In [3]: g1, g2, g3 = galois.irreducible_poly(3, 3), galois.irreducible_poly(3, 4),␣
↪galois.irreducible_poly(3, 5)

In [4]: g1, g2, g3
Out[4]:
(Poly(x^3 + 2x + 1, GF(3)),
 Poly(x^4 + x + 2, GF(3)),
 Poly(x^5 + 2x + 1, GF(3)))

In [5]: e1, e2, e3 = 5, 4, 3
```

(continues on next page)

(continued from previous page)

```
# Construct the composite polynomial
In [6]: f = g1**e1 * g2**e2 * g3**e3

In [7]: galois.factors(f)
Out[7]:
([Poly(x^3 + 2x + 1, GF(3)),
  Poly(x^4 + x + 2, GF(3)),
  Poly(x^5 + 2x + 1, GF(3))],
 [5, 4, 3])
```

galois.square_free_factorization

galois.**square_free_factorization**(*poly*)

Factors the monic polynomial $f(x)$ into a product of square-free polynomials.

Parameters **poly** (`galois.Poly`) – A non-constant, monic polynomial $f(x)$ over $\mathrm{GF}(p^m)$.

Returns

- *list* – The list of non-constant, square-free polynomials $h_i(x)$ in the factorization.
- *list* – The list of corresponding multiplicities i.

Notes

The Square-Free Factorization algorithm factors $f(x)$ into a product of m square-free polynomials $h_j(x)$ with multiplicity j.

$$f(x) = \prod_{j=1}^{m} h_j(x)^j$$

Some $h_j(x) = 1$, but those polynomials are not returned by this function.

A complete polynomial factorization is implemented in *`galois.factors()`*.

References

- D. Hachenberger, D. Jungnickel. Topics in Galois Fields. Algorithm 6.1.7.
- Section 2.1 from https://people.csail.mit.edu/dmoshkov/courses/codes/poly-factorization.pdf

Examples

Suppose $f(x) = x(x^3 + 2x + 4)(x^2 + 4x + 1)^3$ over $\mathrm{GF}(5)$. Each polynomial x, $x^3 + 2x + 4$, and $x^2 + 4x + 1$ are all irreducible over $\mathrm{GF}(5)$.

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

In [2]: a = galois.Poly([1,0], field=GF); a, galois.is_irreducible(a)
Out[2]: (Poly(x, GF(5)), True)
```

(continues on next page)

(continued from previous page)

```
In [3]: b = galois.Poly([1,0,2,4], field=GF); b, galois.is_irreducible(b)
Out[3]: (Poly(x^3 + 2x + 4, GF(5)), True)

In [4]: c = galois.Poly([1,4,1], field=GF); c, galois.is_irreducible(c)
Out[4]: (Poly(x^2 + 4x + 1, GF(5)), True)

In [5]: f = a * b * c**3; f
Out[5]: Poly(x^10 + 2x^9 + 3x^8 + x^7 + x^6 + 2x^5 + 3x^3 + 4x, GF(5))
```

The square-free factorization is $\{x(x^3 + 2x + 4), x^2 + 4x + 1\}$ with multiplicities $\{1, 3\}$.

```
In [6]: galois.square_free_factorization(f)
Out[6]: ([Poly(x^4 + 2x^2 + 4x, GF(5)), Poly(x^2 + 4x + 1, GF(5))], [1, 3])

In [7]: [a*b, c], [1, 3]
Out[7]: ([Poly(x^4 + 2x^2 + 4x, GF(5)), Poly(x^2 + 4x + 1, GF(5))], [1, 3])
```

galois.distinct_degree_factorization

`galois.distinct_degree_factorization(poly)`

Factors the monic, square-free polynomial $f(x)$ into a product of polynomials whose irreducible factors all have the same degree.

Parameters **poly** (`galois.Poly`) – A monic, square-free polynomial $f(x)$ over $\mathrm{GF}(p^m)$.

Returns

- *list* – The list of polynomials $f_i(x)$ whose irreducible factors all have degree i.
- *list* – The list of corresponding distinct degrees i.

Notes

The Distinct-Degree Factorization algorithm factors a square-free polynomial $f(x)$ with degree d into a product of d polynomials $f_i(x)$, where $f_i(x)$ is the product of all irreducible factors of $f(x)$ with degree i.

$$f(x) = \prod_{i=1}^{d} f_i(x)$$

For example, suppose $f(x) = x(x+1)(x^2+x+1)(x^3+x+1)(x^3+x^2+1)$ over $\mathrm{GF}(2)$, then the distinct-degree factorization is

$$\begin{aligned}
f_1(x) &= x(x+1) = x^2 + x \\
f_2(x) &= x^2 + x + 1 \\
f_3(x) &= (x^3 + x + 1)(x^3 + x^2 + 1) = x^6 + x^5 + x^4 + x^3 + x^2 + x + 1 \\
f_i(x) &= 1 \text{ for } i = 4, \dots, 10.
\end{aligned}$$

Some $f_i(x) = 1$, but those polynomials are not returned by this function. In this example, the function returns $\{f_1(x), f_2(x), f_3(x)\}$ and $\{1, 2, 3\}$.

The Distinct-Degree Factorization algorithm is often applied after the Square-Free Factorization algorithm, see *`galois.square_free_factorization()`*. A complete polynomial factorization is implemented in *`galois.factors()`*.

References

- D. Hachenberger, D. Jungnickel. Topics in Galois Fields. Algorithm 6.2.2.
- Section 2.2 from https://people.csail.mit.edu/dmoshkov/courses/codes/poly-factorization.pdf

Examples

From the example in the notes, suppose $f(x) = x(x+1)(x^2+x+1)(x^3+x+1)(x^3+x^2+1)$ over $\mathrm{GF}(2)$.

```
In [1]: a = galois.Poly([1,0]); a, galois.is_irreducible(a)
Out[1]: (Poly(x, GF(2)), True)

In [2]: b = galois.Poly([1,1]); b, galois.is_irreducible(b)
Out[2]: (Poly(x + 1, GF(2)), True)

In [3]: c = galois.Poly([1,1,1]); c, galois.is_irreducible(c)
Out[3]: (Poly(x^2 + x + 1, GF(2)), True)

In [4]: d = galois.Poly([1,0,1,1]); d, galois.is_irreducible(d)
Out[4]: (Poly(x^3 + x + 1, GF(2)), True)

In [5]: e = galois.Poly([1,1,0,1]); e, galois.is_irreducible(e)
Out[5]: (Poly(x^3 + x^2 + 1, GF(2)), True)

In [6]: f = a * b * c * d * e; f
Out[6]: Poly(x^10 + x^9 + x^8 + x^3 + x^2 + x, GF(2))
```

The distinct-degree factorization is $\{x(x+1), x^2+x+1, (x^3+x+1)(x^3+x^2+1)\}$ whose irreducible factors have degrees $\{1, 2, 3\}$.

```
In [7]: galois.distinct_degree_factorization(f)
Out[7]:
([Poly(x^2 + x, GF(2)),
  Poly(x^2 + x + 1, GF(2)),
  Poly(x^6 + x^5 + x^4 + x^3 + x^2 + x + 1, GF(2))],
 [1, 2, 3])

In [8]: [a*b, c, d*e], [1, 2, 3]
Out[8]:
([Poly(x^2 + x, GF(2)),
  Poly(x^2 + x + 1, GF(2)),
  Poly(x^6 + x^5 + x^4 + x^3 + x^2 + x + 1, GF(2))],
 [1, 2, 3])
```

galois.equal_degree_factorization

`galois.equal_degree_factorization(poly, degree)`

Factors the monic, square-free polynomial $f(x)$ of degree rd into a product of r irreducible factors with degree d.

Parameters

- **poly** (`galois.Poly`) – A monic, square-free polynomial $f(x)$ over $\mathrm{GF}(p^m)$.
- **degree** (*int*) – The degree d of each irreducible factor of $f(x)$.

Returns The list of r irreducible factors $\{g_1(x), \dots, g_r(x)\}$ in lexicographically-increasing order.

Return type list

Notes

The Equal-Degree Factorization algorithm factors a square-free polynomial $f(x)$ with degree rd into a product of r irreducible polynomials each with degree d. This function implements the Cantor-Zassenhaus algorithm, which is probabilistic.

The Equal-Degree Factorization algorithm is often applied after the Distinct-Degree Factorization algorithm, see `galois.distinct_degree_factorization()`. A complete polynomial factorization is implemented in `galois.factors()`.

References

- Section 2.3 from https://people.csail.mit.edu/dmoshkov/courses/codes/poly-factorization.pdf
- Section 1 from https://www.csa.iisc.ac.in/~chandan/courses/CNT/notes/lec8.pdf

Examples

Factor a product of degree-1 irreducible polynomials over $\mathrm{GF}(2)$.

```
In [1]: a = galois.Poly([1,0]); a, galois.is_irreducible(a)
Out[1]: (Poly(x, GF(2)), True)

In [2]: b = galois.Poly([1,1]); b, galois.is_irreducible(b)
Out[2]: (Poly(x + 1, GF(2)), True)

In [3]: f = a * b; f
Out[3]: Poly(x^2 + x, GF(2))

In [4]: galois.equal_degree_factorization(f, 1)
Out[4]: [Poly(x, GF(2)), Poly(x + 1, GF(2))]
```

Factor a product of degree-3 irreducible polynomials over $\mathrm{GF}(5)$.

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

In [6]: a = galois.Poly([1,0,2,1], field=GF); a, galois.is_irreducible(a)
Out[6]: (Poly(x^3 + 2x + 1, GF(5)), True)
```

(continues on next page)

(continued from previous page)

```
In [7]: b = galois.Poly([1,4,4,4], field=GF); b, galois.is_irreducible(b)
Out[7]: (Poly(x^3 + 4x^2 + 4x + 4, GF(5)), True)

In [8]: f = a * b; f
Out[8]: Poly(x^6 + 4x^5 + x^4 + 3x^3 + 2x^2 + 2x + 4, GF(5))

In [9]: galois.equal_degree_factorization(f, 3)
Out[9]: [Poly(x^3 + 2x + 1, GF(5)), Poly(x^3 + 4x^2 + 4x + 4, GF(5))]
```

Polynomial tests

is_monic(poly)	Determines whether the polynomial is monic, i.e. having leading coefficient equal to 1.
is_irreducible(poly)	Determines whether the polynomial $f(x)$ over $\mathrm{GF}(p^m)$ is irreducible.
is_primitive(poly)	Determines whether the polynomial $f(x)$ over $\mathrm{GF}(q)$ is primitive.
is_square_free(value)	Determines if the positive integer or the non-constant, monic polynomial is square-free.

galois.is_monic

galois.**is_monic**(*poly*)

Determines whether the polynomial is monic, i.e. having leading coefficient equal to 1.

Parameters **poly** (galois.Poly) – A polynomial over a Galois field.

Returns True if the polynomial is monic.

Return type bool

Examples

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

In [2]: p = galois.Poly([1,0,4,5], field=GF); p
Out[2]: Poly(x^3 + 4x + 5, GF(7))

In [3]: galois.is_monic(p)
Out[3]: True
```

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

In [5]: galois.is_monic(p)
Out[5]: False
```

galois.is_square_free

`galois.is_square_free(value)`

Determines if the positive integer or the non-constant, monic polynomial is square-free.

Parameters **value** (*int*, galois.Poly) – A positive integer n or a non-constant, monic polynomial $f(x)$.

Returns `True` if the integer or polynomial is square-free.

Return type bool

Notes

A square-free integer n is divisible by no perfect squares. As a consequence, the prime factorization of a square-free integer n is

$$n = \prod_{i=1}^{k} p_i^{e_i} = \prod_{i=1}^{k} p_i.$$

Similarly, a square-free polynomial $f(x)$ has no irreducible factors with multiplicity greater than one. Therefore, its canonical factorization is

$$f(x) = \prod_{i=1}^{k} g_i(x)^{e_i} = \prod_{i=1}^{k} g_i(x).$$

Examples

Determine if an integer is square-free.

```
In [1]: galois.is_square_free(10)
Out[1]: True

In [2]: galois.is_square_free(16)
Out[2]: False
```

Determine if a polynomial is square-free over $\mathrm{GF}(3)$.

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

In [4]: g3 = galois.irreducible_poly(3, 3); g3
Out[4]: Poly(x^3 + 2x + 1, GF(3))

In [5]: g4 = galois.irreducible_poly(3, 4); g4
Out[5]: Poly(x^4 + x + 2, GF(3))

In [6]: galois.is_square_free(g3 * g4)
Out[6]: True

In [7]: galois.is_square_free(g3**2 * g4)
Out[7]: False
```

7.3 Forward Error Correcting Codes

This section contains classes and functions for constructing forward error correction codes.

7.3.1 FEC classes

BCH(n, k[, primitive_poly, …])	A primitive, narrow-sense binary BCH(n, k) code.
ReedSolomon(n, k[, c, primitive_poly, …])	A general RS(n, k) code.

galois.BCH

class galois.**BCH**(*n*, *k*, *primitive_poly=None*, *primitive_element=None*, *systematic=True*)

A primitive, narrow-sense binary BCH(n, k) code.

A BCH(n, k) code is a $[n, k, d]_2$ linear block code with codeword size n, message size k, minimum distance d, and symbols taken from an alphabet of size 2.

To create the shortened BCH$(n-s, k-s)$ code, construct the full-sized BCH(n, k) code and then pass $k-s$ bits into `encode()` and $n-s$ bits into `decode()`. Shortened codes are only applicable for systematic codes.

Examples

Construct the BCH code.

```
In [1]: galois.bch_valid_codes(15)
Out[1]: [(15, 11, 1), (15, 7, 2), (15, 5, 3), (15, 1, 7)]

In [2]: bch = galois.BCH(15, 7); bch
Out[2]: <BCH Code: [15, 7, 5] over GF(2)>
```

Encode a message.

```
In [3]: m = galois.GF2.Random(bch.k); m
Out[3]: GF([1, 1, 1, 1, 1, 1, 1], order=2)

In [4]: c = bch.encode(m); c
Out[4]: GF([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1], order=2)
```

Corrupt the codeword and decode the message.

```
# Corrupt the first bit in the codeword
In [5]: c[0] ^= 1

In [6]: dec_m = bch.decode(c); dec_m
Out[6]: GF([1, 1, 1, 1, 1, 1, 1], order=2)

In [7]: np.array_equal(dec_m, m)
Out[7]: True
```

```
# Instruct the decoder to return the number of corrected bit errors
In [8]: dec_m, N = bch.decode(c, errors=True); dec_m, N
Out[8]: (GF([1, 1, 1, 1, 1, 1, 1], order=2), 1)

In [9]: np.array_equal(dec_m, m)
Out[9]: True
```

Constructors

__init__(n, k[, primitive_poly, ...])	Constructs a primitive, narrow-sense binary BCH(n, k) code.

Methods

decode(codeword[, errors])	Decodes the BCH codeword $\mathbf{c}$ into the message $\mathbf{m}$.
detect(codeword)	Detects if errors are present in the BCH codeword $\mathbf{c}$.
encode(message[, parity_only])	Encodes the message $\mathbf{m}$ into the BCH codeword $\mathbf{c}$.

Attributes

G	The generator matrix $\mathbf{G}$ with shape (k, n).
H	The parity-check matrix $\mathbf{H}$ with shape $(2t, n)$.
d	The design distance d of the $[n, k, d]_2$ code.
field	The Galois field $\mathrm{GF}(2^m)$ that defines the BCH code.
generator_poly	The generator polynomial $g(x)$ whose roots are *roots*.
is_narrow_sense	Indicates if the BCH code is narrow sense, meaning the roots of the generator polynomial are consecutive powers of α starting at 1, i.e. $\alpha, \alpha^2, \dots, \alpha^{2t}$.
is_primitive	Indicates if the BCH code is primitive, meaning $n = 2^m - 1$.
k	The message size k of the $[n, k, d]_2$ code
n	The codeword size n of the $[n, k, d]_2$ code
roots	The $2t$ roots of the generator polynomial.
systematic	Indicates if the code is configured to return codewords in systematic form.
t	The error-correcting capability of the code.

__init__(*n*, *k*, *primitive_poly=None*, *primitive_element=None*, *systematic=True*)

Constructs a primitive, narrow-sense binary BCH(n, k) code.

Parameters

- **n** (*int*) – The codeword size n, must be $n = 2^m - 1$.
- **k** (*int*) – The message size k.
- **primitive_poly** (galois.Poly, *optional*) – Optionally specify the primitive polynomial that defines the extension field $\mathrm{GF}(2^m)$. The default is `None` which uses

Matlab's default, see `galois.matlab_primitive_poly()`. Matlab tends to use the lexicographically-minimal primitive polynomial as a default instead of the Conway polynomial.

- **primitive_element** (*int*, `galois.Poly`, *optional*) – Optionally specify the primitive element α whose powers are roots of the generator polynomial $g(x)$. The default is `None` which uses the lexicographically-minimal primitive element in $\mathrm{GF}(2^m)$, see `galois.primitive_element()`.
- **systematic** (*bool*, *optional*) – Optionally specify if the encoding should be systematic, meaning the codeword is the message with parity appended. The default is `True`.

Returns A primitive, narrow-sense binary $\mathrm{BCH}(n, k)$ code object.

Return type *galois.BCH*

decode(*codeword*, *errors=False*)

Decodes the BCH codeword $\mathbf{c}$ into the message $\mathbf{m}$.

Parameters

- **codeword** (*numpy.ndarray*, `galois.FieldArray`) – The codeword as either a n-length vector or (N, n) matrix, where N is the number of codewords. For systematic codes, codeword lengths less than n may be provided for shortened codewords.
- **errors** (*bool*, *optional*) – Optionally specify whether to return the nubmer of corrected errors.

Returns

- *numpy.ndarray, galois.FieldArray* – The decoded message as either a k-length vector or (N, k) matrix.
- *int, np.ndarray* – Optional return argument of the number of corrected bit errors as either a scalar or n-length vector. Valid number of corrections are in $[0, t]$. If a codeword has too many errors and cannot be corrected, -1 will be returned.

Notes

The codeword vector $\mathbf{c}$ is defined as $\mathbf{c} = [c_{n-1}, \dots, c_1, c_0] \in \mathrm{GF}(2)^n$, which corresponds to the codeword polynomial $c(x) = c_{n-1}x^{n-1} + \dots + c_1x + c_0$. The message vector $\mathbf{m}$ is defined as $\mathbf{m} = [m_{k-1}, \dots, m_1, m_0] \in \mathrm{GF}(2)^k$, which corresponds to the message polynomial $m(x) = m_{k-1}x^{k-1} + \dots + m_1x + m_0$.

In decoding, the syndrome vector s is computed by $\mathbf{s} = \mathbf{c}\mathbf{H}^T$, where $\mathbf{H}$ is the parity-check matrix. The equivalent polynomial operation is $s(x) = c(x) \bmod g(x)$. A syndrome of zeros indicates the received codeword is a valid codeword and there are no errors. If the syndrome is non-zero, the decoder will find an error-locator polynomial $\sigma(x)$ and the corresponding error locations and values.

For the shortened $\mathrm{BCH}(n - s, k - s)$ code (only applicable for systematic codes), pass $n - s$ bits into `decode()` to return the $k - s$-bit message.

Examples

Decode a single codeword.

```
In [1]: bch = galois.BCH(15, 7)

In [2]: m = galois.GF2.Random(bch.k); m
Out[2]: GF([0, 0, 0, 0, 1, 0, 0], order=2)

In [3]: c = bch.encode(m); c
Out[3]: GF([0, 0, 0, 0, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1, 0], order=2)

# Corrupt the first bit in the codeword
In [4]: c[0] ^= 1

In [5]: dec_m = bch.decode(c); dec_m
Out[5]: GF([0, 0, 0, 0, 1, 0, 0], order=2)

In [6]: np.array_equal(dec_m, m)
Out[6]: True

# Instruct the decoder to return the number of corrected bit errors
In [7]: dec_m, N = bch.decode(c, errors=True); dec_m, N
Out[7]: (GF([0, 0, 0, 0, 1, 0, 0], order=2), 1)

In [8]: np.array_equal(dec_m, m)
Out[8]: True
```

Decode a single, shortened codeword.

```
In [9]: m = galois.GF2.Random(bch.k - 3); m
Out[9]: GF([0, 0, 0, 1], order=2)

In [10]: c = bch.encode(m); c
Out[10]: GF([0, 0, 0, 1, 1, 1, 0, 1, 0, 0, 0, 1], order=2)

# Corrupt the first bit in the codeword
In [11]: c[0] ^= 1

In [12]: dec_m = bch.decode(c); dec_m
Out[12]: GF([0, 0, 0, 1], order=2)

In [13]: np.array_equal(dec_m, m)
Out[13]: True
```

Decode a matrix of codewords.

```
In [14]: m = galois.GF2.Random((5, bch.k)); m
Out[14]:
GF([[1, 1, 1, 1, 0, 1, 1],
    [0, 0, 0, 0, 1, 0, 1],
    [0, 0, 1, 1, 0, 1, 1],
    [1, 0, 1, 0, 1, 0, 0],
    [1, 1, 0, 0, 0, 0, 1]], order=2)
```

(continues on next page)

(continued from previous page)

```
In [15]: c = bch.encode(m); c
Out[15]:
GF([[1, 1, 1, 1, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 1],
    [0, 0, 0, 0, 1, 0, 1, 0, 0, 1, 1, 0, 1, 1, 1],
    [0, 0, 1, 1, 0, 1, 1, 1, 0, 0, 0, 0, 1, 0, 1],
    [1, 0, 1, 0, 1, 0, 0, 0, 0, 1, 1, 0, 1, 0, 0],
    [1, 1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 1, 1, 0, 1]], order=2)

# Corrupt the first bit in each codeword
In [16]: c[:,0] ^= 1

In [17]: dec_m = bch.decode(c); dec_m
Out[17]:
GF([[1, 1, 1, 1, 0, 1, 1],
    [0, 0, 0, 0, 1, 0, 1],
    [0, 0, 1, 1, 0, 1, 1],
    [1, 0, 1, 0, 1, 0, 0],
    [1, 1, 0, 0, 0, 0, 1]], order=2)

In [18]: np.array_equal(dec_m, m)
Out[18]: True

# Instruct the decoder to return the number of corrected bit errors
In [19]: dec_m, N = bch.decode(c, errors=True); dec_m, N
Out[19]:
(GF([[1, 1, 1, 1, 0, 1, 1],
     [0, 0, 0, 0, 1, 0, 1],
     [0, 0, 1, 1, 0, 1, 1],
     [1, 0, 1, 0, 1, 0, 0],
     [1, 1, 0, 0, 0, 0, 1]], order=2),
 array([1, 1, 1, 1, 1]))

In [20]: np.array_equal(dec_m, m)
Out[20]: True
```

detect(*codeword*)

Detects if errors are present in the BCH codeword **c**.

The $[n, k, d]_2$ BCH code has $d_{min} \geq d$ minimum distance. It can detect up to $d_{min} - 1$ errors.

Parameters **codeword** (`numpy.ndarray`, `galois.FieldArray`) – The codeword as either a n-length vector or (N, n) matrix, where N is the number of codewords. For systematic codes, codeword lengths less than n may be provided for shortened codewords.

Returns A boolean scalar or array indicating if errors were detected in the corresponding codeword `True` or not `False`.

Return type bool, numpy.ndarray

Examples

Detect errors in a valid codeword.

```
In [1]: bch = galois.BCH(15, 7)

# The minimum distance of the code
In [2]: bch.d
Out[2]: 5

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

In [4]: c = bch.encode(m); c
Out[4]: GF([1, 0, 0, 1, 1, 0, 1, 1, 1, 0, 0, 0, 0, 1, 0], order=2)

In [5]: bch.detect(c)
Out[5]: False
```

Detect $d_{min} - 1$ errors in a received codeword.

```
# Corrupt the first `d - 1` bits in the codeword
In [6]: c[0:bch.d - 1] ^= 1

In [7]: bch.detect(c)
Out[7]: True
```

encode(*message*, *parity_only=False*)

Encodes the message $\mathbf{m}$ into the BCH codeword $\mathbf{c}$.

Parameters

- **message** (*numpy.ndarray*, `galois.FieldArray`) – The message as either a k-length vector or (N, k) matrix, where N is the number of messages. For systematic codes, message lengths less than k may be provided to produce shortened codewords.
- **parity_only** (*bool*, *optional*) – Optionally specify whether to return only the parity bits. This only applies to systematic codes. The default is `False`.

Returns The codeword as either a n-length vector or (N, n) matrix. The return type matches the message type. If `parity_only=True`, the parity bits are returned as either a $n - k$-length vector or $(N, n - k)$ matrix.

Return type numpy.ndarray, *galois.FieldArray*

Notes

The message vector $\mathbf{m}$ is defined as $\mathbf{m} = [m_{k-1}, \dots, m_1, m_0] \in \mathrm{GF}(2)^k$, which corresponds to the message polynomial $m(x) = m_{k-1}x^{k-1} + \dots + m_1 x + m_0$. The codeword vector $\mathbf{c}$ is defined as $\mathbf{c} = [c_{n-1}, \dots, c_1, c_0] \in \mathrm{GF}(2)^n$, which corresponds to the codeword polynomial $c(x) = c_{n-1}x^{n-1} + \dots + c_1 x + c_0$.

The codeword vector is computed from the message vector by $\mathbf{c} = \mathbf{m}\mathbf{G}$, where $\mathbf{G}$ is the generator matrix. The equivalent polynomial operation is $c(x) = m(x)g(x)$. For systematic codes, $\mathbf{G} = [\mathbf{I}\ |\ \mathbf{P}]$ such that $\mathbf{c} = [\mathbf{m}\ |\ \mathbf{p}]$. And in polynomial form, $p(x) = -(m(x)x^{n-k}\ \textrm{mod}\ g(x))$ with $c(x) = m(x)x^{n-k} + p(x)$. For systematic and non-systematic codes, each codeword is a multiple of the generator polynomial, i.e. $g(x)\ |\ c(x)$.

For the shortened BCH$(n - s, k - s)$ code (only applicable for systematic codes), pass $k - s$ bits into `encode()` to return the $n - s$-bit codeword.

Examples

Encode a single codeword.

```
In [1]: bch = galois.BCH(15, 7)

In [2]: m = galois.GF2.Random(bch.k); m
Out[2]: GF([0, 0, 1, 1, 0, 0, 1], order=2)

In [3]: c = bch.encode(m); c
Out[3]: GF([0, 0, 1, 1, 0, 0, 1, 1, 1, 1, 1, 0, 1, 1, 0], order=2)

In [4]: p = bch.encode(m, parity_only=True); p
Out[4]: GF([1, 1, 1, 1, 0, 1, 1, 0], order=2)
```

Encode a single, shortened codeword.

```
In [5]: m = galois.GF2.Random(bch.k - 3); m
Out[5]: GF([0, 1, 1, 0], order=2)

In [6]: c = bch.encode(m); c
Out[6]: GF([0, 1, 1, 0, 1, 0, 0, 1, 0, 1, 0, 1], order=2)
```

Encode a matrix of codewords.

```
In [7]: m = galois.GF2.Random((5, bch.k)); m
Out[7]:
GF([[1, 1, 1, 0, 1, 1, 0],
    [0, 0, 1, 0, 1, 1, 0],
    [1, 0, 0, 0, 0, 0, 0],
    [1, 1, 0, 1, 0, 0, 1],
    [0, 1, 0, 0, 0, 0, 1]], order=2)

In [8]: c = bch.encode(m); c
Out[8]:
GF([[1, 1, 1, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 1, 1],
    [0, 0, 1, 0, 1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1],
    [1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 0, 0, 0],
    [1, 1, 0, 1, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0],
    [0, 1, 0, 0, 0, 0, 1, 1, 0, 1, 0, 0, 1, 0, 1]], order=2)

In [9]: p = bch.encode(m, parity_only=True); p
Out[9]:
GF([[0, 0, 1, 1, 0, 0, 1, 1],
    [1, 0, 1, 0, 1, 1, 1, 1],
    [1, 1, 1, 0, 1, 0, 0, 0],
    [0, 1, 0, 1, 0, 0, 0, 0],
    [1, 0, 1, 0, 0, 1, 0, 1]], order=2)
```

property G

The generator matrix $\mathbf{G}$ with shape (k, n).

Examples

```
In [1]: bch = galois.BCH(15, 7); bch
Out[1]: <BCH Code: [15, 7, 5] over GF(2)>

In [2]: bch.G
Out[2]:
GF([[1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 0, 0, 0],
    [0, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 0, 0],
    [0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 0],
    [0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1],
    [0, 0, 0, 0, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1, 0],
    [0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1],
    [0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 0, 0, 0, 1]], order=2)
```

Type *galois.GF2*

property H

The parity-check matrix **H** with shape $(2t, n)$.

Examples

```
In [1]: bch = galois.BCH(15, 7); bch
Out[1]: <BCH Code: [15, 7, 5] over GF(2)>

In [2]: bch.H
Out[2]:
GF([[ 9, 13, 15, 14,  7, 10,  5, 11, 12,  6,  3,  8,  4,  2,  1],
    [13, 14, 10, 11,  6,  8,  2,  9, 15,  7,  5, 12,  3,  4,  1],
    [15, 10, 12,  8,  1, 15, 10, 12,  8,  1, 15, 10, 12,  8,  1],
    [14, 11,  8,  9,  7, 12,  4, 13, 10,  6,  2, 15,  5,  3,  1]],
   order=2^4)
```

Type *galois.FieldArray*

property d

The design distance d of the $[n, k, d]_2$ code. The minimum distance of a BCH code may be greater than the design distance, $d_{min} \geq d$.

Examples

```
In [1]: bch = galois.BCH(15, 7); bch
Out[1]: <BCH Code: [15, 7, 5] over GF(2)>

In [2]: bch.d
Out[2]: 5
```

Type int

property field

The Galois field $\mathrm{GF}(2^m)$ that defines the BCH code.

Examples

```
In [1]: bch = galois.BCH(15, 7); bch
Out[1]: <BCH Code: [15, 7, 5] over GF(2)>

In [2]: bch.field
Out[2]: <class 'numpy.ndarray over GF(2^4)'>

In [3]: print(bch.field.properties)
GF(2^4):
  characteristic: 2
  degree: 4
  order: 16
  irreducible_poly: x^4 + x + 1
  is_primitive_poly: True
  primitive_element: x
```

Type *galois.FieldClass*

property generator_poly

The generator polynomial $g(x)$ whose roots are `roots`.

Examples

```
In [1]: bch = galois.BCH(15, 7); bch
Out[1]: <BCH Code: [15, 7, 5] over GF(2)>

In [2]: bch.generator_poly
Out[2]: Poly(x^8 + x^7 + x^6 + x^4 + 1, GF(2))

# Evaluate the generator polynomial at its roots in GF(2^m)
In [3]: bch.generator_poly(bch.roots, field=bch.field)
Out[3]: GF([0, 0, 0, 0], order=2^4)
```

Type *galois.Poly*

property is_narrow_sense

Indicates if the BCH code is narrow sense, meaning the roots of the generator polynomial are consecutive powers of α starting at 1, i.e. $\alpha, \alpha^2, \dots, \alpha^{2t}$.

Examples

```
In [1]: bch = galois.BCH(15, 7); bch
Out[1]: <BCH Code: [15, 7, 5] over GF(2)>

In [2]: bch.is_narrow_sense
Out[2]: True

In [3]: bch.roots
Out[3]: GF([2, 4, 8, 3], order=2^4)

In [4]: bch.field.primitive_element**(np.arange(1, 2*bch.t + 1))
Out[4]: GF([2, 4, 8, 3], order=2^4)
```

Type bool

property is_primitive

Indicates if the BCH code is primitive, meaning $n = 2^m - 1$.

Examples

```
In [1]: bch = galois.BCH(15, 7); bch
Out[1]: <BCH Code: [15, 7, 5] over GF(2)>

In [2]: bch.is_primitive
Out[2]: True
```

Type bool

property k

The message size k of the $[n, k, d]_2$ code

Examples

```
In [1]: bch = galois.BCH(15, 7); bch
Out[1]: <BCH Code: [15, 7, 5] over GF(2)>

In [2]: bch.k
Out[2]: 7
```

Type int

property n

The codeword size n of the $[n, k, d]_2$ code

Examples

```
In [1]: bch = galois.BCH(15, 7); bch
Out[1]: <BCH Code: [15, 7, 5] over GF(2)>

In [2]: bch.n
Out[2]: 15
```

Type int

property roots

The $2t$ roots of the generator polynomial. These are consecutive powers of α, specifically $\alpha, \alpha^2, \dots, \alpha^{2t}$.

Examples

```
In [1]: bch = galois.BCH(15, 7); bch
Out[1]: <BCH Code: [15, 7, 5] over GF(2)>

In [2]: bch.roots
Out[2]: GF([2, 4, 8, 3], order=2^4)

# Evaluate the generator polynomial at its roots in GF(2^m)
In [3]: bch.generator_poly(bch.roots, field=bch.field)
Out[3]: GF([0, 0, 0, 0], order=2^4)
```

Type *galois.FieldArray*

property systematic

Indicates if the code is configured to return codewords in systematic form.

Examples

```
In [1]: bch = galois.BCH(15, 7); bch
Out[1]: <BCH Code: [15, 7, 5] over GF(2)>

In [2]: bch.systematic
Out[2]: True
```

Type bool

property t

The error-correcting capability of the code. The code can correct t bit errors in a codeword.

Examples

```
In [1]: bch = galois.BCH(15, 7); bch
Out[1]: <BCH Code: [15, 7, 5] over GF(2)>

In [2]: bch.t
Out[2]: 2
```

Type int

galois.ReedSolomon

class galois.**ReedSolomon**(*n*, *k*, *c=1*, *primitive_poly=None*, *primitive_element=None*, *systematic=True*)

A general RS(n, k) code.

A RS(n, k) code is a $[n, k, d]_q$ linear block code with codeword size n, message size k, minimum distance d, and symbols taken from an alphabet of size q (a prime power).

To create the shortened RS$(n-s, k-s)$ code, construct the full-sized RS(n, k) code and then pass $k-s$ symbols into `encode()` and $n-s$ symbols into `decode()`. Shortened codes are only applicable for systematic codes.

Examples

Construct the Reed-Solomon code.

```
In [1]: rs = galois.ReedSolomon(15, 9)

In [2]: GF = rs.field
```

Encode a message.

```
In [3]: m = GF.Random(rs.k); m
Out[3]: GF([15,  2,  5,  5, 13,  9,  6,  6,  4], order=2^4)

In [4]: c = rs.encode(m); c
Out[4]:
GF([15,  2,  5,  5, 13,  9,  6,  6,  4,  7,  2,  6, 10,  7,  7],
   order=2^4)
```

Corrupt the codeword and decode the message.

```
# Corrupt the first symbol in the codeword
In [5]: c[0] ^= 13

In [6]: dec_m = rs.decode(c); dec_m
Out[6]: GF([15,  2,  5,  5, 13,  9,  6,  6,  4], order=2^4)

In [7]: np.array_equal(dec_m, m)
Out[7]: True
```

```
# Instruct the decoder to return the number of corrected symbol errors
In [8]: dec_m, N = rs.decode(c, errors=True); dec_m, N
```

(continues on next page)

(continued from previous page)

```
Out[8]: (GF([15,  2,  5,  5, 13,  9,  6,  6,  4], order=2^4), 1)

In [9]: np.array_equal(dec_m, m)
Out[9]: True
```

Constructors

__init__(n, k[, c, primitive_poly, ...])	Constructs a general RS(n, k) code.

Methods

decode(codeword[, errors])	Decodes the Reed-Solomon codeword $\mathbf{c}$ into the message $\mathbf{m}$.
detect(codeword)	Detects if errors are present in the Reed-Solomon codeword $\mathbf{c}$.
encode(message[, parity_only])	Encodes the message $\mathbf{m}$ into the Reed-Solomon codeword $\mathbf{c}$.

Attributes

G	The generator matrix $\mathbf{G}$ with shape (k, n).
H	The parity-check matrix $\mathbf{H}$ with shape $(2t, n)$.
c	The degree of the first consecutive root.
d	The design distance d of the $[n, k, d]_q$ code.
field	The Galois field $\mathrm{GF}(q)$ that defines the Reed-Solomon code.
generator_poly	The generator polynomial $g(x)$ whose roots are *roots*.
is_narrow_sense	Indicates if the Reed-Solomon code is narrow sense, meaning the roots of the generator polynomial are consecutive powers of α starting at 1, i.e. $\alpha, \alpha^2, \dots, \alpha^{2t-1}$.
k	The message size k of the $[n, k, d]_q$ code.
n	The codeword size n of the $[n, k, d]_q$ code.
roots	The $2t$ roots of the generator polynomial.
systematic	Indicates if the code is configured to return codewords in systematic form.
t	The error-correcting capability of the code.

__init__(*n*, *k*, *c=1*, *primitive_poly=None*, *primitive_element=None*, *systematic=True*)

Constructs a general RS(n, k) code.

Parameters

- **n** (*int*) – The codeword size n, must be $n = q - 1$ where q is a prime power.
- **k** (*int*) – The message size k. The error-correcting capability t is defined by $n - k = 2t$.

- **c** (*int*, *optional*) – The first consecutive power of α. The default is 1.
- **primitive_poly** (galois.Poly, *optional*) – Optionally specify the primitive polynomial that defines the extension field $\mathrm{GF}(q)$. The default is None which uses Matlab's default, see *galois.matlab_primitive_poly()*. Matlab tends to use the lexicographically-minimal primitive polynomial as a default instead of the Conway polynomial.
- **primitive_element** (*int*, galois.Poly, *optional*) – Optionally specify the primitive element α of $\mathrm{GF}(q)$ whose powers are roots of the generator polynomial $g(x)$. The default is None which uses the lexicographically-minimal primitive element in $\mathrm{GF}(q)$, see *galois.primitive_element()*.
- **systematic** (*bool*, *optional*) – Optionally specify if the encoding should be systematic, meaning the codeword is the message with parity appended. The default is True.

Returns A general $\mathrm{RS}(n, k)$ code object.

Return type *galois.ReedSolomon*

decode(*codeword*, *errors=False*)

Decodes the Reed-Solomon codeword $\mathbf{c}$ into the message $\mathbf{m}$.

Parameters

- **codeword** (*numpy.ndarray*, galois.FieldArray) – The codeword as either a n-length vector or (N, n) matrix, where N is the number of codewords. For systematic codes, codeword lengths less than n may be provided for shortened codewords.
- **errors** (*bool*, *optional*) – Optionally specify whether to return the nubmer of corrected errors.

Returns

- *numpy.ndarray, galois.FieldArray* – The decoded message as either a k-length vector or (N, k) matrix.
- *int, np.ndarray* – Optional return argument of the number of corrected symbol errors as either a scalar or n-length vector. Valid number of corrections are in $[0, t]$. If a codeword has too many errors and cannot be corrected, -1 will be returned.

Notes

The codeword vector $\mathbf{c}$ is defined as $\mathbf{c} = [c_{n-1}, \dots, c_1, c_0] \in \mathrm{GF}(q)^n$, which corresponds to the codeword polynomial $c(x) = c_{n-1}x^{n-1} + \dots + c_1 x + c_0$. The message vector $\mathbf{m}$ is defined as $\mathbf{m} = [m_{k-1}, \dots, m_1, m_0] \in \mathrm{GF}(q)^k$, which corresponds to the message polynomial $m(x) = m_{k-1}x^{k-1} + \dots + m_1 x + m_0$.

In decoding, the syndrome vector s is computed by $\mathbf{s} = \mathbf{c}\mathbf{H}^T$, where $\mathbf{H}$ is the parity-check matrix. The equivalent polynomial operation is $s(x) = c(x) \bmod g(x)$. A syndrome of zeros indicates the received codeword is a valid codeword and there are no errors. If the syndrome is non-zero, the decoder will find an error-locator polynomial $\sigma(x)$ and the corresponding error locations and values.

For the shortened $\mathrm{RS}(n - s, k - s)$ code (only applicable for systematic codes), pass $n - s$ symbols into *decode()* to return the $k - s$-symbol message.

Examples

Decode a single codeword.

```
In [1]: rs = galois.ReedSolomon(15, 9)

In [2]: GF = rs.field

In [3]: m = GF.Random(rs.k); m
Out[3]: GF([11, 14, 12,  1,  4,  8,  5,  9,  1], order=2^4)

In [4]: c = rs.encode(m); c
Out[4]:
GF([11, 14, 12,  1,  4,  8,  5,  9,  1,  8,  3,  1,  0,  2, 14],
   order=2^4)

# Corrupt the first symbol in the codeword
In [5]: c[0] += GF(13)

In [6]: dec_m = rs.decode(c); dec_m
Out[6]: GF([11, 14, 12,  1,  4,  8,  5,  9,  1], order=2^4)

In [7]: np.array_equal(dec_m, m)
Out[7]: True

# Instruct the decoder to return the number of corrected symbol errors
In [8]: dec_m, N = rs.decode(c, errors=True); dec_m, N
Out[8]: (GF([11, 14, 12,  1,  4,  8,  5,  9,  1], order=2^4), 1)

In [9]: np.array_equal(dec_m, m)
Out[9]: True
```

Decode a single, shortened codeword.

```
In [10]: m = GF.Random(rs.k - 4); m
Out[10]: GF([ 7, 14,  5,  7,  3], order=2^4)

In [11]: c = rs.encode(m); c
Out[11]: GF([ 7, 14,  5,  7,  3,  2,  8,  4,  0, 11,  5], order=2^4)

# Corrupt the first symbol in the codeword
In [12]: c[0] += GF(13)

In [13]: dec_m = rs.decode(c); dec_m
Out[13]: GF([ 7, 14,  5,  7,  3], order=2^4)

In [14]: np.array_equal(dec_m, m)
Out[14]: True
```

Decode a matrix of codewords.

```
In [15]: m = GF.Random((5, rs.k)); m
Out[15]:
GF([[15,  1, 12, 14, 12,  7,  1,  8, 10],
```

(continues on next page)

(continued from previous page)

```
    [12,  4, 13,  4,  6, 15, 15,  9,  0],
    [ 9,  5,  6,  2,  8,  8,  2,  0, 10],
    [12, 15,  4,  7,  4, 10, 14,  7,  8],
    [ 8,  0,  7,  3,  5,  1,  7,  9,  2]], order=2^4)

In [16]: c = rs.encode(m); c
Out[16]:
GF([[15,  1, 12, 14, 12,  7,  1,  8, 10,  0, 14,  6,  2,  2, 10],
    [12,  4, 13,  4,  6, 15, 15,  9,  0, 13,  9, 11,  0,  6,  1],
    [ 9,  5,  6,  2,  8,  8,  2,  0, 10,  8, 14,  4, 10, 13,  6],
    [12, 15,  4,  7,  4, 10, 14,  7,  8,  9,  6,  7, 13,  1,  9],
    [ 8,  0,  7,  3,  5,  1,  7,  9,  2, 10, 14,  5,  1, 11,  4]],
   order=2^4)

# Corrupt the first symbol in each codeword
In [17]: c[:,0] += GF(13)

In [18]: dec_m = rs.decode(c); dec_m
Out[18]:
GF([[15,  1, 12, 14, 12,  7,  1,  8, 10],
    [12,  4, 13,  4,  6, 15, 15,  9,  0],
    [ 9,  5,  6,  2,  8,  8,  2,  0, 10],
    [12, 15,  4,  7,  4, 10, 14,  7,  8],
    [ 8,  0,  7,  3,  5,  1,  7,  9,  2]], order=2^4)

In [19]: np.array_equal(dec_m, m)
Out[19]: True

# Instruct the decoder to return the number of corrected symbol errors
In [20]: dec_m, N = rs.decode(c, errors=True); dec_m, N
Out[20]:
(GF([[15,  1, 12, 14, 12,  7,  1,  8, 10],
     [12,  4, 13,  4,  6, 15, 15,  9,  0],
     [ 9,  5,  6,  2,  8,  8,  2,  0, 10],
     [12, 15,  4,  7,  4, 10, 14,  7,  8],
     [ 8,  0,  7,  3,  5,  1,  7,  9,  2]], order=2^4),
 array([1, 1, 1, 1, 1]))

In [21]: np.array_equal(dec_m, m)
Out[21]: True
```

detect(*codeword*)

Detects if errors are present in the Reed-Solomon codeword **c**.

The $[n, k, d]_q$ Reed-Solomon code has $d_{min} = d$ minimum distance. It can detect up to $d_{min} - 1$ errors.

Parameters **codeword** (*numpy.ndarray*, galois.FieldArray) – The codeword as either a n-length vector or (N, n) matrix, where N is the number of codewords. For systematic codes, codeword lengths less than n may be provided for shortened codewords.

Returns A boolean scalar or array indicating if errors were detected in the corresponding codeword `True` or not `False`.

Return type bool, numpy.ndarray

Examples

Detect errors in a valid codeword.

```
In [1]: rs = galois.ReedSolomon(15, 9)

In [2]: GF = rs.field

# The minimum distance of the code
In [3]: rs.d
Out[3]: 7

In [4]: m = GF.Random(rs.k); m
Out[4]: GF([15, 14, 13, 14, 14, 10, 15,  7,  0], order=2^4)

In [5]: c = rs.encode(m); c
Out[5]:
GF([15, 14, 13, 14, 14, 10, 15,  7,  0, 13,  8, 11,  6, 13,  8],
   order=2^4)

In [6]: rs.detect(c)
Out[6]: False
```

Detect $d_{min} - 1$ errors in a received codeword.

```
# Corrupt the first `d - 1` symbols in the codeword
In [7]: c[0:rs.d - 1] += GF(13)

In [8]: rs.detect(c)
Out[8]: True
```

encode(*message*, *parity_only=False*)

Encodes the message $\mathbf{m}$ into the Reed-Solomon codeword $\mathbf{c}$.

Parameters

- **message** (*numpy.ndarray*, galois.FieldArray) – The message as either a k-length vector or (N, k) matrix, where N is the number of messages. For systematic codes, message lengths less than k may be provided to produce shortened codewords.
- **parity_only** (*bool*, *optional*) – Optionally specify whether to return only the parity symbols. This only applies to systematic codes. The default is `False`.

Returns The codeword as either a n-length vector or (N, n) matrix. The return type matches the message type. If `parity_only=True`, the parity symbols are returned as either a $n-k$-length vector or $(N, n-k)$ matrix.

Return type numpy.ndarray, *galois.FieldArray*

Notes

The message vector $\mathbf{m}$ is defined as $\mathbf{m} = [m_{k-1}, \dots, m_1, m_0] \in \mathrm{GF}(q)^k$, which corresponds to the message polynomial $m(x) = m_{k-1}x^{k-1} + \cdots + m_1 x + m_0$. The codeword vector $\mathbf{c}$ is defined as $\mathbf{c} = [c_{n-1}, \dots, c_1, c_0] \in \mathrm{GF}(q)^n$, which corresponds to the codeword polynomial $c(x) = c_{n-1}x^{n-1} + \cdots + c_1 x + c_0$.

The codeword vector is computed from the message vector by $\mathbf{c} = \mathbf{m}\mathbf{G}$, where $\mathbf{G}$ is the generator matrix. The equivalent polynomial operation is $c(x) = m(x)g(x)$. For systematic codes, $\mathbf{G} = [\mathbf{I}\ |\ \mathbf{P}]$ such that $\mathbf{c} = [\mathbf{m}\ |\ \mathbf{p}]$. And in polynomial form, $p(x) = -(m(x)x^{n-k}\ \mathrm{mod}\ g(x))$ with $c(x) = m(x)x^{n-k} + p(x)$. For systematic and non-systematic codes, each codeword is a multiple of the generator polynomial, i.e. $g(x)\ |\ c(x)$.

For the shortened $\mathrm{RS}(n - s, k - s)$ code (only applicable for systematic codes), pass $k - s$ symbols into `encode()` to return the $n - s$-symbol codeword.

Examples

Encode a single codeword.

```
In [1]: rs = galois.ReedSolomon(15, 9)

In [2]: GF = rs.field

In [3]: m = GF.Random(rs.k); m
Out[3]: GF([ 0,  5,  7,  7,  1, 11,  7, 12,  5], order=2^4)

In [4]: c = rs.encode(m); c
Out[4]:
GF([ 0,  5,  7,  7,  1, 11,  7, 12,  5,  7,  5,  4, 14,  2,  3],
   order=2^4)

In [5]: p = rs.encode(m, parity_only=True); p
Out[5]: GF([ 7,  5,  4, 14,  2,  3], order=2^4)
```

Encode a single, shortened codeword.

```
In [6]: m = GF.Random(rs.k - 4); m
Out[6]: GF([11,  3, 14,  0, 14], order=2^4)

In [7]: c = rs.encode(m); c
Out[7]: GF([11,  3, 14,  0, 14,  2,  3,  6,  8, 12, 15], order=2^4)
```

Encode a matrix of codewords.

```
In [8]: m = GF.Random((5, rs.k)); m
Out[8]:
GF([[ 6, 10, 15, 10,  3, 12, 15, 10,  5],
    [ 2,  1,  4,  0,  0,  1,  7,  3,  1],
    [ 6, 10,  5, 13, 13, 10,  6,  1, 11],
    [11,  7,  2,  6,  4, 11,  3,  2,  3],
    [ 3, 15,  6,  6,  8, 15, 14,  6,  2]], order=2^4)

In [9]: c = rs.encode(m); c
```

(continues on next page)

(continued from previous page)

```
Out[9]:
GF([[ 6, 10, 15, 10,  3, 12, 15, 10,  5, 14,  9, 10,  8,  7, 12],
    [ 2,  1,  4,  0,  0,  1,  7,  3,  1, 12,  3,  8,  6,  2, 13],
    [ 6, 10,  5, 13, 13, 10,  6,  1, 11,  7, 11, 13, 14,  3,  4],
    [11,  7,  2,  6,  4, 11,  3,  2,  3,  9,  0,  0,  4,  4, 14],
    [ 3, 15,  6,  6,  8, 15, 14,  6,  2, 10, 15,  7,  1,  4,  5]],
   order=2^4)

In [10]: p = rs.encode(m, parity_only=True); p
Out[10]:
GF([[14,  9, 10,  8,  7, 12],
    [12,  3,  8,  6,  2, 13],
    [ 7, 11, 13, 14,  3,  4],
    [ 9,  0,  0,  4,  4, 14],
    [10, 15,  7,  1,  4,  5]], order=2^4)
```

property G

The generator matrix $\mathbf{G}$ with shape (k, n).

Examples

```
In [1]: rs = galois.ReedSolomon(15, 9); rs
Out[1]: <Reed-Solomon Code: [15, 9, 7] over GF(2^4)>

In [2]: rs.G
Out[2]:
GF([[ 1,  0,  0,  0,  0,  0,  0,  0,  0, 10,  3,  5, 13,  1,  8],
    [ 0,  1,  0,  0,  0,  0,  0,  0,  0, 15,  1, 13,  7,  5, 13],
    [ 0,  0,  1,  0,  0,  0,  0,  0,  0, 11, 11, 13,  3, 10,  7],
    [ 0,  0,  0,  1,  0,  0,  0,  0,  0,  3,  2,  3,  8,  4,  7],
    [ 0,  0,  0,  0,  1,  0,  0,  0,  0,  3, 10, 10,  6, 15,  9],
    [ 0,  0,  0,  0,  0,  1,  0,  0,  0,  5, 11,  1,  5, 15, 11],
    [ 0,  0,  0,  0,  0,  0,  1,  0,  0,  2, 11, 10,  7, 14,  8],
    [ 0,  0,  0,  0,  0,  0,  0,  1,  0, 15,  9,  5,  8, 15,  2],
    [ 0,  0,  0,  0,  0,  0,  0,  0,  1,  7,  9,  3, 12, 10, 12]],
   order=2^4)
```

Type *galois.FieldArray*

property H

The parity-check matrix $\mathbf{H}$ with shape $(2t, n)$.

Examples

```
In [1]: rs = galois.ReedSolomon(15, 9); rs
Out[1]: <Reed-Solomon Code: [15, 9, 7] over GF(2^4)>

In [2]: rs.H
Out[2]:
GF([[ 9, 13, 15, 14,  7, 10,  5, 11, 12,  6,  3,  8,  4,  2,  1],
    [13, 14, 10, 11,  6,  8,  2,  9, 15,  7,  5, 12,  3,  4,  1],
    [15, 10, 12,  8,  1, 15, 10, 12,  8,  1, 15, 10, 12,  8,  1],
    [14, 11,  8,  9,  7, 12,  4, 13, 10,  6,  2, 15,  5,  3,  1],
    [ 7,  6,  1,  7,  6,  1,  7,  6,  1,  7,  6,  1,  7,  6,  1],
    [10,  8, 15, 12,  1, 10,  8, 15, 12,  1, 10,  8, 15, 12,  1]],
   order=2^4)
```

Type *galois.FieldArray*

property c

The degree of the first consecutive root.

Examples

```
In [1]: rs = galois.ReedSolomon(15, 9); rs
Out[1]: <Reed-Solomon Code: [15, 9, 7] over GF(2^4)>

In [2]: rs.c
Out[2]: 1
```

Type int

property d

The design distance d of the $[n, k, d]_q$ code. The minimum distance of a Reed-Solomon code is exactly equal to the design distance, $d_{min} = d$.

Examples

```
In [1]: rs = galois.ReedSolomon(15, 9); rs
Out[1]: <Reed-Solomon Code: [15, 9, 7] over GF(2^4)>

In [2]: rs.d
Out[2]: 7
```

Type int

property field

The Galois field $\mathrm{GF}(q)$ that defines the Reed-Solomon code.

Examples

```
In [1]: rs = galois.ReedSolomon(15, 9); rs
Out[1]: <Reed-Solomon Code: [15, 9, 7] over GF(2^4)>

In [2]: rs.field
Out[2]: <class 'numpy.ndarray over GF(2^4)'>

In [3]: print(rs.field.properties)
GF(2^4):
  characteristic: 2
  degree: 4
  order: 16
  irreducible_poly: x^4 + x + 1
  is_primitive_poly: True
  primitive_element: x
```

Type *galois.FieldClass*

property generator_poly

The generator polynomial $g(x)$ whose roots are *roots*.

Examples

```
In [1]: rs = galois.ReedSolomon(15, 9); rs
Out[1]: <Reed-Solomon Code: [15, 9, 7] over GF(2^4)>

In [2]: rs.generator_poly
Out[2]: Poly(x^6 + 7x^5 + 9x^4 + 3x^3 + 12x^2 + 10x + 12, GF(2^4))

# Evaluate the generator polynomial at its roots
In [3]: rs.generator_poly(rs.roots)
Out[3]: GF([0, 0, 0, 0, 0, 0], order=2^4)
```

Type *galois.Poly*

property is_narrow_sense

Indicates if the Reed-Solomon code is narrow sense, meaning the roots of the generator polynomial are consecutive powers of α starting at 1, i.e. $\alpha, \alpha^2, \ldots, \alpha^{2t-1}$.

Examples

```
In [1]: rs = galois.ReedSolomon(15, 9); rs
Out[1]: <Reed-Solomon Code: [15, 9, 7] over GF(2^4)>

In [2]: rs.is_narrow_sense
Out[2]: True

In [3]: rs.roots
Out[3]: GF([ 2,  4,  8,  3,  6, 12], order=2^4)
```

(continues on next page)

(continued from previous page)

```
In [4]: rs.field.primitive_element**(np.arange(1, 2*rs.t + 1))
Out[4]: GF([ 2,  4,  8,  3,  6, 12], order=2^4)
```

Type bool

property k

The message size k of the $[n, k, d]_q$ code.

Examples

```
In [1]: rs = galois.ReedSolomon(15, 9); rs
Out[1]: <Reed-Solomon Code: [15, 9, 7] over GF(2^4)>

In [2]: rs.k
Out[2]: 9
```

Type int

property n

The codeword size n of the $[n, k, d]_q$ code.

Examples

```
In [1]: rs = galois.ReedSolomon(15, 9); rs
Out[1]: <Reed-Solomon Code: [15, 9, 7] over GF(2^4)>

In [2]: rs.n
Out[2]: 15
```

Type int

property roots

The $2t$ roots of the generator polynomial. These are consecutive powers of α, specifically $\alpha^c, \alpha^{c+1}, \dots, \alpha^{c+2t-1}$.

Examples

```
In [1]: rs = galois.ReedSolomon(15, 9); rs
Out[1]: <Reed-Solomon Code: [15, 9, 7] over GF(2^4)>

In [2]: rs.roots
Out[2]: GF([ 2,  4,  8,  3,  6, 12], order=2^4)

# Evaluate the generator polynomial at its roots
In [3]: rs.generator_poly(rs.roots)
Out[3]: GF([0, 0, 0, 0, 0, 0], order=2^4)
```

Type *galois.FieldArray*

property systematic

Indicates if the code is configured to return codewords in systematic form.

Examples

```
In [1]: rs = galois.ReedSolomon(15, 9); rs
Out[1]: <Reed-Solomon Code: [15, 9, 7] over GF(2^4)>

In [2]: rs.systematic
Out[2]: True
```

Type bool

property t

The error-correcting capability of the code. The code can correct t symbol errors in a codeword.

Examples

```
In [1]: rs = galois.ReedSolomon(15, 9); rs
Out[1]: <Reed-Solomon Code: [15, 9, 7] over GF(2^4)>

In [2]: rs.t
Out[2]: 3
```

Type int

7.3.2 Linear block code functions

generator_to_parity_check_matrix(G)	Converts the generator matrix $\mathbf{G}$ of a linear $[n, k]$ code into its parity-check matrix $\mathbf{H}$.
parity_check_to_generator_matrix(H)	Converts the parity-check matrix $\mathbf{H}$ of a linear $[n, k]$ code into its generator matrix $\mathbf{G}$.

galois.generator_to_parity_check_matrix

galois.**generator_to_parity_check_matrix**(G)

Converts the generator matrix $\mathbf{G}$ of a linear $[n, k]$ code into its parity-check matrix $\mathbf{H}$.

The generator and parity-check matrices satisfy the equations $\mathbf{G}\mathbf{H}^T = \mathbf{0}$.

Parameters **G** (galois.FieldArray) – The (k, n) generator matrix $\mathbf{G}$ in systematic form $\mathbf{G} = [\mathbf{I}_{k,k} \mid \mathbf{P}_{k,n-k}]$.

Returns The $(n-k, n)$ parity-check matrix $\mathbf{H} = [-\mathbf{P}_{k,n-k}^T \mid \mathbf{I}_{n-k,n-k}]$.

Return type *galois.FieldArray*

Examples

```
In [1]: g = galois.primitive_poly(2, 3); g
Out[1]: Poly(x^3 + x + 1, GF(2))

In [2]: G = galois.poly_to_generator_matrix(7, g); G
Out[2]:
GF([[1, 0, 0, 0, 1, 0, 1],
    [0, 1, 0, 0, 1, 1, 1],
    [0, 0, 1, 0, 1, 1, 0],
    [0, 0, 0, 1, 0, 1, 1]], order=2)

In [3]: H = galois.generator_to_parity_check_matrix(G); H
Out[3]:
GF([[1, 1, 1, 0, 1, 0, 0],
    [0, 1, 1, 1, 0, 1, 0],
    [1, 1, 0, 1, 0, 0, 1]], order=2)

In [4]: G @ H.T
Out[4]:
GF([[0, 0, 0],
    [0, 0, 0],
    [0, 0, 0],
    [0, 0, 0]], order=2)
```

galois.parity_check_to_generator_matrix

galois.**parity_check_to_generator_matrix**(H)

Converts the parity-check matrix $\mathbf{H}$ of a linear $[n, k]$ code into its generator matrix $\mathbf{G}$.

The generator and parity-check matrices satisfy the equations $\mathbf{G}\mathbf{H}^T = \mathbf{0}$.

Parameters `H` (`galois.FieldArray`) – The $(n-k, n)$ parity-check matrix $\mathbf{G}$ in systematic form $\mathbf{H} = [-\mathbf{P}_{k,n-k}^T \mid \mathbf{I}_{n-k,n-k}]$.

Returns The (k, n) generator matrix $\mathbf{G} = [\mathbf{I}_{k,k} \mid \mathbf{P}_{k,n-k}]$.

Return type *galois.FieldArray*

Examples

```
In [1]: g = galois.primitive_poly(2, 3); g
Out[1]: Poly(x^3 + x + 1, GF(2))

In [2]: G = galois.poly_to_generator_matrix(7, g); G
Out[2]:
GF([[1, 0, 0, 0, 1, 0, 1],
    [0, 1, 0, 0, 1, 1, 1],
    [0, 0, 1, 0, 1, 1, 0],
    [0, 0, 0, 1, 0, 1, 1]], order=2)

In [3]: H = galois.generator_to_parity_check_matrix(G); H
Out[3]:
```

(continues on next page)

(continued from previous page)

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

In [4]: G2 = galois.parity_check_to_generator_matrix(H); G2
Out[4]:
GF([[1, 0, 0, 0, 1, 0, 1],
    [0, 1, 0, 0, 1, 1, 1],
    [0, 0, 1, 0, 1, 1, 0],
    [0, 0, 0, 1, 0, 1, 1]], order=2)

In [5]: G2 @ H.T
Out[5]:
GF([[0, 0, 0],
    [0, 0, 0],
    [0, 0, 0],
    [0, 0, 0]], order=2)
```

7.3.3 Cyclic code functions

bch_valid_codes(n[, t_min])	Returns a list of (n, k, t) tuples of valid primitive binary BCH codes.
poly_to_generator_matrix(n, generator_poly)	Converts the generator polynomial $g(x)$ into the generator matrix $\mathbf{G}$ for an $[n, k]$ cyclic code.
roots_to_parity_check_matrix(n, roots)	Converts the generator polynomial roots into the parity-check matrix $\mathbf{H}$ for an $[n, k]$ cyclic code.

galois.bch_valid_codes

galois.**bch_valid_codes**(*n*, *t_min=1*)

Returns a list of (n, k, t) tuples of valid primitive binary BCH codes.

A BCH code with parameters (n, k, t) is represented as a $[n, k, d]_2$ linear block code with $d = 2t + 1$.

Parameters

- **n** (*int*) – The codeword size n, must be $n = 2^m - 1$.
- **t_min** (*int, optional*) – The minimum error-correcting capability. The default is 1.

Returns A list of (n, k, t) tuples of valid primitive BCH codes.

Return type list

References

- https://link.springer.com/content/pdf/bbm%3A978-1-4899-2174-1%2F1.pdf

Examples

```
In [1]: galois.bch_valid_codes(31)
Out[1]: [(31, 26, 1), (31, 21, 2), (31, 16, 3), (31, 11, 5), (31, 6, 7), (31, 1,␣
↪15)]

In [2]: galois.bch_valid_codes(31, t_min=3)
Out[2]: [(31, 16, 3), (31, 11, 5), (31, 6, 7), (31, 1, 15)]
```

galois.poly_to_generator_matrix

galois.**poly_to_generator_matrix**(*n*, *generator_poly*, *systematic=True*)

Converts the generator polynomial $g(x)$ into the generator matrix $\mathbf{G}$ for an $[n, k]$ cyclic code.

Parameters

- **n** (*int*) – The codeword size n.
- **generator_poly** (galois.Poly) – The generator polynomial $g(x)$.
- **systematic** (*bool, optional*) – Optionally specify if the encoding should be systematic, meaning the codeword is the message with parity appended. The default is True.

Returns The (k, n) generator matrix $\mathbf{G}$, such that given a message $\mathbf{m}$, a codeword is defined by $\mathbf{c} = \mathbf{m}\mathbf{G}$.

Return type *galois.FieldArray*

Examples

Compute the generator matrix for the $\mathrm{Hamming}(7, 4)$ code.

```
In [1]: g = galois.primitive_poly(2, 3); g
Out[1]: Poly(x^3 + x + 1, GF(2))

In [2]: galois.poly_to_generator_matrix(7, g, systematic=False)
Out[2]:
GF([[1, 0, 1, 1, 0, 0, 0],
    [0, 1, 0, 1, 1, 0, 0],
    [0, 0, 1, 0, 1, 1, 0],
    [0, 0, 0, 1, 0, 1, 1]], order=2)

In [3]: galois.poly_to_generator_matrix(7, g, systematic=True)
Out[3]:
GF([[1, 0, 0, 0, 1, 0, 1],
    [0, 1, 0, 0, 1, 1, 1],
    [0, 0, 1, 0, 1, 1, 0],
    [0, 0, 0, 1, 0, 1, 1]], order=2)
```

galois.roots_to_parity_check_matrix

galois.**roots_to_parity_check_matrix**(*n*, *roots*)

Converts the generator polynomial roots into the parity-check matrix $\mathbf{H}$ for an $[n, k]$ cyclic code.

Parameters

- **n** (*int*) – The codeword size n.
- **roots** (galois.FieldArray) – The $2t$ roots of the generator polynomial $g(x)$.

Returns The $(2t, n)$ parity-check matrix $\mathbf{H}$, such that given a codeword $\mathbf{c}$, the syndrome is defined by $\mathbf{s} = \mathbf{c}\mathbf{H}^T$.

Return type *galois.FieldArray*

Examples

Compute the parity-check matrix for the $\textrm{RS}(15, 9)$ code.

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

In [2]: alpha = GF.primitive_element

In [3]: t = 3

In [4]: roots = alpha**np.arange(1, 2*t + 1); roots
Out[4]: GF([ 2,  4,  8,  3,  6, 12], order=2^4)

In [5]: g = galois.Poly.Roots(roots); g
Out[5]: Poly(x^6 + 7x^5 + 9x^4 + 3x^3 + 12x^2 + 10x + 12, GF(2^4))

In [6]: galois.roots_to_parity_check_matrix(15, roots)
Out[6]:
GF([[ 9, 13, 15, 14,  7, 10,  5, 11, 12,  6,  3,  8,  4,  2,  1],
    [13, 14, 10, 11,  6,  8,  2,  9, 15,  7,  5, 12,  3,  4,  1],
    [15, 10, 12,  8,  1, 15, 10, 12,  8,  1, 15, 10, 12,  8,  1],
    [14, 11,  8,  9,  7, 12,  4, 13, 10,  6,  2, 15,  5,  3,  1],
    [ 7,  6,  1,  7,  6,  1,  7,  6,  1,  7,  6,  1,  7,  6,  1],
    [10,  8, 15, 12,  1, 10,  8, 15, 12,  1, 10,  8, 15, 12,  1]],
   order=2^4)
```

7.4 Linear Sequences

This section contains classes and functions for creating and analyzing linear sequences.

7.4.1 Linear-feedback shift registers

LFSR(poly[, state, config])	A linear-feedback shift register (LFSR).

galois.LFSR

class galois.**LFSR**(*poly*, *state=1*, *config='fibonacci'*)

A linear-feedback shift register (LFSR).

This class implements an LFSR in either the Fibonacci or Galois configuration. An LFSR is defined by its generator polynomial $g(x) = g_n x^n + \dots + g_1 x + g_0$ and initial state vector $s = [s_{n-1}, \dots, s_1, s_0]$. Below are diagrams for a degree-3 LFSR in the Fibonacci and Galois configuration. The generator polynomial is $g(x) = g_3 x^3 + g_2 x^2 + g_1 x + g_0$ and state vector is $s = [s_2, s_1, s_0]$.

Listing 1: Fibonacci LFSR Configuration

```
     ____________________________________
    |                                    |
g0 -      g1 -       g2 -       g3 -
    |     |    |    |
    └─  s2  ───  s1  ───  s0  ────  y[n]
```

In the Fibonacci configuration, at time instant i the next $n-1$ outputs are the current state reversed, that is $[y_i, y_{i+1}, \dots, y_{i+n-1}] = [s_0, s_1, \dots, s_{n-1}]$. And the n-th output is a linear combination of the current state and the generator polynomial $y_{i+n} = (g_n s_0 + g_{n-1} s_1 + \dots + g_1 s_{n-1}) g_0$.

Listing 2: Galois LFSR Configuration

```
     ______________________________________
g0 -      g1 -       g2 -       g3 -
    |            |             |             |
    |            |
    └─  s0  ───  s1  ───  s2  ────  y[n]
```

In the Galois configuration, the next output is $y = s_{n-1}$ and the next state is computed by $s_k = s_{n-1} g_n g_k + s_{k-1}$. In the case of s_0 there is no previous state added.

References

- https://core.ac.uk/download/pdf/288371609.pdf
- https://www.wseas.org/multimedia/journals/control/2018/a945903-022.pdf
- https://jhafranco.com/2014/02/15/n-ary-m-sequence-generator-in-python/
- https://www.cs.uky.edu/~klapper/pdf/galois.pdf

Constructors

__init__(poly[, state, config])	Constructs a linear-feedback shift register.

Methods

reset()	Resets the LFSR state to the initial state.
step([steps])	Steps the LFSR and produces `steps` output symbols.

Attributes

config	The LFSR configuration, either `"fibonacci"` or `"galois"`.
field	The Galois field that defines the LFSR arithmetic.
initial_state	The initial state vector $s = [s_{n-1}, \dots, s_1, s_0]$.
poly	The generator polynomial $g(x) = g_n x^n + \dots + g_1 x + g_0$.
state	The current state vector $s = [s_{n-1}, \dots, s_1, s_0]$.

__init__(*poly*, *state=1*, *config='fibonacci'*)

Constructs a linear-feedback shift register.

Parameters

- **poly** (`galois.Poly`) – The generator polynomial $g(x) = g_n x^n + \dots + g_1 x + g_0$.
- **state** (*int, tuple, list, numpy.ndarray,* `galois.FieldArray`*, optional*) – The initial state vector $s = [s_{n-1}, \dots, s_1, s_0]$. If specified as an integer, then s_{n-1} is interpreted as the MSB and s_0 as the LSB. The default is 1 which corresponds to $s = [0, \dots, 0, 1]$.
- **config** (*str, optional*) – A string indicating the LFSR feedback configuration, either `"fibonacci"` (default) or `"galois"`.

Returns A linear-feedback shift register object.

Return type *galois.LFSR*

reset()

Resets the LFSR state to the initial state.

Examples

```
In [1]: lfsr = galois.LFSR(galois.primitive_poly(2, 4)); lfsr
Out[1]: <Fibonacci LFSR: poly=Poly(x^4 + x + 1, GF(2))>

In [2]: lfsr.state
Out[2]: GF([0, 0, 0, 1], order=2)

In [3]: lfsr.step(10)
Out[3]: GF([1, 0, 0, 0, 1, 1, 1, 1, 0, 1], order=2)
```

(continues on next page)

(continued from previous page)

```
In [4]: lfsr.state
Out[4]: GF([0, 1, 1, 0], order=2)

In [5]: lfsr.reset()

In [6]: lfsr.state
Out[6]: GF([0, 0, 0, 1], order=2)
```

step(*steps=1*)

Steps the LFSR and produces `steps` output symbols.

Parameters **steps** (*int, optional*) – The number of output symbols to produce. The default is 1.

Returns An array of output symbols of type *field* with size `steps`.

Return type *galois.FieldArray*

Examples

Step the LFSR one output at a time.

```
In [1]: lfsr = galois.LFSR(galois.primitive_poly(2, 4)); lfsr
Out[1]: <Fibonacci LFSR: poly=Poly(x^4 + x + 1, GF(2))>

In [2]: lfsr.state
Out[2]: GF([0, 0, 0, 1], order=2)

In [3]: lfsr.state, lfsr.step()
Out[3]: (GF([0, 0, 0, 1], order=2), GF(1, order=2))

In [4]: lfsr.state, lfsr.step()
Out[4]: (GF([1, 0, 0, 0], order=2), GF(0, order=2))

In [5]: lfsr.state, lfsr.step()
Out[5]: (GF([1, 1, 0, 0], order=2), GF(0, order=2))

In [6]: lfsr.state, lfsr.step()
Out[6]: (GF([1, 1, 1, 0], order=2), GF(0, order=2))
```

Step the LFSR 10 steps.

```
In [7]: lfsr.reset()

In [8]: lfsr.step(10)
Out[8]: GF([1, 0, 0, 0, 1, 1, 1, 1, 0, 1], order=2)
```

property config

The LFSR configuration, either `"fibonacci"` or `"galois"`. See the Notes section of *LFSR* for descriptions of the two configurations.

Examples

```
In [1]: lfsr = galois.LFSR(galois.primitive_poly(2, 4)); lfsr
Out[1]: <Fibonacci LFSR: poly=Poly(x^4 + x + 1, GF(2))>

In [2]: lfsr.config
Out[2]: 'fibonacci'
```

```
In [3]: lfsr = galois.LFSR(galois.primitive_poly(2, 4), config="galois"); lfsr
Out[3]: <Galois LFSR: poly=Poly(x^4 + x + 1, GF(2))>

In [4]: lfsr.config
Out[4]: 'galois'
```

Type str

property field

The Galois field that defines the LFSR arithmetic. The generator polynomial $g(x)$ is over this field and the state vector contains values in this field.

Examples

```
In [1]: lfsr = galois.LFSR(galois.primitive_poly(2, 4)); lfsr
Out[1]: <Fibonacci LFSR: poly=Poly(x^4 + x + 1, GF(2))>

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

In [3]: print(lfsr.field.properties)
GF(2):
  characteristic: 2
  degree: 1
  order: 2
  irreducible_poly: x + 1
  is_primitive_poly: True
  primitive_element: 1
```

Type *galois.FieldClass*

property initial_state

The initial state vector $s = [s_{n-1}, \dots, s_1, s_0]$.

Examples

```
In [1]: lfsr = galois.LFSR(galois.primitive_poly(2, 4)); lfsr
Out[1]: <Fibonacci LFSR: poly=Poly(x^4 + x + 1, GF(2))>

In [2]: lfsr.initial_state
Out[2]: GF([0, 0, 0, 1], order=2)
```

Type *galois.FieldArray*

property poly

The generator polynomial $g(x) = g_n x^n + \cdots + g_1 x + g_0$.

Examples

```
In [1]: lfsr = galois.LFSR(galois.primitive_poly(2, 4)); lfsr
Out[1]: <Fibonacci LFSR: poly=Poly(x^4 + x + 1, GF(2))>

In [2]: lfsr.poly
Out[2]: Poly(x^4 + x + 1, GF(2))
```

Type *galois.Poly*

property state

The current state vector $s = [s_{n-1}, \dots, s_1, s_0]$.

Examples

```
In [1]: lfsr = galois.LFSR(galois.primitive_poly(2, 4)); lfsr
Out[1]: <Fibonacci LFSR: poly=Poly(x^4 + x + 1, GF(2))>

In [2]: lfsr.state
Out[2]: GF([0, 0, 0, 1], order=2)

In [3]: lfsr.step(10)
Out[3]: GF([1, 0, 0, 0, 1, 1, 1, 1, 0, 1], order=2)

In [4]: lfsr.state
Out[4]: GF([0, 1, 1, 0], order=2)
```

Type *galois.FieldArray*

7.4.2 Sequence analysis functions

berlekamp_massey(sequence[, config, state])	Finds the minimum-degree polynomial $c(x)$ that produces the sequence in $\mathrm{GF}(p^m)$.

galois.berlekamp_massey

galois.**berlekamp_massey**(*sequence*, *config='fibonacci'*, *state=False*)

Finds the minimum-degree polynomial $c(x)$ that produces the sequence in $\mathrm{GF}(p^m)$.

This function implements the Berlekamp-Massey algorithm.

Parameters

- **sequence** (galois.FieldArray) – A 1-D sequence of Galois field elements in $\mathrm{GF}(p^m)$.
- **config** (*str, optional*) – A string indicating the LFSR feedback configuration for the returned connection polynomial, either `"fibonacci"` (default) or `"galois"`. See the LFSR configurations in *galois.LFSR*. The LFSR configuration will indicate if the connection polynomial coefficients should be reversed or not.
- **state** (*bool, optional*) – Indicates whether to return the LFSR initial state such that the output is the input sequence. The default is `False`.

Returns

- *galois.Poly* – The minimum-degree polynomial $c(x) \in \mathrm{GF}(p^m)[x]$ that produces the input sequence.
- *galois.FieldArray* – The initial state of the LFSR such that the output will generate the input sequence. Only returned if `state=True`.

References

- https://crypto.stanford.edu/~mironov/cs359/massey.pdf
- https://www.embeddedrelated.com/showarticle/1099.php

Examples

The Berlekamp-Massey algorithm requires $2n$ output symbols to determine the n-degree minimum connection polynomial.

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

In [2]: lfsr = galois.LFSR(g, state=1); lfsr
Out[2]: <Fibonacci LFSR: poly=Poly(x^8 + x^4 + x^3 + x^2 + 1, GF(2))>

In [3]: s = lfsr.step(16); s
Out[3]: GF([1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 1], order=2)

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

7.5 Number Theory

This section contains functions for performing modular arithmetic and other number theoretic routines.

7.5.1 Divisibility

`gcd`(a, b)	Finds the greatest common divisor of a and b.
`egcd`(a, b)	Finds the multiplicands of a and b such that $as + bt = \gcd(a, b)$.
`lcm`(*values)	Computes the least common multiple of the arguments.
`prod`(*values)	Computes the product of the arguments.
`euler_phi`(n)	Counts the positive integers (totatives) in $[1, n]$ that are coprime to n.
`totatives`(n)	Returns the positive integers (totatives) in $[1, n]$ that are coprime to n.
`are_coprime`(*values)	Determines if the arguments are pairwise coprime.

galois.euler_phi

`galois.euler_phi(n)`

Counts the positive integers (totatives) in $[1, n]$ that are coprime to n.

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

Returns The number of totatives that are coprime to n.

Return type int

Notes

This function implements the Euler totient function

$$\phi(n) = n \prod_{p\ |\ n} \left(1 - \frac{1}{p}\right) = \prod_{i=1}^{k} p_i^{e_i - 1}\big(p_i - 1\big)$$

for prime p and the prime factorization $n = p_1^{e_1} \dots p_k^{e_k}$.

References

- Section 2.4.1 from https://cacr.uwaterloo.ca/hac/about/chap2.pdf
- https://oeis.org/A000010

Examples

```
In [1]: n = 20

In [2]: phi = galois.euler_phi(n); phi
Out[2]: 8

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

# The number of totatives is phi
In [4]: len(totatives) == phi
Out[4]: True

# For prime n, (n) = n - 1
In [5]: galois.euler_phi(13)
Out[5]: 12
```

galois.totatives

`galois.totatives(n)`

Returns the positive integers (totatives) in $[1, n]$ that are coprime to n.

The totatives of n form the multiplicative group $(\mathbb{Z}/n\mathbb{Z})^{\times}$.

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

Returns The totatives of n.

Return type list

References

- Section 2.4.3 from https://cacr.uwaterloo.ca/hac/about/chap2.pdf
- https://oeis.org/A000010

Examples

```
In [1]: n = 20

In [2]: totatives = galois.totatives(n); totatives
Out[2]: [1, 3, 7, 9, 11, 13, 17, 19]

In [3]: phi = galois.euler_phi(n); phi
Out[3]: 8

In [4]: len(totatives) == phi
Out[4]: True
```

7.5.2 Congruences

pow(base, exponent, modulus)	Efficiently performs modular exponentiation.
crt(remainders, moduli)	Solves the simultaneous system of congruences for x.
primitive_root(n[, start, stop, reverse])	Finds the smallest primitive root modulo n.
primitive_roots(n[, start, stop, reverse])	Finds all primitive roots modulo n.
carmichael_lambda(n)	Finds the smallest positive integer m such that $a^m \equiv 1 \pmod n$ for every integer a in $[1, n]$ that is coprime to n.
legendre_symbol(a, p)	Computes the Legendre symbol $(\frac{a}{p})$.
jacobi_symbol(a, n)	Computes the Jacobi symbol $(\frac{a}{n})$.
kronecker_symbol(a, n)	Computes the Kronecker symbol $(\frac{a}{n})$.
is_primitive_root(g, n)	Determines if g is a primitive root modulo n.
is_cyclic(n)	Determines whether the multiplicative group $(\mathbb{Z}/n\mathbb{Z})^\times$ is cyclic.

galois.carmichael_lambda

galois.**carmichael_lambda**(*n*)

Finds the smallest positive integer m such that $a^m \equiv 1 \pmod n$ for every integer a in $[1, n]$ that is coprime to n.

This function implements the Carmichael function $\lambda(n)$.

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

Returns The smallest positive integer m such that $a^m \equiv 1 (\bmod n)$ for every a in $[1, n]$ that is coprime to n.

Return type int

References

- https://oeis.org/A002322

Examples

The Carmichael lambda function and Euler totient function are often equal. However, there are notable exceptions.

```
In [1]: [galois.euler_phi(n) for n in range(1, 20)]
Out[1]: [1, 1, 2, 2, 4, 2, 6, 4, 6, 4, 10, 4, 12, 6, 8, 8, 16, 6, 18]

In [2]: [galois.carmichael_lambda(n) for n in range(1, 20)]
Out[2]: [1, 1, 2, 2, 4, 2, 6, 2, 6, 4, 10, 2, 12, 6, 4, 4, 16, 6, 18]
```

For prime n, $\phi(n) = \lambda(n) = n - 1$. And for most composite n, $\phi(n) = \lambda(n) < n - 1$.

```
In [3]: n = 9

In [4]: phi = galois.euler_phi(n); phi
Out[4]: 6
```

(continues on next page)

(continued from previous page)

```
In [5]: lambda_ = galois.carmichael_lambda(n); lambda_
Out[5]: 6

In [6]: totatives = galois.totatives(n); totatives
Out[6]: [1, 2, 4, 5, 7, 8]

In [7]: for power in range(1, phi + 1):
   ...:     y = [pow(a, power, n) for a in totatives]
   ...:     print("Power {}: {} (mod {})".format(power, y, n))
   ...:
Power 1: [1, 2, 4, 5, 7, 8] (mod 9)
Power 2: [1, 4, 7, 7, 4, 1] (mod 9)
Power 3: [1, 8, 1, 8, 1, 8] (mod 9)
Power 4: [1, 7, 4, 4, 7, 1] (mod 9)
Power 5: [1, 5, 7, 2, 4, 8] (mod 9)
Power 6: [1, 1, 1, 1, 1, 1] (mod 9)

In [8]: galois.is_cyclic(n)
Out[8]: True
```

When $\phi(n) \neq \lambda(n)$, the multiplicative group $(\mathbb{Z}/n\mathbb{Z})^\times$ is not cyclic. See `galois.is_cyclic()`.

```
In [9]: n = 8

In [10]: phi = galois.euler_phi(n); phi
Out[10]: 4

In [11]: lambda_ = galois.carmichael_lambda(n); lambda_
Out[11]: 2

In [12]: totatives = galois.totatives(n); totatives
Out[12]: [1, 3, 5, 7]

In [13]: for power in range(1, phi + 1):
   ....:     y = [pow(a, power, n) for a in totatives]
   ....:     print("Power {}: {} (mod {})".format(power, y, n))
   ....:
Power 1: [1, 3, 5, 7] (mod 8)
Power 2: [1, 1, 1, 1] (mod 8)
Power 3: [1, 3, 5, 7] (mod 8)
Power 4: [1, 1, 1, 1] (mod 8)

In [14]: galois.is_cyclic(n)
Out[14]: False
```

galois.legendre_symbol

galois.**legendre_symbol**(a, p)

Computes the Legendre symbol $(\frac{a}{p})$.

Parameters

- **a** (*int*) – An integer.
- **p** (*int*) – An odd prime $p \geq 3$.

Returns The Legendre symbol $(\frac{a}{p})$ with value in $\{0, 1, -1\}$.

Return type int

Notes

The Legendre symbol is useful for determining if a is a quadratic residue modulo p, namely $a \in Q_p$. A quadratic residue a modulo p satisfies $x^2 \equiv a \pmod{p}$ for some x.

$$\left(\frac{a}{p}\right) = \begin{cases} 0, & p \mid a \\ 1, & a \in Q_p \\ -1, & a \in \overline{Q}_p \end{cases}$$

References

- Algorithm 2.149 from https://cacr.uwaterloo.ca/hac/about/chap2.pdf

Examples

The quadratic residues modulo 7 are $Q_7 = \{1, 2, 4\}$. The quadratic non-residues modulo 7 are $\overline{Q}_7 = \{3, 5, 6\}$.

```
In [1]: [pow(x, 2, 7) for x in range(7)]
Out[1]: [0, 1, 4, 2, 2, 4, 1]

In [2]: for a in range(7):
   ...:     print(f"({a} / 7) = {galois.legendre_symbol(a, 7)}")
   ...:
(0 / 7) = 0
(1 / 7) = 1
(2 / 7) = 1
(3 / 7) = -1
(4 / 7) = 1
(5 / 7) = -1
(6 / 7) = -1
```

galois.jacobi_symbol

`galois.jacobi_symbol(a, n)`

Computes the Jacobi symbol $(\frac{a}{n})$.

Parameters

- **a** (*int*) – An integer.
- **n** (*int*) – An odd integer $n \geq 3$.

Returns The Jacobi symbol $(\frac{a}{n})$ with value in $\{0, 1, -1\}$.

Return type int

Notes

The Jacobi symbol extends the Legendre symbol for odd $n \geq 3$. Unlike the Legendre symbol, $(\frac{a}{n}) = 1$ does not imply a is a quadratic residue modulo n. However, all $a \in Q_n$ have $(\frac{a}{n}) = 1$.

References

- Algorithm 2.149 from https://cacr.uwaterloo.ca/hac/about/chap2.pdf

Examples

The quadratic residues modulo 9 are $Q_9 = \{1, 4, 7\}$ and these all satisfy $(\frac{a}{9}) = 1$. The quadratic non-residues modulo 9 are $\overline{Q}_9 = \{2, 3, 5, 6, 8\}$, but notice $\{2, 5, 8\}$ also satisfy $(\frac{a}{9}) = 1$. The set of integers $\{3, 6\}$ not coprime to 9 satisfies $(\frac{a}{9}) = 0$.

```
In [1]: [pow(x, 2, 9) for x in range(9)]
Out[1]: [0, 1, 4, 0, 7, 7, 0, 4, 1]

In [2]: for a in range(9):
   ...:     print(f"({a} / 9) = {galois.jacobi_symbol(a, 9)}")
   ...:
(0 / 9) = 0
(1 / 9) = 1
(2 / 9) = 1
(3 / 9) = 0
(4 / 9) = 1
(5 / 9) = 1
(6 / 9) = 0
(7 / 9) = 1
(8 / 9) = 1
```

galois.kronecker_symbol

galois.**kronecker_symbol**(a, n)

Computes the Kronecker symbol $(\frac{a}{n})$.

The Kronecker symbol extends the Jacobi symbol for all n.

Parameters

- **a** (*int*) – An integer.
- **n** (*int*) – An integer.

Returns The Kronecker symbol $(\frac{a}{n})$ with value in $\{0, -1, 1\}$.

Return type int

References

- Algorithm 2.149 from https://cacr.uwaterloo.ca/hac/about/chap2.pdf

galois.is_cyclic

galois.**is_cyclic**(n)

Determines whether the multiplicative group $(\mathbb{Z}/n\mathbb{Z})^\times$ is cyclic.

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

Returns `True` if the multiplicative group $(\mathbb{Z}/n\mathbb{Z})^\times$ is cyclic.

Return type bool

Notes

The multiplicative group $(\mathbb{Z}/n\mathbb{Z})^\times$ is the set of positive integers $1 \le a < n$ that are coprime with n. $(\mathbb{Z}/n\mathbb{Z})^\times$ being cyclic means that some primitive root of n, or generator, g can generate the group $\{g^0, g^1, g^2, \dots, g^{\phi(n)-1}\}$, where $\phi(n)$ is Euler's totient function and calculates the order of the group. If $(\mathbb{Z}/n\mathbb{Z})^\times$ is cyclic, the number of primitive roots is found by $\phi(\phi(n))$.

$(\mathbb{Z}/n\mathbb{Z})^\times$ is *cyclic* if and only if n is 2, 4, p^k, or $2p^k$, where p is an odd prime and k is a positive integer.

Examples

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

```
# n is of type 2*p^e, which is cyclic
In [1]: n = 14

In [2]: galois.is_cyclic(n)
Out[2]: True

In [3]: Znx = set(galois.totatives(n)); Znx
Out[3]: {1, 3, 5, 9, 11, 13}
```

(continues on next page)

(continued from previous page)

```
In [4]: phi = galois.euler_phi(n); phi
Out[4]: 6

In [5]: len(Znx) == phi
Out[5]: True

# The primitive roots are the elements in Znx that multiplicatively generate the␣
↪group
In [6]: for a in Znx:
   ...:     span = set([pow(a, i, n) for i in range(1, phi + 1)])
   ...:     primitive_root = galois.is_primitive_root(a, n)
   ...:     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

# Find the smallest primitive root
In [7]: galois.primitive_root(n)
Out[7]: 3

# Find all primitive roots
In [8]: roots = galois.primitive_roots(n); roots
Out[8]: [3, 5]

# Euler's totient function ((n)) counts the primitive roots of n
In [9]: len(roots) == galois.euler_phi(phi)
Out[9]: True
```

A counterexample is $n = 15 = 3 \cdot 5$, which doesn't fit the condition for cyclicness. $(\mathbb{Z}/15\mathbb{Z})^\times = \{1, 2, 4, 7, 8, 11, 13, 14\}$. Since the group is not cyclic, it has no primitive roots.

```
# n is of type p1^e1 * p2^e2, which is not cyclic
In [10]: n = 15

In [11]: galois.is_cyclic(n)
Out[11]: False

In [12]: Znx = set(galois.totatives(n)); Znx
Out[12]: {1, 2, 4, 7, 8, 11, 13, 14}

In [13]: phi = galois.euler_phi(n); phi
Out[13]: 8

In [14]: len(Znx) == phi
Out[14]: True

# The primitive roots are the elements in Znx that multiplicatively generate the␣
↪group
```

(continues on next page)

(continued from previous page)

```
In [15]: for a in Znx:
   ....:     span = set([pow(a, i, n) for i in range(1, phi + 1)])
   ....:     primitive_root = galois.is_primitive_root(a, n)
   ....:     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

# Find the smallest primitive root
In [16]: galois.primitive_root(n)

# Find all primitive roots
In [17]: roots = galois.primitive_roots(n); roots
Out[17]: []

# Note the max order of any element is 4, not 8, which is Carmichael's lambda␣
↪function
In [18]: galois.carmichael_lambda(n)
Out[18]: 4
```

For prime n, a primitive root modulo n is also a primitive element of the Galois field $\mathrm{GF}(n)$. A primitive element is a generator of the multiplicative group $\mathrm{GF}(p)^\times = \{1, 2, \dots, p-1\} = \{g^0, g^1, g^2, \dots, g^{\phi(n)-1}\}$.

```
# n is of type p, which is cyclic
In [19]: n = 7

In [20]: galois.is_cyclic(n)
Out[20]: True

In [21]: Znx = set(galois.totatives(n)); Znx
Out[21]: {1, 2, 3, 4, 5, 6}

In [22]: phi = galois.euler_phi(n); phi
Out[22]: 6

In [23]: len(Znx) == phi
Out[23]: True

# The primitive roots are the elements in Znx that multiplicatively generate the␣
↪group
In [24]: for a in Znx:
   ....:     span = set([pow(a, i, n) for i in range(1, phi + 1)])
   ....:     primitive_root = galois.is_primitive_root(a, n)
   ....:     print("Element: {:2d}, Span: {:<18}, Primitive root: {}".format(a,␣
↪str(span), primitive_root))
```

(continues on next page)

(continued from previous page)

```
   ....:
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

# Find the smallest primitive root
In [25]: galois.primitive_root(n)
Out[25]: 3

# Find all primitive roots
In [26]: roots = galois.primitive_roots(n); roots
Out[26]: [3, 5]

# Euler's totient function ((n)) counts the primitive roots of n
In [27]: len(roots) == galois.euler_phi(phi)
Out[27]: True
```

7.5.3 Integer arithmetic

isqrt(n)	Computes $x = \lfloor\sqrt{n}\rfloor$ such that $x^2 \le n < (x+1)^2$.
iroot(n, k)	Computes $x = \lfloor n^{\frac{1}{k}}\rfloor$ such that $x^k \le n < (x+1)^k$.
ilog(n, b)	Computes $x = \lfloor \log_b(n) \rfloor$ such that $b^x \le n < b^{x+1}$.

galois.isqrt

galois.**isqrt**(*n*)

Computes $x = \lfloor\sqrt{n}\rfloor$ such that $x^2 \le n < (x+1)^2$.

Note: This function is included for Python versions before 3.8. For Python 3.8 and later, this function calls `math.isqrt()` from the standard library.

Parameters **n** (*int*) – A non-negative integer.

Returns The integer square root of n.

Return type int

Examples

```
In [1]: n = 1000

In [2]: x = galois.isqrt(n); x
Out[2]: 31

In [3]: print(f"{x**2} <= {n} < {(x + 1)**2}")
961 <= 1000 < 1024
```

galois.iroot

galois.**iroot**(n, k)

Computes $x = \lfloor n^{\frac{1}{k}} \rfloor$ such that $x^k \leq n < (x+1)^k$.

Parameters

- **n** (*int*) – A non-negative integer.
- **k** (*int*) – The root k, must be at least 2.

Returns The integer k-th root of n.

Return type int

Examples

```
In [1]: n = 1000

In [2]: x = galois.iroot(n, 5); x
Out[2]: 3

In [3]: print(f"{x**5} <= {n} < {(x + 1)**5}")
243 <= 1000 < 1024
```

galois.ilog

galois.**ilog**(n, b)

Computes $x = \lfloor \log_b(n) \rfloor$ such that $b^x \leq n < b^{x+1}$.

Parameters

- **n** (*int*) – A positive integer.
- **b** (*int*) – The logarithm base b.

Returns The integer logarithm base b of n.

Return type int

Examples

```
In [1]: n = 1000

In [2]: x = galois.ilog(n, 5); x
Out[2]: 4

In [3]: print(f"{5**x} <= {n} < {5**(x + 1)}")
625 <= 1000 < 3125
```

7.6 Integer Factorization

This section contains functions for factoring integers and analyzing their properties.

7.6.1 Prime factorization

factors(value)	Computes the prime factors of a positive integer or the irreducible factors of a non-constant, monic polynomial.

7.6.2 Composite factorization

divisors(n)	Computes all positive integer divisors d of the integer n such that $d \mid n$.
divisor_sigma(n[, k])	Returns the sum of k-th powers of the positive divisors of n.

galois.divisors

galois.**divisors**(*n*)

Computes all positive integer divisors d of the integer n such that $d \mid n$.

Parameters **n** (*int*) – Any integer.

Returns Sorted list of positive integer divisors d.

Return type list

Notes

galois.divisors() find *all* positive integer divisors or factors of n, where *galois.factors()* only finds the prime factors of n.

Examples

```
In [1]: galois.divisors(0)
Out[1]: []

In [2]: galois.divisors(1)
Out[2]: [1]

In [3]: galois.divisors(24)
Out[3]: [1, 2, 3, 4, 6, 8, 12, 24]

In [4]: galois.divisors(-24)
Out[4]: [1, 2, 3, 4, 6, 8, 12, 24]

In [5]: galois.factors(24)
Out[5]: ([2, 3], [3, 1])
```

galois.divisor_sigma

galois.**divisor_sigma**(*n*, *k=1*)

Returns the sum of k-th powers of the positive divisors of n.

Parameters

- **n** (*int*) – Any integer.
- **k** (*int, optional*) – The degree of the positive divisors. The default is 1 which corresponds to $\sigma_1(n)$ which is the sum of positive divisors.

Returns The sum of divisors function $\sigma_k(n)$.

Return type int

Notes

This function implements the $\sigma_k(n)$ function. It is defined as:

$$\sigma_k(n) = \sum_{d\ |\ n} d^k$$

Examples

```
In [1]: galois.divisors(9)
Out[1]: [1, 3, 9]

In [2]: galois.divisor_sigma(9, k=0)
Out[2]: 3

In [3]: galois.divisor_sigma(9, k=1)
Out[3]: 13

In [4]: galois.divisor_sigma(9, k=2)
Out[4]: 91
```

7.6.3 Specific factorization algorithms

perfect_power(n)	Returns the integer base $c > 1$ and exponent $e > 1$ of $n = c^e$ if n is a perfect power.
trial_division(n[, B])	Finds all the prime factors $p_i^{e_i}$ of n for $p_i \le B$.
pollard_p1(n, B[, B2])	Attempts to find a non-trivial factor of n if it has a prime factor p such that $p - 1$ is B-smooth.
pollard_rho(n[, c])	Attempts to find a non-trivial factor of n using cycle detection.

galois.perfect_power

galois.**perfect_power**(*n*)

Returns the integer base $c > 1$ and exponent $e > 1$ of $n = c^e$ if n is a perfect power.

Parameters **n** (*int*) – A positive integer $n > 1$.

Returns None is n is not a perfect power. Otherwise, (c, e) such that $n = c^e$. c may be composite.

Return type None, tuple

Examples

```
# Primes are not perfect powers
In [1]: galois.perfect_power(5)

# Products of primes are not perfect powers
In [2]: galois.perfect_power(2*3)

# Products of prime powers were the GCD of the exponents is 1 are not perfect powers
In [3]: galois.perfect_power(2 * 3 * 5**3)

# Products of prime powers were the GCD of the exponents is > 1 are perfect powers
In [4]: galois.perfect_power(2**2 * 3**2 * 5**4)
Out[4]: (150, 2)

In [5]: galois.perfect_power(36)
Out[5]: (6, 2)

In [6]: galois.perfect_power(125)
Out[6]: (5, 3)
```

galois.trial_division

galois.**trial_division**(*n*, *B=None*)

Finds all the prime factors $p_i^{e_i}$ of n for $p_i \leq B$.

The trial division factorization will find all prime factors $p_i \leq B$ such that n factors as $n = p_1^{e_1} \dots p_k^{e_k} n_r$ where n_r is a residual factor (which may be composite).

Parameters

- **n** (*int*) – A positive integer.
- **B** (*int, optional*) – The max divisor in the trial division. The default is `None` which corresponds to $B = \sqrt{n}$. If $B > \sqrt{n}$, the algorithm will only search up to $\sqrt{n}$, since a prime factor of n cannot be larger than $\sqrt{n}$.

Returns

- *list* – The discovered prime factors $\{p_1, \dots, p_k\}$.
- *list* – The corresponding prime exponents $\{e_1, \dots, e_k\}$.
- *int* – The residual factor n_r.

Examples

```
In [1]: n = 2**4 * 17**3 * 113 * 15013

In [2]: galois.trial_division(n)
Out[2]: ([2, 17, 113, 15013], [4, 3, 1, 1], 1)

In [3]: galois.trial_division(n, B=500)
Out[3]: ([2, 17, 113], [4, 3, 1], 15013)

In [4]: galois.trial_division(n, B=100)
Out[4]: ([2, 17], [4, 3], 1696469)
```

galois.pollard_p1

galois.**pollard_p1**(*n*, *B*, *B2=None*)

Attempts to find a non-trivial factor of n if it has a prime factor p such that $p-1$ is B-smooth.

For a given odd composite n with a prime factor p, Pollard's $p-1$ algorithm can discover a non-trivial factor of n if $p-1$ is B-smooth. Specifically, the prime factorization must satisfy $p - 1 = p_1^{e_1} \dots p_k^{e_k}$ with each $p_i \leq B$.

A extension of Pollard's $p-1$ algorithm allows a prime factor p to be B-smooth with the exception of one prime factor $B < p_{k+1} \leq B_2$. In this case, the prime factorization is $p - 1 = p_1^{e_1} \dots p_k^{e_k} p_{k+1}$. Often B_2 is chosen such that $B_2 \gg B$.

Parameters

- **n** (*int*) – An odd composite integer $n > 2$ that is not a prime power.
- **B** (*int*) – The smoothness bound $B > 2$.
- **B2** (*int, optional*) – The smoothness bound B_2 for the optional second step of the algorithm. The default is `None`, which will not perform the second step.

Returns A non-trivial factor of n, if found. `None` if not found.

Return type None, int

References

- Section 3.2.3 from https://cacr.uwaterloo.ca/hac/about/chap3.pdf

Examples

Here, $n = pq$ where $p - 1$ is 1039-smooth and $q - 1$ is 17-smooth.

```
In [1]: p, q = 1458757, 1326001

In [2]: galois.factors(p - 1)
Out[2]: ([2, 3, 13, 1039], [2, 3, 1, 1])

In [3]: galois.factors(q - 1)
Out[3]: ([2, 3, 5, 13, 17], [4, 1, 3, 1, 1])
```

Searching with $B = 15$ will not recover a prime factor.

```
In [4]: galois.pollard_p1(p*q, 15)
```

Searching with $B = 17$ will recover the prime factor q.

```
In [5]: galois.pollard_p1(p*q, 17)
Out[5]: 1326001
```

Searching $B = 15$ will not recover a prime factor in the first step, but will find q in the second step because $p_{k+1} = 17$ satisfies $15 < 17 \leq 100$.

```
In [6]: galois.pollard_p1(p*q, 15, B2=100)
Out[6]: 1326001
```

Pollard's $p - 1$ algorithm may return a composite factor.

```
In [7]: n = 2133861346249

In [8]: galois.factors(n)
Out[8]: ([37, 41, 5471, 257107], [1, 1, 1, 1])

In [9]: galois.pollard_p1(n, 10)
Out[9]: 1517

In [10]: 37*41
Out[10]: 1517
```

galois.pollard_rho

`galois.pollard_rho(n, c=1)`

Attempts to find a non-trivial factor of n using cycle detection.

Pollard's ρ algorithm seeks to find a non-trivial factor of n by finding a cycle in a sequence of integers $x_0, x_1, \dots$ defined by $x_i = f(x_{i-1}) = x_{i-1}^2 + 1 \mod p$ where p is an unknown small prime factor of n. This happens when $x_m \equiv x_{2m} \pmod p$. Because p is unknown, this is accomplished by computing the sequence modulo n and looking for $\gcd(x_m - x_{2m}, n) > 1$.

Parameters

- **n** (*int*) – An odd composite integer $n > 2$ that is not a prime power.
- **c** (*int, optional*) – The constant offset in the function $f(x) = x^2 + c \mod n$. The default is 1. A requirement of the algorithm is that $c \notin \{0, -2\}$.

Returns A non-trivial factor m of n, if found. `None` if not found.

Return type None, int

References

- Section 3.2.2 from https://cacr.uwaterloo.ca/hac/about/chap3.pdf

Examples

Pollard's ρ is especially good at finding small factors.

```
In [1]: n = 503**7 * 10007 * 1000003

In [2]: galois.pollard_rho(n)
Out[2]: 503
```

It is also efficient for finding relatively small factors.

```
In [3]: n = 1182640843 * 1716279751

In [4]: galois.pollard_rho(n)
Out[4]: 1716279751
```

7.6.4 Integer tests

is_prime(n)	Determines if n is prime.
is_prime_power(n)	Determines if n is a prime power $n = p^k$ for prime p and $k \ge 1$.
is_composite(n)	Determines if n is composite.
is_perfect_power(n)	Determines if n is a perfect power $n = x^k$ for $x > 0$ and $k \ge 2$.
is_square_free(value)	Determines if the positive integer or the non-constant, monic polynomial is square-free.
is_smooth(n, B)	Determines if the positive integer n is B-smooth.
is_powersmooth(n, B)	Determines if the positive integer n is B-powersmooth.

galois.is_prime

`galois.is_prime(n)`

Determines if n is prime.

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

Returns `True` if the integer n is prime.

Return type bool

Notes

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 10 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.

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(1000000000000000035000061)
Out[3]: True
```

galois.is_prime_power

`galois.is_prime_power(n)`

Determines if n is a prime power $n = p^k$ for prime p and $k \geq 1$.

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

Returns `True` if the integer n is a prime power.

Return type bool

Notes

There is some controversy over whether 1 is a prime power p^0. Since 1 is the 0-th power of all primes, it is often regarded not as a prime power. This function returns `False` for 1.

Examples

```
In [1]: galois.is_prime_power(8)
Out[1]: True

In [2]: galois.is_prime_power(6)
Out[2]: False

In [3]: galois.is_prime_power(1)
Out[3]: False
```

galois.is_composite

`galois.is_composite(n)`

Determines if n is composite.

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

Returns `True` if the integer n is composite.

Return type bool

Examples

```
In [1]: galois.is_composite(13)
Out[1]: False

In [2]: galois.is_composite(15)
Out[2]: True
```

galois.is_perfect_power

`galois.is_perfect_power(n)`

Determines if n is a perfect power $n = x^k$ for $x > 0$ and $k \geq 2$.

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

Returns `True` if the integer n is a perfect power.

Return type bool

Examples

```
In [1]: galois.is_perfect_power(8)
Out[1]: True

In [2]: galois.is_perfect_power(16)
Out[2]: True

In [3]: galois.is_perfect_power(20)
Out[3]: False
```

galois.is_smooth

`galois.is_smooth(n, B)`

Determines if the positive integer n is B-smooth.

Parameters

- **n** (*int*) – A positive integer.
- **B** (*int*) – The smoothness bound $B \geq 2$.

Returns `True` if n is B-smooth.

Return type bool

Notes

An integer n with prime factorization $n = p_1^{e_1} \dots p_k^{e_k}$ is B-smooth if $p_k \leq B$. The 2-smooth numbers are the powers of 2. The 5-smooth numbers are known as *regular numbers*. The 7-smooth numbers are known as *humble numbers* or *highly composite numbers*.

Examples

```
In [1]: galois.is_smooth(2**10, 2)
Out[1]: True

In [2]: galois.is_smooth(10, 5)
Out[2]: True

In [3]: galois.is_smooth(12, 5)
Out[3]: True

In [4]: galois.is_smooth(60**2, 5)
Out[4]: True
```

galois.is_powersmooth

`galois.is_powersmooth(n, B)`

Determines if the positive integer n is B-powersmooth.

Parameters

- **n** (*int*) – A positive integer.
- **B** (*int*) – The smoothness bound $B \geq 2$.

Returns `True` if n is B-powersmooth.

Return type bool

Notes

An integer n with prime factorization $n = p_1^{e_1} \dots p_k^{e_k}$ is B-powersmooth if $p_i^{e_i} \le B$ for $1 \le i \le k$.

Examples

Comparison of B-smooth and B-powersmooth. Necessarily, any n that is B-powersmooth must be B-smooth.

```
In [1]: galois.is_smooth(2**4 * 3**2 * 5, 5)
Out[1]: True

In [2]: galois.is_powersmooth(2**4 * 3**2 * 5, 5)
Out[2]: False
```

7.7 Primes

This section contains functions for generating primes and analyzing primality.

7.7.1 Prime number generation

primes(n)	Returns all primes p for $p \le n$.
kth_prime(k)	Returns the k-th prime.
prev_prime(n)	Returns the nearest prime p, such that $p \le n$.
next_prime(n)	Returns the nearest prime p, such that $p > n$.
random_prime(bits)	Returns a random prime p with b bits, such that $2^b \le p < 2^{b+1}$.
mersenne_exponents([n])	Returns all known Mersenne exponents e for $e \le n$.
mersenne_primes([n])	Returns all known Mersenne primes p for $p \le 2^n - 1$.

galois.primes

`galois.primes(n)`

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

Parameters **n** (*int*) – An integer.

Returns All primes up to and including n. If $n < 2$, the function returns an empty list.

Return type list

Notes

This function implements the Sieve of Eratosthenes to efficiently find the primes.

References

- https://oeis.org/A000040

Examples

```
In [1]: galois.primes(19)
Out[1]: [2, 3, 5, 7, 11, 13, 17, 19]

In [2]: galois.primes(20)
Out[2]: [2, 3, 5, 7, 11, 13, 17, 19]
```

galois.kth_prime

galois.**kth_prime**(k)

Returns the k-th prime.

Parameters **k** (*int*) – The prime index (1-indexed), where $k = \{1, 2, 3, 4, \dots\}$ for primes $p = \{2, 3, 5, 7, \dots\}$.

Returns The k-th prime.

Return type int

Examples

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

In [2]: galois.kth_prime(3)
Out[2]: 5

In [3]: galois.kth_prime(1000)
Out[3]: 7919
```

galois.prev_prime

galois.**prev_prime**(n)

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

Parameters **n** (*int*) – An integer.

Returns The nearest prime $p \leq n$. If $n < 2$, the function returns `None`.

Return type int

Examples

```
In [1]: galois.prev_prime(13)
Out[1]: 13

In [2]: galois.prev_prime(15)
Out[2]: 13
```

galois.next_prime

galois.**next_prime**(*n*)

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

Parameters **n** (*int*) – An integer.

Returns The nearest prime $p > n$.

Return type int

Examples

```
In [1]: galois.next_prime(13)
Out[1]: 17

In [2]: galois.next_prime(15)
Out[2]: 17
```

galois.random_prime

galois.**random_prime**(*bits*)

Returns a random prime p with b bits, such that $2^b \leq p < 2^{b+1}$.

This function randomly generates integers with b bits and uses the primality tests in *galois.is_prime()* to determine if p is prime.

Parameters **bits** (*int*) – The number of bits in the prime p.

Returns A random prime in $2^b \leq p < 2^{b+1}$.

Return type int

References

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

Examples

Generate a random 1024-bit prime.

```
In [1]: p = galois.random_prime(1024); p
Out[1]: 
↪302711876155417019217947447879278627191564746205258677658512247564508724574369810999060567097765

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

```
$ openssl prime 
↪23686178792695738220699688608721459202975252407802639235893684447966742357083311612650692787877
1514D68EDB7C650F1FF713531A1A43255A4BE6D66EE1FDBD96F4EB32757C1B1BAF16A5933E24D45FAD6C6A814F3C8C14F3C
↪(23686178792695738220699688608721459202975252407802639235893684447966742357083311612650692787877
↪is prime
```

galois.mersenne_exponents

galois.**mersenne_exponents**(*n=None*)

Returns all known Mersenne exponents e for $e \leq n$.

A Mersenne exponent e is an exponent of 2 such that $2^e - 1$ is prime.

Parameters **n** (*int, optional*) – The max exponent of 2. The default is None which returns all known Mersenne exponents.

Returns The list of Mersenne exponents e for $e \leq n$.

Return type list

References

- https://oeis.org/A000043

Examples

```
# List all Mersenne exponents for Mersenne primes up to 2000 bits
In [1]: e = galois.mersenne_exponents(2000); e
Out[1]: [2, 3, 5, 7, 13, 17, 19, 31, 61, 89, 107, 127, 521, 607, 1279]

# Select one Merseene exponent and compute its Mersenne prime
In [2]: p = 2**e[-1] - 1; p
Out[2]: 
↪104079321946643990819252403273640855386152622472667048053191123504036080596733602980122394417323

In [3]: galois.is_prime(p)
Out[3]: True
```

galois.mersenne_primes

galois.**mersenne_primes**(*n=None*)

Returns all known Mersenne primes p for $p \leq 2^n - 1$.

Mersenne primes are primes that are one less than a power of 2.

Parameters **n** (*int, optional*) – The max power of 2. The default is `None` which returns all known Mersenne exponents.

Returns The list of known Mersenne primes p for $p \leq 2^n - 1$.

Return type list

References

- https://oeis.org/A000668

Examples

```
# List all Mersenne primes up to 2000 bits
In [1]: p = galois.mersenne_primes(2000); p
Out[1]:
[3,
 7,
 31,
 127,
 8191,
 131071,
 524287,
 2147483647,
 2305843009213693951,
 618970019642690137449562111,
 162259276829213363391578010288127,
 170141183460469231731687303715884105727,
␣
↪68647976601306097149819007990813932172694353001433054093944634591855431833976560521225596406614545
↪
␣
↪531137992816767098689588206552468627329593117727031923199444138200403559860852242739162502265229
↪
␣
↪104079321946643990819252403273640855386152622472667048053191123504036080596733602980122394417323

In [2]: galois.is_prime(p[-1])
Out[2]: True
```

7.7.2 Primality tests

`is_prime(n)`	Determines if n is prime.
`is_prime_power(n)`	Determines if n is a prime power $n = p^k$ for prime p and $k \geq 1$.
`is_perfect_power(n)`	Determines if n is a perfect power $n = x^k$ for $x > 0$ and $k \geq 2$.
`is_composite(n)`	Determines if n is composite.
`is_square_free(value)`	Determines if the positive integer or the non-constant, monic polynomial is square-free.
`is_smooth(n, B)`	Determines if the positive integer n is B-smooth.
`is_powersmooth(n, B)`	Determines if the positive integer n is B-powersmooth.

7.7.3 Specific primality tests

`fermat_primality_test(n[, a, rounds])`	Determines if n is composite using Fermat's primality test.
`miller_rabin_primality_test(n[, a, rounds])`	Determines if n is composite using the Miller-Rabin primality test.

galois.fermat_primality_test

`galois.fermat_primality_test(n, a=None, rounds=1)`

Determines if n is composite using Fermat's primality test.

Parameters

- **n** (*int*) – An odd integer $n \geq 3$.
- **a** (*int, optional*) – An integer in $2 \leq a \leq n - 2$. The default is `None` which selects a random a.
- **rounds** (*int, optional*) – The number of iterations attempting to detect n as composite. Additional rounds will choose a new a. The default is 1.

Returns `False` if n is shown to be composite. `True` if n is probable prime.

Return type bool

Notes

Fermat's theorem says that for prime p and $1 \leq a \leq p - 1$, the congruence $a^{p-1} \equiv 1\ (\mathrm{mod}\ p)$ holds. Fermat's primality test of n computes $a^{n-1} \bmod n$ for some $1 \leq a \leq n - 1$. If a is such that $a^{p-1} \not\equiv 1\ (\mathrm{mod}\ p)$, then a is said to be a *Fermat witness* to the compositeness of n. If n is composite and $a^{p-1} \equiv 1\ (\mathrm{mod}\ p)$, then a is said to be a *Fermat liar* to the primality of n.

Since $a = \{1, n - 1\}$ are Fermat liars for all composite n, it is common to reduce the range of possible a to $2 \leq a \leq n - 2$.

References

- Section 4.2.1 from https://cacr.uwaterloo.ca/hac/about/chap4.pdf

Examples

Fermat's primality test will never mark a true prime as composite.

```
In [1]: primes = [257, 24841, 65497]

In [2]: [galois.is_prime(p) for p in primes]
Out[2]: [True, True, True]

In [3]: [galois.fermat_primality_test(p) for p in primes]
Out[3]: [True, True, True]
```

However, Fermat's primality test may mark a composite as probable prime. Here are pseudoprimes base 2 from A001567.

```
# List of some Fermat pseudoprimes to base 2
In [4]: pseudoprimes = [2047, 29341, 65281]

In [5]: [galois.is_prime(p) for p in pseudoprimes]
Out[5]: [False, False, False]

# The pseudoprimes base 2 satisfy 2^(p-1) = 1 (mod p)
In [6]: [galois.fermat_primality_test(p, a=2) for p in pseudoprimes]
Out[6]: [True, True, True]

# But they may not satisfy a^(p-1) = 1 (mod p) for other a
In [7]: [galois.fermat_primality_test(p) for p in pseudoprimes]
Out[7]: [False, False, False]
```

And the pseudoprimes base 3 from A005935.

```
# List of some Fermat pseudoprimes to base 3
In [8]: pseudoprimes = [2465, 7381, 16531]

In [9]: [galois.is_prime(p) for p in pseudoprimes]
Out[9]: [False, False, False]

# The pseudoprimes base 3 satisfy 3^(p-1) = 1 (mod p)
In [10]: [galois.fermat_primality_test(p, a=3) for p in pseudoprimes]
Out[10]: [True, True, True]

# But they may not satisfy a^(p-1) = 1 (mod p) for other a
In [11]: [galois.fermat_primality_test(p) for p in pseudoprimes]
Out[11]: [True, False, False]
```

galois.miller_rabin_primality_test

`galois.miller_rabin_primality_test(n, a=2, rounds=1)`

Determines if n is composite using the Miller-Rabin primality test.

Parameters

- **n** (*int*) – An odd integer $n \geq 3$.
- **a** (*int, optional*) – An integer in $2 \leq a \leq n-2$. The default is 2.
- **rounds** (*int, optional*) – The number of iterations attempting to detect n as composite. Additional rounds will choose consecutive primes for a.

Returns `False` if n is shown to be composite. `True` if n is probable prime.

Return type bool

Notes

The Miller-Rabin primality test is based on the fact that for odd n with factorization $n = 2^s r$ for odd r and integer a such that $\gcd(a, n) = 1$, then either $a^r \equiv 1 \pmod n$ or $a^{2^j r} \equiv -1 \pmod n$ for some j in $0 \leq j \leq s-1$.

In the Miller-Rabin primality test, if $a^r \not\equiv 1 \pmod n$ and $a^{2^j r} \not\equiv -1 \pmod n$ for all j in $0 \leq j \leq s-1$, then a is called a *strong witness* to the compositeness of n. If not, namely $a^r \equiv 1 \pmod n$ or $a^{2^j r} \equiv -1 \pmod n$ for any j in $0 \leq j \leq s-1$, then a is called a *strong liar* to the primality of n and n is called a *strong pseudoprime to the base a*.

Since $a = \{1, n-1\}$ are strong liars for all composite n, it is common to reduce the range of possible a to $2 \leq a \leq n-2$.

For composite odd n, the probability that the Miller-Rabin test declares it a probable prime is less than $(\frac{1}{4})^t$, where t is the number of rounds, and is often much lower.

References

- Section 4.2.3 from https://cacr.uwaterloo.ca/hac/about/chap4.pdf
- https://math.dartmouth.edu/~carlp/PDF/paper25.pdf

Examples

The Miller-Rabin primality test will never mark a true prime as composite.

```
In [1]: primes = [257, 24841, 65497]

In [2]: [galois.is_prime(p) for p in primes]
Out[2]: [True, True, True]

In [3]: [galois.miller_rabin_primality_test(p) for p in primes]
Out[3]: [True, True, True]
```

However, a composite n may have strong liars. 91 has $\{9, 10, 12, 16, 17, 22, 29, 38, 53, 62, 69, 74, 75, 79, 81, 82\}$ as strong liars.

```
In [4]: strong_liars = [9,10,12,16,17,22,29,38,53,62,69,74,75,79,81,82]

In [5]: witnesses = [a for a in range(2, 90) if a not in strong_liars]

# All strong liars falsely assert that 91 is prime
In [6]: [galois.miller_rabin_primality_test(91, a=a) for a in strong_liars] ==␣
↪[True,]*len(strong_liars)
Out[6]: True

# All other a are witnesses to the compositeness of 91
In [7]: [galois.miller_rabin_primality_test(91, a=a) for a in witnesses] == [False,
↪]*len(witnesses)
Out[7]: True
```

7.8 Numpy Examples

This section contains examples of some numpy functions when called on Galois field arrays. Many more functions are supported, just not explicitly documented here.

7.8.1 General

np.copy(a)	Returns a copy of a given Galois field array.
np.concatenate(arrays[, axis])	Concatenates the input arrays along the given axis.
np.insert(array, object, values[, axis])	Inserts values along the given axis.

np.copy

np.copy(*a*)

Returns a copy of a given Galois field array.

See: https://numpy.org/doc/stable/reference/generated/numpy.copy.html

Warning: This function returns an `numpy.ndarray`, not an instance of the subclass. To return a copy of the subclass, pass `subok=True` (for numpy version 1.19 and above) or use `a.copy()`.

Examples

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

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

# NOTE: b is an ndarray
In [3]: b = np.copy(a); b
Out[3]: array([3, 7, 7, 4, 3], dtype=uint8)
```

(continues on next page)

(continued from previous page)

```
In [4]: type(b)
Out[4]: numpy.ndarray

In [5]: a[0] = 0; a
Out[5]: GF([0, 7, 7, 4, 3], order=2^3)

# b is unmodified
In [6]: b
Out[6]: array([3, 7, 7, 4, 3], dtype=uint8)
```

```
In [7]: a.copy()
Out[7]: GF([0, 7, 7, 4, 3], order=2^3)
```

np.concatenate

`np.concatenate`(*arrays*, *axis=0*)

Concatenates the input arrays along the given axis.

See: https://numpy.org/doc/stable/reference/generated/numpy.concatenate.html

Examples

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

In [2]: A = GF.Random((2,2)); A
Out[2]:
GF([[3, 2],
    [5, 0]], order=2^3)

In [3]: B = GF.Random((2,2)); B
Out[3]:
GF([[2, 5],
    [5, 6]], order=2^3)

In [4]: np.concatenate((A,B), axis=0)
Out[4]:
GF([[3, 2],
    [5, 0],
    [2, 5],
    [5, 6]], order=2^3)

In [5]: np.concatenate((A,B), axis=1)
Out[5]:
GF([[3, 2, 2, 5],
    [5, 0, 5, 6]], order=2^3)
```

np.insert

`np.insert(array, object, values, axis=None)`

Inserts values along the given axis.

See: https://numpy.org/doc/stable/reference/generated/numpy.insert.html

Examples

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

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

In [3]: np.insert(x, 1, [0,1,2,3])
Out[3]: GF([7, 0, 1, 2, 3, 6, 1, 3, 2], order=2^3)
```

7.8.2 Arithmetic

np.add(x, y)	Adds two Galois field arrays element-wise.
np.subtract(x, y)	Subtracts two Galois field arrays element-wise.
np.multiply(x, y)	Multiplies two Galois field arrays element-wise.
np.divide(x, y)	Divides two Galois field arrays element-wise.
np.negative(x)	Returns the element-wise additive inverse of a Galois field array.
np.reciprocal(x)	Returns the element-wise multiplicative inverse of a Galois field array.
np.power(x, y)	Exponentiates a Galois field array element-wise.
np.square(x)	Squares a Galois field array element-wise.
np.log(x)	Computes the logarithm (base `GF.primitive_element`) of a Galois field array element-wise.
np.matmul(a, b)	Returns the matrix multiplication of two Galois field arrays.

np.add

`np.add(x, y)`

Adds two Galois field arrays element-wise.

References

- https://numpy.org/doc/stable/reference/generated/numpy.add.html

Examples

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

In [2]: x = GF.Random(10); x
Out[2]: GF([13, 27, 21,  9,  3, 15,  0, 14, 19, 26], order=31)

In [3]: y = GF.Random(10); y
Out[3]: GF([12,  1, 22,  5, 29,  5, 23,  8, 28, 21], order=31)

In [4]: np.add(x, y)
Out[4]: GF([25, 28, 12, 14,  1, 20, 23, 22, 16, 16], order=31)

In [5]: x + y
Out[5]: GF([25, 28, 12, 14,  1, 20, 23, 22, 16, 16], order=31)
```

np.subtract

`np.subtract(x, y)`

Subtracts two Galois field arrays element-wise.

References

- https://numpy.org/doc/stable/reference/generated/numpy.subtract.html

Examples

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

In [2]: x = GF.Random(10); x
Out[2]: GF([29, 11,  8,  2, 11,  1, 12,  0, 22, 20], order=31)

In [3]: y = GF.Random(10); y
Out[3]: GF([ 7, 21,  2, 19, 24, 16, 18,  9, 19,  4], order=31)

In [4]: np.subtract(x, y)
Out[4]: GF([22, 21,  6, 14, 18, 16, 25, 22,  3, 16], order=31)

In [5]: x - y
Out[5]: GF([22, 21,  6, 14, 18, 16, 25, 22,  3, 16], order=31)
```

np.multiply

`np.multiply(x, y)`

Multiplies two Galois field arrays element-wise.

References

- https://numpy.org/doc/stable/reference/generated/numpy.multiply.html

Examples

Multiplying two Galois field arrays results in field multiplication.

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

In [2]: x = GF.Random(10); x
Out[2]: GF([ 0, 21, 16, 16, 15, 28, 13,  8, 29, 27], order=31)

In [3]: y = GF.Random(10); y
Out[3]: GF([ 2, 26, 25,  3, 17, 14, 29,  5,  8,  9], order=31)

In [4]: np.multiply(x, y)
Out[4]: GF([ 0, 19, 28, 17,  7, 20,  5,  9, 15, 26], order=31)

In [5]: x * y
Out[5]: GF([ 0, 19, 28, 17,  7, 20,  5,  9, 15, 26], 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([ 2, 29,  0,  6,  4, 25,  1, 13,  2, 11], order=31)

In [8]: np.multiply(x, 3)
Out[8]: GF([ 6, 25,  0, 18, 12, 13,  3,  8,  6,  2], order=31)

In [9]: x * 3
Out[9]: GF([ 6, 25,  0, 18, 12, 13,  3,  8,  6,  2], order=31)
```

```
In [10]: print(GF.properties)
GF(31):
  characteristic: 31
  degree: 1
  order: 31
  irreducible_poly: x + 28
  is_primitive_poly: True
  primitive_element: 3

# Adding `characteristic` copies of any element always results in zero
In [11]: x * GF.characteristic
Out[11]: GF([0, 0, 0, 0, 0, 0, 0, 0, 0, 0], order=31)
```

np.divide

np.**divide**(*x*, *y*)

Divides two Galois field arrays element-wise.

References

- https://numpy.org/doc/stable/reference/generated/numpy.divide.html

Examples

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

In [2]: x = GF.Random(10); x
Out[2]: GF([ 7, 22, 29,  3,  3, 10,  2, 17,  3, 30], order=31)

In [3]: y = GF.Random(10, low=1); y
Out[3]: GF([20,  1, 16,  7, 27, 16,  2,  3, 15,  5], order=31)

In [4]: z = np.divide(x, y); z
Out[4]: GF([ 5, 22, 27, 27,  7, 20,  1, 16, 25,  6], order=31)

In [5]: y * z
Out[5]: GF([ 7, 22, 29,  3,  3, 10,  2, 17,  3, 30], order=31)
```

```
In [6]: np.true_divide(x, y)
Out[6]: GF([ 5, 22, 27, 27,  7, 20,  1, 16, 25,  6], order=31)

In [7]: x / y
Out[7]: GF([ 5, 22, 27, 27,  7, 20,  1, 16, 25,  6], order=31)

In [8]: np.floor_divide(x, y)
Out[8]: GF([ 5, 22, 27, 27,  7, 20,  1, 16, 25,  6], order=31)

In [9]: x // y
Out[9]: GF([ 5, 22, 27, 27,  7, 20,  1, 16, 25,  6], order=31)
```

np.negative

np.**negative**(*x*)

Returns the element-wise additive inverse of a Galois field array.

References

- https://numpy.org/doc/stable/reference/generated/numpy.negative.html

Examples

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

In [2]: x = GF.Random(10); x
Out[2]: GF([12, 11,  5, 28, 30, 15, 13, 22,  7, 20], order=31)

In [3]: y = np.negative(x); y
Out[3]: GF([19, 20, 26,  3,  1, 16, 18,  9, 24, 11], 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([19, 20, 26,  3,  1, 16, 18,  9, 24, 11], order=31)

In [6]: -1*x
Out[6]: GF([19, 20, 26,  3,  1, 16, 18,  9, 24, 11], order=31)
```

np.reciprocal

`np.reciprocal(x)`

Returns the element-wise multiplicative inverse of a Galois field array.

References

- https://numpy.org/doc/stable/reference/generated/numpy.reciprocal.html

Examples

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

In [2]: x = GF.Random(5, low=1); x
Out[2]: GF([29, 10, 19, 30, 18], order=31)

In [3]: y = np.reciprocal(x); y
Out[3]: GF([15, 28, 18, 30, 19], order=31)

In [4]: x * y
Out[4]: GF([1, 1, 1, 1, 1], order=31)
```

```
In [5]: x ** -1
Out[5]: GF([15, 28, 18, 30, 19], order=31)

In [6]: GF(1) / x
```

(continues on next page)

(continued from previous page)

```
Out[6]: GF([15, 28, 18, 30, 19], order=31)

In [7]: GF(1) // x
Out[7]: GF([15, 28, 18, 30, 19], order=31)
```

np.power

np.**power**(*x*, *y*)

Exponentiates a Galois field array element-wise.

References

- https://numpy.org/doc/stable/reference/generated/numpy.power.html

Examples

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

In [2]: x = GF.Random(10); x
Out[2]: GF([24,  2, 20, 13, 26, 14, 17, 29, 16,  6], order=31)

In [3]: np.power(x, 3)
Out[3]: GF([29,  8,  2, 27, 30, 16, 15, 23,  4, 30], order=31)

In [4]: x ** 3
Out[4]: GF([29,  8,  2, 27, 30, 16, 15, 23,  4, 30], order=31)

In [5]: x * x * x
Out[5]: GF([29,  8,  2, 27, 30, 16, 15, 23,  4, 30], order=31)
```

```
In [6]: x = GF.Random(10, low=1); x
Out[6]: GF([ 9, 22, 13, 23,  5, 11, 30, 21, 17, 17], order=31)

In [7]: y = np.random.randint(-10, 10, 10); y
Out[7]: array([-1,  3, -6, -8,  7,  0,  8, -1,  8,  8])

In [8]: np.power(x, y)
Out[8]: GF([ 7, 15,  2,  2,  5,  1,  1,  3, 18, 18], order=31)

In [9]: x ** y
Out[9]: GF([ 7, 15,  2,  2,  5,  1,  1,  3, 18, 18], order=31)
```

np.square

np.**square**(x)

Squares a Galois field array element-wise.

References

- https://numpy.org/doc/stable/reference/generated/numpy.square.html

Examples

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

In [2]: x = GF.Random(10); x
Out[2]: GF([13, 10,  8, 26,  8, 11, 16,  5, 22, 18], order=31)

In [3]: np.square(x)
Out[3]: GF([14,  7,  2, 25,  2, 28,  8, 25, 19, 14], order=31)

In [4]: x ** 2
Out[4]: GF([14,  7,  2, 25,  2, 28,  8, 25, 19, 14], order=31)

In [5]: x * x
Out[5]: GF([14,  7,  2, 25,  2, 28,  8, 25, 19, 14], order=31)
```

np.log

np.**log**(x)

Computes the logarithm (base `GF.primitive_element`) of a Galois field array element-wise.

Calling *np.log()* implicitly uses base *galois.FieldClass.primitive_element*. See `galois.FieldArray.log()` for logarithm with arbitrary base.

References

- https://numpy.org/doc/stable/reference/generated/numpy.log.html

Examples

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

In [2]: alpha = GF.primitive_element; alpha
Out[2]: GF(3, order=31)

In [3]: x = GF.Random(10, low=1); x
Out[3]: GF([14,  8, 23, 13, 24, 24, 23, 26, 22, 25], order=31)

In [4]: y = np.log(x); y
Out[4]: array([22, 12, 27, 11, 13, 13, 27,  5, 17, 10])
```

(continues on next page)

(continued from previous page)

```
In [5]: alpha ** y
Out[5]: GF([14,  8, 23, 13, 24, 24, 23, 26, 22, 25], order=31)
```

np.matmul

`np.matmul(a, b)`

Returns the matrix multiplication of two Galois field arrays.

References

- https://numpy.org/doc/stable/reference/generated/numpy.matmul.html

Examples

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

In [2]: A = GF.Random((3,3)); A
Out[2]:
GF([[27,  8, 17],
    [30, 27, 27],
    [ 7, 15,  1]], order=31)

In [3]: B = GF.Random((3,3)); B
Out[3]:
GF([[16,  0,  0],
    [11, 10, 23],
    [29, 23, 18]], order=31)

In [4]: np.matmul(A, B)
Out[4]:
GF([[21,  6, 25],
    [10, 23, 22],
    [27, 18, 22]], order=31)

In [5]: A @ B
Out[5]:
GF([[21,  6, 25],
    [10, 23, 22],
    [27, 18, 22]], order=31)
```

7.8.3 Advanced Arithmetic

np.convolve(a, b)	Convolves the input arrays.

np.convolve

np.**convolve**(*a*, *b*)

Convolves the input arrays.

See: https://numpy.org/doc/stable/reference/generated/numpy.convolve.html

Examples

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

In [2]: a = GF.Random(10)

In [3]: b = GF.Random(10)

In [4]: np.convolve(a, b)
Out[4]:
GF([ 3, 18,  1,  8, 14, 13, 29, 25, 17, 29, 23,  2,  2, 12, 20,  5,  6,
     3, 10], order=31)

# Equivalent implementation with native numpy
In [5]: np.convolve(a.view(np.ndarray).astype(int), b.view(np.ndarray).astype(int))
↪% 31
Out[5]:
array([ 3, 18,  1,  8, 14, 13, 29, 25, 17, 29, 23,  2,  2, 12, 20,  5,  6,
        3, 10])
```

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

In [7]: a = GF.Random(10)

In [8]: b = GF.Random(10)

In [9]: np.convolve(a, b)
Out[9]:
GF([197, 176, 120, 213, 244,  48,  52,  59, 134, 244,  62,  70, 214,  71,
    222, 101, 126, 171, 219], order=2^8)
```

7.8.4 Linear Algebra

np.dot(a, b)	Returns the dot product of two Galois field arrays.
np.vdot(a, b)	Returns the dot product of two Galois field vectors.
np.inner(a, b)	Returns the inner product of two Galois field arrays.
np.outer(a, b)	Returns the outer product of two Galois field arrays.
np.matmul(a, b)	Returns the matrix multiplication of two Galois field arrays.
np.linalg.matrix_power(x)	Raises a square Galois field matrix to an integer power.
np.linalg.det(A)	Computes the determinant of the matrix.
np.linalg.matrix_rank(x)	Returns the rank of a Galois field matrix.
np.trace(x)	Returns the sum along the diagonal of a Galois field array.
np.linalg.solve(x)	Solves the system of linear equations.
np.linalg.inv(A)	Computes the inverse of the matrix.

np.dot

np.**dot**(*a*, *b*)

Returns the dot product of two Galois field arrays.

References

- https://numpy.org/doc/stable/reference/generated/numpy.dot.html

Examples

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

In [2]: a = GF.Random(3); a
Out[2]: GF([29, 15, 20], order=31)

In [3]: b = GF.Random(3); b
Out[3]: GF([30,  5, 21], order=31)

In [4]: np.dot(a, b)
Out[4]: GF(1, order=31)
```

```
In [5]: A = GF.Random((3,3)); A
Out[5]:
GF([[ 8, 11,  4],
    [ 5, 14, 20],
    [27, 17, 29]], order=31)

In [6]: B = GF.Random((3,3)); B
Out[6]:
GF([[ 1,  0, 19],
    [28, 29, 25],
    [22,  3, 26]], order=31)
```

(continues on next page)

(continued from previous page)

```
In [7]: np.dot(A, B)
Out[7]:
GF([[ 1, 21,  4],
    [ 0,  1,  4],
    [25, 22, 18]], order=31)
```

np.vdot

np.**vdot**(*a*, *b*)

Returns the dot product of two Galois field vectors.

References

- https://numpy.org/doc/stable/reference/generated/numpy.vdot.html

Examples

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

In [2]: a = GF.Random(3); a
Out[2]: GF([14,  2, 24], order=31)

In [3]: b = GF.Random(3); b
Out[3]: GF([13, 19, 21], order=31)

In [4]: np.vdot(a, b)
Out[4]: GF(11, order=31)
```

```
In [5]: A = GF.Random((3,3)); A
Out[5]:
GF([[ 4, 13, 13],
    [10,  7, 22],
    [16, 10, 16]], order=31)

In [6]: B = GF.Random((3,3)); B
Out[6]:
GF([[ 1,  9,  8],
    [ 9, 30, 28],
    [29, 12,  7]], order=31)

In [7]: np.vdot(A, B)
Out[7]: GF(8, order=31)
```

np.inner

np.**inner**(*a*, *b*)

Returns the inner product of two Galois field arrays.

References

- https://numpy.org/doc/stable/reference/generated/numpy.inner.html

Examples

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

In [2]: a = GF.Random(3); a
Out[2]: GF([12,  3, 19], order=31)

In [3]: b = GF.Random(3); b
Out[3]: GF([20, 20, 10], order=31)

In [4]: np.inner(a, b)
Out[4]: GF(25, order=31)
```

```
In [5]: A = GF.Random((3,3)); A
Out[5]:
GF([[27, 10, 15],
    [28,  3, 13],
    [ 1, 18, 13]], order=31)

In [6]: B = GF.Random((3,3)); B
Out[6]:
GF([[25,  5, 23],
    [ 2, 20, 11],
    [25, 30, 25]], order=31)

In [7]: np.inner(A, B)
Out[7]:
GF([[16, 16, 17],
    [22, 11, 30],
    [11,  9, 22]], order=31)
```

np.outer

np.**outer**(*a*, *b*)

Returns the outer product of two Galois field arrays.

References

- https://numpy.org/doc/stable/reference/generated/numpy.outer.html

Examples

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

In [2]: a = GF.Random(3); a
Out[2]: GF([11, 12, 19], order=31)

In [3]: b = GF.Random(3); b
Out[3]: GF([14, 23, 19], order=31)

In [4]: np.outer(a, b)
Out[4]:
GF([[30,  5, 23],
    [13, 28, 11],
    [18,  3, 20]], order=31)
```

np.linalg.matrix_power

`np.linalg.matrix_power(x)`

Raises a square Galois field matrix to an integer power.

References

- https://numpy.org/doc/stable/reference/generated/numpy.linalg.matrix_power.html

Examples

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

In [2]: A = GF.Random((3,3)); A
Out[2]:
GF([[21, 26,  5],
    [16, 19, 13],
    [16,  7,  2]], order=31)

In [3]: np.linalg.matrix_power(A, 3)
Out[3]:
GF([[12,  1,  5],
    [20, 29, 17],
    [ 2,  5, 25]], order=31)

In [4]: A @ A @ A
Out[4]:
GF([[12,  1,  5],
    [20, 29, 17],
    [ 2,  5, 25]], 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([[28,  5, 19],
    [16,  6, 11],
    [12, 10, 23]], order=31)

In [8]: np.linalg.matrix_power(A, -3)
Out[8]:
GF([[18,  0,  5],
    [ 8, 20, 13],
    [19,  7, 10]], order=31)

In [9]: A_inv = np.linalg.inv(A)

In [10]: A_inv @ A_inv @ A_inv
Out[10]:
GF([[18,  0,  5],
    [ 8, 20, 13],
    [19,  7, 10]], order=31)
```

np.linalg.det

np.linalg.**det**(*A*)

Computes the determinant of the matrix.

References

- https://numpy.org/doc/stable/reference/generated/numpy.linalg.det.html

Examples

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

In [2]: A = GF.Random((2,2)); A
Out[2]:
GF([[22, 23],
    [18, 17]], order=31)

In [3]: np.linalg.det(A)
Out[3]: GF(22, order=31)
```

(continues on next page)

(continued from previous page)

```
In [4]: A[0,0]*A[1,1] - A[0,1]*A[1,0]
Out[4]: GF(22, order=31)
```

np.linalg.matrix_rank

`np.linalg.matrix_rank(x)`

Returns the rank of a Galois field matrix.

References

- https://numpy.org/doc/stable/reference/generated/numpy.linalg.matrix_rank.html

Examples

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

In [2]: A = GF.Identity(4); A
Out[2]:
GF([[1, 0, 0, 0],
    [0, 1, 0, 0],
    [0, 0, 1, 0],
    [0, 0, 0, 1]], order=31)

In [3]: np.linalg.matrix_rank(A)
Out[3]: 4
```

One column is a linear combination of another.

```
In [4]: GF = galois.GF(31)

In [5]: A = GF.Random((4,4)); A
Out[5]:
GF([[ 4,  0,  3,  4],
    [14, 28, 19, 16],
    [ 8, 21, 26, 20],
    [ 0, 17, 12,  2]], order=31)

In [6]: A[:,2] = A[:,1] * GF(17); A
Out[6]:
GF([[ 4,  0,  0,  4],
    [14, 28, 11, 16],
    [ 8, 21, 16, 20],
    [ 0, 17, 10,  2]], 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([[19,  6,  3,  6],
    [ 5, 23, 28,  4],
    [25, 11, 20, 27],
    [18,  6,  0, 12]], order=31)

In [10]: A[3,:] = A[2,:] * GF(8); A
Out[10]:
GF([[19,  6,  3,  6],
    [ 5, 23, 28,  4],
    [25, 11, 20, 27],
    [14, 26,  5, 30]], order=31)

In [11]: np.linalg.matrix_rank(A)
Out[11]: 3
```

np.trace

`np.trace(x)`

Returns the sum along the diagonal of a Galois field array.

References

- https://numpy.org/doc/stable/reference/generated/numpy.trace.html

Examples

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

In [2]: A = GF.Random((5,6)); A
Out[2]:
GF([[11,  8,  8, 13, 27,  1],
    [26, 16, 17, 13, 25,  2],
    [26,  8, 23, 12,  7,  4],
    [14, 28, 23, 21, 19, 19],
    [30, 12,  5, 28, 21, 22]], order=31)

In [3]: np.trace(A)
Out[3]: GF(30, order=31)

In [4]: A[0,0] + A[1,1] + A[2,2] + A[3,3] + A[4,4]
Out[4]: GF(30, order=31)
```

```
In [5]: np.trace(A, offset=1)
Out[5]: GF(16, order=31)
```

(continues on next page)

(continued from previous page)

```
In [6]: A[0,1] + A[1,2] + A[2,3] + A[3,4] + A[4,5]
Out[6]: GF(16, order=31)
```

np.linalg.solve

`np.linalg.solve(x)`

Solves the system of linear equations.

References

- https://numpy.org/doc/stable/reference/generated/numpy.linalg.solve.html

Examples

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

# Ensure A is full rank and invertible
In [2]: while True:
   ...:     A = GF.Random((4,4))
   ...:     if np.linalg.matrix_rank(A) == 4:
   ...:         break
   ...:

In [3]: A
Out[3]:
GF([[23, 17, 10,  9],
    [20, 29,  9, 11],
    [ 2,  8,  1, 21],
    [13,  5,  8,  0]], order=31)

In [4]: b = GF.Random(4); b
Out[4]: GF([ 7, 19,  3, 22], order=31)

In [5]: x = np.linalg.solve(A, b); x
Out[5]: GF([14,  3, 13,  0], order=31)

In [6]: A @ x
Out[6]: GF([ 7, 19,  3, 22], 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
```

(continues on next page)

(continued from previous page)

```
Out[9]:
GF([[28,  6, 26,  1],
    [25,  3, 14, 25],
    [20, 26, 10,  7],
    [ 6, 18,  1,  4]], order=31)

In [10]: B = GF.Random((4,2)); B
Out[10]:
GF([[ 4, 13],
    [ 8, 16],
    [ 7,  4],
    [18, 25]], order=31)

In [11]: X = np.linalg.solve(A, B); X
Out[11]:
GF([[13, 13],
    [ 4,  1],
    [27, 10],
    [30,  3]], order=31)

In [12]: A @ X
Out[12]:
GF([[ 4, 13],
    [ 8, 16],
    [ 7,  4],
    [18, 25]], order=31)
```

np.linalg.inv

`np.linalg.inv(A)`
Computes the inverse of the matrix.

References

- https://numpy.org/doc/stable/reference/generated/numpy.linalg.inv.html

Examples

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

# Ensure A is full rank and invertible
In [2]: while True:
   ...:     A = GF.Random((3,3))
   ...:     if np.linalg.matrix_rank(A) == 3:
   ...:         break
   ...:

In [3]: A
Out[3]:
```

(continues on next page)

(continued from previous page)

```
GF([[10,  0, 11],
    [23, 30, 15],
    [13, 30, 15]], order=31)

In [4]: A_inv = np.linalg.inv(A); A_inv
Out[4]:
GF([[ 0, 28,  3],
    [ 7, 16, 14],
    [17, 14, 17]], order=31)

In [5]: A_inv @ A
Out[5]:
GF([[1, 0, 0],
    [0, 1, 0],
    [0, 0, 1]], order=31)
```

CHAPTER

EIGHT

ACKNOWLEDGEMENTS

- This library is an extension of, and completely dependent on, NumPy.
- We heavily rely on Numba and its just-in-time compiler for optimizing performance of Galois field arithmetic.
- We use Frank Luebeck's compilation of Conway polynomials.
- We also use Wolfram's compilation of primitive polynomials.
- We extensively use SageMath for generating test vectors.
- We also use Octave for generating test vectors.

Many thanks!

CHAPTER

NINE

CITATION

If this library was useful to you in your research, please cite us. Following the GitHub citation standards, here is the recommended citation.

Bibtex:

```
@misc{Hostetter_Galois_2020,
    title = {{Galois: A performant NumPy extension for Galois fields}},
    author = {Hostetter, Matt},
    month = {11},
    year = {2020},
    url = {https://github.com/mhostetter/galois},
}
```

APA:

```
Hostetter, M. (2020). Galois: A performant NumPy extension for Galois fields [Computer␣
↪software]. https://github.com/mhostetter/galois
```

CHAPTER

TEN

RELEASE NOTES

10.1 v0.0.21

10.1.1 Changes

- Fix docstrings and code completion for Python classes that weren't rendering correctly in an IDE.
- Various documentation improvements.

10.1.2 Contributors

- Matt Hostetter (@mhostetter)

10.2 v0.0.20

10.2.1 Breaking Changes

- Move `poly_gcd()` functionality into `gcd()`.
- Move `poly_egcd()` functionality into `egcd()`.
- Move `poly_factors()` functionality into `factors()`.

10.2.2 Changes

- Fix polynomial factorization algorithms. Previously only parital factorization was implemented.
- Support generating and testing irreducible and primitive polynomials over extension fields.
- Support polynomial input to `is_square_free()`.
- Minor documentation improvements.
- Pin Numba dependency to <0.54

10.2.3 Contributors

- Matt Hostetter (@mhostetter)

10.3 v0.0.19

10.3.1 Breaking Changes

- Remove unnecessary `is_field()` function. Use `isinstance(x, galois.FieldClass)` or `isinstance(x, galois.FieldArray)` instead.
- Remove `log_naive()` function. Might be re-added later through `np.log()` on a multiplicative group array.
- Rename `mode` kwarg in `galois.GF()` to `compile`.
- Revert `np.copy()` override that always returns a subclass. Now, by default it does not return a sublcass. To return a Galois field array, use `x.copy()` or `np.copy(x, subok=True)` instead.

10.3.2 Changes

- Improve documentation.
- Improve unit test coverage.
- Add benchmarking tests.
- Add initial LFSR implementation.
- Add `display` kwarg to `galois.GF()` class factory to set the display mode at class-creation time.
- Add `Poly.reverse()` method.
- Allow polynomial strings as input to `galois.GF()`. For example, `galois.GF(2**4, irreducible_poly="x^4 + x + 1")`.
- Enable `np.divmod()` and `np.remainder()` on Galois field arrays. The remainder is always zero, though.
- Fix bug in `bch_valid_codes()` where repetition codes weren't included.
- Various minor bug fixes.

10.3.3 Contributors

- Matt Hostetter (@mhostetter)

10.4 v0.0.18

10.4.1 Breaking Changes

- Make API more consistent with software like Matlab and Wolfram:
  - Rename `galois.prime_factors()` to `galois.factors()`.
  - Rename `galois.gcd()` to `galois.egcd()` and add `galois.gcd()` for conventional GCD.

  - Rename `galois.poly_gcd()` to `galois.poly_egcd()` and add `galois.poly_gcd()` for conventional GCD.
  - Rename `galois.euler_totient()` to `galois.euler_phi()`.
  - Rename `galois.carmichael()` to `galois.carmichael_lambda()`.
  - Rename `galois.is_prime_fermat()` to `galois.fermat_primality_test()`.
  - Rename `galois.is_prime_miller_rabin()` to `galois.miller_rabin_primality_test()`.
- Rename polynomial search `method` keyword argument values from `["smallest", "largest", "random"]` to `["min", "max", "random"]`.

10.4.2 Changes

- Clean up `galois` API and `dir()` so only public classes and functions are displayed.
- Speed-up `galois.is_primitive()` test and search for primitive polynomials in `galois.primitive_poly()`.
- Speed-up `galois.is_smooth()`.
- Add Reed-Solomon codes in `galois.ReedSolomon`.
- Add shortened BCH and Reed-Solomon codes.
- Add error detection for BCH and Reed-Solomon with the `detect()` method.
- Add general cyclic linear block code functions.
- Add Matlab default primitive polynomial with `galois.matlab_primitive_poly()`.
- Add number theoretic functions:
  - Add `galois.legendre_symbol()`, `galois.jacobi_symbol()`, `galois.kronecker_symbol()`.
  - Add `galois.divisors()`, `galois.divisor_sigma()`.
  - Add `galois.is_composite()`, `galois.is_prime_power()`, `galois.is_perfect_power()`, `galois.is_square_free()`, `galois.is_powersmooth()`.
  - Add `galois.are_coprime()`.
- Clean up integer factorization algorithms and add some to public API:
  - Add `galois.perfect_power()`, `galois.trial_division()`, `galois.pollard_p1()`, `galois.pollard_rho()`.
- Clean up API reference structure and hierarchy.
- Fix minor bugs in BCH codes.

10.4.3 Contributors

- Matt Hostetter (@mhostetter)

10.5 v0.0.17

10.5.1 Breaking Changes

- Rename `FieldMeta` to `FieldClass`.
- Remove `target` keyword from `FieldClass.compile()` until there is better support for GPUs.
- Consolidate `verify_irreducible` and `verify_primitive` keyword arguments into `verify` for the `galois.GF()` class factory function.
- Remove group arrays until there is more complete support.

10.5.2 Changes

- Speed-up Galois field class creation time.
- Speed-up JIT compilation time by caching functions.
- Speed-up `Poly.roots()` by JIT compiling it.
- Add BCH codes with `galois.BCH`.
- Add ability to generate irreducible polynomials with `irreducible_poly()` and `irreducible_polys()`.
- Add ability to generate primitive polynomials with `primitive_poly()` and `primitive_polys()`.
- Add computation of the minimal polynomial of an element of an extension field with `minimal_poly()`.
- Add display of arithmetic tables with `FieldClass.arithmetic_table()`.
- Add display of field element representation table with `FieldClass.repr_table()`.
- Add Berlekamp-Massey algorithm in `berlekamp_massey()`.
- Enable ipython tab-completion of Galois field classes.
- Cleanup API reference page.
- Add introduction to Galois fields tutorials.
- Fix bug in `is_primitive()` where some reducible polynomials were marked irreducible.
- Fix bug in integer<–>polynomial conversions for large binary polynomials.
- Fix bug in "power" display mode of 0.
- Other minor bug fixes.

10.5.3 Contributors

- Dominik Wernberger (@Werni2A)
- Matt Hostetter (@mhostetter)

10.6 v0.0.16

10.6.1 Changes

- Add `Field()` alias of `GF()` class factory.
- Add finite groups modulo `n` with `Group()` class factory.
- Add `is_group()`, `is_field()`, `is_prime_field()`, `is_extension_field()`.
- Add polynomial constructor `Poly.String()`.
- Add polynomial factorization in `poly_factors()`.
- Add `np.vdot()` support.
- Fix PyPI packaging issue from v0.0.15.
- Fix bug in creation of 0-degree polynomials.
- Fix bug in `poly_gcd()` not returning monic GCD polynomials.

10.6.2 Contributors

- Matt Hostetter (@mhostetter)

10.7 v0.0.15

10.7.1 Breaking Changes

- Rename `poly_exp_mod()` to `poly_pow()` to mimic the native `pow()` function.
- Rename `fermat_primality_test()` to `is_prime_fermat()`.
- Rename `miller_rabin_primality_test()` to `is_prime_miller_rabin()`.

10.7.2 Changes

- Massive linear algebra speed-ups. (See #88)
- Massive polynomial speed-ups. (See #88)
- Various Galois field performance enhancements. (See #92)
- Support `np.convolve()` for two Galois field arrays.
- Allow polynomial arithmetic with Galois field scalars (of the same field). (See #99), e.g.

```
>>> GF = galois.GF(3)

>>> p = galois.Poly([1,2,0], field=GF)
Poly(x^2 + 2x, GF(3))

>>> p * GF(2)
Poly(2x^2 + x, GF(3))
```

- Allow creation of 0-degree polynomials from integers. (See #99), e.g.

```
>>> p = galois.Poly(1)
Poly(1, GF(2))
```

- Add the four Oakley fields from RFC 2409.
- Speed-up unit tests.
- Restructure API reference.

10.7.3 Contributors

- Matt Hostetter (@mhostetter)

10.8 v0.0.14

10.8.1 Breaking Changes

- Rename `GFArray.Eye()` to `GFArray.Identity()`.
- Rename `chinese_remainder_theorem()` to `crt()`.

10.8.2 Changes

- Lots of performance improvements.
- Additional linear algebra support.
- Various bug fixes.

10.8.3 Contributors

- Baalateja Kataru (@BK-Modding)
- Matt Hostetter (@mhostetter)

CHAPTER

ELEVEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O