galois

Matt Hostetter

Feb 02, 2024

GETTING STARTED

galois

A performant NumPy extension for Galois fields

The *galois* library is a Python 3 package that extends NumPy arrays to operate over finite fields.

The user creates a *FieldArray* subclass using `GF = galois.GF(p**m)`. `GF` is a subclass of `numpy.ndarray` and its constructor `x = GF(array_like)` mimics the signature of `numpy.array()`. The *FieldArray* `x` is operated on like any other NumPy array except all arithmetic is performed in $\mathrm{GF}(p^m)$, not $\mathbb{R}$.

Internally, the finite field arithmetic is implemented by replacing NumPy ufuncs. The new ufuncs are written in pure Python and just-in-time compiled with Numba. The ufuncs can be configured to use either lookup tables (for speed) or explicit calculation (for memory savings).

Disclaimer

The algorithms implemented in the NumPy ufuncs are not constant-time, but were instead designed for performance. As such, the library could be vulnerable to a side-channel timing attack. This library is not intended for production security, but instead for research & development, reverse engineering, cryptanalysis, experimentation, and general education.

- 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 *FieldArray* instances.
- Linear algebra over finite fields using normal `numpy.linalg` functions.
- Linear transforms over finite fields, such as the FFT with `numpy.fft.fft()` and the NTT with *ntt()*.
- Functions to generate irreducible, primitive, and Conway polynomials.
- Univariate polynomials over finite fields with *Poly*.
- Forward error correction codes with *BCH* and *ReedSolomon*.
- Fibonacci and Galois linear-feedback shift registers over any finite field with *FLFSR* and *GLFSR*.
- Various number theoretic functions.
- Integer factorization and accompanying algorithms.
- Prime number generation and primality testing.

CHAPTER

ONE

ROADMAP

- Elliptic curves over finite fields
- Galois ring arrays
- GPU support

CHAPTER

TWO

ACKNOWLEDGEMENTS

The `galois` library is an extension of, and completely dependent on, NumPy. It also heavily relies on Numba and the LLVM just-in-time compiler for optimizing performance of the finite field arithmetic.

Frank Luebeck's compilation of Conway polynomials and Wolfram's compilation of primitive polynomials are used for efficient polynomial lookup, when possible.

The Cunningham Book's tables of prime factorizations, $b^n \pm 1$ for $b \in \{2, 3, 5, 6, 7, 10, 11, 12\}$, are used to generate factorization lookup tables. These lookup tables speed-up the creation of large finite fields by avoiding the need to factor large integers.

Sage is used extensively for generating test vectors for finite field arithmetic and polynomial arithmetic. SymPy is used to generate some test vectors. Octave is used to generate test vectors for forward error correction codes.

This library would not be possible without all of the other libraries mentioned. Thank you to all their developers!

CHAPTER THREE

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

```
@software{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
```

3.1 Getting Started

The *Getting Started* guide is intended to assist the user with installing the library, creating two example arrays, and performing basic array arithmetic. See *Basic Usage* for more detailed discussions and examples.

3.1.1 Install the package

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

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

Import the `galois` package in Python.

```
In [1]: import galois

In [2]: galois.__version__
Out[2]: '0.3.8.dev8+g25baec4'
```

3.1.2 Create a `FieldArray` subclass

Next, create a *FieldArray* subclass for the specific finite field you'd like to work in. This is created using the *GF()* class factory. In this example, we are working in $\mathrm{GF}(3^5)$.

```
In [3]: GF = galois.GF(3**5)

In [4]: print(GF.properties)
Galois Field:
  name: GF(3^5)
  characteristic: 3
  degree: 5
  order: 243
  irreducible_poly: x^5 + 2x + 1
  is_primitive_poly: True
  primitive_element: x
```

The *FieldArray* subclass `GF` is a subclass of `ndarray` that performs all arithmetic in the Galois field $\mathrm{GF}(3^5)$, not in $\mathbb{R}$.

```
In [5]: issubclass(GF, galois.FieldArray)
Out[5]: True

In [6]: issubclass(GF, np.ndarray)
Out[6]: True
```

See *Array Classes* for more details.

3.1.3 Create two `FieldArray` instances

Next, create a new *FieldArray* `x` by passing an *ArrayLike* object to `GF`'s constructor.

```
In [7]: x = GF([236, 87, 38, 112]); x
Out[7]: GF([236,  87,  38, 112], order=3^5)
```

The array `x` is an instance of *FieldArray* and also an instance of `ndarray`.

```
In [8]: isinstance(x, galois.FieldArray)
Out[8]: True

In [9]: isinstance(x, np.ndarray)
Out[9]: True
```

Create a second *FieldArray* `y` by converting an existing NumPy array (without copying it) by invoking `.view()`. When finished working in the finite field, view it back as a NumPy array with `.view(np.ndarray)`.

```
# y represents an array created elsewhere in the code
In [10]: y = np.array([109, 17, 108, 224]); y
Out[10]: array([109,  17, 108, 224])

In [11]: y = y.view(GF); y
Out[11]: GF([109,  17, 108, 224], order=3^5)
```

See *Array Creation* for more details.

3.1.4 Change the element representation

The representation of finite field elements can be set to either the integer ("int"), polynomial ("poly"), or power ("power") representation. The default representation is the integer representation since integers are natural when working with integer NumPy arrays.

Set the element representation by passing the `repr` keyword argument to *GF()* or by calling the *repr()* classmethod. Choose whichever element representation is most convenient.

```
# The default is the integer representation
In [12]: x
Out[12]: GF([236,  87,  38, 112], order=3^5)

In [13]: GF.repr("poly"); x
Out[13]:
GF([2^4 + 2^3 + 2^2 + 2,               ^4 + 2,
             ^3 + ^2 + 2,      ^4 + ^3 +  + 1], order=3^5)

In [14]: GF.repr("power"); x
Out[14]: GF([^204,  ^16, ^230,  ^34], order=3^5)

# Reset to the integer representation
In [15]: GF.repr("int");
```

See *Element Representation* for more details.

3.1.5 Perform array arithmetic

Once you have two Galois field arrays, nearly any arithmetic operation can be performed using normal NumPy arithmetic. The traditional NumPy broadcasting rules apply.

Standard element-wise array arithmetic – addition, subtraction, multiplication, and division – are easily preformed.

```
In [16]: x + y
Out[16]: GF([ 18,  95, 146,   0], order=3^5)

In [17]: x - y
Out[17]: GF([127, 100, 173, 224], order=3^5)

In [18]: x * y
Out[18]: GF([ 21, 241, 179,  82], order=3^5)

In [19]: x / y
Out[19]: GF([ 67,  47, 192,   2], order=3^5)
```

More complicated arithmetic, like square root and logarithm base α, are also supported.

```
In [20]: np.sqrt(x)
Out[20]: GF([ 51, 135,  40,  16], order=3^5)

In [21]: np.log(x)
Out[21]: array([204,  16, 230,  34])
```

See *Array Arithmetic* for more details.

3.2 Array Classes

The *galois* library subclasses `ndarray` to provide arithmetic over Galois fields and rings (future).

3.2.1 `Array` subclasses

The main abstract base class is *Array*. It has two abstract subclasses: *FieldArray* and `RingArray` (future). None of these abstract classes may be instantiated directly. Instead, specific subclasses for $\mathrm{GF}(p^m)$ and $\mathrm{GR}(p^e, m)$ are created at runtime with *GF()* and `GR()` (future).

3.2.2 `FieldArray` subclasses

A *FieldArray* subclass is created using the class factory function *GF()*.

Integer

```
In [1]: GF = galois.GF(3**5)

In [2]: print(GF.properties)
Galois Field:
  name: GF(3^5)
  characteristic: 3
  degree: 5
  order: 243
  irreducible_poly: x^5 + 2x + 1
  is_primitive_poly: True
  primitive_element: x
```

Polynomial

```
In [3]: GF = galois.GF(3**5, repr="poly")

In [4]: print(GF.properties)
Galois Field:
  name: GF(3^5)
  characteristic: 3
  degree: 5
  order: 243
  irreducible_poly: x^5 + 2x + 1
  is_primitive_poly: True
  primitive_element: x
```

Power

```
In [5]: GF = galois.GF(3**5, repr="power")

In [6]: print(GF.properties)
Galois Field:
  name: GF(3^5)
  characteristic: 3
  degree: 5
  order: 243
  irreducible_poly: x^5 + 2x + 1
  is_primitive_poly: True
  primitive_element: x
```

Speed up creation of large finite field classes

For very large finite fields, the *FieldArray* subclass creation time can be reduced by explicitly specifying p and m. This eliminates the need to factor the order p^m.

```
In [7]: GF = galois.GF(2, 100)

In [8]: print(GF.properties)
Galois Field:
  name: 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
```

Furthermore, if you already know the desired irreducible polynomial is irreducible and the primitive element is a generator of the multiplicative group, you can specify `verify=False` to skip the verification step. This eliminates the need to factor $p^m - 1$.

```
In [9]: GF = galois.GF(109987, 4, irreducible_poly="x^4 + 3x^2 + 100525x + 3", primitive_
↪element="x", verify=False)

In [10]: print(GF.properties)
Galois Field:
  name: GF(109987^4)
  characteristic: 109987
  degree: 4
  order: 146340800268433348561
  irreducible_poly: x^4 + 3x^2 + 100525x + 3
  is_primitive_poly: True
  primitive_element: x
```

The `GF` class is a subclass of *FieldArray* and a subclasses of `ndarray`.

```
In [11]: issubclass(GF, galois.FieldArray)
Out[11]: True

In [12]: issubclass(GF, galois.Array)
Out[12]: True

In [13]: issubclass(GF, np.ndarray)
Out[13]: True
```

Class singletons

FieldArray subclasses of the same type (order, irreducible polynomial, and primitive element) are singletons.

Here is the creation (twice) of the field $\mathrm{GF}(3^5)$ defined with the default irreducible polynomial $x^5 + 2x + 1$. They *are* the same class (a singleton), not just equivalent classes.

```
In [14]: galois.GF(3**5) is galois.GF(3**5)
Out[14]: True
```

The expense of class creation is incurred only once. So, subsequent calls of `galois.GF(3**5)` are extremely inexpensive.

However, the field $\mathrm{GF}(3^5)$ defined with irreducible polynomial $x^5 + x^2 + x + 2$, while isomorphic to the first field, has different arithmetic. As such, *GF()* returns a unique *FieldArray* subclass.

```
In [15]: galois.GF(3**5) is galois.GF(3**5, irreducible_poly="x^5 + x^2 + x + 2")
Out[15]: False
```

Methods and properties

All of the methods and properties related to $\mathrm{GF}(p^m)$, not one of its arrays, are documented as class methods and class properties in *FieldArray*. For example, the irreducible polynomial of the finite field is accessed with *irreducible_poly*.

```
In [16]: GF.irreducible_poly
Out[16]: Poly(x^5 + 2x + 1, GF(3))
```

3.2.3 `FieldArray` instances

A *FieldArray* instance is created using `GF`'s constructor.

Integer

```
In [17]: x = GF([23, 78, 163, 124])

In [18]: x
Out[18]: GF([ 23,  78, 163, 124], order=3^5)
```

Polynomial

```
In [19]: x = GF([23, 78, 163, 124])

In [20]: x
Out[20]:
GF([            2^2 +   + 2,          2^3 + 2^2 + 2,
                    2^4 + 1, ^4 + ^3 + ^2 + 2 + 1], order=3^5)
```

Power

```
In [21]: x = GF([23, 78, 163, 124])

In [22]: x
Out[22]: GF([ ^17, ^132, ^241,  ^41], order=3^5)
```

The array `x` is an instance of *`FieldArray`* and also an instance of `ndarray`.

```
In [23]: isinstance(x, GF)
Out[23]: True

In [24]: isinstance(x, galois.FieldArray)
Out[24]: True

In [25]: isinstance(x, galois.Array)
Out[25]: True

In [26]: isinstance(x, np.ndarray)
Out[26]: True
```

The *`FieldArray`* subclass is easily recovered from a *`FieldArray`* instance using `type()`.

```
In [27]: type(x) is GF
Out[27]: True
```

Constructors

Several classmethods are defined in *FieldArray* that function as alternate constructors. By convention, alternate constructors use `PascalCase` while other classmethods use `snake_case`.

For example, to generate a random array of given shape call *Random()*.

Integer

```
In [28]: GF.Random((3, 2), seed=1)
Out[28]:
GF([[242, 216],
    [ 32, 114],
    [230, 179]], order=3^5)
```

Polynomial

```
In [29]: GF.Random((3, 2), seed=1)
Out[29]:
GF([[2^4 + 2^3 + 2^2 + 2 + 2,                  2^4 + 2^3],
    [                ^3 +   + 2,             ^4 + ^3 + 2],
    [  2^4 + 2^3 + ^2 +   + 2,          2^4 + ^2 + 2 + 2]], order=3^5)
```

Power

```
In [30]: GF.Random((3, 2), seed=1)
Out[30]:
GF([[^185, ^193],
    [ ^49, ^231],
    [ ^81,  ^60]], order=3^5)
```

Or, create an identity matrix using *Identity()*.

Integer

```
In [31]: GF.Identity(4)
Out[31]:
GF([[1, 0, 0, 0],
    [0, 1, 0, 0],
    [0, 0, 1, 0],
    [0, 0, 0, 1]], order=3^5)
```

Polynomial

```
In [32]: GF.Identity(4)
Out[32]:
GF([[1, 0, 0, 0],
    [0, 1, 0, 0],
    [0, 0, 1, 0],
    [0, 0, 0, 1]], order=3^5)
```

Power

```
In [33]: GF.Identity(4)
Out[33]:
GF([[1, 0, 0, 0],
    [0, 1, 0, 0],
    [0, 0, 1, 0],
    [0, 0, 0, 1]], order=3^5)
```

Methods

All of the methods that act on *FieldArray* instances are documented as instance methods in *FieldArray*. For example, the multiplicative order of each finite field element is calculated using *multiplicative_order()*.

```
In [34]: x.multiplicative_order()
Out[34]: array([242,  11, 242, 242])
```

3.3 Compilation Modes

The *galois* library supports finite field arithmetic on NumPy arrays by just-in-time compiling custom NumPy ufuncs. It uses Numba to JIT compile ufuncs written in pure Python. The created *FieldArray* subclass `GF` intercepts NumPy calls to a given ufunc, JIT compiles the finite field ufunc (if not already cached), and then invokes the new ufunc on the input array(s).

There are two primary compilation modes: `"jit-lookup"` and `"jit-calculate"`. The supported ufunc compilation modes of a given finite field are listed in *ufunc_modes*.

```
In [1]: GF = galois.GF(3**5)

In [2]: GF.ufunc_modes
Out[2]: ['jit-lookup', 'jit-calculate']
```

Large finite fields, which have `numpy.object_` data types, use `"python-calculate"` which utilizes non-compiled, pure-Python ufuncs.

```
In [3]: GF = galois.GF(2**100)

In [4]: GF.ufunc_modes
Out[4]: ['python-calculate']
```

3.3.1 Lookup tables

The lookup table compilation mode "jit-lookup" uses exponential, logarithm, and Zech logarithm lookup tables to speed up arithmetic computations. These tables are built once at *FieldArray* subclass-creation time during the call to *GF()*.

The exponential and logarithm lookup tables map every finite field element to a power of the primitive element α.

$$x = \alpha^i$$

$$\log_\alpha(x) = i$$

With these lookup tables, many arithmetic operations are simplified. For instance, multiplication of two finite field elements is reduced to three lookups and one integer addition.

$$\begin{aligned} x \cdot y &= \alpha^m \cdot \alpha^n \\ &= \alpha^{m+n} \end{aligned}$$

The Zech logarithm is defined below. A similar lookup table is created for it.

$$1 + \alpha^i = \alpha^{Z(i)}$$

$$Z(i) = \log_\alpha(1 + \alpha^i)$$

With Zech logarithms, addition of two finite field elements becomes three lookups, one integer addition, and one integer subtraction.

$$\begin{aligned} x + y &= \alpha^m + \alpha^n \\ &= \alpha^m(1 + \alpha^{n-m}) \\ &= \alpha^m \alpha^{Z(n-m)} \\ &= \alpha^{m+Z(n-m)} \end{aligned}$$

Finite fields with order less than 2^{20} use lookup tables by default. In the limited cases where explicit calculation is faster than table lookup, the explicit calculation is used.

```
In [5]: GF = galois.GF(3**5)

In [6]: GF.ufunc_mode
Out[6]: 'jit-lookup'
```

3.3.2 Explicit calculation

Finite fields with order greater than 2^{20} use explicit calculation by default. This eliminates the need to store large lookup tables. However, explicit calculation is usually slower than table lookup.

```
In [7]: GF = galois.GF(2**24)

In [8]: GF.ufunc_mode
Out[8]: 'jit-calculate'
```

However, if memory is of no concern, even large fields can be compiled to use lookup tables. Initially constructing the lookup tables may take some time, however.

```
In [9]: GF = galois.GF(2**24, compile="jit-lookup")

In [10]: GF.ufunc_mode
Out[10]: 'jit-lookup'
```

3.3.3 Python explicit calculation

Large finite fields cannot use JIT compiled ufuncs. This is because they cannot use NumPy integer data types. This is either because the order of the field or an intermediate arithmetic result is larger than the max value of `numpy.int64`.

These finite fields use the `numpy.object_` data type and have ufunc compilation mode `"python-calculate"`. This mode does *not* compile the Python functions, but rather converts them into Python ufuncs using `numpy.frompyfunc()`. The lack of JIT compilation allows the ufuncs to operate on Python integers, which have unlimited size. This does come with a performance penalty, however.

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

In [12]: GF.ufunc_mode
Out[12]: 'python-calculate'
```

3.3.4 Recompile the ufuncs

The compilation mode may be explicitly set during creation of the *FieldArray* subclass using the `compile` keyword argument to *GF()*.

Here, the *FieldArray* subclass for $\mathrm{GF}(3^5)$ would normally select `"jit-lookup"` as its default compilation mode. However, we can intentionally choose explicit calculation.

```
In [13]: GF = galois.GF(3**5, compile="jit-calculate")

In [14]: GF.ufunc_mode
Out[14]: 'jit-calculate'
```

After a *FieldArray* subclass has been created, its compilation mode may be changed using the *compile()* method.

```
In [15]: GF.compile("jit-lookup")

In [16]: GF.ufunc_mode
Out[16]: 'jit-lookup'
```

This will not immediately recompile all of the ufuncs. The ufuncs are compiled on-demand (during their first invocation) and only if a cached version is not available.

3.4 Element Representation

The representation of finite field elements can be set to either their integer ("int"), polynomial ("poly"), or power ("power") representation.

In prime fields $\mathrm{GF}(p)$, elements are integers in $\{0, 1, \dots, p-1\}$. Their two useful representations are the integer and power representation.

In extension fields $\mathrm{GF}(p^m)$, elements are polynomials over $\mathrm{GF}(p)$ with degree less than m. All element representations are useful. The polynomial representation allows *proper* representation of the element as a polynomial over its prime subfield. However, the integer representation is more compact for displaying large arrays.

3.4.1 Set the element representation

The field element representation can be set during *FieldArray* subclass creation by passing the `repr` keyword argument to the *GF()* class factory.

```
In [1]: GF = galois.GF(3**5, repr="poly")

In [2]: x = GF([17, 4])

In [3]: x
Out[3]: GF([^2 + 2 + 2,          + 1], order=3^5)

In [4]: print(x)
[^2 + 2 + 2          + 1]
```

The current element representation is accessed with the *element_repr* class property.

```
In [5]: GF.element_repr
Out[5]: 'poly'
```

The element representation can be temporarily changed using the *repr()* classmethod as a context manager.

```
# Inside the context manager, x prints using the power representation
In [6]: with GF.repr("power"):
   ...:     print(x)
   ...:
[^222  ^69]

# Outside the context manager, x prints using the previous representation
In [7]: print(x)
[^2 + 2 + 2          + 1]
```

The element representation can be permanently changed using the *repr()* classmethod (not as a context manager).

```
# The old polynomial representation
In [8]: x
Out[8]: GF([^2 + 2 + 2,          + 1], order=3^5)
```

(continues on next page)

(continued from previous page)

```
In [9]: GF.repr("int");

# The new integer representation
In [10]: x
Out[10]: GF([17,  4], order=3^5)
```

3.4.2 Integer representation

The integer representation (the default) displays all finite field elements as integers in $\{0, 1, \dots, p^m - 1\}$.

In prime fields, the integer representation is simply the integer element in $\{0, 1, \dots, p - 1\}$.

```
In [11]: GF = galois.GF(31)

In [12]: GF(11)
Out[12]: GF(11, order=31)
```

In extension fields, the integer representation converts an element's degree-$m - 1$ polynomial over $\mathrm{GF}(p)$ into its integer equivalent. The integer equivalent of a polynomial is a radix-p integer of its coefficients, with the highest-degree coefficient as the most-significant digit and zero-degree coefficient as the least-significant digit.

```
In [13]: GF = galois.GF(3**5)

In [14]: GF(17)
Out[14]: GF(17, order=3^5)

In [15]: GF("x^2 + 2x + 2")
Out[15]: GF(17, order=3^5)

# Integer/polynomial equivalence
In [16]: p = 3; p**2 + 2*p + 2 == 17
Out[16]: True
```

3.4.3 Polynomial representation

The polynomial representation displays all finite field elements as polynomials over their prime subfield with degree less than m.

In prime fields $m = 1$, therefore the polynomial representation is equivalent to the integer representation because the polynomials all have degree 0.

```
In [17]: GF = galois.GF(31, repr="poly")

In [18]: GF(11)
Out[18]: GF(11, order=31)
```

In extension fields, the polynomial representation displays the elements naturally as polynomials over their prime subfield. This is useful, however it can become cluttered for large arrays.

```
In [19]: GF = galois.GF(3**5, repr="poly")

In [20]: GF(17)
Out[20]: GF(^2 + 2 + 2, order=3^5)

In [21]: GF("x^2 + 2x + 2")
Out[21]: GF(^2 + 2 + 2, order=3^5)

# Integer/polynomial equivalence
In [22]: p = 3; p**2 + 2*p + 2 == 17
Out[22]: True
```

Tip

Use `set_printoptions()` to display the polynomial coefficients in degree-ascending order. Use `numpy.set_printoptions()` to increase the line width to display large arrays more clearly. See *NumPy print options* for more details.

3.4.4 Power representation

The power representation displays all finite field elements as powers of the field's primitive element α.

Slower performance

To display elements in the power representation, `galois` must compute the discrete logarithm of each element displayed. For large fields or fields using *explicit calculation*, this process can take a while. However, when using *lookup tables* this representation is just as fast as the others.

In prime fields, the elements are displayed as $\{0, 1, \alpha, \alpha^2, \dots, \alpha^{p-2}\}$.

```
In [23]: GF = galois.GF(31, repr="power")

In [24]: GF(11)
Out[24]: GF(^23, order=31)
```

```
In [25]: GF.repr("int");

In [26]: alpha = GF.primitive_element; alpha
Out[26]: GF(3, order=31)

In [27]: alpha ** 23
Out[27]: GF(11, order=31)
```

In extension fields, the elements are displayed as $\{0, 1, \alpha, \alpha^2, \dots, \alpha^{p^m-2}\}$.

```
In [28]: GF = galois.GF(3**5, repr="power")

In [29]: GF(17)
Out[29]: GF(^222, order=3^5)
```

```
In [30]: GF.repr("int");

In [31]: alpha = GF.primitive_element; alpha
Out[31]: GF(3, order=3^5)

In [32]: alpha ** 222
Out[32]: GF(17, order=3^5)
```

3.4.5 Vector representation

The vector representation, while not a valid input to *repr()*, represents finite field elements as vectors of their polynomial coefficients.

The vector representation is accessed using the *vector()* method.

```
In [33]: GF = galois.GF(3**5, repr="poly")

In [34]: GF("x^2 + 2x + 2")
Out[34]: GF(^2 + 2 + 2, order=3^5)

In [35]: GF("x^2 + 2x + 2").vector()
Out[35]: GF([0, 0, 1, 2, 2], order=3)
```

An N-D array over $\mathrm{GF}(p^m)$ is converted to a (N + 1)-D array over $\mathrm{GF}(p)$ with the added dimension having size m. The first value of the vector is the highest-degree coefficient.

```
In [36]: GF(["x^2 + 2x + 2", "2x^4 + x"])
Out[36]: GF([^2 + 2 + 2,     2^4 + ], order=3^5)

In [37]: GF(["x^2 + 2x + 2", "2x^4 + x"]).vector()
Out[37]:
GF([[0, 0, 1, 2, 2],
    [2, 0, 0, 1, 0]], order=3)
```

Arrays can be created from the vector representation using the *Vector()* classmethod.

```
In [38]: GF.Vector([[0, 0, 1, 2, 2], [2, 0, 0, 1, 0]])
Out[38]: GF([^2 + 2 + 2,     2^4 + ], order=3^5)
```

3.4.6 NumPy print options

NumPy displays arrays with a default line width of 75 characters. This is problematic for large arrays. It is especially problematic for arrays using the polynomial representation, where each element occupies a lot of space. This can be changed by modifying NumPy's print options.

For example, below is a 5×5 matrix over $\mathrm{GF}(3^5)$ displayed in the polynomial representation. With the default line width, the array is quite difficult to read.

```
In [39]: GF = galois.GF(3**5, repr="poly")

In [40]: x = GF.Random((5, 5)); x
Out[40]:
```

(continues on next page)

(continued from previous page)

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

The readability is improved by increasing the line width using `numpy.set_printoptions()`.

```
In [41]: np.set_printoptions(linewidth=200)

In [42]: x
Out[42]:
GF([[          ^3 + ^2 +  + 1,  ^4 + 2^3 + 2^2 + 2 + 2,      2^4 + 2^3 + 2^2 + 2,       2^
↪4 + 2^3 + ^2 + 2,            2^4 + 2^2 + 2],
    [               ^4 + 2 + 1,        2^4 + ^3 + ^2 + 1,       2^4 + ^3 + 2^2 + 2,      ␣
↪    ^4 + ^2 + 2 + 1,               2^3 +  + 2],
    [                    ^2 + 1,        2^3 + 2^2 + 2 + 2,                   2^2 + 2,    ␣
↪      ^4 + ^3 + 2 + 2,               ^4 + ^2 + ],
    [2^4 + 2^3 + 2^2 + 2 + 1,     2^4 + 2^3 + 2^2 + 2,              2^2 + 2 + 2,         ␣
↪                  ^3,          2^4 + ^3 + ^2 + ],
    [             2^4 + ^2 + 1,                2^4 +  + 2,                  ^4 + ^2,     ␣
↪      ^4 + 2^2 +  + 2,                    ^4 + 1]], order=3^5)
```

3.4.7 Representation comparisons

For any finite field, each of the four element representations can be easily compared using the *repr_table()* class-method.

```
In [43]: GF = galois.GF(3**3)

In [44]: print(GF.repr_table())
 Power     Polynomial       Vector     Integer
------- --------------- ----------- ---------
   0           0         [0, 0, 0]      0
  x^0          1         [0, 0, 1]      1
  x^1          x         [0, 1, 0]      3
  x^2         x^2        [1, 0, 0]      9
  x^3        x + 2       [0, 1, 2]      5
  x^4       x^2 + 2x     [1, 2, 0]      15
  x^5     2x^2 + x + 2   [2, 1, 2]      23
  x^6     x^2 + x + 1    [1, 1, 1]      13
```

(continues on next page)

(continued from previous page)

```
x^7     x^2 + 2x + 2   [1, 2, 2]      17
x^8       2x^2 + 2     [2, 0, 2]      20
x^9        x + 1       [0, 1, 1]      4
x^10      x^2 + x      [1, 1, 0]      12
x^11    x^2 + x + 2    [1, 1, 2]      14
x^12      x^2 + 2      [1, 0, 2]      11
x^13         2         [0, 0, 2]      2
x^14         2x        [0, 2, 0]      6
x^15        2x^2       [2, 0, 0]      18
x^16       2x + 1      [0, 2, 1]      7
x^17      2x^2 + x     [2, 1, 0]      21
x^18    x^2 + 2x + 1   [1, 2, 1]      16
x^19   2x^2 + 2x + 2   [2, 2, 2]      26
x^20    2x^2 + x + 1   [2, 1, 1]      22
x^21      x^2 + 1      [1, 0, 1]      10
x^22       2x + 2      [0, 2, 2]      8
x^23     2x^2 + 2x     [2, 2, 0]      24
x^24   2x^2 + 2x + 1   [2, 2, 1]      25
x^25      2x^2 + 1     [2, 0, 1]      19
```

3.5 Array Creation

This page discusses the multiple ways to create arrays over finite fields. For this discussion, we are working in the finite field $\mathrm{GF}(3^5)$.

Integer

```
In [1]: GF = galois.GF(3**5)

In [2]: print(GF.properties)
Galois Field:
  name: GF(3^5)
  characteristic: 3
  degree: 5
  order: 243
  irreducible_poly: x^5 + 2x + 1
  is_primitive_poly: True
  primitive_element: x

In [3]: alpha = GF.primitive_element; alpha
Out[3]: GF(3, order=3^5)
```

Polynomial

```
In [4]: GF = galois.GF(3**5, repr="poly")

In [5]: print(GF.properties)
Galois Field:
  name: GF(3^5)
  characteristic: 3
  degree: 5
  order: 243
  irreducible_poly: x^5 + 2x + 1
  is_primitive_poly: True
  primitive_element: x

In [6]: alpha = GF.primitive_element; alpha
Out[6]: GF(, order=3^5)
```

Power

```
In [7]: GF = galois.GF(3**5, repr="power")

In [8]: print(GF.properties)
Galois Field:
  name: GF(3^5)
  characteristic: 3
  degree: 5
  order: 243
  irreducible_poly: x^5 + 2x + 1
  is_primitive_poly: True
  primitive_element: x

In [9]: alpha = GF.primitive_element; alpha
Out[9]: GF(, order=3^5)
```

3.5.1 Create a scalar

A single finite field element (a scalar) is a 0-D *FieldArray*. They are created by passing a single *ElementLike* object to GF's constructor. A finite field scalar may also be created by exponentiating the primitive element to a scalar power.

Integer

```
In [10]: GF(17)
Out[10]: GF(17, order=3^5)

In [11]: GF("x^2 + 2x + 2")
Out[11]: GF(17, order=3^5)

In [12]: alpha ** 222
Out[12]: GF(17, order=3^5)
```

Polynomial

```
In [13]: GF(17)
Out[13]: GF(^2 + 2 + 2, order=3^5)

In [14]: GF("x^2 + 2x + 2")
Out[14]: GF(^2 + 2 + 2, order=3^5)

In [15]: alpha ** 222
Out[15]: GF(^2 + 2 + 2, order=3^5)
```

Power

```
In [16]: GF(17)
Out[16]: GF(^222, order=3^5)

In [17]: GF("x^2 + 2x + 2")
Out[17]: GF(^222, order=3^5)

In [18]: alpha ** 222
Out[18]: GF(^222, order=3^5)
```

3.5.2 Create a new array

Array-like objects

A *FieldArray* can be created from various *ArrayLike* objects. A finite field array may also be created by exponentiating the primitive element to a an array of powers.

Integer

```
In [19]: GF([17, 4, 148, 205])
Out[19]: GF([ 17,   4, 148, 205], order=3^5)

In [20]: GF([["x^2 + 2x + 2", 4], ["x^4 + 2x^3 + x^2 + x + 1", 205]])
Out[20]:
GF([[ 17,   4],
    [148, 205]], order=3^5)

In [21]: alpha ** np.array([[222, 69], [54, 24]])
Out[21]:
GF([[ 17,   4],
    [148, 205]], order=3^5)
```

Polynomial

```
In [22]: GF([17, 4, 148, 205])
Out[22]:
GF([             ^2 + 2 + 2,                      + 1,
     ^4 + 2^3 + ^2 +  + 1, 2^4 + ^3 + ^2 + 2 + 1], order=3^5)

In [23]: GF([["x^2 + 2x + 2", 4], ["x^4 + 2x^3 + x^2 + x + 1", 205]])
Out[23]:
GF([[             ^2 + 2 + 2,                      + 1],
    [ ^4 + 2^3 + ^2 +  + 1, 2^4 + ^3 + ^2 + 2 + 1]], order=3^5)

In [24]: alpha ** np.array([[222, 69], [54, 24]])
Out[24]:
GF([[             ^2 + 2 + 2,                      + 1],
    [ ^4 + 2^3 + ^2 +  + 1, 2^4 + ^3 + ^2 + 2 + 1]], order=3^5)
```

Power

```
In [25]: GF([17, 4, 148, 205])
Out[25]: GF([^222,  ^69,  ^54,  ^24], order=3^5)

In [26]: GF([["x^2 + 2x + 2", 4], ["x^4 + 2x^3 + x^2 + x + 1", 205]])
Out[26]:
GF([[^222,  ^69],
    [ ^54,  ^24]], order=3^5)

In [27]: alpha ** np.array([[222, 69], [54, 24]])
Out[27]:
GF([[^222,  ^69],
    [ ^54,  ^24]], order=3^5)
```

Polynomial coefficients

Rather than strings, the polynomial coefficients may be passed into GF's constructor as length-m vectors using the *Vector()* classmethod.

Integer

```
In [28]: GF.Vector([[0, 0, 1, 2, 2], [0, 0, 0, 1, 1]])
Out[28]: GF([17,  4], order=3^5)
```

Polynomial

```
In [29]: GF.Vector([[0, 0, 1, 2, 2], [0, 0, 0, 1, 1]])
Out[29]: GF([^2 + 2 + 2,          + 1], order=3^5)
```

Power

```
In [30]: GF.Vector([[0, 0, 1, 2, 2], [0, 0, 0, 1, 1]])
Out[30]: GF([^222,  ^69], order=3^5)
```

The *vector()* method is the opposite operation. It converts extension field elements from $\mathrm{GF}(p^m)$ into length-m vectors over $\mathrm{GF}(p)$.

Integer

```
In [31]: GF([17, 4]).vector()
Out[31]:
GF([[0, 0, 1, 2, 2],
    [0, 0, 0, 1, 1]], order=3)
```

Polynomial

```
In [32]: GF([17, 4]).vector()
Out[32]:
GF([[0, 0, 1, 2, 2],
    [0, 0, 0, 1, 1]], order=3)
```

Power

```
In [33]: GF([17, 4]).vector()
Out[33]:
GF([[0, 0, 1, 2, 2],
    [0, 0, 0, 1, 1]], order=3)
```

NumPy array

An integer NumPy array may also be passed into `GF`. The default keyword argument `copy=True` of the *FieldArray* constructor will create a copy of the array.

Integer

```
In [34]: x_np = np.array([213, 167, 4, 214, 209]); x_np
Out[34]: array([213, 167,   4, 214, 209])

In [35]: x = GF(x_np); x
Out[35]: GF([213, 167,   4, 214, 209], order=3^5)

# Modifying x does not modify x_np
In [36]: x[0] = 0; x_np
Out[36]: array([213, 167,   4, 214, 209])
```

Polynomial

```
In [37]: x_np = np.array([213, 167, 4, 214, 209]); x_np
Out[37]: array([213, 167,   4, 214, 209])

In [38]: x = GF(x_np); x
Out[38]:
GF([    2^4 + ^3 + 2^2 + 2,               2^4 +  + 2,
                         + 1, 2^4 + ^3 + 2^2 + 2 + 1,
         2^4 + ^3 + 2^2 + 2], order=3^5)

# Modifying x does not modify x_np
In [39]: x[0] = 0; x_np
Out[39]: array([213, 167,   4, 214, 209])
```

Power

```
In [40]: x_np = np.array([213, 167, 4, 214, 209]); x_np
Out[40]: array([213, 167,   4, 214, 209])

In [41]: x = GF(x_np); x
Out[41]: GF([^183,   ^9,  ^69, ^153,  ^58], order=3^5)

# Modifying x does not modify x_np
In [42]: x[0] = 0; x_np
Out[42]: array([213, 167,   4, 214, 209])
```

3.5.3 View an existing array

Instead of creating a *FieldArray* explicitly, you can convert an existing NumPy array into a *FieldArray* temporarily and work with it in-place.

Simply call `.view(GF)` to *view* the NumPy array as a *FieldArray*. When finished working in the finite field, call `.view(np.ndarray)` to *view* it back to a NumPy array.

Integer

```
In [43]: x_np = np.array([213, 167, 4, 214, 209], dtype=int); x_np
Out[43]: array([213, 167,   4, 214, 209])

In [44]: x = x_np.view(GF); x
Out[44]: GF([213, 167,   4, 214, 209], order=3^5)

# Modifying x does modify x_np!
In [45]: x[0] = 0; x_np
Out[45]: array([  0, 167,   4, 214, 209])
```

Polynomial

```
In [46]: x_np = np.array([213, 167, 4, 214, 209], dtype=int); x_np
Out[46]: array([213, 167,   4, 214, 209])

In [47]: x = x_np.view(GF); x
Out[47]:
GF([    2^4 + ^3 + 2^2 + 2,               2^4 +  + 2,
                          + 1, 2^4 + ^3 + 2^2 + 2 + 1,
         2^4 + ^3 + 2^2 + 2], order=3^5)

# Modifying x does modify x_np!
In [48]: x[0] = 0; x_np
Out[48]: array([  0, 167,   4, 214, 209])
```

Power

```
In [49]: x_np = np.array([213, 167, 4, 214, 209], dtype=int); x_np
Out[49]: array([213, 167,   4, 214, 209])

In [50]: x = x_np.view(GF); x
Out[50]: GF([^183,   ^9,  ^69, ^153,  ^58], order=3^5)

# Modifying x does modify x_np!
In [51]: x[0] = 0; x_np
Out[51]: array([  0, 167,   4, 214, 209])
```

3.5.4 Classmethods

Several classmethods are provided in *FieldArray* to assist with creating arrays.

Constant arrays

The *Zeros()* and *Ones()* classmethods provide constant arrays that are useful for initializing empty arrays.

Integer

```
In [52]: GF.Zeros(4)
Out[52]: GF([0, 0, 0, 0], order=3^5)

In [53]: GF.Ones(4)
Out[53]: GF([1, 1, 1, 1], order=3^5)
```

Polynomial

```
In [54]: GF.Zeros(4)
Out[54]: GF([0, 0, 0, 0], order=3^5)

In [55]: GF.Ones(4)
Out[55]: GF([1, 1, 1, 1], order=3^5)
```

Power

```
In [56]: GF.Zeros(4)
Out[56]: GF([0, 0, 0, 0], order=3^5)

In [57]: GF.Ones(4)
Out[57]: GF([1, 1, 1, 1], order=3^5)
```

There is no `numpy.empty()` equivalent.

This is because *FieldArray* instances must have values in $[0, p^m)$. Empty NumPy arrays have whatever values are currently in memory, and therefore would fail those bounds checks during instantiation.

Ordered arrays

The *Range()* classmethod produces a range of elements similar to `numpy.arange()`. The integer `start` and `stop` values are the *integer representation* of the polynomial field elements.

Integer

```
In [58]: GF.Range(10, 20)
Out[58]: GF([10, 11, 12, 13, 14, 15, 16, 17, 18, 19], order=3^5)

In [59]: GF.Range(10, 20, 2)
Out[59]: GF([10, 12, 14, 16, 18], order=3^5)
```

Polynomial

```
In [60]: GF.Range(10, 20)
Out[60]:
GF([     ^2 + 1,       ^2 + 2,       ^2 + ,   ^2 +  + 1,  ^2 +  + 2,
        ^2 + 2, ^2 + 2 + 1, ^2 + 2 + 2,         2^2,      2^2 + 1],
   order=3^5)

In [61]: GF.Range(10, 20, 2)
Out[61]:
GF([     ^2 + 1,       ^2 + ,   ^2 +  + 2, ^2 + 2 + 1,         2^2],
   order=3^5)
```

Power

```
In [62]: GF.Range(10, 20)
Out[62]:
GF([ ^46,  ^74,  ^70,  ^10, ^209,   ^6, ^138, ^222, ^123, ^195],
   order=3^5)

In [63]: GF.Range(10, 20, 2)
Out[63]: GF([ ^46,  ^70, ^209, ^138, ^123], order=3^5)
```

Random arrays

The *Random()* classmethod provides a random array of the specified shape. This is convenient for testing. The integer `low` and `high` values are the *integer representation* of the polynomial field elements.

Integer

```
In [64]: GF.Random(4, seed=1)
Out[64]: GF([242, 216,  32, 114], order=3^5)

In [65]: GF.Random(4, low=10, high=20, seed=2)
Out[65]: GF([17, 13, 14, 18], order=3^5)
```

Polynomial

```
In [66]: GF.Random(4, seed=1)
Out[66]:
GF([2^4 + 2^3 + 2^2 + 2 + 2,                    2^4 + 2^3,
                      ^3 +   + 2,               ^4 + ^3 + 2], order=3^5)

In [67]: GF.Random(4, low=10, high=20, seed=2)
Out[67]: GF([^2 + 2 + 2,   ^2 +   + 1,   ^2 +   + 2,          2^2], order=3^5)
```

Power

```
In [68]: GF.Random(4, seed=1)
Out[68]: GF([^185, ^193,  ^49, ^231], order=3^5)

In [69]: GF.Random(4, low=10, high=20, seed=2)
Out[69]: GF([^222,  ^10, ^209, ^123], order=3^5)
```

3.5.5 Class properties

Certain class properties, such as *elements*, *units*, *squares*, and *primitive_elements*, provide an array of elements with the specified properties.

Integer

```
In [70]: GF = galois.GF(3**2)

In [71]: GF.elements
Out[71]: GF([0, 1, 2, 3, 4, 5, 6, 7, 8], order=3^2)

In [72]: GF.units
Out[72]: GF([1, 2, 3, 4, 5, 6, 7, 8], order=3^2)

In [73]: GF.squares
Out[73]: GF([0, 1, 2, 4, 8], order=3^2)

In [74]: GF.primitive_elements
Out[74]: GF([3, 5, 6, 7], order=3^2)
```

Polynomial

```
In [75]: GF = galois.GF(3**2, repr="poly")

In [76]: GF.elements
Out[76]:
GF([     0,      1,      2,      ,    + 1,    + 2,     2, 2 + 1,
    2 + 2], order=3^2)
```

(continues on next page)

(continued from previous page)

```
In [77]: GF.units
Out[77]:
GF([      1,       2,       ,    + 1,    + 2,      2, 2 + 1, 2 + 2],
   order=3^2)

In [78]: GF.squares
Out[78]: GF([      0,       1,       2,    + 1, 2 + 2], order=3^2)

In [79]: GF.primitive_elements
Out[79]: GF([      ,    + 2,      2, 2 + 1], order=3^2)
```

Power

```
In [80]: GF = galois.GF(3**2, repr="power")

In [81]: GF.elements
Out[81]: GF([  0,    1, ^4,    , ^2, ^7, ^5, ^3, ^6], order=3^2)

In [82]: GF.units
Out[82]: GF([  1, ^4,    , ^2, ^7, ^5, ^3, ^6], order=3^2)

In [83]: GF.squares
Out[83]: GF([  0,    1, ^4, ^2, ^6], order=3^2)

In [84]: GF.primitive_elements
Out[84]: GF([  , ^7, ^5, ^3], order=3^2)
```

3.5.6 Data types

`FieldArray` instances support a fixed set of NumPy data types (`numpy.dtype`). The data type must be able to store all the field elements (in their *integer representation*).

Valid data types

For small finite fields, like $\mathrm{GF}(2^4)$, every NumPy integer data type is supported.

```
In [85]: GF = galois.GF(2**4)

In [86]: GF.dtypes
Out[86]:
[numpy.uint8,
 numpy.uint16,
 numpy.uint32,
 numpy.int8,
 numpy.int16,
 numpy.int32,
 numpy.int64]
```

For medium finite fields, like $\mathrm{GF}(2^{10})$, some NumPy integer data types are not supported. Here, `numpy.uint8` and `numpy.int8` are not supported.

```
In [87]: GF = galois.GF(2**10)

In [88]: GF.dtypes
Out[88]: [numpy.uint16, numpy.uint32, numpy.int16, numpy.int32, numpy.int64]
```

For large finite fields, like $\mathrm{GF}(2^{100})$, only the "object" data type (`numpy.object_`) is supported. This uses arrays of Python objects, rather than integer data types. The Python objects used are Python integers, which have unlimited size.

```
In [89]: GF = galois.GF(2**100)

In [90]: GF.dtypes
Out[90]: [numpy.object_]
```

Default data type

When arrays are created, unless otherwise specified, they use the default data type. The default data type is the smallest unsigned data type (the first in the *dtypes* list).

```
In [91]: GF = galois.GF(2**10)

In [92]: GF.dtypes
Out[92]: [numpy.uint16, numpy.uint32, numpy.int16, numpy.int32, numpy.int64]

In [93]: x = GF.Random(4); x
Out[93]: GF([ 22, 838, 153, 814], order=2^10)

In [94]: x.dtype
Out[94]: dtype('uint16')
```

```
In [95]: GF = galois.GF(2**100)

In [96]: GF.dtypes
Out[96]: [numpy.object_]

In [97]: x = GF.Random(4); x
Out[97]:
GF([ 232789809848978601640925623182, 1070044931957156231825552086524,
    1204669815427895811988291871992, 1038460390379605993393850736300],
   order=2^100)

In [98]: x.dtype
Out[98]: dtype('O')
```

Changing data types

The data type may be explicitly set during array creation by setting the `dtype` keyword argument of the *FieldArray* constructor.

```
In [99]: GF = galois.GF(2**10)

In [100]: x = GF([273, 388, 124, 400], dtype=np.uint32); x
Out[100]: GF([273, 388, 124, 400], order=2^10)

In [101]: x.dtype
Out[101]: dtype('uint32')
```

Arrays may also have their data types changed using `.astype()`. The data type must be valid, however.

```
In [102]: x.dtype
Out[102]: dtype('uint32')

In [103]: x = x.astype(np.int64)

In [104]: x.dtype
Out[104]: dtype('int64')
```

3.5.7 NumPy functions

Most native NumPy functions work on *FieldArray* instances as expected. For example, reshaping a `(10,)`-shape array into a `(2, 5)`-shape array works as desired and returns a *FieldArray* instance.

```
In [105]: GF = galois.GF(7)

In [106]: x = GF.Random(10, seed=1); x
Out[106]: GF([6, 6, 0, 3, 6, 5, 0, 3, 2, 4], order=7)

In [107]: np.reshape(x, (2, 5))
Out[107]:
GF([[6, 6, 0, 3, 6],
    [5, 0, 3, 2, 4]], order=7)
```

However, some functions have a `subok` keyword argument. This indicates whether to return a `numpy.ndarray` subclass from the function. Most notably, `numpy.copy()` defaults `subok` to `False`.

```
In [108]: x
Out[108]: GF([6, 6, 0, 3, 6, 5, 0, 3, 2, 4], order=7)

# Returns np.ndarray!
In [109]: np.copy(x)
Out[109]: array([6, 6, 0, 3, 6, 5, 0, 3, 2, 4], dtype=uint8)

In [110]: np.copy(x, subok=True)
Out[110]: GF([6, 6, 0, 3, 6, 5, 0, 3, 2, 4], order=7)
```

The `numpy.ndarray.copy()` method will, however, return a subclass. Be mindful of the `subok` keyword argument!

```
In [111]: x.copy()
Out[111]: GF([6, 6, 0, 3, 6, 5, 0, 3, 2, 4], order=7)
```

3.6 Array Arithmetic

After creating a *FieldArray* subclass and one or two arrays, nearly any arithmetic operation can be performed using normal NumPy ufuncs or Python operators.

In the sections below, the finite field $\mathrm{GF}(3^5)$ and arrays x and y are used.

```
In [1]: GF = galois.GF(3**5)

In [2]: x = GF([184, 25, 157, 31]); x
Out[2]: GF([184,  25, 157,  31], order=3^5)

In [3]: y = GF([179, 9, 139, 27]); y
Out[3]: GF([179,   9, 139,  27], order=3^5)
```

3.6.1 Standard arithmetic

NumPy ufuncs are universal functions that operate on scalars. Unary ufuncs operate on a single scalar and binary ufuncs operate on two scalars. NumPy extends the scalar operation of ufuncs to operate on arrays in various ways. This extensibility enables NumPy broadcasting.

Expand any section for more details.

Addition: `x + y == np.add(x, y)`

Integer

```
In [4]: x
Out[4]: GF([184,  25, 157,  31], order=3^5)

In [5]: y
Out[5]: GF([179,   9, 139,  27], order=3^5)

In [6]: x + y
Out[6]: GF([ 81,   7, 215,  58], order=3^5)

In [7]: np.add(x, y)
Out[7]: GF([ 81,   7, 215,  58], order=3^5)
```

Polynomial

```
In [8]: x
Out[8]:
GF([      2^4 + 2^2 +   + 1,             2^2 + 2 + 1,
    ^4 + 2^3 + 2^2 +   + 1,               ^3 +   + 1], order=3^5)

In [9]: y
Out[9]:
GF([2^4 + ^2 + 2 + 2,                   ^2,   ^4 + 2^3 +   + 1,
                    ^3], order=3^5)

In [10]: x + y
Out[10]:
GF([                       ^4,                       2 + 1,
    2^4 + ^3 + 2^2 + 2 + 2,              2^3 +   + 1], order=3^5)

In [11]: np.add(x, y)
Out[11]:
GF([                       ^4,                       2 + 1,
    2^4 + ^3 + 2^2 + 2 + 2,              2^3 +   + 1], order=3^5)
```

Power

```
In [12]: x
Out[12]: GF([^156,  ^88, ^176, ^214], order=3^5)

In [13]: y
Out[13]: GF([^60,  ^2, ^59,  ^3], order=3^5)

In [14]: x + y
Out[14]: GF([  ^4, ^126, ^175,  ^28], order=3^5)

In [15]: np.add(x, y)
Out[15]: GF([  ^4, ^126, ^175,  ^28], order=3^5)
```

Additive inverse: `-x == np.negative(x)`

Integer

```
In [16]: x
Out[16]: GF([184,  25, 157,  31], order=3^5)

In [17]: -x
Out[17]: GF([ 98,  14, 206,  62], order=3^5)

In [18]: np.negative(x)
Out[18]: GF([ 98,  14, 206,  62], order=3^5)
```

Polynomial

```
In [19]: x
Out[19]:
GF([      2^4 + 2^2 +   + 1,             2^2 + 2 + 1,
    ^4 + 2^3 + 2^2 +   + 1,              ^3 +   + 1], order=3^5)

In [20]: -x
Out[20]:
GF([       ^4 + ^2 + 2 + 2,               ^2 +   + 2,
    2^4 + ^3 + ^2 + 2 + 2,             2^3 + 2 + 2], order=3^5)

In [21]: np.negative(x)
Out[21]:
GF([       ^4 + ^2 + 2 + 2,               ^2 +   + 2,
    2^4 + ^3 + ^2 + 2 + 2,             2^3 + 2 + 2], order=3^5)
```

Power

```
In [22]: x
Out[22]: GF([^156,  ^88, ^176, ^214], order=3^5)

In [23]: -x
Out[23]: GF([ ^35, ^209,  ^55,  ^93], order=3^5)

In [24]: np.negative(x)
Out[24]: GF([ ^35, ^209,  ^55,  ^93], order=3^5)
```

Any array added to its additive inverse results in zero.

Integer

```
In [25]: x
Out[25]: GF([184,  25, 157,  31], order=3^5)

In [26]: x + np.negative(x)
Out[26]: GF([0, 0, 0, 0], order=3^5)
```

Polynomial

```
In [27]: x
Out[27]:
GF([      2^4 + 2^2 +   + 1,             2^2 + 2 + 1,
    ^4 + 2^3 + 2^2 +   + 1,              ^3 +   + 1], order=3^5)

In [28]: x + np.negative(x)
Out[28]: GF([0, 0, 0, 0], order=3^5)
```

Power

```
In [29]: x
Out[29]: GF([^156,  ^88, ^176, ^214], order=3^5)

In [30]: x + np.negative(x)
Out[30]: GF([0, 0, 0, 0], order=3^5)
```

Subtraction: x - y == np.subtract(x, y)

Integer

```
In [31]: x
Out[31]: GF([184,  25, 157,  31], order=3^5)

In [32]: y
Out[32]: GF([179,   9, 139,  27], order=3^5)

In [33]: x - y
Out[33]: GF([17, 16, 18,  4], order=3^5)

In [34]: np.subtract(x, y)
Out[34]: GF([17, 16, 18,  4], order=3^5)
```

Polynomial

```
In [35]: x
Out[35]:
GF([      2^4 + 2^2 +  + 1,              2^2 + 2 + 1,
    ^4 + 2^3 + 2^2 +  + 1,                ^3 +  + 1], order=3^5)

In [36]: y
Out[36]:
GF([2^4 + ^2 + 2 + 2,                 ^2,  ^4 + 2^3 +  + 1,
                    ^3], order=3^5)

In [37]: x - y
Out[37]: GF([^2 + 2 + 2, ^2 + 2 + 1,         2^2,         + 1], order=3^5)

In [38]: np.subtract(x, y)
Out[38]: GF([^2 + 2 + 2, ^2 + 2 + 1,         2^2,         + 1], order=3^5)
```

Power

```
In [39]: x
Out[39]: GF([^156,  ^88, ^176, ^214], order=3^5)

In [40]: y
Out[40]: GF([^60,  ^2, ^59,  ^3], order=3^5)

In [41]: x - y
Out[41]: GF([^222, ^138, ^123,  ^69], order=3^5)

In [42]: np.subtract(x, y)
Out[42]: GF([^222, ^138, ^123,  ^69], order=3^5)
```

Multiplication: `x * y == np.multiply(x, y)`

Integer

```
In [43]: x
Out[43]: GF([184,  25, 157,  31], order=3^5)

In [44]: y
Out[44]: GF([179,   9, 139,  27], order=3^5)

In [45]: x * y
Out[45]: GF([ 41, 225, 106, 123], order=3^5)

In [46]: np.multiply(x, y)
Out[46]: GF([ 41, 225, 106, 123], order=3^5)
```

Polynomial

```
In [47]: x
Out[47]:
GF([      2^4 + 2^2 +   + 1,             2^2 + 2 + 1,
    ^4 + 2^3 + 2^2 +   + 1,               ^3 +   + 1], order=3^5)

In [48]: y
Out[48]:
GF([2^4 + ^2 + 2 + 2,                 ^2,  ^4 + 2^3 +   + 1,
                    ^3], order=3^5)

In [49]: x * y
Out[49]:
GF([   ^3 + ^2 +   + 2,     2^4 + 2^3 + ^2,  ^4 + 2^2 + 2 + 1,
    ^4 + ^3 + ^2 + 2], order=3^5)

In [50]: np.multiply(x, y)
```

(continues on next page)

(continued from previous page)

```
Out[50]:
GF([   ^3 + ^2 +   + 2,     2^4 + 2^3 + ^2,   ^4 + 2^2 + 2 + 1,
    ^4 + ^3 + ^2 + 2], order=3^5)
```

Power

```
In [51]: x
Out[51]: GF([^156,  ^88, ^176, ^214], order=3^5)

In [52]: y
Out[52]: GF([^60,  ^2, ^59,  ^3], order=3^5)

In [53]: x * y
Out[53]: GF([^216,  ^90, ^235, ^217], order=3^5)

In [54]: np.multiply(x, y)
Out[54]: GF([^216,  ^90, ^235, ^217], order=3^5)
```

Scalar multiplication: `x * 4 == np.multiply(x, 4)`

Scalar multiplication is essentially *repeated addition*. It is the "multiplication" of finite field elements and integers. The integer value indicates how many additions of the field element to sum.

Integer

```
In [55]: x
Out[55]: GF([184,  25, 157,  31], order=3^5)

In [56]: x * 4
Out[56]: GF([184,  25, 157,  31], order=3^5)

In [57]: np.multiply(x, 4)
Out[57]: GF([184,  25, 157,  31], order=3^5)

In [58]: x + x + x + x
Out[58]: GF([184,  25, 157,  31], order=3^5)
```

Polynomial

```
In [59]: x
Out[59]:
GF([      2^4 + 2^2 +   + 1,              2^2 + 2 + 1,
    ^4 + 2^3 + 2^2 +   + 1,                ^3 +   + 1], order=3^5)

In [60]: x * 4
Out[60]:
```

(continues on next page)

(continued from previous page)

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

In [61]: np.multiply(x, 4)
Out[61]:
GF([      2^4 + 2^2 +  + 1,             2^2 + 2 + 1,
    ^4 + 2^3 + 2^2 +  + 1,               ^3 +  + 1], order=3^5)

In [62]: x + x + x + x
Out[62]:
GF([      2^4 + 2^2 +  + 1,             2^2 + 2 + 1,
    ^4 + 2^3 + 2^2 +  + 1,               ^3 +  + 1], order=3^5)
```

Power

```
In [63]: x
Out[63]: GF([^156,  ^88, ^176, ^214], order=3^5)

In [64]: x * 4
Out[64]: GF([^156,  ^88, ^176, ^214], order=3^5)

In [65]: np.multiply(x, 4)
Out[65]: GF([^156,  ^88, ^176, ^214], order=3^5)

In [66]: x + x + x + x
Out[66]: GF([^156,  ^88, ^176, ^214], order=3^5)
```

In finite fields $\mathrm{GF}(p^m)$, the characteristic p is the smallest value when multiplied by any non-zero field element that results in 0.

Integer

```
In [67]: p = GF.characteristic; p
Out[67]: 3

In [68]: x * p
Out[68]: GF([0, 0, 0, 0], order=3^5)
```

Polynomial

```
In [69]: p = GF.characteristic; p
Out[69]: 3

In [70]: x * p
Out[70]: GF([0, 0, 0, 0], order=3^5)
```

Power

```
In [71]: p = GF.characteristic; p
Out[71]: 3

In [72]: x * p
Out[72]: GF([0, 0, 0, 0], order=3^5)
```

Multiplicative inverse: `y ** -1 == np.reciprocal(y)`

Integer

```
In [73]: y
Out[73]: GF([179,   9, 139,  27], order=3^5)

In [74]: y ** -1
Out[74]: GF([ 71, 217, 213, 235], order=3^5)

In [75]: GF(1) / y
Out[75]: GF([ 71, 217, 213, 235], order=3^5)

In [76]: np.reciprocal(y)
Out[76]: GF([ 71, 217, 213, 235], order=3^5)
```

Polynomial

```
In [77]: y
Out[77]:
GF([2^4 + ^2 + 2 + 2,                 ^2,  ^4 + 2^3 +  + 1,
                    ^3], order=3^5)

In [78]: y ** -1
Out[78]:
GF([   2^3 + ^2 + 2 + 2,        2^4 + 2^3 + 1,
    2^4 + ^3 + 2^2 + 2, 2^4 + 2^3 + 2^2 + 1], order=3^5)

In [79]: GF(1) / y
Out[79]:
GF([   2^3 + ^2 + 2 + 2,        2^4 + 2^3 + 1,
    2^4 + ^3 + 2^2 + 2, 2^4 + 2^3 + 2^2 + 1], order=3^5)

In [80]: np.reciprocal(y)
Out[80]:
GF([   2^3 + ^2 + 2 + 2,        2^4 + 2^3 + 1,
    2^4 + ^3 + 2^2 + 2, 2^4 + 2^3 + 2^2 + 1], order=3^5)
```

Power

```
In [81]: y
Out[81]: GF([^60,  ^2, ^59,  ^3], order=3^5)

In [82]: y ** -1
Out[82]: GF([^182, ^240, ^183, ^239], order=3^5)

In [83]: GF(1) / y
Out[83]: GF([^182, ^240, ^183, ^239], order=3^5)

In [84]: np.reciprocal(y)
Out[84]: GF([^182, ^240, ^183, ^239], order=3^5)
```

Any array multiplied by its multiplicative inverse results in one.

Integer

```
In [85]: y * np.reciprocal(y)
Out[85]: GF([1, 1, 1, 1], order=3^5)
```

Polynomial

```
In [86]: y * np.reciprocal(y)
Out[86]: GF([1, 1, 1, 1], order=3^5)
```

Power

```
In [87]: y * np.reciprocal(y)
Out[87]: GF([1, 1, 1, 1], order=3^5)
```

Division: `x / y == x // y == np.divide(x, y)`

Integer

```
In [88]: x
Out[88]: GF([184,  25, 157,  31], order=3^5)

In [89]: y
Out[89]: GF([179,   9, 139,  27], order=3^5)

In [90]: x / y
Out[90]: GF([237,  56, 122, 126], order=3^5)

In [91]: x // y
Out[91]: GF([237,  56, 122, 126], order=3^5)
```

(continues on next page)

(continued from previous page)

```
In [92]: np.divide(x, y)
Out[92]: GF([237,  56, 122, 126], order=3^5)
```

Polynomial

```
In [93]: x
Out[93]:
GF([      2^4 + 2^2 +  + 1,             2^2 + 2 + 1,
    ^4 + 2^3 + 2^2 +  + 1,               ^3 +  + 1], order=3^5)

In [94]: y
Out[94]:
GF([2^4 + ^2 + 2 + 2,                 ^2,  ^4 + 2^3 +  + 1,
                    ^3], order=3^5)

In [95]: x / y
Out[95]:
GF([ 2^4 + 2^3 + 2^2 + ,                 2^3 + 2,
    ^4 + ^3 + ^2 +  + 2,        ^4 + ^3 + 2^2], order=3^5)

In [96]: x // y
Out[96]:
GF([ 2^4 + 2^3 + 2^2 + ,                 2^3 + 2,
    ^4 + ^3 + ^2 +  + 2,        ^4 + ^3 + 2^2], order=3^5)

In [97]: np.divide(x, y)
Out[97]:
GF([ 2^4 + 2^3 + 2^2 + ,                 2^3 + 2,
    ^4 + ^3 + ^2 +  + 2,        ^4 + ^3 + 2^2], order=3^5)
```

Power

```
In [98]: x
Out[98]: GF([^156,  ^88, ^176, ^214], order=3^5)

In [99]: y
Out[99]: GF([^60,  ^2, ^59,  ^3], order=3^5)

In [100]: x / y
Out[100]: GF([ ^96,  ^86, ^117, ^211], order=3^5)

In [101]: x // y
Out[101]: GF([ ^96,  ^86, ^117, ^211], order=3^5)

In [102]: np.divide(x, y)
Out[102]: GF([ ^96,  ^86, ^117, ^211], order=3^5)
```

Remainder: `x % y == np.remainder(x, y)`

Integer

```
In [103]: x
Out[103]: GF([184,  25, 157,  31], order=3^5)

In [104]: y
Out[104]: GF([179,   9, 139,  27], order=3^5)

In [105]: x % y
Out[105]: GF([0, 0, 0, 0], order=3^5)

In [106]: np.remainder(x, y)
Out[106]: GF([0, 0, 0, 0], order=3^5)
```

Polynomial

```
In [107]: x
Out[107]:
GF([      2^4 + 2^2 +  + 1,             2^2 + 2 + 1,
    ^4 + 2^3 + 2^2 +  + 1,               ^3 +  + 1], order=3^5)

In [108]: y
Out[108]:
GF([2^4 + ^2 + 2 + 2,                 ^2,  ^4 + 2^3 +  + 1,
                    ^3], order=3^5)

In [109]: x % y
Out[109]: GF([0, 0, 0, 0], order=3^5)

In [110]: np.remainder(x, y)
Out[110]: GF([0, 0, 0, 0], order=3^5)
```

Power

```
In [111]: x
Out[111]: GF([^156,  ^88, ^176, ^214], order=3^5)

In [112]: y
Out[112]: GF([^60,  ^2, ^59,  ^3], order=3^5)

In [113]: x % y
Out[113]: GF([0, 0, 0, 0], order=3^5)

In [114]: np.remainder(x, y)
Out[114]: GF([0, 0, 0, 0], order=3^5)
```

Divmod: `divmod(x, y) == np.divmod(x, y)`

Integer

```
In [115]: x
Out[115]: GF([184,  25, 157,  31], order=3^5)

In [116]: y
Out[116]: GF([179,   9, 139,  27], order=3^5)

In [117]: x // y, x % y
Out[117]: (GF([237,  56, 122, 126], order=3^5), GF([0, 0, 0, 0], order=3^5))

In [118]: divmod(x, y)
Out[118]: (GF([237,  56, 122, 126], order=3^5), GF([0, 0, 0, 0], order=3^5))

In [119]: np.divmod(x, y)
Out[119]: (GF([237,  56, 122, 126], order=3^5), GF([0, 0, 0, 0], order=3^5))
```

Polynomial

```
In [120]: x
Out[120]:
GF([      2^4 + 2^2 +  + 1,              2^2 + 2 + 1,
    ^4 + 2^3 + 2^2 +  + 1,                ^3 +  + 1], order=3^5)

In [121]: y
Out[121]:
GF([2^4 + ^2 + 2 + 2,                  ^2,  ^4 + 2^3 +  + 1,
                    ^3], order=3^5)

In [122]: x // y, x % y
Out[122]:
(GF([ 2^4 + 2^3 + 2^2 + ,                   2^3 + 2,
     ^4 + ^3 + ^2 +  + 2,        ^4 + ^3 + 2^2], order=3^5),
 GF([0, 0, 0, 0], order=3^5))

In [123]: divmod(x, y)
Out[123]:
(GF([ 2^4 + 2^3 + 2^2 + ,                   2^3 + 2,
     ^4 + ^3 + ^2 +  + 2,        ^4 + ^3 + 2^2], order=3^5),
 GF([0, 0, 0, 0], order=3^5))

In [124]: np.divmod(x, y)
Out[124]:
(GF([ 2^4 + 2^3 + 2^2 + ,                   2^3 + 2,
     ^4 + ^3 + ^2 +  + 2,        ^4 + ^3 + 2^2], order=3^5),
 GF([0, 0, 0, 0], order=3^5))
```

Power

```
In [125]: x
Out[125]: GF([^156,  ^88, ^176, ^214], order=3^5)

In [126]: y
Out[126]: GF([^60,  ^2, ^59,  ^3], order=3^5)

In [127]: x // y, x % y
Out[127]: (GF([ ^96,  ^86, ^117, ^211], order=3^5), GF([0, 0, 0, 0], order=3^5))

In [128]: divmod(x, y)
Out[128]: (GF([ ^96,  ^86, ^117, ^211], order=3^5), GF([0, 0, 0, 0], order=3^5))

In [129]: np.divmod(x, y)
Out[129]: (GF([ ^96,  ^86, ^117, ^211], order=3^5), GF([0, 0, 0, 0], order=3^5))
```

Integer

```
In [130]: q, r = divmod(x, y)

In [131]: q*y + r == x
Out[131]: array([ True,  True,  True,  True])
```

Polynomial

```
In [132]: q, r = divmod(x, y)

In [133]: q*y + r == x
Out[133]: array([ True,  True,  True,  True])
```

Power

```
In [134]: q, r = divmod(x, y)

In [135]: q*y + r == x
Out[135]: array([ True,  True,  True,  True])
```

Exponentiation: `x ** 3 == np.power(x, 3)`

Integer

```
In [136]: x
Out[136]: GF([184,  25, 157,  31], order=3^5)

In [137]: x ** 3
Out[137]: GF([175,  76, 218, 192], order=3^5)

In [138]: np.power(x, 3)
Out[138]: GF([175,  76, 218, 192], order=3^5)

In [139]: x * x * x
Out[139]: GF([175,  76, 218, 192], order=3^5)
```

Polynomial

```
In [140]: x
Out[140]:
GF([      2^4 + 2^2 +  + 1,               2^2 + 2 + 1,
    ^4 + 2^3 + 2^2 +  + 1,                ^3 +  + 1], order=3^5)

In [141]: x ** 3
Out[141]:
GF([ 2^4 + ^2 +  + 1, 2^3 + 2^2 +  + 1,     2^4 + 2^3 + 2,
         2^4 + ^3 + ], order=3^5)

In [142]: np.power(x, 3)
Out[142]:
GF([ 2^4 + ^2 +  + 1, 2^3 + 2^2 +  + 1,     2^4 + 2^3 + 2,
         2^4 + ^3 + ], order=3^5)

In [143]: x * x * x
Out[143]:
GF([ 2^4 + ^2 +  + 1, 2^3 + 2^2 +  + 1,     2^4 + 2^3 + 2,
         2^4 + ^3 + ], order=3^5)
```

Power

```
In [144]: x
Out[144]: GF([^156,  ^88, ^176, ^214], order=3^5)

In [145]: x ** 3
Out[145]: GF([^226,  ^22,  ^44, ^158], order=3^5)

In [146]: np.power(x, 3)
Out[146]: GF([^226,  ^22,  ^44, ^158], order=3^5)

In [147]: x * x * x
Out[147]: GF([^226,  ^22,  ^44, ^158], order=3^5)
```

Square root: `np.sqrt(x)`

Integer

```
In [148]: x
Out[148]: GF([184,  25, 157,  31], order=3^5)

In [149]: x.is_square()
Out[149]: array([ True,  True,  True,  True])

In [150]: z = np.sqrt(x); z
Out[150]: GF([102, 109,  14, 111], order=3^5)

In [151]: z ** 2 == x
Out[151]: array([ True,  True,  True,  True])
```

Polynomial

```
In [152]: x
Out[152]:
GF([      2^4 + 2^2 +  + 1,             2^2 + 2 + 1,
    ^4 + 2^3 + 2^2 +  + 1,               ^3 +  + 1], order=3^5)

In [153]: x.is_square()
Out[153]: array([ True,  True,  True,  True])

In [154]: z = np.sqrt(x); z
Out[154]:
GF([^4 + 2^2 + ,  ^4 + ^3 + 1,    ^2 +  + 2,  ^4 + ^3 + ],
   order=3^5)

In [155]: z ** 2 == x
Out[155]: array([ True,  True,  True,  True])
```

Power

```
In [156]: x
Out[156]: GF([^156,  ^88, ^176, ^214], order=3^5)

In [157]: x.is_square()
Out[157]: array([ True,  True,  True,  True])

In [158]: z = np.sqrt(x); z
Out[158]: GF([^199, ^165, ^209, ^228], order=3^5)

In [159]: z ** 2 == x
Out[159]: array([ True,  True,  True,  True])
```

See also *is_square()*, *squares()*, and *non_squares()*.

Logarithm: `np.log(x)` or `x.log()`

Compute the logarithm base α, the primitive element of the field.

Integer

```
In [160]: y
Out[160]: GF([179,   9, 139,  27], order=3^5)

In [161]: z = np.log(y); z
Out[161]: array([60,  2, 59,  3])

In [162]: alpha = GF.primitive_element; alpha
Out[162]: GF(3, order=3^5)

In [163]: alpha ** z == y
Out[163]: array([ True,  True,  True,  True])
```

Polynomial

```
In [164]: y
Out[164]:
GF([2^4 + ^2 + 2 + 2,                 ^2,  ^4 + 2^3 +  + 1,
                    ^3], order=3^5)

In [165]: z = np.log(y); z
Out[165]: array([60,  2, 59,  3])

In [166]: alpha = GF.primitive_element; alpha
Out[166]: GF(, order=3^5)

In [167]: alpha ** z == y
Out[167]: array([ True,  True,  True,  True])
```

Power

```
In [168]: y
Out[168]: GF([^60,  ^2, ^59,  ^3], order=3^5)

In [169]: z = np.log(y); z
Out[169]: array([60,  2, 59,  3])

In [170]: alpha = GF.primitive_element; alpha
Out[170]: GF(, order=3^5)

In [171]: alpha ** z == y
Out[171]: array([ True,  True,  True,  True])
```

Compute the logarithm base β, a different primitive element of the field. See `FieldArray.log()` for more details.

Integer

```
In [172]: y
Out[172]: GF([179,   9, 139,  27], order=3^5)

In [173]: beta = GF.primitive_elements[-1]; beta
Out[173]: GF(242, order=3^5)

In [174]: z = y.log(beta); z
Out[174]: array([190, 208, 207, 191])

In [175]: beta ** z == y
Out[175]: array([ True,  True,  True,  True])
```

Polynomial

```
In [176]: y
Out[176]:
GF([2^4 + ^2 + 2 + 2,                 ^2,  ^4 + 2^3 +  + 1,
                    ^3], order=3^5)

In [177]: beta = GF.primitive_elements[-1]; beta
Out[177]: GF(2^4 + 2^3 + 2^2 + 2 + 2, order=3^5)

In [178]: z = y.log(beta); z
Out[178]: array([190, 208, 207, 191])

In [179]: beta ** z == y
Out[179]: array([ True,  True,  True,  True])
```

Power

```
In [180]: y
Out[180]: GF([^60,  ^2, ^59,  ^3], order=3^5)

In [181]: beta = GF.primitive_elements[-1]; beta
Out[181]: GF(^185, order=3^5)

In [182]: z = y.log(beta); z
Out[182]: array([190, 208, 207, 191])

In [183]: beta ** z == y
Out[183]: array([ True,  True,  True,  True])
```

3.6.2 Ufunc methods

FieldArray instances support NumPy ufunc methods. Ufunc methods allow a user to apply a NumPy ufunc in a unique way across the target array. All arithmetic ufuncs are supported.

Expand any section for more details.

---

reduce()

The `reduce` methods reduce the input array's dimension by one, applying the ufunc across one axis.

Integer

```
In [184]: x
Out[184]: GF([184,  25, 157,  31], order=3^5)

In [185]: np.add.reduce(x)
Out[185]: GF(7, order=3^5)

In [186]: x[0] + x[1] + x[2] + x[3]
Out[186]: GF(7, order=3^5)
```

Polynomial

```
In [187]: x
Out[187]:
GF([      2^4 + 2^2 +   + 1,                2^2 + 2 + 1,
    ^4 + 2^3 + 2^2 +   + 1,                 ^3 +   + 1], order=3^5)

In [188]: np.add.reduce(x)
Out[188]: GF(2 + 1, order=3^5)

In [189]: x[0] + x[1] + x[2] + x[3]
Out[189]: GF(2 + 1, order=3^5)
```

Power

```
In [190]: x
Out[190]: GF([^156,  ^88, ^176, ^214], order=3^5)

In [191]: np.add.reduce(x)
Out[191]: GF(^126, order=3^5)

In [192]: x[0] + x[1] + x[2] + x[3]
Out[192]: GF(^126, order=3^5)
```

Integer

```
In [193]: np.multiply.reduce(x)
Out[193]: GF(105, order=3^5)

In [194]: x[0] * x[1] * x[2] * x[3]
Out[194]: GF(105, order=3^5)
```

Polynomial

```
In [195]: np.multiply.reduce(x)
Out[195]: GF(^4 + 2^2 + 2, order=3^5)

In [196]: x[0] * x[1] * x[2] * x[3]
Out[196]: GF(^4 + 2^2 + 2, order=3^5)
```

Power

```
In [197]: np.multiply.reduce(x)
Out[197]: GF(^150, order=3^5)

In [198]: x[0] * x[1] * x[2] * x[3]
Out[198]: GF(^150, order=3^5)
```

`accumulate()`

The `accumulate` methods accumulate the result of the ufunc across a specified axis.

Integer

```
In [199]: x
Out[199]: GF([184,  25, 157,  31], order=3^5)

In [200]: np.add.accumulate(x)
Out[200]: GF([184, 173,  57,   7], order=3^5)

In [201]: GF([x[0], x[0] + x[1], x[0] + x[1] + x[2], x[0] + x[1] + x[2] + x[3]])
Out[201]: GF([184, 173,  57,   7], order=3^5)
```

Polynomial

```
In [202]: x
Out[202]:
GF([      2^4 + 2^2 +  + 1,              2^2 + 2 + 1,
    ^4 + 2^3 + 2^2 +  + 1,               ^3 +  + 1], order=3^5)

In [203]: np.add.accumulate(x)
Out[203]:
GF([2^4 + 2^2 +  + 1,      2^4 + ^2 + 2,            2^3 + ,
                2 + 1], order=3^5)

In [204]: GF([x[0], x[0] + x[1], x[0] + x[1] + x[2], x[0] + x[1] + x[2] + x[3]])
Out[204]:
GF([2^4 + 2^2 +  + 1,      2^4 + ^2 + 2,            2^3 + ,
                2 + 1], order=3^5)
```

Power

```
In [205]: x
Out[205]: GF([^156,  ^88, ^176, ^214], order=3^5)

In [206]: np.add.accumulate(x)
Out[206]: GF([^156, ^213, ^196, ^126], order=3^5)

In [207]: GF([x[0], x[0] + x[1], x[0] + x[1] + x[2], x[0] + x[1] + x[2] + x[3]])
Out[207]: GF([^156, ^213, ^196, ^126], order=3^5)
```

Integer

```
In [208]: np.multiply.accumulate(x)
Out[208]: GF([184,   9, 211, 105], order=3^5)

In [209]: GF([x[0], x[0] * x[1], x[0] * x[1] * x[2], x[0] * x[1] * x[2] * x[3]])
Out[209]: GF([184,   9, 211, 105], order=3^5)
```

Polynomial

```
In [210]: np.multiply.accumulate(x)
Out[210]:
GF([      2^4 + 2^2 +  + 1,                       ^2,
    2^4 + ^3 + 2^2 +  + 1,           ^4 + 2^2 + 2], order=3^5)

In [211]: GF([x[0], x[0] * x[1], x[0] * x[1] * x[2], x[0] * x[1] * x[2] * x[3]])
Out[211]:
GF([      2^4 + 2^2 +  + 1,                       ^2,
    2^4 + ^3 + 2^2 +  + 1,           ^4 + 2^2 + 2], order=3^5)
```

Power

```
In [212]: np.multiply.accumulate(x)
Out[212]: GF([^156,   ^2, ^178, ^150], order=3^5)

In [213]: GF([x[0], x[0] * x[1], x[0] * x[1] * x[2], x[0] * x[1] * x[2] * x[3]])
Out[213]: GF([^156,   ^2, ^178, ^150], order=3^5)
```

---

reduceat()

The `reduceat` methods reduces the input array's dimension by one, applying the ufunc across one axis in-between certain indices.

Integer

```
In [214]: x
Out[214]: GF([184,  25, 157,  31], order=3^5)

In [215]: np.add.reduceat(x, [0, 3])
Out[215]: GF([57, 31], order=3^5)

In [216]: GF([x[0] + x[1] + x[2], x[3]])
Out[216]: GF([57, 31], order=3^5)
```

Polynomial

```
In [217]: x
Out[217]:
GF([      2^4 + 2^2 +  + 1,             2^2 + 2 + 1,
    ^4 + 2^3 + 2^2 +  + 1,               ^3 +  + 1], order=3^5)

In [218]: np.add.reduceat(x, [0, 3])
Out[218]: GF([   2^3 + , ^3 +  + 1], order=3^5)

In [219]: GF([x[0] + x[1] + x[2], x[3]])
Out[219]: GF([   2^3 + , ^3 +  + 1], order=3^5)
```

Power

```
In [220]: x
Out[220]: GF([^156,  ^88, ^176, ^214], order=3^5)

In [221]: np.add.reduceat(x, [0, 3])
Out[221]: GF([^196, ^214], order=3^5)

In [222]: GF([x[0] + x[1] + x[2], x[3]])
Out[222]: GF([^196, ^214], order=3^5)
```

Integer

```
In [223]: np.multiply.reduceat(x, [0, 3])
Out[223]: GF([211,  31], order=3^5)

In [224]: GF([x[0] * x[1] * x[2], x[3]])
Out[224]: GF([211,  31], order=3^5)
```

Polynomial

```
In [225]: np.multiply.reduceat(x, [0, 3])
Out[225]: GF([2^4 + ^3 + 2^2 +  + 1,                ^3 +  + 1], order=3^5)

In [226]: GF([x[0] * x[1] * x[2], x[3]])
Out[226]: GF([2^4 + ^3 + 2^2 +  + 1,                ^3 +  + 1], order=3^5)
```

Power

```
In [227]: np.multiply.reduceat(x, [0, 3])
Out[227]: GF([^178, ^214], order=3^5)

In [228]: GF([x[0] * x[1] * x[2], x[3]])
Out[228]: GF([^178, ^214], order=3^5)
```

---

outer()

The `outer` methods applies the ufunc to each pair of inputs.

Integer

```
In [229]: x
Out[229]: GF([184,  25, 157,  31], order=3^5)

In [230]: y
Out[230]: GF([179,   9, 139,  27], order=3^5)

In [231]: np.add.outer(x, y)
Out[231]:
GF([[ 81, 166,  80, 211],
    [165,   7, 155,  52],
    [ 54, 139, 215, 103],
    [198,  40,  89,  58]], order=3^5)
```

Polynomial

```
In [232]: x
Out[232]:
GF([      2^4 + 2^2 +   + 1,              2^2 + 2 + 1,
    ^4 + 2^3 + 2^2 +   + 1,               ^3 +   + 1], order=3^5)

In [233]: y
Out[233]:
GF([2^4 + ^2 + 2 + 2,                  ^2,  ^4 + 2^3 +   + 1,
                    ^3], order=3^5)

In [234]: np.add.outer(x, y)
Out[234]:
GF([[                       ^4,                  2^4 +   + 1,
           2^3 + 2^2 + 2 + 2,  2^4 + ^3 + 2^2 +   + 1],
    [                  2^4 +  ,                       2 + 1,
          ^4 + 2^3 + 2^2 + 2,        ^3 + 2^2 + 2 + 1],
    [                      2^3,         ^4 + 2^3 +   + 1,
     2^4 + ^3 + 2^2 + 2 + 2,         ^4 + 2^2 +   + 1],
    [          2^4 + ^3 + ^2,          ^3 + ^2 +   + 1,
                  ^4 + 2 + 2,                2^3 +   + 1]], order=3^5)
```

Power

```
In [235]: x
Out[235]: GF([^156,  ^88, ^176, ^214], order=3^5)

In [236]: y
Out[236]: GF([^60,  ^2, ^59,  ^3], order=3^5)

In [237]: np.add.outer(x, y)
Out[237]:
GF([[  ^4,  ^45, ^236, ^178],
    [^137, ^126,  ^43,  ^79],
    [^124,  ^59, ^175, ^181],
    [^103, ^115, ^166,  ^28]], order=3^5)
```

Integer

```
In [238]: np.multiply.outer(x, y)
Out[238]:
GF([[ 41, 192,  93,  97],
    [ 91, 225, 193, 196],
    [ 80, 211, 106, 145],
    [149,  41, 129, 123]], order=3^5)
```

Polynomial

```
In [239]: np.multiply.outer(x, y)
Out[239]:
GF([[        ^3 + ^2 +  + 2,            2^4 + ^3 + ,
                ^4 + ^2 + ,         ^4 + ^2 + 2 + 1],
    [           ^4 + ^2 + 1,          2^4 + 2^3 + ^2,
           2^4 + ^3 +  + 1,        2^4 + ^3 + 2 + 1],
    [     2^3 + 2^2 + 2 + 2, 2^4 + ^3 + 2^2 +  + 1,
           ^4 + 2^2 + 2 + 1,       ^4 + 2^3 + ^2 + 1],
    [ ^4 + 2^3 + ^2 +  + 2,          ^3 + ^2 +  + 2,
          ^4 + ^3 + 2^2 + ,       ^4 + ^3 + ^2 + 2]], order=3^5)
```

Power

```
In [240]: np.multiply.outer(x, y)
Out[240]:
GF([[^216, ^158, ^215, ^159],
    [^148,  ^90, ^147,  ^91],
    [^236, ^178, ^235, ^179],
    [ ^32, ^216,  ^31, ^217]], order=3^5)
```

---

at()

The `at` methods performs the ufunc in-place at the specified indices.

Integer

```
In [241]: x
Out[241]: GF([184,  25, 157,  31], order=3^5)

In [242]: z = x.copy()

# Negate indices 0 and 1 in-place
In [243]: np.negative.at(x, [0, 1]); x
Out[243]: GF([ 98,  14, 157,  31], order=3^5)

In [244]: z[0:1] *= -1; z
Out[244]: GF([ 98,  25, 157,  31], order=3^5)
```

Polynomial

```
In [245]: x
Out[245]:
GF([       ^4 + ^2 + 2 + 2,                 ^2 +   + 2,
    ^4 + 2^3 + 2^2 +   + 1,                ^3 +   + 1], order=3^5)

In [246]: z = x.copy()

# Negate indices 0 and 1 in-place
In [247]: np.negative.at(x, [0, 1]); x
Out[247]:
GF([      2^4 + 2^2 +   + 1,              2^2 + 2 + 1,
    ^4 + 2^3 + 2^2 +   + 1,                ^3 +   + 1], order=3^5)

In [248]: z[0:1] *= -1; z
Out[248]:
GF([      2^4 + 2^2 +   + 1,                ^2 +   + 2,
    ^4 + 2^3 + 2^2 +   + 1,                ^3 +   + 1], order=3^5)
```

Power

```
In [249]: x
Out[249]: GF([^156,  ^88, ^176, ^214], order=3^5)

In [250]: z = x.copy()

# Negate indices 0 and 1 in-place
In [251]: np.negative.at(x, [0, 1]); x
Out[251]: GF([ ^35, ^209, ^176, ^214], order=3^5)

In [252]: z[0:1] *= -1; z
Out[252]: GF([ ^35,  ^88, ^176, ^214], order=3^5)
```

3.6.3 Advanced arithmetic

Convolution: `np.convolve(x, y)`

Integer

```
In [253]: x
Out[253]: GF([ 98,  14, 157,  31], order=3^5)

In [254]: y
Out[254]: GF([179,   9, 139,  27], order=3^5)
```

(continues on next page)

(continued from previous page)

```
In [255]: np.convolve(x, y)
Out[255]: GF([ 79,  80,   5,  79, 167, 166, 123], order=3^5)
```

Polynomial

```
In [256]: x
Out[256]:
GF([      ^4 + ^2 + 2 + 2,              ^2 +  + 2,
    ^4 + 2^3 + 2^2 +  + 1,               ^3 +  + 1], order=3^5)

In [257]: y
Out[257]:
GF([2^4 + ^2 + 2 + 2,                   ^2,  ^4 + 2^3 +  + 1,
                    ^3], order=3^5)

In [258]: np.convolve(x, y)
Out[258]:
GF([2^3 + 2^2 + 2 + 1, 2^3 + 2^2 + 2 + 2,                  + 2,
    2^3 + 2^2 + 2 + 1,         2^4 +  + 2,         2^4 +  + 1,
    ^4 + ^3 + ^2 + 2], order=3^5)
```

Power

```
In [259]: x
Out[259]: GF([ ^35, ^209, ^176, ^214], order=3^5)

In [260]: y
Out[260]: GF([^60,  ^2, ^59,  ^3], order=3^5)

In [261]: np.convolve(x, y)
Out[261]: GF([ ^95, ^236,   ^5,  ^95,   ^9,  ^45, ^217], order=3^5)
```

FFT: `np.fft.fft(x)`

The Discrete Fourier Transform (DFT) of size n over the finite field $\mathrm{GF}(p^m)$ exists when there exists a primitive n-th root of unity. This occurs when $n\ |p^m - 1$.

Integer

```
In [262]: GF = galois.GF(7**5)

In [263]: n = 6

# n divides p^m - 1
In [264]: (GF.order - 1) % n
Out[264]: 0
```

(continues on next page)

(continued from previous page)

```
In [265]: x = GF.Random(n, seed=1); x
Out[265]: GF([ 7952, 12470,  8601, 11055, 12691,  9895], order=7^5)

In [266]: X = np.fft.fft(x); X
Out[266]: GF([ 9387, 10789, 14695, 13079, 14025,  5694], order=7^5)

In [267]: np.fft.ifft(X)
Out[267]: GF([ 7952, 12470,  8601, 11055, 12691,  9895], order=7^5)
```

Polynomial

```
In [268]: GF = galois.GF(7**5, repr="poly")

In [269]: n = 6

# n divides p^m - 1
In [270]: (GF.order - 1) % n
Out[270]: 0

In [271]: x = GF.Random(n, seed=1); x
Out[271]:
GF([    3^4 + 2^3 + ^2 + 2, 5^4 + ^3 + 2^2 + 3 + 3,
          3^4 + 4^3 + 3 + 5, 4^4 + 4^3 + ^2 + 4 + 2,
                   5^4 + 2^3,       4^4 + 5^2 + 6 + 4], order=7^5)

In [272]: X = np.fft.fft(x); X
Out[272]:
GF([   3^4 + 6^3 + 2^2 + 4, 4^4 + 3^3 + 3^2 +  + 2,
          6^4 + 5^2 + 6 + 2,       5^4 + 3^3 + 6 + 3,
    5^4 + 5^3 + 6^2 +  + 4, 2^4 + 2^3 + 4^2 +  + 3], order=7^5)

In [273]: np.fft.ifft(X)
Out[273]:
GF([    3^4 + 2^3 + ^2 + 2, 5^4 + ^3 + 2^2 + 3 + 3,
          3^4 + 4^3 + 3 + 5, 4^4 + 4^3 + ^2 + 4 + 2,
                   5^4 + 2^3,       4^4 + 5^2 + 6 + 4], order=7^5)
```

Power

```
In [274]: GF = galois.GF(7**5, repr="power")

In [275]: n = 6

# n divides p^m - 1
In [276]: (GF.order - 1) % n
Out[276]: 0

In [277]: x = GF.Random(n, seed=1); x
```

(continues on next page)

(continued from previous page)

```
Out[277]: GF([^11363,  ^2127, ^15189,  ^5863,  ^1240,   ^255], order=7^5)

In [278]: X = np.fft.fft(x); X
Out[278]: GF([ ^7664, ^14905, ^15266, ^13358,  ^9822, ^16312], order=7^5)

In [279]: np.fft.ifft(X)
Out[279]: GF([^11363,  ^2127, ^15189,  ^5863,  ^1240,   ^255], order=7^5)
```

See also *ntt()* and *primitive_root_of_unity*.

Inverse FFT: np.fft.ifft(X)

The inverse Discrete Fourier Transform (DFT) of size n over the finite field $\mathrm{GF}(p^m)$ exists when there exists a primitive n-th root of unity. This occurs when $n\ |p^m - 1$.

Integer

```
In [280]: GF = galois.GF(7**5)

In [281]: n = 6

# n divides p^m - 1
In [282]: (GF.order - 1) % n
Out[282]: 0

In [283]: x = GF.Random(n, seed=1); x
Out[283]: GF([ 7952, 12470,  8601, 11055, 12691,  9895], order=7^5)

In [284]: X = np.fft.fft(x); X
Out[284]: GF([ 9387, 10789, 14695, 13079, 14025,  5694], order=7^5)

In [285]: np.fft.ifft(X)
Out[285]: GF([ 7952, 12470,  8601, 11055, 12691,  9895], order=7^5)
```

Polynomial

```
In [286]: GF = galois.GF(7**5, repr="poly")

In [287]: n = 6

# n divides p^m - 1
In [288]: (GF.order - 1) % n
Out[288]: 0

In [289]: x = GF.Random(n, seed=1); x
Out[289]: 
GF([    3^4 + 2^3 + ^2 + 2, 5^4 + ^3 + 2^2 + 3 + 3,
          3^4 + 4^3 + 3 + 5, 4^4 + 4^3 + ^2 + 4 + 2,
```

(continues on next page)

(continued from previous page)

```
                   5^4 + 2^3,       4^4 + 5^2 + 6 + 4], order=7^5)

In [290]: X = np.fft.fft(x); X
Out[290]:
GF([   3^4 + 6^3 + 2^2 + 4, 4^4 + 3^3 + 3^2 +  + 2,
          6^4 + 5^2 + 6 + 2,       5^4 + 3^3 + 6 + 3,
    5^4 + 5^3 + 6^2 +  + 4, 2^4 + 2^3 + 4^2 +  + 3], order=7^5)

In [291]: np.fft.ifft(X)
Out[291]:
GF([    3^4 + 2^3 + ^2 + 2, 5^4 + ^3 + 2^2 + 3 + 3,
          3^4 + 4^3 + 3 + 5, 4^4 + 4^3 + ^2 + 4 + 2,
                   5^4 + 2^3,       4^4 + 5^2 + 6 + 4], order=7^5)
```

Power

```
In [292]: GF = galois.GF(7**5, repr="power")

In [293]: n = 6

# n divides p^m - 1
In [294]: (GF.order - 1) % n
Out[294]: 0

In [295]: x = GF.Random(n, seed=1); x
Out[295]: GF([^11363,  ^2127, ^15189,  ^5863,  ^1240,   ^255], order=7^5)

In [296]: X = np.fft.fft(x); X
Out[296]: GF([ ^7664, ^14905, ^15266, ^13358,  ^9822, ^16312], order=7^5)

In [297]: np.fft.ifft(X)
Out[297]: GF([^11363,  ^2127, ^15189,  ^5863,  ^1240,   ^255], order=7^5)
```

See also *ntt()* and *primitive_root_of_unity*.

3.6.4 Linear algebra

Linear algebra on *FieldArray* arrays/matrices is supported through both native NumPy linear algebra functions in `numpy.linalg` and additional linear algebra methods not included in NumPy.

Expand any section for more details.

Dot product: np.dot(a, b)

```
In [298]: GF = galois.GF(31)

In [299]: a = GF([29, 0, 2, 21]); a
Out[299]: GF([29,  0,  2, 21], order=31)
```

(continues on next page)

(continued from previous page)

```
In [300]: b = GF([23, 5, 15, 12]); b
Out[300]: GF([23,  5, 15, 12], order=31)

In [301]: np.dot(a, b)
Out[301]: GF(19, order=31)
```

Vector dot product: `np.vdot(a, b)`

```
In [302]: GF = galois.GF(31)

In [303]: a = GF([29, 0, 2, 21]); a
Out[303]: GF([29,  0,  2, 21], order=31)

In [304]: b = GF([23, 5, 15, 12]); b
Out[304]: GF([23,  5, 15, 12], order=31)

In [305]: np.vdot(a, b)
Out[305]: GF(19, order=31)
```

Inner product: `np.inner(a, b)`

```
In [306]: GF = galois.GF(31)

In [307]: a = GF([29, 0, 2, 21]); a
Out[307]: GF([29,  0,  2, 21], order=31)

In [308]: b = GF([23, 5, 15, 12]); b
Out[308]: GF([23,  5, 15, 12], order=31)

In [309]: np.inner(a, b)
Out[309]: GF(19, order=31)
```

Outer product: `np.outer(a, b)`

```
In [310]: GF = galois.GF(31)

In [311]: a = GF([29, 0, 2, 21]); a
Out[311]: GF([29,  0,  2, 21], order=31)

In [312]: b = GF([23, 5, 15, 12]); b
Out[312]: GF([23,  5, 15, 12], order=31)

In [313]: np.outer(a, b)
Out[313]: 
GF([[16, 21,  1,  7],
    [ 0,  0,  0,  0],
```

(continues on next page)

(continued from previous page)

```
    [15, 10, 30, 24],
    [18, 12,  5,  4]], order=31)
```

Matrix multiplication: `A @ B == np.matmul(A, B)`

```
In [314]: GF = galois.GF(31)

In [315]: A = GF([[17, 25, 18, 8], [7, 9, 21, 15], [6, 16, 6, 30]]); A
Out[315]:
GF([[17, 25, 18,  8],
    [ 7,  9, 21, 15],
    [ 6, 16,  6, 30]], order=31)

In [316]: B = GF([[8, 18], [22, 0], [7, 8], [20, 24]]); B
Out[316]:
GF([[ 8, 18],
    [22,  0],
    [ 7,  8],
    [20, 24]], order=31)

In [317]: A @ B
Out[317]:
GF([[11, 22],
    [19,  3],
    [19,  8]], order=31)

In [318]: np.matmul(A, B)
Out[318]:
GF([[11, 22],
    [19,  3],
    [19,  8]], order=31)
```

Matrix exponentiation: `np.linalg.matrix_power(A, 3)`

```
In [319]: GF = galois.GF(31)

In [320]: A = GF([[14, 1, 5], [3, 23, 6], [24, 27, 4]]); A
Out[320]:
GF([[14,  1,  5],
    [ 3, 23,  6],
    [24, 27,  4]], order=31)

In [321]: np.linalg.matrix_power(A, 3)
Out[321]:
GF([[ 1, 16,  4],
    [11,  9,  9],
    [ 8, 24, 29]], order=31)
```

(continues on next page)

(continued from previous page)

```
In [322]: A @ A @ A
Out[322]:
GF([[ 1, 16,  4],
    [11,  9,  9],
    [ 8, 24, 29]], order=31)
```

Matrix determinant: `np.linalg.det(A)`

```
In [323]: GF = galois.GF(31)

In [324]: A = GF([[23, 11, 3, 3], [13, 6, 16, 4], [12, 10, 5, 3], [17, 23, 15, 28]]); A
Out[324]:
GF([[23, 11,  3,  3],
    [13,  6, 16,  4],
    [12, 10,  5,  3],
    [17, 23, 15, 28]], order=31)

In [325]: np.linalg.det(A)
Out[325]: GF(0, order=31)
```

Matrix rank: `np.linalg.matrix_rank(A)`

```
In [326]: GF = galois.GF(31)

In [327]: A = GF([[23, 11, 3, 3], [13, 6, 16, 4], [12, 10, 5, 3], [17, 23, 15, 28]]); A
Out[327]:
GF([[23, 11,  3,  3],
    [13,  6, 16,  4],
    [12, 10,  5,  3],
    [17, 23, 15, 28]], order=31)

In [328]: np.linalg.matrix_rank(A)
Out[328]: 3

In [329]: A.row_reduce()
Out[329]:
GF([[ 1,  0,  0, 11],
    [ 0,  1,  0, 25],
    [ 0,  0,  1, 11],
    [ 0,  0,  0,  0]], order=31)
```

Matrix trace: `np.trace(A)`

```
In [330]: GF = galois.GF(31)

In [331]: A = GF([[23, 11, 3, 3], [13, 6, 16, 4], [12, 10, 5, 3], [17, 23, 15, 28]]); A
Out[331]:
```

(continues on next page)

(continued from previous page)

```
GF([[23, 11,  3,  3],
    [13,  6, 16,  4],
    [12, 10,  5,  3],
    [17, 23, 15, 28]], order=31)

In [332]: np.trace(A)
Out[332]: GF(0, order=31)

In [333]: A[0,0] + A[1,1] + A[2,2] + A[3,3]
Out[333]: GF(0, order=31)
```

Solve a system of equations: `np.linalg.solve(A, b)`

```
In [334]: GF = galois.GF(31)

In [335]: A = GF([[14, 21, 14, 28], [24, 22, 23, 23], [16, 30, 26, 18], [4, 23, 18, 3]]);
↪ A
Out[335]:
GF([[14, 21, 14, 28],
    [24, 22, 23, 23],
    [16, 30, 26, 18],
    [ 4, 23, 18,  3]], order=31)

In [336]: b = GF([15, 11, 6, 29]); b
Out[336]: GF([15, 11,  6, 29], order=31)

In [337]: x = np.linalg.solve(A, b)

In [338]: np.array_equal(A @ x, b)
Out[338]: True
```

Matrix inverse: `np.linalg.inv(A)`

```
In [339]: GF = galois.GF(31)

In [340]: A = GF([[14, 21, 14, 28], [24, 22, 23, 23], [16, 30, 26, 18], [4, 23, 18, 3]]);
↪ A
Out[340]:
GF([[14, 21, 14, 28],
    [24, 22, 23, 23],
    [16, 30, 26, 18],
    [ 4, 23, 18,  3]], order=31)

In [341]: A_inv = np.linalg.inv(A); A_inv
Out[341]:
GF([[27, 17,  9,  8],
    [20, 21, 12,  4],
    [30, 10, 23, 22],
```

(continues on next page)

(continued from previous page)

```
    [13, 25,  6, 13]], order=31)

In [342]: A @ A_inv
Out[342]:
GF([[1, 0, 0, 0],
    [0, 1, 0, 0],
    [0, 0, 1, 0],
    [0, 0, 0, 1]], order=31)
```

3.7 Polynomials

Univariate polynomials over finite fields are supported with the *Poly* class.

3.7.1 Create a polynomial

Create a polynomial by specifying its coefficients in degree-descending order and the finite field its over.

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

In [2]: galois.Poly([1, 0, 0, 55, 23], field=GF)
Out[2]: Poly(x^4 + 55x + 23, GF(2^8))
```

Or pass a *FieldArray* of coefficients without explicitly specifying the finite field.

```
In [3]: coeffs = GF([1, 0, 0, 55, 23]); coeffs
Out[3]: GF([ 1,  0,  0, 55, 23], order=2^8)

In [4]: galois.Poly(coeffs)
Out[4]: Poly(x^4 + 55x + 23, GF(2^8))
```

Use `set_printoptions()` to display the polynomial coefficients in degree-ascending order.

```
In [5]: galois.set_printoptions(coeffs="asc")

In [6]: galois.Poly(coeffs)
Out[6]: Poly(23 + 55x + x^4, GF(2^8))
```

3.7.2 Element representation

As with *FieldArray* instances, the finite field element representation of the polynomial coefficients may be changed by setting the `repr` keyword argument of *GF()* or using the *repr()* classmethod.

```
In [7]: GF = galois.GF(3**5)

# Display f(x) using the default integer representation
In [8]: f = galois.Poly([13, 0, 4, 2], field=GF); print(f)
13x^3 + 4x + 2

# Display f(x) using the polynomial representation
In [9]: GF.repr("poly"); print(f)
(^2 +  + 1)x^3 + ( + 1)x + 2

# Display f(x) using the power representation
In [10]: GF.repr("power"); print(f)
(^10)x^3 + (^69)x + ^121

In [11]: GF.repr("int");
```

See *Element Representation* for more details.

3.7.3 Alternate constructors

There are several additional ways to create a polynomial. These alternate constructors are included as classmethods in *Poly*. By convention, alternate constructors use `PascalCase` while other classmethods use `snake_case`.

Create a polynomial by specifying its non-zero degrees and coefficients using *Degrees()*.

```
In [12]: galois.Poly.Degrees([1000, 1], coeffs=[1, 179], field=GF)
Out[12]: Poly(x^1000 + 179x, GF(3^5))
```

Create a polynomial from its integer representation using *Int()*. Additionally, one may create a polynomial from a binary, octal, or hexadecimal string of its integer representation.

Integer

```
In [13]: galois.Poly.Int(268, field=GF)
Out[13]: Poly(x + 25, GF(3^5))
```

Binary string

```
In [14]: galois.Poly.Int(int("0b1011", 2))
Out[14]: Poly(x^3 + x + 1, GF(2))
```

Octal string

```
In [15]: galois.Poly.Int(int("0o5034", 8), field=galois.GF(2**3))
Out[15]: Poly(5x^3 + 3x + 4, GF(2^3))
```

Hex string

```
In [16]: galois.Poly.Int(int("0xf700a275", 16), field=galois.GF(2**8))
Out[16]: Poly(247x^3 + 162x + 117, GF(2^8))
```

Create a polynomial from its string representation using *Str()*.

```
In [17]: galois.Poly.Str("x^5 + 143", field=GF)
Out[17]: Poly(x^5 + 143, GF(3^5))
```

Create a polynomial from its roots using *Roots()*.

```
In [18]: f = galois.Poly.Roots([137, 22, 51], field=GF); f
Out[18]: Poly(x^3 + 180x^2 + 19x + 58, GF(3^5))

In [19]: f.roots()
Out[19]: GF([ 22,  51, 137], order=3^5)
```

The *Zero()*, *One()*, and *Identity()* classmethods create common, simple polynomials. They are included for convenience.

```
In [20]: galois.Poly.Zero(GF)
Out[20]: Poly(0, GF(3^5))

In [21]: galois.Poly.One(GF)
Out[21]: Poly(1, GF(3^5))

In [22]: galois.Poly.Identity(GF)
Out[22]: Poly(x, GF(3^5))
```

Random polynomials of a given degree are easily created with *Random()*.

```
In [23]: galois.Poly.Random(4, field=GF)
Out[23]: Poly(50x^4 + 153x^3 + 10x^2 + 50x + 1, GF(3^5))
```

3.7.4 Methods

Polynomial objects have several methods that modify or perform operations on the polynomial. Below are some examples.

Compute the derivative of a polynomial using *derivative()*.

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

In [25]: f = galois.Poly([1, 0, 5, 2, 3], field=GF); f
Out[25]: Poly(x^4 + 5x^2 + 2x + 3, GF(7))
```

(continues on next page)

(continued from previous page)

```
In [26]: f.derivative()
Out[26]: Poly(4x^3 + 3x + 2, GF(7))
```

Compute the roots of a polynomial using *roots()*.

```
In [27]: f.roots()
Out[27]: GF([5, 6], order=7)
```

3.7.5 Properties

Polynomial objects have several instance properties. Below are some examples.

Find the non-zero degrees and coefficients of the polynomial using *nonzero_degrees* and *nonzero_coeffs*.

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

In [29]: f = galois.Poly([1, 0, 3], field=GF); f
Out[29]: Poly(x^2 + 3, GF(7))

In [30]: f.nonzero_degrees
Out[30]: array([2, 0])

In [31]: f.nonzero_coeffs
Out[31]: GF([1, 3], order=7)
```

Find the integer equivalent of the polynomial using `int()`, see *__int__()*. Additionally, one may convert a polynomial into the binary, octal, or hexadecimal string of its integer representation.

Integer

```
In [32]: int(f)
Out[32]: 52
```

Binary string

```
In [33]: g = galois.Poly([1, 0, 1, 1]); g
Out[33]: Poly(x^3 + x + 1, GF(2))

In [34]: bin(g)
Out[34]: '0b1011'
```

Octal string

```
In [35]: g = galois.Poly([5, 0, 3, 4], field=galois.GF(2**3)); g
Out[35]: Poly(5x^3 + 3x + 4, GF(2^3))

In [36]: oct(g)
Out[36]: '0o5034'
```

Hex string

```
In [37]: g = galois.Poly([0xf7, 0x00, 0xa2, 0x75], field=galois.GF(2**8)); g
Out[37]: Poly(247x^3 + 162x + 117, GF(2^8))

In [38]: hex(g)
Out[38]: '0xf700a275'
```

Get the string representation of the polynomial using `str()`.

```
In [39]: str(f)
Out[39]: 'x^2 + 3'
```

3.7.6 Special polynomials

The `galois` library also includes several functions to find certain *special* polynomials. Below are some examples.

Find one or all irreducible polynomials with `irreducible_poly()` and `irreducible_polys()`.

```
In [40]: galois.irreducible_poly(3, 3)
Out[40]: Poly(x^3 + 2x + 1, GF(3))

In [41]: list(galois.irreducible_polys(3, 3))
Out[41]:
[Poly(x^3 + 2x + 1, GF(3)),
 Poly(x^3 + 2x + 2, GF(3)),
 Poly(x^3 + x^2 + 2, GF(3)),
 Poly(x^3 + x^2 + x + 2, GF(3)),
 Poly(x^3 + x^2 + 2x + 1, GF(3)),
 Poly(x^3 + 2x^2 + 1, GF(3)),
 Poly(x^3 + 2x^2 + x + 1, GF(3)),
 Poly(x^3 + 2x^2 + 2x + 2, GF(3))]
```

Find one or all primitive polynomials with `primitive_poly()` and `primitive_polys()`.

```
In [42]: galois.primitive_poly(3, 3)
Out[42]: Poly(x^3 + 2x + 1, GF(3))

In [43]: list(galois.primitive_polys(3, 3))
Out[43]:
[Poly(x^3 + 2x + 1, GF(3)),
 Poly(x^3 + x^2 + 2x + 1, GF(3)),
 Poly(x^3 + 2x^2 + 1, GF(3)),
 Poly(x^3 + 2x^2 + x + 1, GF(3))]
```

Find the Conway polynomial using *conway_poly()*.

```
In [44]: galois.conway_poly(3, 3)
Out[44]: Poly(x^3 + 2x + 1, GF(3))
```

3.8 Polynomial Arithmetic

In the sections below, the finite field $\mathrm{GF}(7)$ and polynomials $f(x)$ and $g(x)$ are used.

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

In [2]: f = galois.Poly([1, 0, 4, 3], field=GF); f
Out[2]: Poly(x^3 + 4x + 3, GF(7))

In [3]: g = galois.Poly([2, 1, 3], field=GF); g
Out[3]: Poly(2x^2 + x + 3, GF(7))
```

3.8.1 Standard arithmetic

After creating a *polynomial over a finite field*, nearly any polynomial arithmetic operation can be performed using Python operators. Expand any section for more details.

Addition: `f + g`

Add two polynomials.

```
In [4]: f
Out[4]: Poly(x^3 + 4x + 3, GF(7))

In [5]: g
Out[5]: Poly(2x^2 + x + 3, GF(7))

In [6]: f + g
Out[6]: Poly(x^3 + 2x^2 + 5x + 6, GF(7))
```

Add a polynomial and a finite field scalar. The scalar is treated as a 0-degree polynomial.

```
In [7]: f + GF(3)
Out[7]: Poly(x^3 + 4x + 6, GF(7))

In [8]: GF(3) + f
Out[8]: Poly(x^3 + 4x + 6, GF(7))
```

Additive inverse: `-f`

```
In [9]: f
Out[9]: Poly(x^3 + 4x + 3, GF(7))
```

(continues on next page)

(continued from previous page)

```
In [10]: -f
Out[10]: Poly(6x^3 + 3x + 4, GF(7))
```

Any polynomial added to its additive inverse results in zero.

```
In [11]: f
Out[11]: Poly(x^3 + 4x + 3, GF(7))

In [12]: f + -f
Out[12]: Poly(0, GF(7))
```

Subtraction: `f - g`

Subtract one polynomial from another.

```
In [13]: f
Out[13]: Poly(x^3 + 4x + 3, GF(7))

In [14]: g
Out[14]: Poly(2x^2 + x + 3, GF(7))

In [15]: f - g
Out[15]: Poly(x^3 + 5x^2 + 3x, GF(7))
```

Subtract finite field scalar from a polynomial, or vice versa. The scalar is treated as a 0-degree polynomial.

```
In [16]: f - GF(3)
Out[16]: Poly(x^3 + 4x, GF(7))

In [17]: GF(3) - f
Out[17]: Poly(6x^3 + 3x, GF(7))
```

Multiplication: `f * g`

Multiply two polynomials.

```
In [18]: f
Out[18]: Poly(x^3 + 4x + 3, GF(7))

In [19]: g
Out[19]: Poly(2x^2 + x + 3, GF(7))

In [20]: f * g
Out[20]: Poly(2x^5 + x^4 + 4x^3 + 3x^2 + x + 2, GF(7))
```

Multiply a polynomial and a finite field scalar. The scalar is treated as a 0-degree polynomial.

```
In [21]: f * GF(3)
Out[21]: Poly(3x^3 + 5x + 2, GF(7))
```

(continues on next page)

(continued from previous page)

```
In [22]: GF(3) * f
Out[22]: Poly(3x^3 + 5x + 2, GF(7))
```

Scalar multiplication: `f * 3`

Scalar multiplication is essentially *repeated addition*. It is the "multiplication" of finite field elements and integers. The integer value indicates how many additions of the field element to sum.

```
In [23]: f * 4
Out[23]: Poly(4x^3 + 2x + 5, GF(7))

In [24]: f + f + f + f
Out[24]: Poly(4x^3 + 2x + 5, GF(7))
```

In finite fields $\mathrm{GF}(p^m)$, the characteristic p is the smallest value when multiplied by any non-zero field element that always results in 0.

```
In [25]: p = GF.characteristic; p
Out[25]: 7

In [26]: f * p
Out[26]: Poly(0, GF(7))
```

Division: `f // g`

Divide one polynomial by another. Floor division is supported. True division is not supported since fractional polynomials are not currently supported.

```
In [27]: f
Out[27]: Poly(x^3 + 4x + 3, GF(7))

In [28]: g
Out[28]: Poly(2x^2 + x + 3, GF(7))

In [29]: f // g
Out[29]: Poly(4x + 5, GF(7))
```

Divide a polynomial by a finite field scalar, or vice versa. The scalar is treated as a 0-degree polynomial.

```
In [30]: f // GF(3)
Out[30]: Poly(5x^3 + 6x + 1, GF(7))

In [31]: GF(3) // g
Out[31]: Poly(0, GF(7))
```

Remainder: `f % g`

Divide one polynomial by another and keep the remainder.

```
In [32]: f
Out[32]: Poly(x^3 + 4x + 3, GF(7))

In [33]: g
Out[33]: Poly(2x^2 + x + 3, GF(7))

In [34]: f % g
Out[34]: Poly(x + 2, GF(7))
```

Divide a polynomial by a finite field scalar, or vice versa, and keep the remainder. The scalar is treated as a 0-degree polynomial.

```
In [35]: f % GF(3)
Out[35]: Poly(0, GF(7))

In [36]: GF(3) % g
Out[36]: Poly(3, GF(7))
```

Divmod: `divmod(f, g)`

Divide one polynomial by another and return the quotient and remainder.

```
In [37]: f
Out[37]: Poly(x^3 + 4x + 3, GF(7))

In [38]: g
Out[38]: Poly(2x^2 + x + 3, GF(7))

In [39]: divmod(f, g)
Out[39]: (Poly(4x + 5, GF(7)), Poly(x + 2, GF(7)))
```

Divide a polynomial by a finite field scalar, or vice versa, and keep the remainder. The scalar is treated as a 0-degree polynomial.

```
In [40]: divmod(f, GF(3))
Out[40]: (Poly(5x^3 + 6x + 1, GF(7)), Poly(0, GF(7)))

In [41]: divmod(GF(3), g)
Out[41]: (Poly(0, GF(7)), Poly(3, GF(7)))
```

Exponentiation: `f ** 3`

Exponentiate a polynomial to a non-negative exponent.

```
In [42]: f
Out[42]: Poly(x^3 + 4x + 3, GF(7))

In [43]: f ** 3
Out[43]: Poly(x^9 + 5x^7 + 2x^6 + 6x^5 + 2x^4 + 4x^2 + 3x + 6, GF(7))
```

(continues on next page)

(continued from previous page)

```
In [44]: pow(f, 3)
Out[44]: Poly(x^9 + 5x^7 + 2x^6 + 6x^5 + 2x^4 + 4x^2 + 3x + 6, GF(7))

In [45]: f * f * f
Out[45]: Poly(x^9 + 5x^7 + 2x^6 + 6x^5 + 2x^4 + 4x^2 + 3x + 6, GF(7))
```

Modular exponentiation: `pow(f, 123456789, g)`

Exponentiate a polynomial to a non-negative exponent and reduce modulo another polynomial. This performs efficient modular exponentiation.

```
In [46]: f
Out[46]: Poly(x^3 + 4x + 3, GF(7))

In [47]: g
Out[47]: Poly(2x^2 + x + 3, GF(7))

# Efficiently computes (f ** 123456789) % g
In [48]: pow(f, 123456789, g)
Out[48]: Poly(x + 2, GF(7))
```

3.8.2 Evaluation

Polynomial objects may also be evaluated at scalars, arrays, or square matrices. Expand any section for more details.

Evaluation (element-wise): `f(x)` or `f(X)`

Polynomials are evaluated by invoking `__call__()`. They can be evaluated at scalars.

```
In [49]: f
Out[49]: Poly(x^3 + 4x + 3, GF(7))

In [50]: f(5)
Out[50]: GF(1, order=7)

# The equivalent field calculation
In [51]: GF(5)**3 + 4*GF(5) + GF(3)
Out[51]: GF(1, order=7)
```

Or they can be evaluated at arrays element-wise.

```
In [52]: x = GF([5, 6, 3, 4])

# Evaluate f(x) element-wise at a 1-D array
In [53]: f(x)
Out[53]: GF([1, 5, 0, 6], order=7)
```

```
In [54]: X = GF([[5, 6], [3, 4]])

# Evaluate f(x) element-wise at a 2-D array
In [55]: f(X)
Out[55]:
GF([[1, 5],
    [0, 6]], order=7)
```

Evaluation (square matrix): `f(X, elementwise=False)`

Polynomials can also be evaluated at square matrices. Note, this is different than element-wise array evaluation. Here, the square matrix indeterminate is exponentiated using matrix multiplication. So $f(x) = x^3$ evaluated at the square matrix `X` equals `X @ X @ X`.

```
In [56]: f
Out[56]: Poly(x^3 + 4x + 3, GF(7))

In [57]: f(X, elementwise=False)
Out[57]:
GF([[1, 1],
    [4, 2]], order=7)

# The equivalent matrix operation
In [58]: np.linalg.matrix_power(X, 3) + 4*X + GF(3)*GF.Identity(X.shape[0])
Out[58]:
GF([[1, 1],
    [4, 2]], order=7)
```

Composition: `f(g)`

Polynomial composition $f(g(x))$ is easily performed using an overload to `__call__()`.

```
In [59]: f
Out[59]: Poly(x^3 + 4x + 3, GF(7))

In [60]: g
Out[60]: Poly(2x^2 + x + 3, GF(7))

In [61]: f(g)
Out[61]: Poly(x^6 + 5x^5 + 2x^3 + x^2 + 3x, GF(7))
```

3.8.3 Special arithmetic

Polynomial objects also work on several special arithmetic operations. Expand any section for more details.

Greatest common denominator: `galois.gcd(f, g)`

```
In [62]: f
Out[62]: Poly(x^3 + 4x + 3, GF(7))

In [63]: g
Out[63]: Poly(2x^2 + x + 3, GF(7))

In [64]: d = galois.gcd(f, g); d
Out[64]: Poly(1, GF(7))

In [65]: f % d
Out[65]: Poly(0, GF(7))

In [66]: g % d
Out[66]: Poly(0, GF(7))
```

See *gcd()* for more details.

Extended greatest common denominator: `galois.egcd(f, g)`

```
In [67]: f
Out[67]: Poly(x^3 + 4x + 3, GF(7))

In [68]: g
Out[68]: Poly(2x^2 + x + 3, GF(7))

In [69]: d, s, t = galois.egcd(f, g)

In [70]: d, s, t
Out[70]: (Poly(1, GF(7)), Poly(6x + 5, GF(7)), Poly(4x^2 + 6x, GF(7)))

In [71]: f*s + g*t == d
Out[71]: True
```

See *egcd()* for more details.

Factor into irreducible polynomials: `galois.factors(f) == f.factors()`

```
In [72]: f
Out[72]: Poly(x^3 + 4x + 3, GF(7))

In [73]: galois.factors(f)
Out[73]: ([Poly(x + 4, GF(7)), Poly(x^2 + 3x + 6, GF(7))], [1, 1])

In [74]: f.factors()
Out[74]: ([Poly(x + 4, GF(7)), Poly(x^2 + 3x + 6, GF(7))], [1, 1])
```

See *factors()* or *galois.Poly.factors()* for more details.

3.9 Intro to 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 another element from the set. A *finite field* is a field with a finite set of elements.

Évariste Galois

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

- May 29, 1832 (two days before his death)

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.

3.9.1 Prime field

In this tutorial, we will consider the prime field $\mathrm{GF}(7)$. Using the *galois* library, the *FieldArray* subclass GF7 is created using the class factory *GF()*.

Integer

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

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

Power

```
In [3]: GF7 = galois.GF(7, repr="power")

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

Info

In this tutorial, we suggest using the integer representation to display the elements. However, sometimes it is useful to view elements in their power representation $\{0, 1, \alpha, \alpha^2, \dots, \alpha^{p^m-2}\}$. Switch the display between these two representations using the tabbed sections. Note, the polynomial representation is not shown because it is identical to the integer representation for prime fields.

See *Element Representation* for more details.

3.9.2 Elements

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

The elements of the finite field are retrieved in a 1-D array using the `Elements()` classmethod.

Integer

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

Power

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

3.9.3 Arithmetic

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

In this tutorial, consider two field elements $a = 3$ and $b = 5$. We will use `galois` to perform explicit modular integer arithmetic and then prime field arithmetic.

Here are a and b represented as Python integers.

```
In [7]: a_int = 3

In [8]: b_int = 5

In [9]: p = GF7.characteristic; p
Out[9]: 7
```

Here are a and b represented as prime field elements. See *Array Creation* for more details.

Integer

```
In [10]: a = GF7(3); a
Out[10]: GF(3, order=7)

In [11]: b = GF7(5); b
Out[11]: GF(5, order=7)
```

Power

```
In [12]: a = GF7(3); a
Out[12]: GF(, order=7)

In [13]: b = GF7(5); b
Out[13]: GF(^5, order=7)
```

Addition

We can see that $3 + 5 \equiv 1 \pmod{7}$. So accordingly, $3 + 5 = 1$ in $\mathrm{GF}(7)$.

Integer

```
In [14]: (a_int + b_int) % p
Out[14]: 1

In [15]: a + b
Out[15]: GF(1, order=7)
```

Power

```
In [16]: (a_int + b_int) % p
Out[16]: 1

In [17]: a + b
Out[17]: GF(1, order=7)
```

The `galois` library includes the ability to display the arithmetic tables for any 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.

Integer

```
In [18]: 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
    5 | 5  6  0  1  2  3  4
    6 | 6  0  1  2  3  4  5
```

Power

```
In [19]: print(GF7.arithmetic_table("+"))
x + y |   0    1      ^2  ^3  ^4  ^5
------|-----------------------------------
    0 |   0    1      ^2  ^3  ^4  ^5
    1 |   1  ^2  ^4        0  ^5  ^3
     |     ^4  ^3  ^5  ^2    0    1
  ^2 | ^2      ^5  ^4    1  ^3    0
  ^3 | ^3    0  ^2    1  ^5      ^4
  ^4 | ^4  ^5    0  ^3        1  ^2
  ^5 | ^5  ^3    1    0  ^4  ^2
```

Subtraction

As with addition, we can see that $3 - 5 \equiv 5 \pmod 7$. So accordingly, $3 - 5 = 5$ in $\mathrm{GF}(7)$.

Integer

```
In [20]: (a_int - b_int) % p
Out[20]: 5

In [21]: a - b
Out[21]: GF(5, order=7)
```

Power

```
In [22]: (a_int - b_int) % p
Out[22]: 5

In [23]: a - b
Out[23]: GF(^5, order=7)
```

Here is the subtraction table for completeness.

Integer

```
In [24]: 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
```

Power

```
In [25]: print(GF7.arithmetic_table("-"))
x - y |   0    1      ^2  ^3  ^4  ^5
------|-----------------------------------
    0 |   0  ^3  ^4  ^5    1      ^2
    1 |   1    0  ^5  ^3  ^2  ^4
     |     ^2    0    1  ^4  ^3  ^5
  ^2 | ^2    1  ^3    0      ^5  ^4
  ^3 | ^3  ^5      ^4    0  ^2    1
  ^4 | ^4        1  ^2  ^5    0  ^3
  ^5 | ^5  ^4  ^2      ^3    1    0
```

Multiplication

Similarly, we can see that $3 \cdot 5 \equiv 1 \pmod 7$. So accordingly, $3 \cdot 5 = 1$ in $\mathrm{GF}(7)$.

Integer

```
In [26]: (a_int * b_int) % p
Out[26]: 1

In [27]: a * b
Out[27]: GF(1, order=7)
```

Power

```
In [28]: (a_int * b_int) % p
Out[28]: 1

In [29]: a * b
Out[29]: GF(1, order=7)
```

Here is the multiplication table for completeness.

Integer

```
In [30]: 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
```

Power

```
In [31]: print(GF7.arithmetic_table("*"))
x * y |   0    1      ^2   ^3   ^4   ^5
------|-------------------------------------
    0 |   0    0    0    0    0    0    0
    1 |   0    1      ^2   ^3   ^4   ^5
      |  0      ^2   ^3   ^4   ^5    1
   ^2 |  0  ^2   ^3   ^4   ^5    1
   ^3 |  0  ^3   ^4   ^5    1      ^2
   ^4 |  0  ^4   ^5    1      ^2   ^3
   ^5 |  0  ^5    1      ^2   ^3   ^4
```

Multiplicative inverse

Division in $\mathrm{GF}(p)$ is a little more difficult. Division can't be as simple as taking $a/b \pmod p$ because many integer divisions do not result in integers! The division a/b can be reformulated into ab^{-1}, where b^{-1} is the multiplicative inverse of b. Let's first learn the multiplicative inverse before returning to division.

Euclid discovered an efficient algorithm to solve the Bézout Identity, which is used to find the multiplicative inverse. It is now called the Extended Euclidean Algorithm. Given two integers x and y, the Extended Euclidean Algorithm finds the integers s and t such that $xs + yt = \gcd(x, y)$. This algorithm is implemented in *egcd()*.

If $x = 5$ is a field element of $\mathrm{GF}(7)$ and $y = 7$ is the prime characteristic, then $s = x^{-1}$ in $\mathrm{GF}(7)$. Note, the GCD will always be 1 because y is prime.

```
# Returns (gcd, s, t)
In [32]: galois.egcd(b_int, p)
Out[32]: (1, 3, -2)
```

The *galois* library uses the Extended Euclidean Algorithm to compute multiplicative inverses (and division) in prime fields. The inverse of 5 in $\mathrm{GF}(7)$ can be easily computed in the following way.

Integer

```
In [33]: b ** -1
Out[33]: GF(3, order=7)

In [34]: np.reciprocal(b)
Out[34]: GF(3, order=7)
```

Power

```
In [35]: b ** -1
Out[35]: GF(, order=7)

In [36]: np.reciprocal(b)
Out[36]: GF(, order=7)
```

Division

Now let's return to division in finite fields. As mentioned earlier, a/b is equivalent to ab^{-1}, and we have already learned multiplication and multiplicative inversion in finite fields.

To compute $3/5$ in $\mathrm{GF}(7)$, we can equivalently compute $3 \cdot 5^{-1}$ in $\mathrm{GF}(7)$.

Integer

```
In [37]: _, b_inv_int, _ = galois.egcd(b_int, p)

In [38]: (a_int * b_inv_int) % p
Out[38]: 2

In [39]: a * b**-1
Out[39]: GF(2, order=7)

In [40]: a / b
Out[40]: GF(2, order=7)
```

Power

```
In [41]: _, b_inv_int, _ = galois.egcd(b_int, p)

In [42]: (a_int * b_inv_int) % p
Out[42]: 2

In [43]: a * b**-1
Out[43]: GF(^2, order=7)

In [44]: a / b
Out[44]: GF(^2, order=7)
```

Here is the division table for completeness. Notice that division is not defined for $y = 0$.

Integer

```
In [45]: 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
```

Power

```
In [46]: print(GF7.arithmetic_table("/"))
x / y |    1       ^2   ^3   ^4   ^5
------|-----------------------------
    0 |    0    0    0    0    0    0
    1 |    1   ^5   ^4   ^3   ^2
      |         1   ^5   ^4   ^3   ^2
   ^2 |  ^2          1   ^5   ^4   ^3
   ^3 |  ^3   ^2          1   ^5   ^4
   ^4 |  ^4   ^3   ^2          1   ^5
   ^5 |  ^5   ^4   ^3   ^2          1
```

3.9.4 Primitive elements

A property of finite fields is that some elements produce the non-zero elements of the field by their powers.

A *primitive element* g of $\mathrm{GF}(p)$ is an element such that $\mathrm{GF}(p) = \{0, 1, g, g^2, \dots, g^{p-2}\}$. The non-zero elements $\{1, g, g^2, \dots, g^{p-2}\}$ form the cyclic multiplicative group $\mathrm{GF}(p)^\times$. A primitive element has multiplicative order $\mathrm{ord}(g) = p - 1$.

In prime fields $\mathrm{GF}(p)$, the generators or primitive elements of $\mathrm{GF}(p)$ are *primitive roots mod p*.

Primitive roots mod p

An 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 `primitive_root()` and `primitive_roots()`.

```
In [47]: galois.primitive_root(7)
Out[47]: 3
```

A primitive element

In *galois*, a primitive element of a finite field is provided by the *primitive_element* class property.

Integer

```
In [48]: print(GF7.properties)
Galois Field:
  name: GF(7)
  characteristic: 7
  degree: 1
  order: 7
  irreducible_poly: x + 4
  is_primitive_poly: True
  primitive_element: 3

In [49]: g = GF7.primitive_element; g
Out[49]: GF(3, order=7)
```

Power

```
In [50]: print(GF7.properties)
Galois Field:
  name: GF(7)
  characteristic: 7
  degree: 1
  order: 7
  irreducible_poly: x + 4
  is_primitive_poly: True
  primitive_element: 3

In [51]: g = GF7.primitive_element; g
Out[51]: GF(, order=7)
```

The *galois* package allows you to easily display all powers of an element and their equivalent polynomial, vector, and integer representations using *repr_table()*. Let's ignore the polynomial and vector representations for now. They will become useful for extension fields.

Here is the representation table using the default generator $g = 3$. Notice its multiplicative order is $p - 1$.

```
In [52]: g.multiplicative_order()
Out[52]: 6

In [53]: 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
```

(continues on next page)

(continued from previous page)

```
  3^4         4          [4]         4
  3^5         5          [5]         5
```

Other primitive elements

There are multiple primitive elements of any finite field. All primitive elements are provided in the *primitive_elements* class property.

Integer

```
In [54]: list(galois.primitive_roots(7))
Out[54]: [3, 5]

In [55]: GF7.primitive_elements
Out[55]: GF([3, 5], order=7)

In [56]: g = GF7(5); g
Out[56]: GF(5, order=7)
```

Power

```
In [57]: list(galois.primitive_roots(7))
Out[57]: [3, 5]

In [58]: GF7.primitive_elements
Out[58]: GF([  , ^5], order=7)

In [59]: g = GF7(5); g
Out[59]: GF(^5, order=7)
```

This means that 3 and 5 generate the multiplicative group $\mathrm{GF}(7)^\times$. We can examine this by viewing the representation table using different generators.

Here is the representation table using a different generator $g = 5$. Notice it also has multiplicative order $p - 1$.

```
In [60]: g.multiplicative_order()
Out[60]: 6

In [61]: print(GF7.repr_table(g))
 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
```

Non-primitive elements

All other elements of the field cannot generate the multiplicative group. They have multiplicative orders less than $p-1$.

For example, the element $e = 2$ is not a primitive element.

Integer

```
In [62]: e = GF7(2); e
Out[62]: GF(2, order=7)
```

Power

```
In [63]: e = GF7(2); e
Out[63]: GF(^2, order=7)
```

It has $\text{ord}(e) = 3$. Notice elements 3, 5, and 6 are not represented by the powers of e.

```
In [64]: e.multiplicative_order()
Out[64]: 3

In [65]: print(GF7.repr_table(e))
 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
```

3.10 Intro to Extension Fields

As discussed in the *Intro to Prime Fields* 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 is *mostly* computed using integer arithmetic modulo p. When the order is a prime power, namely extension fields $\text{GF}(p^m)$, the arithmetic is *mostly* computed using polynomial arithmetic modulo the irreducible polynomial $f(x)$.

3.10.1 Extension field

In this tutorial, we will consider the extension field $\mathrm{GF}(3^2)$. Using the *galois* library, the *FieldArray* subclass GF9 is created using the class factory *GF()*.

Integer

```
In [1]: GF9 = galois.GF(3**2)

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

Polynomial

```
In [3]: GF9 = galois.GF(3**2, repr="poly")

In [4]: print(GF9.properties)
Galois Field:
  name: GF(3^2)
  characteristic: 3
  degree: 2
  order: 9
  irreducible_poly: x^2 + 2x + 2
  is_primitive_poly: True
  primitive_element: x
```

Power

```
In [5]: GF9 = galois.GF(3**2, repr="power")

In [6]: print(GF9.properties)
Galois Field:
  name: GF(3^2)
  characteristic: 3
  degree: 2
  order: 9
  irreducible_poly: x^2 + 2x + 2
  is_primitive_poly: True
  primitive_element: x
```

Info

In this tutorial, we suggest using the polynomial representation to display the elements. Although, it is common to use the default integer representation $\{0, 1, \dots, p^m - 1\}$ to display the arrays more compactly. Switch the display between the three representations using the tabbed sections.

See *Element Representation* for more details.

3.10.2 Elements

The elements of $\mathrm{GF}(p^m)$ are polynomials over $\mathrm{GF}(p)$ with degree less than m. Formally, they are all polynomials $a_{m-1}x^{m-1} + \cdots + a_1x^1 + a_0 \in \mathrm{GF}(p)[x]$. There are exactly p^m elements.

The elements of the finite field are retrieved in a 1-D array using the `Elements()` classmethod.

Integer

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

Polynomial

```
In [8]: GF9.elements
Out[8]:
GF([     0,       1,       2,       ,    + 1,    + 2,      2, 2 + 1,
    2 + 2], order=3^2)
```

Power

```
In [9]: GF9.elements
Out[9]: GF([  0,    1, ^4,    , ^2, ^7, ^5, ^3, ^6], order=3^2)
```

3.10.3 Irreducible polynomial

Every extension field must be defined with respect to an irreducible polynomial $f(x)$. This polynomial defines the arithmetic of the field.

When creating a `FieldArray` subclass in `galois`, if an irreducible polynomial is not explicitly specified, a default is chosen. The default is the Conway polynomial $C_{p,m}(x)$, which is irreducible *and* primitive. See `conway_poly()` for more information.

Notice $f(x)$ is over $\mathrm{GF}(3)$ with degree 2.

```
In [10]: f = GF9.irreducible_poly; f
Out[10]: Poly(x^2 + 2x + 2, GF(3))
```

Also note, when factored, $f(x)$ has no irreducible factors other than itself – an analogue of a prime number.

```
In [11]: f.is_irreducible()
Out[11]: True

In [12]: f.factors()
Out[12]: ([Poly(x^2 + 2x + 2, GF(3))], [1])
```

3.10.4 Arithmetic

Addition, subtraction, and multiplication in $\mathrm{GF}(p^m)$ with irreducible polynomial $f(x)$ is equivalent to polynomial addition, subtraction, and multiplication over $\mathrm{GF}(p)$ reduced modulo $f(x)$. Mathematically speaking, this is the polynomial ring $\mathrm{GF}(p)[x]/f(x)$.

In this tutorial, consider two field elements $a = x + 2$ and $b = x + 1$. We will use `galois` to perform explicit polynomial calculations and then extension field arithmetic.

Here are a and b represented using `Poly` objects.

```
In [13]: GF3 = galois.GF(3)

In [14]: a_poly = galois.Poly([1, 2], field=GF3); a_poly
Out[14]: Poly(x + 2, GF(3))

In [15]: b_poly = galois.Poly([1, 1], field=GF3); b_poly
Out[15]: Poly(x + 1, GF(3))
```

Here are a and b represented as extension field elements. Extension field elements can be specified as integers or polynomial strings. See *Array Creation* for more details.

Integer

```
In [16]: a = GF9("x + 2"); a
Out[16]: GF(5, order=3^2)

In [17]: b = GF9("x + 1"); b
Out[17]: GF(4, order=3^2)
```

Polynomial

```
In [18]: a = GF9("x + 2"); a
Out[18]: GF( + 2, order=3^2)

In [19]: b = GF9("x + 1"); b
Out[19]: GF( + 1, order=3^2)
```

Power

```
In [20]: a = GF9("x + 2"); a
Out[20]: GF(^7, order=3^2)

In [21]: b = GF9("x + 1"); b
Out[21]: GF(^2, order=3^2)
```

Addition

In polynomial addition, the polynomial coefficients add degree-wise in $\mathrm{GF}(p)$. Addition of polynomials with degree less than m will never result in a polynomial of degree m or greater. Therefore, it is unnecessary to reduce modulo the degree-m polynomial $f(x)$, since the quotient will always be zero.

We can see that $a + b = (1 + 1)x + (2 + 1) = 2x$.

Integer

```
In [22]: a_poly + b_poly
Out[22]: Poly(2x, GF(3))

In [23]: a + b
Out[23]: GF(6, order=3^2)
```

Polynomial

```
In [24]: a_poly + b_poly
Out[24]: Poly(2x, GF(3))

In [25]: a + b
Out[25]: GF(2, order=3^2)
```

Power

```
In [26]: a_poly + b_poly
Out[26]: Poly(2x, GF(3))

In [27]: a + b
Out[27]: GF(^5, order=3^2)
```

The *galois* library includes the ability to display the arithmetic tables for any 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.

Integer

```
In [28]: 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
    7 | 7  8  6  1  2  0  4  5  3
    8 | 8  6  7  2  0  1  5  3  4
```

Polynomial

```
In [29]: 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
```

Power

```
In [30]: print(GF9.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
```

Subtraction

Subtraction, like addition, is performed on coefficients degree-wise and will never result in a polynomial with greater degree.

We can see that $a - b = (1 - 1)x + (2 - 1) = 1$.

Integer

```
In [31]: a_poly - b_poly
Out[31]: Poly(1, GF(3))

In [32]: a - b
Out[32]: GF(1, order=3^2)
```

Polynomial

```
In [33]: a_poly - b_poly
Out[33]: Poly(1, GF(3))

In [34]: a - b
Out[34]: GF(1, order=3^2)
```

Power

```
In [35]: a_poly - b_poly
Out[35]: Poly(1, GF(3))

In [36]: a - b
Out[36]: GF(1, order=3^2)
```

Here is the entire subtraction table for completeness.

Integer

```
In [37]: print(GF9.arithmetic_table("-"))
x - y | 0  1  2  3  4  5  6  7  8
------|---------------------------
    0 | 0  2  1  6  8  7  3  5  4
    1 | 1  0  2  7  6  8  4  3  5
    2 | 2  1  0  8  7  6  5  4  3
    3 | 3  5  4  0  2  1  6  8  7
    4 | 4  3  5  1  0  2  7  6  8
    5 | 5  4  3  2  1  0  8  7  6
    6 | 6  8  7  3  5  4  0  2  1
    7 | 7  6  8  4  3  5  1  0  2
    8 | 8  7  6  5  4  3  2  1  0
```

Polynomial

```
In [38]: print(GF9.arithmetic_table("-"))
 x - y |      0       1       2           + 1    + 2      2  2 + 1  2 + 2
-------|------------------------------------------------------------------------
     0 |      0       2       1      2  2 + 2  2 + 1           + 2    + 1
     1 |      1       0       2  2 + 1      2  2 + 2    + 1           + 2
     2 |      2       1       0  2 + 2  2 + 1      2    + 2    + 1
      |          + 2    + 1       0       2       1      2  2 + 2  2 + 1
  + 1 |   + 1          + 2       1       0       2  2 + 1      2  2 + 2
  + 2 |   + 2    + 1             2       1       0  2 + 2  2 + 1      2
    2 |     2  2 + 2  2 + 1           + 2    + 1       0       2       1
2 + 1 | 2 + 1      2  2 + 2    + 1           + 2       1       0       2
2 + 2 | 2 + 2  2 + 1      2    + 2    + 1              2       1       0
```

Power

```
In [39]: print(GF9.arithmetic_table("-"))
x - y |   0    1      ^2  ^3  ^4  ^5  ^6  ^7
------|---------------------------------------------
    0 |   0  ^4  ^5  ^6  ^7    1      ^2  ^3
    1 |   1    0  ^3  ^5      ^4  ^2  ^7  ^6
     |     ^7    0  ^4  ^6  ^2  ^5  ^3    1
  ^2 | ^2         1    0  ^5  ^7  ^3  ^6  ^4
  ^3 | ^3  ^5  ^2        0  ^6    1  ^4  ^7
  ^4 | ^4    1  ^6  ^3  ^2    0  ^7      ^5
  ^5 | ^5  ^6      ^7  ^4  ^3    0    1  ^2
  ^6 | ^6  ^3  ^7  ^2    1  ^5  ^4    0
  ^7 | ^7  ^2  ^4    1  ^3      ^6  ^5    0
```

Multiplication

Multiplication of polynomials with degree less than m, however, will often result in a polynomial of degree m or greater. Therefore, it is necessary to reduce the result modulo $f(x)$.

First compute $ab = (x+2)(x+1) = x^2+2$. Notice that x^2+2 has degree 2, but the elements of $\mathrm{GF}(3^2)$ can have degree at most 1. Therefore, reduction modulo $f(x)$ is required. After remainder division, we see that $ab \equiv x \bmod f(x)$.

Integer

```
# Note the degree is greater than 1
In [40]: a_poly * b_poly
Out[40]: Poly(x^2 + 2, GF(3))

In [41]: (a_poly * b_poly) % f
Out[41]: Poly(x, GF(3))

In [42]: a * b
Out[42]: GF(3, order=3^2)
```

Polynomial

```
# Note the degree is greater than 1
In [43]: a_poly * b_poly
Out[43]: Poly(x^2 + 2, GF(3))

In [44]: (a_poly * b_poly) % f
Out[44]: Poly(x, GF(3))

In [45]: a * b
Out[45]: GF(, order=3^2)
```

Power

```
# Note the degree is greater than 1
In [46]: a_poly * b_poly
Out[46]: Poly(x^2 + 2, GF(3))

In [47]: (a_poly * b_poly) % f
Out[47]: Poly(x, GF(3))

In [48]: a * b
Out[48]: GF(, order=3^2)
```

Here is the entire multiplication table for completeness.

Integer

```
In [49]: print(GF9.arithmetic_table("*"))
x * y | 0  1  2  3  4  5  6  7  8
------|---------------------------
    0 | 0  0  0  0  0  0  0  0  0
    1 | 0  1  2  3  4  5  6  7  8
    2 | 0  2  1  6  8  7  3  5  4
    3 | 0  3  6  4  7  1  8  2  5
    4 | 0  4  8  7  2  3  5  6  1
    5 | 0  5  7  1  3  8  2  4  6
    6 | 0  6  3  8  5  2  4  1  7
    7 | 0  7  5  2  6  4  1  8  3
    8 | 0  8  4  5  1  6  7  3  2
```

Polynomial

```
In [50]: 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
2 + 2 |       0  2 + 2    + 1    + 2       1      2  2 + 1              2
```

Power

```
In [51]: print(GF9.arithmetic_table("*"))
x * y |    0    1      ^2  ^3  ^4  ^5  ^6  ^7
------|---------------------------------------------
    0 |    0    0    0    0    0    0    0    0    0
    1 |    0    1      ^2  ^3  ^4  ^5  ^6  ^7
     |    0      ^2  ^3  ^4  ^5  ^6  ^7    1
  ^2 |    0  ^2  ^3  ^4  ^5  ^6  ^7    1
  ^3 |    0  ^3  ^4  ^5  ^6  ^7    1      ^2
  ^4 |    0  ^4  ^5  ^6  ^7    1      ^2  ^3
  ^5 |    0  ^5  ^6  ^7    1      ^2  ^3  ^4
  ^6 |    0  ^6  ^7    1      ^2  ^3  ^4  ^5
  ^7 |    0  ^7    1      ^2  ^3  ^4  ^5  ^6
```

Multiplicative inverse

As with prime fields, the division $a(x)/b(x)$ is reformulated into $a(x)b(x)^{-1}$. So, first we must compute the multiplicative inverse b^{-1} before continuing onto division.

The Extended Euclidean Algorithm, which was used in prime fields on integers, can be used for extension fields on polynomials. Given two polynomials $a(x)$ and $b(x)$, the Extended Euclidean Algorithm finds the polynomials $s(x)$ and $t(x)$ such that $a(x)s(x) + b(x)t(x) = \gcd(a(x), b(x))$. This algorithm is implemented in `egcd()`.

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

```
# Returns (gcd, s, t)
In [52]: galois.egcd(b_poly, f)
Out[52]: (Poly(1, GF(3)), Poly(2x + 2, GF(3)), Poly(1, GF(3)))
```

The `galois` library uses the Extended Euclidean Algorithm to compute multiplicative inverses (and division) in extension fields. The inverse of $x + 1$ in $\mathrm{GF}(3^2)$ can be easily computed in the following way.

Integer

```
In [53]: b ** -1
Out[53]: GF(8, order=3^2)

In [54]: np.reciprocal(b)
Out[54]: GF(8, order=3^2)
```

Polynomial

```
In [55]: b ** -1
Out[55]: GF(2 + 2, order=3^2)

In [56]: np.reciprocal(b)
Out[56]: GF(2 + 2, order=3^2)
```

Power

```
In [57]: b ** -1
Out[57]: GF(^6, order=3^2)

In [58]: np.reciprocal(b)
Out[58]: GF(^6, order=3^2)
```

Division

Now let's return to division in finite fields. As mentioned earlier, $a(x)/b(x)$ is equivalent to $a(x)b(x)^{-1}$, and we have already learned multiplication and multiplicative inversion in finite fields.

Let's compute $a/b = (x + 2)(x + 1)^{-1}$ in $\mathrm{GF}(3^2)$.

Integer

```
In [59]: _, b_inv_poly, _ = galois.egcd(b_poly, f)

In [60]: (a_poly * b_inv_poly) % f
Out[60]: Poly(2x, GF(3))

In [61]: a * b**-1
Out[61]: GF(6, order=3^2)

In [62]: a / b
Out[62]: GF(6, order=3^2)
```

Polynomial

```
In [63]: _, b_inv_poly, _ = galois.egcd(b_poly, f)

In [64]: (a_poly * b_inv_poly) % f
Out[64]: Poly(2x, GF(3))

In [65]: a * b**-1
Out[65]: GF(2, order=3^2)

In [66]: a / b
Out[66]: GF(2, order=3^2)
```

Power

```
In [67]: _, b_inv_poly, _ = galois.egcd(b_poly, f)

In [68]: (a_poly * b_inv_poly) % f
Out[68]: Poly(2x, GF(3))

In [69]: a * b**-1
Out[69]: GF(^5, order=3^2)

In [70]: a / b
Out[70]: GF(^5, order=3^2)
```

Here is the division table for completeness. Notice that division is not defined for $y = 0$.

Integer

```
In [71]: print(GF9.arithmetic_table("/"))
x / y | 1  2  3  4  5  6  7  8
------|------------------------
    0 | 0  0  0  0  0  0  0  0
    1 | 1  2  5  8  3  7  6  4
    2 | 2  1  7  4  6  5  3  8
    3 | 3  6  1  5  4  2  8  7
    4 | 4  8  3  1  7  6  5  2
    5 | 5  7  8  6  1  4  2  3
    6 | 6  3  2  7  8  1  4  5
    7 | 7  5  4  3  2  8  1  6
    8 | 8  4  6  2  5  3  7  1
```

Polynomial

```
In [72]: print(GF9.arithmetic_table("/"))
 x / y |      1       2           + 1    + 2      2  2 + 1  2 + 2
-------|----------------------------------------------------------------
     0 |      0       0       0       0       0       0       0       0
     1 |      1       2    + 2  2 + 2         2 + 1      2    + 1
     2 |      2       1  2 + 1    + 1      2    + 2         2 + 2
      |            2       1    + 2    + 1       2  2 + 2  2 + 1
  + 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
```

Power

```
In [73]: print(GF9.arithmetic_table("/"))
x / y |   1      ^2  ^3  ^4  ^5  ^6  ^7
------|----------------------------------------
    0 |   0    0    0    0    0    0    0    0
    1 |   1  ^7  ^6  ^5  ^4  ^3  ^2
     |       1  ^7  ^6  ^5  ^4  ^3  ^2
  ^2 | ^2        1  ^7  ^6  ^5  ^4  ^3
  ^3 | ^3  ^2        1  ^7  ^6  ^5  ^4
  ^4 | ^4  ^3  ^2        1  ^7  ^6  ^5
  ^5 | ^5  ^4  ^3  ^2        1  ^7  ^6
  ^6 | ^6  ^5  ^4  ^3  ^2        1  ^7
  ^7 | ^7  ^6  ^5  ^4  ^3  ^2        1
```

3.10.5 Primitive elements

A property of finite fields is that some elements produce the non-zero elements of the field by their powers.

A *primitive element* g of $\mathrm{GF}(p^m)$ is an element such that $\mathrm{GF}(p^m) = \{0, 1, g, g^2, \dots, g^{p^m-2}\}$. The non-zero elements $\{1, g, g^2, \dots, g^{p^m-2}\}$ form the cyclic multiplicative group $\mathrm{GF}(p^m)^\times$. A primitive element has multiplicative order $\mathrm{ord}(g) = p^m - 1$.

A primitive element

In `galois`, a primitive element of a finite field is provided by the `primitive_element` class property.

Integer

```
In [74]: print(GF9.properties)
Galois Field:
  name: GF(3^2)
  characteristic: 3
  degree: 2
  order: 9
  irreducible_poly: x^2 + 2x + 2
  is_primitive_poly: True
  primitive_element: x

In [75]: g = GF9.primitive_element; g
Out[75]: GF(3, order=3^2)
```

Polynomial

```
In [76]: print(GF9.properties)
Galois Field:
  name: GF(3^2)
  characteristic: 3
  degree: 2
  order: 9
  irreducible_poly: x^2 + 2x + 2
  is_primitive_poly: True
  primitive_element: x

In [77]: g = GF9.primitive_element; g
Out[77]: GF(, order=3^2)
```

Power

```
In [78]: print(GF9.properties)
Galois Field:
  name: GF(3^2)
  characteristic: 3
  degree: 2
  order: 9
  irreducible_poly: x^2 + 2x + 2
  is_primitive_poly: True
  primitive_element: x

In [79]: g = GF9.primitive_element; g
Out[79]: GF(, order=3^2)
```

The `galois` package allows you to easily display all powers of an element and their equivalent polynomial, vector, and integer representations using `repr_table()`.

Here is the representation table using the default generator $g = x$. Notice its multiplicative order is $p^m - 1$.

```
In [80]: g.multiplicative_order()
Out[80]: 8

In [81]: 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
  x^5       2x        [2, 0]      6
  x^6     2x + 2      [2, 2]      8
  x^7      x + 2      [1, 2]      5
```

Other primitive elements

There are multiple primitive elements of any finite field. All primitive elements are provided in the *primitive_elements* class property.

Integer

```
In [82]: GF9.primitive_elements
Out[82]: GF([3, 5, 6, 7], order=3^2)

In [83]: g = GF9("2x + 1"); g
Out[83]: GF(7, order=3^2)
```

Polynomial

```
In [84]: GF9.primitive_elements
Out[84]: GF([     ,    + 2,      2, 2 + 1], order=3^2)

In [85]: g = GF9("2x + 1"); g
Out[85]: GF(2 + 1, order=3^2)
```

Power

```
In [86]: GF9.primitive_elements
Out[86]: GF([  , ^7, ^5, ^3], order=3^2)

In [87]: g = GF9("2x + 1"); g
Out[87]: GF(^3, order=3^2)
```

This means that x, $x + 2$, $2x$, and $2x + 1$ all generate the 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 a different generator $g = 2x + 1$. Notice it also has multiplicative order $p^m - 1$.

```
In [88]: g.multiplicative_order()
Out[88]: 8

In [89]: print(GF9.repr_table(g))
   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
```

Non-primitive elements

All other elements of the field cannot generate the multiplicative group. They have multiplicative orders less than $p^m - 1$.

For example, the element $e = x + 1$ is not a primitive element. It has $\mathrm{ord}(e) = 4$. Notice elements x, $x + 2$, $2x$, and $2x + 1$ are not represented by the powers of e.

Integer

```
In [90]: e = GF9("x + 1"); e
Out[90]: GF(4, order=3^2)
```

Polynomial

```
In [91]: e = GF9("x + 1"); e
Out[91]: GF( + 1, order=3^2)
```

Power

```
In [92]: e = GF9("x + 1"); e
Out[92]: GF(^2, order=3^2)
```

```
In [93]: e.multiplicative_order()
Out[93]: 4

In [94]: print(GF9.repr_table(e))
   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
```

3.11 Prime Fields

This page compares the performance of *galois* to native NumPy when performing finite field multiplication in $\mathrm{GF}(p)$. Native NumPy can perform finite field multiplication in $\mathrm{GF}(p)$ because prime fields are very simple. Multiplication is simply xy mod p.

3.11.1 Lookup table performance

This section tests *galois* when using the "jit-lookup" compilation mode. For finite fields with order less than or equal to 2^{20}, *galois* uses lookup tables by default for efficient arithmetic.

Below are examples computing 10 million multiplications in the prime field $\mathrm{GF}(31)$.

```
In [1]: import galois

In [2]: GF = galois.GF(31)

In [3]: GF.ufunc_mode
Out[3]: 'jit-lookup'

In [4]: a = GF.Random(10_000_000, seed=1, dtype=int)

In [5]: b = GF.Random(10_000_000, seed=2, dtype=int)

# Invoke the ufunc once to JIT compile it, if necessary
In [6]: a * b
Out[6]: GF([ 9, 27,  7, ..., 14, 21, 15], order=31)

In [7]: %timeit a * b
36 ms ± 1.07 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

The equivalent operation using native NumPy ufuncs is ~1.8x slower.

```
In [8]: import numpy as np

In [9]: aa, bb = a.view(np.ndarray), b.view(np.ndarray)

In [10]: %timeit (aa * bb) % GF.order
65.3 ms ± 1.26 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

3.11.2 Explicit calculation performance

This section tests `galois` when using the `"jit-calculate"` compilation mode. For finite fields with order greater than 2^{20}, `galois` will use explicit arithmetic calculation by default rather than lookup tables. *Even in these cases*, `galois` is faster than NumPy!

Below are examples computing 10 million multiplications in the prime field GF(2097169).

```
In [1]: import galois

In [2]: GF = galois.GF(2097169)

In [3]: GF.ufunc_mode
Out[3]: 'jit-calculate'

In [4]: a = GF.Random(10_000_000, seed=1, dtype=int)

In [5]: b = GF.Random(10_000_000, seed=2, dtype=int)

# Invoke the ufunc once to JIT compile it, if necessary
In [6]: a * b
Out[6]: GF([1879104, 1566761,  967164, ...,  744769,  975853, 1142138], order=2097169)

In [7]: %timeit a * b
32.7 ms ± 1.44 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

The equivalent operation using native NumPy ufuncs is ~2.5x slower.

```
In [8]: import numpy as np

In [9]: aa, bb = a.view(np.ndarray), b.view(np.ndarray)

In [10]: %timeit (aa * bb) % GF.order
78.8 ms ± 1.6 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

3.11.3 Runtime floor

The `galois` ufunc runtime has a floor, however. This is due to a requirement of the ufuncs to `.view()` the output array and convert its dtype with `.astype()`. Also the `galois` ufuncs must perform input verification that NumPy ufuncs don't.

For example, for small array sizes, NumPy is faster than `galois`. This is true whether using lookup tables or explicit calculation.

```
In [1]: import galois

In [2]: GF = galois.GF(2097169)

In [3]: GF.ufunc_mode
Out[3]: 'jit-calculate'

In [4]: a = GF.Random(10, seed=1, dtype=int)
```

(continues on next page)

(continued from previous page)

```
In [5]: b = GF.Random(10, seed=2, dtype=int)

# Invoke the ufunc once to JIT compile it, if necessary
In [6]: a * b
Out[6]:
GF([1879104, 1566761,  967164, 1403108,  100593,  595358,  852783,
    1035698, 1207498,  989189], order=2097169)

In [7]: %timeit a * b
7.62 µs ± 390 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

The equivalent operation using native NumPy ufuncs is ~6x faster. However, in absolute terms, the difference is only ~6 µs.

```
In [8]: import numpy as np

In [9]: aa, bb = a.view(np.ndarray), b.view(np.ndarray)

In [10]: %timeit (aa * bb) % GF.order
1.29 µs ± 12.6 ns per loop (mean ± std. dev. of 7 runs, 1,000,000 loops each)
```

3.11.4 Linear algebra performance

Linear algebra performance in prime fields is comparable to the native NumPy implementations, which use BLAS/LAPACK. This is because *galois* uses the native NumPy ufuncs when possible.

If overflow is prevented, dot products in $\mathrm{GF}(p)$ can be computed by first computing the dot product in $\mathbb{Z}$ and then reducing modulo p. In this way, the efficient BLAS/LAPACK implementations are used to keep finite field linear algebra fast, whenever possible.

Below are examples computing the matrix multiplication of two 100×100 matrices in the prime field $\mathrm{GF}(2097169)$.

```
In [1]: import galois

In [2]: GF = galois.GF(2097169)

In [3]: GF.ufunc_mode
Out[3]: 'jit-calculate'

In [4]: A = GF.Random((100,100), seed=1, dtype=int)

In [5]: B = GF.Random((100,100), seed=2, dtype=int)

# Invoke the ufunc once to JIT compile it, if necessary
In [6]: A @ B
Out[6]:
GF([[1147163,   59466, 1841183, ...,  667877, 2084618,  799166],
    [ 306714, 1380503,  810935, ..., 1932687, 1690697,  329837],
    [ 325274,  575543, 1327001, ...,  167724,  422518,  696986],
    ...,
    [ 862992, 1143160,  588384, ...,  668891, 1285421, 1196448],
    [1026856, 1413416, 1844802, ...,   38844, 1643604,   10409],
```

(continues on next page)

(continued from previous page)

```
    [ 401717,  329673,  860449, ..., 1551173, 1766877,  986430]],
order=2097169)

In [7]: %timeit A @ B
708 µs ± 1.48 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

The equivalent operation using native NumPy ufuncs is slightly faster. This is because *galois* has some internal overhead before invoking the same NumPy calculation.

```
In [8]: import numpy as np

In [9]: AA, BB = A.view(np.ndarray), B.view(np.ndarray)

In [10]: %timeit (AA @ BB) % GF.order
682 µs ± 11.1 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

3.12 Binary Extension Fields

This page compares the performance of *galois* performing finite field multiplication in $\mathrm{GF}(2^m)$ with native NumPy performing *only* modular multiplication.

Native NumPy cannot easily perform finite field multiplication in $\mathrm{GF}(2^m)$ because it involves polynomial multiplication (convolution) followed by reducing modulo the irreducible polynomial. To make a *similar* comparison, NumPy will perform integer multiplication followed by integer remainder division.

Important

Native NumPy is not computing the correct result! This is not a fair fight!

These are *not* fair comparisons because NumPy is not computing the correct product. However, they are included here to provide a performance reference point with native NumPy.

3.12.1 Lookup table performance

This section tests *galois* when using the `"jit-lookup"` compilation mode. For finite fields with order less than or equal to 2^{20}, *galois* uses lookup tables by default for efficient arithmetic.

Below are examples computing 10 million multiplications in the binary extension field $\mathrm{GF}(2^8)$.

```
In [1]: import galois

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

In [3]: GF.ufunc_mode
Out[3]: 'jit-lookup'

In [4]: a = GF.Random(10_000_000, seed=1, dtype=int)

In [5]: b = GF.Random(10_000_000, seed=2, dtype=int)
```

(continues on next page)

(continued from previous page)

```
# Invoke the ufunc once to JIT compile it, if necessary
In [6]: a * b
Out[6]: GF([181,  92, 148, ..., 255, 220, 153], order=2^8)

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

NumPy, even when computing the incorrect result, is ~1.9x slower than `galois`. This is because `galois` is using lookup tables instead of explicitly performing the polynomial multiplication and division.

```
In [8]: import numpy as np

In [9]: aa, bb = a.view(np.ndarray), b.view(np.ndarray)

In [10]: pp = int(GF.irreducible_poly)

# This does not produce the correct result!
In [11]: %timeit (aa * bb) % pp
64 ms ± 747 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

3.12.2 Explicit calculation performance

This section tests `galois` when using the `"jit-calculate"` compilation mode. For finite fields with order greater than 2^{20}, `galois` will use explicit arithmetic calculation by default rather than lookup tables.

Below are examples computing 10 million multiplications in the binary extension field $\mathrm{GF}(2^{32})$.

```
In [1]: import galois

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

In [3]: GF.ufunc_mode
Out[3]: 'jit-calculate'

In [4]: a = GF.Random(10_000_000, seed=1, dtype=int)

In [5]: b = GF.Random(10_000_000, seed=2, dtype=int)

# Invoke the ufunc once to JIT compile it, if necessary
In [6]: a * b
Out[6]:
GF([1174047800, 3249326965, 3196014003, ..., 3195457330,  100242821,
     338589759], order=2^32)

In [7]: %timeit a * b
386 ms ± 14 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

The `galois` library when using explicit calculation is only ~3.9x slower than native NumPy, which isn't even computing the correct product.

```
In [8]: import numpy as np

```

(continues on next page)

(continued from previous page)

```
In [9]: aa, bb = a.view(np.ndarray), b.view(np.ndarray)

In [10]: pp = int(GF.irreducible_poly)

# This does not produce the correct result!
In [11]: %timeit (aa * bb) % pp
100 ms ± 718 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

3.12.3 Linear algebra performance

Linear algebra performance in extension fields is definitely slower than native NumPy. This is because, unlike with prime fields, it is not possible to use the BLAS/LAPACK implementations. Instead, entirely new JIT-compiled ufuncs are generated, which are not as optimized for parallelism or hardware acceleration as BLAS/LAPACK.

Below are examples computing the matrix multiplication of two 100×100 matrices in the binary extension field $\mathrm{GF}(2^{32})$.

```
In [1]: import galois

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

In [3]: GF.ufunc_mode
Out[3]: 'jit-calculate'

In [4]: A = GF.Random((100,100), seed=1, dtype=int)

In [5]: B = GF.Random((100,100), seed=2, dtype=int)

# Invoke the ufunc once to JIT compile it, if necessary
In [6]: A @ B
Out[6]:
GF([[4203877556, 3977035749, 2623937858, ..., 3721257849, 4250999056,
    4026271867],
    [3120760606, 1017695431, 1111117124, ..., 1638387264, 2988805996,
    1734614583],
    [2508826906, 2800993411, 1720697782, ..., 3858180318, 2521070820,
    3906771227],
    ...,
    [ 624580545,  984724090, 3969931498, ..., 1692192269,  473079794,
    1029376699],
    [1232183301,  209395954, 2659712274, ..., 2967695343, 2747874320,
    1249453570],
    [3938433735,  828783569, 3286222384, ..., 3669775257,   33626526,
    4278384359]], order=2^32)

In [7]: %timeit A @ B
3.88 ms ± 102 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

The *galois* library is about ~5.5x slower than native NumPy (which isn't computing the correct product).

```
In [8]: import numpy as np
```

(continues on next page)

(continued from previous page)

```
In [9]: AA, BB = A.view(np.ndarray), B.view(np.ndarray)

In [10]: pp = int(GF.irreducible_poly)

# This does not produce the correct result!
In [11]: %timeit (AA @ BB) % pp
703 μs ± 1.9 μs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

3.13 Benchmarks

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

3.13.1 Install dependencies

First, `pytest` and `pytest-benchmark` must be installed on your system. Easily install them by installing the development dependencies.

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

3.13.2 Create a benchmark

To create 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`.

```
$ python3 -m 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-4.6.9, py-1.8.1, pluggy-0.13.0
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: /mnt/c/Users/matth/repos/galois, inifile: setup.cfg
plugins: requests-mock-1.9.3, cov-3.0.0, benchmark-3.4.1, typeguard-2.13.3, anyio-3.5.0
collected 56 items

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

-------------------- 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                         16.3810 (1.21)       218.3280 (1.22)       18.9455 (1.
```

(continues on next page)

(continued from previous page)

```
↪17)      5.4959 (1.07)      17.3620 (1.22)
test_additive_inverse             13.5850 (1.0)       206.5360 (1.15)      16.1445 (1.0)␣
↪       5.9249 (1.16)      14.2670 (1.0)
test_divide                      132.0870 (9.72)      191.0680 (1.07)     149.6357 (9.
↪27)      16.9537 (3.31)     145.4920 (10.20)
test_multiplicative_inverse       91.4410 (6.73)      179.0050 (1.0)      102.6590 (6.
↪36)      18.8467 (3.68)      94.4670 (6.62)
test_multiply                     16.0400 (1.18)      229.4400 (1.28)      18.3296 (1.
↪14)       5.1267 (1.0)       16.9010 (1.18)
test_power                       150.2410 (11.06)     212.2870 (1.19)     168.8103 (10.
↪46)     16.4850 (3.22)     166.2860 (11.66)
test_scalar_multiply             543.3970 (40.00)     714.2870 (3.99)     562.2968 (34.
↪83)     12.4125 (2.42)     559.1370 (39.19)
test_subtract                     16.3110 (1.20)    2,233.8710 (12.48)     19.2938 (1.
↪20)      23.4038 (4.57)      17.2520 (1.21)
-----------------------------------------------------------------------------------------
↪-----------------------------------------

---------------------- 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                            78.2860 (1.37)      311.2620 (1.40)        87.9984␣
↪(1.26)      12.1680 (1.04)         81.7530 (1.36)
test_additive_inverse               57.0860 (1.0)       281.9070 (1.27)        69.7403␣
↪(1.0)       17.0927 (1.47)         60.0520 (1.0)
test_divide                      3,274.0860 (57.35)   3,351.6220 (15.09)    3,309.5920␣
↪(47.46)     27.4510 (2.36)      3,307.3240 (55.07)
test_multiplicative_inverse      3,245.1620 (56.85)   4,295.9590 (19.34)    3,350.8016␣
↪(48.05)     96.3332 (8.26)      3,321.1050 (55.30)
test_multiply                      197.1090 (3.45)      305.5620 (1.38)       218.1805␣
↪(3.13)      20.9767 (1.80)        213.6600 (3.56)
test_power                       3,270.7210 (57.29)   3,520.5480 (15.85)    3,349.1942␣
↪(48.02)     91.3962 (7.84)      3,329.6105 (55.45)
test_scalar_multiply               544.0880 (9.53)    1,182.1140 (5.32)       575.6227␣
↪(8.25)      42.0059 (3.60)        562.4830 (9.37)
test_subtract                       77.6160 (1.36)      222.1760 (1.0)         88.3242␣
↪(1.27)      11.6562 (1.0)          82.8905 (1.38)
-----------------------------------------------------------------------------------------
↪-----------------------------------------------

-------------------- 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                          79.0580 (1.37)     393.6670 (2.39)      86.7954 (1.26) ␣
↪     12.6945 (1.0)       81.4630 (1.34)
test_additive_inverse             57.9080 (1.0)      164.6380 (1.0)       69.0218 (1.0)  ␣
```

(continues on next page)

(continued from previous page)

```
↪   21.7213 (1.71)      60.6330 (1.0)
test_divide                       228.7890 (3.95)     280.8050 (1.71)     243.1431 (3.52) ␣
↪   16.6688 (1.31)     241.0210 (3.98)
test_multiplicative_inverse       263.8140 (4.56)     348.4620 (2.12)     290.6663 (4.21) ␣
↪   20.8113 (1.64)     284.3620 (4.69)
test_multiply                     193.5820 (3.34)     475.2490 (2.89)     216.4317 (3.14) ␣
↪   24.6557 (1.94)     212.2370 (3.50)
test_power                        311.6030 (5.38)     389.2180 (2.36)     328.9333 (4.77) ␣
↪   18.9217 (1.49)     326.1145 (5.38)
test_scalar_multiply              539.7710 (9.32)     973.1410 (5.91)     573.4538 (8.31) ␣
↪   49.0047 (3.86)     557.7030 (9.20)
test_subtract                      80.3500 (1.39)     270.0450 (1.64)      97.6062 (1.41) ␣
↪   37.3127 (2.94)      89.1270 (1.47)
-----------------------------------------------------------------------------------------
↪----------------------------------------

------------------------ 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                              16.6110 (1.21)        218.1990 (1.09)         19.
↪4288 (1.21)       5.8745 (1.11)          17.4830 (1.22)
test_additive_inverse                 13.6750 (1.0)         200.7150 (1.0)          16.
↪0465 (1.0)        5.2959 (1.0)           14.3070 (1.0)
test_divide                       13,280.4310 (971.15)   13,367.6440 (66.60)    13,340.
↪0968 (831.34)    36.5738 (6.91)      13,354.6500 (933.43)
test_multiplicative_inverse       11,842.1600 (865.97)   15,404.4870 (76.75)    12,129.
↪1417 (755.88)   529.9702 (100.07)    12,015.3740 (839.82)
test_multiply                      1,079.0300 (78.91)     1,137.0780 (5.67)      1,098.
↪1473 (68.44)     18.6741 (3.53)       1,092.5140 (76.36)
test_power                        12,832.8340 (938.41)   13,115.7640 (65.35)    12,942.
↪1951 (806.54)    92.9381 (17.55)     12,928.9640 (903.68)
test_scalar_multiply                 883.2930 (64.59)     1,192.1310 (5.94)        928.
↪3991 (57.86)     44.9582 (8.49)         912.0860 (63.75)
test_subtract                         16.6210 (1.22)      1,334.7780 (6.65)         19.
↪7528 (1.23)      15.2536 (2.88)          17.4330 (1.22)
-----------------------------------------------------------------------------------------
↪-------------------------------------------------------

--------------------- 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                           16.0900 (1.23)       277.5990 (3.89)       18.8739 (1.
↪24)       5.6347 (1.35)       17.1720 (1.24)
test_additive_inverse              13.1050 (1.0)         71.3340 (1.0)        15.1649 (1.0)␣
↪       4.1860 (1.0)       13.8860 (1.0)
test_divide                       215.6730 (16.46)      271.6490 (3.81)      233.7595 (15.
```

(continues on next page)

(continued from previous page)

```
↪41)    16.0094 (3.82)     229.9500 (16.56)
test_multiplicative_inverse      152.3150 (11.62)      207.4480 (2.91)     167.0589 (11.
↪02)    12.9483 (3.09)     166.4220 (11.98)
test_multiply                    199.3430 (15.21)      250.2580 (3.51)     220.8079 (14.
↪56)    17.1620 (4.10)     216.1740 (15.57)
test_power                       331.7910 (25.32)      401.3410 (5.63)     348.8168 (23.
↪00)    17.4759 (4.17)     348.8730 (25.12)
test_scalar_multiply             850.2810 (64.88)    1,128.3010 (15.82)    884.6499 (58.
↪34)    29.6705 (7.09)     876.5800 (63.13)
test_subtract                     16.0400 (1.22)       83.5460 (1.17)      18.2685 (1.
↪20)      4.4904 (1.07)      16.8610 (1.21)
------------------------------------------------------------------------------------------
↪------------------------------------------

--------------------- benchmark "GF(3^5) Array Arithmetic: shape=(100_000,), ufunc_mode=
↪'jit-lookup'": 8 tests --------------------
Name (time in us)                     Min                    Max                  Mean      ␣
↪       StdDev              Median
------------------------------------------------------------------------------------------
↪------------------------------------------
test_add                         313.4770 (2.04)       358.2300 (1.58)     328.4561 (1.
↪85)     12.1327 (1.0)      326.4100 (1.87)
test_additive_inverse            153.6980 (1.0)        226.6550 (1.0)      177.9128 (1.0)␣
↪     19.6890 (1.62)     174.6160 (1.0)
test_divide                      222.3460 (1.45)       284.7130 (1.26)     235.5486 (1.
↪32)     15.5184 (1.28)     232.1795 (1.33)
test_multiplicative_inverse      165.4600 (1.08)       241.2010 (1.06)     186.5927 (1.
↪05)     23.5185 (1.94)     178.2130 (1.02)
test_multiply                    202.1690 (1.32)       327.1620 (1.44)     231.3098 (1.
↪30)     30.2870 (2.50)     219.7315 (1.26)
test_power                       361.5260 (2.35)       447.0060 (1.97)     385.7585 (2.
↪17)     28.6975 (2.37)     375.4475 (2.15)
test_scalar_multiply             756.5460 (4.92)     1,014.9590 (4.48)     792.1778 (4.
↪45)     29.3465 (2.42)     786.1765 (4.50)
test_subtract                    383.7790 (2.50)       461.3640 (2.04)     411.7450 (2.
↪31)     26.7056 (2.20)     403.7260 (2.31)
------------------------------------------------------------------------------------------
↪------------------------------------------

-------------------------- benchmark "GF(3^5) Array Arithmetic: shape=(10_000,), ufunc_
↪mode='jit-calculate'": 8 tests --------------------------
Name (time in us)                        Min                      Max                     ␣
↪Mean                 StdDev                Median
------------------------------------------------------------------------------------------
↪--------------------------------------------------------
test_add                            876.9310 (1.57)      1,635.8940 (1.52)        936.
↪2487 (1.48)       76.1260 (3.84)        915.1175 (1.58)
test_additive_inverse               557.6440 (1.0)       1,945.0700 (1.81)        632.
↪3527 (1.0)       257.9239 (13.01)       578.4425 (1.0)
test_divide                      90,022.6490 (161.43)   96,282.8560 (89.50)    92,257.
↪7516 (145.90)   2,808.8230 (141.69)   90,481.3870 (156.42)
test_multiplicative_inverse      82,011.9590 (147.07)   83,817.2670 (77.91)    82,897.
```

(continues on next page)

(continued from previous page)

```
↪2702 (131.09)     471.2330 (23.77)    82,992.5040 (143.48)
test_multiply                    6,847.6130 (12.28)     6,894.3920 (6.41)      6,872.
↪3102 (10.87)       19.8231 (1.0)       6,876.2980 (11.89)
test_power                     77,322.3730 (138.66)   78,040.5270 (72.54)    77,650.
↪6814 (122.80)     267.5041 (13.49)    77,693.8380 (134.32)
test_scalar_multiply             6,049.4100 (10.85)     7,260.1360 (6.75)      6,184.
↪4565 (9.78)       146.6458 (7.40)      6,153.1895 (10.64)
test_subtract                      888.4720 (1.59)      1,075.8030 (1.0)         944.
↪4420 (1.49)        47.1406 (2.38)        936.5830 (1.62)
-----------------------------------------------------------------------------------------
↪-------------------------------------------------------

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, 16 warnings in 19.
↪54 seconds ===========================================================
```

3.13.3 Compare with a previous benchmark

If you would like to compare the performance impact of a branch, first run a benchmark on `main` using the `--benchmark-save` option. This will save the file `.benchmarks/0001_master.json`.

```
$ git checkout main
$ python3 -m pytest benchmarks/test_field_arithmetic.py --benchmark-save=main --
↪benchmark-columns=min,max,mean,stddev,median --benchmark-sort=name
```

Next, run a benchmark on the branch under test while comparing against the benchmark from `main`.

```
$ git checkout branch
$ python3 -m pytest benchmarks/test_field_arithmetic.py --benchmark-compare=0001_master -
↪-benchmark-columns=min,max,mean,stddev,median --benchmark-sort=name
```

Or, save a benchmark run from `branch` and compare it explicitly against the one from `main`. This benchmark run will save the file `.benchmarks/0001_branch.json`.

```
$ git checkout branch
$ python3 -m pytest benchmarks/test_field_arithmetic.py --benchmark-save=branch --
↪benchmark-columns=min,max,mean,stddev,median --benchmark-sort=name
$ python3 -m pytest-benchmark compare 0001_master 0001_branch
```

3.14 Installation

3.14.1 Install from PyPI

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

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

3.14.2 Install from GitHub

The latest code on `main` can be installed using `pip` in this way.

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

Or from a specific branch.

```
$ python3 -m pip install git+https://github.com/mhostetter/galois.git@branch
```

3.14.3 Editable install from local folder

To actively develop the library, it is beneficial to `pip install` the library in an editable fashion from a local folder. This allows any changes in the current directory to be immediately seen upon the next `import galois`.

Clone the repo wherever you'd like.

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

Install the local folder using the `-e` or `--editable` flag.

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

3.14.4 Install the dev dependencies

The development dependencies include packages for linting and unit testing. These dependencies are stored in `requirements-dev.txt`.

Listing 1: requirements-dev.txt

```
ruff == 0.1.15
pytest
pytest-cov[toml]
pytest-xdist
pytest-benchmark >= 4.0.0
requests
pdfminer.six
```

Install the development dependencies by passing `-r` to `pip install`.

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

3.15 Formatting

The *galois* library uses Ruff for static analysis, linting, and code formatting.

3.15.1 Install

First, `ruff` needs to be installed on your system. Easily install it by installing the development dependencies.

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

3.15.2 Configuration

The `ruff` configuration is provided in `pyproject.toml`.

Listing 2: pyproject.toml

```
[tool.ruff]
src = ["src"]
extend-include = ["*.ipynb"]
extend-exclude = ["build", "dist", "docs", "src/galois/_version.py"]
line-length = 120

[tool.ruff.lint]
select = [
    "E",  # pycodestyle
    "F",  # Pyflakes
    "UP", # pyupgrade
    "B",  # flake8-bugbear
    # "SIM",# flake8-simplify
    "I",  # isort
    "PL", # pylint
]
ignore = [
    "E501",    # line-too-long
    "E713",    # not-in-test
    "E714",    # not-is-test
    "E741",    # ambiguous-variable-name
    "PLR0911", # too-many-return-statements
    "PLR0912", # too-many-branches
    "PLR0913", # too-many-arguments
    "PLR0915", # too-many-statements
    "PLR2004", # magic-value-comparison
    "PLR5501", # collapsible-else-if
    "PLW0603", # global-statement
    "PLW2901", # redefined-loop-name
]

[tool.ruff.lint.per-file-ignores]
"__init__.py" = ["F401", "F403"]
```

3.15.3 Run the linter

Run the Ruff linter manually from the command line.

```
$ python3 -m ruff check .
```

3.15.4 Run the formatter

Run the Ruff formatter manually from the command line.

```
$ python3 -m ruff format --check .
```

3.15.5 Pre-commit

A `pre-commit` configuration file with various hooks is provided in `.pre-commit-config.yaml`.

Listing 3: .pre-commit-config.yaml

```yaml
repos:
- repo: https://github.com/pre-commit/pre-commit-hooks
  rev: v2.3.0
  hooks:
  - id: check-added-large-files
  - id: check-yaml
  - id: check-toml
  - id: end-of-file-fixer
  - id: trailing-whitespace
- repo: https://github.com/astral-sh/ruff-pre-commit
  rev: v0.1.15
  hooks:
  - id: ruff
  - id: ruff-format
```

Enable `pre-commit` by installing the pre-commit hooks.

```
$ pre-commit install
```

Run `pre-commit` on all files.

```
$ pre-commit run --all-files
```

Disable `pre-commit` by uninstalling the pre-commit hooks.

```
$ pre-commit uninstall
```

3.15.6 Run from VS Code

Install the Ruff extension for VS Code. Included is a VS Code configuration file `.vscode/settings.json`. VS Code will run the linter and formatter as you view and edit files.

3.16 Unit Tests

The *galois* library uses pytest for unit testing.

3.16.1 Install

First, `pytest` needs to be installed on your system. Easily install it by installing the development dependencies.

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

3.16.2 Configuration

The `pytest` configuration is stored in `pyproject.toml`.

Listing 4: pyproject.toml

```
[tool.pytest.ini_options]
minversion = "6.2"
addopts = "-s --showlocals"
testpaths = ["tests"]
```

3.16.3 Run from the command line

Execute all of the unit tests manually from the command line.

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

Or only run a specific test file.

```
$ python3 -m pytest tests/test_math.py
```

Or only run a specific unit test.

```
$ python3 -m pytest tests/test_math.py::test_gcd
```

3.16.4 Run from VS Code

Included is a VS Code configuration file `.vscode/settings.json`. This instructs VS Code about how to invoke `pytest`. VS Code's integrated test infrastructure will locate the tests and allow you to run or debug any test.

3.16.5 Test vectors

Test vectors are generated by third-party tools and stored in `.pkl` files. Most test vectors are stored in these folders:

- `tests/data/`
- `tests/fields/data/GF(*)/`
- `tests/polys/data/GF(*)/`

The scripts that generate the test vectors are:

- `scripts/generate_int_test_vectors.py`
- `scripts/generate_field_test_vectors.py`

The two primary third-party tools are Sage and SymPy.

Install Sage

```
$ sudo apt install sagemath
```

Install SymPy

```
$ python3 -m pip install sympy
```

Generate test vectors

To re-generate the test vectors locally, run:

```
$ python3 scripts/generate_int_test_vectors.py
$ python3 scripts/generate_field_test_vectors.py
```

The scripts use random number generator seeds to generate reproducible test vectors. To generate *different* test vectors, modify the seeds. It's also easy to increase the number of test cases for any individual test.

3.17 Documentation

The `galois` documentation is generated with Sphinx and the Sphinx Immaterial theme.

3.17.1 Install

The documentation dependencies are stored in `docs/requirements.txt`.

Listing 5: docs/requirements.txt

```
sphinx < 7.3
sphinx-immaterial == 0.11.9
# For some reason, if the below versions aren't specified, the dependency resolution␣
↪takes 18 minutes in CI.
myst-parser == 0.18.1
sphinx-design  == 0.5.0
sphinx-last-updated-by-git == 0.3.6
sphinx-math-dollar == 1.2.1
ipykernel == 6.26.0
myst-nb == 0.17.2
pickleshare == 0.7.5  # Needed by ipykernel
```

Install the documentation dependencies by passing the `-r` switch to `pip install`.

```
$ python3 -m pip install -r docs/requirements.txt
```

3.17.2 Build the docs

The documentation is built by running the `sphinx-build` command.

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

The HTML output is located in `docs/build/`. The home page is `docs/build/index.html`.

3.17.3 Serve the docs

Since the site is built to use directories (`*/getting-started/` not `*/getting-started.html`), it is necessary to serve the webpages locally with a webserver. This is easily done using the built-in Python `http` module.

```
$ python3 -m http.server 8080 -d docs/build/
```

The documentation is accessible from a web browser at `http://localhost:8080/`.

3.18 Arrays

class `galois.`***Array***(numpy.ndarray)

An abstract `ndarray` subclass over a Galois field or Galois ring.

`galois.typing.`***ArrayLike***

A `Union` representing objects that can be coerced into a Galois field array.

`galois.typing.`***DTypeLike***

A `Union` representing objects that can be coerced into a NumPy data type.

`galois.typing.`***`ElementLike`***

A `Union` representing objects that can be coerced into a Galois field element.

`galois.typing.`***`IterableLike`***

A `Union` representing iterable objects that can be coerced into a Galois field array.

`galois.typing.`***`ShapeLike`***

A `Union` representing objects that can be coerced into a NumPy `shape` tuple.

`class` `galois.`**`Array`**(numpy.ndarray)

An abstract `ndarray` subclass over a Galois field or Galois ring.

Abstract

`Array` is an abstract base class for *`FieldArray`* and cannot be instantiated directly.

Constructors

`classmethod` ***`Identity`***(size: *int*, ...) → Self

Creates an $n \times n$ identity matrix.

`classmethod` ***`Ones`***(shape: ShapeLike, ...) → Self

Creates an array of all ones.

`classmethod` ***`Random`***(shape: ShapeLike = (), ...) → Self

Creates an array with random elements.

`classmethod` ***`Range`***(start: ElementLike, stop, ...) → Self

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

`classmethod` ***`Zeros`***(shape: ShapeLike, ...) → Self

Creates an array of all zeros.

`classmethod` `galois.Array.`**`Identity`**(size: *int*, dtype: DTypeLike | *None* = ***None***) → Self

Creates an $n \times n$ identity matrix.

Parameters

size: int
The size n along one dimension of the identity matrix.

dtype: *DTypeLike* | None = None
The `numpy.dtype` of the array elements. The default is **`None`** which represents the smallest unsigned data type for this *`Array`* subclass (the first element in *`dtypes`*).

Returns
A 2-D identity matrix with shape `(size, size)`.

`classmethod` `galois.Array.`**`Ones`**(shape: ShapeLike, dtype: DTypeLike | *None* = ***None***) → Self

Creates an array of all ones.

Parameters

shape: *ShapeLike*
A NumPy-compliant `shape` tuple.

dtype: *DTypeLike* | None = None
The `numpy.dtype` of the array elements. The default is `None` which represents the smallest unsigned data type for this *Array* subclass (the first element in *dtypes*).

Returns
An array of ones.

classmethod galois.Array.Random(*shape: ShapeLike = ()*, *low: ElementLike = 0*, *high: ElementLike | None = None*, *seed: int | Generator | None = None*, *dtype: DTypeLike | None = None*) → Self

Creates an array with random elements.

Parameters

shape: *ShapeLike* = ()
A NumPy-compliant `shape` tuple. The default is `()` which represents a scalar.

low: *ElementLike* = 0
The smallest element (inclusive). The default is 0.

high: *ElementLike* | None = None
The largest element (exclusive). The default is `None` which represents *order*.

seed: int | Generator | None = None
Non-negative integer used to initialize the PRNG. The default is `None` which means that unpredictable entropy will be pulled from the OS to be used as the seed. A `numpy.random.Generator` can also be passed.

dtype: *DTypeLike* | None = None
The `numpy.dtype` of the array elements. The default is `None` which represents the smallest unsigned data type for this *Array* subclass (the first element in *dtypes*).

Returns
An array of random elements.

classmethod galois.Array.Range(*start: ElementLike*, *stop: ElementLike*, *step: int = 1*, *dtype: DTypeLike | None = None*) → Self

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

Parameters

start: *ElementLike*
The starting element (inclusive).

stop: *ElementLike*
The stopping element (exclusive).

step: int = 1
The increment between elements. The default is 1.

dtype: *DTypeLike* | None = None
The `numpy.dtype` of the array elements. The default is `None` which represents the smallest unsigned data type for this *Array* subclass (the first element in *dtypes*).

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

classmethod galois.Array.Zeros(*shape: ShapeLike*, *dtype: DTypeLike | None = None*) → Self

Creates an array of all zeros.

Parameters

shape: ***ShapeLike***

A NumPy-compliant `shape` tuple.

dtype: ***DTypeLike*** **| None = None**

The `numpy.dtype` of the array elements. The default is `None` which represents the smallest unsigned data type for this *Array* subclass (the first element in *dtypes*).

Returns

An array of zeros.

Methods

`classmethod` ***compile***(mode)

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

`classmethod` ***repr***(...) → Generator[None, None, None]

Sets the element representation for all arrays from this *FieldArray* subclass.

`classmethod galois.Array.compile`(mode: *Literal[auto] | typing.Literal[jit - lookup] | typing.Literal[jit - calculate] | typing.Literal[python - calculate]*)

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

This function updates *ufunc_mode*.

Parameters

mode: Literal[auto] | typing.Literal[jit - lookup] | typing.Literal[jit - calculate] | typing.Literal[python - calculate]

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).

`classmethod galois.Array.repr`(element_repr: *Literal[int] | Literal[poly] | Literal[power] = 'int'*) → Generator[None, None, None]

Sets the element representation for all arrays from this *FieldArray* subclass.

Parameters

element_repr: Literal[int] | Literal[poly] | Literal[power] = 'int'

The field element representation.

- `"int"` (default): Sets the representation to the *integer representation*.
- `"poly"`: Sets the representation to the *polynomial representation*.

- `"power"`: Sets the representation to the *power representation*.

Slower performance

To display elements in the power representation, `galois` must compute the discrete logarithm of each element displayed. For large fields or fields using *explicit calculation*, this process can take a while. However, when using *lookup tables* this representation is just as fast as the others.

Returns

A context manager for use in a `with` statement. If permanently setting the element representation, disregard the return value.

Notes

This function updates *element_repr*.

Properties

class property ***characteristic*** : int

The characteristic p of the Galois field $\mathrm{GF}(p^m)$ or p^e of the Galois ring $\mathrm{GR}(p^e, m)$.

class property ***default_ufunc_mode*** : Literal[jit - lookup] | typing.Literal[jit - calculate] | typing.Literal[python - calculate]

The default compilation mode of the Galois field or Galois ring.

class property ***degree*** : int

The degree m of the Galois field $\mathrm{GF}(p^m)$ or Galois ring $\mathrm{GR}(p^e, m)$.

class property ***dtypes*** : list[np.dtype]

List of valid integer `numpy.dtype` values that are compatible with this Galois field or Galois ring.

class property ***element_repr*** : Literal[int] | Literal[poly] | Literal[power]

The current element representation of the Galois field or Galois ring.

class property ***elements*** : *Array*

All elements of the Galois field or Galois ring.

class property ***irreducible_poly*** : *Poly*

The irreducible polynomial of the Galois field or Galois ring.

class property ***name*** : str

The name of the Galois field or Galois ring.

class property ***order*** : int

The order p^m of the Galois field $\mathrm{GF}(p^m)$ or p^{em} of the Galois ring $\mathrm{GR}(p^e, m)$.

class property ***primitive_element*** : *Array*

A primitive element of the Galois field or Galois ring.

class property ***ufunc_mode*** : Literal[jit - lookup] | typing.Literal[jit - calculate] | typing.Literal[python - calculate]

The current compilation mode of the Galois field or Galois ring.

class property *ufunc_modes* : list[str]

All supported compilation modes of the Galois field or Galois ring.

class property *units* : *Array*

All units of the Galois field or Galois ring.

class property galois.Array.**characteristic** : int

The characteristic p of the Galois field $\mathrm{GF}(p^m)$ or p^e of the Galois ring $\mathrm{GR}(p^e, m)$.

class property galois.Array.**default_ufunc_mode** : Literal[jit - lookup] | typing.Literal[jit - calculate] | typing.Literal[python - calculate]

The default compilation mode of the Galois field or Galois ring.

class property galois.Array.**degree** : int

The degree m of the Galois field $\mathrm{GF}(p^m)$ or Galois ring $\mathrm{GR}(p^e, m)$.

class property galois.Array.**dtypes** : list[np.dtype]

List of valid integer `numpy.dtype` values that are compatible with this Galois field or Galois ring.

class property galois.Array.**element_repr** : Literal[int] | Literal[poly] | Literal[power]

The current element representation of the Galois field or Galois ring.

class property galois.Array.**elements** : *Array*

All elements of the Galois field or Galois ring.

class property galois.Array.**irreducible_poly** : *Poly*

The irreducible polynomial of the Galois field or Galois ring.

class property galois.Array.**name** : str

The name of the Galois field or Galois ring.

class property galois.Array.**order** : int

The order p^m of the Galois field $\mathrm{GF}(p^m)$ or p^{em} of the Galois ring $\mathrm{GR}(p^e, m)$.

class property galois.Array.**primitive_element** : *Array*

A primitive element of the Galois field or Galois ring.

class property galois.Array.**ufunc_mode** : Literal[jit - lookup] | typing.Literal[jit - calculate] | typing.Literal[python - calculate]

The current compilation mode of the Galois field or Galois ring.

class property galois.Array.**ufunc_modes** : list[str]

All supported compilation modes of the Galois field or Galois ring.

class property galois.Array.**units** : *Array*

All units of the Galois field or Galois ring.

galois.typing.**ArrayLike**

A `Union` representing objects that can be coerced into a Galois field array.

Union

- *IterableLike*: A recursive iterable of iterables of elements.

Integer

```
In [1]: GF = galois.GF(3**5)

In [2]: GF([[17, 4], [148, 205]])
Out[2]:
GF([[ 17,   4],
    [148, 205]], order=3^5)

# Mix and match integers and strings
In [3]: GF([["x^2 + 2x + 2", 4], ["x^4 + 2x^3 + x^2 + x + 1", 205]])
Out[3]:
GF([[ 17,   4],
    [148, 205]], order=3^5)
```

Polynomial

```
In [4]: GF = galois.GF(3**5, repr="poly")

In [5]: GF([[17, 4], [148, 205]])
Out[5]:
GF([[             ^2 + 2 + 2,                       + 1],
    [ ^4 + 2^3 + ^2 +  + 1, 2^4 + ^3 + ^2 + 2 + 1]], order=3^5)

# Mix and match integers and strings
In [6]: GF([["x^2 + 2x + 2", 4], ["x^4 + 2x^3 + x^2 + x + 1", 205]])
Out[6]:
GF([[             ^2 + 2 + 2,                       + 1],
    [ ^4 + 2^3 + ^2 +  + 1, 2^4 + ^3 + ^2 + 2 + 1]], order=3^5)
```

Power

```
In [7]: GF = galois.GF(3**5, repr="power")

In [8]: GF([[17, 4], [148, 205]])
Out[8]:
GF([[^222,  ^69],
    [ ^54,  ^24]], order=3^5)

# Mix and match integers and strings
In [9]: GF([["x^2 + 2x + 2", 4], ["x^4 + 2x^3 + x^2 + x + 1", 205]])
Out[9]:
GF([[^222,  ^69],
    [ ^54,  ^24]], order=3^5)
```

- `ndarray`: A NumPy array of integers, representing finite field elements in their *integer representation*.

Integer

```
In [10]: x = np.array([[17, 4], [148, 205]]); x
Out[10]:
array([[ 17,   4],
       [148, 205]])

In [11]: GF(x)
Out[11]:
GF([[ 17,   4],
    [148, 205]], order=3^5)
```

Polynomial

```
In [12]: x = np.array([[17, 4], [148, 205]]); x
Out[12]:
array([[ 17,   4],
       [148, 205]])

In [13]: GF(x)
Out[13]:
GF([[             ^2 + 2 + 2,                        + 1],
    [ ^4 + 2^3 + ^2 +  + 1, 2^4 + ^3 + ^2 + 2 + 1]], order=3^5)
```

Power

```
In [14]: x = np.array([[17, 4], [148, 205]]); x
Out[14]:
array([[ 17,   4],
       [148, 205]])

In [15]: GF(x)
Out[15]:
GF([[^222,  ^69],
    [ ^54,  ^24]], order=3^5)
```

- *Array*: A previously-created *Array* object. No coercion is necessary.

Alias

alias of `Union[Sequence[Union[int, str, Array]], Sequence[IterableLike], ndarray, Array]`

`galois.typing.DTypeLike`

A `Union` representing objects that can be coerced into a NumPy data type.

Union

- numpy.integer: A fixed-width NumPy integer data type.

```
In [1]: GF = galois.GF(3**5)

In [2]: x = GF.Random(4, dtype=np.uint16); x.dtype
Out[2]: dtype('uint16')

In [3]: x = GF.Random(4, dtype=np.int32); x.dtype
Out[3]: dtype('int32')
```

- int: The system default integer.

```
In [4]: x = GF.Random(4, dtype=int); x.dtype
Out[4]: dtype('int64')
```

- str: The string that can be coerced with numpy.dtype.

```
In [5]: x = GF.Random(4, dtype="uint16"); x.dtype
Out[5]: dtype('uint16')

In [6]: x = GF.Random(4, dtype="int32"); x.dtype
Out[6]: dtype('int32')
```

- object: A Python object data type. This applies to non-compiled fields.

```
In [7]: GF = galois.GF(2**100)

In [8]: x = GF.Random(4, dtype=object); x.dtype
Out[8]: dtype('O')
```

Alias

alias of Union[integer, int, str, object]

galois.typing.**ElementLike**

A Union representing objects that can be coerced into a Galois field element.

Scalars are 0-D *Array* objects.

Union

- int: A finite field element in its *integer representation*.

Integer

```
In [1]: GF = galois.GF(3**5)

In [2]: GF(17)
Out[2]: GF(17, order=3^5)
```

Polynomial

```
In [3]: GF = galois.GF(3**5, repr="poly")

In [4]: GF(17)
Out[4]: GF(^2 + 2 + 2, order=3^5)
```

Power

```
In [5]: GF = galois.GF(3**5, repr="power")

In [6]: GF(17)
Out[6]: GF(^222, order=3^5)
```

- `str`: A finite field element in its *polynomial representation*. Many string conventions are accepted, including: with/without `*`, with/without spaces, `^` or `**`, any indeterminate variable, increasing/decreasing degrees, etc. Or any combination of the above.

Integer

```
In [7]: GF("x^2 + 2x + 2")
Out[7]: GF(17, order=3^5)

# Add explicit * for multiplication
In [8]: GF("x^2 + 2*x + 2")
Out[8]: GF(17, order=3^5)

# No spaces
In [9]: GF("x^2+2x+2")
Out[9]: GF(17, order=3^5)

# ** instead of ^
In [10]: GF("x**2 + 2x + 2")
Out[10]: GF(17, order=3^5)

# Different indeterminate
In [11]: GF("^2 + 2 + 2")
Out[11]: GF(17, order=3^5)

# Ascending degrees
In [12]: GF("2 + 2x + x^2")
Out[12]: GF(17, order=3^5)
```

Polynomial

```
In [13]: GF("x^2 + 2x + 2")
Out[13]: GF(^2 + 2 + 2, order=3^5)

# Add explicit * for multiplication
In [14]: GF("x^2 + 2*x + 2")
Out[14]: GF(^2 + 2 + 2, order=3^5)

# No spaces
In [15]: GF("x^2+2x+2")
Out[15]: GF(^2 + 2 + 2, order=3^5)

# ** instead of ^
In [16]: GF("x**2 + 2x + 2")
Out[16]: GF(^2 + 2 + 2, order=3^5)

# Different indeterminate
In [17]: GF("^2 + 2 + 2")
Out[17]: GF(^2 + 2 + 2, order=3^5)

# Ascending degrees
In [18]: GF("2 + 2x + x^2")
Out[18]: GF(^2 + 2 + 2, order=3^5)
```

Power

```
In [19]: GF("x^2 + 2x + 2")
Out[19]: GF(^222, order=3^5)

# Add explicit * for multiplication
In [20]: GF("x^2 + 2*x + 2")
Out[20]: GF(^222, order=3^5)

# No spaces
In [21]: GF("x^2+2x+2")
Out[21]: GF(^222, order=3^5)

# ** instead of ^
In [22]: GF("x**2 + 2x + 2")
Out[22]: GF(^222, order=3^5)

# Different indeterminate
In [23]: GF("^2 + 2 + 2")
Out[23]: GF(^222, order=3^5)

# Ascending degrees
In [24]: GF("2 + 2x + x^2")
Out[24]: GF(^222, order=3^5)
```

- *Array*: A previously-created scalar *Array* object. No coercion is necessary.

Alias

alias of `Union`[`int`, `str`, `Array`]

`galois.typing.IterableLike`

A `Union` representing iterable objects that can be coerced into a Galois field array.

Union

- `Sequence` [*`ElementLike`*]: An iterable of elements.

Integer

```
In [1]: GF = galois.GF(3**5)

In [2]: GF([17, 4])
Out[2]: GF([17,  4], order=3^5)

# Mix and match integers and strings
In [3]: GF([17, "x + 1"])
Out[3]: GF([17,  4], order=3^5)
```

Polynomial

```
In [4]: GF = galois.GF(3**5, repr="poly")

In [5]: GF([17, 4])
Out[5]: GF([^2 + 2 + 2,          + 1], order=3^5)

# Mix and match integers and strings
In [6]: GF([17, "x + 1"])
Out[6]: GF([^2 + 2 + 2,          + 1], order=3^5)
```

Power

```
In [7]: GF = galois.GF(3**5, repr="power")

In [8]: GF([17, 4])
Out[8]: GF([^222,   ^69], order=3^5)

# Mix and match integers and strings
In [9]: GF([17, "x + 1"])
Out[9]: GF([^222,   ^69], order=3^5)
```

- `Sequence` [*`IterableLike`*]: A recursive iterable of iterables of elements.

Integer

```
In [10]: GF = galois.GF(3**5)

In [11]: GF([[17, 4], [148, 205]])
Out[11]:
GF([[ 17,   4],
    [148, 205]], order=3^5)

# Mix and match integers and strings
In [12]: GF([["x^2 + 2x + 2", 4], ["x^4 + 2x^3 + x^2 + x + 1", 205]])
Out[12]:
GF([[ 17,   4],
    [148, 205]], order=3^5)
```

Polynomial

```
In [13]: GF = galois.GF(3**5, repr="poly")

In [14]: GF([[17, 4], [148, 205]])
Out[14]:
GF([[             ^2 + 2 + 2,                        + 1],
    [ ^4 + 2^3 + ^2 +  + 1, 2^4 + ^3 + ^2 + 2 + 1]], order=3^5)

# Mix and match integers and strings
In [15]: GF([["x^2 + 2x + 2", 4], ["x^4 + 2x^3 + x^2 + x + 1", 205]])
Out[15]:
GF([[             ^2 + 2 + 2,                        + 1],
    [ ^4 + 2^3 + ^2 +  + 1, 2^4 + ^3 + ^2 + 2 + 1]], order=3^5)
```

Power

```
In [16]: GF = galois.GF(3**5, repr="power")

In [17]: GF([[17, 4], [148, 205]])
Out[17]:
GF([[^222,  ^69],
    [ ^54,  ^24]], order=3^5)

# Mix and match integers and strings
In [18]: GF([["x^2 + 2x + 2", 4], ["x^4 + 2x^3 + x^2 + x + 1", 205]])
Out[18]:
GF([[^222,  ^69],
    [ ^54,  ^24]], order=3^5)
```

Alias

alias of Union[Sequence[Union[int, str, Array]], Sequence[*IterableLike*]]

galois.typing.**ShapeLike**

A Union representing objects that can be coerced into a NumPy shape tuple.

Union

- int: The size of a 1-D array.

```
In [1]: GF = galois.GF(3**5)

In [2]: x = GF.Random(4); x
Out[2]: GF([207, 130, 105, 242], order=3^5)

In [3]: x.shape
Out[3]: (4,)
```

- Sequence [int]: An iterable of integer dimensions. Tuples or lists are allowed. An empty iterable, () or [], represents a 0-D array (scalar).

```
In [4]: x = GF.Random((2, 3)); x
Out[4]:
GF([[149,   9, 194],
    [169,  38, 166]], order=3^5)

In [5]: x.shape
Out[5]: (2, 3)

In [6]: x = GF.Random([2, 3, 4]); x
Out[6]:
GF([[[198,  81,   5,  31],
     [189, 230, 234, 158],
     [134, 193, 206, 233]],

    [[ 89,  94, 165,   3],
     [ 60, 170, 188, 215],
     [152, 167,  69, 218]]], order=3^5)

In [7]: x.shape
Out[7]: (2, 3, 4)

In [8]: x = GF.Random(()); x
Out[8]: GF(123, order=3^5)

In [9]: x.shape
Out[9]: ()
```

Alias

alias of Union[int, Sequence[int]]

3.19 Galois fields

class galois.***FieldArray***(*galois.Array*)

An abstract ndarray subclass over $\mathrm{GF}(p^m)$.

class galois.***GF2***(*galois.FieldArray*)

A *FieldArray* subclass over $\mathrm{GF}(2)$.

galois.***Field***(*order: int*, *, ...) → type[*FieldArray*]

galois.***Field***(*characteristic: int*, *degree*, ...) → type[*FieldArray*]

Alias of *GF()*.

galois.***GF***(order: *int*, *, ...) → type[*FieldArray*]

galois.***GF***(characteristic: *int*, degree: *int*, ...) → type[*FieldArray*]

Creates a *FieldArray* subclass for $\mathrm{GF}(p^m)$.

class galois.**FieldArray**(*galois.Array*)

An abstract ndarray subclass over $\mathrm{GF}(p^m)$.

Abstract

FieldArray is an abstract base class and cannot be instantiated directly. Instead, *FieldArray* subclasses are created using the class factory *GF()*.

Examples

Create a *FieldArray* subclass over $\mathrm{GF}(3^5)$ using the class factory *GF()*.

Integer

```
In [1]: GF = galois.GF(3**5)

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

In [3]: print(GF.properties)
Galois Field:
  name: GF(3^5)
  characteristic: 3
  degree: 5
  order: 243
  irreducible_poly: x^5 + 2x + 1
  is_primitive_poly: True
  primitive_element: x
```

Polynomial

```
In [4]: GF = galois.GF(3**5, repr="poly")

In [5]: issubclass(GF, galois.FieldArray)
Out[5]: True

In [6]: print(GF.properties)
Galois Field:
  name: GF(3^5)
  characteristic: 3
  degree: 5
  order: 243
  irreducible_poly: x^5 + 2x + 1
  is_primitive_poly: True
  primitive_element: x
```

Power

```
In [7]: GF = galois.GF(3**5, repr="power")

In [8]: issubclass(GF, galois.FieldArray)
Out[8]: True

In [9]: print(GF.properties)
Galois Field:
  name: GF(3^5)
  characteristic: 3
  degree: 5
  order: 243
  irreducible_poly: x^5 + 2x + 1
  is_primitive_poly: True
  primitive_element: x
```

Create a *FieldArray* instance using GF's constructor.

Integer

```
In [10]: x = GF([44, 236, 206, 138]); x
Out[10]: GF([ 44, 236, 206, 138], order=3^5)

In [11]: isinstance(x, GF)
Out[11]: True
```

Polynomial

```
In [12]: x = GF([44, 236, 206, 138]); x
Out[12]:
GF([      ^3 + ^2 + 2 + 2,    2^4 + 2^3 + 2^2 + 2,
    2^4 + ^3 + ^2 + 2 + 2,           ^4 + 2^3 + ], order=3^5)

In [13]: isinstance(x, GF)
Out[13]: True
```

Power

```
In [14]: x = GF([44, 236, 206, 138]); x
Out[14]: GF([^143, ^204,  ^55, ^113], order=3^5)

In [15]: isinstance(x, GF)
Out[15]: True
```

Constructors

FieldArray(x: ElementLike | ArrayLike, ...)

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

classmethod ***Identity***(size: *int*, ...) → Self

Creates an $n \times n$ identity matrix.

classmethod ***Ones***(shape: ShapeLike, ...) → Self

Creates an array of all ones.

classmethod ***Random***(shape: ShapeLike = (), ...) → Self

Creates an array with random elements.

classmethod ***Range***(start: ElementLike, stop, ...) → Self

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

classmethod ***Vandermonde***(element: ElementLike, rows, ...) → Self

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

classmethod ***Zeros***(shape: ShapeLike, ...) → Self

Creates an array of all zeros.

galois.**FieldArray**(x: ElementLike | ArrayLike, dtype: DTypeLike | *None* = ***None***, copy: *bool* = ***True***, order: *'K' | 'A' | 'C' | 'F'* = *'K'*, ndmin: *int* = *0*)

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

Parameters

x: *ElementLike | ArrayLike*
A finite field scalar or array.

dtype: *DTypeLike* | None = None
The `numpy.dtype` of the array elements. The default is `None` which represents the smallest unsigned data type for this *FieldArray* subclass (the first element in *dtypes*).

copy: bool = True
The `copy` keyword argument from `numpy.array()`. The default is `True`.

order: 'K' | 'A' | 'C' | 'F' = 'K'
The `order` keyword argument from `numpy.array()`. The default is `"K"`.

ndmin: int = 0
The `ndmin` keyword argument from `numpy.array()`. The default is 0.

Examples

Create a *FieldArray* subclass for $\mathrm{GF}(3^5)$.

Integer

```
In [1]: GF = galois.GF(3**5)

In [2]: print(GF.properties)
Galois Field:
  name: GF(3^5)
  characteristic: 3
  degree: 5
  order: 243
  irreducible_poly: x^5 + 2x + 1
  is_primitive_poly: True
  primitive_element: x

In [3]: alpha = GF.primitive_element; alpha
Out[3]: GF(3, order=3^5)
```

Polynomial

```
In [4]: GF = galois.GF(3**5, repr="poly")

In [5]: print(GF.properties)
Galois Field:
  name: GF(3^5)
  characteristic: 3
  degree: 5
  order: 243
```

(continues on next page)

(continued from previous page)

```
  irreducible_poly: x^5 + 2x + 1
  is_primitive_poly: True
  primitive_element: x

In [6]: alpha = GF.primitive_element; alpha
Out[6]: GF(, order=3^5)
```

Power

```
In [7]: GF = galois.GF(3**5, repr="power")

In [8]: print(GF.properties)
Galois Field:
  name: GF(3^5)
  characteristic: 3
  degree: 5
  order: 243
  irreducible_poly: x^5 + 2x + 1
  is_primitive_poly: True
  primitive_element: x

In [9]: alpha = GF.primitive_element; alpha
Out[9]: GF(, order=3^5)
```

Create a finite field scalar from its integer representation, polynomial representation, or a power of the primitive element.

Integer

```
In [10]: GF(17)
Out[10]: GF(17, order=3^5)

In [11]: GF("x^2 + 2x + 2")
Out[11]: GF(17, order=3^5)

In [12]: alpha ** 222
Out[12]: GF(17, order=3^5)
```

Polynomial

```
In [13]: GF(17)
Out[13]: GF(^2 + 2 + 2, order=3^5)

In [14]: GF("x^2 + 2x + 2")
Out[14]: GF(^2 + 2 + 2, order=3^5)

In [15]: alpha ** 222
Out[15]: GF(^2 + 2 + 2, order=3^5)
```

Power

```
In [16]: GF(17)
Out[16]: GF(^222, order=3^5)

In [17]: GF("x^2 + 2x + 2")
Out[17]: GF(^222, order=3^5)

In [18]: alpha ** 222
Out[18]: GF(^222, order=3^5)
```

Create a finite field array from its integer representation, polynomial representation, or powers of the primitive element.

Integer

```
In [19]: GF([17, 4, 148, 205])
Out[19]: GF([ 17,   4, 148, 205], order=3^5)

In [20]: GF([["x^2 + 2x + 2", 4], ["x^4 + 2x^3 + x^2 + x + 1", 205]])
Out[20]:
GF([[ 17,   4],
    [148, 205]], order=3^5)

In [21]: alpha ** np.array([[222, 69], [54, 24]])
Out[21]:
GF([[ 17,   4],
    [148, 205]], order=3^5)
```

Polynomial

```
In [22]: GF([17, 4, 148, 205])
Out[22]:
GF([                 ^2 + 2 + 2,                             + 1,
      ^4 + 2^3 + ^2 +  + 1, 2^4 + ^3 + ^2 + 2 + 1], order=3^5)

In [23]: GF([["x^2 + 2x + 2", 4], ["x^4 + 2x^3 + x^2 + x + 1", 205]])
Out[23]:
GF([[                 ^2 + 2 + 2,                             + 1],
    [ ^4 + 2^3 + ^2 +  + 1, 2^4 + ^3 + ^2 + 2 + 1]], order=3^5)

In [24]: alpha ** np.array([[222, 69], [54, 24]])
Out[24]:
GF([[                 ^2 + 2 + 2,                             + 1],
    [ ^4 + 2^3 + ^2 +  + 1, 2^4 + ^3 + ^2 + 2 + 1]], order=3^5)
```

Power

```
In [25]: GF([17, 4, 148, 205])
Out[25]: GF([^222,  ^69,  ^54,  ^24], order=3^5)

In [26]: GF([["x^2 + 2x + 2", 4], ["x^4 + 2x^3 + x^2 + x + 1", 205]])
Out[26]:
GF([[^222,  ^69],
    [ ^54,  ^24]], order=3^5)

In [27]: alpha ** np.array([[222, 69], [54, 24]])
Out[27]:
GF([[^222,  ^69],
    [ ^54,  ^24]], order=3^5)
```

classmethod `galois.FieldArray.Identity`(size: *int*, dtype: DTypeLike | *None* = ***None***) → Self

Creates an $n \times n$ identity matrix.

Parameters

size: int
The size n along one dimension of the identity matrix.

dtype: *DTypeLike* | None = None
The `numpy.dtype` of the array elements. The default is `None` which represents the smallest unsigned data type for this *`FieldArray`* subclass (the first element in *`dtypes`*).

Returns
A 2-D identity matrix with shape `(size, size)`.

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 `galois.FieldArray.Ones`(shape: ShapeLike, dtype: DTypeLike | *None* = ***None***) → Self

Creates an array of all ones.

Parameters

shape: *ShapeLike*
A NumPy-compliant `shape` tuple.

dtype: *DTypeLike* | None = None
The `numpy.dtype` of the array elements. The default is `None` which represents the smallest unsigned data type for this *`FieldArray`* subclass (the first element in *`dtypes`*).

Returns
An array of ones.

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 galois.FieldArray.**Random**(*shape: ShapeLike = ()*, *low: ElementLike = 0*, *high: ElementLike | None = None*, *seed: int | Generator | None = None*, *dtype: DTypeLike | None = None*) → Self

Creates an array with random elements.

Parameters

shape: *ShapeLike* = ()
A NumPy-compliant `shape` tuple. The default is `()` which represents a scalar.

low: *ElementLike* = 0
The smallest element (inclusive). The default is 0.

high: *ElementLike* | None = None
The largest element (exclusive). The default is `None` which represents `order`.

seed: int | Generator | None = None
Non-negative integer used to initialize the PRNG. The default is `None` which means that unpredictable entropy will be pulled from the OS to be used as the seed. A `numpy.random.Generator` can also be passed.

dtype: *DTypeLike* | None = None
The `numpy.dtype` of the array elements. The default is `None` which represents the smallest unsigned data type for this `FieldArray` subclass (the first element in `dtypes`).

Returns
An array of random elements.

Examples

Generate a random matrix with an unpredictable seed.

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

In [2]: GF.Random((2, 5))
Out[2]:
GF([[10, 29, 23,  2, 11],
    [13,  6, 18, 17,  2]], order=31)
```

Generate a random array with a specified seed. This produces repeatable outputs.

```
In [3]: GF.Random(10, seed=123456789)
Out[3]: GF([ 7, 29, 20, 27, 18,  5,  2,  0, 24, 24], order=31)

In [4]: GF.Random(10, seed=123456789)
Out[4]: GF([ 7, 29, 20, 27, 18,  5,  2,  0, 24, 24], order=31)
```

Generate a group of random arrays using a single global seed.

```
In [5]: rng = np.random.default_rng(123456789)

In [6]: GF.Random(10, seed=rng)
Out[6]: GF([ 7, 29, 20, 27, 18,  5,  2,  0, 24, 24], order=31)

In [7]: GF.Random(10, seed=rng)
Out[7]: GF([20, 15,  3, 28, 22,  0,  5, 10,  1,  0], order=31)
```

classmethod galois.FieldArray.Range(start: ElementLike, stop: ElementLike, step: *int* = *1*, dtype: DTypeLike | *None* = **None**) → Self

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

Parameters

start: ***ElementLike***
The starting element (inclusive).

stop: ***ElementLike***
The stopping element (exclusive).

step: **int** = **1**
The increment between elements. The default is 1.

dtype: ***DTypeLike*** **| None = None**
The `numpy.dtype` of the array elements. The default is **None** which represents the smallest unsigned data type for this *FieldArray* subclass (the first element in *dtypes*).

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

Examples

For prime fields, the increment is simply a finite field element, since all elements are integers.

```
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)

In [3]: GF.Range(10, 20, 2)
Out[3]: GF([10, 12, 14, 16, 18], order=31)
```

For extension fields, the increment is the integer increment between finite field elements in their *integer representation*.

Integer

```
In [4]: GF = galois.GF(3**3)

In [5]: GF.Range(10, 20)
Out[5]: GF([10, 11, 12, 13, 14, 15, 16, 17, 18, 19], order=3^3)

In [6]: GF.Range(10, 20, 2)
Out[6]: GF([10, 12, 14, 16, 18], order=3^3)
```

Polynomial

```
In [7]: GF = galois.GF(3**3, repr="poly")

In [8]: GF.Range(10, 20)
Out[8]:
GF([    ^2 + 1,      ^2 + 2,      ^2 + ,   ^2 +  + 1,  ^2 +  + 2,
       ^2 + 2, ^2 + 2 + 1, ^2 + 2 + 2,         2^2,     2^2 + 1],
   order=3^3)

In [9]: GF.Range(10, 20, 2)
Out[9]:
GF([    ^2 + 1,      ^2 + ,  ^2 +  + 2, ^2 + 2 + 1,         2^2],
   order=3^3)
```

classmethod galois.FieldArray.Vandermonde(*element: ElementLike*, *rows: int*, *cols: int*, *dtype: DTypeLike | None = None*) → Self

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

Parameters

element: *ElementLike*
An element a of $\mathrm{GF}(q)$.

rows: int
The number of rows m in the Vandermonde matrix.

cols: int
The number of columns n in the Vandermonde matrix.

dtype: *DTypeLike* | None = None
The `numpy.dtype` of the array elements. The default is `None` which represents the smallest unsigned data type for this *FieldArray* subclass (the first element in *dtypes*).

Returns
A $m \times n$ Vandermonde matrix.

Examples

Integer

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

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

In [3]: V = GF.Vandermonde(a, 7, 7); V
Out[3]:
GF([[1, 1, 1, 1, 1, 1, 1],
    [1, 2, 4, 3, 6, 7, 5],
    [1, 4, 6, 5, 2, 3, 7],
    [1, 3, 5, 4, 7, 2, 6],
    [1, 6, 2, 7, 4, 5, 3],
    [1, 7, 3, 2, 5, 6, 4],
    [1, 5, 7, 6, 3, 4, 2]], order=2^3)
```

Polynomial

```
In [4]: GF = galois.GF(2**3)

In [5]: a = GF.primitive_element; a
Out[5]: GF(, order=2^3)

In [6]: V = GF.Vandermonde(a, 7, 7); V
Out[6]:
GF([[          1,            1,            1,            1,            1,          ␣
↪1,           1],
    [          1,             ,          ^2,          + 1,      ^2 + , ^2 +  + 1,     ␣
↪^2 + 1],
    [          1,          ^2,      ^2 + ,      ^2 + 1,             ,          + 1, ^2␣
↪+  + 1],
    [          1,         + 1,      ^2 + 1,          ^2, ^2 +  + 1,            ,     ␣
↪ ^2 + ],
    [          1,      ^2 + ,            , ^2 +  + 1,          ^2,      ^2 + 1,     ␣
↪  + 1],
    [          1, ^2 +  + 1,         + 1,            ,      ^2 + 1,      ^2 + ,     ␣
↪   ^2],
    [          1,      ^2 + 1, ^2 +  + 1,      ^2 + ,         + 1,          ^2,     ␣
↪       ]], order=2^3)
```

Power

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

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

In [9]: V = GF.Vandermonde(a, 7, 7); V
Out[9]:
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 galois.FieldArray.**Zeros**(shape: ShapeLike, dtype: DTypeLike | *None* = ***None***) → Self

Creates an array of all zeros.

Parameters

shape: ***ShapeLike***
A NumPy-compliant shape tuple.

dtype: ***DTypeLike*** **| None = None**
The `numpy.dtype` of the array elements. The default is `None` which represents the smallest unsigned data type for this *FieldArray* subclass (the first element in *dtypes*).

Returns
An array of zeros.

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)
```

Conversions

`classmethod` ***Vector***(array: ArrayLike, ...) → *FieldArray*

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

vector(dtype: DTypeLike | *None* = ***None***) → *FieldArray*

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

`classmethod galois.FieldArray.`**`Vector`**(array: ArrayLike, dtype: DTypeLike | *None* = ***None***) → *FieldArray*

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

Parameters

array: *ArrayLike*
An array over $\mathrm{GF}(p)$ with last dimension m. An array with shape `(n1, n2, m)` has output shape `(n1, n2)`. By convention, the vectors are ordered from degree $m-1$ to degree 0.

dtype: *DTypeLike* | None = None
The `numpy.dtype` of the array elements. The default is `None` which represents the smallest unsigned data type for this *`FieldArray`* subclass (the first element in *`dtypes`*).

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

Notes

This method is the inverse of the *`vector()`* method.

Examples

Integer

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

In [2]: a = GF.Vector([[1, 0, 2], [0, 2, 1]]); a
Out[2]: GF([11,  7], order=3^3)

In [3]: a.vector()
```

(continues on next page)

(continued from previous page)

```
Out[3]:
GF([[1, 0, 2],
    [0, 2, 1]], order=3)
```

Polynomial

```
In [4]: GF = galois.GF(3**3, repr="poly")

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

In [6]: a.vector()
Out[6]:
GF([[1, 0, 2],
    [0, 2, 1]], order=3)
```

Power

```
In [7]: GF = galois.GF(3**3, repr="power")

In [8]: a = GF.Vector([[1, 0, 2], [0, 2, 1]]); a
Out[8]: GF([^12, ^16], order=3^3)

In [9]: a.vector()
Out[9]:
GF([[1, 0, 2],
    [0, 2, 1]], order=3)
```

galois.FieldArray.**vector**(dtype: DTypeLike | *None* = ***None***) → *FieldArray*

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

Parameters

dtype: *DTypeLike* | None = None
The `numpy.dtype` of the array elements. The default is `None` which represents the smallest unsigned data type for this *FieldArray* subclass (the first element in *dtypes*).

Returns

An array over $\mathrm{GF}(p)$ with last dimension m.

Notes

This method is the inverse 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 degree $m - 1$ to degree 0.

Examples

Integer

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

In [2]: a = GF([11, 7]); a
Out[2]: GF([11,  7], order=3^3)

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

In [4]: GF.Vector(vec)
Out[4]: GF([11,  7], order=3^3)
```

Polynomial

```
In [5]: GF = galois.GF(3**3, repr="poly")

In [6]: a = GF([11, 7]); a
Out[6]: GF([^2 + 2,  2 + 1], order=3^3)

In [7]: vec = a.vector(); vec
Out[7]:
GF([[1, 0, 2],
    [0, 2, 1]], order=3)

In [8]: GF.Vector(vec)
Out[8]: GF([^2 + 2,  2 + 1], order=3^3)
```

Power

```
In [9]: GF = galois.GF(3**3, repr="power")

In [10]: a = GF([11, 7]); a
Out[10]: GF([^12, ^16], order=3^3)

In [11]: vec = a.vector(); vec
Out[11]:
GF([[1, 0, 2],
    [0, 2, 1]], order=3)

In [12]: GF.Vector(vec)
Out[12]: GF([^12, ^16], order=3^3)
```

Elements

class property ***elements*** : *FieldArray*

All of the finite field's elements $\{0, \ldots, p^m - 1\}$.

class property ***non_squares*** : *FieldArray*

All non-squares in the Galois field.

class property ***primitive_element*** : *FieldArray*

A primitive element α of the Galois field $\mathrm{GF}(p^m)$.

class property ***primitive_elements*** : *FieldArray*

All primitive elements α of the Galois field $\mathrm{GF}(p^m)$.

classmethod ***primitive_root_of_unity***(n: *int*) → Self

Finds a primitive n-th root of unity in the finite field.

classmethod ***primitive_roots_of_unity***(n: *int*) → Self

Finds all primitive n-th roots of unity in the finite field.

class property ***squares*** : *FieldArray*

All squares in the finite field.

class property ***units*** : *FieldArray*

All of the finite field's units $\{1, \ldots, p^m - 1\}$.

class property galois.FieldArray.**elements** : *FieldArray*

All of the finite field's elements $\{0, \ldots, p^m - 1\}$.

Examples

All elements of the prime field $\mathrm{GF}(31)$ in increasing order.

Integer

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

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

Power

```
In [3]: GF = galois.GF(31, repr="power")

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

All elements of the extension field $\mathrm{GF}(5^2)$ in lexicographical order.

Integer

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

In [6]: GF.elements
Out[6]:
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], order=5^2)
```

Polynomial

```
In [7]: GF = galois.GF(5**2, repr="poly")

In [8]: GF.elements
Out[8]:
GF([     0,      1,      2,      3,      4,      ,   + 1,   + 2,
      + 3,   + 4,     2, 2 + 1, 2 + 2, 2 + 3, 2 + 4,     3,
    3 + 1, 3 + 2, 3 + 3, 3 + 4,     4, 4 + 1, 4 + 2, 4 + 3,
    4 + 4], order=5^2)
```

Power

```
In [9]: GF = galois.GF(5**2, repr="power")

In [10]: GF.elements
Out[10]:
GF([   0,    1,  ^6, ^18, ^12,     , ^22, ^15,  ^2, ^17,  ^7,
     ^8,  ^4, ^23, ^21, ^19,  ^9, ^11, ^16, ^20, ^13,  ^5,
    ^14,  ^3, ^10], order=5^2)
```

class property `galois.FieldArray.`**`non_squares`** : *FieldArray*

All non-squares in the Galois field.

Notes

An element x in $\mathrm{GF}(p^m)$ is a *non-square* if there does not exist a y such that $y^2 = x$ in the field.

See also

is_square

Examples

In fields with characteristic 2, no elements are non-squares.

Integer

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

In [2]: GF.non_squares
Out[2]: GF([], order=2^3)
```

Polynomial

```
In [3]: GF = galois.GF(2**3, repr="poly")

In [4]: GF.non_squares
Out[4]: GF([], order=2^3)
```

Power

```
In [5]: GF = galois.GF(2**3, repr="power")

In [6]: GF.non_squares
Out[6]: GF([], order=2^3)
```

In fields with characteristic greater than 2, exactly half of the nonzero elements are non-squares.

Integer

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

In [8]: GF.non_squares
Out[8]: GF([ 2,  6,  7,  8, 10], order=11)
```

Power

```
In [9]: GF = galois.GF(11, repr="power")

In [10]: GF.non_squares
Out[10]: GF([  , ^9, ^7, ^3, ^5], order=11)
```

class property galois.FieldArray.**primitive_element** : *FieldArray*

A primitive element α of the Galois field $\mathrm{GF}(p^m)$.

Notes

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

Examples

The smallest primitive element of the prime field $\mathrm{GF}(31)$.

Integer

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

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

Power

```
In [3]: GF = galois.GF(31, repr="power")

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

The smallest primitive element of the extension field $\mathrm{GF}(5^2)$.

Integer

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

In [6]: GF.elements
Out[6]:
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], order=5^2)
```

Polynomial

```
In [7]: GF = galois.GF(5**2, repr="poly")

In [8]: GF.elements
Out[8]:
GF([     0,       1,       2,       3,       4,       ,    + 1,    + 2,
       + 3,    + 4,      2, 2 + 1, 2 + 2, 2 + 3, 2 + 4,      3,
    3 + 1, 3 + 2, 3 + 3, 3 + 4,      4, 4 + 1, 4 + 2, 4 + 3,
    4 + 4], order=5^2)
```

Power

```
In [9]: GF = galois.GF(5**2, repr="power")

In [10]: GF.elements
Out[10]:
GF([   0,     1,  ^6, ^18, ^12,     , ^22, ^15,  ^2, ^17,  ^7,
     ^8,  ^4, ^23, ^21, ^19,  ^9, ^11, ^16, ^20, ^13,  ^5,
    ^14,  ^3, ^10], order=5^2)
```

class property galois.FieldArray.**primitive_elements** : *FieldArray*

All primitive elements α of the Galois field $\mathrm{GF}(p^m)$.

Notes

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

All primitive elements of the prime field $\mathrm{GF}(31)$ in increasing order.

Integer

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

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

Power

```
In [3]: GF = galois.GF(31, repr="power")

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

All primitive elements of the extension field $\mathrm{GF}(5^2)$ in lexicographical order.

Integer

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

In [6]: GF.elements
Out[6]:
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], order=5^2)
```

Polynomial

```
In [7]: GF = galois.GF(5**2, repr="poly")

In [8]: GF.elements
Out[8]:
GF([      0,       1,       2,       3,       4,        ,   + 1,   + 2,
      + 3,   + 4,      2, 2 + 1, 2 + 2, 2 + 3, 2 + 4,      3,
    3 + 1, 3 + 2, 3 + 3, 3 + 4,      4, 4 + 1, 4 + 2, 4 + 3,
    4 + 4], order=5^2)
```

Power

```
In [9]: GF = galois.GF(5**2, repr="power")

In [10]: GF.elements
Out[10]:
GF([   0,    1,  ^6, ^18, ^12,    , ^22, ^15,  ^2, ^17,  ^7,
     ^8,  ^4, ^23, ^21, ^19,  ^9, ^11, ^16, ^20, ^13,  ^5,
    ^14,  ^3, ^10], order=5^2)
```

classmethod `galois.FieldArray.primitive_root_of_unity(n: int) → Self`

Finds a primitive n-th root of unity in the finite field.

Parameters

n: int
The root of unity.

Returns
The primitive n-th root of unity, a 0-D scalar array.

Raises
`ValueError` – If no primitive n-th roots of unity exist. This happens when n is not a divisor of $p^m - 1$.

Notes

A primitive n-th root of unity ω_n is such that $\omega_n^n = 1$ and $\omega_n^k \neq 1$ for all $1 \leq kn$.

In $\mathrm{GF}(p^m)$, a primitive n-th root of unity exists when n divides $p^m - 1$. Then, the primitive root is $\omega_n = \alpha^{(p^m-1)/n}$ where α is a primitive element of the field.

Examples

In $\mathrm{GF}(31)$, primitive roots exist for all divisors of 30.

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

In [2]: GF.primitive_root_of_unity(2)
Out[2]: GF(30, order=31)

In [3]: GF.primitive_root_of_unity(5)
Out[3]: GF(16, order=31)

In [4]: GF.primitive_root_of_unity(15)
Out[4]: GF(9, order=31)
```

However, they do not exist for n that do not divide 30.

```
In [5]: GF.primitive_root_of_unity(7)
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[5], line 1
----> 1 GF.primitive_root_of_unity(7)

File ~/checkouts/readthedocs.org/user_builds/galois/envs/latest/lib/python3.8/
↪site-packages/galois/_fields/_array.py:888, in FieldArray.primitive_root_of_
↪unity(cls, n)
    886     raise ValueError(f"Argument 'n' must be in [1, {cls.order}), not {n}
↪.")
    887 if not (cls.order - 1) % n == 0:
--> 888     raise ValueError(f"There are no primitive {n}-th roots of unity in
↪{cls.name}.")
    890 return cls.primitive_element ** ((cls.order - 1) // n)

ValueError: There are no primitive 7-th roots of unity in GF(31).
```

For ω_5, one can see that $\omega_5^5 = 1$ and $\omega_5^k \neq 1$ for $1 \leq k5$.

```
In [6]: root = GF.primitive_root_of_unity(5); root
Out[6]: GF(16, order=31)
```

(continues on next page)

(continued from previous page)

```
In [7]: powers = np.arange(1, 5 + 1); powers
Out[7]: array([1, 2, 3, 4, 5])

In [8]: root ** powers
Out[8]: GF([16,  8,  4,  2,  1], order=31)
```

classmethod galois.FieldArray.**primitive_roots_of_unity**(n: *int*) → Self

Finds all primitive n-th roots of unity in the finite field.

Parameters

n: int
The root of unity.

Returns
All primitive n-th roots of unity, a 1-D array. The roots are sorted in lexicographical order.

Raises
`ValueError` – If no primitive n-th roots of unity exist. This happens when n is not a divisor of $p^m - 1$.

Notes

A primitive n-th root of unity ω_n is such that $\omega_n^n = 1$ and $\omega_n^k \neq 1$ for all $1 \leq kn$.

In $\mathrm{GF}(p^m)$, a primitive n-th root of unity exists when n divides $p^m - 1$. Then, the primitive root is $\omega_n = \alpha^{(p^m-1)/n}$ where α is a primitive element of the field.

Examples

In $\mathrm{GF}(31)$, primitive roots exist for all divisors of 30.

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

In [2]: GF.primitive_roots_of_unity(2)
Out[2]: GF([30], order=31)

In [3]: GF.primitive_roots_of_unity(5)
Out[3]: GF([ 2,  4,  8, 16], order=31)

In [4]: GF.primitive_roots_of_unity(15)
Out[4]: GF([ 7,  9, 10, 14, 18, 19, 20, 28], order=31)
```

However, they do not exist for n that do not divide 30.

```
In [5]: GF.primitive_roots_of_unity(7)
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[5], line 1
----> 1 GF.primitive_roots_of_unity(7)

File ~/checkouts/readthedocs.org/user_builds/galois/envs/latest/lib/python3.8/
```

(continues on next page)

(continued from previous page)

```
↪site-packages/galois/_fields/_array.py:950, in FieldArray.primitive_roots_of_
↪unity(cls, n)
    948     raise TypeError(f"Argument 'n' must be an int, not {type(n)!r}.")
    949 if not (cls.order - 1) % n == 0:
--> 950     raise ValueError(f"There are no primitive {n}-th roots of unity in
↪{cls.name}.")
    952 roots = np.unique(cls.primitive_elements ** ((cls.order - 1) // n))
    953 roots = np.sort(roots)

ValueError: There are no primitive 7-th roots of unity in GF(31).
```

For ω_5, one can see that $\omega_5^5 = 1$ and $\omega_5^k \neq 1$ for $1 \leq k5$.

```
In [6]: root = GF.primitive_roots_of_unity(5); root
Out[6]: GF([ 2,  4,  8, 16], order=31)

In [7]: powers = np.arange(1, 5 + 1); powers
Out[7]: array([1, 2, 3, 4, 5])

In [8]: np.power.outer(root, powers)
Out[8]:
GF([[ 2,  4,  8, 16,  1],
    [ 4, 16,  2,  8,  1],
    [ 8,  2, 16,  4,  1],
    [16,  8,  4,  2,  1]], order=31)
```

class property `galois.FieldArray.squares` : *FieldArray*

All squares in the finite field.

Notes

An element x in $\mathrm{GF}(p^m)$ is a *square* if there exists a y such that $y^2 = x$ in the field.

See also

is_square

Examples

In fields with characteristic 2, every element is a square (with two identical square roots).

Integer

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

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

In [3]: y1 = np.sqrt(x); y1
Out[3]: GF([0, 1, 6, 7, 2, 3, 4, 5], order=2^3)

In [4]: y2 = -y1; y2
Out[4]: GF([0, 1, 6, 7, 2, 3, 4, 5], order=2^3)

In [5]: np.array_equal(y1 ** 2, x)
Out[5]: True

In [6]: np.array_equal(y2 ** 2, x)
Out[6]: True
```

Polynomial

```
In [7]: GF = galois.GF(2**3, repr="poly")

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

In [9]: y1 = np.sqrt(x); y1
Out[9]:
GF([          0,            1,     ^2 + , ^2 +   + 1,              ,
           + 1,          ^2,     ^2 + 1], order=2^3)

In [10]: y2 = -y1; y2
Out[10]:
GF([          0,            1,     ^2 + , ^2 +   + 1,              ,
           + 1,          ^2,     ^2 + 1], order=2^3)

In [11]: np.array_equal(y1 ** 2, x)
Out[11]: True

In [12]: np.array_equal(y2 ** 2, x)
Out[12]: True
```

Power

```
In [13]: GF = galois.GF(2**3, repr="power")

In [14]: x = GF.squares; x
Out[14]: GF([  0,   1,    , ^3, ^2, ^6, ^4, ^5], order=2^3)

In [15]: y1 = np.sqrt(x); y1
Out[15]: GF([  0,   1, ^4, ^5,    , ^3, ^2, ^6], order=2^3)

In [16]: y2 = -y1; y2
Out[16]: GF([  0,   1, ^4, ^5,    , ^3, ^2, ^6], order=2^3)

In [17]: np.array_equal(y1 ** 2, x)
Out[17]: True

In [18]: np.array_equal(y2 ** 2, x)
Out[18]: True
```

In fields with characteristic greater than 2, exactly half of the nonzero elements are squares (with two unique square roots).

Integer

```
In [19]: GF = galois.GF(11)

In [20]: x = GF.squares; x
Out[20]: GF([0, 1, 3, 4, 5, 9], order=11)

In [21]: y1 = np.sqrt(x); y1
Out[21]: GF([0, 1, 5, 2, 4, 3], order=11)

In [22]: y2 = -y1; y2
Out[22]: GF([ 0, 10,  6,  9,  7,  8], order=11)

In [23]: np.array_equal(y1 ** 2, x)
Out[23]: True

In [24]: np.array_equal(y2 ** 2, x)
Out[24]: True
```

Power

```
In [25]: GF = galois.GF(11, repr="power")

In [26]: x = GF.squares; x
Out[26]: GF([  0,   1, ^8, ^2, ^4, ^6], order=11)

In [27]: y1 = np.sqrt(x); y1
Out[27]: GF([  0,   1, ^4,    , ^2, ^8], order=11)
```

(continues on next page)

(continued from previous page)

```
In [28]: y2 = -y1; y2
Out[28]: GF([  0, ^5, ^9, ^6, ^7, ^3], order=11)

In [29]: np.array_equal(y1 ** 2, x)
Out[29]: True

In [30]: np.array_equal(y2 ** 2, x)
Out[30]: True
```

class property galois.FieldArray.**units** : *FieldArray*

All of the finite field's units $\{1, \dots, p^m - 1\}$.

Notes

A unit is an element with a multiplicative inverse.

Examples

All units of the prime field $\mathrm{GF}(31)$ in increasing order.

Integer

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

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

Power

```
In [3]: GF = galois.GF(31, repr="power")

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

All units of the extension field $\mathrm{GF}(5^2)$ in lexicographical order.

Integer

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

In [6]: GF.units
Out[6]:
GF([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16, 17,
    18, 19, 20, 21, 22, 23, 24], order=5^2)
```

Polynomial

```
In [7]: GF = galois.GF(5**2, repr="poly")

In [8]: GF.units
Out[8]:
GF([     1,       2,       3,       4,        ,    + 1,    + 2,    + 3,
      + 4,      2, 2 + 1, 2 + 2, 2 + 3, 2 + 4,      3, 3 + 1,
    3 + 2, 3 + 3, 3 + 4,      4, 4 + 1, 4 + 2, 4 + 3, 4 + 4],
   order=5^2)
```

Power

```
In [9]: GF = galois.GF(5**2, repr="power")

In [10]: GF.units
Out[10]:
GF([   1,  ^6, ^18, ^12,     , ^22, ^15,  ^2, ^17,  ^7,  ^8,
     ^4, ^23, ^21, ^19,  ^9, ^11, ^16, ^20, ^13,  ^5, ^14,
     ^3, ^10], order=5^2)
```

String representation

__repr__() → str

Displays the array specifying the class and finite field order.

__str__() → str

Displays the array without specifying the class or finite field order.

classmethod ***arithmetic_table***(operation, ...) → str

Generates the specified arithmetic table for the finite field.

class property ***properties*** : str

A formatted string of relevant properties of the Galois field.

classmethod ***repr_table***(...) → str

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

galois.FieldArray.**__repr__**() → str

Displays the array specifying the class and finite field order.

Notes

This function prepends `GF(` and appends `, order=p^m)`.

Examples

Integer

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

In [2]: x = GF([4, 2, 7, 5])

In [3]: x
Out[3]: GF([4, 2, 7, 5], order=3^2)
```

Polynomial

```
In [4]: GF = galois.GF(3**2, repr="poly")

In [5]: x = GF([4, 2, 7, 5])

In [6]: x
Out[6]: GF([  + 1,      2, 2 + 1,    + 2], order=3^2)
```

Power

```
In [7]: GF = galois.GF(3**2, repr="power")

In [8]: x = GF([4, 2, 7, 5])

In [9]: x
Out[9]: GF([^2, ^4, ^3, ^7], order=3^2)
```

`galois.FieldArray.__str__()` → str

Displays the array without specifying the class or finite field order.

Notes

This function does not prepend `GF(` and or append `, order=p^m)`. It also omits the comma separators.

Examples

Integer

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

In [2]: x = GF([4, 2, 7, 5])

In [3]: print(x)
[4 2 7 5]
```

Polynomial

```
In [4]: GF = galois.GF(3**2, repr="poly")

In [5]: x = GF([4, 2, 7, 5])

In [6]: print(x)
[  + 1      2 2 + 1   + 2]
```

Power

```
In [7]: GF = galois.GF(3**2, repr="power")

In [8]: x = GF([4, 2, 7, 5])

In [9]: print(x)
[^2 ^4 ^3 ^7]
```

classmethod `galois.FieldArray.arithmetic_table`(*operation: '+' | '-' | '*' | '/'*, *x: FieldArray | None = None*, *y: FieldArray | None = None*) → str

Generates the specified arithmetic table for the finite field.

Parameters

operation: '+' | '-' | '*' | '/'
The arithmetic operation.

x: *FieldArray* | None = None
Optionally specify the x values for the arithmetic table. The default is `None` which represents $\{0, \dots, p^m - 1\}$.

y: *FieldArray* | None = None
Optionally specify the y values for the arithmetic table. The default is `None` which represents $\{0, \dots, p^m - 1\}$ for addition, subtraction, and multiplication and $\{1, \dots, p^m - 1\}$ for division.

Returns

A string representation of the arithmetic table.

Examples

Arithmetic tables can be displayed using any element representation.

Integer

```
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
```

Polynomial

```
In [3]: GF = galois.GF(3**2, repr="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
      |          + 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
```

Power

```
In [5]: GF = galois.GF(3**2, repr="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
```

(continues on next page)

(continued from previous page)

```
  ^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
```

An arithmetic table may also be constructed from arbitrary x and y.

Integer

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

In [8]: x = GF([7, 2, 8]); x
Out[8]: GF([7, 2, 8], order=3^2)

In [9]: y = GF([1, 4, 5, 3]); y
Out[9]: GF([1, 4, 5, 3], order=3^2)

In [10]: print(GF.arithmetic_table("+", x=x, y=y))
x + y | 1  4  5  3
------|------------
    7 | 8  2  0  1
    2 | 0  3  4  5
    8 | 6  0  1  2
```

Polynomial

```
In [11]: GF = galois.GF(3**2, repr="poly")

In [12]: x = GF([7, 2, 8]); x
Out[12]: GF([2 + 1,      2, 2 + 2], order=3^2)

In [13]: y = GF([1, 4, 5, 3]); y
Out[13]: GF([    1,   + 1,   + 2,     ], order=3^2)

In [14]: print(GF.arithmetic_table("+", x=x, y=y))
 x + y |      1    + 1    + 2
-------|--------------------------------
2 + 1 | 2 + 2       2       0       1
     2 |      0            + 1    + 2
2 + 2 |     2       0       1       2
```

Power

```
In [15]: GF = galois.GF(3**2, repr="power")

In [16]: x = GF([7, 2, 8]); x
Out[16]: GF([^3, ^4, ^6], order=3^2)

In [17]: y = GF([1, 4, 5, 3]); y
Out[17]: GF([  1, ^2, ^7,   ], order=3^2)

In [18]: print(GF.arithmetic_table("+", x=x, y=y))
x + y |    1  ^2  ^7
------|--------------------
  ^3 | ^6  ^4    0    1
  ^4 |   0      ^2  ^7
  ^6 | ^5    0    1  ^4
```

class property galois.FieldArray.**properties** : str

A formatted string of relevant properties of the Galois field.

Examples

```
In [1]: GF = galois.GF(7**5)

In [2]: print(GF.properties)
Galois Field:
  name: GF(7^5)
  characteristic: 7
  degree: 5
  order: 16807
  irreducible_poly: x^5 + x + 4
  is_primitive_poly: True
  primitive_element: x
```

classmethod galois.FieldArray.**repr_table**(element: ElementLike | *None* = None, sort: *'power'* | *'poly'* | *'vector'* | *'int'* = 'power') → str

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

Parameters

element: *ElementLike* | None = None
An element to use as the exponent base in the power representation. The default is `None` which corresponds to *primitive_element*.

sort: 'power' | 'poly' | 'vector' | 'int' = 'power'
The sorting method for the table. The default is `"power"`. Sorting by `"power"` will order the rows of the table by ascending powers of `element`. Sorting by any of the others will order the rows in lexicographical polynomial/vector order, which is equivalent to ascending order of the integer representation.

Returns

A string representation of the table comparing the power, polynomial, vector, and integer representations of each field element.

Examples

Create a *FieldArray* subclass for $\mathrm{GF}(3^3)$.

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

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

Generate a representation table for $\mathrm{GF}(3^3)$. Since $x^3 + 2x + 1$ is a primitive polynomial, x is a primitive element of the field. Notice, $\mathrm{ord}(x) = 26$.

```
In [3]: print(GF.repr_table())
 Power      Polynomial       Vector    Integer
------- --------------- ----------- ---------
   0           0         [0, 0, 0]      0
  x^0          1         [0, 0, 1]      1
  x^1          x         [0, 1, 0]      3
  x^2         x^2        [1, 0, 0]      9
  x^3        x + 2       [0, 1, 2]      5
  x^4       x^2 + 2x     [1, 2, 0]      15
  x^5     2x^2 + x + 2   [2, 1, 2]      23
  x^6     x^2 + x + 1    [1, 1, 1]      13
  x^7     x^2 + 2x + 2   [1, 2, 2]      17
  x^8       2x^2 + 2     [2, 0, 2]      20
  x^9        x + 1       [0, 1, 1]      4
  x^10      x^2 + x      [1, 1, 0]      12
  x^11    x^2 + x + 2    [1, 1, 2]      14
  x^12      x^2 + 2      [1, 0, 2]      11
  x^13         2         [0, 0, 2]      2
  x^14        2x         [0, 2, 0]      6
  x^15       2x^2        [2, 0, 0]      18
  x^16      2x + 1       [0, 2, 1]      7
  x^17      2x^2 + x     [2, 1, 0]      21
  x^18    x^2 + 2x + 1   [1, 2, 1]      16
  x^19   2x^2 + 2x + 2   [2, 2, 2]      26
  x^20    2x^2 + x + 1   [2, 1, 1]      22
  x^21      x^2 + 1      [1, 0, 1]      10
  x^22      2x + 2       [0, 2, 2]      8
  x^23     2x^2 + 2x     [2, 2, 0]      24
  x^24   2x^2 + 2x + 1   [2, 2, 1]      25
  x^25      2x^2 + 1     [2, 0, 1]      19

In [4]: GF("x").multiplicative_order()
Out[4]: 26
```

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

$\mathrm{ord}(2x^2 + 2x + 2) = 26$.

```
In [5]: print(GF.repr_table("2x^2 + 2x + 2"))
       Power              Polynomial       Vector     Integer
-------------------- --------------- ----------- ---------
         0                  0          [0, 0, 0]      0
 (2x^2 + 2x + 2)^0          1          [0, 0, 1]      1
 (2x^2 + 2x + 2)^1    2x^2 + 2x + 2    [2, 2, 2]      26
 (2x^2 + 2x + 2)^2       x^2 + 2       [1, 0, 2]      11
 (2x^2 + 2x + 2)^3     2x^2 + x + 2    [2, 1, 2]      23
 (2x^2 + 2x + 2)^4    2x^2 + 2x + 1    [2, 2, 1]      25
 (2x^2 + 2x + 2)^5       2x^2 + x      [2, 1, 0]      21
 (2x^2 + 2x + 2)^6       x^2 + x       [1, 1, 0]      12
 (2x^2 + 2x + 2)^7        x + 2        [0, 1, 2]      5
 (2x^2 + 2x + 2)^8        2x + 2       [0, 2, 2]      8
 (2x^2 + 2x + 2)^9         2x^2        [2, 0, 0]      18
 (2x^2 + 2x + 2)^10      2x^2 + 2      [2, 0, 2]      20
 (2x^2 + 2x + 2)^11         x          [0, 1, 0]      3
 (2x^2 + 2x + 2)^12    2x^2 + x + 1    [2, 1, 1]      22
 (2x^2 + 2x + 2)^13         2          [0, 0, 2]      2
 (2x^2 + 2x + 2)^14    x^2 + x + 1     [1, 1, 1]      13
 (2x^2 + 2x + 2)^15      2x^2 + 1      [2, 0, 1]      19
 (2x^2 + 2x + 2)^16    x^2 + 2x + 1    [1, 2, 1]      16
 (2x^2 + 2x + 2)^17    x^2 + x + 2     [1, 1, 2]      14
 (2x^2 + 2x + 2)^18      x^2 + 2x      [1, 2, 0]      15
 (2x^2 + 2x + 2)^19     2x^2 + 2x      [2, 2, 0]      24
 (2x^2 + 2x + 2)^20       2x + 1       [0, 2, 1]      7
 (2x^2 + 2x + 2)^21        x + 1       [0, 1, 1]      4
 (2x^2 + 2x + 2)^22         x^2        [1, 0, 0]      9
 (2x^2 + 2x + 2)^23      x^2 + 1       [1, 0, 1]      10
 (2x^2 + 2x + 2)^24         2x         [0, 2, 0]      6
 (2x^2 + 2x + 2)^25    x^2 + 2x + 2    [1, 2, 2]      17

In [6]: GF("2x^2 + 2x + 2").multiplicative_order()
Out[6]: 26
```

Generate a representation table for $\mathrm{GF}(3^3)$ using a non-primitive element x^2. Notice, $\mathrm{ord}(x^2) = 13 \neq 26$.

```
In [7]: print(GF.repr_table("x^2"))
  Power       Polynomial       Vector     Integer
---------- --------------- ----------- ---------
    0             0         [0, 0, 0]      0
 (x^2)^0          1         [0, 0, 1]      1
 (x^2)^1         x^2        [1, 0, 0]      9
 (x^2)^2       x^2 + 2x     [1, 2, 0]      15
 (x^2)^3     x^2 + x + 1    [1, 1, 1]      13
 (x^2)^4       2x^2 + 2     [2, 0, 2]      20
 (x^2)^5       x^2 + x      [1, 1, 0]      12
 (x^2)^6       x^2 + 2      [1, 0, 2]      11
 (x^2)^7          2x        [0, 2, 0]      6
 (x^2)^8        2x + 1      [0, 2, 1]      7
 (x^2)^9     x^2 + 2x + 1   [1, 2, 1]      16
 (x^2)^10    2x^2 + x + 1   [2, 1, 1]      22
 (x^2)^11       2x + 2      [0, 2, 2]      8
```

(continues on next page)

(continued from previous page)

```
 (x^2)^12   2x^2 + 2x + 1   [2, 2, 1]      25
 (x^2)^13         1         [0, 0, 1]      1
 (x^2)^14        x^2        [1, 0, 0]      9
 (x^2)^15      x^2 + 2x     [1, 2, 0]      15
 (x^2)^16    x^2 + x + 1    [1, 1, 1]      13
 (x^2)^17      2x^2 + 2     [2, 0, 2]      20
 (x^2)^18      x^2 + x      [1, 1, 0]      12
 (x^2)^19      x^2 + 2      [1, 0, 2]      11
 (x^2)^20         2x        [0, 2, 0]      6
 (x^2)^21       2x + 1      [0, 2, 1]      7
 (x^2)^22    x^2 + 2x + 1   [1, 2, 1]      16
 (x^2)^23    2x^2 + x + 1   [2, 1, 1]      22
 (x^2)^24       2x + 2      [0, 2, 2]      8
 (x^2)^25   2x^2 + 2x + 1   [2, 2, 1]      25

In [8]: GF("x^2").multiplicative_order()
Out[8]: 13
```

Element representation

class property ***element_repr*** : Literal[int] | Literal[poly] | Literal[power]

The current finite field element representation.

classmethod ***repr***(...) → Generator[None, None, None]

Sets the element representation for all arrays from this *FieldArray* subclass.

class property galois.FieldArray.**element_repr** : Literal[int] | Literal[poly] | Literal[power]

The current finite field element representation.

Notes

This can be changed with *repr()*. See *Element Representation* for a further discussion.

Examples

The default element representation is the integer representation.

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

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

In [3]: GF.element_repr
Out[3]: 'int'
```

Permanently modify the element representation by calling *repr()*.

```
In [4]: GF.repr("poly");

In [5]: x
```

(continues on next page)

(continued from previous page)

```
Out[5]:
GF([     0,       1,       2,       ,    + 1,    + 2,      2, 2 + 1,
    2 + 2], order=3^2)

In [6]: GF.element_repr
Out[6]: 'poly'
```

classmethod galois.FieldArray.**repr**(*element_repr: Literal[int] | Literal[poly] | Literal[power] = 'int'*) → Generator[None, None, None]

Sets the element representation for all arrays from this *FieldArray* subclass.

Parameters

element_repr: Literal[int] | Literal[poly] | Literal[power] = 'int'

The field element representation to be set.

- "int" (default): The *integer representation*.
- "poly": The *polynomial representation*.
- "power": The *power representation*.

Slower performance

To display elements in the power representation, *galois* must compute the discrete logarithm of each element displayed. For large fields or fields using *explicit calculation*, this process can take a while. However, when using *lookup tables* this representation is just as fast as the others.

Returns

A context manager for use in a **with** statement. If permanently setting the element representation, disregard the return value.

Notes

This function updates *element_repr*.

Examples

The default element representation is the integer representation.

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

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

Permanently set the element representation by calling *repr()*.

Polynomial

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

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

Power

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

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

Temporarily modify the element representation by using *repr()* as a context manager.

Polynomial

```
In [7]: print(x)
[0 1 2 3 4 5 6 7 8]

In [8]: with GF.repr("poly"):
   ...:     print(x)
   ...:
[     0      1      2          + 1    + 2      2 2 + 1 2 + 2]

# Outside the context manager, the element representation reverts to its␣
↪previous value
In [9]: print(x)
[0 1 2 3 4 5 6 7 8]
```

Power

```
In [10]: print(x)
[0 1 2 3 4 5 6 7 8]

In [11]: with GF.repr("power"):
   ....:     print(x)
   ....:
[  0   1 ^4     ^2 ^7 ^5 ^3 ^6]

# Outside the context manager, the element representation reverts to its␣
↪previous value
In [12]: print(x)
[0 1 2 3 4 5 6 7 8]
```

Arithmetic compilation

class property ***default_ufunc_mode*** : Literal[jit - lookup] | typing.Literal[jit - calculate] | typing.Literal[python - calculate]

The default ufunc compilation mode for this *FieldArray* subclass.

class property ***dtypes*** : list[np.dtype]

List of valid integer `numpy.dtype` values that are compatible with this finite field.

class property ***ufunc_mode*** : Literal[jit - lookup] | typing.Literal[jit - calculate] | typing.Literal[python - calculate]

The current ufunc compilation mode for this *FieldArray* subclass.

class property ***ufunc_modes*** : list[str]

All supported ufunc compilation modes for this *FieldArray* subclass.

classmethod ***compile***(mode)

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

class property galois.FieldArray.**default_ufunc_mode** : Literal[jit - lookup] | typing.Literal[jit - calculate] | typing.Literal[python - calculate]

The default ufunc compilation mode for this *FieldArray* subclass.

Notes

The ufuncs may be recompiled with *compile()*.

Examples

Fields with order less than 2^{20} are compiled, by default, using lookup tables for speed.

```
In [1]: galois.GF(65537).default_ufunc_mode
Out[1]: 'jit-lookup'

In [2]: galois.GF(2**16).default_ufunc_mode
Out[2]: 'jit-lookup'
```

Fields with order greater than 2^{20} are compiled, by default, using explicit calculation for memory savings. The field elements and arithmetic must still fit within `numpy.int64`.

```
In [3]: galois.GF(2147483647).default_ufunc_mode
Out[3]: 'jit-calculate'

In [4]: galois.GF(2**32).default_ufunc_mode
Out[4]: 'jit-calculate'
```

Fields whose elements and arithmetic cannot fit within `numpy.int64` use pure-Python explicit calculation.

```
In [5]: galois.GF(36893488147419103183).default_ufunc_mode
Out[5]: 'python-calculate'

In [6]: galois.GF(2**100).default_ufunc_mode
Out[6]: 'python-calculate'
```

class property galois.FieldArray.**dtypes** : list[np.dtype]

List of valid integer `numpy.dtype` values that are compatible with this finite field.

Notes

Creating an array with an unsupported dtype will raise a `TypeError` exception.

For finite fields whose elements cannot be represented with `numpy.int64`, the only valid data type is `numpy.object_`.

Examples

For small finite fields, all integer data types are acceptable, with the exception of `numpy.uint64`. This is because all arithmetic is done using `numpy.int64`.

```
In [1]: GF = galois.GF(31); GF.dtypes
Out[1]:
[numpy.uint8,
 numpy.uint16,
 numpy.uint32,
 numpy.int8,
 numpy.int16,
 numpy.int32,
 numpy.int64]
```

Some data types are too small for certain finite fields, such as `numpy.int16` for $\mathrm{GF}(7^5)$.

```
In [2]: GF = galois.GF(7**5); GF.dtypes
Out[2]: [numpy.uint16, numpy.uint32, numpy.int16, numpy.int32, numpy.int64]
```

Large fields must use `numpy.object_` which uses Python `int` for its unlimited size.

```
In [3]: GF = galois.GF(2**100); GF.dtypes
Out[3]: [numpy.object_]

In [4]: GF = galois.GF(36893488147419103183); GF.dtypes
Out[4]: [numpy.object_]
```

class property galois.FieldArray.**ufunc_mode** : Literal[jit - lookup] | typing.Literal[jit - calculate] | typing.Literal[python - calculate]

The current ufunc compilation mode for this *FieldArray* subclass.

Notes

The ufuncs may be recompiled with *compile()*.

Examples

Fields with order less than 2^{20} are compiled, by default, using lookup tables for speed.

```
In [1]: galois.GF(65537).ufunc_mode
Out[1]: 'jit-lookup'

In [2]: galois.GF(2**16).ufunc_mode
Out[2]: 'jit-lookup'
```

Fields with order greater than 2^{20} are compiled, by default, using explicit calculation for memory savings. The field elements and arithmetic must still fit within `numpy.int64`.

```
In [3]: galois.GF(2147483647).ufunc_mode
Out[3]: 'jit-calculate'

In [4]: galois.GF(2**32).ufunc_mode
Out[4]: 'jit-calculate'
```

Fields whose elements and arithmetic cannot fit within `numpy.int64` use pure-Python explicit calculation.

```
In [5]: galois.GF(36893488147419103183).ufunc_mode
Out[5]: 'python-calculate'

In [6]: galois.GF(2**100).ufunc_mode
Out[6]: 'python-calculate'
```

class property `galois.FieldArray.`**`ufunc_modes`** : list[str]

All supported ufunc compilation modes for this *FieldArray* subclass.

Notes

The ufuncs may be recompiled with *compile()*.

Examples

Fields whose elements and arithmetic can fit within `numpy.int64` can be JIT compiled to use either lookup tables or explicit calculation.

```
In [1]: galois.GF(65537).ufunc_modes
Out[1]: ['jit-lookup', 'jit-calculate']

In [2]: galois.GF(2**32).ufunc_modes
Out[2]: ['jit-lookup', 'jit-calculate']
```

Fields whose elements and arithmetic cannot fit within `numpy.int64` may only use pure-Python explicit calculation.

```
In [3]: galois.GF(36893488147419103183).ufunc_modes
Out[3]: ['python-calculate']

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

`classmethod galois.FieldArray.compile(mode: Literal[auto] | typing.Literal[jit - lookup] | typing.Literal[jit - calculate] | typing.Literal[python - calculate])`

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

This function updates *ufunc_mode*.

Parameters

mode: Literal[auto] | typing.Literal[jit - lookup] | typing.Literal[jit - calculate] | typing.Literal[python - calculate]

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).

Methods

`additive_order()` → int | ndarray

Computes the additive order of each element in x.

`characteristic_poly()` → *Poly*

Computes the characteristic polynomial of a finite field element a or a square matrix $\mathbf{A}$.

`field_norm()` → *FieldArray*

Computes the field norm $\mathrm{N}_{L/K}(x)$ of the elements of x.

`field_trace()` → *FieldArray*

Computes the field trace $\mathrm{Tr}_{L/K}(x)$ of the elements of x.

`log(base: ElementLike | ArrayLike | None = None)` → int | ndarray

Computes the discrete logarithm of the array x base β.

`minimal_poly()` → *Poly*

Computes the minimal polynomial of a finite field element a.

`multiplicative_order()` → int | ndarray

Computes the multiplicative order $\mathrm{ord}(x)$ of each element in x.

`galois.FieldArray.additive_order()` → int | ndarray

Computes the additive order of each element in x.

Returns

An integer array of the additive order of each element in x. The return value is a single integer if the input array x is a scalar.

Notes

The additive order a of x in $\mathrm{GF}(p^m)$ is the smallest integer a such that $xa = 0$. With the exception of 0, the additive order of every element is the finite field's characteristic.

Examples

Compute the additive order of each element of $\mathrm{GF}(3^2)$.

Integer

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

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

In [3]: order = x.additive_order(); order
Out[3]: array([1, 3, 3, 3, 3, 3, 3, 3, 3])

In [4]: x * order
Out[4]: GF([0, 0, 0, 0, 0, 0, 0, 0, 0], order=3^2)
```

Polynomial

```
In [5]: GF = galois.GF(3**2, repr="poly")

In [6]: x = GF.elements; x
Out[6]:
GF([     0,       1,       2,        ,     + 1,    + 2,      2, 2 + 1,
    2 + 2], order=3^2)

In [7]: order = x.additive_order(); order
Out[7]: array([1, 3, 3, 3, 3, 3, 3, 3, 3])

In [8]: x * order
Out[8]: GF([0, 0, 0, 0, 0, 0, 0, 0, 0], order=3^2)
```

Power

```
In [9]: GF = galois.GF(3**2, repr="power")

In [10]: x = GF.elements; x
Out[10]: GF([  0,    1, ^4,    , ^2, ^7, ^5, ^3, ^6], order=3^2)

In [11]: order = x.additive_order(); order
Out[11]: array([1, 3, 3, 3, 3, 3, 3, 3, 3])

In [12]: x * order
Out[12]: GF([0, 0, 0, 0, 0, 0, 0, 0, 0], order=3^2)
```

`galois.FieldArray.characteristic_poly()` → *Poly*

Computes the characteristic polynomial of a finite field element a or a square matrix $\mathbf{A}$.

Returns

For scalar inputs, the degree-m characteristic polynomial $c_a(x)$ of a over $\mathrm{GF}(p)$. For square $n \times n$ matrix inputs, the degree-n characteristic polynomial $c_A(x)$ of $\mathbf{A}$ over $\mathrm{GF}(p^m)$.

Raises

`ValueError` – If the array is not a single finite field element (scalar 0-D array) or a square $n \times n$ matrix (2-D array).

Notes

An element a of $\mathrm{GF}(p^m)$ has characteristic polynomial $c_a(x)$ over $\mathrm{GF}(p)$. The characteristic polynomial when evaluated in $\mathrm{GF}(p^m)$ annihilates a, that is $c_a(a) = 0$. In prime fields $\mathrm{GF}(p)$, the characteristic polynomial of a is simply $c_a(x) = x - a$.

An $n \times n$ matrix $\mathbf{A}$ has characteristic polynomial $c_A(x) = \det(x\mathbf{I} - \mathbf{A})$ over $\mathrm{GF}(p^m)$. The constant coefficient of the characteristic polynomial is $\det(-\mathbf{A})$. The x^{n-1} coefficient of the characteristic polynomial is $-\mathrm{Tr}(\mathbf{A})$. The characteristic polynomial annihilates $\mathbf{A}$, that is $c_A(\mathbf{A}) = \mathbf{0}$.

References

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

Examples

The characteristic polynomial of the element a.

Integer

```
In [1]: GF = galois.GF(3**5)

In [2]: a = GF.Random(); a
Out[2]: GF(100, order=3^5)

In [3]: poly = a.characteristic_poly(); poly
Out[3]: Poly(x^5 + x^2 + x + 2, GF(3))

# The characteristic polynomial annihilates a
In [4]: poly(a, field=GF)
Out[4]: GF(0, order=3^5)
```

Polynomial

```
In [5]: GF = galois.GF(3**5, repr="poly")

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

In [7]: poly = a.characteristic_poly(); poly
Out[7]: Poly(x^5 + x^3 + 2x^2 + 1, GF(3))

# The characteristic polynomial annihilates a
In [8]: poly(a, field=GF)
Out[8]: GF(0, order=3^5)
```

Power

```
In [9]: GF = galois.GF(3**5, repr="power")

In [10]: a = GF.Random(); a
Out[10]: GF(^210, order=3^5)

In [11]: poly = a.characteristic_poly(); poly
Out[11]: Poly(x^5 + x^3 + x^2 + 2x + 2, GF(3))

# The characteristic polynomial annihilates a
In [12]: poly(a, field=GF)
Out[12]: GF(0, order=3^5)
```

The characteristic polynomial of the square matrix $\mathbf{A}$.

Integer

```
In [13]: GF = galois.GF(3**5)

In [14]: A = GF.Random((3,3)); A
Out[14]:
GF([[ 59,  55, 139],
    [ 99,  76,  34],
    [ 63, 200, 145]], order=3^5)

In [15]: poly = A.characteristic_poly(); poly
Out[15]: Poly(x^3 + 167x^2 + 42x + 203, GF(3^5))

# The x^0 coefficient is det(-A)
In [16]: poly.coeffs[-1] == np.linalg.det(-A)
Out[16]: True

# The x^n-1 coefficient is -Tr(A)
In [17]: poly.coeffs[1] == -np.trace(A)
Out[17]: True
```

(continues on next page)

(continued from previous page)

```
# The characteristic polynomial annihilates the matrix A
In [18]: poly(A, elementwise=False)
Out[18]:
GF([[0, 0, 0],
    [0, 0, 0],
    [0, 0, 0]], order=3^5)
```

Polynomial

```
In [19]: GF = galois.GF(3**5, repr="poly")

In [20]: A = GF.Random((3,3)); A
Out[20]:
GF([[     ^3 + ^2 + 2 + 1,                  2 + 2,
          2^4 + ^3 +  + 1],
    [     ^3 + 2^2 +  + 2, ^4 + ^3 + ^2 +  + 1,
       ^4 + 2^3 + ^2 + 2],
    [        2^4 + 2^2 + 2,                 ^4 + ,
                 2^2 + ]], order=3^5)

In [21]: poly = A.characteristic_poly(); poly
Out[21]: Poly(x^3 + (2^4 + ^3 + 2^2 + 2 + 1)x^2 + (2^4 + ^3 + 2)x + (^4 + ^3 +␣
↪1), GF(3^5))

# The x^0 coefficient is det(-A)
In [22]: poly.coeffs[-1] == np.linalg.det(-A)
Out[22]: True

# The x^n-1 coefficient is -Tr(A)
In [23]: poly.coeffs[1] == -np.trace(A)
Out[23]: True

# The characteristic polynomial annihilates the matrix A
In [24]: poly(A, elementwise=False)
Out[24]:
GF([[0, 0, 0],
    [0, 0, 0],
    [0, 0, 0]], order=3^5)
```

Power

```
In [25]: GF = galois.GF(3**5, repr="power")

In [26]: A = GF.Random((3,3)); A
Out[26]:
GF([[ ^96,  ^15, ^193],
    [^197, ^104, ^196],
    [^131, ^176,  ^52]], order=3^5)
```

(continues on next page)

(continued from previous page)

```
In [27]: poly = A.characteristic_poly(); poly
Out[27]: Poly(x^3 + (^141)x^2 + (^110)x + ^190, GF(3^5))

# The x^0 coefficient is det(-A)
In [28]: poly.coeffs[-1] == np.linalg.det(-A)
Out[28]: True

# The x^n-1 coefficient is -Tr(A)
In [29]: poly.coeffs[1] == -np.trace(A)
Out[29]: True

# The characteristic polynomial annihilates the matrix A
In [30]: poly(A, elementwise=False)
Out[30]:
GF([[0, 0, 0],
    [0, 0, 0],
    [0, 0, 0]], order=3^5)
```

galois.FieldArray.**field_norm**() → *FieldArray*

Computes the field norm $\mathrm{N}_{L/K}(x)$ of the elements of x.

Returns

The field norm of x in the prime subfield $\mathrm{GF}(p)$.

Notes

The `self` array x is over the extension field $L = \mathrm{GF}(p^m)$. The field norm of x is over the subfield $K = \mathrm{GF}(p)$. In other words, $\mathrm{N}_{L/K}(x) : L \rightarrow K$.

For finite fields, since L is a Galois extension of K, the field norm of x is defined as a product of the Galois conjugates of x.

$$\mathrm{N}_{L/K}(x) = \prod_{i=0}^{m-1} x^{p^i} = x^{(p^m-1)/(p-1)}$$

References

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

Examples

Compute the field norm of the elements of $\mathrm{GF}(3^2)$.

Integer

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

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

In [3]: y = x.field_norm(); y
Out[3]: GF([0, 1, 1, 2, 1, 2, 2, 2, 1], order=3)
```

Polynomial

```
In [4]: GF = galois.GF(3**2, repr="poly")

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

In [6]: y = x.field_norm(); y
Out[6]: GF([0, 1, 1, 2, 1, 2, 2, 2, 1], order=3)
```

Power

```
In [7]: GF = galois.GF(3**2, repr="power")

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

In [9]: y = x.field_norm(); y
Out[9]: GF([0, 1, 1, 2, 1, 2, 2, 2, 1], order=3)
```

`galois.FieldArray.field_trace()` → *FieldArray*

Computes the field trace $\mathrm{Tr}_{L/K}(x)$ of the elements of x.

Returns

The field trace of x in the prime subfield $\mathrm{GF}(p)$.

Notes

The `self` array x is over the extension field $L = \mathrm{GF}(p^m)$. The field trace of x is over the subfield $K = \mathrm{GF}(p)$. In other words, $\mathrm{Tr}_{L/K}(x) : L \rightarrow K$.

For finite fields, since L is a Galois extension of K, the field trace of x is defined as a sum of the Galois conjugates of x.

$$\mathrm{Tr}_{L/K}(x) = \sum_{i=0}^{m-1} x^{p^i}$$

References

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

Examples

Compute the field trace of the elements of $\mathrm{GF}(3^2)$.

Integer

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

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

In [3]: y = x.field_trace(); y
Out[3]: GF([0, 2, 1, 1, 0, 2, 2, 1, 0], order=3)
```

Polynomial

```
In [4]: GF = galois.GF(3**2, repr="poly")

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

In [6]: y = x.field_trace(); y
Out[6]: GF([0, 2, 1, 1, 0, 2, 2, 1, 0], order=3)
```

Power

```
In [7]: GF = galois.GF(3**2, repr="power")

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

In [9]: y = x.field_trace(); y
Out[9]: GF([0, 2, 1, 1, 0, 2, 2, 1, 0], order=3)
```

`galois.FieldArray.log`(base: ElementLike | ArrayLike | *None* = ***None***) → int | ndarray

Computes the discrete logarithm of the array x base β.

Parameters

base: *ElementLike* | *ArrayLike* | None = None

A primitive element or elements β of the finite field that is the base of the logarithm. The default is `None` which uses `primitive_element`.

Slower performance

If the *FieldArray* is configured to use lookup tables (`ufunc_mode == "jit-lookup"`) and this method is invoked with a base different from *primitive_element*, then explicit calculation will be used (which is slower than using lookup tables).

Returns

An integer array i of powers of β such that $\beta^i = x$. The return array shape obeys NumPy broadcasting rules.

Examples

Compute the logarithm of x with default base α, which is the specified primitive element of the field.

Integer

```
In [1]: GF = galois.GF(3**5)

In [2]: alpha = GF.primitive_element; alpha
Out[2]: GF(3, order=3^5)

In [3]: x = GF.Random(10, low=1); x
Out[3]: GF([ 66,  44, 185, 157, 187, 112,  79, 118, 219,   5], order=3^5)

In [4]: i = x.log(); i
Out[4]: array([102, 143,  38, 176, 108,  34,  95,  83, 110,   5])

In [5]: np.array_equal(alpha ** i, x)
Out[5]: True
```

Polynomial

```
In [6]: GF = galois.GF(3**5, repr="poly")

In [7]: alpha = GF.primitive_element; alpha
Out[7]: GF(, order=3^5)

In [8]: x = GF.Random(10, low=1); x
Out[8]:
GF([          2^4 + 2^3 + ,              2^2 + 2 + 2,
          2^3 + 2^2 +  + 1, 2^4 + 2^3 + ^2 +  + 1,
              2^3 + 2^2 + ,         2^4 + ^2 +  + 2,
       2^4 + 2^3 + 2^2 + 1,         ^4 + 2^2 +  + 2,
       2^4 + ^3 + 2^2 + 2,        2^3 + ^2 +  + 2], order=3^5)

In [9]: i = x.log(); i
Out[9]: array([110, 131,  22, 163,  89, 114, 239, 232, 183, 200])
```

(continues on next page)

(continued from previous page)

```
In [10]: np.array_equal(alpha ** i, x)
Out[10]: True
```

Power

```
In [11]: GF = galois.GF(3**5, repr="power")

In [12]: alpha = GF.primitive_element; alpha
Out[12]: GF(, order=3^5)

In [13]: x = GF.Random(10, low=1); x
Out[13]:
GF([ ^96, ^171,  ^72, ^147, ^139, ^107, ^136,  ^74, ^146, ^155],
   order=3^5)

In [14]: i = x.log(); i
Out[14]: array([ 96, 171,  72, 147, 139, 107, 136,  74, 146, 155])

In [15]: np.array_equal(alpha ** i, x)
Out[15]: True
```

With the default argument, `numpy.log()` and *log()* are equivalent.

```
In [16]: np.array_equal(np.log(x), x.log())
Out[16]: True
```

Compute the logarithm of x with a different base β, which is another primitive element of the field.

Integer

```
In [17]: beta = GF.primitive_elements[-1]; beta
Out[17]: GF(242, order=3^5)

In [18]: i = x.log(beta); i
Out[18]: array([ 62, 239, 228, 163,  57, 117, 108, 194, 180,  27])

In [19]: np.array_equal(beta ** i, x)
Out[19]: True
```

Polynomial

```
In [20]: beta = GF.primitive_elements[-1]; beta
Out[20]: GF(2^4 + 2^3 + 2^2 + 2 + 2, order=3^5)

In [21]: i = x.log(beta); i
Out[21]: array([ 62, 239, 228, 163,  57, 117, 108, 194, 180,  27])

In [22]: np.array_equal(beta ** i, x)
Out[22]: True
```

Power

```
In [23]: beta = GF.primitive_elements[-1]; beta
Out[23]: GF(^185, order=3^5)

In [24]: i = x.log(beta); i
Out[24]: array([ 62, 239, 228, 163,  57, 117, 108, 194, 180,  27])

In [25]: np.array_equal(beta ** i, x)
Out[25]: True
```

Compute the logarithm of a single finite field element base all of the primitive elements of the field.

Integer

```
In [26]: x = GF.Random(low=1); x
Out[26]: GF(71, order=3^5)

In [27]: bases = GF.primitive_elements

In [28]: i = x.log(bases); i
Out[28]:
array([182, 178, 230, 212, 130, 146, 232, 124, 210, 172, 236, 222, 140,
       238, 112, 142,  96, 188,  98,   4,  10,  26, 158, 122,  70, 156,
        36,  54, 200, 168, 166, 126, 138, 240, 114, 234,  72,  92,  56,
       228, 164, 102,  42, 204, 120, 216,  34,  30,  14,  78, 136, 196,
       152, 148,  80,  46, 162,  28,  68,  40,  18,  24, 192, 190,  38,
        58, 180,  76, 106,  60, 208, 160,  74,  16,  82, 194,  48,   6,
       150, 134, 186, 144, 170, 174, 128, 118,   2,  84, 218,  90,  86,
        64,   8, 202, 104, 214, 100, 224,  32, 108, 226,  50, 206,  62,
       184,  94, 116,  20,  12,  52])

In [29]: np.all(bases ** i == x)
Out[29]: True
```

Polynomial

```
In [30]: x = GF.Random(low=1); x
Out[30]: GF(^4 + ^3 + ^2 + 2 + 2, order=3^5)

In [31]: bases = GF.primitive_elements

In [32]: i = x.log(bases); i
Out[32]:
array([ 65,  29,  13,  93,  81, 225,  31,  27,  75, 217,  67, 183, 171,
        85, 161, 189,  17, 119,  35, 157, 211, 113,  91,   9,  25,  73,
       203, 123, 227, 181, 163,  45, 153, 103, 179,  49,  43, 223, 141,
       237, 145,  71,  15,  21, 233, 129, 185, 149,   5,  97, 135, 191,
        37,   1, 115,  51, 127, 131,   7, 239,  41,  95, 155, 137, 221,
       159,  47,  79, 107, 177,  57, 109,  61,  23, 133, 173,  69, 175,
```

(continues on next page)

(continued from previous page)

```
        19, 117, 101, 207, 199, 235,  63, 215, 139, 151, 147, 205, 169,
       213, 193,   3,  89, 111,  53, 201, 167, 125, 219,  87,  39, 195,
        83, 241, 197,  59, 229, 105])

In [33]: np.all(bases ** i == x)
Out[33]: True
```

Power

```
In [34]: x = GF.Random(low=1); x
Out[34]: GF(^39, order=3^5)

In [35]: bases = GF.primitive_elements

In [36]: i = x.log(bases); i
Out[36]:
array([ 39, 211, 153, 201,  97, 135,  67, 113,  45, 227, 137,  13, 151,
        51, 145,  65, 107,  23,  21, 191, 175, 213, 103, 199,  15, 189,
        25, 219, 233, 157,   1,  27, 237, 207,  59, 223, 171,  37, 133,
       239,  87,  91,   9,  61,  43,  29, 111,  41,   3, 155,  81, 163,
       119,  49,  69,  79, 173, 127, 101,  95,  73,  57,  93, 179, 181,
        47, 125, 241, 161, 203, 131,  17,  85, 159, 225,   7, 235, 105,
       205, 167, 109, 221,  71, 141, 183, 129,  35, 139, 185, 123,  53,
        31,  19, 147,   5, 115, 177, 169, 197,  75,  83, 149, 217, 117,
       195, 193, 215, 229,  89,  63])

In [37]: np.all(bases ** i == x)
Out[37]: True
```

galois.FieldArray.**minimal_poly**() → *Poly*

Computes the minimal polynomial of a finite field element a.

Returns

For scalar inputs, the minimal polynomial $m_a(x)$ of a over $\mathrm{GF}(p)$.

Raises

- `NotImplementedError` – If the array is a a square $n \times n$ matrix (2-D array).
- `ValueError` – If the array is not a single finite field element (scalar 0-D array).

Notes

An element a of $\mathrm{GF}(p^m)$ has minimal polynomial $m_a(x)$ over $\mathrm{GF}(p)$. The minimal polynomial when evaluated in $\mathrm{GF}(p^m)$ annihilates a, that is $m_a(a) = 0$. The minimal polynomial always divides the characteristic polynomial. In prime fields $\mathrm{GF}(p)$, the minimal polynomial of a is simply $m_a(x) = x - a$.

References

- https://en.wikipedia.org/wiki/Minimal_polynomial_(field_theory)
- https://en.wikipedia.org/wiki/Minimal_polynomial_(linear_algebra)

Examples

The minimal polynomial of the element a.

Integer

```
In [1]: GF = galois.GF(3**5)

In [2]: a = GF.Random(); a
Out[2]: GF(112, order=3^5)

In [3]: poly = a.minimal_poly(); poly
Out[3]: Poly(x^5 + x^2 + x + 2, GF(3))

# The minimal polynomial annihilates a
In [4]: poly(a, field=GF)
Out[4]: GF(0, order=3^5)

# The minimal polynomial always divides the characteristic polynomial
In [5]: divmod(a.characteristic_poly(), poly)
Out[5]: (Poly(1, GF(3)), Poly(0, GF(3)))
```

Polynomial

```
In [6]: GF = galois.GF(3**5, repr="poly")

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

In [8]: poly = a.minimal_poly(); poly
Out[8]: Poly(x^5 + x^4 + 2x^3 + 2x + 2, GF(3))

# The minimal polynomial annihilates a
In [9]: poly(a, field=GF)
Out[9]: GF(0, order=3^5)

# The minimal polynomial always divides the characteristic polynomial
In [10]: divmod(a.characteristic_poly(), poly)
Out[10]: (Poly(1, GF(3)), Poly(0, GF(3)))
```

Power

```
In [11]: GF = galois.GF(3**5, repr="power")

In [12]: a = GF.Random(); a
Out[12]: GF(^225, order=3^5)

In [13]: poly = a.minimal_poly(); poly
Out[13]: Poly(x^5 + x^3 + 2x^2 + 2x + 1, GF(3))

# The minimal polynomial annihilates a
In [14]: poly(a, field=GF)
Out[14]: GF(0, order=3^5)

# The minimal polynomial always divides the characteristic polynomial
In [15]: divmod(a.characteristic_poly(), poly)
Out[15]: (Poly(1, GF(3)), Poly(0, GF(3)))
```

galois.FieldArray.**multiplicative_order**() → int | ndarray

Computes the multiplicative order $\mathrm{ord}(x)$ of each element in x.

Returns

An integer array of the multiplicative order of each element in x. The return value is a single integer if the input array x is a scalar.

Raises

`ArithmeticError` – If zero is provided as an input. The multiplicative order of 0 is not defined. There is no power of 0 that ever results in 1.

Notes

The multiplicative order $\mathrm{ord}(x) = a$ of x in $\mathrm{GF}(p^m)$ is the smallest power a such that $x^a = 1$. If $a = p^m - 1$, a is said to be a generator of the multiplicative group $\mathrm{GF}(p^m)^\times$.

Note, *multiplicative_order()* should not be confused with *order*. The former returns the multiplicative order of *FieldArray* elements. The latter is a property of the field, namely the finite field's order or size.

Examples

Compute the multiplicative order of each non-zero element of $\mathrm{GF}(3^2)$.

Integer

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

In [2]: x = GF.units; x
Out[2]: GF([1, 2, 3, 4, 5, 6, 7, 8], order=3^2)

In [3]: order = x.multiplicative_order(); order
Out[3]: array([1, 2, 8, 4, 8, 8, 8, 4])
```

(continues on next page)

(continued from previous page)

```
In [4]: x ** order
Out[4]: GF([1, 1, 1, 1, 1, 1, 1, 1], order=3^2)
```

Polynomial

```
In [5]: GF = galois.GF(3**2, repr="poly")

In [6]: x = GF.units; x
Out[6]:
GF([     1,       2,       ,    + 1,    + 2,      2, 2 + 1, 2 + 2],
   order=3^2)

In [7]: order = x.multiplicative_order(); order
Out[7]: array([1, 2, 8, 4, 8, 8, 8, 4])

In [8]: x ** order
Out[8]: GF([1, 1, 1, 1, 1, 1, 1, 1], order=3^2)
```

Power

```
In [9]: GF = galois.GF(3**2, repr="power")

In [10]: x = GF.units; x
Out[10]: GF([  1, ^4,   , ^2, ^7, ^5, ^3, ^6], order=3^2)

In [11]: order = x.multiplicative_order(); order
Out[11]: array([1, 2, 8, 4, 8, 8, 8, 4])

In [12]: x ** order
Out[12]: GF([1, 1, 1, 1, 1, 1, 1, 1], order=3^2)
```

The elements with $\mathrm{ord}(x) = 8$ are multiplicative generators of $\mathrm{GF}(3^2)^\times$, which are also called primitive elements.

Integer

```
In [13]: GF.primitive_elements
Out[13]: GF([3, 5, 6, 7], order=3^2)
```

Polynomial

```
In [14]: GF.primitive_elements
Out[14]: GF([     ,    + 2,      2, 2 + 1], order=3^2)
```

Power

```
In [15]: GF.primitive_elements
Out[15]: GF([  , ^7, ^5, ^3], order=3^2)
```

Linear algebra

column_space() → Self

Computes the column space of the matrix $\mathbf{A}$.

left_null_space() → Self

Computes the left null space of the matrix $\mathbf{A}$.

lu_decompose() → tuple[Self, Self]

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

null_space() → Self

Computes the null space of the matrix $\mathbf{A}$.

plu_decompose() → tuple[Self, Self, Self]

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

row_reduce(ncols: *int* | *None* = **None**, ...) → Self

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

row_space() → Self

Computes the row space of the matrix $\mathbf{A}$.

galois.FieldArray.**column_space**() → Self

Computes the column space of the matrix $\mathbf{A}$.

Returns

The column space basis matrix. The rows of the basis matrix are the basis vectors that span the column space. The number of rows of the basis matrix is the dimension of the column space.

Notes

Given an $m \times n$ matrix $\mathbf{A}$ over $\mathrm{GF}(q)$, the *column space* of $\mathbf{A}$ is the vector space $\{\mathbf{x} \in \mathrm{GF}(q)^m\}$ defined by all linear combinations of the columns of $\mathbf{A}$. The column space has at most dimension $\text{min}(m, n)$.

The column space has properties $\mathcal{C}(\mathbf{A}) = \mathcal{R}(\mathbf{A}^T)$ and $\text{dim}(\mathcal{C}(\mathbf{A})) + \text{dim}(\mathcal{N}(\mathbf{A})) = n$.

Examples

The `column_space()` method defines basis vectors (its rows) that span the column space of $\mathbf{A}$.

```
In [1]: m, n = 3, 5

In [2]: GF = galois.GF(31)

In [3]: A = GF.Random((m, n)); A
Out[3]:
GF([[24, 28, 18,  4, 13],
    [23, 28, 18, 10,  0],
    [28, 28, 10, 24, 21]], order=31)

In [4]: C = A.column_space(); C
Out[4]:
GF([[1, 0, 0],
    [0, 1, 0],
    [0, 0, 1]], order=31)
```

The dimension of the column space and null space sum to n.

```
In [5]: N = A.null_space(); N
Out[5]:
GF([[ 1,  0, 25, 24, 11],
    [ 0,  1, 15, 26, 12]], order=31)

In [6]: C.shape[0] + N.shape[0] == n
Out[6]: True
```

`galois.FieldArray.`**`left_null_space`**`()` → Self

Computes the left null space of the matrix $\mathbf{A}$.

Returns

The left null space basis matrix. The rows of the basis matrix are the basis vectors that span the left null space. The number of rows of the basis matrix is the dimension of the left null space.

Notes

Given an $m \times n$ matrix $\mathbf{A}$ over $\mathrm{GF}(q)$, the *left null space* of $\mathbf{A}$ is the vector space $\{\mathbf{x} \in \mathrm{GF}(q)^m\}$ that annihilates the rows of $\mathbf{A}$, i.e. $\mathbf{x}\mathbf{A} = \mathbf{0}$.

The left null space has properties $\mathcal{LN}(\mathbf{A}) = \mathcal{N}(\mathbf{A}^T)$ and $\dim(\mathcal{R}(\mathbf{A})) + \dim(\mathcal{LN}(\mathbf{A})) = m$.

Examples

The *left_null_space()* method defines basis vectors (its rows) that span the left null space of $\mathbf{A}$.

```
In [1]: m, n = 5, 3

In [2]: GF = galois.GF(31)

In [3]: A = GF.Random((m, n)); A
Out[3]:
GF([[ 2, 14, 29],
    [ 5, 26,  7],
    [28, 16, 13],
    [25, 15,  1],
    [28, 10, 17]], order=31)

In [4]: LN = A.left_null_space(); LN
Out[4]:
GF([[ 1,  0, 26, 25, 28],
    [ 0,  1, 19, 21, 13]], order=31)
```

The left null space is the set of vectors that sum the rows to 0.

```
In [5]: LN @ A
Out[5]:
GF([[0, 0, 0],
    [0, 0, 0]], order=31)
```

The dimension of the row space and left null space sum to m.

```
In [6]: R = A.row_space(); R
Out[6]:
GF([[1, 0, 0],
    [0, 1, 0],
    [0, 0, 1]], order=31)

In [7]: R.shape[0] + LN.shape[0] == m
Out[7]: True
```

`galois.FieldArray.lu_decompose()` → tuple[Self, Self]

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

Returns

- The lower triangular matrix.
- The upper triangular matrix.

Notes

The LU decomposition of $\mathbf{A}$ is defined as $\mathbf{A} = \mathbf{LU}$.

Examples

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

# Not every square matrix has an LU decomposition
In [2]: A = GF([[22, 11, 25, 11], [30, 27, 10, 3], [21, 16, 29, 7]]); A
Out[2]:
GF([[22, 11, 25, 11],
    [30, 27, 10,  3],
    [21, 16, 29,  7]], order=31)

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

In [4]: L
Out[4]:
GF([[ 1,  0,  0],
    [ 7,  1,  0],
    [ 8, 25,  1]], order=31)

In [5]: U
Out[5]:
GF([[22, 11, 25, 11],
    [ 0, 12, 21, 19],
    [ 0,  0, 17,  2]], order=31)

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

galois.FieldArray.**null_space**() → Self

Computes the null space of the matrix $\mathbf{A}$.

Returns

The null space basis matrix. The rows of the basis matrix are the basis vectors that span the null space. The number of rows of the basis matrix is the dimension of the null space.

Notes

Given an $m \times n$ matrix $\mathbf{A}$ over $\mathrm{GF}(q)$, the *null space* of $\mathbf{A}$ is the vector space $\{\mathbf{x} \in \mathrm{GF}(q)^n\}$ that annihilates the columns of $\mathbf{A}$, i.e. $\mathbf{Ax} = \mathbf{0}$.

The null space has properties $\mathcal{N}(\mathbf{A}) = \mathcal{LN}(\mathbf{A}^T)$ and $\dim(\mathcal{C}(\mathbf{A})) + \dim(\mathcal{N}(\mathbf{A})) = n$.

Examples

The *null_space()* method defines basis vectors (its rows) that span the null space of $\mathbf{A}$.

```
In [1]: m, n = 3, 5

In [2]: GF = galois.GF(31)

In [3]: A = GF.Random((m, n)); A
Out[3]:
GF([[ 1,  6,  7,  1, 29],
    [ 7, 26, 25,  2,  5],
    [18, 17, 30, 26, 12]], order=31)

In [4]: N = A.null_space(); N
Out[4]:
GF([[ 1,  0,  4, 29, 29],
    [ 0,  1, 14, 24,  2]], order=31)
```

The null space is the set of vectors that sum the columns to 0.

```
In [5]: A @ N.T
Out[5]:
GF([[0, 0],
    [0, 0],
    [0, 0]], order=31)
```

The dimension of the column space and null space sum to n.

```
In [6]: C = A.column_space(); C
Out[6]:
GF([[1, 0, 0],
    [0, 1, 0],
    [0, 0, 1]], order=31)

In [7]: C.shape[0] + N.shape[0] == n
Out[7]: True
```

galois.FieldArray.**plu_decompose**() → tuple[Self, Self, Self]

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

Returns

- The column permutation matrix.
- The lower triangular matrix.
- The upper triangular matrix.

Notes

The PLU decomposition of $\mathbf{A}$ is defined as $\mathbf{A} = \mathbf{PLU}$. This is equivalent to $\mathbf{P}^T\mathbf{A} = \mathbf{LU}$.

Examples

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

In [2]: A = GF([[0, 29, 2, 9], [20, 24, 5, 1], [2, 24, 1, 7]]); A
Out[2]:
GF([[ 0, 29,  2,  9],
    [20, 24,  5,  1],
    [ 2, 24,  1,  7]], order=31)

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

In [4]: P
Out[4]:
GF([[0, 1, 0],
    [1, 0, 0],
    [0, 0, 1]], order=31)

In [5]: L
Out[5]:
GF([[ 1,  0,  0],
    [ 0,  1,  0],
    [28, 14,  1]], order=31)

In [6]: U
Out[6]:
GF([[20, 24,  5,  1],
    [ 0, 29,  2,  9],
    [ 0,  0, 19,  8]], order=31)

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

In [8]: np.array_equal(P.T @ A, L @ U)
Out[8]: True
```

`galois.FieldArray.row_reduce(ncols: int | None = None, eye: 'left' | 'right' = 'left')` → Self

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

Parameters

ncols: int | None = None
The number of columns to perform Gaussian elimination over. The default is `None` which represents the number of columns of the matrix.

eye: 'left' | 'right' = 'left'
The location of the identity matrix $\mathbf{I}$, either on the left or the right.

Returns

The reduced row echelon form of the input matrix.

Notes

The elementary row operations in Gaussian elimination are:

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

Examples

Perform Gaussian elimination to get the reduced row echelon form of $\mathbf{A}$.

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

In [2]: A = GF([[16, 12, 1, 25], [1, 10, 27, 29], [1, 0, 3, 19]]); A
Out[2]:
GF([[16, 12,  1, 25],
    [ 1, 10, 27, 29],
    [ 1,  0,  3, 19]], order=31)

In [3]: A.row_reduce()
Out[3]:
GF([[ 1,  0,  0, 11],
    [ 0,  1,  0,  7],
    [ 0,  0,  1, 13]], order=31)

In [4]: np.linalg.matrix_rank(A)
Out[4]: 3
```

Perform Gaussian elimination to get an $\mathbf{I}$ on the right side of $\mathbf{A}$.

```
In [5]: A.row_reduce(eye="right")
Out[5]:
GF([[ 5,  1,  0,  0],
    [27,  0,  1,  0],
    [17,  0,  0,  1]], order=31)
```

Or only perform Gaussian elimination over 2 columns.

```
In [6]: A.row_reduce(ncols=2)
Out[6]:
GF([[ 1,  0,  5, 14],
    [ 0,  1, 27, 17],
    [ 0,  0, 29,  5]], order=31)
```

`galois.FieldArray.`**`row_space`**`()` → Self

Computes the row space of the matrix $\mathbf{A}$.

Returns

The row space basis matrix. The rows of the basis matrix are the basis vectors that span the row space. The number of rows of the basis matrix is the dimension of the row space.

Notes

Given an $m \times n$ matrix $\mathbf{A}$ over $\mathrm{GF}(q)$, the *row space* of $\mathbf{A}$ is the vector space $\{\mathbf{x} \in \mathrm{GF}(q)^n\}$ defined by all linear combinations of the rows of $\mathbf{A}$. The row space has at most dimension $\min(m, n)$.

The row space has properties $\mathcal{R}(\mathbf{A}) = \mathcal{C}(\mathbf{A}^T)$ and $\dim(\mathcal{R}(\mathbf{A})) + \dim(\mathcal{LN}(\mathbf{A})) = m$.

Examples

The `row_space()` method defines basis vectors (its rows) that span the row space of $\mathbf{A}$.

```
In [1]: m, n = 5, 3

In [2]: GF = galois.GF(31)

In [3]: A = GF.Random((m, n)); A
Out[3]:
GF([[14,  9,  8],
    [12, 27,  9],
    [12, 16, 22],
    [25, 30, 11],
    [12, 29,  1]], order=31)

In [4]: R = A.row_space(); R
Out[4]:
GF([[1, 0, 0],
    [0, 1, 0],
    [0, 0, 1]], order=31)
```

The dimension of the row space and left null space sum to m.

```
In [5]: LN = A.left_null_space(); LN
Out[5]:
GF([[ 1,  0, 11, 22,  4],
    [ 0,  1,  7,  0, 23]], order=31)

In [6]: R.shape[0] + LN.shape[0] == m
Out[6]: True
```

Properties

`class property` ***characteristic*** : int

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

`class property` ***degree*** : int

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

`class property` ***irreducible_poly*** : *Poly*

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

`class property` ***name*** : str

The finite field's name as a string `GF(p)` or `GF(p^m)`.

class property ***order*** : int

The order p^m of the Galois field $\mathrm{GF}(p^m)$.

class property ***prime_subfield*** : type[*FieldArray*]

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

class property galois.FieldArray.**characteristic** : int

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

Notes

Adding p copies of any element will always result in 0.

Examples

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

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

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

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

class property galois.FieldArray.**degree** : int

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

Notes

The degree is a positive integer. For prime fields, the degree is 1.

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
```

class property galois.FieldArray.**irreducible_poly** : *Poly*

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

Notes

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))
```

class property `galois.FieldArray.name` : str

The finite field's name as a string `GF(p)` or `GF(p^m)`.

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)'
```

class property `galois.FieldArray.order` : int

The order p^m of the Galois field $\mathrm{GF}(p^m)$.

Notes

The order of the field is 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
```

class property galois.FieldArray.**prime_subfield** : type[*FieldArray*]

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

Notes

For the prime field $\mathrm{GF}(p)$, the prime subfield is itself.

Examples

```
In [1]: galois.GF(2).prime_subfield
Out[1]: <class 'galois.GF(2)'>

In [2]: galois.GF(2**8).prime_subfield
Out[2]: <class 'galois.GF(2)'>

In [3]: galois.GF(31).prime_subfield
Out[3]: <class 'galois.GF(31)'>

In [4]: galois.GF(7**5).prime_subfield
Out[4]: <class 'galois.GF(7)'>
```

Attributes

class property ***is_extension_field*** : bool

Indicates if the finite field is an extension field, having prime power order.

class property ***is_prime_field*** : bool

Indicates if the finite field is a prime field, having prime order.

class property ***is_primitive_poly*** : bool

Indicates whether the *irreducible_poly* is a primitive polynomial.

is_square() → bool | ndarray

Determines if the elements of x are squares in the finite field.

class property galois.FieldArray.**is_extension_field** : bool

Indicates if the finite field is an extension field, having prime power order.

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
```

class property galois.FieldArray.**is_prime_field** : bool

Indicates if the finite field is a prime field, having prime order.

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

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

class property galois.FieldArray.**is_primitive_poly** : bool

Indicates whether the *irreducible_poly* is a primitive polynomial.

Notes

If the irreducible polynomial is primitive, then x is a primitive element of the finite field.

Examples

The default $\mathrm{GF}(2^8)$ field uses a primitive polynomial.

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

In [2]: print(GF.properties)
Galois Field:
  name: GF(2^8)
  characteristic: 2
  degree: 8
  order: 256
```

(continues on next page)

(continued from previous page)

```
  irreducible_poly: x^8 + x^4 + x^3 + x^2 + 1
  is_primitive_poly: True
  primitive_element: x

In [3]: GF.is_primitive_poly
Out[3]: True
```

The $\mathrm{GF}(2^8)$ field from AES uses a non-primitive polynomial.

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

In [5]: print(GF.properties)
Galois Field:
  name: 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 [6]: GF.is_primitive_poly
Out[6]: False
```

galois.FieldArray.**is_square**() → bool | ndarray

Determines if the elements of x are squares in the finite field.

Returns

A boolean array indicating if each element in x is a square. The return value is a single boolean if the input array x is a scalar.

See also

squares, *non_squares*

Notes

An element x in $\mathrm{GF}(p^m)$ is a *square* if there exists a y such that $y^2 = x$ in the field.

In fields with characteristic 2, every element is a square (with two identical square roots). In fields with characteristic greater than 2, exactly half of the nonzero elements are squares (with two unique square roots).

References

- Section 3.5.1 from https://cacr.uwaterloo.ca/hac/about/chap3.pdf.

Examples

Since $\mathrm{GF}(2^3)$ has characteristic 2, every element has a square root.

Integer

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

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

In [3]: x.is_square()
Out[3]: array([ True,  True,  True,  True,  True,  True,  True,  True])
```

Polynomial

```
In [4]: GF = galois.GF(2**3, repr="poly")

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

In [6]: x.is_square()
Out[6]: array([ True,  True,  True,  True,  True,  True,  True,  True])
```

Power

```
In [7]: GF = galois.GF(2**3, repr="power")

In [8]: x = GF.elements; x
Out[8]: GF([  0,   1,    , ^3, ^2, ^6, ^4, ^5], order=2^3)

In [9]: x.is_square()
Out[9]: array([ True,  True,  True,  True,  True,  True,  True,  True])
```

In $\mathrm{GF}(11)$, the characteristic is greater than 2 so only half of the elements have square roots.

Integer

```
In [10]: GF = galois.GF(11)

In [11]: x = GF.elements; x
Out[11]: GF([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10], order=11)

In [12]: x.is_square()
Out[12]:
array([ True,  True, False,  True,  True,  True, False, False, False,
        True, False])
```

Power

```
In [13]: GF = galois.GF(11, repr="power")

In [14]: x = GF.elements; x
Out[14]: GF([  0,   1,    , ^8, ^2, ^4, ^9, ^7, ^3, ^6, ^5], order=11)

In [15]: x.is_square()
Out[15]:
array([ True,  True, False,  True,  True,  True, False, False, False,
        True, False])
```

class galois.**GF2**(*galois.FieldArray*)

A *FieldArray* subclass over GF(2).

Info

This class is a pre-generated *FieldArray* subclass generated with `galois.GF(2)` and is included in the API for convenience.

Examples

This class is equivalent, and in fact identical, to the *FieldArray* subclass returned from the class factory *GF()*.

```
In [1]: galois.GF2 is galois.GF(2)
Out[1]: True

In [2]: issubclass(galois.GF2, galois.FieldArray)
Out[2]: True

In [3]: print(galois.GF2.properties)
Galois Field:
  name: GF(2)
  characteristic: 2
  degree: 1
  order: 2
  irreducible_poly: x + 1
```

(continues on next page)

(continued from previous page)

```
  is_primitive_poly: True
  primitive_element: 1
```

Create a `FieldArray` instance using `GF2`'s constructor.

```
In [4]: x = galois.GF2([1, 0, 1, 1]); x
Out[4]: GF([1, 0, 1, 1], order=2)

In [5]: isinstance(x, galois.GF2)
Out[5]: True
```

Constructors

GF2(x: ElementLike | ArrayLike, dtype: DTypeLike | *None* = **None**, ...)
: Creates an array over $\mathrm{GF}(p^m)$.

classmethod ***Identity***(size: *int*, ...) → Self
: Creates an $n \times n$ identity matrix.

classmethod ***Ones***(shape: ShapeLike, ...) → Self
: Creates an array of all ones.

classmethod ***Random***(shape: ShapeLike = *()*, ...) → Self
: Creates an array with random elements.

classmethod ***Range***(start: ElementLike, stop, ...) → Self
: Creates a 1-D array with a range of elements.

classmethod ***Vandermonde***(element: ElementLike, rows, ...) → Self
: Creates an $m \times n$ Vandermonde matrix of $a \in \mathrm{GF}(q)$.

classmethod ***Zeros***(shape: ShapeLike, ...) → Self
: Creates an array of all zeros.

Conversions

classmethod ***Vector***(array: ArrayLike, ...) → *FieldArray*
: Converts length-m vectors over the prime subfield $\mathrm{GF}(p)$ to an array over $\mathrm{GF}(p^m)$.

vector(dtype: DTypeLike | *None* = **None**) → *FieldArray*
: Converts an array over $\mathrm{GF}(p^m)$ to length-m vectors over the prime subfield $\mathrm{GF}(p)$.

Elements

class property ***elements*** : *FieldArray*
: All of the finite field's elements $\{0, \dots, p^m - 1\}$.

class property ***non_squares*** : *FieldArray*
: All non-squares in the Galois field.

class property ***primitive_element*** : *FieldArray*
: A primitive element α of the Galois field $\mathrm{GF}(p^m)$.

`class property` ***primitive_elements*** : *FieldArray*

All primitive elements α of the Galois field $\mathrm{GF}(p^m)$.

`classmethod` ***primitive_root_of_unity***(n: *int*) → Self

Finds a primitive n-th root of unity in the finite field.

`classmethod` ***primitive_roots_of_unity***(n: *int*) → Self

Finds all primitive n-th roots of unity in the finite field.

`class property` ***squares*** : *FieldArray*

All squares in the finite field.

`class property` ***units*** : *FieldArray*

All of the finite field's units $\{1, \dots, p^m - 1\}$.

String representation

__repr__() → str

Displays the array specifying the class and finite field order.

__str__() → str

Displays the array without specifying the class or finite field order.

`classmethod` ***arithmetic_table***(operation, ...) → str

Generates the specified arithmetic table for the finite field.

`class property` ***properties*** : str

A formatted string of relevant properties of the Galois field.

`classmethod` ***repr_table***(...) → str

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

Element representation

`class property` ***element_repr*** : Literal[int] | Literal[poly] | Literal[power]

The current finite field element representation.

`classmethod` ***repr***(...) → Generator[None, None, None]

Sets the element representation for all arrays from this `FieldArray` subclass.

Arithmetic compilation

`class property` ***default_ufunc_mode*** : Literal[jit - lookup] | typing.Literal[jit - calculate] | typing.Literal[python - calculate]

The default ufunc compilation mode for this `FieldArray` subclass.

`class property` ***dtypes*** : list[np.dtype]

List of valid integer `numpy.dtype` values that are compatible with this finite field.

`class property` ***ufunc_mode*** : Literal[jit - lookup] | typing.Literal[jit - calculate] | typing.Literal[python - calculate]

The current ufunc compilation mode for this `FieldArray` subclass.

`class property` ***ufunc_modes*** : list[str]

All supported ufunc compilation modes for this *FieldArray* subclass.

`classmethod` ***compile***(mode)

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

Methods

additive_order() → int | ndarray

Computes the additive order of each element in x.

characteristic_poly() → *Poly*

Computes the characteristic polynomial of a finite field element a or a square matrix $\mathbf{A}$.

field_norm() → *FieldArray*

Computes the field norm $\mathrm{N}_{L/K}(x)$ of the elements of x.

field_trace() → *FieldArray*

Computes the field trace $\mathrm{Tr}_{L/K}(x)$ of the elements of x.

log(base: ElementLike | ArrayLike | *None* = **None**) → int | ndarray

Computes the discrete logarithm of the array x base β.

minimal_poly() → *Poly*

Computes the minimal polynomial of a finite field element a.

multiplicative_order() → int | ndarray

Computes the multiplicative order $\mathrm{ord}(x)$ of each element in x.

Linear algebra

column_space() → Self

Computes the column space of the matrix $\mathbf{A}$.

left_null_space() → Self

Computes the left null space of the matrix $\mathbf{A}$.

lu_decompose() → tuple[Self, Self]

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

null_space() → Self

Computes the null space of the matrix $\mathbf{A}$.

plu_decompose() → tuple[Self, Self, Self]

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

row_reduce(ncols: *int* | *None* = **None**, ...) → Self

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

row_space() → Self

Computes the row space of the matrix $\mathbf{A}$.

Properties

class property ***characteristic*** : int

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

class property ***degree*** : int

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

class property ***irreducible_poly*** : *Poly*

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

class property ***name*** : str

The finite field's name as a string `GF(p)` or `GF(p^m)`.

class property ***order*** : int

The order p^m of the Galois field $\mathrm{GF}(p^m)$.

class property ***prime_subfield*** : type[*FieldArray*]

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

Attributes

class property ***is_extension_field*** : bool

Indicates if the finite field is an extension field, having prime power order.

class property ***is_prime_field*** : bool

Indicates if the finite field is a prime field, having prime order.

class property ***is_primitive_poly*** : bool

Indicates whether the *irreducible_poly* is a primitive polynomial.

is_square() → bool | ndarray

Determines if the elements of x are squares in the finite field.

galois.**Field**(*order: int*, *, *irreducible_poly:* PolyLike | *None* = **None**, *primitive_element: int* | PolyLike | *None* = **None**, *verify: bool* = **True**, *compile: 'auto'* | *'jit-lookup'* | *'jit-calculate'* | *'python-calculate'* | *None* = **None**, *repr: 'int'* | *'poly'* | *'power'* | *None* = **None**) → type[*FieldArray*]

galois.**Field**(*characteristic: int*, *degree: int*, *, *irreducible_poly:* PolyLike | *None* = **None**, *primitive_element: int* | PolyLike | *None* = **None**, *verify: bool* = **True**, *compile: 'auto'* | *'jit-lookup'* | *'jit-calculate'* | *'python-calculate'* | *None* = **None**, *repr: 'int'* | *'poly'* | *'power'* | *None* = **None**) → type[*FieldArray*]

Alias of *GF()*.

galois.**GF**(order: *int*, *, irreducible_poly: PolyLike | *None* = **None**, primitive_element: *int* | PolyLike | *None* = **None**, verify: *bool* = **True**, compile: *'auto'* | *'jit-lookup'* | *'jit-calculate'* | *'python-calculate'* | *None* = **None**, repr: *'int'* | *'poly'* | *'power'* | *None* = **None**) → type[*FieldArray*]

galois.**GF**(characteristic: *int*, degree: *int*, *, irreducible_poly: PolyLike | *None* = **None**, primitive_element: *int* | PolyLike | *None* = **None**, verify: *bool* = **True**, compile: *'auto'* | *'jit-lookup'* | *'jit-calculate'* | *'python-calculate'* | *None* = **None**, repr: *'int'* | *'poly'* | *'power'* | *None* = **None**) → type[*FieldArray*]

Creates a *FieldArray* subclass 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.

characteristic: int
The characteristic p of the field $\mathrm{GF}(p^m)$. The characteristic must be prime.

degree: int
The degree m of the field $\mathrm{GF}(p^m)$. The degree must be a positive integer.

irreducible_poly: *PolyLike* | None = None
Optionally specify an irreducible polynomial of degree m over $\mathrm{GF}(p)$ that defines the finite field arithmetic. The default is `None` which uses the Conway polynomial $C_{p,m}$, see `conway_poly()`.

primitive_element: int | *PolyLike* | None = None

Optionally specify a primitive element of the field. This value is used when building the exponential and logarithm lookup tables and as the base of `numpy.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. The default is `None` which uses `primitive_root()`.

For extension fields $\mathrm{GF}(p^m)$, the primitive element is a polynomial of degree less than m over $\mathrm{GF}(p)$. The default is `None` which uses `primitive_element()`.

verify: bool = True

Indicates whether to verify that the user-provided irreducible polynomial is in fact irreducible and that the user-provided 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 while to verify), this argument may be set to `False`.

The default irreducible polynomial and primitive element are never verified because they are already known to be irreducible and a multiplicative generator, respectively.

compile: 'auto' | 'jit-lookup' | 'jit-calculate' | 'python-calculate' | None = None

The ufunc calculation mode. This can be modified after class construction with the `compile()` method. See *Compilation Modes* for a further discussion.

- `None` (default): For a newly-created `FieldArray` subclass, `None` corresponds to `"auto"`. If the `FieldArray` subclass already exists, `None` does not modify its current 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).

repr: 'int' | 'poly' | 'power' | None = None

The field element representation. This can be modified after class construction with the `repr()` method. See *Element Representation* for a further discussion.

- `None` (default): For a newly-created `FieldArray` subclass, `None` corresponds to `"int"`. If the `FieldArray` subclass already exists, `None` does not modify its current element representation.
- `"int"`: Sets the element representation to the *integer representation*.
- `"poly"`: Sets the element representation to the *polynomial representation*.
- `"power"`: Sets the element representation to the *power representation*.

Returns

A `FieldArray` subclass for $\mathrm{GF}(p^m)$.

Notes

`FieldArray` subclasses of the same type (order, irreducible polynomial, and primitive element) are singletons. So, calling this class factory with arguments that correspond to the same subclass will return the same class object.

Examples

Create a `FieldArray` subclass for each type of finite field.

GF(2)

Construct the binary field.

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

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

GF(p)

Construct a prime field.

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

In [4]: print(GF.properties)
Galois Field:
  name: GF(31)
  characteristic: 31
  degree: 1
```

(continues on next page)

(continued from previous page)

```
  order: 31
  irreducible_poly: x + 28
  is_primitive_poly: True
  primitive_element: 3
```

GF(2^m)

Construct a binary extension field. Notice the default irreducible polynomial is primitive and x is a primitive element.

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

In [6]: print(GF.properties)
Galois Field:
  name: 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
```

GF(p^m)

Construct a prime extension field. Notice the default irreducible polynomial is primitive and x is a primitive element.

```
In [7]: GF = galois.GF(3**5)

In [8]: print(GF.properties)
Galois Field:
  name: GF(3^5)
  characteristic: 3
  degree: 5
  order: 243
  irreducible_poly: x^5 + 2x + 1
  is_primitive_poly: True
  primitive_element: x
```

Create a *FieldArray* subclass for extension fields and specify their irreducible polynomials.

GF(2^m)

Construct the $\mathrm{GF}(2^8)$ field that is used in AES. Notice the irreducible polynomial is not primitive and x is not a primitive element.

```
In [9]: GF = galois.GF(2**8, irreducible_poly="x^8 + x^4 + x^3 + x + 1")

In [10]: print(GF.properties)
Galois Field:
  name: 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
```

GF(p^m)

Construct $\mathrm{GF}(3^5)$ with an irreducible, but not primitive, polynomial. Notice that x is not a primitive element.

```
In [11]: GF = galois.GF(3**5, irreducible_poly="x^5 + 2x + 2")

In [12]: print(GF.properties)
Galois Field:
  name: GF(3^5)
  characteristic: 3
  degree: 5
  order: 243
  irreducible_poly: x^5 + 2x + 2
  is_primitive_poly: False
  primitive_element: 2x
```

Finite fields with arbitrarily-large orders are supported.

GF(p)

Construct a large prime field.

```
In [13]: GF = galois.GF(36893488147419103183)

In [14]: print(GF.properties)
Galois Field:
  name: GF(36893488147419103183)
  characteristic: 36893488147419103183
  degree: 1
  order: 36893488147419103183
  irreducible_poly: x + 36893488147419103180
  is_primitive_poly: True
  primitive_element: 3
```

GF(2^m)

Construct a large binary extension field.

```
In [15]: GF = galois.GF(2**100)

In [16]: print(GF.properties)
Galois Field:
  name: 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
```

The construction of large fields can be sped up by explicitly specifying p and m. This avoids the need to factor the order p^m.

```
In [17]: GF = galois.GF(2, 100)

In [18]: print(GF.properties)
Galois Field:
  name: 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
```

GF(p^m)

Construct a large prime extension field.

```
In [19]: GF = galois.GF(109987**4)

In [20]: print(GF.properties)
Galois Field:
  name: GF(109987^4)
  characteristic: 109987
  degree: 4
  order: 146340800268433348561
  irreducible_poly: x^4 + 3x^2 + 100525x + 3
  is_primitive_poly: True
  primitive_element: x
```

The construction of large fields can be sped up by explicitly specifying p and m. This avoids the need to factor the order p^m.

```
In [21]: GF = galois.GF(109987, 4)

In [22]: print(GF.properties)
Galois Field:
  name: GF(109987^4)
  characteristic: 109987
  degree: 4
  order: 146340800268433348561
  irreducible_poly: x^4 + 3x^2 + 100525x + 3
  is_primitive_poly: True
  primitive_element: x
```

3.19.1 Primitive elements

`galois.primitive_element(irreducible_poly: Poly, ...) → Poly`

Finds a primitive element g of the Galois field $\mathrm{GF}(q^m)$ with degree-m irreducible polynomial $f(x)$ over $\mathrm{GF}(q)$.

`galois.primitive_elements(irreducible_poly: Poly) → list[Poly]`

Finds all primitive elements g of the Galois field $\mathrm{GF}(q^m)$ with degree-m irreducible polynomial $f(x)$ over $\mathrm{GF}(q)$.

`galois.is_primitive_element(element: PolyLike, ...) → bool`

Determines if g is a primitive element of the Galois field $\mathrm{GF}(q^m)$ with degree-m irreducible polynomial $f(x)$ over $\mathrm{GF}(q)$.

`galois.primitive_element(irreducible_poly: Poly, method: Literal[min] | Literal[max] | Literal[random] = 'min') → Poly`

Finds a primitive element g of the Galois field $\mathrm{GF}(q^m)$ with degree-m irreducible polynomial $f(x)$ over $\mathrm{GF}(q)$.

Parameters

irreducible_poly: *Poly*
The degree-m irreducible polynomial $f(x)$ over $\mathrm{GF}(q)$ that defines the extension field $\mathrm{GF}(q^m)$.

method: Literal[min] | Literal[max] | Literal[random] = 'min'
The search method for finding the primitive element.

Returns

A primitive element g of $\mathrm{GF}(q^m)$ with irreducible polynomial $f(x)$. The primitive element g is a polynomial over $\mathrm{GF}(q)$ with degree less than m.

See also

is_primitive_element, *FieldArray.primitive_element*

Examples

Construct the extension field $\mathrm{GF}(7^5)$.

```
In [1]: f = galois.irreducible_poly(7, 5, method="max"); f
Out[1]: Poly(x^5 + 6x^4 + 6x^3 + 6x^2 + 6x + 6, GF(7))

In [2]: GF = galois.GF(7**5, irreducible_poly=f, repr="poly")

In [3]: print(GF.properties)
Galois Field:
  name: GF(7^5)
  characteristic: 7
  degree: 5
  order: 16807
  irreducible_poly: x^5 + 6x^4 + 6x^3 + 6x^2 + 6x + 6
  is_primitive_poly: False
  primitive_element: x + 3
```

Find the smallest primitive element for the degree-5 extension of $\mathrm{GF}(7)$ with irreducible polynomial $f(x)$.

```
In [4]: g = galois.primitive_element(f); g
Out[4]: Poly(x + 3, GF(7))

# Convert the polynomial over GF(7) into an element of GF(7^5)
In [5]: g = GF(int(g)); g
Out[5]: GF(x + 3, order=7^5)

In [6]: g.multiplicative_order() == GF.order - 1
Out[6]: True
```

Find the largest primitive element for the degree-5 extension of $\mathrm{GF}(7)$ with irreducible polynomial $f(x)$.

```
In [7]: g = galois.primitive_element(f, method="max"); g
Out[7]: Poly(6x^4 + 6x^3 + 6x^2 + 6x + 3, GF(7))

# Convert the polynomial over GF(7) into an element of GF(7^5)
In [8]: g = GF(int(g)); g
Out[8]: GF(6x^4 + 6x^3 + 6x^2 + 6x + 3, order=7^5)

In [9]: g.multiplicative_order() == GF.order - 1
Out[9]: True
```

Find a random primitive element for the degree-5 extension of $\mathrm{GF}(7)$ with irreducible polynomial $f(x)$.

```
In [10]: g = galois.primitive_element(f, method="random"); g
Out[10]: Poly(x^4 + 5x^3 + 3x^2 + 4x + 2, GF(7))

# Convert the polynomial over GF(7) into an element of GF(7^5)
In [11]: g = GF(int(g)); g
Out[11]: GF(x^4 + 5x^3 + 3x^2 + 4x + 2, order=7^5)

In [12]: g.multiplicative_order() == GF.order - 1
Out[12]: True
```

galois.**primitive_elements**(irreducible_poly: Poly) → list[*Poly*]

Finds all primitive elements g of the Galois field $\mathrm{GF}(q^m)$ with degree-m irreducible polynomial $f(x)$ over $\mathrm{GF}(q)$.

Parameters

irreducible_poly: *Poly*
The degree-m irreducible polynomial $f(x)$ over $\mathrm{GF}(q)$ that defines the extension field $\mathrm{GF}(q^m)$.

Returns
List of all primitive elements of $\mathrm{GF}(q^m)$ with irreducible polynomial $f(x)$. Each primitive element g is a polynomial over $\mathrm{GF}(q)$ with degree less than m.

See also

is_primitive_element, *FieldArray.primitive_elements*

Notes

The number of primitive elements of $\mathrm{GF}(q^m)$ is $\phi(q^m - 1)$, where $\phi(n)$ is the Euler totient function. See *euler_phi*.

Examples

Construct the extension field $\mathrm{GF}(3^4)$.

```
In [1]: f = galois.irreducible_poly(3, 4, method="max"); f
Out[1]: Poly(x^4 + 2x^3 + 2x^2 + x + 2, GF(3))

In [2]: GF = galois.GF(3**4, irreducible_poly=f, repr="poly")

In [3]: print(GF.properties)
Galois Field:
  name: GF(3^4)
  characteristic: 3
  degree: 4
  order: 81
  irreducible_poly: x^4 + 2x^3 + 2x^2 + x + 2
  is_primitive_poly: True
  primitive_element: x
```

Find all primitive elements for the degree-4 extension of $\mathrm{GF}(3)$.

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

(continues on next page)

(continued from previous page)

```
Poly(2x^2 + 2, GF(3)),
Poly(2x^2 + x + 1, GF(3)),
Poly(x^3, GF(3)),
Poly(x^3 + 1, GF(3)),
Poly(x^3 + x^2, GF(3)),
Poly(x^3 + x^2 + 2, GF(3)),
Poly(x^3 + x^2 + x, GF(3)),
Poly(x^3 + x^2 + 2x, GF(3)),
Poly(x^3 + x^2 + 2x + 2, GF(3)),
Poly(x^3 + 2x^2, GF(3)),
Poly(x^3 + 2x^2 + 2, GF(3)),
Poly(x^3 + 2x^2 + x, GF(3)),
Poly(x^3 + 2x^2 + x + 1, GF(3)),
Poly(x^3 + 2x^2 + 2x + 1, GF(3)),
Poly(2x^3, GF(3)),
Poly(2x^3 + 2, GF(3)),
Poly(2x^3 + x^2, GF(3)),
Poly(2x^3 + x^2 + 1, GF(3)),
Poly(2x^3 + x^2 + x + 2, GF(3)),
Poly(2x^3 + x^2 + 2x, GF(3)),
Poly(2x^3 + x^2 + 2x + 2, GF(3)),
Poly(2x^3 + 2x^2, GF(3)),
Poly(2x^3 + 2x^2 + 1, GF(3)),
Poly(2x^3 + 2x^2 + x, GF(3)),
Poly(2x^3 + 2x^2 + x + 1, GF(3)),
Poly(2x^3 + 2x^2 + 2x, GF(3))]
```

The number of primitive elements is given by $\phi(q^m - 1)$.

```
In [5]: phi = galois.euler_phi(3**4 - 1); phi
Out[5]: 16

In [6]: len(g) == phi
Out[6]: False
```

Shows that each primitive element has an order of $q^m - 1$.

```
# Convert the polynomials over GF(3) into elements of GF(3^4)
In [7]: g = GF([int(gi) for gi in g]); g
Out[7]:
GF([                  ,                + 1,                    2,
                 2 + 2,              ^2 + 1,          ^2 + 2 + 2,
               2^2 + 2,        2^2 +   + 1,                   ^3,
                ^3 + 1,             ^3 + ^2,        ^3 + ^2 + 2,
          ^3 + ^2 +  ,       ^3 + ^2 + 2,   ^3 + ^2 + 2 + 2,
             ^3 + 2^2,        ^3 + 2^2 + 2,       ^3 + 2^2 +  ,
     ^3 + 2^2 +   + 1, ^3 + 2^2 + 2 + 1,                 2^3,
               2^3 + 2,            2^3 + ^2,        2^3 + ^2 + 1,
     2^3 + ^2 +   + 2,      2^3 + ^2 + 2, 2^3 + ^2 + 2 + 2,
            2^3 + 2^2,      2^3 + 2^2 + 1,      2^3 + 2^2 +  ,
    2^3 + 2^2 +   + 1,     2^3 + 2^2 + 2], order=3^4)
```

(continues on next page)

(continued from previous page)

```
In [8]: np.all(g.multiplicative_order() == GF.order - 1)
Out[8]: True
```

`galois.is_primitive_element(element: PolyLike, irreducible_poly: Poly) → bool`

Determines if g is a primitive element of the Galois field $\mathrm{GF}(q^m)$ with degree-m irreducible polynomial $f(x)$ over $\mathrm{GF}(q)$.

Parameters

element: *PolyLike*
An element g of $\mathrm{GF}(q^m)$ is a polynomial over $\mathrm{GF}(q)$ with degree less than m.

irreducible_poly: *Poly*
The degree-m irreducible polynomial $f(x)$ over $\mathrm{GF}(q)$ that defines the extension field $\mathrm{GF}(q^m)$.

Returns
`True` if g is a primitive element of $\mathrm{GF}(q^m)$.

See also

primitive_element, *FieldArray.primitive_element*

Examples

In the extension field $\mathrm{GF}(3^4)$, the element $x + 2$ is a primitive element whose order is $3^4 - 1 = 80$.

```
In [1]: GF = galois.GF(3**4)

In [2]: f = GF.irreducible_poly; f
Out[2]: Poly(x^4 + 2x^3 + 2, GF(3))

In [3]: galois.is_primitive_element("x + 2", f)
Out[3]: True

In [4]: GF("x + 2").multiplicative_order()
Out[4]: 80
```

However, the element $x + 1$ is not a primitive element, as noted by its order being only 20.

```
In [5]: galois.is_primitive_element("x + 1", f)
Out[5]: False

In [6]: GF("x + 1").multiplicative_order()
Out[6]: 20
```

3.20 Polynomials

`class galois.Poly`

A univariate polynomial $f(x)$ over $\mathrm{GF}(p^m)$.

`galois.typing.PolyLike`

A `Union` representing objects that can be coerced into a polynomial.

`class galois.Poly`

A univariate polynomial $f(x)$ over $\mathrm{GF}(p^m)$.

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))
```

Create a polynomial over $\mathrm{GF}(3^5)$.

```
In [2]: GF = galois.GF(3**5)

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

See *Polynomials* and *Polynomial Arithmetic* for more examples.

Constructors

Poly(coeffs: ArrayLike, field: *type[*Array*]* | *None* = **None**, ...)

Creates a polynomial $f(x)$ over $\mathrm{GF}(p^m)$.

`classmethod` **Degrees**(degrees: *Sequence[int]* | *ndarray*, ...) → Self

Constructs a polynomial over $\mathrm{GF}(p^m)$ from its non-zero degrees.

`classmethod` **Identity**(field: *type[*Array*]* | *None* = **None**) → Self

Constructs the polynomial $f(x) = x$ over $\mathrm{GF}(p^m)$.

`classmethod` **Int**(integer: *int*, ...) → Self

Constructs a polynomial over $\mathrm{GF}(p^m)$ from its integer representation.

`classmethod` **One**(field: *type[*Array*]* | *None* = **None**) → Self

Constructs the polynomial $f(x) = 1$ over $\mathrm{GF}(p^m)$.

`classmethod` **Random**(degree: *int*, ...) → Self

Constructs a random polynomial over $\mathrm{GF}(p^m)$ with degree d.

`classmethod` **Roots**(roots: ArrayLike, ...) → Self

Constructs a monic polynomial over $\mathrm{GF}(p^m)$ from its roots.

`classmethod` **Str**(string: *str*, ...) → Self

Constructs a polynomial over $\mathrm{GF}(p^m)$ from its string representation.

`classmethod Zero(field: type[Array] | None = None) → Self`

Constructs the polynomial $f(x) = 0$ over $\mathrm{GF}(p^m)$.

`galois.Poly(coeffs: ArrayLike, field: type[Array] | None = None, order: 'desc' | 'asc' = 'desc')`

Creates a polynomial $f(x)$ over $\mathrm{GF}(p^m)$.

The polynomial $f(x) = a_d x^d + a_{d-1} x^{d-1} + \cdots + a_1 x + a_0$ with degree d has coefficients $\{a_d, a_{d-1}, \dots, a_1, a_0\}$ in $\mathrm{GF}(p^m)$.

Parameters

coeffs: *ArrayLike*
The polynomial coefficients $\{a_d, a_{d-1}, \dots, a_1, a_0\}$.

field: type[*Array*] | None = None

The Galois field $\mathrm{GF}(p^m)$ the polynomial is over.

- `None` (default): If the coefficients are an *Array*, 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)`.
- *Array* subclass: The coefficients are explicitly converted to this Galois field using `field(coeffs)`.

order: 'desc' | 'asc' = 'desc'

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. $\{a_0, a_1, \dots, a_{d-1}, a_d\}$.

`classmethod galois.Poly.Degrees(degrees: Sequence[int] | ndarray, coeffs: ArrayLike | None = None, field: type[Array] | None = None) → Self`

Constructs a polynomial over $\mathrm{GF}(p^m)$ from its non-zero degrees.

Parameters

degrees: Sequence[int] | ndarray
The polynomial degrees with non-zero coefficients.

coeffs: *ArrayLike* | None = None
The corresponding non-zero polynomial coefficients. The default is `None` which corresponds to all ones.

field: type[*Array*] | None = None

The Galois field $\mathrm{GF}(p^m)$ the polynomial is over.

- `None` (default): If the coefficients are an *Array*, 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)`.
- *Array* subclass: The coefficients are explicitly converted to this Galois field using `field(coeffs)`.

Returns

The polynomial $f(x)$.

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 $\mathrm{GF}(3^5)$ by specifying the degrees with non-zero coefficients and their coefficient values.

```
In [2]: GF = galois.GF(3**5)

In [3]: galois.Poly.Degrees([3, 1, 0], coeffs=[214, 73, 185], field=GF)
Out[3]: Poly(214x^3 + 73x + 185, GF(3^5))
```

classmethod galois.Poly.Identity(*field: type[Array] | None = None*) → Self

Constructs the polynomial $f(x) = x$ over $\mathrm{GF}(p^m)$.

Parameters

field: type[*Array*] | None = None
The Galois field $\mathrm{GF}(p^m)$ the polynomial is over. The default is `None` which corresponds to *GF2*.

Returns
The polynomial $f(x) = x$.

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 $\mathrm{GF}(3^5)$.

```
In [2]: GF = galois.GF(3**5)

In [3]: galois.Poly.Identity(GF)
Out[3]: Poly(x, GF(3^5))
```

classmethod galois.Poly.Int(*integer: int, field: type[Array] | None = None*) → Self

Constructs a polynomial over $\mathrm{GF}(p^m)$ from its integer representation.

Parameters

integer: int
The integer representation of the polynomial $f(x)$.

field: type[*Array*] | None = None
The Galois field $\mathrm{GF}(p^m)$ the polynomial is over. The default is `None` which corresponds to *GF2*.

Returns
The polynomial $f(x)$.

Notes

Int() and *__int__()* are inverse operations.

Examples

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

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

In [2]: int(f)
Out[2]: 5
```

Construct a polynomial over $\mathrm{GF}(3^5)$ from its integer representation.

```
In [3]: GF = galois.GF(3**5)

In [4]: f = galois.Poly.Int(186535908, field=GF); f
Out[4]: Poly(13x^3 + 117, GF(3^5))

In [5]: int(f)
Out[5]: 186535908

# The polynomial/integer equivalence
In [6]: int(f) == 13*GF.order**3 + 117
Out[6]: True
```

Construct a polynomial over $\mathrm{GF}(2)$ from its binary string.

```
In [7]: f = galois.Poly.Int(int("0b1011", 2)); f
Out[7]: Poly(x^3 + x + 1, GF(2))

In [8]: bin(f)
Out[8]: '0b1011'
```

Construct a polynomial over $\mathrm{GF}(2^3)$ from its octal string.

```
In [9]: GF = galois.GF(2**3)

In [10]: f = galois.Poly.Int(int("0o5034", 8), field=GF); f
Out[10]: Poly(5x^3 + 3x + 4, GF(2^3))

In [11]: oct(f)
Out[11]: '0o5034'
```

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

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

In [13]: f = galois.Poly.Int(int("0xf700a275", 16), field=GF); f
Out[13]: Poly(247x^3 + 162x + 117, GF(2^8))
```

(continues on next page)

(continued from previous page)

```
In [14]: hex(f)
Out[14]: '0xf700a275'
```

classmethod galois.Poly.**One**(field: *type[Array] | None* = ***None***) → Self

Constructs the polynomial $f(x) = 1$ over $\mathrm{GF}(p^m)$.

Parameters

field: type[*Array*] | None = None
The Galois field $\mathrm{GF}(p^m)$ the polynomial is over. The default is `None` which corresponds to *GF2*.

Returns
The polynomial $f(x) = 1$.

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 $\mathrm{GF}(3^5)$.

```
In [2]: GF = galois.GF(3**5)

In [3]: galois.Poly.One(GF)
Out[3]: Poly(1, GF(3^5))
```

classmethod galois.Poly.**Random**(degree: *int*, seed: *int | integer | Generator | None* = ***None***, field: *type[Array] | None* = ***None***) → Self

Constructs a random polynomial over $\mathrm{GF}(p^m)$ with degree d.

Parameters

degree: int
The degree of the polynomial.

seed: int | integer | Generator | None = None
Non-negative integer used to initialize the PRNG. The default is `None` which means that unpredictable entropy will be pulled from the OS to be used as the seed. A `numpy.random.Generator` can also be passed.

field: type[*Array*] | None = None
The Galois field $\mathrm{GF}(p^m)$ the polynomial is over. The default is `None` which corresponds to *GF2*.

Returns
The polynomial $f(x)$.

Examples

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

```
In [1]: galois.Poly.Random(5)
Out[1]: Poly(x^5 + x^4 + x^3, GF(2))
```

Construct a random degree-5 polynomial over $\mathrm{GF}(3^5)$ with a given seed. This produces repeatable results.

```
In [2]: GF = galois.GF(3**5)

In [3]: galois.Poly.Random(5, seed=123456789, field=GF)
Out[3]: Poly(56x^5 + 228x^4 + 157x^3 + 218x^2 + 148x + 43, GF(3^5))

In [4]: galois.Poly.Random(5, seed=123456789, field=GF)
Out[4]: Poly(56x^5 + 228x^4 + 157x^3 + 218x^2 + 148x + 43, GF(3^5))
```

Construct multiple polynomials with one global seed.

```
In [5]: rng = np.random.default_rng(123456789)

In [6]: galois.Poly.Random(5, seed=rng, field=GF)
Out[6]: Poly(56x^5 + 228x^4 + 157x^3 + 218x^2 + 148x + 43, GF(3^5))

In [7]: galois.Poly.Random(5, seed=rng, field=GF)
Out[7]: Poly(194x^5 + 195x^4 + 200x^3 + 141x^2 + 164x + 119, GF(3^5))
```

classmethod galois.Poly.Roots(roots: ArrayLike, multiplicities: *Sequence[int]* | *ndarray* | *None* = **None**, field: *type[*Array*]* | *None* = **None**) → Self

Constructs a monic polynomial over $\mathrm{GF}(p^m)$ from its roots.

Parameters

roots: *ArrayLike*
The roots of the desired polynomial.

multiplicities: Sequence[int] | ndarray | None = None
The corresponding root multiplicities. The default is `None` which corresponds to all ones.

field: type[*Array*] | None = None

The Galois field $\mathrm{GF}(p^m)$ the polynomial is over.

- `None` (default): If the roots are an *Array*, 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)`.
- *Array* subclass: The roots are explicitly converted to this Galois field using `field(roots)`.

Returns
The polynomial $f(x)$.

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

$$\begin{aligned} f(x) &= (x - r_1)^{m_1}(x - r_2)^{m_2} \dots (x - r_k)^{m_k} \\ &= a_d x^d + a_{d-1}x^{d-1} + \dots + a_1 x + a_0 \end{aligned}$$

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

# Evaluate the polynomial at its roots
In [3]: f(roots)
Out[3]: GF([0, 0, 0], order=2)
```

Construct a polynomial over $\mathrm{GF}(3^5)$ from a list of its roots with specific multiplicities.

```
In [4]: GF = galois.GF(3**5)

In [5]: roots = [121, 198, 225]

In [6]: f = galois.Poly.Roots(roots, multiplicities=[1, 2, 1], field=GF); f
Out[6]: Poly(x^4 + 215x^3 + 90x^2 + 183x + 119, GF(3^5))

# Evaluate the polynomial at its roots
In [7]: f(roots)
Out[7]: GF([0, 0, 0], order=3^5)
```

classmethod galois.Poly.Str(string: *str*, field: *type[Array] | None* = **None**) → Self

Constructs a polynomial over $\mathrm{GF}(p^m)$ from its string representation.

Parameters

string: str
The string representation of the polynomial $f(x)$.

field: type[*Array*] | None = None
The Galois field $\mathrm{GF}(p^m)$ the polynomial is over. The default is `None` which corresponds to *GF2*.

Returns
The polynomial $f(x)$.

Notes

Str() and *__str__()* are inverse operations.

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]: f = galois.Poly.Str("x^2 + 1"); f
Out[1]: Poly(x^2 + 1, GF(2))

In [2]: str(f)
Out[2]: 'x^2 + 1'
```

Construct a polynomial over $\mathrm{GF}(3^5)$ from its string representation.

```
In [3]: GF = galois.GF(3**5)

In [4]: f = galois.Poly.Str("13x^3 + 117", field=GF); f
Out[4]: Poly(13x^3 + 117, GF(3^5))

In [5]: str(f)
Out[5]: '13x^3 + 117'
```

classmethod galois.Poly.Zero(field: *type[Array] | None* = **None**) → Self

Constructs the polynomial $f(x) = 0$ over $\mathrm{GF}(p^m)$.

Parameters

field: type[*Array*] | None = None
The Galois field $\mathrm{GF}(p^m)$ the polynomial is over. The default is **None** which corresponds to *GF2*.

Returns
The polynomial $f(x) = 0$.

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 $\mathrm{GF}(3^5)$.

```
In [2]: GF = galois.GF(3**5)

In [3]: galois.Poly.Zero(GF)
Out[3]: Poly(0, GF(3^5))
```

Special methods

__call__(at: ElementLike | ArrayLike, ...) → *Array*

__call__(at: Poly) → *Poly*

Evaluates the polynomial $f(x)$ at x_0 or the polynomial composition $f(g(x))$.

__eq__(other: PolyLike) → bool

Determines if two polynomials are equal.

__int__() → int

The integer representation of the polynomial.

__len__() → int

Returns the length of the coefficient array, which is equivalent to `Poly.degree + 1`.

`galois.Poly.`**`__call__`**(at: ElementLike | ArrayLike, field: *type[*Array*]* | *None* = ***None***, elementwise: *bool* = ***True***) → *Array*

`galois.Poly.`**`__call__`**(at: Poly) → *Poly*

Evaluates the polynomial $f(x)$ at x_0 or the polynomial composition $f(g(x))$.

Parameters

at: *ElementLike* | *ArrayLike*
at: *Poly*
A finite field scalar or array x_0 to evaluate the polynomial at or the polynomial $g(x)$ to evaluate the polynomial composition $f(g(x))$.

field: type[*Array*] | None = None
The Galois field to evaluate the polynomial over. The default is `None` which represents the polynomial's current field, i.e. *`field`*.

elementwise: bool = True
Indicates whether to evaluate x_0 element-wise. The default is `True`. If `False` (only valid for square matrices), the polynomial indeterminate x is exponentiated using matrix powers (repeated matrix multiplication).

Returns

The result of the polynomial evaluation $f(x_0)$. The resulting array has the same shape as x_0. Or the polynomial composition $f(g(x))$.

Examples

Create a polynomial over $\mathrm{GF}(3^5)$.

```
In [1]: GF = galois.GF(3**5)

In [2]: f = galois.Poly([37, 123, 0, 201], field=GF); f
Out[2]: Poly(37x^3 + 123x^2 + 201, GF(3^5))
```

Evaluate the polynomial element-wise at x_0.

```
In [3]: x0 = GF([185, 218, 84, 163])

In [4]: f(x0)
Out[4]: GF([ 33, 163, 146,  96], order=3^5)

# The equivalent calculation
In [5]: GF(37)*x0**3 + GF(123)*x0**2 + GF(201)
Out[5]: GF([ 33, 163, 146,  96], order=3^5)
```

Evaluate the polynomial at the square matrix X_0.

```
In [6]: X0 = GF([[185, 218], [84, 163]])

# This is performed element-wise. Notice the values are equal to the vector x0.
In [7]: f(X0)
Out[7]:
GF([[ 33, 163],
    [146,  96]], order=3^5)

In [8]: f(X0, elementwise=False)
Out[8]:
GF([[103, 192],
    [156,  10]], order=3^5)

# The equivalent calculation
In [9]: GF(37)*np.linalg.matrix_power(X0, 3) + GF(123)*np.linalg.matrix_
↪power(X0, 2) + GF(201)*GF.Identity(2)
Out[9]:
GF([[103, 192],
    [156,  10]], order=3^5)
```

Evaluate the polynomial $f(x)$ at the polynomial $g(x)$.

```
In [10]: g = galois.Poly([55, 0, 1], field=GF); g
Out[10]: Poly(55x^2 + 1, GF(3^5))

In [11]: f(g)
Out[11]: Poly(77x^6 + 5x^4 + 104x^2 + 1, GF(3^5))

# The equivalent calculation
In [12]: GF(37)*g**3 + GF(123)*g**2 + GF(201)
Out[12]: Poly(77x^6 + 5x^4 + 104x^2 + 1, GF(3^5))
```

galois.Poly.**__eq__**(other: PolyLike) → bool

Determines if two polynomials are equal.

Parameters

other: ***PolyLike***
The polynomial to compare against.

Returns
`True` if the two polynomials have the same coefficients and are over the same finite field.

Examples

Compare two polynomials over the same field.

```
In [1]: a = galois.Poly([3, 0, 5], field=galois.GF(7)); a
Out[1]: Poly(3x^2 + 5, GF(7))

In [2]: b = galois.Poly([3, 0, 5], field=galois.GF(7)); b
Out[2]: Poly(3x^2 + 5, GF(7))

In [3]: a == b
Out[3]: True

# They are still two distinct objects, however
In [4]: a is b
Out[4]: False
```

Compare two polynomials with the same coefficients but over different fields.

```
In [5]: a = galois.Poly([3, 0, 5], field=galois.GF(7)); a
Out[5]: Poly(3x^2 + 5, GF(7))

In [6]: b = galois.Poly([3, 0, 5], field=galois.GF(7**2)); b
Out[6]: Poly(3x^2 + 5, GF(7^2))

In [7]: a == b
Out[7]: False
```

Comparison with `PolyLike` objects is allowed for convenience.

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

In [9]: a = galois.Poly([3, 0, 2], field=GF); a
Out[9]: Poly(3x^2 + 2, GF(7))

In [10]: a == GF([3, 0, 2])
Out[10]: True

In [11]: a == [3, 0, 2]
Out[11]: True

In [12]: a == "3x^2 + 2"
Out[12]: True
```

(continues on next page)

(continued from previous page)

```
In [13]: a == 3*7**2 + 2
Out[13]: True
```

galois.Poly.**__int__**() → int

The integer representation of the polynomial.

Notes

Int() and *__int__()* are inverse operations.

For the polynomial $f(x) = a_d x^d + a_{d-1} x^{d-1} + \cdots + a_1 x + a_0$ over the field $\mathrm{GF}(p^m)$, the integer representation is $i = a_d (p^m)^d + a_{d-1} (p^m)^{d-1} + \cdots + a_1 (p^m) + a_0$ using integer arithmetic, not finite field arithmetic.

Said differently, the polynomial coefficients $\{a_d, a_{d-1}, \dots, a_1, a_0\}$ are considered as the d "digits" of a radix-p^m value. The polynomial's integer representation is that value in decimal (radix-10).

Examples

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

In [2]: f = galois.Poly([3, 0, 5, 2], field=GF); f
Out[2]: Poly(3x^3 + 5x + 2, GF(7))

In [3]: int(f)
Out[3]: 1066

In [4]: int(f) == 3*GF.order**3 + 5*GF.order**1 + 2*GF.order**0
Out[4]: True
```

galois.Poly.**__len__**() → int

Returns the length of the coefficient array, which is equivalent to `Poly.degree + 1`.

Returns
The length of the coefficient array.

Examples

```
In [1]: GF = galois.GF(3**5)

In [2]: f = galois.Poly([37, 123, 0, 201], field=GF); f
Out[2]: Poly(37x^3 + 123x^2 + 201, GF(3^5))

In [3]: f.coeffs
Out[3]: GF([ 37, 123,   0, 201], order=3^5)

In [4]: len(f)
Out[4]: 4
```

(continues on next page)

(continued from previous page)

```
In [5]: f.degree + 1
Out[5]: 4
```

String representation

__repr__() → str

A representation of the polynomial and the finite field it's over.

__str__() → str

The string representation of the polynomial, without specifying the finite field it's over.

galois.Poly.**__repr__**() → str

A representation of the polynomial and the finite field it's over.

Tip

Use *set_printoptions()* to display the polynomial coefficients in degree-ascending order.

Examples

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

In [2]: f = galois.Poly([3, 0, 5, 2], field=GF); f
Out[2]: Poly(3x^3 + 5x + 2, GF(7))

In [3]: repr(f)
Out[3]: 'Poly(3x^3 + 5x + 2, GF(7))'
```

galois.Poly.**__str__**() → str

The string representation of the polynomial, without specifying the finite field it's over.

Tip

Use *set_printoptions()* to display the polynomial coefficients in degree-ascending order.

Notes

Str() and *__str__()* are inverse operations.

Examples

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

In [2]: f = galois.Poly([3, 0, 5, 2], field=GF); f
Out[2]: Poly(3x^3 + 5x + 2, GF(7))

In [3]: str(f)
Out[3]: '3x^3 + 5x + 2'

In [4]: print(f)
3x^3 + 5x + 2
```

Methods

derivative(k: *int* = *1*) → *Poly*

Computes the k-th formal derivative $\frac{d^k}{dx^k} f(x)$ of the polynomial $f(x)$.

reverse() → *Poly*

Returns the d-th reversal $x^d f(\frac{1}{x})$ of the polynomial $f(x)$ with degree d.

roots(multiplicity: *False* = ***False***) → *Array*

roots(multiplicity: *True*) → tuple[*Array*, np.ndarray]

Calculates the roots r of the polynomial $f(x)$, such that $f(r) = 0$.

galois.Poly.**derivative**(k: *int* = *1*) → *Poly*

Computes the k-th formal derivative $\frac{d^k}{dx^k} f(x)$ of the polynomial $f(x)$.

Parameters

k: int = 1
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 $f(x)$.

Notes

For the polynomial

$$f(x) = a_d x^d + a_{d-1} x^{d-1} + \cdots + a_1 x + a_0$$

the first formal derivative is defined as

$$f'(x) = (d) \cdot a_d x^{d-1} + (d-1) \cdot a_{d-1} x^{d-2} + \cdots + (2) \cdot a_2 x + a_1$$

where $\cdot$ represents scalar multiplication (repeated addition), not finite field multiplication. The exponent that is "brought down" and multiplied by the coefficient is an integer, not a finite field element. 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]: f = galois.Poly.Random(7); f
Out[1]: Poly(x^7 + x^3 + 1, GF(2))

In [2]: f.derivative()
Out[2]: Poly(x^6 + x^2, GF(2))

# p derivatives of a polynomial, where p is the field's characteristic, will␣
↪always result in 0
In [3]: f.derivative(GF.characteristic)
Out[3]: Poly(0, GF(2))
```

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

```
In [4]: GF = galois.GF(7)

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

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

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

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

# p derivatives of a polynomial, where p is the field's characteristic, will␣
↪always result in 0
In [9]: f.derivative(GF.characteristic)
Out[9]: Poly(0, GF(7))
```

Compute the derivatives of a polynomial over $\mathrm{GF}(3^5)$.

```
In [10]: GF = galois.GF(3**5)

In [11]: f = galois.Poly.Random(7, field=GF); f
Out[11]: Poly(180x^7 + 212x^6 + 164x^5 + 33x^4 + 208x^3 + 33x^2 + 222x + 66,␣
↪GF(3^5))

In [12]: f.derivative()
Out[12]: Poly(180x^6 + 82x^4 + 33x^3 + 57x + 222, GF(3^5))

In [13]: f.derivative(2)
```

(continues on next page)

(continued from previous page)

```
Out[13]: Poly(82x^3 + 57, GF(3^5))

# p derivatives of a polynomial, where p is the field's characteristic, will␣
↪always result in 0
In [14]: f.derivative(GF.characteristic)
Out[14]: Poly(0, GF(3^5))
```

galois.Poly.**reverse**() → *Poly*

Returns the d-th reversal $x^d f(\frac{1}{x})$ of the polynomial $f(x)$ with degree d.

Returns

The n-th reversal $x^n f(\frac{1}{x})$.

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))
```

galois.Poly.**roots**(multiplicity: *False* = **False**) → *Array*

galois.Poly.**roots**(multiplicity: *True*) → tuple[*Array*, np.ndarray]

Calculates the roots r of the polynomial $f(x)$, such that $f(r) = 0$.

Parameters

multiplicity: False = False
multiplicity: True
Optionally return the multiplicity of each root. The default is `False` which only returns the unique roots.

Returns

- An array of roots of $f(x)$. The roots are ordered in increasing order.
- The multiplicity of each root. This is 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 $f(\alpha^i) = 0$.

$$\begin{aligned} f(\alpha^i) &= a_d(\alpha^i)^d + \dots + a_1(\alpha^i) + a_0 \\ &\triangleq \lambda_{i,d} + \dots + \lambda_{i,1} + \lambda_{i,0} \\ &= \sum_{j=0}^{d} \lambda_{i,j} \end{aligned}$$

The next power of α can be easily calculated from the previous calculation.

$$\begin{aligned} f(\alpha^{i+1}) &= a_d(\alpha^{i+1})^d + \dots + a_1(\alpha^{i+1}) + a_0 \\ &= a_d(\alpha^i)^d \alpha^d + \dots + a_1(\alpha^i)\alpha + a_0 \\ &= \lambda_{i,d}\alpha^d + \dots + \lambda_{i,1}\alpha + \lambda_{i,0} \\ &= \sum_{j=0}^{d} \lambda_{i,j}\alpha^j \end{aligned}$$

Examples

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

```
In [1]: f = galois.Poly.Roots([1, 0], multiplicities=[7, 3]); f
Out[1]: Poly(x^10 + x^9 + x^8 + x^7 + x^6 + x^5 + x^4 + x^3, GF(2))

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

In [3]: f.roots(multiplicity=True)
Out[3]: (GF([0, 1], order=2), array([3, 7]))
```

Find the roots of a polynomial over $\mathrm{GF}(3^5)$.

```
In [4]: GF = galois.GF(3**5)

In [5]: f = galois.Poly.Roots([18, 227, 153], multiplicities=[5, 7, 3],␣
↪field=GF); f
Out[5]: Poly(x^15 + 118x^14 + 172x^13 + 50x^12 + 204x^11 + 202x^10 + 141x^9 +␣
↪153x^8 + 107x^7 + 187x^6 + 66x^5 + 221x^4 + 114x^3 + 121x^2 + 226x + 13, GF(3^
↪5))

In [6]: f.roots()
Out[6]: GF([ 18, 153, 227], order=3^5)

In [7]: f.roots(multiplicity=True)
Out[7]: (GF([ 18, 153, 227], order=3^5), array([5, 3, 7]))
```

Factorization methods

distinct_degree_factors() → tuple[list[*Poly*], list[int]]

Factors the monic, square-free polynomial $f(x)$ into a product of polynomials whose irreducible factors all have the same degree.

equal_degree_factors(degree: *int*) → list[*Poly*]

Factors the monic, square-free polynomial $f(x)$ of degree rd into a product of r irreducible factors with degree d.

factors() → tuple[list[*Poly*], list[int]]

Computes the irreducible factors of the non-constant, monic polynomial $f(x)$.

square_free_factors() → tuple[list[*Poly*], list[int]]

Factors the monic polynomial $f(x)$ into a product of square-free polynomials.

galois.Poly.**distinct_degree_factors**() → tuple[list[*Poly*], list[int]]

Factors the monic, square-free polynomial $f(x)$ into a product of polynomials whose irreducible factors all have the same degree.

Returns

- The list of polynomials $f_i(x)$ whose irreducible factors all have degree i.
- The list of corresponding distinct degrees i.

Raises

`ValueError` – If $f(x)$ is not monic, has degree 0, or is not square-free.

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

$$f_1(x) = x(x+1) = x^2 + x$$
$$f_2(x) = x^2 + x + 1$$
$$f_3(x) = (x^3 + x + 1)(x^3 + x^2 + 1) = x^6 + x^5 + x^4 + x^3 + x^2 + x + 1$$
$$f_i(x) = 1 \text{ for } i = 4, \dots, 10.$$

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 *square_free_factors()*. A complete polynomial factorization is implemented in *factors()*.

References

- Hachenberger, D. and Jungnickel, D. 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 GF(2).

```
In [1]: a = galois.Poly([1, 0]); a, a.is_irreducible()
Out[1]: (Poly(x, GF(2)), True)

In [2]: b = galois.Poly([1, 1]); b, b.is_irreducible()
Out[2]: (Poly(x + 1, GF(2)), True)

In [3]: c = galois.Poly([1, 1, 1]); c, c.is_irreducible()
Out[3]: (Poly(x^2 + x + 1, GF(2)), True)

In [4]: d = galois.Poly([1, 0, 1, 1]); d, d.is_irreducible()
Out[4]: (Poly(x^3 + x + 1, GF(2)), True)

In [5]: e = galois.Poly([1, 1, 0, 1]); e, e.is_irreducible()
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]: f.distinct_degree_factors()
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])
```

(continues on next page)

(continued from previous page)

```
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.Poly.**equal_degree_factors**(degree: *int*) → list[*Poly*]

Factors the monic, square-free polynomial $f(x)$ of degree rd into a product of r irreducible factors with degree d.

Parameters

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 lexicographical order.

Raises
ValueError – If $f(x)$ is not monic, has degree 0, or is not square-free.

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 *distinct_degree_factors()*. A complete polynomial factorization is implemented in *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, a.is_irreducible()
Out[1]: (Poly(x, GF(2)), True)

In [2]: b = galois.Poly([1, 1]); b, b.is_irreducible()
Out[2]: (Poly(x + 1, GF(2)), True)

In [3]: f = a * b; f
Out[3]: Poly(x^2 + x, GF(2))

In [4]: f.equal_degree_factors(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, a.is_irreducible()
Out[6]: (Poly(x^3 + 2x + 1, GF(5)), True)

In [7]: b = galois.Poly([1, 4, 4, 4], field=GF); b, b.is_irreducible()
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]: f.equal_degree_factors(3)
Out[9]: [Poly(x^3 + 2x + 1, GF(5)), Poly(x^3 + 4x^2 + 4x + 4, GF(5))]
```

`galois.Poly.factors()` → tuple[list[*Poly*], list[int]]

Computes the irreducible factors of the non-constant, monic polynomial $f(x)$.

Returns

- Sorted list of irreducible factors $\{g_1(x), g_2(x), \dots, g_k(x)\}$ of $f(x)$ sorted in lexicographical order.
- List of corresponding multiplicities $\{e_1, e_2, \dots, e_k\}$.

Raises

`ValueError` – If $f(x)$ is not monic or has degree 0.

Notes

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. See *square_free_factors()*.
2. Apply the Distinct-Degree Factorization algorithm to factor each square-free polynomial into a product of factors with the same degree. See *distinct_degree_factors()*.
3. Apply the Equal-Degree Factorization algorithm to factor the product of factors of equal degree into their irreducible factors. See *equal_degree_factors()*.

References

- Hachenberger, D. and Jungnickel, D. Topics in Galois Fields. Algorithm 6.1.7.
- Section 2.1 from https://people.csail.mit.edu/dmoshkov/courses/codes/poly-factorization.pdf

Examples

Generate irreducible polynomials over $\mathrm{GF}(3)$.

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

In [2]: g1 = galois.irreducible_poly(3, 3); g1
Out[2]: Poly(x^3 + 2x + 1, GF(3))

In [3]: g2 = galois.irreducible_poly(3, 4); g2
Out[3]: Poly(x^4 + x + 2, GF(3))

In [4]: g3 = galois.irreducible_poly(3, 5); g3
Out[4]: Poly(x^5 + 2x + 1, GF(3))
```

Construct a composite polynomial.

```
In [5]: e1, e2, e3 = 5, 4, 3

In [6]: f = g1**e1 * g2**e2 * g3**e3; f
Out[6]: Poly(x^46 + x^44 + 2x^41 + x^40 + 2x^39 + 2x^38 + 2x^37 + 2x^36 + 2x^34␣
↪+ x^33 + 2x^32 + x^31 + 2x^30 + 2x^29 + 2x^28 + 2x^25 + 2x^24 + 2x^23 + x^20␣
↪+ x^19 + x^18 + x^15 + 2x^10 + 2x^8 + x^5 + x^4 + x^3 + 1, GF(3))
```

Factor the polynomial into its irreducible factors over $\mathrm{GF}(3)$.

```
In [7]: f.factors()
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.Poly.square_free_factors()` → tuple[list[*Poly*], list[int]]

Factors the monic polynomial $f(x)$ into a product of square-free polynomials.

Returns

- The list of non-constant, square-free polynomials $h_j(x)$ in the factorization.
- The list of corresponding multiplicities j.

Raises

`ValueError` – If $f(x)$ is not monic or has degree 0.

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 *factors()*.

References

- Hachenberger, D. and Jungnickel, D. 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, a.is_irreducible()
Out[2]: (Poly(x, GF(5)), True)

In [3]: b = galois.Poly([1,0,2,4], field=GF); b, b.is_irreducible()
Out[3]: (Poly(x^3 + 2x + 4, GF(5)), True)

In [4]: c = galois.Poly([1,4,1], field=GF); c, c.is_irreducible()
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]: f.square_free_factors()
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])
```

Coefficients

coefficients(size: *int* | *None* = **None**, ...) → *Array*

Returns the polynomial coefficients in the order and size specified.

property ***coeffs*** : *Array*

The coefficients of the polynomial in degree-descending order.

property ***degrees*** : ndarray

An array of the polynomial degrees in descending order.

property ***nonzero_coeffs*** : *Array*

The non-zero coefficients of the polynomial in degree-descending order.

property ***nonzero_degrees*** : ndarray

An array of the polynomial degrees that have non-zero coefficients in descending order.

galois.Poly.**coefficients**(*size: int | None = None*, *order: 'desc' | 'asc' = 'desc'*) → *Array*

Returns the polynomial coefficients in the order and size specified.

Parameters

size: int | None = None
The fixed size of the coefficient array. Zeros will be added for higher-order terms. This value must be at least `degree + 1` or a `ValueError` will be raised. The default is `None` which corresponds to `degree + 1`.

order: 'desc' | 'asc' = 'desc'
The order of the coefficient degrees, either descending (default) or ascending.

Returns
An array of the polynomial coefficients with length `size`, either in descending order or ascending order.

Notes

This accessor is similar to the *coeffs* property, but it has more settings. By default, `Poly.coeffs == Poly.coefficients()`.

Examples

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

In [2]: f = galois.Poly([3, 0, 5, 2], field=GF); f
Out[2]: Poly(3x^3 + 5x + 2, GF(7))

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

In [4]: f.coefficients()
Out[4]: GF([3, 0, 5, 2], order=7)
```

Return the coefficients in ascending order.

```
In [5]: f.coefficients(order="asc")
Out[5]: GF([2, 5, 0, 3], order=7)
```

Return the coefficients in ascending order with size 8.

```
In [6]: f.coefficients(8, order="asc")
Out[6]: GF([2, 5, 0, 3, 0, 0, 0, 0], order=7)
```

property galois.Poly.**coeffs** : *Array*

The coefficients of the polynomial in degree-descending order.

Notes

The entries of *coeffs* are paired with *degrees*.

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)
```

property galois.Poly.**degrees** : ndarray

An array of the polynomial degrees in descending order.

Notes

The entries of *coeffs* are paired with *degrees*.

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])
```

property galois.Poly.**nonzero_coeffs** : *Array*

The non-zero coefficients of the polynomial in degree-descending order.

Notes

The entries of *nonzero_coeffs* are paired with *nonzero_degrees*.

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)
```

property galois.Poly.**nonzero_degrees** : ndarray

An array of the polynomial degrees that have non-zero coefficients in descending order.

Notes

The entries of *nonzero_coeffs* are paired with *nonzero_degrees*.

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])
```

Properties

property ***degree*** : int

The degree of the polynomial. The degree of a polynomial is the highest degree with a non-zero coefficient.

property ***field*** : type[*Array*]

The *Array* subclass for the finite field the coefficients are over.

property galois.Poly.**degree** : int

The degree of the polynomial. The degree of a polynomial is the highest degree with a 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
```

property galois.Poly.**field** : type[*Array*]

The *Array* subclass for the finite field the coefficients are over.

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.field
Out[2]: <class 'galois.GF(2)'>
```

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

In [4]: b = galois.Poly.Random(5, field=GF); b
Out[4]: Poly(201x^5 + 152x^4 + 80x^3 + 136x^2 + 6x + 134, GF(2^8))

In [5]: b.field
Out[5]: <class 'galois.GF(2^8)'>
```

Attributes

is_conway(search: *bool* = ***False***) → bool

Checks whether the degree-m polynomial $f(x)$ over $\mathrm{GF}(p)$ is the Conway polynomial $C_{p,m}(x)$.

is_conway_consistent(search: *bool* = ***False***) → bool

Determines whether the degree-m polynomial $f(x)$ over $\mathrm{GF}(p)$ is consistent with smaller Conway polynomials $C_{p,n}(x)$ for all $n\ |m$.

is_irreducible() → bool

Determines whether the polynomial $f(x)$ over $\mathrm{GF}(p^m)$ is irreducible.

property ***is_monic*** : bool

Returns whether the polynomial is monic, meaning its highest-degree coefficient is one.

is_primitive() → bool

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

is_square_free() → bool

Determines whether the polynomial $f(x)$ over $\mathrm{GF}(q)$ is square-free.

galois.Poly.**is_conway**(search: *bool* = ***False***) → bool

Checks whether the degree-m polynomial $f(x)$ over $\mathrm{GF}(p)$ is the Conway polynomial $C_{p,m}(x)$.

Why is this a method and not a property?

This is a method to indicate it is a computationally-expensive task.

Parameters

search: bool = False

Manually search for Conway polynomials if they are not included in Frank Luebeck's database. The default is **False**.

Slower performance

Manually searching for a Conway polynomial is *very* computationally expensive.

Returns

`True` if the polynomial $f(x)$ is the Conway polynomial $C_{p,m}(x)$.

Raises

`LookupError` – If `search=False` and the Conway polynomial $C_{p,m}$ is not found in Frank Luebeck's database.

See also

conway_poly, *Poly.is_conway_consistent*, *Poly.is_primitive*

Notes

A degree-m polynomial $f(x)$ over $\mathrm{GF}(p)$ is the *Conway polynomial* $C_{p,m}(x)$ if it is monic, primitive, compatible with Conway polynomials $C_{p,n}(x)$ for all $n\ |m$, and is lexicographically first according to a special ordering.

A Conway polynomial $C_{p,m}(x)$ is *compatible* with Conway polynomials $C_{p,n}(x)$ for $n\ |\ m$ if $C_{p,n}(x^r)$ divides $C_{p,m}(x)$, where $r = \frac{p^m-1}{p^n-1}$.

The Conway lexicographic ordering is defined as follows. Given two degree-m polynomials $g(x) = \sum_{i=0}^{m} g_i x^i$ and $h(x) = \sum_{i=0}^{m} h_i x^i$, then $g < h$ if and only if there exists i such that $g_j = h_j$ for all $j > i$ and $(-1)^{m-i} g_i < (-1)^{m-i} h_i$.

References

- http://www.math.rwth-aachen.de/~Frank.Luebeck/data/ConwayPol/CP7.html
- Lenwood S. Heath, Nicholas A. Loehr, New algorithms for generating Conway polynomials over finite fields, Journal of Symbolic Computation, Volume 38, Issue 2, 2004, Pages 1003-1024, https://www.sciencedirect.com/science/article/pii/S0747717104000331.

Examples

All Conway polynomials are primitive.

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

In [2]: f = galois.Poly([1, 1, 2, 4], field=GF); f
Out[2]: Poly(x^3 + x^2 + 2x + 4, GF(7))

In [3]: g = galois.Poly([1, 6, 0, 4], field=GF); g
Out[3]: Poly(x^3 + 6x^2 + 4, GF(7))

In [4]: f.is_primitive()
Out[4]: True

In [5]: g.is_primitive()
Out[5]: True
```

They are also consistent with all smaller Conway polynomials.

```
In [6]: f.is_conway_consistent()
Out[6]: True

In [7]: g.is_conway_consistent()
Out[7]: True
```

Among the multiple candidate Conway polynomials, the lexicographically-first (accordingly to a special lexicographical order) is the Conway polynomial.

```
In [8]: f.is_conway()
Out[8]: False

In [9]: g.is_conway()
Out[9]: True

In [10]: galois.conway_poly(7, 3)
Out[10]: Poly(x^3 + 6x^2 + 4, GF(7))
```

galois.Poly.**is_conway_consistent**(search: *bool* = **False**) → bool

Determines whether the degree-m polynomial $f(x)$ over $\mathrm{GF}(p)$ is consistent with smaller Conway polynomials $C_{p,n}(x)$ for all $n\ |m$.

Why is this a method and not a property?

This is a method to indicate it is a computationally-expensive task.

Parameters

search: bool = False

Manually search for Conway polynomials if they are not included in Frank Luebeck's database. The default is **False**.

Slower performance

Manually searching for a Conway polynomial is *very* computationally expensive.

Returns

True if the polynomial $f(x)$ is primitive and consistent with smaller Conway polynomials $C_{p,n}(x)$ for all $n\ |m$.

Raises

LookupError – If search=**False** and a smaller Conway polynomial $C_{p,n}$ is not found in Frank Luebeck's database.

See also

conway_poly, *Poly.is_conway*, *Poly.is_primitive*

Notes

A degree-m polynomial $f(x)$ over $\mathrm{GF}(p)$ is *compatible* with Conway polynomials $C_{p,n}(x)$ for $n\ |\ m$ if $C_{p,n}(x^r)$ divides $f(x)$, where $r = \frac{p^m-1}{p^n-1}$.

A Conway-consistent polynomial has all the properties of a Conway polynomial except that it is not necessarily lexicographically first (according to a special ordering).

References

- http://www.math.rwth-aachen.de/~Frank.Luebeck/data/ConwayPol/CP7.html
- Lenwood S. Heath, Nicholas A. Loehr, New algorithms for generating Conway polynomials over finite fields, Journal of Symbolic Computation, Volume 38, Issue 2, 2004, Pages 1003-1024, https://www.sciencedirect.com/science/article/pii/S0747717104000331.

Examples

All Conway polynomials are primitive.

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

In [2]: f = galois.Poly([1, 1, 2, 4], field=GF); f
Out[2]: Poly(x^3 + x^2 + 2x + 4, GF(7))

In [3]: g = galois.Poly([1, 6, 0, 4], field=GF); g
Out[3]: Poly(x^3 + 6x^2 + 4, GF(7))

In [4]: f.is_primitive()
Out[4]: True

In [5]: g.is_primitive()
Out[5]: True
```

They are also consistent with all smaller Conway polynomials.

```
In [6]: f.is_conway_consistent()
Out[6]: True

In [7]: g.is_conway_consistent()
Out[7]: True
```

Among the multiple candidate Conway polynomials, the lexicographically-first (accordingly to a special lexicographical order) is the Conway polynomial.

```
In [8]: f.is_conway()
Out[8]: False

In [9]: g.is_conway()
Out[9]: True

In [10]: galois.conway_poly(7, 3)
Out[10]: Poly(x^3 + 6x^2 + 4, GF(7))
```

`galois.Poly.is_irreducible()` → bool

Determines whether the polynomial $f(x)$ over $\mathrm{GF}(p^m)$ is irreducible.

Why is this a method and not a property?

This is a method to indicate it is a computationally-expensive task.

Returns

`True` if the polynomial is irreducible.

See also

irreducible_poly, *irreducible_polys*

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)\ |\ (x^{q^m} - x)\$ and $\gcd(f(x),\ x^{q^{m_i}} - x) = 1$ for $1 \le i \le k$, where $m_i = m/p_i$ for the k prime divisors p_i of m.

References

- Rabin, M. Probabilistic algorithms in finite fields. SIAM Journal on Computing (1980), 273-280. https://apps.dtic.mil/sti/pdfs/ADA078416.pdf
- Gao, S. and Panarino, D. 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]: f.is_irreducible()
Out[3]: True
```

```
In [4]: g = galois.irreducible_poly(2**4, 2, method="random"); g
Out[4]: Poly(x^2 + 7x + 4, GF(2^4))

In [5]: h = galois.irreducible_poly(2**4, 3, method="random"); h
Out[5]: Poly(x^3 + 9x^2 + 12x + 11, GF(2^4))

In [6]: f = g * h; f
Out[6]: Poly(x^5 + 14x^4 + 2x^3 + 11x^2 + x + 10, GF(2^4))

In [7]: f.is_irreducible()
Out[7]: False
```

property `galois.Poly.is_monic` : bool

Returns whether the polynomial is monic, meaning its highest-degree coefficient is one.

Examples

A monic polynomial over $\mathrm{GF}(7)$.

```
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]: p.is_monic
Out[3]: True
```

A non-monic polynomial over $\mathrm{GF}(7)$.

```
In [4]: GF = galois.GF(7)

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

In [6]: p.is_monic
Out[6]: False
```

`galois.Poly.is_primitive()` → bool

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

Why is this a method and not a property?

This is a method to indicate it is a computationally-expensive task.

Returns

`True` if the polynomial is primitive.

See also

primitive_poly, *primitive_polys*, *conway_poly*, *matlab_primitive_poly*

Notes

A degree-m polynomial $f(x)$ over $\mathrm{GF}(q)$ is *primitive* if it is irreducible and $f(x)|\ (x^k - 1)$ for $k = q^m - 1$ and no k less than $q^m - 1$.

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]: f.is_primitive()
Out[2]: True

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

In [4]: f.is_primitive()
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]: f.is_irreducible()
Out[6]: True

In [7]: f.is_primitive()
Out[7]: False
```

`galois.Poly.is_square_free()` → bool

Determines whether the polynomial $f(x)$ over $\mathrm{GF}(q)$ is square-free.

Why is this a method and not a property?

This is a method to indicate it is a computationally-expensive task.

Returns

`True` if the polynomial is square-free.

Notes

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

Generate irreducible polynomials over $\mathrm{GF}(3)$.

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

In [2]: f1 = galois.irreducible_poly(3, 3); f1
Out[2]: Poly(x^3 + 2x + 1, GF(3))

In [3]: f2 = galois.irreducible_poly(3, 4); f2
Out[3]: Poly(x^4 + x + 2, GF(3))
```

Determine if composite polynomials are square-free over $\mathrm{GF}(3)$.

```
In [4]: (f1 * f2).is_square_free()
Out[4]: True

In [5]: (f1**2 * f2).is_square_free()
Out[5]: False
```

`galois.typing.`**`PolyLike`**

A `Union` representing objects that can be coerced into a polynomial.

Union

- `int`: A polynomial in its integer representation, see *`Int()`*. The Galois field must be known from context.

```
# Known from context
In [1]: GF = galois.GF(3)

In [2]: galois.Poly.Int(19, field=GF)
Out[2]: Poly(2x^2 + 1, GF(3))
```

- `str`: A polynomial in its string representation, see *`Str()`*. The Galois field must be known from context.

```
In [3]: galois.Poly.Str("2x^2 + 1", field=GF)
Out[3]: Poly(2x^2 + 1, GF(3))
```

- *`ArrayLike`*: An array of polynomial coefficients in degree-descending order. If the coefficients are not *`Array`*, then the Galois field must be known from context.

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

In [5]: galois.Poly(GF([2, 0, 1]))
Out[5]: Poly(2x^2 + 1, GF(3))
```

- `Poly`: A previously-created `Poly` object. No coercion is necessary.

Alias

alias of `Union`[`int`, `str`, `Sequence`[`Union`[`int`, `str`, `Array`]], `Sequence`[*`IterableLike`*], `ndarray`, `Array`, `Poly`]

See also

The *Number theory* section contains many functions that operate on polynomials.

3.20.1 Irreducible polynomials

`galois.irreducible_poly(order: int, degree: int, ...) → Poly`

Returns a monic irreducible polynomial $f(x)$ over $\mathrm{GF}(q)$ with degree m.

`galois.irreducible_polys(order: int, degree, ...) → Iterator[Poly]`

Iterates through all monic irreducible polynomials $f(x)$ over $\mathrm{GF}(q)$ with degree m.

`galois.irreducible_poly(order: int, degree: int, terms: int | str | None = None, method: 'min' | 'max' | 'random' = 'min') → Poly`

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.

terms: int | str | None = None

The desired number of non-zero terms t in the polynomial.

- `None` (default): Disregards the number of terms while searching for the polynomial.
- `int`: The exact number of non-zero terms in the polynomial.
- `"min"`: The minimum possible number of non-zero terms.

method: 'min' | 'max' | 'random' = 'min'

The search method for finding the irreducible polynomial.

- `"min"` (default): Returns the lexicographically-first polynomial.
- `"max"`: Returns the lexicographically-last polynomial.
- `"random"`: Returns a random polynomial.

Faster performance

Depending on the type of polynomial requested, this function may use a database of precomputed polynomials. Under these conditions, this function returns very quickly.

- For $q = 2$, $2 \le m \le 10000$, `terms="min"`, and `method="min"`, the HP Table of Low-Weight Binary Irreducible Polynomials is used.

Returns

The degree-m monic irreducible polynomial over $\mathrm{GF}(q)$.

Raises

`RuntimeError` – If no monic irreducible polynomial of degree m over $\mathrm{GF}(q)$ with t terms exists. If `terms` is `None` or `"min"`, this should never be raised.

See also

Poly.is_irreducible, *primitive_poly*, *conway_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

Find the lexicographically-first, lexicographically-last, and a random monic irreducible polynomial.

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

In [2]: galois.irreducible_poly(7, 3, method="max")
Out[2]: Poly(x^3 + 6x^2 + 6x + 4, GF(7))

In [3]: galois.irreducible_poly(7, 3, method="random")
Out[3]: Poly(x^3 + 3x^2 + 4, GF(7))
```

Find the lexicographically-first monic irreducible polynomial with four terms.

```
In [4]: galois.irreducible_poly(7, 3, terms=4)
Out[4]: Poly(x^3 + x^2 + x + 2, GF(7))
```

Find the lexicographically-first monic irreducible polynomial with the minimum number of non-zero terms.

```
In [5]: galois.irreducible_poly(7, 3, terms="min")
Out[5]: Poly(x^3 + 2, GF(7))
```

Monic irreducible polynomials scaled by non-zero field elements (now non-monic) are also irreducible.

```
In [6]: GF = galois.GF(7)

In [7]: f = galois.irreducible_poly(7, 5, method="random"); f
Out[7]: Poly(x^5 + 6x^4 + 4x^3 + x^2 + 4x + 4, GF(7))

In [8]: f.is_irreducible()
Out[8]: True

In [9]: g = f * GF(3); g
Out[9]: Poly(3x^5 + 4x^4 + 5x^3 + 3x^2 + 5x + 5, GF(7))

In [10]: g.is_irreducible()
Out[10]: True
```

`galois.`**`irreducible_polys`**(order: *int*, degree: *int*, terms: *int* | *str* | *None* = ***None***, reverse: *bool* = ***False***) → Iterator[*Poly*]

Iterates through 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.

terms: int | str | None = None

The desired number of non-zero terms t in the polynomial.

- `None` (default): Disregards the number of terms while searching for the polynomial.
- `int`: The exact number of non-zero terms in the polynomial.
- `"min"`: The minimum possible number of non-zero terms.

reverse: bool = False
Indicates to return the irreducible polynomials from lexicographically last to first. The default is `False`.

Returns
An iterator over all degree-m monic irreducible polynomials over $\mathrm{GF}(q)$.

See also

`Poly.is_irreducible`, *`primitive_polys`*

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

Find all monic irreducible polynomials over $\mathrm{GF}(3)$ with degree 4. You may also use `tuple()` on the returned generator.

```
In [1]: list(galois.irreducible_polys(3, 4))
Out[1]:
[Poly(x^4 + x + 2, GF(3)),
 Poly(x^4 + 2x + 2, GF(3)),
 Poly(x^4 + x^2 + 2, GF(3)),
 Poly(x^4 + x^2 + x + 1, GF(3)),
 Poly(x^4 + x^2 + 2x + 1, GF(3)),
 Poly(x^4 + 2x^2 + 2, GF(3)),
 Poly(x^4 + x^3 + 2, GF(3)),
 Poly(x^4 + x^3 + 2x + 1, GF(3)),
 Poly(x^4 + x^3 + x^2 + 1, GF(3)),
 Poly(x^4 + x^3 + x^2 + x + 1, GF(3)),
 Poly(x^4 + x^3 + x^2 + 2x + 2, GF(3)),
 Poly(x^4 + x^3 + 2x^2 + 2x + 2, GF(3)),
 Poly(x^4 + 2x^3 + 2, GF(3)),
 Poly(x^4 + 2x^3 + x + 1, GF(3)),
 Poly(x^4 + 2x^3 + x^2 + 1, GF(3)),
 Poly(x^4 + 2x^3 + x^2 + x + 2, GF(3)),
 Poly(x^4 + 2x^3 + x^2 + 2x + 1, GF(3)),
 Poly(x^4 + 2x^3 + 2x^2 + x + 2, GF(3))]
```

Find all monic irreducible polynomials with four terms.

```
In [2]: list(galois.irreducible_polys(3, 4, terms=4))
Out[2]:
[Poly(x^4 + x^2 + x + 1, GF(3)),
 Poly(x^4 + x^2 + 2x + 1, GF(3)),
 Poly(x^4 + x^3 + 2x + 1, GF(3)),
 Poly(x^4 + x^3 + x^2 + 1, GF(3)),
 Poly(x^4 + 2x^3 + x + 1, GF(3)),
 Poly(x^4 + 2x^3 + x^2 + 1, GF(3))]
```

Find all monic irreducible polynomials with the minimum number of non-zero terms.

```
In [3]: list(galois.irreducible_polys(3, 4, terms="min"))
Out[3]:
[Poly(x^4 + x + 2, GF(3)),
 Poly(x^4 + 2x + 2, GF(3)),
 Poly(x^4 + x^2 + 2, GF(3)),
 Poly(x^4 + 2x^2 + 2, GF(3)),
 Poly(x^4 + x^3 + 2, GF(3)),
 Poly(x^4 + 2x^3 + 2, GF(3))]
```

Loop over all the polynomials in reversed order, only finding them as needed. The search cost for the polynomials that would have been found after the **break** condition is never incurred.

```
In [4]: for poly in galois.irreducible_polys(3, 4, reverse=True):
   ...:     if poly.coeffs[1] < 2:  # Early exit condition
   ...:         break
   ...:     print(poly)
   ...:
x^4 + 2x^3 + 2x^2 + x + 2
x^4 + 2x^3 + x^2 + 2x + 1
x^4 + 2x^3 + x^2 + x + 2
x^4 + 2x^3 + x^2 + 1
x^4 + 2x^3 + x + 1
x^4 + 2x^3 + 2
```

Or, manually iterate over the generator.

```
In [5]: generator = galois.irreducible_polys(3, 4, reverse=True); generator
Out[5]: <generator object irreducible_polys at 0x7fb35e17a820>

In [6]: next(generator)
Out[6]: Poly(x^4 + 2x^3 + 2x^2 + x + 2, GF(3))

In [7]: next(generator)
Out[7]: Poly(x^4 + 2x^3 + x^2 + 2x + 1, GF(3))

In [8]: next(generator)
Out[8]: Poly(x^4 + 2x^3 + x^2 + x + 2, GF(3))
```

3.20.2 Primitive polynomials

`galois.conway_poly`(characteristic: *int*, degree: *int*, ...) → *Poly*

Returns the Conway polynomial $C_{p,m}(x)$ over $\mathrm{GF}(p)$ with degree m.

`galois.matlab_primitive_poly`(characteristic: *int*, degree) → *Poly*

Returns Matlab's default primitive polynomial $f(x)$ over $\mathrm{GF}(p)$ with degree m.

`galois.primitive_poly`(order: *int*, degree: *int*, ...) → *Poly*

Returns a monic primitive polynomial $f(x)$ over $\mathrm{GF}(q)$ with degree m.

`galois.primitive_polys`(order: *int*, degree, ...) → Iterator[*Poly*]

Iterates through all monic primitive polynomials $f(x)$ over $\mathrm{GF}(q)$ with degree m.

`galois.conway_poly`(characteristic: *int*, degree: *int*, search: *bool* = **False**) → *Poly*

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.

search: bool = False

Manually search for Conway polynomials if they are not included in Frank Luebeck's database. The default is **False**.

Slower performance

Manually searching for a Conway polynomial is *very* computationally expensive.

Returns

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

See also

Poly.is_conway, *Poly.is_conway_consistent*, *Poly.is_primitive*, *primitive_poly*

Raises

LookupError – If search=**False** and the Conway polynomial $C_{p,m}$ is not found in Frank Luebeck's database.

Notes

A degree-m polynomial $f(x)$ over $\mathrm{GF}(p)$ is the *Conway polynomial* $C_{p,m}(x)$ if it is monic, primitive, compatible with Conway polynomials $C_{p,n}(x)$ for all \$n\ |m\$, and is lexicographically first according to a special ordering.

A Conway polynomial $C_{p,m}(x)$ is *compatible* with Conway polynomials $C_{p,n}(x)$ for \$n\ |\ m\$ if $C_{p,n}(x^r)$ divides $C_{p,m}(x)$, where $r = \frac{p^m-1}{p^n-1}$.

The Conway lexicographic ordering is defined as follows. Given two degree-m polynomials $g(x) = \sum_{i=0}^m g_i x^i$ and $h(x) = \sum_{i=0}^m h_i x^i$, then $g < h$ if and only if there exists i such that $g_j = h_j$ for all $j > i$ and $(-1)^{m-i} g_i < (-1)^{m-i} h_i$.

The Conway polynomial $C_{p,m}(x)$ provides a standard representation of $\mathrm{GF}(p^m)$ as a splitting field of $C_{p,m}(x)$. Conway polynomials provide compatibility between fields and their subfields and, hence, are the common way to represent extension fields.

References

- http://www.math.rwth-aachen.de/~Frank.Luebeck/data/ConwayPol/CP7.html
- Lenwood S. Heath, Nicholas A. Loehr, New algorithms for generating Conway polynomials over finite fields, Journal of Symbolic Computation, Volume 38, Issue 2, 2004, Pages 1003-1024, https://www.sciencedirect.com/science/article/pii/S0747717104000331.

Examples

All Conway polynomials are primitive.

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

In [2]: f = galois.Poly([1, 1, 2, 4], field=GF); f
Out[2]: Poly(x^3 + x^2 + 2x + 4, GF(7))
```

(continues on next page)

(continued from previous page)

```
In [3]: g = galois.Poly([1, 6, 0, 4], field=GF); g
Out[3]: Poly(x^3 + 6x^2 + 4, GF(7))

In [4]: f.is_primitive()
Out[4]: True

In [5]: g.is_primitive()
Out[5]: True
```

They are also consistent with all smaller Conway polynomials.

```
In [6]: f.is_conway_consistent()
Out[6]: True

In [7]: g.is_conway_consistent()
Out[7]: True
```

Among the multiple candidate Conway polynomials, the lexicographically-first (accordingly to a special lexicographical order) is the Conway polynomial.

```
In [8]: f.is_conway()
Out[8]: False

In [9]: g.is_conway()
Out[9]: True

In [10]: galois.conway_poly(7, 3)
Out[10]: Poly(x^3 + 6x^2 + 4, GF(7))
```

galois.**matlab_primitive_poly**(characteristic: *int*, degree: *int*) → *Poly*

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 $\mathrm{GF}(p)$.

See also

Poly.is_primitive, *primitive_poly*, *conway_poly*

Notes

This function returns the same result as Matlab's `gfprimdf(m, p)`. Matlab uses the lexicographically-first primitive polynomial with minimum terms, which is equivalent to `galois.primitive_poly(p, m, terms="min")`. There are three notable exceptions, however:

1. In $\mathrm{GF}(2^7)$, Matlab uses $x^7 + x^3 + 1$, not $x^7 + x + 1$.
2. In $\mathrm{GF}(2^{14})$, Matlab uses $x^{14} + x^{10} + x^6 + x + 1$, not $x^{14} + x^5 + x^3 + x + 1$.
3. In $\mathrm{GF}(2^{16})$, Matlab uses $x^{16} + x^{12} + x^3 + x + 1$, not $x^{16} + x^5 + x^3 + x^2 + 1$.

Warning

This has been tested for all the $\mathrm{GF}(2^m)$ fields for $2 \leq m \leq 16$ (Matlab doesn't support larger than 16). And it has been spot-checked for $\mathrm{GF}(p^m)$. There may exist other exceptions. Please submit a GitHub issue if you discover one.

References

- Lin, S. and Costello, D. Error Control Coding. Table 2.7.

Examples

```
In [1]: galois.primitive_poly(2, 6, terms="min")
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))
```

Below is one of the exceptions.

```
In [3]: galois.primitive_poly(2, 7, terms="min")
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.primitive_poly`(order: *int*, degree: *int*, terms: *int* | *str* | *None* = `None`, method: *'min'* | *'max'* | *'random'* = *'min'*) → *Poly*

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.

terms: int | str | None = None

The desired number of non-zero terms t in the polynomial.

- `None` (default): Disregards the number of terms while searching for the polynomial.

- `int`: The exact number of non-zero terms in the polynomial.
- `"min"`: The minimum possible number of non-zero terms.

method: 'min' | 'max' | 'random' = `'min'`

The search method for finding the primitive polynomial.

- `"min"` (default): Returns the lexicographically-first polynomial.
- `"max"`: Returns the lexicographically-last polynomial.
- `"random"`: Returns a random polynomial.

Returns

The degree-m monic primitive polynomial over $\mathrm{GF}(q)$.

Raises

`RuntimeError` – If no monic primitive polynomial of degree m over $\mathrm{GF}(q)$ with t terms exists. If `terms` is `None` or `"min"`, this should never be raised.

See also

Poly.is_primitive, *matlab_primitive_poly*, *conway_poly*

Notes

If $f(x)$ is a primitive polynomial over $\mathrm{GF}(q)$ and $a \in \mathrm{GF}(q) \backslash \{0\}$, then $a \cdot f(x)$ is also primitive.

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

Find the lexicographically-first, lexicographically-last, and a random monic primitive polynomial.

```
In [1]: galois.primitive_poly(7, 3)
Out[1]: Poly(x^3 + 3x + 2, GF(7))

In [2]: galois.primitive_poly(7, 3, method="max")
Out[2]: Poly(x^3 + 6x^2 + 6x + 4, GF(7))

In [3]: galois.primitive_poly(7, 3, method="random")
Out[3]: Poly(x^3 + 6x + 2, GF(7))
```

Find the lexicographically-first monic primitive polynomial with four terms.

```
In [4]: galois.primitive_poly(7, 3, terms=4)
Out[4]: Poly(x^3 + x^2 + x + 2, GF(7))
```

Find the lexicographically-first monic irreducible polynomial with the minimum number of non-zero terms.

```
In [5]: galois.primitive_poly(7, 3, terms="min")
Out[5]: Poly(x^3 + 3x + 2, GF(7))
```

Notice `primitive_poly()` returns the lexicographically-first primitive polynomial but `conway_poly()` returns the lexicographically-first primitive polynomial that is *consistent* with smaller Conway polynomials. This is sometimes the same polynomial.

```
In [6]: galois.primitive_poly(2, 4)
Out[6]: Poly(x^4 + x + 1, GF(2))

In [7]: galois.conway_poly(2, 4)
Out[7]: Poly(x^4 + x + 1, GF(2))
```

However, it is not always.

```
In [8]: galois.primitive_poly(7, 10)
Out[8]: Poly(x^10 + 5x^2 + x + 5, GF(7))

In [9]: galois.conway_poly(7, 10)
Out[9]: Poly(x^10 + x^6 + x^5 + 4x^4 + x^3 + 2x^2 + 3x + 3, GF(7))
```

Monic primitive polynomials scaled by non-zero field elements (now non-monic) are also primitive.

```
In [10]: GF = galois.GF(7)

In [11]: f = galois.primitive_poly(7, 5, method="random"); f
Out[11]: Poly(x^5 + 5x^4 + 2, GF(7))

In [12]: f.is_primitive()
Out[12]: True

In [13]: g = f * GF(3); g
Out[13]: Poly(3x^5 + x^4 + 6, GF(7))

In [14]: g.is_primitive()
Out[14]: True
```

`galois.`**`primitive_polys`**(order: *int*, degree: *int*, terms: *int* | *str* | *None* = **None**, reverse: *bool* = **False**) → Iterator[*Poly*]

Iterates through all monic primitive 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 primitive polynomial.

terms: int | str | None = None

The desired number of non-zero terms t in the polynomial.

- `None` (default): Disregards the number of terms while searching for the polynomial.
- `int`: The exact number of non-zero terms in the polynomial.
- `"min"`: The minimum possible number of non-zero terms.

reverse: bool = False
Indicates to return the primitive polynomials from lexicographically last to first. The default is `False`.

Returns
An iterator over all degree-m monic primitive polynomials over $\mathrm{GF}(q)$.

See also

Poly.is_primitive, *irreducible_polys*

Notes

If $f(x)$ is a primitive polynomial over $\mathrm{GF}(q)$ and $a \in \mathrm{GF}(q)\backslash\{0\}$, then $a \cdot f(x)$ is also primitive.

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

Find all monic primitive polynomials over $\mathrm{GF}(3)$ with degree 4. You may also use `tuple()` on the returned generator.

```
In [1]: list(galois.primitive_polys(3, 4))
Out[1]:
[Poly(x^4 + x + 2, GF(3)),
 Poly(x^4 + 2x + 2, GF(3)),
 Poly(x^4 + x^3 + 2, GF(3)),
 Poly(x^4 + x^3 + x^2 + 2x + 2, GF(3)),
 Poly(x^4 + x^3 + 2x^2 + 2x + 2, GF(3)),
 Poly(x^4 + 2x^3 + 2, GF(3)),
 Poly(x^4 + 2x^3 + x^2 + x + 2, GF(3)),
 Poly(x^4 + 2x^3 + 2x^2 + x + 2, GF(3))]
```

Find all monic primitive polynomials with five terms.

```
In [2]: list(galois.primitive_polys(3, 4, terms=5))
Out[2]:
[Poly(x^4 + x^3 + x^2 + 2x + 2, GF(3)),
 Poly(x^4 + x^3 + 2x^2 + 2x + 2, GF(3)),
 Poly(x^4 + 2x^3 + x^2 + x + 2, GF(3)),
 Poly(x^4 + 2x^3 + 2x^2 + x + 2, GF(3))]
```

Find all monic primitive polynomials with the minimum number of non-zero terms.

```
In [3]: list(galois.primitive_polys(3, 4, terms="min"))
Out[3]:
[Poly(x^4 + x + 2, GF(3)),
 Poly(x^4 + 2x + 2, GF(3)),
 Poly(x^4 + x^3 + 2, GF(3)),
 Poly(x^4 + 2x^3 + 2, GF(3))]
```

Loop over all the polynomials in reversed order, only finding them as needed. The search cost for the polynomials that would have been found after the **`break`** condition is never incurred.

```
In [4]: for poly in galois.primitive_polys(3, 4, reverse=True):
   ...:     if poly.coeffs[1] < 2:  # Early exit condition
   ...:         break
   ...:     print(poly)
   ...:
x^4 + 2x^3 + 2x^2 + x + 2
x^4 + 2x^3 + x^2 + x + 2
x^4 + 2x^3 + 2
```

Or, manually iterate over the generator.

```
In [5]: generator = galois.primitive_polys(3, 4, reverse=True); generator
Out[5]: <generator object primitive_polys at 0x7fb35d16fb30>

In [6]: next(generator)
Out[6]: Poly(x^4 + 2x^3 + 2x^2 + x + 2, GF(3))

In [7]: next(generator)
Out[7]: Poly(x^4 + 2x^3 + x^2 + x + 2, GF(3))

In [8]: next(generator)
Out[8]: Poly(x^4 + 2x^3 + 2, GF(3))
```

3.20.3 Interpolating polynomials

`galois.lagrange_poly(x: Array, y: Array) → Poly`

Computes the Lagrange interpolating polynomial $L(x)$ such that $L(x_i) = y_i$.

`galois.lagrange_poly(x: Array, y: Array) → Poly`

Computes the Lagrange interpolating polynomial $L(x)$ such that $L(x_i) = y_i$.

Parameters

x: ***Array***
An array of x_i values for the coordinates (x_i, y_i). Must be 1-D. Must have no duplicate entries.

y: ***Array***
An array of y_i values for the coordinates (x_i, y_i). Must be 1-D. Must be the same size as x.

Returns

The Lagrange polynomial $L(x)$.

Notes

The Lagrange interpolating polynomial is defined as

$$L(x) = \sum_{j=0}^{k-1} y_j \ell_j(x)$$

$$\ell_j(x) = \prod_{\substack{0 \le m < k \\ m \ne j}} \frac{x - x_m}{x_j - x_m}.$$

It is the polynomial of minimal degree that satisfies $L(x_i) = y_i$.

References

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

Examples

Create random (x, y) pairs in $\mathrm{GF}(3^2)$.

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

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

In [3]: y = GF.Random(x.size); y
Out[3]: GF([7, 4, 1, 1, 2, 2, 3, 6, 1], order=3^2)
```

Find the Lagrange polynomial that interpolates the coordinates.

```
In [4]: L = galois.lagrange_poly(x, y); L
Out[4]: Poly(7x^7 + 6x^6 + 3x^5 + 7x^4 + 8x^2 + 8x + 7, GF(3^2))
```

Show that the polynomial evaluated at x is y.

```
In [5]: np.array_equal(L(x), y)
Out[5]: True
```

3.21 Forward error correction

`class galois.BCH`

A general $\mathrm{BCH}(n, k)$ code over $\mathrm{GF}(q)$.

`class galois.ReedSolomon`

A general $\mathrm{RS}(n, k)$ code over $\mathrm{GF}(q)$.

`class galois.BCH`

A general $\mathrm{BCH}(n, k)$ code over $\mathrm{GF}(q)$.

A $\mathrm{BCH}(n, k)$ 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.

Shortened codes

To create the shortened $\mathrm{BCH}(n-s, k-s)$ code, construct the full-sized $\mathrm{BCH}(n, k)$ code and then pass $k-s$ symbols into *encode()* and $n-s$ symbols into *decode()*. Shortened codes are only applicable for systematic codes.

A BCH code is a cyclic code over $\mathrm{GF}(q)$ with generator polynomial $g(x)$. The generator polynomial is over $\mathrm{GF}(q)$ and has $d-1$ roots $\alpha^c, \dots, \alpha^{c+d-2}$ when evaluated in $\mathrm{GF}(q^m)$. The element α is a primitive n-th root of unity in $\mathrm{GF}(q^m)$.

$$g(x) = \mathrm{LCM}(m_{\alpha^c}(x), \dots, m_{\alpha^{c+d-2}}(x))$$

Examples

Construct a binary $\mathrm{BCH}(15, 7)$ code.

```
In [1]: bch = galois.BCH(15, 7); bch
Out[1]: <BCH Code: [15, 7, 5] over GF(2)>

In [2]: GF = bch.field; GF
Out[2]: <class 'galois.GF(2)'>
```

Encode a message.

```
In [3]: m = GF.Random(bch.k); m
Out[3]: GF([1, 1, 0, 0, 1, 0, 0], order=2)

In [4]: c = bch.encode(m); c
Out[4]: GF([1, 1, 0, 0, 1, 0, 0, 0, 1, 1, 1, 1, 0, 1, 0], order=2)
```

Corrupt the codeword and decode the message.

```
# Corrupt the first symbol in the codeword
In [5]: c[0] ^= 1; c
Out[5]: GF([0, 1, 0, 0, 1, 0, 0, 0, 1, 1, 1, 1, 0, 1, 0], order=2)

In [6]: dec_m = bch.decode(c); dec_m
Out[6]: GF([1, 1, 0, 0, 1, 0, 0], order=2)

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 = bch.decode(c, errors=True); dec_m, N
Out[8]: (GF([1, 1, 0, 0, 1, 0, 0], order=2), 1)

In [9]: np.array_equal(dec_m, m)
Out[9]: True
```

Constructors

BCH(n: *int*, k: *int* | *None* = ***None***, d: *int* | *None* = ***None***, ...)

Constructs a general $\mathrm{BCH}(n, k)$ code over $\mathrm{GF}(q)$.

`galois.BCH`(n: *int*, k: *int* | *None* = ***None***, d: *int* | *None* = ***None***, field: *type[*FieldArray*]* | *None* = ***None***, extension_field: *type[*FieldArray*]* | *None* = ***None***, alpha: ElementLike | *None* = ***None***, c: *int* = *1*, systematic: *bool* = ***True***)

Constructs a general $\mathrm{BCH}(n, k)$ code over $\mathrm{GF}(q)$.

Parameters

n: int
The codeword size n. If $n = q^m - 1$, the BCH code is *primitive*.

k: int | None = None

The message size k.

Important

Either `k` or `d` must be provided to define the code. Both may be provided as long as they are consistent.

d: int | None = None
The design distance d. This defines the number of roots $d - 1$ in the generator polynomial $g(x)$ over $\mathrm{GF}(q^m)$.

field: type[*FieldArray*] | None = None
The Galois field $\mathrm{GF}(q)$ that defines the alphabet of the codeword symbols. The default is `None` which corresponds to $\mathrm{GF}(2)$.

extension_field: type[*FieldArray*] | None = None
The Galois field $\mathrm{GF}(q^m)$ that defines the syndrome arithmetic. The default is `None` which corresponds to $\mathrm{GF}(q^m)$ where $q^{m-1} \le n < q^m$. The default extension field will use `matlab_primitive_poly(q, m)` for the irreducible polynomial.

alpha: *ElementLike* | None = None
A primitive n-th root of unity α in $\mathrm{GF}(q^m)$ that defines the $\alpha^c, \dots, \alpha^{c+d-2}$ roots of the generator polynomial $g(x)$.

c: int = 1
The first consecutive power c of α that defines the $\alpha^c, \dots, \alpha^{c+d-2}$ roots of the generator polynomial $g(x)$. The default is 1. If $c = 1$, the BCH code is *narrow-sense*.

systematic: bool = True
Indicates if the encoding should be systematic, meaning the codeword is the message with parity appended. The default is `True`.

See also

`matlab_primitive_poly`, *`FieldArray.primitive_root_of_unity`*

Examples

Construct a binary primitive, narrow-sense $\mathrm{BCH}(15, 7)$ code.

```
In [1]: galois.BCH(15, 7)
Out[1]: <BCH Code: [15, 7, 5] over GF(2)>

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

In [3]: galois.BCH(15, 7, 5)
Out[3]: <BCH Code: [15, 7, 5] over GF(2)>
```

Construct a primitive, narrow-sense $\mathrm{BCH}(26, 17)$ code over $\mathrm{GF}(3)$.

```
In [4]: GF = galois.GF(3)

In [5]: galois.BCH(26, 17, field=GF)
Out[5]: <BCH Code: [26, 17, 5] over GF(3)>

In [6]: galois.BCH(26, d=5, field=GF)
Out[6]: <BCH Code: [26, 17, 5] over GF(3)>

In [7]: galois.BCH(26, 17, 5, field=GF)
Out[7]: <BCH Code: [26, 17, 5] over GF(3)>
```

Construct a non-primitive, narrow-sense $\mathrm{BCH}(13, 4)$ code over $\mathrm{GF}(3)$.

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

In [9]: galois.BCH(13, 4, field=GF)
Out[9]: <BCH Code: [13, 4, 7] over GF(3)>

In [10]: galois.BCH(13, d=7, field=GF)
Out[10]: <BCH Code: [13, 4, 7] over GF(3)>

In [11]: galois.BCH(13, 4, 7, field=GF)
Out[11]: <BCH Code: [13, 4, 7] over GF(3)>
```

Discover primitive BCH codes over $\mathrm{GF}(5)$ by looping over the design distance.

```
In [12]: GF = galois.GF(5)

In [13]: n = 5**2 - 1

In [14]: for d in range(2, 11):
   ....:     bch = galois.BCH(n, d=d, field=GF)
   ....:     print(repr(bch))
   ....:
<BCH Code: [24, 22, 2] over GF(5)>
<BCH Code: [24, 20, 3] over GF(5)>
<BCH Code: [24, 18, 4] over GF(5)>
<BCH Code: [24, 16, 5] over GF(5)>
<BCH Code: [24, 16, 6] over GF(5)>
```

(continues on next page)

(continued from previous page)

```
<BCH Code: [24, 15, 7] over GF(5)>
<BCH Code: [24, 13, 8] over GF(5)>
<BCH Code: [24, 11, 9] over GF(5)>
<BCH Code: [24, 9, 10] over GF(5)>
```

String representation

__repr__() → str

A terse representation of the BCH code.

__str__() → str

A formatted string with relevant properties of the BCH code.

galois.BCH.**__repr__**() → str

A terse representation of the BCH code.

Examples

Construct a binary primitive BCH$(15, 7)$ code.

```
In [1]: bch = galois.BCH(15, 7)

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

Construct a primitive BCH$(26, 14)$ code over $\mathrm{GF}(3)$.

```
In [3]: bch = galois.BCH(26, 14, field=galois.GF(3))

In [4]: bch
Out[4]: <BCH Code: [26, 14, 7] over GF(3)>
```

Construct a non-primitive BCH$(13, 4)$ code over $\mathrm{GF}(3)$.

```
In [5]: bch = galois.BCH(13, 4, field=galois.GF(3))

In [6]: bch
Out[6]: <BCH Code: [13, 4, 7] over GF(3)>
```

galois.BCH.**__str__**() → str

A formatted string with relevant properties of the BCH code.

Examples

Construct a binary primitive $\mathrm{BCH}(15, 7)$ code.

```
In [1]: bch = galois.BCH(15, 7)

In [2]: print(bch)
BCH Code:
  [n, k, d]: [15, 7, 5]
  field: GF(2)
  extension_field: GF(2^4)
  generator_poly: x^8 + x^7 + x^6 + x^4 + 1
  is_primitive: True
  is_narrow_sense: True
  is_systematic: True
```

Construct a primitive $\mathrm{BCH}(26, 14)$ code over $\mathrm{GF}(3)$.

```
In [3]: bch = galois.BCH(26, 14, field=galois.GF(3))

In [4]: print(bch)
BCH Code:
  [n, k, d]: [26, 14, 7]
  field: GF(3)
  extension_field: GF(3^3)
  generator_poly: x^12 + x^11 + 2x^6 + x^3 + 2x^2 + 2x + 1
  is_primitive: True
  is_narrow_sense: True
  is_systematic: True
```

Construct a non-primitive $\mathrm{BCH}(13, 4)$ code over $\mathrm{GF}(3)$.

```
In [5]: bch = galois.BCH(13, 4, field=galois.GF(3))

In [6]: print(bch)
BCH Code:
  [n, k, d]: [13, 4, 7]
  field: GF(3)
  extension_field: GF(3^3)
  generator_poly: x^9 + x^8 + 2x^7 + x^5 + 2x^3 + 2x^2 + 2
  is_primitive: False
  is_narrow_sense: True
  is_systematic: True
```

Methods

decode(codeword: ArrayLike, ...) → *FieldArray*

decode(codeword, ...) → tuple[*FieldArray*, int | np.ndarray]

Decodes the codeword $\mathbf{c}$ into the message $\mathbf{m}$.

detect(codeword: ArrayLike) → bool | ndarray

Detects if errors are present in the codeword $\mathbf{c}$.

encode(message: ArrayLike, ...) → *FieldArray*

Encodes the message $\mathbf{m}$ into the codeword $\mathbf{c}$.

`galois.BCH.decode`(codeword: ArrayLike, output: *Literal[message] | Literal[codeword] = 'message'*, errors: *False = False*) → *FieldArray*

`galois.BCH.decode`(codeword: ArrayLike, output: *Literal[message] | Literal[codeword] = 'message'*, errors: *True = True*) → tuple[*FieldArray*, int | np.ndarray]

Decodes the codeword $\mathbf{c}$ into the message $\mathbf{m}$.

Parameters

codeword: ***ArrayLike***

The codeword as either a n-length vector or (N, n) matrix, where N is the number of codewords.

Shortened codes

For the shortened $[n-s,\ k-s,\ d]$ code (only applicable for systematic codes), pass $n-s$ symbols into *decode()* to return the $k-s$-symbol message.

---

output: Literal[message] | Literal[codeword] = `'message'`

Specify whether to return the error-corrected message or entire codeword. The default is `"message"`.

errors: False = `False`

errors: True = `True`

Optionally specify whether to return the number of corrected errors. The default is `False`.

Returns

- If `output="message"`, the error-corrected message as either a k-length vector or (N, k) matrix. If `output="codeword"`, the error-corrected codeword as either a n-length vector or (N, n) matrix.
- If `errors=True`, returns the number of corrected symbol errors as either a scalar or N-length array. 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 message vector $\mathbf{m}$ is a member of $\mathrm{GF}(q)^k$. The corresponding message polynomial $m(x)$ is a degree-k polynomial over $\mathrm{GF}(q)$.

$$\mathbf{m} = [m_{k-1},\ \dots,\ m_1,\ m_0] \in \mathrm{GF}(q)^k$$

$$m(x) = m_{k-1} x^{k-1} + \dots + m_1 x + m_0 \in \mathrm{GF}(q)[x]$$

The codeword vector $\mathbf{c}$ is a member of $\mathrm{GF}(q)^n$. The corresponding codeword polynomial $c(x)$ is a degree-n polynomial over $\mathrm{GF}(q)$. Each codeword polynomial $c(x)$ is divisible by the generator polynomial $g(x)$.

$$\mathbf{c} = [c_{n-1},\ \dots,\ c_1,\ c_0] \in \mathrm{GF}(q)^n$$

$$c(x) = c_{n-1} x^{n-1} + \dots + c_1 x + c_0 \in \mathrm{GF}(q)[x]$$

In decoding, the syndrome vector $\mathbf{s}$ is computed by evaluating the received codeword $\mathbf{r}$ in the extension field $\mathrm{GF}(q^m)$ at the roots $\alpha^c, \dots, \alpha^{c+d-2}$ of the generator polynomial $g(x)$. The equivalent polynomial operation computes the remainder of $r(x)$ by $g(x)$ in the extension field $\mathrm{GF}(q^m)$.

$$\mathbf{s} = [r(\alpha^c),\ \dots,\ r(\alpha^{c+d-2})] \in \mathrm{GF}(q^m)^{d-1}$$

$$s(x) = r(x)\ \textrm{mod}\ g(x) \in \mathrm{GF}(q^m)[x]$$

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.

Note

The $[n, k, d]_q$ code has $d_{min} \ge d$ minimum distance. It can detect up to $d_{min} - 1$ errors.

Examples

Vector

Encode a single message using the $\mathrm{BCH}(15, 7)$ code.

```
In [1]: bch = galois.BCH(15, 7)

In [2]: GF = bch.field

In [3]: m = GF.Random(bch.k); m
Out[3]: GF([0, 1, 0, 1, 1, 1, 0], order=2)

In [4]: c = bch.encode(m); c
Out[4]: GF([0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 0], order=2)
```

Corrupt t symbols of the codeword.

```
In [5]: bch.t
Out[5]: 2

In [6]: c[0:bch.t] ^= 1; c
Out[6]: GF([1, 0, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 0], order=2)
```

Decode the codeword and recover the message.

```
In [7]: d = bch.decode(c); d
Out[7]: GF([0, 1, 0, 1, 1, 1, 0], order=2)

In [8]: np.array_equal(d, m)
Out[8]: True
```

Decode the codeword, specifying the number of corrected errors, and recover the message.

```
In [9]: d, e = bch.decode(c, errors=True); d, e
Out[9]: (GF([0, 1, 0, 1, 1, 1, 0], order=2), 2)

In [10]: np.array_equal(d, m)
Out[10]: True
```

Vector (shortened)

Encode a single message using the shortened $\mathrm{BCH}(12, 4)$ code.

```
In [11]: bch = galois.BCH(15, 7)

In [12]: GF = bch.field

In [13]: m = GF.Random(bch.k - 3); m
Out[13]: GF([1, 0, 1, 0], order=2)

In [14]: c = bch.encode(m); c
Out[14]: GF([1, 0, 1, 0, 0, 1, 1, 0, 1, 1, 1, 0], order=2)
```

Corrupt t symbols of the codeword.

```
In [15]: bch.t
Out[15]: 2
```

(continues on next page)

(continued from previous page)

```
In [16]: c[0:bch.t] ^= 1; c
Out[16]: GF([0, 1, 1, 0, 0, 1, 1, 0, 1, 1, 1, 0], order=2)
```

Decode the codeword and recover the message.

```
In [17]: d = bch.decode(c); d
Out[17]: GF([1, 0, 1, 0], order=2)

In [18]: np.array_equal(d, m)
Out[18]: True
```

Decode the codeword, specifying the number of corrected errors, and recover the message.

```
In [19]: d, e = bch.decode(c, errors=True); d, e
Out[19]: (GF([1, 0, 1, 0], order=2), 2)

In [20]: np.array_equal(d, m)
Out[20]: True
```

Matrix

Encode a matrix of three messages using the $\mathrm{BCH}(15, 7)$ code.

```
In [21]: bch = galois.BCH(15, 7)

In [22]: GF = bch.field

In [23]: m = GF.Random((3, bch.k)); m
Out[23]:
GF([[1, 0, 0, 0, 1, 1, 1],
    [0, 1, 1, 0, 0, 1, 0],
    [1, 0, 0, 0, 1, 0, 1]], order=2)

In [24]: c = bch.encode(m); c
Out[24]:
GF([[1, 0, 0, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 0, 0],
    [0, 1, 1, 0, 0, 1, 0, 0, 0, 1, 1, 1, 1, 0, 1],
    [1, 0, 0, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1]], order=2)
```

Corrupt the codeword. Add zero errors to the first codeword, one to the second, and two to the third.

```
In [25]: c[1,0:1] ^= 1

In [26]: c[2,0:2] ^= 1

In [27]: c
Out[27]:
GF([[1, 0, 0, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 0, 0],
    [1, 1, 1, 0, 0, 1, 0, 0, 0, 1, 1, 1, 1, 0, 1],
    [0, 1, 0, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1]], order=2)
```

Decode the codeword and recover the message.

```
In [28]: d = bch.decode(c); d
Out[28]:
GF([[1, 0, 0, 0, 1, 1, 1],
    [0, 1, 1, 0, 0, 1, 0],
    [1, 0, 0, 0, 1, 0, 1]], order=2)

In [29]: np.array_equal(d, m)
Out[29]: True
```

Decode the codeword, specifying the number of corrected errors, and recover the message.

```
In [30]: d, e = bch.decode(c, errors=True); d, e
Out[30]:
(GF([[1, 0, 0, 0, 1, 1, 1],
     [0, 1, 1, 0, 0, 1, 0],
     [1, 0, 0, 0, 1, 0, 1]], order=2),
 array([0, 1, 2]))

In [31]: np.array_equal(d, m)
Out[31]: True
```

Matrix (shortened)

Encode a matrix of three messages using the shortened $\mathrm{BCH}(12, 4)$ code.

```
In [32]: bch = galois.BCH(15, 7)

In [33]: GF = bch.field

In [34]: m = GF.Random((3, bch.k - 3)); m
Out[34]:
GF([[0, 0, 0, 1],
    [0, 1, 0, 0],
    [1, 0, 1, 1]], order=2)

In [35]: c = bch.encode(m); c
Out[35]:
GF([[0, 0, 0, 1, 1, 1, 0, 1, 0, 0, 0, 1],
    [0, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1, 0],
    [1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1]], order=2)
```

Corrupt the codeword. Add zero errors to the first codeword, one to the second, and two to the third.

```
In [36]: c[1,0:1] ^= 1

In [37]: c[2,0:2] ^= 1

In [38]: c
Out[38]:
GF([[0, 0, 0, 1, 1, 1, 0, 1, 0, 0, 0, 1],
```

(continues on next page)

(continued from previous page)

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

Decode the codeword and recover the message.

```
In [39]: d = bch.decode(c); d
Out[39]:
GF([[0, 0, 0, 1],
    [0, 1, 0, 0],
    [1, 0, 1, 1]], order=2)

In [40]: np.array_equal(d, m)
Out[40]: True
```

Decode the codeword, specifying the number of corrected errors, and recover the message.

```
In [41]: d, e = bch.decode(c, errors=True); d, e
Out[41]:
(GF([[0, 0, 0, 1],
     [0, 1, 0, 0],
     [1, 0, 1, 1]], order=2),
 array([0, 1, 2]))

In [42]: np.array_equal(d, m)
Out[42]: True
```

`galois.BCH.detect(codeword: ArrayLike) → bool | ndarray`

Detects if errors are present in the codeword $\mathbf{c}$.

Parameters

codeword: *ArrayLike*

The codeword as either a n-length vector or (N, n) matrix, where N is the number of codewords.

Shortened codes

For the shortened $[n-s,\ k-s,\ d]$ code (only applicable for systematic codes), pass $n-s$ symbols into `detect()`.

Returns

A boolean scalar or N-length array indicating if errors were detected in the corresponding codeword.

Examples

Vector

Encode a single message using the $\mathrm{BCH}(15, 7)$ code.

```
In [1]: bch = galois.BCH(15, 7)

In [2]: GF = bch.field

In [3]: m = GF.Random(bch.k); m
Out[3]: GF([1, 1, 0, 0, 1, 0, 1], order=2)

In [4]: c = bch.encode(m); c
Out[4]: GF([1, 1, 0, 0, 1, 0, 1, 1, 0, 1, 0, 1, 0, 1, 1], order=2)
```

Detect no errors in the valid codeword.

```
In [5]: bch.detect(c)
Out[5]: False
```

Detect $d_{min} - 1$ errors in the codeword.

```
In [6]: bch.d
Out[6]: 5

In [7]: c[0:bch.d - 1] ^= 1; c
Out[7]: GF([0, 0, 1, 1, 1, 0, 1, 1, 0, 1, 0, 1, 0, 1, 1], order=2)

In [8]: bch.detect(c)
Out[8]: True
```

Vector (shortened)

Encode a single message using the shortened $\mathrm{BCH}(12, 4)$ code.

```
In [9]: bch = galois.BCH(15, 7)

In [10]: GF = bch.field

In [11]: m = GF.Random(bch.k - 3); m
Out[11]: GF([1, 0, 1, 1], order=2)

In [12]: c = bch.encode(m); c
Out[12]: GF([1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1], order=2)
```

Detect no errors in the valid codeword.

```
In [13]: bch.detect(c)
Out[13]: False
```

Detect $d_{min} - 1$ errors in the codeword.

```
In [14]: bch.d
Out[14]: 5

In [15]: c[0:bch.d - 1] ^= 1; c
Out[15]: GF([0, 1, 0, 0, 1, 0, 1, 1, 1, 1, 1, 1], order=2)

In [16]: bch.detect(c)
Out[16]: True
```

Matrix

Encode a matrix of three messages using the BCH$(15, 7)$ code.

```
In [17]: bch = galois.BCH(15, 7)

In [18]: GF = bch.field

In [19]: m = GF.Random((3, bch.k)); m
Out[19]:
GF([[0, 1, 0, 0, 0, 0, 1],
    [0, 1, 0, 0, 0, 1, 0],
    [1, 0, 0, 1, 1, 1, 0]], order=2)

In [20]: c = bch.encode(m); c
Out[20]:
GF([[0, 1, 0, 0, 0, 0, 1, 1, 0, 1, 0, 0, 1, 0, 1],
    [0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1],
    [1, 0, 0, 1, 1, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0]], order=2)
```

Detect no errors in the valid codewords.

```
In [21]: bch.detect(c)
Out[21]: array([False, False, False])
```

Detect one, two, and $d_{min} - 1$ errors in the codewords.

```
In [22]: bch.d
Out[22]: 5

In [23]: c[0, 0:1] ^= 1

In [24]: c[1, 0:2] ^= 1

In [25]: c[2, 0:bch.d - 1] ^= 1

In [26]: c
Out[26]:
GF([[1, 1, 0, 0, 0, 0, 1, 1, 0, 1, 0, 0, 1, 0, 1],
    [1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1],
    [0, 1, 1, 0, 1, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0]], order=2)
```

(continues on next page)

(continued from previous page)

```
In [27]: bch.detect(c)
Out[27]: array([ True,  True,  True])
```

Matrix (shortened)

Encode a matrix of three messages using the shortened $\mathrm{BCH}(12, 4)$ code.

```
In [28]: bch = galois.BCH(15, 7)

In [29]: GF = bch.field

In [30]: m = GF.Random((3, bch.k - 3)); m
Out[30]:
GF([[1, 1, 0, 1],
    [0, 0, 1, 0],
    [1, 0, 1, 1]], order=2)

In [31]: c = bch.encode(m); c
Out[31]:
GF([[1, 1, 0, 1, 0, 0, 1, 0, 1, 0, 1, 0],
    [0, 0, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1],
    [1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1]], order=2)
```

Detect no errors in the valid codewords.

```
In [32]: bch.detect(c)
Out[32]: array([False, False, False])
```

Detect one, two, and $d_{min} - 1$ errors in the codewords.

```
In [33]: bch.d
Out[33]: 5

In [34]: c[0, 0:1] ^= 1

In [35]: c[1, 0:2] ^= 1

In [36]: c[2, 0:bch.d - 1] ^= 1

In [37]: c
Out[37]:
GF([[0, 1, 0, 1, 0, 0, 1, 0, 1, 0, 1, 0],
    [1, 1, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1],
    [0, 1, 0, 0, 1, 0, 1, 1, 1, 1, 1, 1]], order=2)

In [38]: bch.detect(c)
Out[38]: array([ True,  True,  True])
```

`galois.BCH.encode`(message: ArrayLike, output: *Literal[codeword]* | *Literal[parity]* = 'codeword') → *FieldArray*

Encodes the message $\mathbf{m}$ into the codeword $\mathbf{c}$.

Parameters

message: *ArrayLike*

The message as either a k-length vector or (N, k) matrix, where N is the number of messages.

Shortened codes

For the shortened $[n-s,\ k-s,\ d]$ code (only applicable for systematic codes), pass $k-s$ symbols into `encode()` to return the $n-s$-symbol message.

---

output: Literal[codeword] | Literal[parity] = 'codeword'

Specify whether to return the codeword or parity symbols only. The default is `"codeword"`.

Returns

If `output="codeword"`, the codeword as either a n-length vector or (N, n) matrix. If `output="parity"`, the parity symbols as either a $n-k$-length vector or $(N, n-k)$ matrix.

Notes

The message vector $\mathbf{m}$ is a member of $\mathrm{GF}(q)^k$. The corresponding message polynomial $m(x)$ is a degree-k polynomial over $\mathrm{GF}(q)$.

$$\mathbf{m} = [m_{k-1},\ \dots,\ m_1,\ m_0] \in \mathrm{GF}(q)^k$$

$$m(x) = m_{k-1} x^{k-1} + \dots + m_1 x + m_0 \in \mathrm{GF}(q)[x]$$

The codeword vector $\mathbf{c}$ is a member of $\mathrm{GF}(q)^n$. The corresponding codeword polynomial $c(x)$ is a degree-n polynomial over $\mathrm{GF}(q)$.

$$\mathbf{c} = [c_{n-1},\ \dots,\ c_1,\ c_0] \in \mathrm{GF}(q)^n$$

$$c(x) = c_{n-1} x^{n-1} + \dots + c_1 x + c_0 \in \mathrm{GF}(q)[x]$$

The codeword vector is computed by matrix multiplication of the message vector with the generator matrix. The equivalent polynomial operation is multiplication of the message polynomial with the generator polynomial.

$$\mathbf{c} = \mathbf{m}\mathbf{G}$$

$$c(x) = m(x)g(x)$$

Examples

Vector

Encode a single message using the BCH$(15, 7)$ code.

```
In [1]: bch = galois.BCH(15, 7)

In [2]: GF = bch.field

In [3]: m = GF.Random(bch.k); m
Out[3]: GF([1, 1, 1, 1, 1, 1, 0], order=2)

In [4]: c = bch.encode(m); c
Out[4]: GF([1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 0, 1, 1, 1, 0], order=2)
```

Compute the parity symbols only.

```
In [5]: p = bch.encode(m, output="parity"); p
Out[5]: GF([0, 0, 1, 0, 1, 1, 1, 0], order=2)
```

Vector (shortened)

Encode a single message using the shortened BCH$(12, 4)$ code.

```
In [6]: bch = galois.BCH(15, 7)

In [7]: GF = bch.field

In [8]: m = GF.Random(bch.k - 3); m
Out[8]: GF([0, 1, 1, 0], order=2)

In [9]: c = bch.encode(m); c
Out[9]: GF([0, 1, 1, 0, 1, 0, 0, 1, 0, 1, 0, 1], order=2)
```

Compute the parity symbols only.

```
In [10]: p = bch.encode(m, output="parity"); p
Out[10]: GF([1, 0, 0, 1, 0, 1, 0, 1], order=2)
```

Matrix

Encode a matrix of three messages using the BCH$(15, 7)$ code.

```
In [11]: bch = galois.BCH(15, 7)

In [12]: GF = bch.field

In [13]: m = GF.Random((3, bch.k)); m
Out[13]:
GF([[1, 1, 1, 0, 0, 0, 0],
    [0, 0, 1, 0, 0, 0, 1],
```

(continues on next page)

(continued from previous page)

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

In [14]: c = bch.encode(m); c
Out[14]:
GF([[1, 1, 1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 1, 1, 0],
    [0, 0, 1, 0, 0, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1],
    [0, 0, 0, 0, 1, 1, 0, 1, 0, 0, 1, 0, 1, 0, 1]], order=2)
```

Compute the parity symbols only.

```
In [15]: p = bch.encode(m, output="parity"); p
Out[15]:
GF([[1, 0, 1, 0, 0, 1, 1, 0],
    [1, 1, 1, 0, 1, 0, 1, 1],
    [1, 0, 0, 1, 0, 1, 0, 1]], order=2)
```

Matrix (shortened)

Encode a matrix of three messages using the shortened BCH$(12, 4)$ code.

```
In [16]: bch = galois.BCH(15, 7)

In [17]: GF = bch.field

In [18]: m = GF.Random((3, bch.k - 3)); m
Out[18]:
GF([[1, 0, 0, 1],
    [1, 0, 1, 1],
    [1, 1, 1, 1]], order=2)

In [19]: c = bch.encode(m); c
Out[19]:
GF([[1, 0, 0, 1, 1, 1, 0, 0, 1, 1, 0, 0],
    [1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1],
    [1, 1, 1, 1, 0, 1, 0, 1, 1, 0, 0, 1]], order=2)
```

Compute the parity symbols only.

```
In [20]: p = bch.encode(m, output="parity"); p
Out[20]:
GF([[1, 1, 0, 0, 1, 1, 0, 0],
    [1, 0, 1, 1, 1, 1, 1, 1],
    [0, 1, 0, 1, 1, 0, 0, 1]], order=2)
```

Properties

property *d* : int

The minimum distance d of the $[n, k, d]_q$ code.

property *extension_field* : type[*FieldArray*]

The Galois field $\mathrm{GF}(q^m)$ that defines the BCH syndrome arithmetic.

property *field* : type[*FieldArray*]

The Galois field $\mathrm{GF}(q)$ that defines the codeword alphabet.

property *k* : int

The message size k of the $[n, k, d]_q$ code. This is also called the code *dimension*.

property *n* : int

The codeword size n of the $[n, k, d]_q$ code. This is also called the code *length*.

property *t* : int

The error-correcting capability t of the code.

property galois.BCH.**d** : int

The minimum distance d of the $[n, k, d]_q$ code.

Notes

The minimum distance of a BCH code may be greater than the design distance, i.e. $d_{min} \geq d$.

Examples

Construct a binary $\mathrm{BCH}(15, 7)$ code.

```
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
```

Construct a $\mathrm{BCH}(26, 14)$ code over $\mathrm{GF}(3)$.

```
In [3]: bch = galois.BCH(26, 14, field=galois.GF(3)); bch
Out[3]: <BCH Code: [26, 14, 7] over GF(3)>

In [4]: bch.d
Out[4]: 7
```

property galois.BCH.**extension_field** : type[*FieldArray*]

The Galois field $\mathrm{GF}(q^m)$ that defines the BCH syndrome arithmetic.

Examples

Construct a binary $\mathrm{BCH}(15, 7)$ code.

```
In [1]: bch = galois.BCH(15, 7); bch
Out[1]: <BCH Code: [15, 7, 5] over GF(2)>

In [2]: bch.extension_field
Out[2]: <class 'galois.GF(2^4)'>

In [3]: print(bch.extension_field.properties)
Galois Field:
  name: GF(2^4)
  characteristic: 2
  degree: 4
  order: 16
  irreducible_poly: x^4 + x + 1
  is_primitive_poly: True
  primitive_element: x
```

Construct a $\mathrm{BCH}(26, 14)$ code over $\mathrm{GF}(3)$.

```
In [4]: bch = galois.BCH(26, 14, field=galois.GF(3)); bch
Out[4]: <BCH Code: [26, 14, 7] over GF(3)>

In [5]: bch.extension_field
Out[5]: <class 'galois.GF(3^3)'>

In [6]: print(bch.extension_field.properties)
Galois Field:
  name: GF(3^3)
  characteristic: 3
  degree: 3
  order: 27
  irreducible_poly: x^3 + 2x + 1
  is_primitive_poly: True
  primitive_element: x
```

property `galois.BCH.`**`field`** : type[*FieldArray*]

The Galois field $\mathrm{GF}(q)$ that defines the codeword alphabet.

Examples

Construct a binary $\mathrm{BCH}(15, 7)$ code.

```
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 'galois.GF(2)'>

In [3]: print(bch.field.properties)
Galois Field:
```

(continues on next page)

(continued from previous page)

```
  name: GF(2)
  characteristic: 2
  degree: 1
  order: 2
  irreducible_poly: x + 1
  is_primitive_poly: True
  primitive_element: 1
```

Construct a $\mathrm{BCH}(26, 14)$ code over $\mathrm{GF}(3)$.

```
In [4]: bch = galois.BCH(26, 14, field=galois.GF(3)); bch
Out[4]: <BCH Code: [26, 14, 7] over GF(3)>

In [5]: bch.field
Out[5]: <class 'galois.GF(3)'>

In [6]: print(bch.field.properties)
Galois Field:
  name: GF(3)
  characteristic: 3
  degree: 1
  order: 3
  irreducible_poly: x + 1
  is_primitive_poly: True
  primitive_element: 2
```

property galois.BCH.**k** : int

The message size k of the $[n, k, d]_q$ code. This is also called the code *dimension*.

Examples

Construct a binary $\mathrm{BCH}(15, 7)$ code.

```
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
```

Construct a $\mathrm{BCH}(26, 14)$ code over $\mathrm{GF}(3)$.

```
In [3]: bch = galois.BCH(26, 14, field=galois.GF(3)); bch
Out[3]: <BCH Code: [26, 14, 7] over GF(3)>

In [4]: bch.k
Out[4]: 14
```

property galois.BCH.**n** : int

The codeword size n of the $[n, k, d]_q$ code. This is also called the code *length*.

Examples

Construct a binary $\mathrm{BCH}(15, 7)$ code.

```
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
```

Construct a $\mathrm{BCH}(26, 14)$ code over $\mathrm{GF}(3)$.

```
In [3]: bch = galois.BCH(26, 14, field=galois.GF(3)); bch
Out[3]: <BCH Code: [26, 14, 7] over GF(3)>

In [4]: bch.n
Out[4]: 26
```

property `galois.BCH.t` : int

The error-correcting capability t of the code.

Notes

The code can correct t symbol errors in a codeword.

$$t = \left\lfloor \frac{d-1}{2} \right\rfloor$$

Examples

Construct a binary $\mathrm{BCH}(15, 7)$ code.

```
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
```

Construct a $\mathrm{BCH}(26, 14)$ code over $\mathrm{GF}(3)$.

```
In [3]: bch = galois.BCH(26, 14, field=galois.GF(3)); bch
Out[3]: <BCH Code: [26, 14, 7] over GF(3)>

In [4]: bch.t
Out[4]: 3
```

Attributes

property ***is_narrow_sense*** : bool

Indicates if the BCH code is *narrow-sense*, meaning the roots of the generator polynomial are consecutive powers of α starting at 1, that is $\alpha, \dots, \alpha^{d-1}$.

property ***is_primitive*** : bool

Indicates if the BCH code is *primitive*, meaning $n = q^m - 1$.

property ***is_systematic*** : bool

Indicates if the code is *systematic*, meaning the codewords have parity appended to the message.

property galois.BCH.**is_narrow_sense** : bool

Indicates if the BCH code is *narrow-sense*, meaning the roots of the generator polynomial are consecutive powers of α starting at 1, that is $\alpha, \dots, \alpha^{d-1}$.

Examples

Construct a binary narrow-sense $\mathrm{BCH}(15, 7)$ code with first consecutive root α.

```
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.c == 1
Out[3]: True

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

In [5]: bch.roots
Out[5]: GF([2, 4, 8, 3], order=2^4)
```

Construct a binary non-narrow-sense $\mathrm{BCH}(15, 7)$ code with first consecutive root α^3. Notice the design distance of this code is only 3.

```
In [6]: bch = galois.BCH(15, 7, c=3); bch
Out[6]: <BCH Code: [15, 7, 3] over GF(2)>

In [7]: bch.is_narrow_sense
Out[7]: False

In [8]: bch.c == 1
Out[8]: False

In [9]: bch.generator_poly
Out[9]: Poly(x^8 + x^7 + x^6 + x^4 + 1, GF(2))

In [10]: bch.roots
Out[10]: GF([8, 3], order=2^4)
```

property `galois.BCH.is_primitive` : bool

Indicates if the BCH code is *primitive*, meaning $n = q^m - 1$.

Examples

Construct a binary primitive $\mathrm{BCH}(15, 7)$ code.

```
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

In [3]: bch.n == bch.extension_field.order - 1
Out[3]: True
```

Construct a non-primitive $\mathrm{BCH}(13, 7)$ code over $\mathrm{GF}(3)$.

```
In [4]: bch = galois.BCH(13, 7, field=galois.GF(3)); bch
Out[4]: <BCH Code: [13, 7, 4] over GF(3)>

In [5]: bch.is_primitive
Out[5]: False

In [6]: bch.n == bch.extension_field.order - 1
Out[6]: False
```

property `galois.BCH.is_systematic` : bool

Indicates if the code is *systematic*, meaning the codewords have parity appended to the message.

Examples

Construct a non-primitive $\mathrm{BCH}(13, 4)$ systematic code over $\mathrm{GF}(3)$.

```
In [1]: bch = galois.BCH(13, 4, field=galois.GF(3)); bch
Out[1]: <BCH Code: [13, 4, 7] over GF(3)>

In [2]: bch.is_systematic
Out[2]: True

In [3]: bch.G
Out[3]:
GF([[1, 0, 0, 0, 2, 2, 1, 0, 2, 0, 1, 1, 0],
    [0, 1, 0, 0, 0, 2, 2, 1, 0, 2, 0, 1, 1],
    [0, 0, 1, 0, 1, 1, 1, 2, 2, 0, 1, 2, 1],
    [0, 0, 0, 1, 1, 2, 0, 1, 0, 2, 2, 0, 2]], order=3)
```

Construct a non-primitive $\mathrm{BCH}(13, 4)$ non-systematic code over $\mathrm{GF}(3)$.

```
In [4]: bch = galois.BCH(13, 4, field=galois.GF(3), systematic=False); bch
Out[4]: <BCH Code: [13, 4, 7] over GF(3)>
```

(continues on next page)

(continued from previous page)

```
In [5]: bch.is_systematic
Out[5]: False

In [6]: bch.G
Out[6]:
GF([[1, 1, 2, 0, 1, 0, 2, 2, 0, 2, 0, 0, 0],
    [0, 1, 1, 2, 0, 1, 0, 2, 2, 0, 2, 0, 0],
    [0, 0, 1, 1, 2, 0, 1, 0, 2, 2, 0, 2, 0],
    [0, 0, 0, 1, 1, 2, 0, 1, 0, 2, 2, 0, 2]], order=3)

In [7]: bch.generator_poly
Out[7]: Poly(x^9 + x^8 + 2x^7 + x^5 + 2x^3 + 2x^2 + 2, GF(3))
```

Matrices

property ***G*** : *FieldArray*

The generator matrix $\mathbf{G}$ with shape (k, n).

property ***H*** : *FieldArray*

The parity-check matrix $\mathbf{H}$ with shape $(n-k, n)$.

property `galois.BCH.G` : *FieldArray*

The generator matrix $\mathbf{G}$ with shape (k, n).

Examples

Construct a binary primitive $\mathrm{BCH}(15, 7)$ code.

```
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)
```

Construct a non-primitive $\mathrm{BCH}(13, 4)$ code over $\mathrm{GF}(3)$.

```
In [3]: bch = galois.BCH(13, 4, field=galois.GF(3)); bch
Out[3]: <BCH Code: [13, 4, 7] over GF(3)>

In [4]: bch.G
Out[4]:
GF([[1, 0, 0, 0, 2, 2, 1, 0, 2, 0, 1, 1, 0],
    [0, 1, 0, 0, 0, 2, 2, 1, 0, 2, 0, 1, 1],
```

(continues on next page)

(continued from previous page)

```
    [0, 0, 1, 0, 1, 1, 1, 2, 2, 0, 1, 2, 1],
    [0, 0, 0, 1, 1, 2, 0, 1, 0, 2, 2, 0, 2]], order=3)
```

```
In [5]: bch = galois.BCH(13, 4, field=galois.GF(3), systematic=False); bch
Out[5]: <BCH Code: [13, 4, 7] over GF(3)>

In [6]: bch.G
Out[6]:
GF([[1, 1, 2, 0, 1, 0, 2, 2, 0, 2, 0, 0, 0],
    [0, 1, 1, 2, 0, 1, 0, 2, 2, 0, 2, 0, 0],
    [0, 0, 1, 1, 2, 0, 1, 0, 2, 2, 0, 2, 0],
    [0, 0, 0, 1, 1, 2, 0, 1, 0, 2, 2, 0, 2]], order=3)

In [7]: bch.generator_poly
Out[7]: Poly(x^9 + x^8 + 2x^7 + x^5 + 2x^3 + 2x^2 + 2, GF(3))
```

property galois.BCH.**H** : *FieldArray*

The parity-check matrix $\mathbf{H}$ with shape $(n-k, n)$.

Examples

Construct a binary primitive BCH$(15, 7)$ code.

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

In [3]: bch.parity_check_poly
Out[3]: Poly(x^7 + x^6 + x^4 + 1, GF(2))
```

Construct a non-primitive BCH$(13, 4)$ code over GF(3).

```
In [4]: bch = galois.BCH(13, 4, field=galois.GF(3)); bch
Out[4]: <BCH Code: [13, 4, 7] over GF(3)>

In [5]: bch.H
Out[5]:
GF([[1, 0, 2, 2, 1, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 1, 0, 2, 2, 1, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 1, 0, 2, 2, 1, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 1, 0, 2, 2, 1, 0, 0, 0, 0, 0],
```

(continues on next page)

(continued from previous page)

```
    [0, 0, 0, 0, 1, 0, 2, 2, 1, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 1, 0, 2, 2, 1, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 1, 0, 2, 2, 1, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 1, 0, 2, 2, 1, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 2, 2, 1]], order=3)

In [6]: bch.parity_check_poly
Out[6]: Poly(x^4 + 2x^3 + 2x^2 + 1, GF(3))
```

Polynomials

property ***alpha*** : *FieldArray*

A primitive n-th root of unity α in $\mathrm{GF}(q^m)$ whose consecutive powers $\alpha^c, \dots, \alpha^{c+d-2}$ are roots of the generator polynomial $g(x)$ in $\mathrm{GF}(q^m)$.

property ***c*** : int

The first consecutive power c of α that defines the roots $\alpha^c, \dots, \alpha^{c+d-2}$ of the generator polynomial $g(x)$.

property ***generator_poly*** : *Poly*

The generator polynomial $g(x)$ over $\mathrm{GF}(q)$.

property ***parity_check_poly*** : *Poly*

The parity-check polynomial $h(x)$.

property ***roots*** : *FieldArray*

The $d-1$ roots of the generator polynomial $g(x)$.

property galois.BCH.**alpha** : *FieldArray*

A primitive n-th root of unity α in $\mathrm{GF}(q^m)$ whose consecutive powers $\alpha^c, \dots, \alpha^{c+d-2}$ are roots of the generator polynomial $g(x)$ in $\mathrm{GF}(q^m)$.

Examples

Construct a binary primitive $\mathrm{BCH}(15, 7)$ code.

```
In [1]: bch = galois.BCH(15, 7); bch
Out[1]: <BCH Code: [15, 7, 5] over GF(2)>

In [2]: bch.alpha
Out[2]: GF(2, order=2^4)

In [3]: bch.roots[0] == bch.alpha ** bch.c
Out[3]: True

In [4]: bch.alpha.multiplicative_order() == bch.n
Out[4]: True
```

Construct a non-primitive $\mathrm{BCH}(13, 7)$ code over $\mathrm{GF}(3)$.

```
In [5]: bch = galois.BCH(13, 7, field=galois.GF(3)); bch
Out[5]: <BCH Code: [13, 7, 4] over GF(3)>
```

(continues on next page)

(continued from previous page)

```
In [6]: bch.alpha
Out[6]: GF(9, order=3^3)

In [7]: bch.roots[0] == bch.alpha ** bch.c
Out[7]: True

In [8]: bch.alpha.multiplicative_order() == bch.n
Out[8]: True
```

property galois.BCH.**c** : int

The first consecutive power c of α that defines the roots $\alpha^c, \dots, \alpha^{c+d-2}$ of the generator polynomial $g(x)$.

Examples

Construct a binary narrow-sense $\mathrm{BCH}(15, 7)$ code with first consecutive root α.

```
In [1]: bch = galois.BCH(15, 7); bch
Out[1]: <BCH Code: [15, 7, 5] over GF(2)>

In [2]: bch.c
Out[2]: 1

In [3]: bch.roots[0] == bch.alpha ** bch.c
Out[3]: True
```

Construct a binary non-narrow-sense $\mathrm{BCH}(15, 7)$ code with first consecutive root α^3. Notice the design distance of this code is only 3.

```
In [4]: bch = galois.BCH(15, 7, c=3); bch
Out[4]: <BCH Code: [15, 7, 3] over GF(2)>

In [5]: bch.c
Out[5]: 3

In [6]: bch.roots[0] == bch.alpha ** bch.c
Out[6]: True
```

property galois.BCH.**generator_poly** : *Poly*

The generator polynomial $g(x)$ over $\mathrm{GF}(q)$.

Notes

Every codeword $\mathbf{c}$ can be represented as a degree-n polynomial $c(x)$. Each codeword polynomial $c(x)$ is a multiple of $g(x)$.

Examples

Construct a binary narrow-sense $\mathrm{BCH}(15, 7)$ code with first consecutive root α.

```
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))

In [3]: bch.roots
Out[3]: GF([2, 4, 8, 3], order=2^4)

# Evaluate the generator polynomial at its roots in GF(q^m)
In [4]: bch.generator_poly(bch.roots, field=bch.extension_field)
Out[4]: GF([0, 0, 0, 0], order=2^4)
```

Construct a binary non-narrow-sense $\mathrm{BCH}(15, 7)$ code with first consecutive root α^3. Notice the design distance of this code is only 3 and it only has 2 roots in $\mathrm{GF}(2^4)$.

```
In [5]: bch = galois.BCH(15, 7, c=3); bch
Out[5]: <BCH Code: [15, 7, 3] over GF(2)>

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

In [7]: bch.roots
Out[7]: GF([8, 3], order=2^4)

# Evaluate the generator polynomial at its roots in GF(q^m)
In [8]: bch.generator_poly(bch.roots, field=bch.extension_field)
Out[8]: GF([0, 0], order=2^4)
```

property galois.BCH.**parity_check_poly** : *Poly*

The parity-check polynomial $h(x)$.

Notes

The parity-check polynomial is the generator polynomial of the dual code.

Examples

Construct a binary primitive $\mathrm{BCH}(15, 7)$ code.

```
In [1]: bch = galois.BCH(15, 7); bch
Out[1]: <BCH Code: [15, 7, 5] over GF(2)>

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

In [3]: bch.H
Out[3]:
```

(continues on next page)

(continued from previous page)

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

Construct a non-primitive $\mathrm{BCH}(13, 4)$ code over $\mathrm{GF}(3)$.

```
In [4]: bch = galois.BCH(13, 4, field=galois.GF(3)); bch
Out[4]: <BCH Code: [13, 4, 7] over GF(3)>

In [5]: bch.parity_check_poly
Out[5]: Poly(x^4 + 2x^3 + 2x^2 + 1, GF(3))

In [6]: bch.H
Out[6]:
GF([[1, 0, 2, 2, 1, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 1, 0, 2, 2, 1, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 1, 0, 2, 2, 1, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 1, 0, 2, 2, 1, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 1, 0, 2, 2, 1, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 1, 0, 2, 2, 1, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 1, 0, 2, 2, 1, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 1, 0, 2, 2, 1, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 2, 2, 1]], order=3)
```

property galois.BCH.**roots** : *FieldArray*

The $d - 1$ roots of the generator polynomial $g(x)$.

These are consecutive powers of α^c, specifically $\alpha^c, \dots, \alpha^{c+d-2}$.

Examples

Construct a binary narrow-sense $\mathrm{BCH}(15, 7)$ code with first consecutive root α.

```
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)

In [3]: bch.generator_poly
Out[3]: Poly(x^8 + x^7 + x^6 + x^4 + 1, GF(2))

# Evaluate the generator polynomial at its roots in GF(q^m)
In [4]: bch.generator_poly(bch.roots, field=bch.extension_field)
Out[4]: GF([0, 0, 0, 0], order=2^4)
```

Construct a binary non-narrow-sense $\mathrm{BCH}(15, 7)$ code with first consecutive root α^3. Notice the design distance of this code is only 3 and it only has 2 roots in $\mathrm{GF}(2^4)$.

```
In [5]: bch = galois.BCH(15, 7, c=3); bch
Out[5]: <BCH Code: [15, 7, 3] over GF(2)>

In [6]: bch.roots
Out[6]: GF([8, 3], order=2^4)

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

# Evaluate the generator polynomial at its roots in GF(q^m)
In [8]: bch.generator_poly(bch.roots, field=bch.extension_field)
Out[8]: GF([0, 0], order=2^4)
```

class `galois.ReedSolomon`

A general $\mathrm{RS}(n, k)$ code over $\mathrm{GF}(q)$.

A $\mathrm{RS}(n, k)$ code is a $[n, k, n-k+1]_q$ linear block code with codeword size n, message size k, minimum distance $d = n - k + 1$, and symbols taken from an alphabet of size q.

Shortened codes

To create the shortened $\mathrm{RS}(n-s, k-s)$ code, construct the full-sized $\mathrm{RS}(n, k)$ code and then pass $k-s$ symbols into `encode()` and $n-s$ symbols into `decode()`. Shortened codes are only applicable for systematic codes.

A Reed-Solomon code is a cyclic code over $\mathrm{GF}(q)$ with generator polynomial $g(x)$. The generator polynomial has $d-1$ roots $\alpha^c, \dots, \alpha^{c+d-2}$. The element α is a primitive n-th root of unity in $\mathrm{GF}(q)$.

$$g(x) = (x - \alpha^c) \dots (x - \alpha^{c+d-2})$$

Examples

Construct a $\mathrm{RS}(15, 9)$ code.

```
In [1]: rs = galois.ReedSolomon(15, 9); rs
Out[1]: <Reed-Solomon Code: [15, 9, 7] over GF(2^4)>

In [2]: GF = rs.field; GF
Out[2]: <class 'galois.GF(2^4)'>
```

Encode a message.

```
In [3]: m = GF.Random(rs.k); m
Out[3]: GF([12, 11, 14,  1,  8,  4,  4,  3,  4], order=2^4)

In [4]: c = rs.encode(m); c
Out[4]: GF([12, 11, 14,  1,  8,  4,  4,  3,  4,  0,  3,  0, 13,  6, 12], order=2^4)
```

Corrupt the codeword and decode the message.

```
# Corrupt the first symbol in the codeword
In [5]: c[0] ^= 13; c
Out[5]: GF([ 1, 11, 14,  1,  8,  4,  4,  3,  4,  0,  3,  0, 13,  6, 12], order=2^4)

In [6]: dec_m = rs.decode(c); dec_m
Out[6]: GF([12, 11, 14,  1,  8,  4,  4,  3,  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
Out[8]: (GF([12, 11, 14,  1,  8,  4,  4,  3,  4], order=2^4), 1)

In [9]: np.array_equal(dec_m, m)
Out[9]: True
```

Constructors

ReedSolomon(n: *int*, k: *int* | *None* = ***None***, d: *int* | *None* = ***None***, ...)

Constructs a general $\mathrm{RS}(n, k)$ code over $\mathrm{GF}(q)$.

`galois.`**`ReedSolomon`**(n: *int*, k: *int* | *None* = ***None***, d: *int* | *None* = ***None***, field: *type[*FieldArray*]* | *None* = ***None***, alpha: ElementLike | *None* = ***None***, c: *int* = *1*, systematic: *bool* = ***True***)

Constructs a general $\mathrm{RS}(n, k)$ code over $\mathrm{GF}(q)$.

Parameters

n: int
The codeword size n. If $n = q - 1$, the Reed-Solomon code is *primitive*.

k: int | None = None

The message size k.

Important

Either `k` or `d` must be provided to define the code. Both may be provided as long as they are consistent.

d: int | None = None
The design distance d. This defines the number of roots $d - 1$ in the generator polynomial $g(x)$ over $\mathrm{GF}(q)$. Reed-Solomon codes achieve the Singleton bound, so $d = n - k + 1$.

field: type[*FieldArray*] | None = None
The Galois field $\mathrm{GF}(q)$ that defines the alphabet of the codeword symbols. The default is `None` which corresponds to $\mathrm{GF}(2^m)$ where $2^{m-1} \le n < 2^m$. The default field will use `matlab_primitive_poly(2, m)` for the irreducible polynomial.

alpha: *ElementLike* | None = None
A primitive n-th root of unity α in $\mathrm{GF}(q)$ that defines the $\alpha^c, \dots, \alpha^{c+d-2}$ roots of the generator polynomial $g(x)$.

c: int = 1

The first consecutive power c of α that defines the $\alpha^c, \dots, \alpha^{c+d-2}$ roots of the generator polynomial $g(x)$. The default is 1. If $c = 1$, the Reed-Solomon code is *narrow-sense*.

systematic: bool = True

Indicates if the encoding should be systematic, meaning the codeword is the message with parity appended. The default is `True`.

See also

matlab_primitive_poly, *FieldArray.primitive_root_of_unity*

Examples

Construct a primitive, narrow-sense $\mathrm{RS}(255, 223)$ code over $\mathrm{GF}(2^8)$.

```
In [1]: galois.ReedSolomon(255, 223)
Out[1]: <Reed-Solomon Code: [255, 223, 33] over GF(2^8)>

In [2]: galois.ReedSolomon(255, d=33)
Out[2]: <Reed-Solomon Code: [255, 223, 33] over GF(2^8)>

In [3]: galois.ReedSolomon(255, 223, 33)
Out[3]: <Reed-Solomon Code: [255, 223, 33] over GF(2^8)>
```

Construct a non-primitive, narrow-sense $\mathrm{RS}(85, 65)$ code over $\mathrm{GF}(2^8)$.

```
In [4]: GF = galois.GF(2**8)

In [5]: galois.ReedSolomon(85, 65, field=GF)
Out[5]: <Reed-Solomon Code: [85, 65, 21] over GF(2^8)>

In [6]: galois.ReedSolomon(85, d=21, field=GF)
Out[6]: <Reed-Solomon Code: [85, 65, 21] over GF(2^8)>

In [7]: galois.ReedSolomon(85, 65, 21, field=GF)
Out[7]: <Reed-Solomon Code: [85, 65, 21] over GF(2^8)>
```

String representation

__repr__() → str

A terse representation of the Reed-Solomon code.

__str__() → str

A formatted string with relevant properties of the Reed-Solomon code.

`galois.ReedSolomon.`**`__repr__`**() → str

A terse representation of the Reed-Solomon code.

Examples

Construct a primitive, narrow-sense $\mathrm{RS}(255, 223)$ code over $\mathrm{GF}(2^8)$.

```
In [1]: rs = galois.ReedSolomon(255, 223)

In [2]: rs
Out[2]: <Reed-Solomon Code: [255, 223, 33] over GF(2^8)>
```

Construct a non-primitive, narrow-sense $\mathrm{RS}(85, 65)$ code over $\mathrm{GF}(2^8)$.

```
In [3]: rs = galois.ReedSolomon(85, 65, field=galois.GF(2**8))

In [4]: rs
Out[4]: <Reed-Solomon Code: [85, 65, 21] over GF(2^8)>
```

galois.ReedSolomon.**__str__**() → str

A formatted string with relevant properties of the Reed-Solomon code.

Examples

Construct a primitive, narrow-sense $\mathrm{RS}(255, 223)$ code over $\mathrm{GF}(2^8)$.

```
In [1]: rs = galois.ReedSolomon(255, 223)

In [2]: print(rs)
Reed-Solomon Code:
  [n, k, d]: [255, 223, 33]
  field: GF(2^8)
  generator_poly: x^32 + 232x^31 + 29x^30 + 189x^29 + 50x^28 + 142x^27 + 246x^
↪26 + 232x^25 + 15x^24 + 43x^23 + 82x^22 + 164x^21 + 238x^20 + x^19 + 158x^18␣
↪+ 13x^17 + 119x^16 + 158x^15 + 224x^14 + 134x^13 + 227x^12 + 210x^11 + 163x^
↪10 + 50x^9 + 107x^8 + 40x^7 + 27x^6 + 104x^5 + 253x^4 + 24x^3 + 239x^2 + 216x␣
↪+ 45
  is_primitive: True
  is_narrow_sense: True
  is_systematic: True
```

Construct a non-primitive, narrow-sense $\mathrm{RS}(85, 65)$ code over $\mathrm{GF}(2^8)$.

```
In [3]: rs = galois.ReedSolomon(85, 65, field=galois.GF(2**8))

In [4]: print(rs)
Reed-Solomon Code:
  [n, k, d]: [85, 65, 21]
  field: GF(2^8)
  generator_poly: x^20 + 126x^19 + 190x^18 + 191x^17 + 96x^16 + 116x^15 + 137x^
↪14 + 26x^13 + 203x^12 + 23x^11 + 208x^10 + 130x^9 + 104x^8 + 53x^7 + 188x^6 +␣
↪94x^5 + 146x^4 + 182x^3 + 210x^2 + 18x + 59
  is_primitive: False
  is_narrow_sense: True
  is_systematic: True
```

Methods

decode(codeword: ArrayLike, ...) → *FieldArray*

decode(codeword, ...) → tuple[*FieldArray*, int | np.ndarray]

Decodes the codeword $\mathbf{c}$ into the message $\mathbf{m}$.

detect(codeword: ArrayLike) → bool | ndarray

Detects if errors are present in the codeword $\mathbf{c}$.

encode(message: ArrayLike, ...) → *FieldArray*

Encodes the message $\mathbf{m}$ into the codeword $\mathbf{c}$.

`galois.ReedSolomon.`**`decode`**(codeword: ArrayLike, output: *Literal[message] | Literal[codeword] = 'message'*, errors: *False* = ***False***) → *FieldArray*

`galois.ReedSolomon.`**`decode`**(codeword: ArrayLike, output: *Literal[message] | Literal[codeword] = 'message'*, errors: *True* = ***True***) → tuple[*FieldArray*, int | np.ndarray]

Decodes the codeword $\mathbf{c}$ into the message $\mathbf{m}$.

Parameters

codeword: *ArrayLike*

The codeword as either a n-length vector or (N, n) matrix, where N is the number of codewords.

Shortened codes

For the shortened $[n-s,\ k-s,\ d]$ code (only applicable for systematic codes), pass $n-s$ symbols into *decode()* to return the $k-s$-symbol message.

---

output: Literal[message] | Literal[codeword] = 'message'

Specify whether to return the error-corrected message or entire codeword. The default is `"message"`.

errors: False = False

errors: True = True

Optionally specify whether to return the number of corrected errors. The default is `False`.

Returns

- If `output="message"`, the error-corrected message as either a k-length vector or (N, k) matrix. If `output="codeword"`, the error-corrected codeword as either a n-length vector or (N, n) matrix.
- If `errors=True`, returns the number of corrected symbol errors as either a scalar or N-length array. 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 message vector $\mathbf{m}$ is a member of $\mathrm{GF}(q)^k$. The corresponding message polynomial $m(x)$ is a degree-k polynomial over $\mathrm{GF}(q)$.

$$\mathbf{m} = [m_{k-1},\ \dots,\ m_1,\ m_0] \in \mathrm{GF}(q)^k$$

$$m(x) = m_{k-1}x^{k-1} + \cdots + m_1 x + m_0 \in \mathrm{GF}(q)[x]$$

The codeword vector $\mathbf{c}$ is a member of $\mathrm{GF}(q)^n$. The corresponding codeword polynomial $c(x)$ is a degree-n polynomial over $\mathrm{GF}(q)$. Each codeword polynomial $c(x)$ is divisible by the generator polynomial $g(x)$.

$$\mathbf{c} = [c_{n-1},\ \dots,\ c_1,\ c_0] \in \mathrm{GF}(q)^n$$

$$c(x) = c_{n-1}x^{n-1} + \cdots + c_1 x + c_0 \in \mathrm{GF}(q)[x]$$

In decoding, the syndrome vector $\mathbf{s}$ is computed by evaluating the received codeword $\mathbf{r}$ at the roots $\alpha^c, \dots, \alpha^{c+d-2}$ of the generator polynomial $g(x)$. The equivalent polynomial operation computes the remainder of $r(x)$ by $g(x)$.

$$\mathbf{s} = [r(\alpha^c),\ \dots,\ r(\alpha^{c+d-2})] \in \mathrm{GF}(q)^{d-1}$$

$$s(x) = r(x) \bmod g(x) \in \mathrm{GF}(q)[x]$$

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.

Examples

Vector

Encode a single message using the $\mathrm{RS}(15, 9)$ code.

```
In [1]: rs = galois.ReedSolomon(15, 9)

In [2]: GF = rs.field

In [3]: m = GF.Random(rs.k); m
Out[3]: GF([15,  0, 12, 14,  3,  4,  7, 14,  6], order=2^4)

In [4]: c = rs.encode(m); c
Out[4]: GF([15,  0, 12, 14,  3,  4,  7, 14,  6,  8,  7,  7,  5,  8,  1],
↪order=2^4)
```

Corrupt t symbols of the codeword.

```
In [5]: e = GF.Random(rs.t, low=1); e
Out[5]: GF([ 5,  9, 14], order=2^4)

In [6]: c[0:rs.t] += e; c
Out[6]: GF([10,  9,  2, 14,  3,  4,  7, 14,  6,  8,  7,  7,  5,  8,  1],␣
↪order=2^4)
```

Decode the codeword and recover the message.

```
In [7]: d = rs.decode(c); d
Out[7]: GF([15,  0, 12, 14,  3,  4,  7, 14,  6], order=2^4)

In [8]: np.array_equal(d, m)
Out[8]: True
```

Decode the codeword, specifying the number of corrected errors, and recover the message.

```
In [9]: d, e = rs.decode(c, errors=True); d, e
Out[9]: (GF([15,  0, 12, 14,  3,  4,  7, 14,  6], order=2^4), 3)

In [10]: np.array_equal(d, m)
Out[10]: True
```

Vector (shortened)

Encode a single message using the shortened $\mathrm{RS}(11, 5)$ code.

```
In [11]: rs = galois.ReedSolomon(15, 9)

In [12]: GF = rs.field

In [13]: m = GF.Random(rs.k - 4); m
Out[13]: GF([ 7, 14, 12,  3,  5], order=2^4)

In [14]: c = rs.encode(m); c
Out[14]: GF([ 7, 14, 12,  3,  5, 11,  5, 12,  2, 12,  7], order=2^4)
```

Corrupt t symbols of the codeword.

```
In [15]: e = GF.Random(rs.t, low=1); e
Out[15]: GF([ 5, 11, 11], order=2^4)

In [16]: c[0:rs.t] += e; c
Out[16]: GF([ 2,  5,  7,  3,  5, 11,  5, 12,  2, 12,  7], order=2^4)
```

Decode the codeword and recover the message.

```
In [17]: d = rs.decode(c); d
Out[17]: GF([ 7, 14, 12,  3,  5], order=2^4)

In [18]: np.array_equal(d, m)
Out[18]: True
```

Decode the codeword, specifying the number of corrected errors, and recover the message.

```
In [19]: d, e = rs.decode(c, errors=True); d, e
Out[19]: (GF([ 7, 14, 12,  3,  5], order=2^4), 3)

In [20]: np.array_equal(d, m)
Out[20]: True
```

Matrix

Encode a matrix of three messages using the RS$(15, 9)$ code.

```
In [21]: rs = galois.ReedSolomon(15, 9)

In [22]: GF = rs.field

In [23]: m = GF.Random((3, rs.k)); m
Out[23]:
GF([[ 2,  8,  9,  9,  0,  6,  7, 14,  8],
    [15,  1, 12,  6, 14,  7, 14, 12,  2],
    [15,  8,  4,  8, 12, 13, 10,  9,  5]], order=2^4)

In [24]: c = rs.encode(m); c
Out[24]:
GF([[ 2,  8,  9,  9,  0,  6,  7, 14,  8,  9, 11,  2,  0,  9,  6],
    [15,  1, 12,  6, 14,  7, 14, 12,  2,  4, 15, 10,  5, 10,  5],
    [15,  8,  4,  8, 12, 13, 10,  9,  5,  6,  0, 14,  6, 10,  6]],
   order=2^4)
```

Corrupt the codeword. Add one error to the first codeword, two to the second, and three to the third.

```
In [25]: c[0,0:1] += GF.Random(1, low=1)

In [26]: c[1,0:2] += GF.Random(2, low=1)

In [27]: c[2,0:3] += GF.Random(3, low=1)

In [28]: c
Out[28]:
GF([[ 6,  8,  9,  9,  0,  6,  7, 14,  8,  9, 11,  2,  0,  9,  6],
    [ 1,  7, 12,  6, 14,  7, 14, 12,  2,  4, 15, 10,  5, 10,  5],
    [ 1,  7,  7,  8, 12, 13, 10,  9,  5,  6,  0, 14,  6, 10,  6]],
   order=2^4)
```

Decode the codeword and recover the message.

```
In [29]: d = rs.decode(c); d
Out[29]:
GF([[ 2,  8,  9,  9,  0,  6,  7, 14,  8],
    [15,  1, 12,  6, 14,  7, 14, 12,  2],
    [15,  8,  4,  8, 12, 13, 10,  9,  5]], order=2^4)
```

(continues on next page)

(continued from previous page)

```
In [30]: np.array_equal(d, m)
Out[30]: True
```

Decode the codeword, specifying the number of corrected errors, and recover the message.

```
In [31]: d, e = rs.decode(c, errors=True); d, e
Out[31]:
(GF([[ 2,  8,  9,  9,  0,  6,  7, 14,  8],
     [15,  1, 12,  6, 14,  7, 14, 12,  2],
     [15,  8,  4,  8, 12, 13, 10,  9,  5]], order=2^4),
 array([1, 2, 3]))

In [32]: np.array_equal(d, m)
Out[32]: True
```

Matrix (shortened)

Encode a matrix of three messages using the shortened $\mathrm{RS}(11, 5)$ code.

```
In [33]: rs = galois.ReedSolomon(15, 9)

In [34]: GF = rs.field

In [35]: m = GF.Random((3, rs.k - 4)); m
Out[35]:
GF([[10,  6, 12,  9, 13],
    [13, 15,  8,  1,  0],
    [10,  4,  0,  0, 10]], order=2^4)

In [36]: c = rs.encode(m); c
Out[36]:
GF([[10,  6, 12,  9, 13,  0,  8,  0,  1,  9,  2],
    [13, 15,  8,  1,  0, 14,  6, 14, 11, 11,  2],
    [10,  4,  0,  0, 10,  9,  7,  1, 15, 13, 14]], order=2^4)
```

Corrupt the codeword. Add one error to the first codeword, two to the second, and three to the third.

```
In [37]: c[0,0:1] += GF.Random(1, low=1)

In [38]: c[1,0:2] += GF.Random(2, low=1)

In [39]: c[2,0:3] += GF.Random(3, low=1)

In [40]: c
Out[40]:
GF([[ 9,  6, 12,  9, 13,  0,  8,  0,  1,  9,  2],
    [ 2, 12,  8,  1,  0, 14,  6, 14, 11, 11,  2],
    [15,  1, 10,  0, 10,  9,  7,  1, 15, 13, 14]], order=2^4)
```

Decode the codeword and recover the message.

```
In [41]: d = rs.decode(c); d
Out[41]:
GF([[10,  6, 12,  9, 13],
    [13, 15,  8,  1,  0],
    [10,  4,  0,  0, 10]], order=2^4)

In [42]: np.array_equal(d, m)
Out[42]: True
```

Decode the codeword, specifying the number of corrected errors, and recover the message.

```
In [43]: d, e = rs.decode(c, errors=True); d, e
Out[43]:
(GF([[10,  6, 12,  9, 13],
     [13, 15,  8,  1,  0],
     [10,  4,  0,  0, 10]], order=2^4),
 array([1, 2, 3]))

In [44]: np.array_equal(d, m)
Out[44]: True
```

galois.ReedSolomon.**detect**(codeword: ArrayLike) → bool | ndarray

Detects if errors are present in the codeword $\mathbf{c}$.

Parameters

codeword: *ArrayLike*

The codeword as either a n-length vector or (N, n) matrix, where N is the number of codewords.

Shortened codes

For the shortened $[n-s,\ k-s,\ d]$ code (only applicable for systematic codes), pass $n-s$ symbols into *detect()*.

Returns

A boolean scalar or N-length array indicating if errors were detected in the corresponding codeword.

Examples

Vector

Encode a single message using the $\mathrm{RS}(15, 9)$ code.

```
In [1]: rs = galois.ReedSolomon(15, 9)

In [2]: GF = rs.field

In [3]: m = GF.Random(rs.k); m
Out[3]: GF([13, 14,  6, 14,  2,  8,  7, 13, 13], order=2^4)
```

(continues on next page)

(continued from previous page)

```
In [4]: c = rs.encode(m); c
Out[4]: GF([13, 14,  6, 14,  2,  8,  7, 13, 13,  4, 14, 11,  4,  6,  4],␣
↪order=2^4)
```

Detect no errors in the valid codeword.

```
In [5]: rs.detect(c)
Out[5]: False
```

Detect $d_{min} - 1$ errors in the codeword.

```
In [6]: rs.d
Out[6]: 7

In [7]: e = GF.Random(rs.d - 1, low=1); e
Out[7]: GF([ 5, 12, 12,  3,  9,  5], order=2^4)

In [8]: c[0:rs.d - 1] += e; c
Out[8]: GF([ 8,  2, 10, 13, 11, 13,  7, 13, 13,  4, 14, 11,  4,  6,  4],␣
↪order=2^4)

In [9]: rs.detect(c)
Out[9]: True
```

Vector (shortened)

Encode a single message using the shortened $\text{RS}(11, 5)$ code.

```
In [10]: rs = galois.ReedSolomon(15, 9)

In [11]: GF = rs.field

In [12]: m = GF.Random(rs.k - 4); m
Out[12]: GF([8, 0, 0, 5, 1], order=2^4)

In [13]: c = rs.encode(m); c
Out[13]: GF([ 8,  0,  0,  5,  1, 10, 13, 14,  7, 13,  2], order=2^4)
```

Detect no errors in the valid codeword.

```
In [14]: rs.detect(c)
Out[14]: False
```

Detect $d_{min} - 1$ errors in the codeword.

```
In [15]: rs.d
Out[15]: 7

In [16]: e = GF.Random(rs.d - 1, low=1); e
Out[16]: GF([ 6,  1, 14,  8,  5, 13], order=2^4)

In [17]: c[0:rs.d - 1] += e; c
```

(continues on next page)

(continued from previous page)

```
Out[17]: GF([14,  1, 14, 13,  4,  7, 13, 14,  7, 13,  2], order=2^4)

In [18]: rs.detect(c)
Out[18]: True
```

Matrix

Encode a matrix of three messages using the RS$(15, 9)$ code.

```
In [19]: rs = galois.ReedSolomon(15, 9)

In [20]: GF = rs.field

In [21]: m = GF.Random((3, rs.k)); m
Out[21]:
GF([[ 0, 11, 15, 11,  3,  7,  8,  4,  5],
    [ 3,  7, 12,  8,  8,  0,  2,  2,  2],
    [ 8,  8,  2, 14,  5,  7, 14,  9,  7]], order=2^4)

In [22]: c = rs.encode(m); c
Out[22]:
GF([[ 0, 11, 15, 11,  3,  7,  8,  4,  5,  1, 10,  2,  1, 10,  8],
    [ 3,  7, 12,  8,  8,  0,  2,  2,  2, 12,  6,  6, 10,  8,  9],
    [ 8,  8,  2, 14,  5,  7, 14,  9,  7, 10,  6,  3, 15,  7,  9]],
   order=2^4)
```

Detect no errors in the valid codewords.

```
In [23]: rs.detect(c)
Out[23]: array([False, False, False])
```

Detect one, two, and $d_{min} - 1$ errors in the codewords.

```
In [24]: rs.d
Out[24]: 7

In [25]: c[0, 0:1] += GF.Random(1, low=1)

In [26]: c[1, 0:2] += GF.Random(2, low=1)

In [27]: c[2, 0:rs.d - 1] += GF.Random(rs.d - 1, low=1)

In [28]: c
Out[28]:
GF([[12, 11, 15, 11,  3,  7,  8,  4,  5,  1, 10,  2,  1, 10,  8],
    [ 8,  4, 12,  8,  8,  0,  2,  2,  2, 12,  6,  6, 10,  8,  9],
    [ 4, 15,  4,  5,  4,  1, 14,  9,  7, 10,  6,  3, 15,  7,  9]],
   order=2^4)

In [29]: rs.detect(c)
Out[29]: array([ True,  True,  True])
```

Matrix (shortened)

Encode a matrix of three messages using the shortened $\text{RS}(11, 5)$ code.

```
In [30]: rs = galois.ReedSolomon(15, 9)

In [31]: GF = rs.field

In [32]: m = GF.Random((3, rs.k - 4)); m
Out[32]:
GF([[12, 12,  4,  9,  4],
    [ 2,  6, 13,  4,  2],
    [14,  6,  9, 13,  7]], order=2^4)

In [33]: c = rs.encode(m); c
Out[33]:
GF([[12, 12,  4,  9,  4,  7,  9,  4,  9, 13,  9],
    [ 2,  6, 13,  4,  2,  5, 13, 11,  9, 13, 15],
    [14,  6,  9, 13,  7, 12,  0,  0,  5,  2,  7]], order=2^4)
```

Detect no errors in the valid codewords.

```
In [34]: rs.detect(c)
Out[34]: array([False, False, False])
```

Detect one, two, and $d_{min} - 1$ errors in the codewords.

```
In [35]: rs.d
Out[35]: 7

In [36]: c[0, 0:1] += GF.Random(1, low=1)

In [37]: c[1, 0:2] += GF.Random(2, low=1)

In [38]: c[2, 0:rs.d - 1] += GF.Random(rs.d - 1, low=1)

In [39]: c
Out[39]:
GF([[ 9, 12,  4,  9,  4,  7,  9,  4,  9, 13,  9],
    [ 5,  8, 13,  4,  2,  5, 13, 11,  9, 13, 15],
    [ 2,  8,  5,  8,  6,  6,  0,  0,  5,  2,  7]], order=2^4)

In [40]: rs.detect(c)
Out[40]: array([ True,  True,  True])
```

`galois.ReedSolomon.encode`(*message: ArrayLike*, *output: Literal[codeword] | Literal[parity] = 'codeword'*) → *FieldArray*

Encodes the message $\mathbf{m}$ into the codeword $\mathbf{c}$.

Parameters

message: ***ArrayLike***

The message as either a k-length vector or (N, k) matrix, where N is the number of messages.

Shortened codes

For the shortened $[n-s,\ k-s,\ d]$ code (only applicable for systematic codes), pass $k-s$ symbols into `encode()` to return the $n-s$-symbol message.

output: Literal[codeword] | Literal[parity] = 'codeword'
Specify whether to return the codeword or parity symbols only. The default is `"codeword"`.

Returns
If `output="codeword"`, the codeword as either a n-length vector or (N, n) matrix. If `output="parity"`, the parity symbols as either a $n-k$-length vector or $(N, n-k)$ matrix.

Notes

The message vector $\mathbf{m}$ is a member of $\mathrm{GF}(q)^k$. The corresponding message polynomial $m(x)$ is a degree-k polynomial over $\mathrm{GF}(q)$.

$$\mathbf{m} = [m_{k-1},\ \dots,\ m_1,\ m_0] \in \mathrm{GF}(q)^k$$

$$m(x) = m_{k-1} x^{k-1} + \dots + m_1 x + m_0 \in \mathrm{GF}(q)[x]$$

The codeword vector $\mathbf{c}$ is a member of $\mathrm{GF}(q)^n$. The corresponding codeword polynomial $c(x)$ is a degree-n polynomial over $\mathrm{GF}(q)$.

$$\mathbf{c} = [c_{n-1},\ \dots,\ c_1,\ c_0] \in \mathrm{GF}(q)^n$$

$$c(x) = c_{n-1} x^{n-1} + \dots + c_1 x + c_0 \in \mathrm{GF}(q)[x]$$

The codeword vector is computed by matrix multiplication of the message vector with the generator matrix. The equivalent polynomial operation is multiplication of the message polynomial with the generator polynomial.

$$\mathbf{c} = \mathbf{m}\mathbf{G}$$

$$c(x) = m(x) g(x)$$

Examples

Vector

Encode a single message using the $\mathrm{RS}(15, 9)$ code.

```
In [1]: rs = galois.ReedSolomon(15, 9)

In [2]: GF = rs.field

In [3]: m = GF.Random(rs.k); m
Out[3]: GF([ 4, 12, 13,  3,  3,  8, 14,  9,  6], order=2^4)

In [4]: c = rs.encode(m); c
Out[4]: GF([ 4, 12, 13,  3,  3,  8, 14,  9,  6, 14, 12, 13, 14,  5,  0],␣
↪order=2^4)
```

Compute the parity symbols only.

```
In [5]: p = rs.encode(m, output="parity"); p
Out[5]: GF([14, 12, 13, 14,  5,  0], order=2^4)
```

Vector (shortened)

Encode a single message using the shortened $\mathrm{RS}(11, 5)$ code.

```
In [6]: rs = galois.ReedSolomon(15, 9)

In [7]: GF = rs.field

In [8]: m = GF.Random(rs.k - 4); m
Out[8]: GF([ 1,  8,  6, 11,  8], order=2^4)

In [9]: c = rs.encode(m); c
Out[9]: GF([ 1,  8,  6, 11,  8, 15, 10,  1,  4,  0,  4], order=2^4)
```

Compute the parity symbols only.

```
In [10]: p = rs.encode(m, output="parity"); p
Out[10]: GF([15, 10,  1,  4,  0,  4], order=2^4)
```

Matrix

Encode a matrix of three messages using the $\mathrm{RS}(15, 9)$ code.

```
In [11]: rs = galois.ReedSolomon(15, 9)

In [12]: GF = rs.field

In [13]: m = GF.Random((3, rs.k)); m
Out[13]:
GF([[14,  3, 10,  6,  7,  4,  8,  0,  7],
```

(continues on next page)

(continued from previous page)

```
    [15, 10, 12,  5, 14,  5,  9, 15, 14],
    [ 2, 13, 10,  9,  6,  2,  2,  2, 13]], order=2^4)

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

Compute the parity symbols only.

```
In [15]: p = rs.encode(m, output="parity"); p
Out[15]:
GF([[ 7,  8,  7,  2, 10,  1],
    [ 6, 13,  0,  2,  5, 11],
    [ 6, 15,  5,  6,  4,  6]], order=2^4)
```

Matrix (shortened)

Encode a matrix of three messages using the shortened RS$(11, 5)$ code.

```
In [16]: rs = galois.ReedSolomon(15, 9)

In [17]: GF = rs.field

In [18]: m = GF.Random((3, rs.k - 4)); m
Out[18]:
GF([[ 6, 13,  0,  1,  9],
    [10,  8,  3,  1,  1],
    [11, 10,  2, 10, 13]], order=2^4)

In [19]: c = rs.encode(m); c
Out[19]:
GF([[ 6, 13,  0,  1,  9,  3, 11,  9,  5,  9,  1],
    [10,  8,  3,  1,  1, 13,  1, 11, 10,  9,  7],
    [11, 10,  2, 10, 13,  7, 15, 15,  9,  7,  9]], order=2^4)
```

Compute the parity symbols only.

```
In [20]: p = rs.encode(m, output="parity"); p
Out[20]:
GF([[ 3, 11,  9,  5,  9,  1],
    [13,  1, 11, 10,  9,  7],
    [ 7, 15, 15,  9,  7,  9]], order=2^4)
```

Properties

property ***c*** : int

The first consecutive power c of α that defines the roots $\alpha^c, \dots, \alpha^{c+d-2}$ of the generator polynomial $g(x)$.

property ***d*** : int

The minimum distance d of the $[n, k, d]_q$ code.

property ***field*** : type[*FieldArray*]

The Galois field $\mathrm{GF}(q)$ that defines the codeword alphabet.

property ***k*** : int

The message size k of the $[n, k, d]_q$ code. This is also called the code *dimension*.

property ***n*** : int

The codeword size n of the $[n, k, d]_q$ code. This is also called the code *length*.

property ***t*** : int

The error-correcting capability t of the code.

property `galois.ReedSolomon.`**c** : int

The first consecutive power c of α that defines the roots $\alpha^c, \dots, \alpha^{c+d-2}$ of the generator polynomial $g(x)$.

Examples

Construct a narrow-sense $\mathrm{RS}(15, 9)$ code over $\mathrm{GF}(2^4)$ with first consecutive root α.

```
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

In [3]: rs.roots[0] == rs.alpha ** rs.c
Out[3]: True

In [4]: rs.generator_poly
Out[4]: Poly(x^6 + 7x^5 + 9x^4 + 3x^3 + 12x^2 + 10x + 12, GF(2^4))
```

Construct a narrow-sense $\mathrm{RS}(15, 9)$ code over $\mathrm{GF}(2^4)$ with first consecutive root α^3. Notice the design distance is the same, however the generator polynomial is different.

```
In [5]: rs = galois.ReedSolomon(15, 9, c=3); rs
Out[5]: <Reed-Solomon Code: [15, 9, 7] over GF(2^4)>

In [6]: rs.c
Out[6]: 3

In [7]: rs.roots[0] == rs.alpha ** rs.c
Out[7]: True

In [8]: rs.generator_poly
Out[8]: Poly(x^6 + 15x^5 + 8x^4 + 7x^3 + 9x^2 + 3x + 8, GF(2^4))
```

property galois.ReedSolomon.**d** : int

The minimum distance d of the $[n, k, d]_q$ code.

Examples

Construct a $\mathrm{RS}(15, 9)$ code over $\mathrm{GF}(2^4)$.

```
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
```

Construct a $\mathrm{RS}(26, 18)$ code over $\mathrm{GF}(3^3)$.

```
In [3]: rs = galois.ReedSolomon(26, 18, field=galois.GF(3**3)); rs
Out[3]: <Reed-Solomon Code: [26, 18, 9] over GF(3^3)>

In [4]: rs.d
Out[4]: 9
```

property galois.ReedSolomon.**field** : type[*FieldArray*]

The Galois field $\mathrm{GF}(q)$ that defines the codeword alphabet.

Examples

Construct a $\mathrm{RS}(15, 9)$ code over $\mathrm{GF}(2^4)$.

```
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 'galois.GF(2^4)'>

In [3]: print(rs.field.properties)
Galois Field:
  name: GF(2^4)
  characteristic: 2
  degree: 4
  order: 16
  irreducible_poly: x^4 + x + 1
  is_primitive_poly: True
  primitive_element: x
```

Construct a $\mathrm{RS}(26, 18)$ code over $\mathrm{GF}(3^3)$.

```
In [4]: rs = galois.ReedSolomon(26, 18, field=galois.GF(3**3)); rs
Out[4]: <Reed-Solomon Code: [26, 18, 9] over GF(3^3)>

In [5]: rs.field
Out[5]: <class 'galois.GF(3^3)'>
```

(continues on next page)

(continued from previous page)

```
In [6]: print(rs.field.properties)
Galois Field:
  name: GF(3^3)
  characteristic: 3
  degree: 3
  order: 27
  irreducible_poly: x^3 + 2x + 1
  is_primitive_poly: True
  primitive_element: x
```

property galois.ReedSolomon.**k** : int

The message size k of the $[n, k, d]_q$ code. This is also called the code *dimension*.

Examples

Construct a $\mathrm{RS}(15, 9)$ code over $\mathrm{GF}(2^4)$.

```
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
```

Construct a $\mathrm{RS}(26, 18)$ code over $\mathrm{GF}(3^3)$.

```
In [3]: rs = galois.ReedSolomon(26, 18, field=galois.GF(3**3)); rs
Out[3]: <Reed-Solomon Code: [26, 18, 9] over GF(3^3)>

In [4]: rs.k
Out[4]: 18
```

property galois.ReedSolomon.**n** : int

The codeword size n of the $[n, k, d]_q$ code. This is also called the code *length*.

Examples

Construct a $\mathrm{RS}(15, 9)$ code over $\mathrm{GF}(2^4)$.

```
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
```

Construct a $\mathrm{RS}(26, 18)$ code over $\mathrm{GF}(3^3)$.

```
In [3]: rs = galois.ReedSolomon(26, 18, field=galois.GF(3**3)); rs
Out[3]: <Reed-Solomon Code: [26, 18, 9] over GF(3^3)>

In [4]: rs.n
Out[4]: 26
```

property galois.ReedSolomon.**t** : int

The error-correcting capability t of the code.

Notes

The code can correct t symbol errors in a codeword.

$$t = \left\lfloor \frac{d-1}{2} \right\rfloor$$

Examples

Construct a $\mathrm{RS}(15, 9)$ code over $\mathrm{GF}(2^4)$.

```
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
```

Construct a $\mathrm{RS}(26, 18)$ code over $\mathrm{GF}(3^3)$.

```
In [3]: rs = galois.ReedSolomon(26, 18, field=galois.GF(3**3)); rs
Out[3]: <Reed-Solomon Code: [26, 18, 9] over GF(3^3)>

In [4]: rs.t
Out[4]: 4
```

Attributes

property ***is_narrow_sense*** : bool

Indicates if the Reed-Solomon code is *narrow-sense*, meaning the roots of the generator polynomial are consecutive powers of α starting at 1, that is $\alpha, \dots, \alpha^{d-1}$.

property ***is_primitive*** : bool

Indicates if the Reed-Solomon code is *primitive*, meaning $n = q - 1$.

property ***is_systematic*** : bool

Indicates if the code is *systematic*, meaning the codewords have parity appended to the message.

property galois.ReedSolomon.**is_narrow_sense** : bool

Indicates if the Reed-Solomon code is *narrow-sense*, meaning the roots of the generator polynomial are consecutive powers of α starting at 1, that is $\alpha, \dots, \alpha^{d-1}$.

Examples

Construct a narrow-sense $\mathrm{RS}(15, 9)$ code over $\mathrm{GF}(2^4)$ with first consecutive root α.

```
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.c == 1
Out[3]: True

In [4]: rs.generator_poly
Out[4]: Poly(x^6 + 7x^5 + 9x^4 + 3x^3 + 12x^2 + 10x + 12, GF(2^4))

In [5]: rs.roots
Out[5]: GF([ 2,  4,  8,  3,  6, 12], order=2^4)
```

Construct a narrow-sense $\mathrm{RS}(15, 9)$ code over $\mathrm{GF}(2^4)$ with first consecutive root α^3. Notice the design distance is the same, however the generator polynomial is different.

```
In [6]: rs = galois.ReedSolomon(15, 9, c=3); rs
Out[6]: <Reed-Solomon Code: [15, 9, 7] over GF(2^4)>

In [7]: rs.is_narrow_sense
Out[7]: False

In [8]: rs.c == 1
Out[8]: False

In [9]: rs.generator_poly
Out[9]: Poly(x^6 + 15x^5 + 8x^4 + 7x^3 + 9x^2 + 3x + 8, GF(2^4))

In [10]: rs.roots
Out[10]: GF([ 8,  3,  6, 12, 11,  5], order=2^4)
```

property `galois.ReedSolomon.is_primitive` : bool

Indicates if the Reed-Solomon code is *primitive*, meaning $n = q - 1$.

Examples

Construct a primitive $\mathrm{RS}(255, 223)$ code over $\mathrm{GF}(2^8)$.

```
In [1]: rs = galois.ReedSolomon(255, 223); rs
Out[1]: <Reed-Solomon Code: [255, 223, 33] over GF(2^8)>

In [2]: rs.is_primitive
Out[2]: True

In [3]: rs.n == rs.field.order - 1
Out[3]: True
```

Construct a non-primitive $\mathrm{RS}(85, 65)$ code over $\mathrm{GF}(2^8)$.

```
In [4]: rs = galois.ReedSolomon(85, 65, field=galois.GF(2**8)); rs
Out[4]: <Reed-Solomon Code: [85, 65, 21] over GF(2^8)>

In [5]: rs.is_primitive
Out[5]: False

In [6]: rs.n == rs.field.order - 1
Out[6]: False
```

property `galois.ReedSolomon.is_systematic` : bool

Indicates if the code is *systematic*, meaning the codewords have parity appended to the message.

Examples

Construct a non-primitive $\mathrm{RS}(13, 9)$ systematic code over $\mathrm{GF}(3^3)$.

```
In [1]: rs = galois.ReedSolomon(13, 9, field=galois.GF(3**3)); rs
Out[1]: <Reed-Solomon Code: [13, 9, 5] over GF(3^3)>

In [2]: rs.is_systematic
Out[2]: True

In [3]: rs.G
Out[3]:
GF([[ 1,  0,  0,  0,  0,  0,  0,  0,  0, 13, 20,  9, 16],
    [ 0,  1,  0,  0,  0,  0,  0,  0,  0, 14, 12,  7,  6],
    [ 0,  0,  1,  0,  0,  0,  0,  0,  0, 17, 15, 17, 21],
    [ 0,  0,  0,  1,  0,  0,  0,  0,  0, 12, 25, 19, 13],
    [ 0,  0,  0,  0,  1,  0,  0,  0,  0, 19, 15,  8,  3],
    [ 0,  0,  0,  0,  0,  1,  0,  0,  0, 22, 24, 13,  9],
    [ 0,  0,  0,  0,  0,  0,  1,  0,  0, 10, 10,  9, 18],
    [ 0,  0,  0,  0,  0,  0,  0,  1,  0, 20, 22, 25,  4],
    [ 0,  0,  0,  0,  0,  0,  0,  0,  1,  9,  8, 11, 22]], order=3^3)
```

Construct a non-primitive $\mathrm{RS}(13, 9)$ non-systematic code over $\mathrm{GF}(3^3)$.

```
In [4]: rs = galois.ReedSolomon(13, 9, field=galois.GF(3**3), systematic=False);
↪ rs
Out[4]: <Reed-Solomon Code: [13, 9, 5] over GF(3^3)>

In [5]: rs.is_systematic
Out[5]: False

In [6]: rs.G
Out[6]:
GF([[ 1,  9,  8, 11, 22,  0,  0,  0,  0,  0,  0,  0,  0],
    [ 0,  1,  9,  8, 11, 22,  0,  0,  0,  0,  0,  0,  0],
    [ 0,  0,  1,  9,  8, 11, 22,  0,  0,  0,  0,  0,  0],
    [ 0,  0,  0,  1,  9,  8, 11, 22,  0,  0,  0,  0,  0],
    [ 0,  0,  0,  0,  1,  9,  8, 11, 22,  0,  0,  0,  0],
    [ 0,  0,  0,  0,  0,  1,  9,  8, 11, 22,  0,  0,  0],
    [ 0,  0,  0,  0,  0,  0,  1,  9,  8, 11, 22,  0,  0],
```

(continues on next page)

(continued from previous page)

```
    [ 0,  0,  0,  0,  0,  0,  0,  1,  9,  8, 11, 22,  0],
    [ 0,  0,  0,  0,  0,  0,  0,  0,  1,  9,  8, 11, 22]], order=3^3)

In [7]: rs.generator_poly
Out[7]: Poly(x^4 + 9x^3 + 8x^2 + 11x + 22, GF(3^3))
```

Matrices

property ***G*** : *FieldArray*

The generator matrix $\mathbf{G}$ with shape (k, n).

property ***H*** : *FieldArray*

The parity-check matrix $\mathbf{H}$ with shape $(n - k, n)$.

property `galois.ReedSolomon.`**`G`** : *FieldArray*

The generator matrix $\mathbf{G}$ with shape (k, n).

Examples

Construct a primitive $\mathrm{RS}(15, 9)$ code over $\mathrm{GF}(2^4)$.

```
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)
```

Construct a non-primitive $\mathrm{RS}(13, 9)$ code over $\mathrm{GF}(3^3)$.

```
In [3]: rs = galois.ReedSolomon(13, 9, field=galois.GF(3**3)); rs
Out[3]: <Reed-Solomon Code: [13, 9, 5] over GF(3^3)>

In [4]: rs.G
Out[4]:
GF([[ 1,  0,  0,  0,  0,  0,  0,  0,  0, 13, 20,  9, 16],
    [ 0,  1,  0,  0,  0,  0,  0,  0,  0, 14, 12,  7,  6],
    [ 0,  0,  1,  0,  0,  0,  0,  0,  0, 17, 15, 17, 21],
    [ 0,  0,  0,  1,  0,  0,  0,  0,  0, 12, 25, 19, 13],
    [ 0,  0,  0,  0,  1,  0,  0,  0,  0, 19, 15,  8,  3],
    [ 0,  0,  0,  0,  0,  1,  0,  0,  0, 22, 24, 13,  9],
    [ 0,  0,  0,  0,  0,  0,  1,  0,  0, 10, 10,  9, 18],
```

(continues on next page)

(continued from previous page)

```
    [ 0,  0,  0,  0,  0,  0,  0,  1,  0, 20, 22, 25,  4],
    [ 0,  0,  0,  0,  0,  0,  0,  0,  1,  9,  8, 11, 22]], order=3^3)
```

```
In [5]: rs = galois.ReedSolomon(13, 9, field=galois.GF(3**3), systematic=False);
↪ rs
Out[5]: <Reed-Solomon Code: [13, 9, 5] over GF(3^3)>

In [6]: rs.G
Out[6]:
GF([[ 1,  9,  8, 11, 22,  0,  0,  0,  0,  0,  0,  0,  0],
    [ 0,  1,  9,  8, 11, 22,  0,  0,  0,  0,  0,  0,  0],
    [ 0,  0,  1,  9,  8, 11, 22,  0,  0,  0,  0,  0,  0],
    [ 0,  0,  0,  1,  9,  8, 11, 22,  0,  0,  0,  0,  0],
    [ 0,  0,  0,  0,  1,  9,  8, 11, 22,  0,  0,  0,  0],
    [ 0,  0,  0,  0,  0,  1,  9,  8, 11, 22,  0,  0,  0],
    [ 0,  0,  0,  0,  0,  0,  1,  9,  8, 11, 22,  0,  0],
    [ 0,  0,  0,  0,  0,  0,  0,  1,  9,  8, 11, 22,  0],
    [ 0,  0,  0,  0,  0,  0,  0,  0,  1,  9,  8, 11, 22]], order=3^3)

In [7]: rs.generator_poly
Out[7]: Poly(x^4 + 9x^3 + 8x^2 + 11x + 22, GF(3^3))
```

property `galois.ReedSolomon.H` : *FieldArray*

The parity-check matrix $\mathbf{H}$ with shape $(n-k, n)$.

Examples

Construct a primitive $\mathrm{RS}(15, 9)$ code over $\mathrm{GF}(2^4)$.

```
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)

In [3]: rs.parity_check_poly
Out[3]: Poly(x^9 + 7x^8 + 15x^7 + 2x^6 + 5x^5 + 3x^4 + 3x^3 + 11x^2 + 15x + 10,␣
↪GF(2^4))
```

Construct a non-primitive $\mathrm{RS}(13, 9)$ code over $\mathrm{GF}(3^3)$.

```
In [4]: rs = galois.ReedSolomon(13, 9, field=galois.GF(3**3)); rs
Out[4]: <Reed-Solomon Code: [13, 9, 5] over GF(3^3)>
```

(continues on next page)

(continued from previous page)

```
In [5]: rs.H
Out[5]:
GF([[25,  8, 22, 16,  7,  6, 11, 12, 20, 13, 15,  9,  1],
    [ 8, 16,  6, 12, 13,  9, 25, 22,  7, 11, 20, 15,  1],
    [22,  6, 20,  9,  8,  7, 12, 15, 25, 16, 11, 13,  1],
    [16, 12,  9, 22, 11, 15,  8,  6, 13, 25,  7, 20,  1]], order=3^3)

In [6]: rs.parity_check_poly
Out[6]: Poly(x^9 + 18x^8 + 10x^7 + 20x^6 + 17x^5 + 11x^4 + 24x^3 + 22x^2 + 25x␣
↪+ 26, GF(3^3))
```

Polynomials

property ***alpha*** : *FieldArray*

A primitive n-th root of unity α in $\mathrm{GF}(q)$ whose consecutive powers $\alpha^c, \dots, \alpha^{c+d-2}$ are roots of the generator polynomial $g(x)$.

property ***generator_poly*** : *Poly*

The generator polynomial $g(x)$ over $\mathrm{GF}(q)$.

property ***parity_check_poly*** : *Poly*

The parity-check polynomial $h(x)$.

property ***roots*** : *FieldArray*

The $d-1$ roots of the generator polynomial $g(x)$.

property galois.ReedSolomon.**alpha** : *FieldArray*

A primitive n-th root of unity α in $\mathrm{GF}(q)$ whose consecutive powers $\alpha^c, \dots, \alpha^{c+d-2}$ are roots of the generator polynomial $g(x)$.

Examples

Construct a primitive $\mathrm{RS}(255, 223)$ code over $\mathrm{GF}(2^8)$.

```
In [1]: rs = galois.ReedSolomon(255, 223); rs
Out[1]: <Reed-Solomon Code: [255, 223, 33] over GF(2^8)>

In [2]: rs.alpha
Out[2]: GF(2, order=2^8)

In [3]: rs.roots[0] == rs.alpha ** rs.c
Out[3]: True

In [4]: rs.alpha.multiplicative_order() == rs.n
Out[4]: True
```

Construct a non-primitive $\mathrm{RS}(85, 65)$ code over $\mathrm{GF}(2^8)$.

```
In [5]: rs = galois.ReedSolomon(85, 65, field=galois.GF(2**8)); rs
Out[5]: <Reed-Solomon Code: [85, 65, 21] over GF(2^8)>
```

(continues on next page)

(continued from previous page)

```
In [6]: rs.alpha
Out[6]: GF(8, order=2^8)

In [7]: rs.roots[0] == rs.alpha ** rs.c
Out[7]: True

In [8]: rs.alpha.multiplicative_order() == rs.n
Out[8]: True
```

property galois.ReedSolomon.**generator_poly** : *Poly*

The generator polynomial $g(x)$ over $\mathrm{GF}(q)$.

Notes

Every codeword $\mathbf{c}$ can be represented as a degree-n polynomial $c(x)$. Each codeword polynomial $c(x)$ is a multiple of $g(x)$.

Examples

Construct a narrow-sense $\mathrm{RS}(15, 9)$ code over $\mathrm{GF}(2^4)$ with first consecutive root α.

```
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))

In [3]: rs.roots
Out[3]: GF([ 2,  4,  8,  3,  6, 12], order=2^4)

# Evaluate the generator polynomial at its roots in GF(q)
In [4]: rs.generator_poly(rs.roots)
Out[4]: GF([0, 0, 0, 0, 0, 0], order=2^4)
```

Construct a non-narrow-sense $\mathrm{RS}(15, 9)$ code over $\mathrm{GF}(2^4)$ with first consecutive root α^3.

```
In [5]: rs = galois.ReedSolomon(15, 9, c=3); rs
Out[5]: <Reed-Solomon Code: [15, 9, 7] over GF(2^4)>

In [6]: rs.generator_poly
Out[6]: Poly(x^6 + 15x^5 + 8x^4 + 7x^3 + 9x^2 + 3x + 8, GF(2^4))

In [7]: rs.roots
Out[7]: GF([ 8,  3,  6, 12, 11,  5], order=2^4)

# Evaluate the generator polynomial at its roots in GF(q)
In [8]: rs.generator_poly(rs.roots)
Out[8]: GF([0, 0, 0, 0, 0, 0], order=2^4)
```

property galois.ReedSolomon.**parity_check_poly** : *Poly*

The parity-check polynomial $h(x)$.

Notes

The parity-check polynomial is the generator polynomial of the dual code.

Examples

Construct a primitive $\mathrm{RS}(15, 9)$ code over $\mathrm{GF}(2^4)$.

```
In [1]: rs = galois.ReedSolomon(15, 9); rs
Out[1]: <Reed-Solomon Code: [15, 9, 7] over GF(2^4)>

In [2]: rs.parity_check_poly
Out[2]: Poly(x^9 + 7x^8 + 15x^7 + 2x^6 + 5x^5 + 3x^4 + 3x^3 + 11x^2 + 15x + 10,␣
↪GF(2^4))

In [3]: rs.H
Out[3]:
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)
```

Construct a non-primitive $\mathrm{RS}(13, 9)$ code over $\mathrm{GF}(3^3)$.

```
In [4]: rs = galois.ReedSolomon(13, 9, field=galois.GF(3**3)); rs
Out[4]: <Reed-Solomon Code: [13, 9, 5] over GF(3^3)>

In [5]: rs.parity_check_poly
Out[5]: Poly(x^9 + 18x^8 + 10x^7 + 20x^6 + 17x^5 + 11x^4 + 24x^3 + 22x^2 + 25x␣
↪+ 26, GF(3^3))

In [6]: rs.H
Out[6]:
GF([[25,  8, 22, 16,  7,  6, 11, 12, 20, 13, 15,  9,  1],
    [ 8, 16,  6, 12, 13,  9, 25, 22,  7, 11, 20, 15,  1],
    [22,  6, 20,  9,  8,  7, 12, 15, 25, 16, 11, 13,  1],
    [16, 12,  9, 22, 11, 15,  8,  6, 13, 25,  7, 20,  1]], order=3^3)
```

property `galois.ReedSolomon.roots` : *FieldArray*

The $d - 1$ roots of the generator polynomial $g(x)$.

These are consecutive powers of α^c, specifically $\alpha^c, \dots, \alpha^{c+d-2}$.

Examples

Construct a narrow-sense $\mathrm{RS}(15, 9)$ code over $\mathrm{GF}(2^4)$ with first consecutive root α.

```
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)

In [3]: rs.generator_poly
Out[3]: Poly(x^6 + 7x^5 + 9x^4 + 3x^3 + 12x^2 + 10x + 12, GF(2^4))

# Evaluate the generator polynomial at its roots in GF(q)
In [4]: rs.generator_poly(rs.roots)
Out[4]: GF([0, 0, 0, 0, 0, 0], order=2^4)
```

Construct a non-narrow-sense $\mathrm{RS}(15, 9)$ code over $\mathrm{GF}(2^4)$ with first consecutive root α^3.

```
In [5]: rs = galois.ReedSolomon(15, 9, c=3); rs
Out[5]: <Reed-Solomon Code: [15, 9, 7] over GF(2^4)>

In [6]: rs.roots
Out[6]: GF([ 8,  3,  6, 12, 11,  5], order=2^4)

In [7]: rs.generator_poly
Out[7]: Poly(x^6 + 15x^5 + 8x^4 + 7x^3 + 9x^2 + 3x + 8, GF(2^4))

# Evaluate the generator polynomial at its roots in GF(q)
In [8]: rs.generator_poly(rs.roots)
Out[8]: GF([0, 0, 0, 0, 0, 0], order=2^4)
```

3.22 Linear sequences

class galois.***FLFSR***

A Fibonacci linear-feedback shift register (LFSR).

class galois.***GLFSR***

A Galois linear-feedback shift register (LFSR).

galois.***berlekamp_massey***(sequence: FieldArray, ...) → *Poly*

galois.***berlekamp_massey***(sequence: FieldArray, output) → *FLFSR*

galois.***berlekamp_massey***(sequence: FieldArray, output) → *GLFSR*

Finds the minimal polynomial $c(x)$ that produces the linear recurrent sequence y.

class galois.**FLFSR**

A Fibonacci linear-feedback shift register (LFSR).

Notes

A Fibonacci LFSR is defined by its feedback polynomial $f(x)$.

$$f(x) = -c_0x^n - c_1x^{n-1} - \cdots - c_{n-2}x^2 - c_{n-1}x + 1 = x^nc(x^{-1})$$

The feedback polynomial is the reciprocal of the characteristic polynomial $c(x)$ of the linear recurrent sequence y produced by the Fibonacci LFSR.

$$c(x) = x^n - c_{n-1}x^{n-1} - c_{n-2}x^{n-2} - \cdots - c_1x - c_0$$

$$y_t = c_{n-1}y_{t-1} + c_{n-2}y_{t-2} + \cdots + c_1y_{t-n+2} + c_0y_{t-n+1}$$

Listing 6: Fibonacci LFSR Configuration

```
+--------------+<-------------+<-------------+<-------------+
|              ^              ^              ^              |
|              | c_n-1        | c_n-2        | c_1          | c_0
|              | T[0]         | T[1]         | T[n-2]       | T[n-1]
|  +--------+  |  +--------+  |              |  +--------+  |
+->|  S[0]  |--+->|  S[1]  |--+---  ...   ---+->| S[n-1] |--+--> y[t]
   +--------+     +--------+                    +--------+
    y[t+n-1]       y[t+n-2]                       y[t+1]
```

The shift register taps T are defined left-to-right as $T = [T_0, T_1, \dots, T_{n-2}, T_{n-1}]$. The state vector S is also defined left-to-right as $S = [S_0, S_1, \dots, S_{n-2}, S_{n-1}]$.

In the Fibonacci configuration, the shift register taps are $T = [c_{n-1}, c_{n-2}, \dots, c_1, c_0]$. Additionally, the state vector is equal to the next n outputs in reversed order, namely $S = [y_{t+n-1}, y_{t+n-2}, \dots, y_{t+2}, y_{t+1}]$.

References

- Gardner, D. 2019. "Applications of the Galois Model LFSR in Cryptography". figshare. https://hdl.handle.net/2134/21932.

See also

berlekamp_massey

Examples

GF(2)

Create a Fibonacci LFSR from a degree-4 primitive characteristic polynomial over $\mathrm{GF}(2)$.

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

In [2]: lfsr = galois.FLFSR(c.reverse())

In [3]: print(lfsr)
Fibonacci LFSR:
  field: GF(2)
  feedback_poly: x^4 + x^3 + 1
  characteristic_poly: x^4 + x + 1
  taps: [0 0 1 1]
  order: 4
  state: [1 1 1 1]
  initial_state: [1 1 1 1]
```

Step the Fibonacci LFSR and produce 10 output symbols.

```
In [4]: lfsr.state
Out[4]: GF([1, 1, 1, 1], order=2)

In [5]: lfsr.step(10)
Out[5]: GF([1, 1, 1, 1, 0, 0, 0, 1, 0, 0], order=2)

In [6]: lfsr.state
Out[6]: GF([1, 0, 1, 1], order=2)
```

GF(p)

Create a Fibonacci LFSR from a degree-4 primitive characteristic polynomial over $\mathrm{GF}(7)$.

```
In [7]: c = galois.primitive_poly(7, 4); c
Out[7]: Poly(x^4 + x^2 + 3x + 5, GF(7))

In [8]: lfsr = galois.FLFSR(c.reverse())

In [9]: print(lfsr)
Fibonacci LFSR:
  field: GF(7)
  feedback_poly: 5x^4 + 3x^3 + x^2 + 1
  characteristic_poly: x^4 + x^2 + 3x + 5
  taps: [0 6 4 2]
  order: 4
  state: [1 1 1 1]
  initial_state: [1 1 1 1]
```

Step the Fibonacci LFSR and produce 10 output symbols.

```
In [10]: lfsr.state
Out[10]: GF([1, 1, 1, 1], order=7)

In [11]: lfsr.step(10)
Out[11]: GF([1, 1, 1, 1, 5, 5, 1, 3, 1, 4], order=7)

In [12]: lfsr.state
Out[12]: GF([5, 5, 6, 6], order=7)
```

GF(2^m)

Create a Fibonacci LFSR from a degree-4 primitive characteristic polynomial over $\mathrm{GF}(2^3)$.

```
In [13]: c = galois.primitive_poly(2**3, 4); c
Out[13]: Poly(x^4 + x + 3, GF(2^3))

In [14]: lfsr = galois.FLFSR(c.reverse())

In [15]: print(lfsr)
Fibonacci LFSR:
  field: GF(2^3)
  feedback_poly: 3x^4 + x^3 + 1
  characteristic_poly: x^4 + x + 3
  taps: [0 0 1 3]
  order: 4
  state: [1 1 1 1]
  initial_state: [1 1 1 1]
```

Step the Fibonacci LFSR and produce 10 output symbols.

```
In [16]: lfsr.state
Out[16]: GF([1, 1, 1, 1], order=2^3)

In [17]: lfsr.step(10)
Out[17]: GF([1, 1, 1, 1, 2, 2, 2, 1, 4, 4], order=2^3)

In [18]: lfsr.state
Out[18]: GF([0, 3, 7, 7], order=2^3)
```

GF(p^m)

Create a Fibonacci LFSR from a degree-4 primitive characteristic polynomial over $\mathrm{GF}(3^3)$.

```
In [19]: c = galois.primitive_poly(3**3, 4); c
Out[19]: Poly(x^4 + x + 10, GF(3^3))

In [20]: lfsr = galois.FLFSR(c.reverse())

In [21]: print(lfsr)
Fibonacci LFSR:
  field: GF(3^3)
```

(continues on next page)

(continued from previous page)

```
feedback_poly: 10x^4 + x^3 + 1
characteristic_poly: x^4 + x + 10
taps: [ 0  0  2 20]
order: 4
state: [1 1 1 1]
initial_state: [1 1 1 1]
```

Step the Fibonacci LFSR and produce 10 output symbols.

```
In [22]: lfsr.state
Out[22]: GF([1, 1, 1, 1], order=3^3)

In [23]: lfsr.step(10)
Out[23]: GF([ 1,  1,  1,  1, 19, 19, 19,  1, 25, 25], order=3^3)

In [24]: lfsr.state
Out[24]: GF([ 6, 24,  4, 16], order=3^3)
```

Constructors

FLFSR(feedback_poly: Poly, state: ArrayLike | *None* = ***None***)

Constructs a Fibonacci LFSR from its feedback polynomial $f(x)$.

classmethod ***Taps***(taps: FieldArray, ...) → Self

Constructs a Fibonacci LFSR from its taps $T = [c_{n-1}, c_{n-2}, \dots, c_1, c_0]$.

galois.**FLFSR**(feedback_poly: Poly, state: ArrayLike | *None* = ***None***)

Constructs a Fibonacci LFSR from its feedback polynomial $f(x)$.

Parameters

feedback_poly: ***Poly***
The feedback polynomial $f(x) = -c_0 x^n - c_1 x^{n-1} - \dots - c_{n-2} x^2 - c_{n-1} x + 1$.

state: ***ArrayLike*** **| None = None**
The initial state vector $S = [S_0, S_1, \dots, S_{n-2}, S_{n-1}]$. The default is `None` which corresponds to all ones.

See also

irreducible_poly, *primitive_poly*

Notes

A Fibonacci LFSR may be constructed from its characteristic polynomial $c(x)$ by passing in its reciprocal as the feedback polynomial. This is because $f(x) = x^n c(x^{-1})$.

classmethod galois.FLFSR.**Taps**(taps: FieldArray, state: ArrayLike | *None* = ***None***) → Self

Constructs a Fibonacci LFSR from its taps $T = [c_{n-1}, c_{n-2}, \dots, c_1, c_0]$.

Parameters

taps: *FieldArray*
The shift register taps $T = [c_{n-1}, c_{n-2}, \dots, c_1, c_0]$.

state: *ArrayLike* | None = None
The initial state vector $S = [S_0, S_1, \dots, S_{n-2}, S_{n-1}]$. The default is `None` which corresponds to all ones.

Returns
A Fibonacci LFSR with taps $T = [c_{n-1}, c_{n-2}, \dots, c_1, c_0]$.

Examples

```
In [1]: c = galois.primitive_poly(7, 4); c
Out[1]: Poly(x^4 + x^2 + 3x + 5, GF(7))

In [2]: taps = -c.coeffs[1:]; taps
Out[2]: GF([0, 6, 4, 2], order=7)

In [3]: lfsr = galois.FLFSR.Taps(taps)

In [4]: print(lfsr)
Fibonacci LFSR:
  field: GF(7)
  feedback_poly: 5x^4 + 3x^3 + x^2 + 1
  characteristic_poly: x^4 + x^2 + 3x + 5
  taps: [0 6 4 2]
  order: 4
  state: [1 1 1 1]
  initial_state: [1 1 1 1]
```

String representation

__repr__() → str
A terse representation of the Fibonacci LFSR.

__str__() → str
A formatted string of relevant properties of the Fibonacci LFSR.

`galois.FLFSR.`**`__repr__`**() → str
A terse representation of the Fibonacci LFSR.

Examples

```
In [1]: c = galois.primitive_poly(7, 4); c
Out[1]: Poly(x^4 + x^2 + 3x + 5, GF(7))

In [2]: lfsr = galois.FLFSR(c.reverse())

In [3]: lfsr
Out[3]: <Fibonacci LFSR: f(x) = 5x^4 + 3x^3 + x^2 + 1 over GF(7)>
```

galois.FLFSR.**__str__**() → str

A formatted string of relevant properties of the Fibonacci LFSR.

Examples

```
In [1]: c = galois.primitive_poly(7, 4); c
Out[1]: Poly(x^4 + x^2 + 3x + 5, GF(7))

In [2]: lfsr = galois.FLFSR(c.reverse())

In [3]: print(lfsr)
Fibonacci LFSR:
  field: GF(7)
  feedback_poly: 5x^4 + 3x^3 + x^2 + 1
  characteristic_poly: x^4 + x^2 + 3x + 5
  taps: [0 6 4 2]
  order: 4
  state: [1 1 1 1]
  initial_state: [1 1 1 1]
```

Methods

reset(state: ArrayLike | *None* = ***None***)

Resets the Fibonacci LFSR state to the specified state.

step(steps: *int* = *1*) → *FieldArray*

Produces the next `steps` output symbols.

to_galois_lfsr() → *GLFSR*

Converts the Fibonacci LFSR to a Galois LFSR that produces the same output.

galois.FLFSR.**reset**(state: ArrayLike | *None* = ***None***)

Resets the Fibonacci LFSR state to the specified state.

Parameters

state: *ArrayLike* | None = None
The state vector $S = [S_0, S_1, \dots, S_{n-2}, S_{n-1}]$. The default is `None` which corresponds to the initial state.

Examples

Step the Fibonacci LFSR 10 steps to modify its state.

```
In [1]: c = galois.primitive_poly(7, 4); c
Out[1]: Poly(x^4 + x^2 + 3x + 5, GF(7))

In [2]: lfsr = galois.FLFSR(c.reverse()); lfsr
Out[2]: <Fibonacci LFSR: f(x) = 5x^4 + 3x^3 + x^2 + 1 over GF(7)>

In [3]: lfsr.state
Out[3]: GF([1, 1, 1, 1], order=7)
```

(continues on next page)

(continued from previous page)

```
In [4]: lfsr.step(10)
Out[4]: GF([1, 1, 1, 1, 5, 5, 1, 3, 1, 4], order=7)

In [5]: lfsr.state
Out[5]: GF([5, 5, 6, 6], order=7)
```

Reset the Fibonacci LFSR state.

```
In [6]: lfsr.reset()

In [7]: lfsr.state
Out[7]: GF([1, 1, 1, 1], order=7)
```

Create an Fibonacci LFSR and view its initial state.

```
In [8]: c = galois.primitive_poly(7, 4); c
Out[8]: Poly(x^4 + x^2 + 3x + 5, GF(7))

In [9]: lfsr = galois.FLFSR(c.reverse()); lfsr
Out[9]: <Fibonacci LFSR: f(x) = 5x^4 + 3x^3 + x^2 + 1 over GF(7)>

In [10]: lfsr.state
Out[10]: GF([1, 1, 1, 1], order=7)
```

Reset the Fibonacci LFSR state to a new state.

```
In [11]: lfsr.reset([1, 2, 3, 4])

In [12]: lfsr.state
Out[12]: GF([1, 2, 3, 4], order=7)
```

`galois.FLFSR.step(steps: int = 1)` → *FieldArray*

Produces the next `steps` output symbols.

Parameters

steps: int = 1
The direction and number of output symbols to produce. The default is 1. If negative, the Fibonacci LFSR will step backwards.

Returns
An array of output symbols of type *field* with size `abs(steps)`.

Examples

Step the Fibonacci LFSR one output at a time. Notice the first n outputs of a Fibonacci LFSR are its state reversed.

```
In [1]: c = galois.primitive_poly(7, 4)

In [2]: lfsr = galois.FLFSR(c.reverse(), state=[1, 2, 3, 4]); lfsr
Out[2]: <Fibonacci LFSR: f(x) = 5x^4 + 3x^3 + x^2 + 1 over GF(7)>
```

(continues on next page)

(continued from previous page)

```
In [3]: lfsr.state, lfsr.step()
Out[3]: (GF([1, 2, 3, 4], order=7), GF(4, order=7))

In [4]: lfsr.state, lfsr.step()
Out[4]: (GF([4, 1, 2, 3], order=7), GF(3, order=7))

In [5]: lfsr.state, lfsr.step()
Out[5]: (GF([6, 4, 1, 2], order=7), GF(2, order=7))

In [6]: lfsr.state, lfsr.step()
Out[6]: (GF([4, 6, 4, 1], order=7), GF(1, order=7))

In [7]: lfsr.state, lfsr.step()
Out[7]: (GF([5, 4, 6, 4], order=7), GF(4, order=7))

# Ending state
In [8]: lfsr.state
Out[8]: GF([0, 5, 4, 6], order=7)
```

Step the Fibonacci LFSR 5 steps in one call. This is more efficient than iterating one output at a time.

```
In [9]: c = galois.primitive_poly(7, 4)

In [10]: lfsr = galois.FLFSR(c.reverse(), state=[1, 2, 3, 4]); lfsr
Out[10]: <Fibonacci LFSR: f(x) = 5x^4 + 3x^3 + x^2 + 1 over GF(7)>

In [11]: lfsr.state
Out[11]: GF([1, 2, 3, 4], order=7)

In [12]: lfsr.step(5)
Out[12]: GF([4, 3, 2, 1, 4], order=7)

# Ending state
In [13]: lfsr.state
Out[13]: GF([0, 5, 4, 6], order=7)
```

Step the Fibonacci LFSR 5 steps backward. Notice the output sequence is the reverse of the original sequence. Also notice the ending state is the same as the initial state.

```
In [14]: lfsr.step(-5)
Out[14]: GF([4, 1, 2, 3, 4], order=7)

In [15]: lfsr.state
Out[15]: GF([1, 2, 3, 4], order=7)
```

galois.FLFSR.**to_galois_lfsr**() → *GLFSR*

Converts the Fibonacci LFSR to a Galois LFSR that produces the same output.

Returns

An equivalent Galois LFSR.

Examples

Create a Fibonacci LFSR with a given initial state.

```
In [1]: c = galois.primitive_poly(7, 4); c
Out[1]: Poly(x^4 + x^2 + 3x + 5, GF(7))

In [2]: fibonacci_lfsr = galois.FLFSR(c.reverse(), state=[1, 2, 3, 4])

In [3]: print(fibonacci_lfsr)
Fibonacci LFSR:
  field: GF(7)
  feedback_poly: 5x^4 + 3x^3 + x^2 + 1
  characteristic_poly: x^4 + x^2 + 3x + 5
  taps: [0 6 4 2]
  order: 4
  state: [1 2 3 4]
  initial_state: [1 2 3 4]
```

Convert the Fibonacci LFSR to an equivalent Galois LFSR. Notice the initial state is different.

```
In [4]: galois_lfsr = fibonacci_lfsr.to_galois_lfsr()

In [5]: print(galois_lfsr)
Galois LFSR:
  field: GF(7)
  feedback_poly: 5x^4 + 3x^3 + x^2 + 1
  characteristic_poly: x^4 + x^2 + 3x + 5
  taps: [2 4 6 0]
  order: 4
  state: [2 6 3 4]
  initial_state: [2 6 3 4]
```

Step both LFSRs and see that their output sequences are identical.

```
In [6]: fibonacci_lfsr.step(10)
Out[6]: GF([4, 3, 2, 1, 4, 6, 4, 5, 0, 2], order=7)

In [7]: galois_lfsr.step(10)
Out[7]: GF([4, 3, 2, 1, 4, 6, 4, 5, 0, 2], order=7)
```

Properties

property ***field*** : type[*FieldArray*]

The `FieldArray` subclass for the finite field that defines the linear arithmetic.

property ***order*** : int

The order of the linear recurrence/linear recurrent sequence. The order of a sequence is defined by the degree of the minimal polynomial that produces it.

property ***taps*** : *FieldArray*

The shift register taps $T = [c_{n-1}, c_{n-2}, \dots, c_1, c_0]$. The taps of the shift register define the linear recurrence relation.

property galois.FLFSR.**field** : type[*FieldArray*]

The *FieldArray* subclass for the finite field that defines the linear arithmetic.

Examples

```
In [1]: c = galois.primitive_poly(7, 4); c
Out[1]: Poly(x^4 + x^2 + 3x + 5, GF(7))

In [2]: lfsr = galois.FLFSR(c.reverse()); lfsr
Out[2]: <Fibonacci LFSR: f(x) = 5x^4 + 3x^3 + x^2 + 1 over GF(7)>

In [3]: lfsr.field
Out[3]: <class 'galois.GF(7)'>
```

property galois.FLFSR.**order** : int

The order of the linear recurrence/linear recurrent sequence. The order of a sequence is defined by the degree of the minimal polynomial that produces it.

property galois.FLFSR.**taps** : *FieldArray*

The shift register taps $T = [c_{n-1}, c_{n-2}, \dots, c_1, c_0]$. The taps of the shift register define the linear recurrence relation.

Examples

```
In [1]: c = galois.primitive_poly(7, 4); c
Out[1]: Poly(x^4 + x^2 + 3x + 5, GF(7))

In [2]: taps = -c.coeffs[1:]; taps
Out[2]: GF([0, 6, 4, 2], order=7)

In [3]: lfsr = galois.FLFSR.Taps(taps); lfsr
Out[3]: <Fibonacci LFSR: f(x) = 5x^4 + 3x^3 + x^2 + 1 over GF(7)>

In [4]: lfsr.taps
Out[4]: GF([0, 6, 4, 2], order=7)
```

Polynomials

property ***characteristic_poly*** : *Poly*

The characteristic polynomial $c(x) = x^n - c_{n-1}x^{n-1} - c_{n-2}x^{n-2} - \dots - c_1 x - c_0$ that defines the linear recurrent sequence.

property ***feedback_poly*** : *Poly*

The feedback polynomial $f(x) = -c_0 x^n - c_1 x^{n-1} - \dots - c_{n-2}x^2 - c_{n-1}x + 1$ that defines the feedback arithmetic.

property galois.FLFSR.**characteristic_poly** : *Poly*

The characteristic polynomial $c(x) = x^n - c_{n-1}x^{n-1} - c_{n-2}x^{n-2} - \dots - c_1 x - c_0$ that defines the linear recurrent sequence.

Notes

The characteristic polynomial is the reciprocal of the feedback polynomial $c(x) = x^n f(x^{-1})$.

Examples

```
In [1]: c = galois.primitive_poly(7, 4); c
Out[1]: Poly(x^4 + x^2 + 3x + 5, GF(7))

In [2]: lfsr = galois.FLFSR(c.reverse()); lfsr
Out[2]: <Fibonacci LFSR: f(x) = 5x^4 + 3x^3 + x^2 + 1 over GF(7)>

In [3]: lfsr.characteristic_poly
Out[3]: Poly(x^4 + x^2 + 3x + 5, GF(7))

In [4]: lfsr.characteristic_poly == lfsr.feedback_poly.reverse()
Out[4]: True
```

property galois.FLFSR.**feedback_poly** : *Poly*

The feedback polynomial $f(x) = -c_0 x^n - c_1 x^{n-1} - \dots - c_{n-2} x^2 - c_{n-1} x + 1$ that defines the feedback arithmetic.

Notes

The feedback polynomial is the reciprocal of the characteristic polynomial $f(x) = x^n c(x^{-1})$.

Examples

```
In [1]: c = galois.primitive_poly(7, 4); c
Out[1]: Poly(x^4 + x^2 + 3x + 5, GF(7))

In [2]: lfsr = galois.FLFSR(c.reverse()); lfsr
Out[2]: <Fibonacci LFSR: f(x) = 5x^4 + 3x^3 + x^2 + 1 over GF(7)>

In [3]: lfsr.feedback_poly
Out[3]: Poly(5x^4 + 3x^3 + x^2 + 1, GF(7))

In [4]: lfsr.feedback_poly == lfsr.characteristic_poly.reverse()
Out[4]: True
```

State

property ***initial_state*** : *FieldArray*

The initial state vector $S = [S_0, S_1, \dots, S_{n-2}, S_{n-1}]$.

property ***state*** : *FieldArray*

The current state vector $S = [S_0, S_1, \dots, S_{n-2}, S_{n-1}]$.

property galois.FLFSR.**initial_state** : *FieldArray*

The initial state vector $S = [S_0, S_1, \dots, S_{n-2}, S_{n-1}]$.

Examples

```
In [1]: c = galois.primitive_poly(7, 4)

In [2]: lfsr = galois.FLFSR(c.reverse(), state=[1, 2, 3, 4]); lfsr
Out[2]: <Fibonacci LFSR: f(x) = 5x^4 + 3x^3 + x^2 + 1 over GF(7)>

In [3]: lfsr.initial_state
Out[3]: GF([1, 2, 3, 4], order=7)
```

The initial state is unaffected as the Fibonacci LFSR is stepped.

```
In [4]: lfsr.step(10)
Out[4]: GF([4, 3, 2, 1, 4, 6, 4, 5, 0, 2], order=7)

In [5]: lfsr.initial_state
Out[5]: GF([1, 2, 3, 4], order=7)
```

property `galois.FLFSR.state` : *FieldArray*

The current state vector $S = [S_0, S_1, \dots, S_{n-2}, S_{n-1}]$.

Examples

```
In [1]: c = galois.primitive_poly(7, 4)

In [2]: lfsr = galois.FLFSR(c.reverse(), state=[1, 2, 3, 4]); lfsr
Out[2]: <Fibonacci LFSR: f(x) = 5x^4 + 3x^3 + x^2 + 1 over GF(7)>

In [3]: lfsr.state
Out[3]: GF([1, 2, 3, 4], order=7)
```

The current state is modified as the Fibonacci LFSR is stepped.

```
In [4]: lfsr.step(10)
Out[4]: GF([4, 3, 2, 1, 4, 6, 4, 5, 0, 2], order=7)

In [5]: lfsr.state
Out[5]: GF([3, 1, 1, 0], order=7)
```

class `galois.GLFSR`

A Galois linear-feedback shift register (LFSR).

Notes

A Galois LFSR is defined by its feedback polynomial $f(x)$.

$$f(x) = -c_0x^n - c_1x^{n-1} - \dots - c_{n-2}x^2 - c_{n-1}x + 1 = x^n c(x^{-1})$$

The feedback polynomial is the reciprocal of the characteristic polynomial $c(x)$ of the linear recurrent sequence y produced by the Galois LFSR.

$$c(x) = x^n - c_{n-1}x^{n-1} - c_{n-2}x^{n-2} - \cdots - c_1 x - c_0$$

$$y_t = c_{n-1}y_{t-1} + c_{n-2}y_{t-2} + \cdots + c_1 y_{t-n+2} + c_0 y_{t-n+1}$$

Listing 7: Galois LFSR Configuration

```
+--------------+<-------------+<-------------+<-------------+
|              |              |              |              |
| c_0          | c_1          | c_2          | c_n-1        |
| T[0]         | T[1]         | T[2]         | T[n-1]       |
|  +--------+  v  +--------+  v              v  +--------+  |
+->|  S[0]  |--+->|  S[1]  |--+---  ...   ---+->| S[n-1] |--+--> y[t]
   +--------+     +--------+                    +--------+
                                                  y[t+1]
```

The shift register taps T are defined left-to-right as $T = [T_0, T_1, \dots, T_{n-2}, T_{n-1}]$. The state vector S is also defined left-to-right as $S = [S_0, S_1, \dots, S_{n-2}, S_{n-1}]$.

In the Galois configuration, the shift register taps are $T = [c_0, c_1, \dots, c_{n-2}, c_{n-1}]$.

References

- Gardner, D. 2019. "Applications of the Galois Model LFSR in Cryptography". figshare. https://hdl.handle.net/2134/21932.

See also

berlekamp_massey

Examples

GF(2)

Create a Galois LFSR from a degree-4 primitive characteristic polynomial over $\mathrm{GF}(2)$.

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

In [2]: lfsr = galois.GLFSR(c.reverse())

In [3]: print(lfsr)
Galois LFSR:
  field: GF(2)
  feedback_poly: x^4 + x^3 + 1
  characteristic_poly: x^4 + x + 1
  taps: [1 1 0 0]
```

(continues on next page)

(continued from previous page)

```
  order: 4
  state: [1 1 1 1]
  initial_state: [1 1 1 1]
```

Step the Galois LFSR and produce 10 output symbols.

```
In [4]: lfsr.state
Out[4]: GF([1, 1, 1, 1], order=2)

In [5]: lfsr.step(10)
Out[5]: GF([1, 1, 1, 0, 0, 0, 1, 0, 0, 1], order=2)

In [6]: lfsr.state
Out[6]: GF([1, 1, 0, 1], order=2)
```

GF(p)

Create a Galois LFSR from a degree-4 primitive characteristic polynomial over $\mathrm{GF}(7)$.

```
In [7]: c = galois.primitive_poly(7, 4); c
Out[7]: Poly(x^4 + x^2 + 3x + 5, GF(7))

In [8]: lfsr = galois.GLFSR(c.reverse())

In [9]: print(lfsr)
Galois LFSR:
  field: GF(7)
  feedback_poly: 5x^4 + 3x^3 + x^2 + 1
  characteristic_poly: x^4 + x^2 + 3x + 5
  taps: [2 4 6 0]
  order: 4
  state: [1 1 1 1]
  initial_state: [1 1 1 1]
```

Step the Galois LFSR and produce 10 output symbols.

```
In [10]: lfsr.state
Out[10]: GF([1, 1, 1, 1], order=7)

In [11]: lfsr.step(10)
Out[11]: GF([1, 1, 0, 4, 6, 5, 3, 6, 1, 2], order=7)

In [12]: lfsr.state
Out[12]: GF([4, 3, 0, 1], order=7)
```

GF(2^m)

Create a Galois LFSR from a degree-4 primitive characteristic polynomial over $\mathrm{GF}(2^3)$.

```
In [13]: c = galois.primitive_poly(2**3, 4); c
Out[13]: Poly(x^4 + x + 3, GF(2^3))

In [14]: lfsr = galois.GLFSR(c.reverse())

In [15]: print(lfsr)
Galois LFSR:
  field: GF(2^3)
  feedback_poly: 3x^4 + x^3 + 1
  characteristic_poly: x^4 + x + 3
  taps: [3 1 0 0]
  order: 4
  state: [1 1 1 1]
  initial_state: [1 1 1 1]
```

Step the Galois LFSR and produce 10 output symbols.

```
In [16]: lfsr.state
Out[16]: GF([1, 1, 1, 1], order=2^3)

In [17]: lfsr.step(10)
Out[17]: GF([1, 1, 1, 0, 2, 2, 3, 2, 4, 5], order=2^3)

In [18]: lfsr.state
Out[18]: GF([4, 2, 2, 7], order=2^3)
```

GF(p^m)

Create a Galois LFSR from a degree-4 primitive characteristic polynomial over $\mathrm{GF}(3^3)$.

```
In [19]: c = galois.primitive_poly(3**3, 4); c
Out[19]: Poly(x^4 + x + 10, GF(3^3))

In [20]: lfsr = galois.GLFSR(c.reverse())

In [21]: print(lfsr)
Galois LFSR:
  field: GF(3^3)
  feedback_poly: 10x^4 + x^3 + 1
  characteristic_poly: x^4 + x + 10
  taps: [20  2  0  0]
  order: 4
  state: [1 1 1 1]
  initial_state: [1 1 1 1]
```

Step the Galois LFSR and produce 10 output symbols.

```
In [22]: lfsr.state
Out[22]: GF([1, 1, 1, 1], order=3^3)
```

(continues on next page)

(continued from previous page)

```
In [23]: lfsr.step(10)
Out[23]: GF([ 1,  1,  1,  0, 19, 19, 20, 11, 25, 24], order=3^3)

In [24]: lfsr.state
Out[24]: GF([23, 25,  6, 26], order=3^3)
```

Constructors

GLFSR(feedback_poly: Poly, state: ArrayLike | *None* = ***None***)

Constructs a Galois LFSR from its feedback polynomial $f(x)$.

classmethod *Taps*(taps: FieldArray, ...) → Self

Constructs a Galois LFSR from its taps $T = [c_0, c_1, \dots, c_{n-2}, c_{n-1}]$.

galois.**GLFSR**(feedback_poly: Poly, state: ArrayLike | *None* = ***None***)

Constructs a Galois LFSR from its feedback polynomial $f(x)$.

Parameters

feedback_poly: *Poly*
The feedback polynomial $f(x) = -c_0x^n - c_1x^{n-1} - \dots - c_{n-2}x^2 - c_{n-1}x + 1$.

state: *ArrayLike* | None = None
The initial state vector $S = [S_0, S_1, \dots, S_{n-2}, S_{n-1}]$. The default is `None` which corresponds to all ones.

See also

irreducible_poly, *primitive_poly*

Notes

A Galois LFSR may be constructed from its characteristic polynomial $c(x)$ by passing in its reciprocal as the feedback polynomial. This is because $f(x) = x^n c(x^{-1})$.

classmethod galois.GLFSR.**Taps**(taps: FieldArray, state: ArrayLike | *None* = ***None***) → Self

Constructs a Galois LFSR from its taps $T = [c_0, c_1, \dots, c_{n-2}, c_{n-1}]$.

Parameters

taps: *FieldArray*
The shift register taps $T = [c_0, c_1, \dots, c_{n-2}, c_{n-1}]$.

state: *ArrayLike* | None = None
The initial state vector $S = [S_0, S_1, \dots, S_{n-2}, S_{n-1}]$. The default is `None` which corresponds to all ones.

Returns
A Galois LFSR with taps $T = [c_0, c_1, \dots, c_{n-2}, c_{n-1}]$.

Examples

```
In [1]: c = galois.primitive_poly(7, 4); c
Out[1]: Poly(x^4 + x^2 + 3x + 5, GF(7))

In [2]: taps = -c.coeffs[1:][::-1]; taps
Out[2]: GF([2, 4, 6, 0], order=7)

In [3]: lfsr = galois.GLFSR.Taps(taps)

In [4]: print(lfsr)
Galois LFSR:
  field: GF(7)
  feedback_poly: 5x^4 + 3x^3 + x^2 + 1
  characteristic_poly: x^4 + x^2 + 3x + 5
  taps: [2 4 6 0]
  order: 4
  state: [1 1 1 1]
  initial_state: [1 1 1 1]
```

String representation

__repr__() → str

A terse representation of the Galois LFSR.

__str__() → str

A formatted string of relevant properties of the Galois LFSR.

galois.GLFSR.**__repr__**() → str

A terse representation of the Galois LFSR.

Examples

```
In [1]: c = galois.primitive_poly(7, 4); c
Out[1]: Poly(x^4 + x^2 + 3x + 5, GF(7))

In [2]: lfsr = galois.GLFSR(c.reverse())

In [3]: lfsr
Out[3]: <Galois LFSR: f(x) = 5x^4 + 3x^3 + x^2 + 1 over GF(7)>
```

galois.GLFSR.**__str__**() → str

A formatted string of relevant properties of the Galois LFSR.

Examples

```
In [1]: c = galois.primitive_poly(7, 4); c
Out[1]: Poly(x^4 + x^2 + 3x + 5, GF(7))

In [2]: lfsr = galois.GLFSR(c.reverse())

In [3]: print(lfsr)
Galois LFSR:
  field: GF(7)
  feedback_poly: 5x^4 + 3x^3 + x^2 + 1
  characteristic_poly: x^4 + x^2 + 3x + 5
  taps: [2 4 6 0]
  order: 4
  state: [1 1 1 1]
  initial_state: [1 1 1 1]
```

Methods

reset(state: ArrayLike | *None* = ***None***)

Resets the Galois LFSR state to the specified state.

step(steps: *int* = *1*) → *FieldArray*

Produces the next `steps` output symbols.

to_fibonacci_lfsr() → *FLFSR*

Converts the Galois LFSR to a Fibonacci LFSR that produces the same output.

galois.GLFSR.**reset**(state: ArrayLike | *None* = ***None***)

Resets the Galois LFSR state to the specified state.

Parameters

state: *ArrayLike* | None = None

The state vector $S = [S_0, S_1, \dots, S_{n-2}, S_{n-1}]$. The default is `None` which corresponds to the initial state.

Examples

Step the Galois LFSR 10 steps to modify its state.

```
In [1]: c = galois.primitive_poly(7, 4); c
Out[1]: Poly(x^4 + x^2 + 3x + 5, GF(7))

In [2]: lfsr = galois.GLFSR(c.reverse()); lfsr
Out[2]: <Galois LFSR: f(x) = 5x^4 + 3x^3 + x^2 + 1 over GF(7)>

In [3]: lfsr.state
Out[3]: GF([1, 1, 1, 1], order=7)

In [4]: lfsr.step(10)
Out[4]: GF([1, 1, 0, 4, 6, 5, 3, 6, 1, 2], order=7)
```

(continues on next page)

(continued from previous page)

```
In [5]: lfsr.state
Out[5]: GF([4, 3, 0, 1], order=7)
```

Reset the Galois LFSR state.

```
In [6]: lfsr.reset()

In [7]: lfsr.state
Out[7]: GF([1, 1, 1, 1], order=7)
```

Create an Galois LFSR and view its initial state.

```
In [8]: c = galois.primitive_poly(7, 4); c
Out[8]: Poly(x^4 + x^2 + 3x + 5, GF(7))

In [9]: lfsr = galois.GLFSR(c.reverse()); lfsr
Out[9]: <Galois LFSR: f(x) = 5x^4 + 3x^3 + x^2 + 1 over GF(7)>

In [10]: lfsr.state
Out[10]: GF([1, 1, 1, 1], order=7)
```

Reset the Galois LFSR state to a new state.

```
In [11]: lfsr.reset([1, 2, 3, 4])

In [12]: lfsr.state
Out[12]: GF([1, 2, 3, 4], order=7)
```

`galois.GLFSR.step`(steps: *int* = *1*) → *FieldArray*

Produces the next `steps` output symbols.

Parameters

steps: int = 1
The direction and number of output symbols to produce. The default is 1. If negative, the Galois LFSR will step backwards.

Returns
An array of output symbols of type *field* with size `abs(steps)`.

Examples

Step the Galois LFSR one output at a time.

```
In [1]: c = galois.primitive_poly(7, 4)

In [2]: lfsr = galois.GLFSR(c.reverse(), state=[1, 2, 3, 4]); lfsr
Out[2]: <Galois LFSR: f(x) = 5x^4 + 3x^3 + x^2 + 1 over GF(7)>

In [3]: lfsr.state, lfsr.step()
Out[3]: (GF([1, 2, 3, 4], order=7), GF(4, order=7))

In [4]: lfsr.state, lfsr.step()
```

(continues on next page)

(continued from previous page)

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

In [5]: lfsr.state, lfsr.step()
Out[5]: (GF([6, 6, 0, 5], order=7), GF(5, order=7))

In [6]: lfsr.state, lfsr.step()
Out[6]: (GF([3, 5, 1, 0], order=7), GF(0, order=7))

In [7]: lfsr.state, lfsr.step()
Out[7]: (GF([0, 3, 5, 1], order=7), GF(1, order=7))

# Ending state
In [8]: lfsr.state
Out[8]: GF([2, 4, 2, 5], order=7)
```

Step the Galois LFSR 5 steps in one call. This is more efficient than iterating one output at a time.

```
In [9]: c = galois.primitive_poly(7, 4)

In [10]: lfsr = galois.GLFSR(c.reverse(), state=[1, 2, 3, 4]); lfsr
Out[10]: <Galois LFSR: f(x) = 5x^4 + 3x^3 + x^2 + 1 over GF(7)>

In [11]: lfsr.state
Out[11]: GF([1, 2, 3, 4], order=7)

In [12]: lfsr.step(5)
Out[12]: GF([4, 3, 5, 0, 1], order=7)

# Ending state
In [13]: lfsr.state
Out[13]: GF([2, 4, 2, 5], order=7)
```

Step the Galois LFSR 5 steps backward. Notice the output sequence is the reverse of the original sequence. Also notice the ending state is the same as the initial state.

```
In [14]: lfsr.step(-5)
Out[14]: GF([1, 0, 5, 3, 4], order=7)

In [15]: lfsr.state
Out[15]: GF([1, 2, 3, 4], order=7)
```

galois.GLFSR.**to_fibonacci_lfsr**() → *FLFSR*

Converts the Galois LFSR to a Fibonacci LFSR that produces the same output.

Returns

An equivalent Fibonacci LFSR.

Examples

Create a Galois LFSR with a given initial state.

```
In [1]: c = galois.primitive_poly(7, 4); c
Out[1]: Poly(x^4 + x^2 + 3x + 5, GF(7))

In [2]: galois_lfsr = galois.GLFSR(c.reverse(), state=[1, 2, 3, 4])

In [3]: print(galois_lfsr)
Galois LFSR:
  field: GF(7)
  feedback_poly: 5x^4 + 3x^3 + x^2 + 1
  characteristic_poly: x^4 + x^2 + 3x + 5
  taps: [2 4 6 0]
  order: 4
  state: [1 2 3 4]
  initial_state: [1 2 3 4]
```

Convert the Galois LFSR to an equivalent Fibonacci LFSR. Notice the initial state is different.

```
In [4]: fibonacci_lfsr = galois_lfsr.to_fibonacci_lfsr()

In [5]: print(fibonacci_lfsr)
Fibonacci LFSR:
  field: GF(7)
  feedback_poly: 5x^4 + 3x^3 + x^2 + 1
  characteristic_poly: x^4 + x^2 + 3x + 5
  taps: [0 6 4 2]
  order: 4
  state: [0 5 3 4]
  initial_state: [0 5 3 4]
```

Step both LFSRs and see that their output sequences are identical.

```
In [6]: galois_lfsr.step(10)
Out[6]: GF([4, 3, 5, 0, 1, 5, 2, 6, 6, 5], order=7)

In [7]: fibonacci_lfsr.step(10)
Out[7]: GF([4, 3, 5, 0, 1, 5, 2, 6, 6, 5], order=7)
```

Properties

property ***field*** : type[*FieldArray*]

The *FieldArray* subclass for the finite field that defines the linear arithmetic.

property ***order*** : int

The order of the linear recurrence/linear recurrent sequence. The order of a sequence is defined by the degree of the minimal polynomial that produces it.

property ***taps*** : *FieldArray*

The shift register taps $T = [c_0, c_1, \dots, c_{n-2}, c_{n-1}]$. The taps of the shift register define the linear recurrence relation.

property galois.GLFSR.**field** : type[*FieldArray*]

The *FieldArray* subclass for the finite field that defines the linear arithmetic.

Examples

```
In [1]: c = galois.primitive_poly(7, 4); c
Out[1]: Poly(x^4 + x^2 + 3x + 5, GF(7))

In [2]: lfsr = galois.GLFSR(c.reverse()); lfsr
Out[2]: <Galois LFSR: f(x) = 5x^4 + 3x^3 + x^2 + 1 over GF(7)>

In [3]: lfsr.field
Out[3]: <class 'galois.GF(7)'>
```

property galois.GLFSR.**order** : int

The order of the linear recurrence/linear recurrent sequence. The order of a sequence is defined by the degree of the minimal polynomial that produces it.

property galois.GLFSR.**taps** : *FieldArray*

The shift register taps $T = [c_0, c_1, \dots, c_{n-2}, c_{n-1}]$. The taps of the shift register define the linear recurrence relation.

Examples

```
In [1]: c = galois.primitive_poly(7, 4); c
Out[1]: Poly(x^4 + x^2 + 3x + 5, GF(7))

In [2]: taps = -c.coeffs[1:][::-1]; taps
Out[2]: GF([2, 4, 6, 0], order=7)

In [3]: lfsr = galois.GLFSR.Taps(taps); lfsr
Out[3]: <Galois LFSR: f(x) = 5x^4 + 3x^3 + x^2 + 1 over GF(7)>

In [4]: lfsr.taps
Out[4]: GF([2, 4, 6, 0], order=7)
```

Polynomials

property ***characteristic_poly*** : *Poly*

The characteristic polynomial $c(x) = x^n - c_{n-1}x^{n-1} - c_{n-2}x^{n-2} - \dots - c_1 x - c_0$ that defines the linear recurrent sequence.

property ***feedback_poly*** : *Poly*

The feedback polynomial $f(x) = -c_0 x^n - c_1 x^{n-1} - \dots - c_{n-2}x^2 - c_{n-1}x + 1$ that defines the feedback arithmetic.

property galois.GLFSR.**characteristic_poly** : *Poly*

The characteristic polynomial $c(x) = x^n - c_{n-1}x^{n-1} - c_{n-2}x^{n-2} - \dots - c_1 x - c_0$ that defines the linear recurrent sequence.

Notes

The characteristic polynomial is the reciprocal of the feedback polynomial $c(x) = x^n f(x^{-1})$.

Examples

```
In [1]: c = galois.primitive_poly(7, 4); c
Out[1]: Poly(x^4 + x^2 + 3x + 5, GF(7))

In [2]: lfsr = galois.GLFSR(c.reverse()); lfsr
Out[2]: <Galois LFSR: f(x) = 5x^4 + 3x^3 + x^2 + 1 over GF(7)>

In [3]: lfsr.characteristic_poly
Out[3]: Poly(x^4 + x^2 + 3x + 5, GF(7))

In [4]: lfsr.characteristic_poly == lfsr.feedback_poly.reverse()
Out[4]: True
```

property `galois.GLFSR.`**`feedback_poly`** : *Poly*

The feedback polynomial $f(x) = -c_0 x^n - c_1 x^{n-1} - \dots - c_{n-2} x^2 - c_{n-1} x + 1$ that defines the feedback arithmetic.

Notes

The feedback polynomial is the reciprocal of the characteristic polynomial $f(x) = x^n c(x^{-1})$.

Examples

```
In [1]: c = galois.primitive_poly(7, 4); c
Out[1]: Poly(x^4 + x^2 + 3x + 5, GF(7))

In [2]: lfsr = galois.GLFSR(c.reverse()); lfsr
Out[2]: <Galois LFSR: f(x) = 5x^4 + 3x^3 + x^2 + 1 over GF(7)>

In [3]: lfsr.feedback_poly
Out[3]: Poly(5x^4 + 3x^3 + x^2 + 1, GF(7))

In [4]: lfsr.feedback_poly == lfsr.characteristic_poly.reverse()
Out[4]: True
```

State

property ***`initial_state`*** : *FieldArray*

The initial state vector $S = [S_0, S_1, \dots, S_{n-2}, S_{n-1}]$.

property ***`state`*** : *FieldArray*

The current state vector $S = [S_0, S_1, \dots, S_{n-2}, S_{n-1}]$.

property `galois.GLFSR.`**`initial_state`** : *FieldArray*

The initial state vector $S = [S_0, S_1, \dots, S_{n-2}, S_{n-1}]$.

Examples

```
In [1]: c = galois.primitive_poly(7, 4)

In [2]: lfsr = galois.GLFSR(c.reverse(), state=[1, 2, 3, 4]); lfsr
Out[2]: <Galois LFSR: f(x) = 5x^4 + 3x^3 + x^2 + 1 over GF(7)>

In [3]: lfsr.initial_state
Out[3]: GF([1, 2, 3, 4], order=7)
```

The initial state is unaffected as the Galois LFSR is stepped.

```
In [4]: lfsr.step(10)
Out[4]: GF([4, 3, 5, 0, 1, 5, 2, 6, 6, 5], order=7)

In [5]: lfsr.initial_state
Out[5]: GF([1, 2, 3, 4], order=7)
```

property `galois.GLFSR.`**`state`** : *FieldArray*

The current state vector $S = [S_0, S_1, \dots, S_{n-2}, S_{n-1}]$.

Examples

```
In [1]: c = galois.primitive_poly(7, 4)

In [2]: lfsr = galois.GLFSR(c.reverse(), state=[1, 2, 3, 4]); lfsr
Out[2]: <Galois LFSR: f(x) = 5x^4 + 3x^3 + x^2 + 1 over GF(7)>

In [3]: lfsr.state
Out[3]: GF([1, 2, 3, 4], order=7)
```

The current state is modified as the Galois LFSR is stepped.

```
In [4]: lfsr.step(10)
Out[4]: GF([4, 3, 5, 0, 1, 5, 2, 6, 6, 5], order=7)

In [5]: lfsr.state
Out[5]: GF([3, 4, 3, 1], order=7)
```

`galois.`**`berlekamp_massey`**(sequence: FieldArray, output: *Literal[minimal]* = `'minimal'`) → *Poly*

`galois.`**`berlekamp_massey`**(sequence: FieldArray, output: *Literal[fibonacci]*) → *FLFSR*

`galois.`**`berlekamp_massey`**(sequence: FieldArray, output: *Literal[galois]*) → *GLFSR*

Finds the minimal polynomial $c(x)$ that produces the linear recurrent sequence y.

This function implements the Berlekamp-Massey algorithm.

Parameters

sequence: *FieldArray*
A linear recurrent sequence y in $\mathrm{GF}(p^m)$.

output: Literal[minimal] = `'minimal'`
output: Literal[fibonacci]
output: Literal[galois]

The output object type.

- "minimal" (default): Returns the minimal polynomial that generates the linear recurrent sequence. The minimal polynomial is a characteristic polynomial $c(x)$ of minimal degree.
- "fibonacci": Returns a Fibonacci LFSR that produces y.
- "galois": Returns a Galois LFSR that produces y.

Returns

The minimal polynomial $c(x)$, a Fibonacci LFSR, or a Galois LFSR, depending on the value of `output`.

Notes

The minimal polynomial is the characteristic polynomial $c(x)$ of minimal degree that produces the linear recurrent sequence y.

$$c(x) = x^n - c_{n-1}x^{n-1} - c_{n-2}x^{n-2} - \dots - c_1 x - c_0$$

$$y_t = c_{n-1}y_{t-1} + c_{n-2}y_{t-2} + \dots + c_1 y_{t-n+2} + c_0 y_{t-n+1}$$

For a linear sequence with order n, at least $2n$ output symbols are required to determine the minimal polynomial.

References

- Gardner, D. 2019. “Applications of the Galois Model LFSR in Cryptography”. https://hdl.handle.net/2134/21932.
- Sachs, J. Linear Feedback Shift Registers for the Uninitiated, Part VI: Sing Along with the Berlekamp-Massey Algorithm. https://www.embeddedrelated.com/showarticle/1099.php
- https://crypto.stanford.edu/~mironov/cs359/massey.pdf

Examples

The sequence below is a degree-4 linear recurrent sequence over $\mathrm{GF}(7)$.

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

In [2]: y = GF([5, 5, 1, 3, 1, 4, 6, 6, 5, 5])
```

The characteristic polynomial is $c(x) = x^4 + x^2 + 3x + 5$ over $\mathrm{GF}(7)$.

```
In [3]: galois.berlekamp_massey(y)
Out[3]: Poly(x^4 + x^2 + 3x + 5, GF(7))
```

Use the Berlekamp-Massey algorithm to return equivalent Fibonacci LFSR that reproduces the sequence.

```
In [4]: lfsr = galois.berlekamp_massey(y, output="fibonacci")

In [5]: print(lfsr)
Fibonacci LFSR:
  field: GF(7)
  feedback_poly: 5x^4 + 3x^3 + x^2 + 1
  characteristic_poly: x^4 + x^2 + 3x + 5
  taps: [0 6 4 2]
  order: 4
  state: [3 1 5 5]
  initial_state: [3 1 5 5]

In [6]: z = lfsr.step(y.size); z
Out[6]: GF([5, 5, 1, 3, 1, 4, 6, 6, 5, 5], order=7)

In [7]: np.array_equal(y, z)
Out[7]: True
```

Use the Berlekamp-Massey algorithm to return equivalent Galois LFSR that reproduces the sequence.

```
In [8]: lfsr = galois.berlekamp_massey(y, output="galois")

In [9]: print(lfsr)
Galois LFSR:
  field: GF(7)
  feedback_poly: 5x^4 + 3x^3 + x^2 + 1
  characteristic_poly: x^4 + x^2 + 3x + 5
  taps: [2 4 6 0]
  order: 4
  state: [2 6 5 5]
  initial_state: [2 6 5 5]

In [10]: z = lfsr.step(y.size); z
Out[10]: GF([5, 5, 1, 3, 1, 4, 6, 6, 5, 5], order=7)

In [11]: np.array_equal(y, z)
Out[11]: True
```

3.23 Transforms

`galois.`***intt***(X: ArrayLike, ...) → *FieldArray*

Computes the Inverse Number-Theoretic Transform (INTT) of X.

`galois.`***ntt***(x: ArrayLike, size: *int* | *None* = ***None***, ...) → *FieldArray*

Computes the Number-Theoretic Transform (NTT) of x.

`galois.intt`(X: ArrayLike, size: *int* | *None* = ***None***, modulus: *int* | *None* = ***None***, scaled: *bool* = ***True***) → *FieldArray*

Computes the Inverse Number-Theoretic Transform (INTT) of X.

Parameters

X: ***ArrayLike***
The input sequence of integers X.

size: int | None = `None`
The size N of the INTT transform, must be at least the length of X. The default is `None` which corresponds to `len(X)`. If `size` is larger than the length of X, X is zero-padded.

modulus: int | None = `None`
The prime modulus p that defines the field $\mathrm{GF}(p)$. The prime modulus must satisfy $p > \max(X)$ and $p = mN + 1$ (i.e., the size of the transform N must divide $p - 1$). The default is `None` which corresponds to the smallest p that satisfies the criteria. However, if x is a $\mathrm{GF}(p)$ array, then `None` corresponds to p from the specified field.

scaled: bool = `True`
Indicates to scale the INTT output by N. The default is `True`. If `True`, $x = \mathrm{INTT}(\mathrm{NTT}(x))$. If `False`, $Nx = \mathrm{INTT}(\mathrm{NTT}(x))$.

Returns
The INTT x of the input X, with length N. The output is a $\mathrm{GF}(p)$ array. It can be viewed as a normal NumPy array with `.view(np.ndarray)` or converted to a Python list with `.tolist()`.

See also

ntt

Notes

The Number-Theoretic Transform (NTT) is a specialized Discrete Fourier Transform (DFT) over a finite field $\mathrm{GF}(p)$ instead of over $\mathbb{C}$. The DFT uses the primitive N-th root of unity $\omega_N = e^{-i2\pi/N}$, but the NTT uses a primitive N-th root of unity in $\mathrm{GF}(p)$. These roots are such that $\omega_N^N = 1$ and $\omega_N^k \neq 1$ for $0 < k < N$.

In $\mathrm{GF}(p)$, where p is prime, a primitive N-th root of unity exists if N divides $p - 1$. If that is true, then $p = mN + 1$ for some integer m. This function finds ω_N by first finding a primitive $p - 1$-th root of unity ω_{p-1} in $\mathrm{GF}(p)$ using *primitive_root()*. From there ω_N is found from $\omega_N = \omega_{p-1}^m$.

The j-th value of the scaled N-point INTT $x = \mathrm{INTT}(X)$ is

$$x_j = \frac{1}{N} \sum_{k=0}^{N-1} X_k \omega_N^{-kj},$$

with all arithmetic performed in $\mathrm{GF}(p)$. The scaled INTT has the property that $x = \mathrm{INTT}(\mathrm{NTT}(x))$.

A radix-2 Cooley-Tukey FFT algorithm is implemented, which achieves $O(N \log(N))$.

References

- https://cgyurgyik.github.io/posts/2021/04/brief-introduction-to-ntt/
- https://www.nayuki.io/page/number-theoretic-transform-integer-dft
- https://www.geeksforgeeks.org/python-number-theoretic-transformation/

Examples

The default modulus is the smallest p such that $p > \max(X)$ and $p = mN + 1$. With the input $X = [0, 4, 3, 2]$ and $N = 4$, the default modulus is $p = 5$.

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

However, other moduli satisfy $p > \max(X)$ and $p = mN + 1$. For instance, $p = 13$ and $p = 17$. Notice the INTT outputs are different with different moduli. So it is important to perform forward and reverse NTTs with the same modulus.

```
In [2]: galois.intt([0, 4, 3, 2], modulus=13)
Out[2]: GF([12,  5,  9,  0], order=13)

In [3]: galois.intt([0, 4, 3, 2], modulus=17)
Out[3]: GF([15, 14, 12, 10], order=17)
```

Instead of explicitly specifying the prime modulus, a $\mathrm{GF}(p)$ array may be explicitly passed in and the modulus is taken as p.

```
In [4]: GF = galois.GF(13)

In [5]: X = GF([10, 8, 11, 1]); X
Out[5]: GF([10,  8, 11,  1], order=13)

In [6]: x = galois.intt(X); x
Out[6]: GF([1, 2, 3, 4], order=13)

In [7]: galois.ntt(x)
Out[7]: GF([10,  8, 11,  1], order=13)
```

The forward NTT and scaled INTT are the identity transform, i.e. $x = \mathrm{INTT}(\mathrm{NTT}(x))$.

```
In [8]: GF = galois.GF(13)

In [9]: x = GF([1, 2, 3, 4]); x
Out[9]: GF([1, 2, 3, 4], order=13)

In [10]: galois.intt(galois.ntt(x))
Out[10]: GF([1, 2, 3, 4], order=13)
```

This is also true in the reverse order, i.e. $x = \mathrm{NTT}(\mathrm{INTT}(x))$.

```
In [11]: galois.ntt(galois.intt(x))
Out[11]: GF([1, 2, 3, 4], order=13)
```

The `numpy.fft.ifft()` function may also be used to compute the inverse NTT over $\mathrm{GF}(p)$.

```
In [12]: X = np.fft.fft(x); X
Out[12]: GF([10,  8, 11,  1], order=13)

In [13]: np.fft.ifft(X)
Out[13]: GF([1, 2, 3, 4], order=13)
```

`galois.ntt(x: ArrayLike, size: int | None = None, modulus: int | None = None) → FieldArray`

Computes the Number-Theoretic Transform (NTT) of x.

Parameters

x: *ArrayLike*
The input sequence of integers x.

size: int | None = None
The size N of the NTT transform, must be at least the length of x. The default is `None` which corresponds to `len(x)`. If `size` is larger than the length of x, x is zero-padded.

modulus: int | None = None
The prime modulus p that defines the field $\mathrm{GF}(p)$. The prime modulus must satisfy $p > \max(x)$ and $p = mN + 1$ (i.e., the size of the transform N must divide $p - 1$). The default is `None` which corresponds to the smallest p that satisfies the criteria. However, if x is a $\mathrm{GF}(p)$ array, then `None` corresponds to p from the specified field.

Returns
The NTT X of the input x, with length N. The output is a $\mathrm{GF}(p)$ array. It can be viewed as a normal NumPy array with `.view(np.ndarray)` or converted to a Python list with `.tolist()`.

See also

intt

Notes

The Number-Theoretic Transform (NTT) is a specialized Discrete Fourier Transform (DFT) over a finite field $\mathrm{GF}(p)$ instead of over $\mathbb{C}$. The DFT uses the primitive N-th root of unity $\omega_N = e^{-i2\pi/N}$, but the NTT uses a primitive N-th root of unity in $\mathrm{GF}(p)$. These roots are such that $\omega_N^N = 1$ and $\omega_N^k \neq 1$ for $0 < k < N$.

In $\mathrm{GF}(p)$, where p is prime, a primitive N-th root of unity exists if N divides $p - 1$. If that is true, then $p = mN + 1$ for some integer m. This function finds ω_N by first finding a primitive $p - 1$-th root of unity ω_{p-1} in $\mathrm{GF}(p)$ using *primitive_root()*. From there ω_N is found from $\omega_N = \omega_{p-1}^m$.

The k-th value of the N-point NTT $X = \mathrm{NTT}(x)$ is

$$X_k = \sum_{j=0}^{N-1} x_j \omega_N^{jk},$$

with all arithmetic performed in $\mathrm{GF}(p)$.

A radix-2 Cooley-Tukey FFT algorithm is implemented, which achieves $O(N \log(N))$.

References

- https://cgyurgyik.github.io/posts/2021/04/brief-introduction-to-ntt/
- https://www.nayuki.io/page/number-theoretic-transform-integer-dft
- https://www.geeksforgeeks.org/python-number-theoretic-transformation/

Examples

The default modulus is the smallest p such that $p > \max(x)$ and $p = mN + 1$. With the input $x = [1, 2, 3, 4]$ and $N = 4$, the default modulus is $p = 5$.

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

However, other moduli satisfy $p > \max(x)$ and $p = mN + 1$. For instance, $p = 13$ and $p = 17$. Notice the NTT outputs are different with different moduli. So it is important to perform forward and reverse NTTs with the same modulus.

```
In [2]: galois.ntt([1, 2, 3, 4], modulus=13)
Out[2]: GF([10,  8, 11,  1], order=13)

In [3]: galois.ntt([1, 2, 3, 4], modulus=17)
Out[3]: GF([10,  6, 15,  7], order=17)
```

Instead of explicitly specifying the prime modulus, a $\mathrm{GF}(p)$ array may be explicitly passed in and the modulus is taken as p.

```
In [4]: GF = galois.GF(13)

In [5]: galois.ntt(GF([1, 2, 3, 4]))
Out[5]: GF([10,  8, 11,  1], order=13)
```

The `size` keyword argument allows convenient zero-padding of the input (to a power of two, for example).

```
In [6]: galois.ntt([1, 2, 3, 4, 5, 6], size=8)
Out[6]: GF([ 4,  8,  4,  1, 14, 11,  2, 15], order=17)

In [7]: galois.ntt([1, 2, 3, 4, 5, 6, 0, 0])
Out[7]: GF([ 4,  8,  4,  1, 14, 11,  2, 15], order=17)
```

The `numpy.fft.fft()` function may also be used to compute the NTT over $\mathrm{GF}(p)$.

```
In [8]: GF = galois.GF(17)

In [9]: x = GF([1, 2, 3, 4, 5, 6])

In [10]: np.fft.fft(x, n=8)
Out[10]: GF([ 4,  8,  4,  1, 14, 11,  2, 15], order=17)
```

3.24 Number theory

3.24.1 Divisibility

`galois.are_coprime(*values: int) → bool`

`galois.are_coprime(*values: Poly) → bool`

Determines if the arguments are pairwise coprime.

`galois.egcd(a: int, b: int) → tuple[int, int, int]`

`galois.egcd(a: Poly, b: Poly) → tuple[Poly, Poly, Poly]`

Finds the multiplicands of a and b such that $as + bt = \gcd(a, b)$.

`galois.euler_phi(n: int) → int`

Counts the positive integers (totatives) in $[1, n)$ that are coprime to n.

`galois.gcd(a: int, b: int) → int`

`galois.gcd(a: Poly, b: Poly) → Poly`

Finds the greatest common divisor of a and b.

`galois.lcm(*values: int) → int`

`galois.lcm(*values: Poly) → Poly`

Computes the least common multiple of the arguments.

`galois.prod(*values: int) → int`

`galois.prod(*values: Poly) → Poly`

Computes the product of the arguments.

`galois.totatives(n: int) → list[int]`

Returns the positive integers (totatives) in $[1, n)$ that are coprime to n.

`galois.are_coprime(*values: int) → bool`

`galois.are_coprime(*values: Poly) → bool`

Determines if the arguments are pairwise coprime.

Parameters

***values: int**
values: *Poly
Each argument must be an integer or polynomial.

Returns

`True` if the arguments are pairwise coprime.

See also

lcm, *prod*

Notes

A set of integers or polynomials are pairwise coprime if their LCM is equal to their product.

Examples

Integers

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
```

Polynomials

Generate irreducible polynomials over $\mathrm{GF}(7)$.

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

In [4]: f1 = galois.irreducible_poly(7, 1); f1
Out[4]: Poly(x, GF(7))

In [5]: f2 = galois.irreducible_poly(7, 2); f2
Out[5]: Poly(x^2 + 1, GF(7))

In [6]: f3 = galois.irreducible_poly(7, 3); f3
Out[6]: Poly(x^3 + 2, GF(7))
```

Determine if combinations of the irreducible polynomials are pairwise coprime.

```
In [7]: galois.are_coprime(f1, f2, f3)
Out[7]: True

In [8]: galois.are_coprime(f1 * f2, f2, f3)
Out[8]: False
```

`galois.egcd(a: int, b: int)` → tuple[int, int, int]

`galois.egcd(a: Poly, b: Poly)` → tuple[*Poly*, *Poly*, *Poly*]

Finds the multiplicands of a and b such that $as + bt = \gcd(a, b)$.

Parameters

a: int
a: *Poly*
The first integer or polynomial argument.

b: int
b: *Poly*
The second integer or polynomial argument.

Returns

- Greatest common divisor of a and b.
- The multiplicand s of a.
- The multiplicand t of b.

See also

gcd, *lcm*, *prod*

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
- Moon, T. "Error Correction Coding", Section 5.2.2: The Euclidean Algorithm and Euclidean Domains, p. 181

Examples

Integers

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
```

Polynomials

Generate irreducible polynomials over $\mathrm{GF}(7)$.

```
In [5]: GF = galois.GF(7)

In [6]: f1 = galois.irreducible_poly(7, 1); f1
Out[6]: Poly(x, GF(7))

In [7]: f2 = galois.irreducible_poly(7, 2); f2
Out[7]: Poly(x^2 + 1, GF(7))
```

(continues on next page)

(continued from previous page)

```
In [8]: f3 = galois.irreducible_poly(7, 3); f3
Out[8]: Poly(x^3 + 2, GF(7))
```

Compute the extended GCD of $f_1(x)^2 f_2(x)$ and $f_1(x) f_3(x)$.

```
In [9]: a = f1**2 * f2

In [10]: b = f1 * f3

In [11]: gcd, s, t = galois.egcd(a, b)

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.**euler_phi**(n: *int*) → int

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.

See also

carmichael_lambda, *totatives*, *is_cyclic*

Notes

This function implements the Euler totient function

$$phi(n) = n prod_{p|n} bigg(1 - frac{1}{p}bigg) = prod_{i=1}^{k} p_i^{e_i-1} big(p_i - 1big)$$

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

Compute $\phi(20)$.

```
In [1]: n = 20

In [2]: phi = galois.euler_phi(n); phi
Out[2]: 8
```

Find the totatives that are coprime with $n = 20$. The number of totatives of n is $\phi(n)$.

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

In [4]: len(x) == phi
Out[4]: True
```

For prime n, $\phi(n) = n - 1$.

```
In [5]: n = 13

In [6]: galois.euler_phi(n)
Out[6]: 12
```

galois.**gcd**(a: *int*, b: *int*) → int

galois.**gcd**(a: Poly, b: Poly) → *Poly*

Finds the greatest common divisor of a and b.

Parameters

a: int

a: *Poly*

The first integer or polynomial argument.

b: int

b: *Poly*

The second integer or polynomial argument.

Returns

Greatest common divisor of a and b.

See also

egcd, *lcm*, *prod*

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

Integers

Compute the GCD of two integers.

```
In [1]: galois.gcd(12, 16)
Out[1]: 4
```

Polynomials

Generate irreducible polynomials over $\mathrm{GF}(7)$.

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

In [3]: f1 = galois.irreducible_poly(7, 1); f1
Out[3]: Poly(x, GF(7))

In [4]: f2 = galois.irreducible_poly(7, 2); f2
Out[4]: Poly(x^2 + 1, GF(7))

In [5]: f3 = galois.irreducible_poly(7, 3); f3
Out[5]: Poly(x^3 + 2, GF(7))
```

Compute the GCD of $f_1(x)^2 f_2(x)$ and $f_1(x) f_3(x)$, which is $f_1(x)$.

```
In [6]: galois.gcd(f1**2 * f2, f1 * f3)
Out[6]: Poly(x, GF(7))
```

`galois.lcm(*values: int)` → int

`galois.lcm(*values: Poly)` → *Poly*

Computes the least common multiple of the arguments.

Parameters

***values: int**
values: *Poly
Each argument must be an integer or polynomial.

Returns
The least common multiple of the arguments.

See also

gcd, *egcd*, *prod*

Examples

Integers

Compute the LCM of three integers.

```
In [1]: galois.lcm(2, 4, 14)
Out[1]: 28
```

Polynomials

Generate irreducible polynomials over GF(7).

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

In [3]: f1 = galois.irreducible_poly(7, 1); f1
Out[3]: Poly(x, GF(7))

In [4]: f2 = galois.irreducible_poly(7, 2); f2
Out[4]: Poly(x^2 + 1, GF(7))

In [5]: f3 = galois.irreducible_poly(7, 3); f3
Out[5]: Poly(x^3 + 2, GF(7))
```

Compute the LCM of three polynomials $f_1(x)^2 f_2(x)$, $f_1(x) f_3(x)$, and $f_2(x) f_3(x)$, which is $f_1(x)^2 f_2(x) f_3(x)$.

```
In [6]: galois.lcm(f1**2 * f2, f1 * f3, f2 * f3)
Out[6]: Poly(x^7 + x^5 + 2x^4 + 2x^2, GF(7))

In [7]: f1**2 * f2 * f3
Out[7]: Poly(x^7 + x^5 + 2x^4 + 2x^2, GF(7))
```

`galois.prod`(*values: *int*) → int

`galois.prod`(*values: Poly) → *Poly*

Computes the product of the arguments.

Parameters

***values: int**

values: *Poly

Each argument must be an integer or polynomial.

Returns

The product of the arguments.

See also

gcd, *egcd*, *lcm*

Examples

Integers

Compute the product of three integers.

```
In [1]: galois.prod(2, 4, 14)
Out[1]: 112
```

Polynomials

Generate random polynomials over $\mathrm{GF}(7)$.

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

In [3]: f1 = galois.Poly.Random(2, field=GF); f1
Out[3]: Poly(4x^2 + 4x + 5, GF(7))

In [4]: f2 = galois.Poly.Random(3, field=GF); f2
Out[4]: Poly(x^3 + 4x^2, GF(7))

In [5]: f3 = galois.Poly.Random(4, field=GF); f3
Out[5]: Poly(4x^4 + 2x^2 + 5x + 2, GF(7))
```

Compute the product of three polynomials.

```
In [6]: galois.prod(f1, f2, f3)
Out[6]: Poly(2x^9 + 3x^8 + x^7 + 3x^5 + 3x^4 + 2x^3 + 5x^2, GF(7))

In [7]: f1 * f2 * f3
Out[7]: Poly(2x^9 + 3x^8 + x^7 + 3x^5 + 3x^4 + 2x^3 + 5x^2, GF(7))
```

`galois.totatives(n: int) → list[int]`

Returns the positive integers (totatives) in $[1, n)$ that are coprime to n.

Parameters

n: int
A positive integer.

Returns
The totatives of n.

See also

euler_phi, *carmichael_lambda*, *is_cyclic*

Notes

The totatives of n form the multiplicative group $(\mathbb{Z}/n\mathbb{Z})^\times$.

References

- Section 2.4.3 from https://cacr.uwaterloo.ca/hac/about/chap2.pdf
- https://oeis.org/A000010

Examples

Find the totatives that are coprime with $n = 20$.

```
In [1]: n = 20

In [2]: x = galois.totatives(n); x
Out[2]: [1, 3, 7, 9, 11, 13, 17, 19]
```

The number of totatives of n is $\phi(n)$.

```
In [3]: phi = galois.euler_phi(n); phi
Out[3]: 8

In [4]: len(x) == phi
Out[4]: True
```

3.24.2 Congruences

`galois.carmichael_lambda(n: int) → int`

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.

`galois.crt(remainders: Sequence[int], moduli: Sequence[int]) → int`

`galois.crt(remainders: Sequence[Poly], moduli) → Poly`

Solves the simultaneous system of congruences for x.

`galois.jacobi_symbol(a: int, n: int) → int`

Computes the Jacobi symbol $(\frac{a}{n})$.

`galois.kronecker_symbol(a: int, n: int) → int`

Computes the Kronecker symbol $(\frac{a}{n})$. The Kronecker symbol extends the Jacobi symbol for all n.

`galois.legendre_symbol(a: int, p: int) → int`

Computes the Legendre symbol $(\frac{a}{p})$.

`galois.is_cyclic(n: int) → bool`

Determines whether the multiplicative group $(\mathbb{Z}/n\mathbb{Z})^\times$ is cyclic.

`galois.carmichael_lambda(n: int) → int`

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.

Parameters

n: int
A positive integer.

Returns
The smallest positive integer m such that $a^m \equiv 1 \pmod{n}$ for every a in $[1, n)$ that is coprime to n.

See also

euler_phi, *totatives*, *is_cyclic*

Notes

This function implements the Carmichael function $\lambda(n)$.

References

- https://oeis.org/A002322

Examples

The Carmichael $\lambda(n)$ function and Euler $\phi(n)$ 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

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)
```

(continues on next page)

(continued from previous page)

```
Power 6: [1, 1, 1, 1, 1, 1] (mod 9)

In [8]: galois.is_cyclic(n)
Out[8]: True
```

When $\phi(n) \ne \lambda(n)$, the multiplicative group $(\mathbb{Z}/n\mathbb{Z})^\times$ is not cyclic. See `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.crt`(remainders: *Sequence[int]*, moduli: *Sequence[int]*) → int

`galois.crt`(remainders: *Sequence[*Poly*]*, moduli: *Sequence[*Poly*]*) → *Poly*

Solves the simultaneous system of congruences for x.

Parameters

remainders: Sequence[int]
remainders: Sequence[*Poly*]
The integer or polynomial remainders a_i.

moduli: Sequence[int]
moduli: Sequence[*Poly*]
The integer or polynomial moduli m_i.

Returns

The simultaneous solution x to the system of congruences.

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

Integers

Define a system of integer congruences.

```
In [1]: a = [0, 3, 4]

In [2]: m = [3, 4, 5]
```

Solve the system of congruences.

```
In [3]: x = galois.crt(a, m); x
Out[3]: 39
```

Show that the solution satisfies each congruence.

```
In [4]: for i in range(len(a)):
   ...:     ai = x % m[i]
   ...:     print(ai, ai == a[i])
   ...:
0 True
3 True
4 True
```

Polynomials

Define a system of polynomial congruences over $\mathrm{GF}(7)$.

```
In [5]: GF = galois.GF(7)

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

In [7]: m3 = galois.Poly.Random(3, field=GF)
```

(continues on next page)

(continued from previous page)

```
In [8]: m4 = galois.Poly.Random(4, field=GF)

In [9]: m5 = galois.Poly.Random(5, field=GF)

In [10]: m = [m3, m4, m5]; m
Out[10]:
[Poly(x^3 + 4x^2 + 4x + 6, GF(7)),
 Poly(5x^4 + x^3 + 3, GF(7)),
 Poly(6x^5 + 5x^4 + x^3 + 6x^2 + 2x + 3, GF(7))]

In [11]: a = [x_truth % m3, x_truth % m4, x_truth % m5]; a
Out[11]:
[Poly(2x^2 + 5x + 2, GF(7)),
 Poly(x^2 + 2x, GF(7)),
 Poly(x^4 + 2x^3 + x^2 + 3x, GF(7))]
```

Solve the system of congruences.

```
In [12]: x = galois.crt(a, m); x
Out[12]: Poly(5x^6 + 2x^5 + x^4 + x^3 + 4x^2 + 4x + 3, GF(7))
```

Show that the solution satisfies each congruence.

```
In [13]: for i in range(len(a)):
   ....:     ai = x % m[i]
   ....:     print(ai, ai == a[i])
   ....:
2x^2 + 5x + 2 True
x^2 + 2x True
x^4 + 2x^3 + x^2 + 3x True
```

galois.**jacobi_symbol**(a: *int*, n: *int*) → int

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 $\{0, 1, -1\}$.

See also

legendre_symbol, *kronecker_symbol*

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**(a: *int*, n: *int*) → int

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 $\{0, -1, 1\}$.

See also

legendre_symbol, *jacobi_symbol*

References

- Algorithm 2.149 from https://cacr.uwaterloo.ca/hac/about/chap2.pdf

galois.**legendre_symbol**(a: *int*, p: *int*) → int

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 $\{0, 1, -1\}$.

See also

jacobi_symbol, *kronecker_symbol*

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.is_cyclic(n: int) → bool`

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.

See also

euler_phi, *carmichael_lambda*, *totatives*

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 $\{1, g, g^2, \dots, g^{\phi(n)-1}\}$, 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

n = 14

The elements of $(\mathbb{Z}/14\mathbb{Z})^\times = \{1, 3, 5, 9, 11, 13\}$ are the totatives of 14.

```
In [1]: n = 14

In [2]: Znx = galois.totatives(n); Znx
Out[2]: [1, 3, 5, 9, 11, 13]
```

The Euler totient $\phi(n)$ function counts the totatives of n, which is equivalent to the order of $(\mathbb{Z}/n\mathbb{Z})^\times$.

```
In [3]: phi = galois.euler_phi(n); phi
Out[3]: 6

In [4]: len(Znx) == phi
Out[4]: True
```

Since 14 is of the form $2p^k$, the multiplicative group $(\mathbb{Z}/14\mathbb{Z})^\times$ is cyclic, meaning there exists at least one element that generates the group by its powers.

```
In [5]: galois.is_cyclic(n)
Out[5]: True
```

Find the smallest primitive root modulo 14. Observe that the powers of g uniquely represent each element in $(\mathbb{Z}/14\mathbb{Z})^\times$.

```
In [6]: g = galois.primitive_root(n); g
Out[6]: 3

In [7]: [pow(g, i, n) for i in range(0, phi)]
Out[7]: [1, 3, 9, 13, 11, 5]
```

Find the largest primitive root modulo 14. Observe that the powers of g also uniquely represent each element in $(\mathbb{Z}/14\mathbb{Z})^{\times}$, although in a different order.

```
In [8]: g = galois.primitive_root(n, method="max"); g
Out[8]: 5

In [9]: [pow(g, i, n) for i in range(0, phi)]
Out[9]: [1, 5, 11, 13, 9, 3]
```

n = 15

A non-cyclic group is $(\mathbb{Z}/15\mathbb{Z})^{\times} = \{1, 2, 4, 7, 8, 11, 13, 14\}$.

```
In [10]: n = 15

In [11]: Znx = galois.totatives(n); Znx
Out[11]: [1, 2, 4, 7, 8, 11, 13, 14]

In [12]: phi = galois.euler_phi(n); phi
Out[12]: 8
```

Since 15 is not of the form 2, 4, p^k, or $2p^k$, the multiplicative group $(\mathbb{Z}/15\mathbb{Z})^{\times}$ is not cyclic, meaning no elements exist whose powers generate the group.

```
In [13]: galois.is_cyclic(n)
Out[13]: False
```

Below, every element is tested to see if it spans the group.

```
In [14]: for a in Znx:
   ....:     span = set([pow(a, i, n) for i in range(0, phi)])
   ....:     primitive_root = span == set(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
```

The Carmichael $\lambda(n)$ function finds the maximum multiplicative order of any element, which is 4 and not 8.

```
In [15]: galois.carmichael_lambda(n)
Out[15]: 4
```

Observe that no primitive roots modulo 15 exist and a RuntimeError is raised.

```
In [16]: galois.primitive_root(n)
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
File ~/checkouts/readthedocs.org/user_builds/galois/envs/latest/lib/python3.8/site-
↪packages/galois/_modular.py:559, in primitive_root(n, start, stop, method)
    558 if method == "min":
--> 559     root = next(primitive_roots(n, start, stop=stop))
    560 elif method == "max":

StopIteration:

The above exception was the direct cause of the following exception:

RuntimeError                              Traceback (most recent call last)
Cell In[16], line 1
----> 1 galois.primitive_root(n)

File ~/checkouts/readthedocs.org/user_builds/galois/envs/latest/lib/python3.8/site-
↪packages/galois/_modular.py:566, in primitive_root(n, start, stop, method)
    564     return root
    565 except StopIteration as e:
--> 566     raise RuntimeError(f"No primitive roots modulo {n} exist in the range [
↪{start}, {stop}).") from e

RuntimeError: No primitive roots modulo 15 exist in the range [1, 15).
```

Prime fields

For prime n, a primitive root modulo n is also a primitive element of the Galois field $\mathrm{GF}(n)$.

```
In [17]: n = 31

In [18]: galois.is_cyclic(n)
Out[18]: True
```

A primitive element is a generator of the multiplicative group $\mathrm{GF}(p)^\times = \{1, 2, \dots, p-1\} = \{1, g, g^2, \dots, g^{\phi(n)-1}\}$.

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

In [20]: galois.primitive_root(n)
Out[20]: 3

In [21]: GF.primitive_element
Out[21]: GF(3, order=31)
```

The number of primitive roots/elements is $\phi(\phi(n))$.

```
In [22]: list(galois.primitive_roots(n))
Out[22]: [3, 11, 12, 13, 17, 21, 22, 24]

In [23]: GF.primitive_elements
Out[23]: GF([ 3, 11, 12, 13, 17, 21, 22, 24], order=31)

In [24]: galois.euler_phi(galois.euler_phi(n))
Out[24]: 8
```

3.24.3 Primitive roots

`galois.primitive_root(n: int, start: int = 1, ...) → int`

Finds a primitive root modulo n in the range $[\text{start}, \text{stop})$.

`galois.primitive_roots(n: int, start: int = 1, ...) → Iterator[int]`

Iterates through all primitive roots modulo n in the range $[\text{start}, \text{stop})$.

`galois.is_primitive_root(g: int, n: int) → bool`

Determines if g is a primitive root modulo n.

`galois.primitive_root(n: int, start: int = 1, stop: int | None = None, method: 'min' | 'max' | 'random' = 'min') → int`

Finds a primitive root modulo n in the range $[\text{start}, \text{stop})$.

Parameters

n: int
A positive integer.

start: int = 1
Starting value (inclusive) in the search for a primitive root.

stop: int | None = None
Stopping value (exclusive) in the search for a primitive root. The default is `None` which corresponds to n.

method: 'min' | 'max' | 'random' = 'min'
The search method for finding the primitive root.

Returns
A primitive root modulo n in the specified range.

Raises
`RuntimeError` – If no primitive roots exist in the specified range.

See also

primitive_roots, is_primitive_root, is_cyclic, totatives, euler_phi, carmichael_lambda

Notes

The integer g is a primitive root modulo n if the totatives of n can be generated by the powers of g. The totatives of n are the positive integers in $[1, n)$ that are coprime with n.

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 $(\mathbb{Z}/n\mathbb{Z})^\times = \{1, g, g^2, \dots, g^{\phi(n)-1}\}$, where $\phi(n)$ is the 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

- Shoup, V. 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
- Hua, L.K. 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

n = 14

The elements of $(\mathbb{Z}/14\mathbb{Z})^\times = \{1, 3, 5, 9, 11, 13\}$ are the totatives of 14.

```
In [1]: n = 14

In [2]: Znx = galois.totatives(n); Znx
Out[2]: [1, 3, 5, 9, 11, 13]
```

The Euler totient $\phi(n)$ function counts the totatives of n, which is equivalent to the order of $(\mathbb{Z}/n\mathbb{Z})^\times$.

```
In [3]: phi = galois.euler_phi(n); phi
Out[3]: 6

In [4]: len(Znx) == phi
Out[4]: True
```

Since 14 is of the form $2p^k$, the multiplicative group $(\mathbb{Z}/14\mathbb{Z})^\times$ is cyclic, meaning there exists at least one element that generates the group by its powers.

```
In [5]: galois.is_cyclic(n)
Out[5]: True
```

Find the smallest primitive root modulo 14. Observe that the powers of g uniquely represent each element in $(\mathbb{Z}/14\mathbb{Z})^\times$.

```
In [6]: g = galois.primitive_root(n); g
Out[6]: 3

In [7]: [pow(g, i, n) for i in range(0, phi)]
Out[7]: [1, 3, 9, 13, 11, 5]
```

Find the largest primitive root modulo 14. Observe that the powers of g also uniquely represent each element in $(\mathbb{Z}/14\mathbb{Z})^\times$, although in a different order.

```
In [8]: g = galois.primitive_root(n, method="max"); g
Out[8]: 5

In [9]: [pow(g, i, n) for i in range(0, phi)]
Out[9]: [1, 5, 11, 13, 9, 3]
```

n = 15

A non-cyclic group is $(\mathbb{Z}/15\mathbb{Z})^\times = \{1, 2, 4, 7, 8, 11, 13, 14\}$.

```
In [10]: n = 15

In [11]: Znx = galois.totatives(n); Znx
Out[11]: [1, 2, 4, 7, 8, 11, 13, 14]

In [12]: phi = galois.euler_phi(n); phi
Out[12]: 8
```

Since 15 is not of the form 2, 4, p^k, or $2p^k$, the multiplicative group $(\mathbb{Z}/15\mathbb{Z})^\times$ is not cyclic, meaning no elements exist whose powers generate the group.

```
In [13]: galois.is_cyclic(n)
Out[13]: False
```

Below, every element is tested to see if it spans the group.

```
In [14]: for a in Znx:
   ....:     span = set([pow(a, i, n) for i in range(0, phi)])
   ....:     primitive_root = span == set(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
```

The Carmichael $\lambda(n)$ function finds the maximum multiplicative order of any element, which is 4 and not 8.

```
In [15]: galois.carmichael_lambda(n)
Out[15]: 4
```

Observe that no primitive roots modulo 15 exist and a `RuntimeError` is raised.

```
In [16]: galois.primitive_root(n)
---------------------------------------------------------------------------
```

(continues on next page)

(continued from previous page)

```
StopIteration                             Traceback (most recent call last)
File ~/checkouts/readthedocs.org/user_builds/galois/envs/latest/lib/python3.8/site-
↪packages/galois/_modular.py:559, in primitive_root(n, start, stop, method)
    558 if method == "min":
--> 559     root = next(primitive_roots(n, start, stop=stop))
    560 elif method == "max":

StopIteration:

The above exception was the direct cause of the following exception:

RuntimeError                              Traceback (most recent call last)
Cell In[16], line 1
----> 1 galois.primitive_root(n)

File ~/checkouts/readthedocs.org/user_builds/galois/envs/latest/lib/python3.8/site-
↪packages/galois/_modular.py:566, in primitive_root(n, start, stop, method)
    564     return root
    565 except StopIteration as e:
--> 566     raise RuntimeError(f"No primitive roots modulo {n} exist in the range [
↪{start}, {stop}).") from e

RuntimeError: No primitive roots modulo 15 exist in the range [1, 15).
```

Very large n

The algorithm is also efficient for very large n.

```
In [17]: n = 1000000000000000035000061

In [18]: phi = galois.euler_phi(n); phi
Out[18]: 1000000000000000035000060
```

Find the smallest, the largest, and a random primitive root modulo n.

```
In [19]: galois.primitive_root(n)
Out[19]: 7

In [20]: galois.primitive_root(n, method="max")
Out[20]: 1000000000000000035000054

In [21]: galois.primitive_root(n, method="random")
Out[21]: 952035921357656539643084
```

galois.**primitive_roots**(n: *int*, start: *int* = 1, stop: *int* | *None* = **None**, reverse: *bool* = **False**) → Iterator[int]

Iterates through all primitive roots modulo n in the range $[\text{start}, \text{stop})$.

Parameters

n: int
A positive integer.

start: int = 1
Starting value (inclusive) in the search for a primitive root. The default is 1.

stop: int | None = None
Stopping value (exclusive) in the search for a primitive root. The default is `None` which corresponds to n.

reverse: bool = False
Indicates to return the primitive roots from largest to smallest. The default is `False`.

Returns
An iterator over the primitive roots modulo n in the specified range.

See also

primitive_root, *is_primitive_root*, *is_cyclic*, *totatives*, *euler_phi*, *carmichael_lambda*

Notes

The integer g is a primitive root modulo n if the totatives of n can be generated by the powers of g. The totatives of n are the positive integers in $[1, n)$ that are coprime with n.

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 $(\mathbb{Z}/n\mathbb{Z})^\times = \{1, g, g^2, \dots, g^{\phi(n)-1}\}$, where $\phi(n)$ is the 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

- Shoup, V. 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
- Hua, L.K. 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

All primitive roots modulo 31. You may also use `tuple()` on the returned generator.

```
In [1]: list(galois.primitive_roots(31))
Out[1]: [3, 11, 12, 13, 17, 21, 22, 24]
```

There are no primitive roots modulo 30.

```
In [2]: list(galois.primitive_roots(30))
Out[2]: []
```

Show the each primitive root modulo 22 generates the multiplicative group $(\mathbb{Z}/22\mathbb{Z})^\times$.

```
In [3]: n = 22

In [4]: Znx = galois.totatives(n); Znx
```

(continues on next page)

(continued from previous page)

```
Out[4]: [1, 3, 5, 7, 9, 13, 15, 17, 19, 21]

In [5]: phi = galois.euler_phi(n); phi
Out[5]: 10

In [6]: for root in galois.primitive_roots(22):
   ...:     span = set(pow(root, i, n) for i in range(0, phi))
   ...:     print(f"Element: {root:>2}, Span: {span}")
   ...:
Element:  7, Span: {1, 3, 5, 7, 9, 13, 15, 17, 19, 21}
Element: 13, Span: {1, 3, 5, 7, 9, 13, 15, 17, 19, 21}
Element: 17, Span: {1, 3, 5, 7, 9, 13, 15, 17, 19, 21}
Element: 19, Span: {1, 3, 5, 7, 9, 13, 15, 17, 19, 21}
```

Find the three largest primitive roots modulo 31 in reversed order.

```
In [7]: generator = galois.primitive_roots(31, reverse=True); generator
Out[7]: <generator object primitive_roots at 0x7fb35c656ba0>

In [8]: [next(generator) for _ in range(3)]
Out[8]: [24, 22, 21]
```

Loop over all the primitive roots in reversed order, only finding them as needed. The search cost for the roots that would have been found after the `break` condition is never incurred.

```
In [9]: for root in galois.primitive_roots(31, reverse=True):
   ...:     print(root)
   ...:     if root % 7 == 0:  # Arbitrary early exit condition
   ...:         break
   ...:
24
22
21
```

`galois.is_primitive_root(g: int, n: int) → bool`

Determines if g is a primitive root modulo n.

Parameters

g: int
A positive integer.

n: int
positive integer.

Returns
`True` if g is a primitive root modulo n.

See also

primitive_root, *primitive_roots*, *is_cyclic*, *euler_phi*

Notes

The integer g is a primitive root modulo n 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,

$$(\mathbb{Z}/n\mathbb{Z})^{\times} = \{1, g, g^2, \dots, g^{\phi(n)-1}\}$$

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]: list(galois.primitive_roots(7))
Out[1]: [3, 5]

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

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

3.24.4 Integer arithmetic

`galois.ilog(n: int, b: int) → int`

Computes $x = \lfloor \log_b(n) \rfloor$ such that $b^x \le n < b^{x+1}$.

`galois.iroot(n: int, k: int) → int`

Computes $x = \lfloor n^{\frac{1}{k}} \rfloor$ such that $x^k \le n < (x+1)^k$.

`galois.isqrt(n: int) → int`

Computes $x = \lfloor \sqrt{n} \rfloor$ such that $x^2 \le n < (x+1)^2$.

`galois.ilog(n: int, b: int) → int`

Computes $x = \lfloor \log_b(n) \rfloor$ such that $b^x \le n < b^{x+1}$.

Parameters

n: int
A positive integer.

b: int
The logarithm base b, must be at least 2.

Returns
The integer logarithm base b of n.

See also

iroot, *isqrt*

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
```

`galois.iroot(n: int, k: int) → int`

Computes $x = \lfloor n^{\frac{1}{k}} \rfloor$ such that $x^k \le n < (x+1)^k$.

Parameters

n: int
A non-negative integer.

k: int
The positive root k.

Returns
The integer k-th root of n.

See also

isqrt, *ilog*

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.isqrt(n: int) → int`

Computes $x = \lfloor \sqrt{n} \rfloor$ such that $x^2 \le n < (x+1)^2$.

Info

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.

See also

iroot, *ilog*

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
```

3.25 Factorization

3.25.1 Prime factorization

`galois.factors(value: int)` → tuple[list[int], list[int]]

`galois.factors(value: Poly)` → tuple[list[*Poly*], list[int]]

Computes the prime factors of a positive integer or the irreducible factors of a non-constant, monic polynomial.

`galois.factors(value: int)` → tuple[list[int], list[int]]

`galois.factors(value: Poly)` → tuple[list[*Poly*], list[int]]

Computes the prime factors of a positive integer or the irreducible factors of a non-constant, monic polynomial.

Parameters

value: int

value: *Poly*

A positive integer n or a non-constant, monic polynomial $f(x)$.

Returns

- Sorted list of prime factors $\{p_1, p_2, \dots, p_k\}$ of n with $p_1 < p_2 < \cdots < p_k$ or irreducible factors $\{g_1(x), g_2(x), \dots, g_k(x)\}$ of $f(x)$ sorted in lexicographical order.
- List of corresponding multiplicities $\{e_1, e_2, \dots, e_k\}$.

Notes

Integers

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:

0. Test if n is in the Cunningham Book's database of $n = p^m \pm 1$ factorizations. If so, return the prime factorization.
1. Test if n is prime. If so, return `[n], [1]`. See *is_prime()*.

2. Test if n is a perfect power, such that $n = x^k$. If so, prime factor x and multiply the exponents by k. See `perfect_power()`.
3. Use trial division with a list of primes up to 10^6. If no residual factors, return the discovered prime factors. See `trial_division()`.
4. Use Pollard's Rho algorithm to find a non-trivial factor of the residual. Continue until all are found. See `pollard_rho()`.

Polynomials

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. See `Poly.square_free_factors()`.
2. Apply the Distinct-Degree Factorization algorithm to factor each square-free polynomial into a product of factors with the same degree. See `Poly.distinct_degree_factors()`.
3. Apply the Equal-Degree Factorization algorithm to factor the product of factors of equal degree into their irreducible factors. See `Poly.equal_degree_factors()`.

This factorization is also available in `Poly.factors()`.

References

- Hachenberger, D. and Jungnickel, D. Topics in Galois Fields. Algorithm 6.1.7.
- Section 2.1 from https://people.csail.mit.edu/dmoshkov/courses/codes/poly-factorization.pdf

Examples

Integers

Construct a composite integer from prime factors.

```
In [1]: n = 2**3 * 3 * 5; n
Out[1]: 120
```

Factor the integer into its prime factors.

```
In [2]: galois.factors(n)
Out[2]: ([2, 3, 5], [3, 1, 1])
```

Polynomials

Generate irreducible polynomials over $\mathrm{GF}(3)$.

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

In [4]: g1 = galois.irreducible_poly(3, 3); g1
Out[4]: Poly(x^3 + 2x + 1, GF(3))

In [5]: g2 = galois.irreducible_poly(3, 4); g2
Out[5]: Poly(x^4 + x + 2, GF(3))

In [6]: g3 = galois.irreducible_poly(3, 5); g3
Out[6]: Poly(x^5 + 2x + 1, GF(3))
```

Construct a composite polynomial.

```
In [7]: e1, e2, e3 = 5, 4, 3

In [8]: f = g1**e1 * g2**e2 * g3**e3; f
Out[8]: Poly(x^46 + x^44 + 2x^41 + x^40 + 2x^39 + 2x^38 + 2x^37 + 2x^36 + 2x^34 + x^
↪33 + 2x^32 + x^31 + 2x^30 + 2x^29 + 2x^28 + 2x^25 + 2x^24 + 2x^23 + x^20 + x^19 +␣
↪x^18 + x^15 + 2x^10 + 2x^8 + x^5 + x^4 + x^3 + 1, GF(3))
```

Factor the polynomial into its irreducible factors over $\mathrm{GF}(3)$.

```
In [9]: galois.factors(f)
Out[9]:
([Poly(x^3 + 2x + 1, GF(3)),
  Poly(x^4 + x + 2, GF(3)),
  Poly(x^5 + 2x + 1, GF(3))],
 [5, 4, 3])
```

3.25.2 Composite factorization

`galois.divisor_sigma`(n: *int*, k: *int* = *1*) → int

Returns the sum of k-th powers of the positive divisors of n.

`galois.divisors`(n: *int*) → list[int]

Computes all positive integer divisors d of the integer n such that $d\ |n$.

`galois.divisor_sigma`(n: *int*, k: *int* = *1*) → int

Returns the sum of k-th powers of the positive divisors of n.

Parameters

n: int
An integer.

k: int = 1
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 $\sigma_k(n)$.

See also

factors, *divisors*

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
```

galois.**divisors**(n: *int*) → list[int]

Computes all positive integer divisors d of the integer n such that $d\ |n$.

Parameters

n: int
An integer.

Returns
Sorted list of positive integer divisors d of n.

See also

factors, *divisor_sigma*

Notes

The *divisors()* function finds *all* positive integer divisors or factors of n, where the *factors()* function 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])
```

3.25.3 Specific factorization algorithms

`galois.perfect_power(n: int) → tuple[int, int]`

Returns the integer base c and exponent e of $n = c^e$. If n is *not* a perfect power, then $c = n$ and $e = 1$.

`galois.pollard_p1(n: int, B: int, B2: int | None = None) → int`

Attempts to find a non-trivial factor of n if it has a prime factor p such that $p - 1$ is B-smooth.

`galois.pollard_rho(n: int, c: int = 1) → int`

Attempts to find a non-trivial factor of n using cycle detection.

`galois.trial_division(n, ...) → tuple[list[int], list[int], int]`

Finds all the prime factors $p_i^{e_i}$ of n for $p_i \leq B$.

`galois.perfect_power(n: int) → tuple[int, int]`

Returns the integer base c and exponent e of $n = c^e$. If n is *not* a perfect power, then $c = n$ and $e = 1$.

Parameters

n: int
An integer.

Returns

- The *potentially* composite base c.
- The exponent e.

See also

factors, *is_perfect_power*, *is_prime_power*

Examples

Primes are not perfect powers because their exponent is 1.

```
In [1]: n = 13

In [2]: galois.perfect_power(n)
Out[2]: (13, 1)

In [3]: galois.is_perfect_power(n)
Out[3]: False
```

Products of primes are not perfect powers.

```
In [4]: n = 5 * 7

In [5]: galois.perfect_power(n)
Out[5]: (35, 1)

In [6]: galois.is_perfect_power(n)
Out[6]: False
```

Products of prime powers where the GCD of the exponents is 1 are not perfect powers.

```
In [7]: n = 2 * 3 * 5**3

In [8]: galois.perfect_power(n)
Out[8]: (750, 1)

In [9]: galois.is_perfect_power(n)
Out[9]: False
```

Products of prime powers where the GCD of the exponents is greater than 1 are perfect powers.

```
In [10]: n = 2**2 * 3**2 * 5**4

In [11]: galois.perfect_power(n)
Out[11]: (150, 2)

In [12]: galois.is_perfect_power(n)
Out[12]: True
```

Negative integers can be perfect powers if they can be factored with an odd exponent.

```
In [13]: n = -64

In [14]: galois.perfect_power(n)
Out[14]: (-4, 3)

In [15]: galois.is_perfect_power(n)
Out[15]: True
```

Negative integers that are only factored with an even exponent are not perfect powers.

```
In [16]: n = -100

In [17]: galois.perfect_power(n)
Out[17]: (-100, 1)

In [18]: galois.is_perfect_power(n)
Out[18]: False
```

`galois.pollard_p1(n: int, B: int, B2: int | None = None) → int`

Attempts to find a non-trivial factor of n if it has a prime factor p such that $p-1$ is B-smooth.

Parameters

n: int
An odd composite integer $n > 2$ that is not a prime power.

B: int
The smoothness bound $B > 2$.

B2: int | None = None
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.

Raises
`RuntimeError` – If a non-trivial factor cannot be found.

See also

factors, *pollard_rho*

Notes

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 $p_i \le B$.

A extension of Pollard's $p-1$ algorithm allows a prime factor p to be B-smooth with the exception of one prime factor $B < p_{k+1} \le B_2$. 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 $B_2 \gg B$.

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)
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
Cell In[4], line 1
----> 1 galois.pollard_p1(p*q, 15)

File ~/checkouts/readthedocs.org/user_builds/galois/envs/latest/lib/python3.8/site-
↪packages/galois/_prime.py:1181, in pollard_p1(n, B, B2)
   1178     if d not in [1, n]:
   1179         return d
-> 1181 raise RuntimeError(
   1182     f"A non-trivial factor of {n} could not be found using the Pollard p-1␣
↪algorithm "
   1183     f"with smoothness bound {B} and secondary bound {B2}."
   1184 )

RuntimeError: A non-trivial factor of 1934313240757 could not be found using the␣
↪Pollard p-1 algorithm with smoothness bound 15 and secondary bound None.
```

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
```

(continues on next page)

(continued from previous page)

```
In [10]: 37*41
Out[10]: 1517
```

galois.**pollard_rho**(n: *int*, c: *int* = *1*) → int

Attempts to find a non-trivial factor of n using cycle detection.

Parameters

n: int
An odd composite integer $n > 2$ that is not a prime power.

c: int = 1
The constant offset in the function $f(x) = x^2 + c \bmod n$. The default is 1. A requirement of the algorithm is that $c \notin \{0, -2\}$.

Returns
A non-trivial factor m of n.

Raises
RuntimeError – If a non-trivial factor cannot be found.

See also

factors, *pollard_p1*

Notes

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 $x_i = f(x_{i-1}) = x_{i-1}^2 + 1 \bmod p$ where p is an unknown small prime factor of n. This happens when $x_m \equiv x_{2m} \pmod p$. Because p is unknown, this is accomplished by computing the sequence modulo n and looking for $\gcd(x_m - x_{2m}, n) > 1$.

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
```

galois.**trial_division**(n: *int*, B: *int* | *None* = **None**) → tuple[list[int], list[int], int]

Finds all the prime factors $p_i^{e_i}$ of n for $p_i \le B$.

The trial division factorization will find all prime factors $p_i \le B$ 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 | None = None
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

- The discovered prime factors $\{p_1, \dots, p_k\}$.
- The corresponding prime exponents $\{e_1, \dots, e_k\}$.
- The residual factor n_r.

See also

factors

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)
```

3.26 Primes

3.26.1 Prime number generation

galois.***kth_prime***(k: *int*) → int

Returns the k-th prime, where $k = \{1, 2, 3, 4, \dots\}$ for primes $p = \{2, 3, 5, 7, \dots\}$.

galois.***mersenne_exponents***(n: *int* | *None* = **None**) → list[int]

Returns all known Mersenne exponents e for $e \le n$.

galois.***mersenne_primes***(n: *int* | *None* = **None**) → list[int]

Returns all known Mersenne primes p for $p \le 2^n - 1$.

`galois.next_prime(n: int) → int`

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

`galois.prev_prime(n: int) → int`

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

`galois.primes(n: int) → list[int]`

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

`galois.random_prime(bits: int, seed: int | None = None) → int`

Returns a random prime p with b bits, such that $2^b \leq p < 2^{b+1}$.

`galois.kth_prime(k: int) → int`

Returns the k-th prime, where $k = \{1, 2, 3, 4, \dots\}$ for primes $p = \{2, 3, 5, 7, \dots\}$.

Parameters

k: int
The prime index (1-indexed).

Returns
The k-th prime.

See also

primes, *prev_prime*, *next_prime*

Examples

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

In [2]: galois.kth_prime(2)
Out[2]: 3

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

In [4]: galois.kth_prime(1000)
Out[4]: 7919
```

`galois.mersenne_exponents(n: int | None = None) → list[int]`

Returns all known Mersenne exponents e for $e \leq n$.

Parameters

n: int | None = None
The max exponent of 2. The default is `None` which returns all known Mersenne exponents.

Returns
The list of Mersenne exponents e for $e \leq n$.

See also

mersenne_primes

Notes

A Mersenne exponent e is an exponent of 2 such that $2^e - 1$ is prime.

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**(n: *int* | *None* = ***None***) → list[int]

Returns all known Mersenne primes p for $p \le 2^n - 1$.

Parameters

n: int | None = None
The max power of 2. The default is **None** which returns all known Mersenne exponents.

Returns
The list of known Mersenne primes p for $p \le 2^n - 1$.

See also

mersenne_exponents

Notes

Mersenne primes are primes that are one less than a power of 2.

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,
␣
↪6864797660130609714981900799081393217269435300143305409394463459185543183397656052122559640661454
↪
␣
↪5311379928167670986895882065524686273295931177270319231994441382004035598608522427391625022652292
↪
␣
↪1040793219466439908192524032736408553861526224726670480531911235040360805967336029801223944173232

In [2]: galois.is_prime(p[-1])
Out[2]: True
```

`galois.`**`next_prime`**(n: *int*) → int

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

Parameters

n: int
An integer.

Returns
The nearest prime $p > n$.

See also

primes, *kth_prime*, *prev_prime*

Examples

```
In [1]: galois.next_prime(13)
Out[1]: 17

In [2]: galois.next_prime(15)
Out[2]: 17

In [3]: galois.next_prime(6852976918500265458318414454675831645298)
Out[3]: 6852976918500265458318414454675831645343
```

`galois.prev_prime(n: int) → int`

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

Parameters

n: int
An integer $n \geq 2$.

Returns
The nearest prime $p \leq n$.

See also

primes, *kth_prime*, *next_prime*

Examples

```
In [1]: galois.prev_prime(13)
Out[1]: 13

In [2]: galois.prev_prime(15)
Out[2]: 13

In [3]: galois.prev_prime(6298891201241929548477199440981228280038)
Out[3]: 6298891201241929548477199440981228279991
```

`galois.primes(n: int) → list[int]`

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

Parameters

n: int
An integer.

Returns
All primes up to and including n. If $n < 2$, the function returns an empty list.

See also

kth_prime, *prev_prime*, *next_prime*

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.random_prime(bits: int, seed: int | None = None)` → int

Returns a random prime p with b bits, such that $2^b \leq p < 2^{b+1}$.

Parameters

bits: int
The number of bits in the prime p.

seed: int | None = None
Non-negative integer used to initialize the PRNG. The default is `None` which means that unpredictable entropy will be pulled from the OS to be used as the seed.

Returns

A random prime in $2^b \leq p < 2^{b+1}$.

See also

prev_prime, *next_prime*

Notes

This function randomly generates integers with b bits and uses the primality tests in `is_prime()` to determine if p is prime.

References

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

Examples

Generate a random 1024-bit prime.

```
In [1]: p = galois.random_prime(1024, seed=1); p
Out[1]:␣
↪32784589758621343675108188287125533128664890283638683908761736860843957469819201604376953382347406

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

Verify that p is prime using the OpenSSL library.

```
$ openssl prime␣
↪32784589758621343675108188287125533128664890283638683908761736860843957469819201604376953382347406
1D2DE38DE88C67E1EAFDEEAE77C40B8709ED9C275522C6D5578976B1ABCBE7E0F8C6DE1271EEC6EB38276491641897 88F9E
↪(3278458975862134367510818828712553312866489028363868390876173686084395746981920160437695338234740
↪is prime
```

3.26.2 Primality tests

galois.***is_composite***(n: *int*) → bool

Determines if n is composite.

galois.***is_perfect_power***(n: *int*) → bool

Determines if n is a perfect power $n = c^e$ with $e > 1$.

galois.***is_powersmooth***(n: *int*, B: *int*) → bool

Determines if the integer n is B-powersmooth.

galois.***is_prime***(n: *int*) → bool

Determines if n is prime.

galois.***is_prime_power***(n: *int*) → bool

Determines if n is a prime power $n = p^k$ for prime p and $k \geq 1$.

galois.***is_smooth***(n: *int*, B: *int*) → bool

Determines if the integer n is B-smooth.

galois.***is_square_free***(value: *int*) → bool

galois.***is_square_free***(value: Poly) → bool

Determines if an integer or polynomial is square-free.

galois.**is_composite**(n: *int*) → bool

Determines if n is composite.

Parameters

n: int
An integer.

Returns
`True` if the integer n is composite.

See also

is_prime, *is_square_free*, *is_perfect_power*

Examples

```
In [1]: galois.is_composite(13)
Out[1]: False

In [2]: galois.is_composite(15)
Out[2]: True
```

galois.**is_perfect_power**(n: *int*) → bool

Determines if n is a perfect power $n = c^e$ with $e > 1$.

Parameters

n: int
An integer.

Returns
True if the integer n is a perfect power.

See also

is_prime_power, *is_square_free*

Examples

Primes are not perfect powers because their exponent is 1.

```
In [1]: galois.perfect_power(13)
Out[1]: (13, 1)

In [2]: galois.is_perfect_power(13)
Out[2]: False
```

Products of primes are not perfect powers.

```
In [3]: galois.perfect_power(5*7)
Out[3]: (35, 1)

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

Products of prime powers where the GCD of the exponents is 1 are not perfect powers.

```
In [5]: galois.perfect_power(2 * 3 * 5**3)
Out[5]: (750, 1)
```

(continues on next page)

(continued from previous page)

```
In [6]: galois.is_perfect_power(2 * 3 * 5**3)
Out[6]: False
```

Products of prime powers where the GCD of the exponents is greater than 1 are perfect powers.

```
In [7]: galois.perfect_power(2**2 * 3**2 * 5**4)
Out[7]: (150, 2)

In [8]: galois.is_perfect_power(2**2 * 3**2 * 5**4)
Out[8]: True
```

Negative integers can be perfect powers if they can be factored with an odd exponent.

```
In [9]: galois.perfect_power(-64)
Out[9]: (-4, 3)

In [10]: galois.is_perfect_power(-64)
Out[10]: True
```

Negative integers that are only factored with an even exponent are not perfect powers.

```
In [11]: galois.perfect_power(-100)
Out[11]: (-100, 1)

In [12]: galois.is_perfect_power(-100)
Out[12]: False
```

galois.**is_powersmooth**(n: *int*, B: *int*) → bool

Determines if the integer n is B-powersmooth.

Parameters

n: int
An integer.

B: int
The smoothness bound $B \geq 2$.

Returns
`True` if n is B-powersmooth.

See also

factors, *is_smooth*

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
```

`galois.is_prime(n: int) → bool`

Determines if n is prime.

Parameters

n: int
An integer.

Returns
`True` if the integer n is prime.

See also

is_composite, *is_prime_power*, *is_perfect_power*

Notes

This algorithm will first run Fermat's primality test to check n for compositeness, see *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 *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**(n: *int*) → bool

Determines if n is a prime power $n = p^k$ for prime p and $k \geq 1$.

Parameters

n: int
An integer.

Returns
True if the integer n is a prime power.

See also

is_perfect_power, *is_prime*

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_smooth**(n: *int*, B: *int*) → bool

Determines if the integer n is B-smooth.

Parameters

n: int
An integer.

B: int
The smoothness bound $B \geq 2$.

Returns
True if n is B-smooth.

See also

factors, *is_powersmooth*

Notes

An integer n with prime factorization $n = p_1^{e_1} \dots p_k^{e_k}$ is B-smooth if $p_k \le B$. 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_square_free**(value: *int*) → bool

galois.**is_square_free**(value: Poly) → bool

Determines if an integer or polynomial is square-free.

Parameters

value: int
value: *Poly*
An integer n or polynomial $f(x)$.

Returns
`True` if the integer or polynomial is square-free.

See also

is_prime_power, *is_perfect_power*

Notes

Integers

A square-free integer n is divisible by no perfect squares. As a consequence, the prime factorization of a square-free integer n is

$$n = \prod_{i=1}^{k} p_i^{e_i} = \prod_{i=1}^{k} p_i.$$

Polynomials

A square-free polynomial $f(x)$ has no irreducible factors with multiplicity greater than one. Therefore, its canonical factorization is

$$f(x) = \prod_{i=1}^{k} g_i(x)^{e_i} = \prod_{i=1}^{k} g_i(x).$$

This test is also available in *Poly.is_square_free()*.

Examples

Integers

Determine if integers are square-free.

```
In [1]: galois.is_square_free(10)
Out[1]: True

In [2]: galois.is_square_free(18)
Out[2]: False
```

Polynomials

Generate irreducible polynomials over $\mathrm{GF}(3)$.

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

In [4]: f1 = galois.irreducible_poly(3, 3); f1
Out[4]: Poly(x^3 + 2x + 1, GF(3))

In [5]: f2 = galois.irreducible_poly(3, 4); f2
Out[5]: Poly(x^4 + x + 2, GF(3))
```

Determine if composite polynomials are square-free over $\mathrm{GF}(3)$.

```
In [6]: galois.is_square_free(f1 * f2)
Out[6]: True

In [7]: galois.is_square_free(f1**2 * f2)
Out[7]: False
```

3.26.3 Specific primality tests

`galois.fermat_primality_test(n: int, ...) → bool`

Determines if n is composite using Fermat's primality test.

`galois.miller_rabin_primality_test(n: int, a: int = 2, ...) → bool`

Determines if n is composite using the Miller-Rabin primality test.

`galois.fermat_primality_test(n: int, a: int | None = None, rounds: int = 1) → bool`

Determines if n is composite using Fermat's primality test.

Parameters

n: int
An odd integer $n \geq 3$.

a: int | None = None
An integer in $2 \leq a \leq n-2$. The default is `None` which selects a random a.

rounds: int = 1
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 a probable prime.

See also

is_prime, *miller_rabin_primality_test*

Notes

Fermat's theorem says that for prime p and $1 \leq a \leq p-1$, the congruence $a^{p-1} \equiv 1 \pmod p$ holds. Fermat's primality test of n computes $a^{n-1} \bmod n$ for some $1 \leq a \leq n-1$. If a is such that $a^{p-1} \not\equiv 1 \pmod p$, then a is said to be a *Fermat witness* to the compositeness of n. If n is composite and $a^{p-1} \equiv 1 \pmod p$, then a is said to be a *Fermat liar* to the primality of n.

Since $a = \{1, n-1\}$ are Fermat liars for all composite n, it is common to reduce the range of possible a to $2 \leq a \leq n-2$.

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]
```

(continues on next page)

(continued from previous page)

```
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]: [True, True, False]
```

And the pseudoprimes base 3 from A005935.

```
# List of some Fermat pseudoprimes to base 3
In [8]: pseudoprimes = [2465, 7381, 16531]

In [9]: [galois.is_prime(p) for p in pseudoprimes]
Out[9]: [False, False, False]

# The pseudoprimes base 3 satisfy 3^(p-1) = 1 (mod p)
In [10]: [galois.fermat_primality_test(p, a=3) for p in pseudoprimes]
Out[10]: [True, True, True]

# But they may not satisfy a^(p-1) = 1 (mod p) for other a
In [11]: [galois.fermat_primality_test(p) for p in pseudoprimes]
Out[11]: [False, True, False]
```

`galois.miller_rabin_primality_test(n: int, a: int = 2, rounds: int = 1) → bool`

Determines if n is composite using the Miller-Rabin primality test.

Parameters

n: int
An odd integer $n \geq 3$.

a: int = 2
An integer in $2 \leq a \leq n - 2$. The default is 2.

rounds: int = 1
The number of iterations attempting to detect n as composite. Additional rounds will choose consecutive primes for a. The default is 1.

Returns
`False` if n is shown to be composite. `True` if n is probable prime.

See also

is_prime, *fermat_primality_test*

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 $0 \le j \le s - 1$.

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 $0 \le j \le s - 1$, 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 $0 \le j \le s - 1$, 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 $2 \le a \le n - 2$.

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
```

3.27 Configuration

galois.***get_printoptions***() → dict[str, Any]

Returns the current print options for the package. This function is the *galois* equivalent of numpy.get_printoptions().

galois.***printoptions***(***kwargs*) → Generator[None, None, None]

A context manager to temporarily modify the print options for the package. This function is the *galois* equivalent of numpy.printoptions().

galois.***set_printoptions***(...)

Modifies the print options for the package. This function is the *galois* equivalent of numpy.set_printoptions().

galois.**get_printoptions**() → dict[str, Any]

Returns the current print options for the package. This function is the *galois* equivalent of numpy.get_printoptions().

Returns

A dictionary of current print options.

See also

set_printoptions, *printoptions*

Examples

```
In [1]: galois.get_printoptions()
Out[1]: {'coeffs': 'desc'}
```

```
In [2]: galois.set_printoptions(coeffs="asc")

In [3]: galois.get_printoptions()
Out[3]: {'coeffs': 'asc'}
```

galois.**printoptions**(***kwargs*) → Generator[None, None, None]

A context manager to temporarily modify the print options for the package. This function is the *galois* equivalent of numpy.printoptions().

See *set_printoptions()* for the full list of available options.

Returns

A context manager for use in a **with** statement. The print options are only modified inside the **with** block.

See also

set_printoptions, *get_printoptions*

Examples

By default, polynomials are displayed with descending degrees.

```
In [1]: GF = galois.GF(3**5, repr="poly")

In [2]: a = GF([109, 83])

In [3]: f = galois.Poly([3, 0, 5, 2], field=galois.GF(7))
```

Modify the print options only inside the context manager.

```
In [4]: print(a); print(f)
[^4 + ^3 + 1       ^4 + 2]
3x^3 + 5x + 2

In [5]: with galois.printoptions(coeffs="asc"):
   ...:     print(a); print(f)
   ...:
[1 + ^3 + ^4       2 + ^4]
2 + 5x + 3x^3

In [6]: print(a); print(f)
[^4 + ^3 + 1       ^4 + 2]
3x^3 + 5x + 2
```

galois.**set_printoptions**(coeffs: *Literal[desc]* | *Literal[asc]* = *'desc'*)

Modifies the print options for the package. This function is the `galois` equivalent of `numpy.set_printoptions()`.

Parameters

coeffs: Literal[desc] | Literal[asc] = 'desc'
The order in which to print the coefficients, either in descending degrees (default) or ascending degrees.

See also

get_printoptions, *printoptions*

Examples

By default, polynomials are displayed with descending degrees.

```
In [1]: GF = galois.GF(3**5, repr="poly")

In [2]: a = GF([109, 83]); a
Out[2]: GF([^4 + ^3 + 1,       ^4 + 2], order=3^5)

In [3]: f = galois.Poly([3, 0, 5, 2], field=galois.GF(7)); f
Out[3]: Poly(3x^3 + 5x + 2, GF(7))
```

Modify the print options to display polynomials with ascending degrees.

```
In [4]: galois.set_printoptions(coeffs="asc")

In [5]: a
Out[5]: GF([1 + ^3 + ^4,        2 + ^4], order=3^5)

In [6]: f
Out[6]: Poly(2 + 5x + 3x^3, GF(7))
```

3.28 Versioning

The *galois* library uses semantic versioning. Releases are versioned `major.minor.patch`.

Major versions introduce API-changing features. Minor versions add features that are backwards-compatible with other releases. Patch versions make backwards-compatible bug fixes.

3.28.1 Alpha releases

Versions before `0.1.0` are alpha releases. Alpha releases are versioned `0.0.alpha`. There is no API compatibility guarantee for them. They can be thought of as `0.0.alpha-major`.

3.28.2 Beta releases

Versions before `1.0.0` are beta releases. Beta releases are versioned `0.beta.x` and are API-compatible. They can be thought of as `0.beta-major.beta-minor`.

3.29 v0.3

3.29.1 v0.3.0

Released December 9, 2022

Breaking changes

- Increased minimum NumPy version to 1.21.0. (#441)
- Increased minimum Numba version to 0.55.0 (#441)
- Modified `galois.GF()` and `galois.Field()` so that keyword arguments `irreducible_poly`, `primitive_element`, `verify`, `compile`, and `repr` may no longer be passed as positional arguments. (#442)

Changes

- Added a `galois.GF(p, m)` call signature in addition to `galois.GF(p**m)`. This also applies to `galois.Field()`. Separately specifying p and m eliminates the need to factor the order p^m in very large finite fields. (#442)

```
>>> import galois
# This is faster than galois.GF(2**409)
>>> GF = galois.GF(2, 409)
>>> print(GF.properties)
Galois Field:
  name: GF(2^409)
  characteristic: 2
  degree: 409
  order:␣
↪132211193758049719790383061606554207965680936592856243856929759054881158247262269165037842087943
  irreducible_poly: x^409 + x^7 + x^5 + x^3 + 1
  is_primitive_poly: True
  primitive_element: x
```

- Optimized matrix multiplication by parallelizing across multiple cores. (#440)

```
In [1]: import galois

In [2]: GF = galois.GF(3**5)

In [3]: A = GF.Random((300, 400), seed=1)

In [4]: B = GF.Random((400, 500), seed=2)

# v0.2.0
In [5]: %timeit A @ B
664 ms ± 3.31 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

# v0.3.0
In [5]: %timeit A @ B
79.1 ms ± 7.32 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

- Optimized polynomial evaluation by parallelizing across multiple cores. (#440)

```
In [1]: import galois

In [2]: GF = galois.GF(3**5)

In [3]: f = galois.Poly.Random(100, seed=1, field=GF)

In [4]: x = GF.Random(100_000, seed=1)

# v0.2.0
In [5]: %timeit f(x)
776 ms ± 2.12 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

# v0.3.0
In [5]: %timeit f(x)
```

(continues on next page)

(continued from previous page)

```
13.9 ms ± 2.51 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

- Fixed an occasional arithmetic type error in binary extension fields $\mathrm{GF}(2^m)$ when using the built-in `np.bitwise_xor()`. (#441)

Contributors

- Matt Hostetter (@mhostetter)

3.29.2 v0.3.1

Released December 12, 2022

Changes

- Fixed a bug in the Pollard ρ factorization algorithm that caused an occasional infinite loop. (#450)

```
In [1]: import galois

# v0.3.0
In [2]: %time galois.GF(2400610585866217)
# Never returns...

# v0.3.1
In [2]: %time galois.GF(2400610585866217)
Wall time: 96 ms
Out[2]: <class 'galois.GF(2400610585866217)'>
```

- Formatted the code and unit tests with `black` and `isort`. (#446, #449)

Contributors

- Matt Hostetter (@mhostetter)
- @pivis

3.29.3 v0.3.2

Released December 18, 2022

Changes

- Added a prime factorization database for $n = b^k \pm 1$, with $b \in \{2, 3, 5, 6, 7, 10, 11, 12\}$. The factorizations are from the Cunningham Book. This speeds up the creation of large finite fields. (#452)

```
In [1]: import galois

# v0.3.1
In [2]: %time galois.factors(2**256 - 1)
# Took forever...

# v0.3.2
In [2]: %time galois.factors(2**256 - 1)
Wall time: 1 ms
Out[2]:
([3,
  5,
  17,
  257,
  641,
  65537,
  274177,
  6700417,
  67280421310721,
  59649589127497217,
  5704689200685129054721],
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1])
```

- Added speed-up when factoring powers of small primes. This speeds up the creation of large finite fields. (#454)

```
In [1]: import galois

# v0.3.1
In [2]: %time galois.factors(2**471)
Wall time: 4.18 s
Out[2]: ([2], [471])

# v0.3.2
In [2]: %time galois.factors(2**471)
Wall time: 2 ms
Out[2]: ([2], [471])
```

- Added four additional Mersenne primes that were discovered between 2013-2018. (#452)

Contributors

- Matt Hostetter (@mhostetter)

3.29.4 v0.3.3

Released February 1, 2023

Changes

- Added a `terms` keyword argument to `irreducible_poly()`, `irreducible_polys()`, `primitive_poly()`, and `primitive_polys()` to find a polynomial with a desired number of non-zero terms. This may be set to an integer or to `"min"`. (#463)

```
>>> import galois
>>> galois.irreducible_poly(7, 9)
Poly(x^9 + 2, GF(7))
>>> galois.irreducible_poly(7, 9, terms=3)
Poly(x^9 + x + 1, GF(7))
>>> galois.primitive_poly(7, 9)
Poly(x^9 + x^2 + x + 2, GF(7))
>>> galois.primitive_poly(7, 9, terms="min")
Poly(x^9 + 3x^2 + 4, GF(7))
```

- Added a database of binary irreducible polynomials with degrees less than 10,000. These polynomials are lexicographically-first and have the minimum number of non-zero terms. The database is accessed in `irreducible_poly()` when `terms="min"` and `method="min"`. (#462)

```
In [1]: import galois

# Manual search
In [2]: %time galois.irreducible_poly(2, 1001)
CPU times: user 6.8 s, sys: 0 ns, total: 6.8 s
Wall time: 6.81 s
Out[2]: Poly(x^1001 + x^5 + x^3 + x + 1, GF(2))

# With the database
In [3]: %time galois.irreducible_poly(2, 1001, terms="min")
CPU times: user 745 µs, sys: 0 ns, total: 745 µs
Wall time: 1.4 ms
Out[3]: Poly(x^1001 + x^17 + 1, GF(2))
```

- Memoized expensive polynomial tests `Poly.is_irreducible()` and `Poly.is_primitive()`. Now, the expense of those calculations for a given polynomial is only incurred once. (#470)

```
In [1]: import galois

In [2]: f = galois.Poly.Str("x^1001 + x^17 + 1"); f
Out[2]: Poly(x^1001 + x^17 + 1, GF(2))

In [3]: %time f.is_irreducible()
CPU times: user 1.05 s, sys: 3.47 ms, total: 1.05 s
```

(continues on next page)

(continued from previous page)

```
Wall time: 1.06 s
Out[3]: True

In [4]: %time f.is_irreducible()
CPU times: user 57 µs, sys: 30 µs, total: 87 µs
Wall time: 68.2 µs
Out[4]: True
```

- Added tests for Conway polynomials `Poly.is_conway()` and `Poly.is_conway_consistent()`. (#469)
- Added the ability to manually search for a Conway polynomial if it is not found in Frank Luebeck's database, using `conway_poly(p, m, search=True)`. (#469)
- Various documentation improvements.

Contributors

- Iyán Méndez Veiga (@iyanmv)
- Matt Hostetter (@mhostetter)

3.29.5 v0.3.4

Released May 2, 2023

Changes

- Added support for Python 3.11. (#415)
- Added support for NumPy 1.24. (#415)
- Fixed indexing bug in `FieldArray.plu_decompose()` for certain input arrays. (#477)

Contributors

- Matt Hostetter (@mhostetter)

3.29.6 v0.3.5

Released May 9, 2023

Changes

- Added `py.typed` file to indicate to `mypy` and other type checkers that `galois` supports typing. (#481)
- Fixed bug with multiple, concurrent BCH and/or Reed Solomon decoders. (#484)

Contributors

- Matt Hostetter (@mhostetter)

3.29.7 v0.3.6

Released October 1, 2023

Changes

- Added support for NumPy 1.25. (#507)
- Added support for Numba 0.58. (#507)
- Fixed rare overflow with computing a large modular exponentiation of polynomials. (#488)
- Resolved various deprecations warnings with NumPy 1.25. (#492)

Contributors

- Iyán Méndez Veiga (@iyanmv)
- @Lasagnenator
- Matt Hostetter (@mhostetter)

3.29.8 v0.3.7

Released November 30, 2023

Changes

- Added wheel factorization for finding large primes. (#527)
- Improved type annotations. (#510, #511)
- Removed optional `[dev]` extra. If developing, install from `requirements-dev.txt`. (#521)
- Fixed bugs in `prev_prime()` and `next_prime()` for large primes. (#527)

Contributors

- @avadov
- Matt Hostetter (@mhostetter)

3.29.9 v0.3.8

Released February 1, 2024

Changes

- Added support for Python 3.12. (#534)
- Added support for NumPy 1.26. (#534)
- Added support for Numba 0.59. (#534)
- Fixed bug in `FieldArray.multiplicative_order()` for large fields. (#533)

Contributors

- Matt Hostetter (@mhostetter)

3.30 v0.2

3.30.1 v0.2.0

Released November 17, 2022

Breaking changes

- Refactored FEC classes and usage. (#413, #435)
  - Modified `BCH` codes to support q-ary, non-primitive, and non narrow-sense codes.
  - Modified `ReedSolomon` codes to support non-primitive codes.
  - Enabled instantiation of a BCH or ReedSolomon code by specifying `(n, k)` or `(n, d)`.
  - Removed `parity_only=False` keyword argument from FEC `encode()` methods and replaced with `output="codeword"`.
  - Removed `bch_valid_codes()` from the API. Instead, use `galois.BCH(n, d=d)` to find and create a BCH code with codeword size `n` and design distance `d`. For example, here is how to find various code sizes of primitive BCH codes over `GF(5)`.

```
>>> import galois
>>> GF = galois.GF(5)
>>> for d in range(3, 10):
...     bch = galois.BCH(5**2 - 1, d=d, field=GF)
...     print(repr(bch))
...
<BCH Code: [24, 20, 3] over GF(5)>
<BCH Code: [24, 18, 4] over GF(5)>
<BCH Code: [24, 16, 5] over GF(5)>
<BCH Code: [24, 16, 6] over GF(5)>
<BCH Code: [24, 15, 7] over GF(5)>
<BCH Code: [24, 13, 8] over GF(5)>
<BCH Code: [24, 11, 9] over GF(5)>
```

- Removed generator_to_parity_check_matrix(), parity_check_to_generator_matrix(), poly_to_generator_matrix(), and roots_to_parity_check_matrix() from the API.

- Renamed properties and methods for changing the finite field element representation. (#436)
  - Renamed display keyword argument in GF() to repr.
  - Renamed FieldArray.display() classmethod to FieldArray.repr().
  - Renamed FieldArray.display_mode property to FieldArray.element_repr.

```
>>> import galois
>>> GF = galois.GF(3**4, repr="poly")
>>> x = GF.Random(2, seed=1); x
GF([2^3 + 2^2 + 2 + 2,          2^3 + 2^2], order=3^4)
>>> GF.repr("power"); x
GF([^46, ^70], order=3^4)
>>> GF.element_repr
'power'
```

Changes

- Added output="codeword" keyword argument to FEC encode() methods. (#435)
- Added output="message" keyword argument to FEC decode() methods. (#435)
- Standardized NumPy scalar return types (np.bool_ and np.int64) to Python types (bool and int). For example, in FieldArray.multiplicative_order(). (#437)
- Improved documentation and published docs for pre-release versions (e.g., v0.3.x).

Contributors

- Matt Hostetter (@mhostetter)

3.31 v0.1

3.31.1 v0.1.0

Released August 27, 2022

Changes

- First beta release!
- Fixed PyPI package metadata.

Contributors

- Matt Hostetter (@mhostetter)

3.31.2 v0.1.1

Released September 2, 2022

Changes

- Added support for NumPy 1.23. (#414)
- Added `seed` keyword argument to `random_prime()`. (#409)

```
>>> galois.random_prime(100, seed=1)
2218840874040723579228056294021
>>> galois.random_prime(100, seed=1)
2218840874040723579228056294021
```

- Deployed documentation to https://mhostetter.github.io/galois/latest/ with GitHub Pages. (#408)

Contributors

- Matt Hostetter (@mhostetter)

3.31.3 v0.1.2

Released November 9, 2022

Changes

- Fixed major inefficiency when dividing an array by a scalar or smaller (broadcasted) array. (#429)

```
In [1]: import galois

In [2]: GF = galois.GF(31**5)

In [3]: x = GF.Random(10_000, seed=1); x
Out[3]:
GF([13546990, 14653018, 21619804, ..., 15507037, 24669161, 19116362],
   order=31^5)

In [4]: y = GF.Random(1, seed=2); y
Out[4]: GF([23979074], order=31^5)

# v0.1.1
In [5]: %timeit x / y
261 ms ± 5.67 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

# v0.1.2
```

(continues on next page)

(continued from previous page)

```
In [5]: %timeit x / y
8.23 ms ± 51 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

- Optimized `lagrange_poly()` by adding a custom JIT-compilable routine. (#432)

```
In [1]: import galois

In [2]: GF = galois.GF(13693)

In [3]: x = GF.Random(100, seed=1)

In [4]: y = GF.Random(100, seed=2)

# v0.1.1
In [5]: %timeit galois.lagrange_poly(x, y)
2.85 s ± 3.25 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

# v0.1.2
In [5]: %timeit galois.lagrange_poly(x, y)
4.77 ms ± 190 µs per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

- Added ability in `FieldArray.row_reduce()` to solve for an identity matrix on the right side of a matrix using the `eye` keyword argument. (#426)

```
>>> import galois
>>> GF = galois.GF(31)
>>> A = GF([[16, 12, 1, 25], [1, 10, 27, 29], [1, 0, 3, 19]])
>>> A.row_reduce()
GF([[ 1,  0,  0, 11],
    [ 0,  1,  0,  7],
    [ 0,  0,  1, 13]], order=31)
>>> A.row_reduce(eye="right")
GF([[ 5,  1,  0,  0],
    [27,  0,  1,  0],
    [17,  0,  0,  1]], order=31)
```

- Removed comma separators in `FieldArray.__str__()` to be consistent with NumPy's use of `str()` and `repr()`. (#432)

```
>>> import galois
>>> GF = galois.GF(3**5, display="power")
>>> x = GF.Random((3, 4), seed=1)
>>> x
GF([[^185, ^193,  ^49, ^231],
    [ ^81,  ^60,   ^5,  ^41],
    [ ^50, ^161, ^151, ^171]], order=3^5)
>>> print(x)
[[^185 ^193  ^49 ^231]
 [ ^81  ^60   ^5  ^41]
 [ ^50 ^161 ^151 ^171]]
```

- Modernized type annotations to use abbreviated notation. For example, `a | b` instead of `Union[a, b]`. (#418)
- Added `Self` type annotation where appropriate. (#420)

- Updated documentation and improved examples. (#424, #430)

Contributors

- Matt Hostetter (@mhostetter)

3.32 v0.0

3.32.1 v0.0.14

Released May 7, 2021

Breaking changes

- Rename `GFArray.Eye()` to `GFArray.Identity()`.
- Rename `chinese_remainder_theorem()` to `crt()`.

Changes

- Lots of performance improvements.
- Additional linear algebra support.
- Various bug fixes.

Contributors

- Baalateja Kataru (@BK-Modding)
- Matt Hostetter (@mhostetter)

3.32.2 v0.0.15

Released May 12, 2021

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()`.

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.

Contributors

- Matt Hostetter (@mhostetter)

3.32.3 v0.0.16

Released May 19, 2021

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.

Contributors

- Matt Hostetter (@mhostetter)

3.32.4 v0.0.17

Released June 15, 2021

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.

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.

Contributors

- Dominik Wernberger (@Werni2A)
- Matt Hostetter (@mhostetter)

3.32.5 v0.0.18

Released July 6, 2021

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"]`.

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.

Contributors

- Matt Hostetter (@mhostetter)

3.32.6 v0.0.19

Released August 9, 2021

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 subclass. To return a Galois field array, use `x.copy()` or `np.copy(x, subok=True)` instead.

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.

Contributors

- Matt Hostetter (@mhostetter)

3.32.7 v0.0.20

Released August 24, 2021

Breaking changes

- Move `poly_gcd()` functionality into `gcd()`.
- Move `poly_egcd()` functionality into `egcd()`.
- Move `poly_factors()` functionality into `factors()`.

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

Contributors

- Matt Hostetter (@mhostetter)

3.32.8 v0.0.21

Released September 2, 2021

Changes

- Fix docstrings and code completion for Python classes that weren't rendering correctly in an IDE.
- Various documentation improvements.

Contributors

- Matt Hostetter (@mhostetter)

3.32.9 v0.0.22

Released December 3, 2021

Breaking changes

- Random integer generation is handled using new style random generators. Now each `.Random()` call will generate a new seed rather than using the NumPy "global" seed used with `np.random.randint()`.
- Add a `seed=None` keyword argument to `FieldArray.Random()` and `Poly.Random()`. A reproducible script can be constructed like this:

```
rng = np.random.default_rng(123456789)
x = GF.Random(10, seed=rng)
y = GF.Random(10, seed=rng)
poly = galois.Poly.Random(5, seed=rng, field=GF)
```

Changes

- Official support for Python 3.9.
- Major performance improvements to "large" finite fields (those with `dtype=np.object_`).
- Minor performance improvements to all finite fields.
- Add the Number Theoretic Transform (NTT) in `ntt()` and `intt()`.
- Add the trace of finite field elements in `FieldArray.field_trace()`.
- Add the norm of finite field elements in `FieldArray.field_norm()`.
- Support `len()` on `Poly` objects, which returns the length of the coefficient array (polynomial order + 1).
- Support `x.dot(y)` syntax for the expression `np.dot(x, y)`.
- Minimum NumPy version bumped to 1.18.4 for new style random usage.
- Various bug fixes.

Contributors

- Iyán Méndez Veiga (@iyanmv)
- Matt Hostetter (@mhostetter)

3.32.10 v0.0.23

Released January 14, 2022

Changes

- Add support for Python 3.10.
- Add support for NumPy 1.21.
- Add support for Numba 0.55.
- Add type hints to library API.
- Add `FieldArray.characteristic_poly()` method to return the characteristic polynomial of a square matrix.
- Add `Poly.coefficients()` method to return the coefficient array with fixed size and order.
- Fix bug in `Poly.Degrees()` when duplicate degrees were present.
- Fix bug in Reed-Solomon decode when `c != 1`.
- Various other bug fixes.

Contributors

- Matt Hostetter (@mhostetter)

3.32.11 v0.0.24

Released February 12, 2022

Breaking changes

- Move `galois.minimal_poly()` functionality into `FieldArray.minimal_poly()`.
- Refactor `FieldArray.lup_decompose()` into `FieldArray.plu_decompose()`.
- Raise `ValueError` instead of returning `None` for `prev_prime(2)`.
- Return `(n, 1)` from `perfect_power(n)` if `n` is not a perfect power rather than returning `None`.

Changes

- Compute a finite field element's square root (if it exists) with `np.sqrt()`.
- Test if finite field elements have a square root with `FieldArray.is_quadratic_residue()`.
- List which finite field elements are/aren't quadratic residues (have a square root) with `FieldClass.quadratic_residues` and `FieldClass.quadratic_non_residues`.
- Compute standard vector spaces with `FieldArray.row_space()`, `FieldArray.column_space()`, `FieldArray.left_null_space()`, and `FieldArray.null_space()`.
- Compute a finite field element's additive and multiplicative orders with `FieldArray.additive_order()` and `FieldArray.multiplicative_order()`.
- Evaluate polynomials at square matrix inputs using `f(X, elementwise=False)`.
- Compute the characteristic polynomial of a single element or square matrix with `FieldArray.characteristic_poly()`.
- Compute the minimal polynomial of a single element with `FieldArray.minimal_poly()`.

- Compute a Lagrange interpolating polynomial with `lagrange_poly(x, y)`.
- Support non-square matrix inputs to `FieldArray.lu_decompose()` and `FieldArray.plu_decompose()`.
- Support polynomial scalar multiplication. Now `p * 3` is valid syntax and represents `p + p + p`.
- Allow polynomial comparison with integers and field scalars. Now `galois.Poly([0]) == 0` and `galois.Poly([0]) == GF(0)` return `True` rather than raising `TypeError`.
- Support testing 0-degree polynomials for irreducibility and primitivity.
- Extend `crt()` to work over non co-prime moduli.
- Extend `prev_prime()` and `next_prime()` to work over arbitrarily-large inputs.
- Allow negative integer inputs to `primes()`, `is_prime()`, `is_composite()`, `is_prime_power()`, `is_perfect_power()`, `is_square_free()`, `is_smooth()`, and `is_powersmooth()`.
- Fix various type hinting errors.
- Various other bug fixes.

Contributors

- Iyán Méndez Veiga (@iyanmv)
- Matt Hostetter (@mhostetter)

3.32.12 v0.0.25

Released March 21, 2022

Breaking changes

- Separated `LFSR` into `FLFSR`/`GLFSR` and fixed/redefined terms (feedback poly, characteristic poly, state). (#285)
- Removed `galois.pow()` and replaced it with the built-in `pow()`. (#300)

```
>>> f = galois.Poly([6, 3, 0, 1], field=galois.GF(7))
>>> g = galois.Poly([5, 0, 3], field=galois.GF(7))
>>> pow(f, 123456789, g)
Poly(6x + 2, GF(7))
```

- Removed `FieldClass.properties` and replaced with `FieldClass.__str__`. (#289)

```
>>> GF = galois.GF(3**5)
>>> print(GF)
Galois Field:
  name: GF(3^5)
  characteristic: 3
  degree: 5
  order: 243
  irreducible_poly: x^5 + 2x + 1
  is_primitive_poly: True
  primitive_element: x
```

- Differentiated `repr()` and `str()` for *Galois field arrays*, like NumPy. `repr()` displays the finite field's order, but `str()` does not.

```
>>> GF = galois.GF(31, display="power")
>>> x = GF([1, 23, 0, 15])
>>> x
GF([   1, ^27,    0, ^21], order=31)
>>> print(x)
[   1, ^27,    0, ^21]
```

- Renamed `Poly.String()` to `Poly.Str()`. Removed `Poly.string` and replaced it with `Poly.__str__`. (#300)

```
>>> f = galois.Poly.Str("x^3 + x + 1"); f
Poly(x^3 + x + 1, GF(2))
>>> str(f)
'x^3 + x + 1'
```

- Renamed `Poly.Integer()` to `Poly.Int()`. Removed `Poly.integer` and replaced it with `Poly.__int__`. (#300)

```
>>> f = galois.Poly.Int(11); f
Poly(x^3 + x + 1, GF(2))
>>> int(f)
11
```

Changes

- Fixed bug in Fibonacci/Galois LFSRs where feedback polynomial wasn't interpreted correctly for fields with characteristic greater than 2. (#299)
- Utilized memoization for expensive search routines (`irreducible_poly()` and `primitive_poly()`) to speed-up subsequent calls. (#295)

```
In [2]: %time galois.primitive_poly(7, 4)
CPU times: user 675 ms, sys: 6.24 ms, total: 682 ms
Wall time: 741 ms
Out[2]: Poly(x^4 + x^2 + 3x + 5, GF(7))

In [3]: %time galois.primitive_poly(7, 4)
CPU times: user 30 µs, sys: 0 ns, total: 30 µs
Wall time: 31.7 µs
Out[3]: Poly(x^4 + x^2 + 3x + 5, GF(7))
```

- Added support for `bin()`, `oct()`, and `hex()` on `Poly` objects. (#300)

```
>>> f = galois.Poly.Int(11); f
Poly(x^3 + x + 1, GF(2))
>>> bin(f)
'0b1011'
>>> oct(f)
'0o13'
>>> hex(f)
'0xb'
```

- Made *Galois field arrays* display with fixed-width elements, like NumPy. (#270)

- Achieved speed-up of `repr()` and `str()` on *Galois field arrays* of at least 25x. Achieved a much greater speed-up for large arrays, since now elements converted to `...` are no longer needlessly converted to their string representation. (#270)
- Overhauled documentation and website. Type hints are now displayed in the API reference. (#263)
- Various bug fixes.

Contributors

- Matt Hostetter (@mhostetter)

3.32.13 v0.0.26

Released March 30, 2022

Breaking changes

- Removed the `Poly.copy()` method as it was unnecessary. Polynomial objects are immutable. Use `g = f` wherever `g = f.copy()` was previously used. (#320)
- Disabled true division `f / g` on polynomials since true division was not actually being performed. Use floor division `f // g` moving forward. (#312)
- Refactored `irreducible_polys()` to return an iterator rather than list. Use `list(irreducible_polys(...))` to obtain the previous output. (#325)
- Refactored `primitive_polys()` to return an iterator rather than list. Use `list(primitive_polys(...))` to obtain the previous output. (#325)
- Refactored `primitive_root()` and `primitive_roots()`. (#325)
  - Added `method` keyword argument and removed `reverse` from `primitive_root()`. Use `primitive_root(..., method="max")` where `primitive_root(..., reverse=True)` was previously used.
  - Refactored `primitive_roots()` to now return an iterator rather than list. Use `list(primitive_roots(...))` to obtain the previous output.
- Refactored `primitive_element()` and `primitive_elements()`. (#325)
  - Added `method` keyword argument to `primitive_element()`.
  - Removed `start`, `stop`, and `reverse` arguments from both functions.
- Search functions now raise `RuntimeError` instead of returning `None` when failing to find an answer. This applies to `primitive_root()`, `pollard_p1()`, and `pollard_rho()`. (#312)

Changes

- The `galois.Poly` class no longer returns subclasses `BinaryPoly`, `DensePoly`, and `SparsePoly`. Instead, only `Poly` classes are returned. The classes otherwise operate the same. (#320)
- Made *Galois field array* creation much more efficient by avoiding redundant element verification. (#317)
  - Scalar creation is **625% faster**.

    ```
    In [2]: GF = galois.GF(3**5)

    # v0.0.25
    In [3]: %timeit GF(10)
    21.2 µs ± 181 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)

    # v0.0.26
    In [3]: %timeit GF(10)
    2.88 µs ± 8.03 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
    ```

  - Nested iterable array creation is **150% faster**.

    ```
    # v0.0.25
    In [4]: %timeit GF([[10, 20], [30, 40]])
    53.6 µs ± 436 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)

    # v0.0.26
    In [4]: %timeit GF([[10, 20], [30, 40]])
    20.9 µs ± 11.2 µs per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
    ```

  - Nested iterable (with strings) array creation is **25% faster**.

    ```
    # v0.0.25
    In [5]: %timeit GF([[10, "2x^2 + 2"], ["x^3 + x", 40]])
    67.9 µs ± 910 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)

    # v0.0.26
    In [5]: %timeit GF([[10, "2x^2 + 2"], ["x^3 + x", 40]])
    54.7 µs ± 16.3 µs per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
    ```

- Made array arithmetic **35% faster** by avoiding unnecessary element verification of outputs. (#309)

```
In [2]: GF = galois.GF(3**5)

In [3]: x = GF.Random((10_000), seed=1)

In [4]: y = GF.Random((10_000), seed=2)

# v0.0.25
In [6]: %timeit x * y
39.4 µs ± 237 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)

# v0.0.26
In [6]: %timeit x * y
28.8 µs ± 172 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

- Made polynomial arithmetic over binary fields **10,900% faster** by making polynomial creation from integers much more efficient. (#320)

```
In [5]: f
Out[5]: Poly(x^10 + x^9 + x^6 + x^5 + x^3 + x, GF(2))

In [6]: g
Out[6]: Poly(x^10 + x^7 + x^4 + 1, GF(2))

# v0.0.25
In [7]: %timeit f * g
283 µs ± 6.31 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)

# v0.0.26
In [7]: %timeit f * g
2.57 µs ± 54.4 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
```

- JIT-compiled polynomial modular exponentiation. (#306)
  - Binary fields are **14,225% faster**.

```
In [5]: f
Out[5]: Poly(x^10 + x^9 + x^6 + x^5 + x^3 + x, GF(2))

In [6]: g
Out[6]: Poly(x^5 + x^2, GF(2))

# v0.0.25
In [7]: %timeit pow(f, 123456789, g)
19.2 ms ± 206 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)

# v0.0.26
In [7]: %timeit pow(f, 123456789, g)
134 µs ± 2.21 µs per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

  - Other fields are **325% faster**.

```
In [6]: f
Out[6]: Poly(242x^10 + 216x^9 + 32x^8 + 114x^7 + 230x^6 + 179x^5 + 5x^4 + 124x^
↪3 + 96x^2 + 159x + 77, GF(3^5))

In [7]: g
Out[7]: Poly(183x^5 + 83x^4 + 101x^3 + 203x^2 + 68x + 2, GF(3^5))

# v0.0.25
In [8]: %timeit pow(f, 123456789, g)
17.6 ms ± 61.7 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)

# v0.0.26
In [8]: %timeit pow(f, 123456789, g)
4.19 ms ± 11.9 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

- Made irreducible and primitive polynomial search routines faster. (#306, #309, #317, #320)
  - Binary fields are **1,950% faster**.

```
# v0.0.25
In [2]: %time f = galois.primitive_poly(2, 14)
CPU times: user 296 ms, sys: 70.9 ms, total: 367 ms
Wall time: 313 ms

# v0.0.26
In [2]: %time f = galois.primitive_poly(2, 14)
CPU times: user 14.7 ms, sys: 5.53 ms, total: 20.2 ms
Wall time: 15.3 ms
```

  - Other fields are **400% faster**.

```
# v0.0.25
In [5]: %time f = galois.primitive_poly(7, 10)
CPU times: user 2.22 s, sys: 0 ns, total: 2.22 s
Wall time: 2.21 s

# v0.0.26
In [4]: %time f = galois.primitive_poly(7, 10)
CPU times: user 442 ms, sys: 0 ns, total: 442 ms
Wall time: 439 ms
```

- Made `FieldArray.Vector()` **100% faster** and `FieldArray.vector()` **25% faster** by making better use of `divmod()` when converting between integer and vector representations. (#322)

Contributors

- Matt Hostetter (@mhostetter)

3.32.14 v0.0.27

Released April 22, 2022

Breaking changes

- Sunsetted support for Python 3.6. This was necessary to support forward references with `from __future__ import annotations` (available in Python 3.7+). That import is required to support the type aliases in the new `galois.typing` subpackage. (#339)
- Removed the `FieldClass` metaclass from the public API. It was previously included due to an inability of Sphinx to document class properties. In this release, we monkey patched Sphinx to document all classmethods, class properties, and instance methods in `FieldArray` itself. (#343)
  - Use `issubclass(GF, galois.FieldArray)` anywhere `isinstance(GF, galois.FieldClass)` was previously used.
  - Annotate with `Type[galois.FieldArray]` anywhere `galois.FieldClass` was previously used.

Changes

- Added the `galois.typing` subpackage, similar to `np.typing`. It contains type hints for common coercible data types used throughout the library, including `ElementLike`, `ArrayLike`, and `PolyLike`. With these type hints, the annotations are simpler and more clear. (#339)
- Modified functions to accept coercible data types wherever possible. For example, functions now accept `PolyLike` objects instead of strictly `Poly` instances. (#339)
- Added `Array` which is an abstract base class of `FieldArray` (and `RingArray` in a future release). (#336)
- Added support for the DFT over any finite field using `np.fft.fft()` and `np.fft.ifft()`. (#335)

```
>>> x
GF([127, 191,  69,  35, 221, 242, 193, 108,  72, 102,  80, 163,  13,  74,
    218, 159, 207,  12, 159, 129,  92,  71], order=3^5)
>>> X = np.fft.fft(x); X
GF([ 16,  17,  20, 137,  58, 166, 178,  52,  19, 109, 115,  93,  99, 214,
    187, 235, 195,  96, 232,  45, 241,  24], order=3^5)
>>> np.fft.ifft(X)
GF([127, 191,  69,  35, 221, 242, 193, 108,  72, 102,  80, 163,  13,  74,
    218, 159, 207,  12, 159, 129,  92,  71], order=3^5)
```

- Implemented the Cooley-Tukey radix-2 $O(N\ \log(N))$ algorithm for the NTT and JIT compiled it. (#333)

```
In [2]: x = list(range(1, 1024 + 1))

# v0.0.26
In [4]: %timeit X = galois.ntt(x)
5.2 ms ± 121 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)

# v0.0.27
In [4]: %timeit X = galois.ntt(x)
695 µs ± 4.56 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

- Added the `FieldArray.primitive_root_of_unity()` classmethod. (#333)

```
>>> GF = galois.GF(3**5)
>>> GF.primitive_root_of_unity(22)
GF(39, order=3^5)
```

- Added the `FieldArray.primitive_roots_of_unity()` classmethod. (#333)

```
>>> GF = galois.GF(3**5)
>>> GF.primitive_roots_of_unity(22)
GF([ 14,  39,  44,  59, 109, 114, 136, 200, 206, 226], order=3^5)
```

- Made 0-th degree coefficients more differentiated when using the polynomial element representation. (#328)

```
# v0.0.26
>>> print(f)
(^2 +  + 1)x^4 + (^3)x + ^3 + 2^2 + 2 + 2
# v0.0.27
>>> print(f)
(^2 +  + 1)x^4 + (^3)x + (^3 + 2^2 + 2 + 2)
```

- Restructured code base for clarity. (#336)

- Fixed display of overloaded functions in API reference. (#337)
- Fixed broken "References" sections in API reference. (#281)
- Fixed other small bugs.

Contributors

- Matt Hostetter (@mhostetter)

3.32.15 v0.0.28

Released May 11, 2022

Changes

- Modified JIT-compiled functions to use explicit calculation *or* lookup tables. Previously, JIT functions only used explicit calculation routines. Now all ufuncs and functions are JIT-compiled once on first invocation, but use the current `ufunc_mode` to determine the arithmetic used. This provides a significant performance boost for fields which use lookup tables by default. The greatest performance improvement can be seen in $\mathrm{GF}(p^m)$ fields. (#354)
  - Polynomial multiplication is **210% faster**.

```
In [2]: GF = galois.GF(7**5)

In [3]: f = galois.Poly.Random(10, seed=1, field=GF)

In [4]: g = galois.Poly.Random(5, seed=2, field=GF)

# v0.0.27
In [6]: %timeit f * g
168 µs ± 722 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)

# v0.0.28
In [6]: %timeit f * g
54 µs ± 574 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

  - Polynomial modular exponentiation is **5,310% faster**.

```
# v0.0.27
In [8]: %timeit pow(f, 123456789, g)
5.9 ms ± 9.4 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)

# v0.0.28
In [8]: %timeit pow(f, 123456789, g)
109 µs ± 527 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

  - Matrix multiplication is **6,690% faster**.

```
In [9]: A = GF.Random((100, 100), seed=1)

In [10]: B = GF.Random((100, 100), seed=2)
```

(continues on next page)

(continued from previous page)

```
# v0.0.27
In [12]: %timeit A @ B
1.1 s ± 4.76 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

# v0.0.28
In [12]: %timeit A @ B
16.2 ms ± 50.1 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

- Simplified `FieldArray` subclasses' `repr()` and `str()`. Since these classes may be displayed in error logs, a concise representation is necessary. (#350)

```
>>> GF = galois.GF(3**5)
>>> GF
<class 'galois.GF(3^5)'>
```

- Added back `FieldArray.properties` for a detailed description of the finite field's relevant properties. (#350)

```
>>> GF = galois.GF(3**5)
>>> print(GF.properties)
Galois Field:
  name: GF(3^5)
  characteristic: 3
  degree: 5
  order: 243
  irreducible_poly: x^5 + 2x + 1
  is_primitive_poly: True
  primitive_element: x
```

- Increased code coverage.
- Various documentation fixes.

Contributors

- Matt Hostetter (@mhostetter)

3.32.16 v0.0.29

Released May 18, 2022

Breaking changes

- Moved `galois.square_free_factorization()` function into `Poly.square_free_factors()` method. (#362)
- Moved `galois.distinct_degree_factorization()` function into `Poly.distinct_degree_factors()` method. (#362)
- Moved `galois.equal_degree_factorization()` function into `Poly.equal_degree_factors()` method. (#362)

- Moved `galois.is_irreducible()` function into `Poly.is_irreducible()` method. This is a method, not property, to indicate it is a computationally-expensive operation. (#362)
- Moved `galois.is_primitive()` function into `Poly.is_primitive()` method. This is a method, not property, to indicate it is a computationally-expensive operation. (#362)
- Moved `galois.is_monic()` function into `Poly.is_monic` property. (#362)

Changes

- Added `galois.set_printoptions()` function to modify package-wide printing options. This is the equivalent of `np.set_printoptions()`. (#363)

```
In [1]: GF = galois.GF(3**5, display="poly")

In [2]: a = GF([109, 83]); a
Out[2]: GF([^4 + ^3 + 1,       ^4 + 2], order=3^5)

In [3]: f = galois.Poly([3, 0, 5, 2], field=galois.GF(7)); f
Out[3]: Poly(3x^3 + 5x + 2, GF(7))

In [4]: galois.set_printoptions(coeffs="asc")

In [5]: a
Out[5]: GF([1 + ^3 + ^4,       2 + ^4], order=3^5)

In [6]: f
Out[6]: Poly(2 + 5x + 3x^3, GF(7))
```

- Added `galois.get_printoptions()` function to return the current package-wide printing options. This is the equivalent of `np.get_printoptions()`. (#363)
- Added `galois.printoptions()` context manager to modify printing options inside of a `with` statement. This is the equivalent of `np.printoptions()`. (#363)
- Added a separate `Poly.factors()` method, in addition to the polymorphic `galois.factors()`. (#362)
- Added a separate `Poly.is_square_free()` method, in addition to the polymorphic `galois.is_square_free()`. This is a method, not property, to indicate it is a computationally-expensive operation. (#362)
- Fixed a bug (believed to be introduced in v0.0.26) where `Poly.degree` occasionally returned `np.int64` instead of `int`. This could cause overflow in certain large integer operations (e.g., computing q^m when determining if a degree-m polynomial over $\mathrm{GF}(q)$ is irreducible). When the integer overflowed, this created erroneous results. (#360, #361)
- Increased code coverage.

Contributors

- Matt Hostetter (@mhostetter)

3.32.17 v0.0.30

Released July 12, 2022

Changes

- Added support for NumPy 1.22 with Numba 0.55.2. This allows users to upgrade NumPy and avoid recently-discovered vulnerabilities CVE-2021-34141, CVE-2021-41496, and CVE-2021-41495. (#366)
- Made `FieldArray.repr_table()` more compact. (#367)

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

In [3]: print(GF.repr_table())
 Power      Polynomial       Vector     Integer
------- --------------- ----------- ---------
   0           0          [0, 0, 0]      0
  x^0          1          [0, 0, 1]      1
  x^1          x          [0, 1, 0]      3
  x^2         x^2         [1, 0, 0]      9
  x^3        x + 2        [0, 1, 2]      5
  x^4      x^2 + 2x       [1, 2, 0]      15
  x^5    2x^2 + x + 2     [2, 1, 2]      23
  x^6    x^2 + x + 1      [1, 1, 1]      13
  x^7    x^2 + 2x + 2     [1, 2, 2]      17
  x^8      2x^2 + 2       [2, 0, 2]      20
  x^9        x + 1        [0, 1, 1]      4
  x^10     x^2 + x        [1, 1, 0]      12
  x^11   x^2 + x + 2      [1, 1, 2]      14
  x^12     x^2 + 2        [1, 0, 2]      11
  x^13         2          [0, 0, 2]      2
  x^14        2x          [0, 2, 0]      6
  x^15       2x^2         [2, 0, 0]      18
  x^16      2x + 1        [0, 2, 1]      7
  x^17     2x^2 + x       [2, 1, 0]      21
  x^18   x^2 + 2x + 1     [1, 2, 1]      16
  x^19  2x^2 + 2x + 2     [2, 2, 2]      26
  x^20   2x^2 + x + 1     [2, 1, 1]      22
  x^21     x^2 + 1        [1, 0, 1]      10
  x^22      2x + 2        [0, 2, 2]      8
  x^23    2x^2 + 2x       [2, 2, 0]      24
  x^24  2x^2 + 2x + 1     [2, 2, 1]      25
  x^25     2x^2 + 1       [2, 0, 1]      19
```

- Made `FieldArray.arithmetic_table()` more compact. (#367)

```
In [2]: GF = galois.GF(13)
```

(continues on next page)

(continued from previous page)

```
In [3]: print(GF.arithmetic_table("*"))
x * y |  0   1   2   3   4   5   6   7   8   9  10  11  12
------|----------------------------------------------------
    0 |  0   0   0   0   0   0   0   0   0   0   0   0   0
    1 |  0   1   2   3   4   5   6   7   8   9  10  11  12
    2 |  0   2   4   6   8  10  12   1   3   5   7   9  11
    3 |  0   3   6   9  12   2   5   8  11   1   4   7  10
    4 |  0   4   8  12   3   7  11   2   6  10   1   5   9
    5 |  0   5  10   2   7  12   4   9   1   6  11   3   8
    6 |  0   6  12   5  11   4  10   3   9   2   8   1   7
    7 |  0   7   1   8   2   9   3  10   4  11   5  12   6
    8 |  0   8   3  11   6   1   9   4  12   7   2  10   5
    9 |  0   9   5   1  10   6   2  11   7   3  12   8   4
   10 |  0  10   7   4   1  11   8   5   2  12   9   6   3
   11 |  0  11   9   7   5   3   1  12  10   8   6   4   2
   12 |  0  12  11  10   9   8   7   6   5   4   3   2   1
```

Contributors

- Iyán Méndez Veiga (@iyanmv)
- Matt Hostetter (@mhostetter)

3.32.18 v0.0.31

Released July 24, 2022

Breaking changes

- Renamed `FieldArray.Elements()` classmethod to `FieldArray.elements` class property. This naming convention is more consistent with `primitive_elements`, `units`, `quadratic_residues`, and `quadratic_non_residues`. (#373)

```
>>> GF = galois.GF(3**2, display="poly")
>>> GF.elements
GF([      0,       1,       2,       ,   + 1,   + 2,      2, 2 + 1,
    2 + 2], order=3^2)
```

- Renamed `BCH.systematic` to `BCH.is_systematic`. (#376)
- Renamed `ReedSolomon.systematic` to `ReedSolomon.is_systematic`. (#376)

Changes

- Added support for polynomial composition in `Poly.__call__()`. (#377)

```
>>> GF = galois.GF(3**5)
>>> f = galois.Poly([37, 123, 0, 201], field=GF); f
Poly(37x^3 + 123x^2 + 201, GF(3^5))
>>> g = galois.Poly([55, 0, 1], field=GF); g
Poly(55x^2 + 1, GF(3^5))
>>> f(g)
Poly(77x^6 + 5x^4 + 104x^2 + 1, GF(3^5))
```

- Added `FieldArray.units` class property. (#373)

```
>>> GF = galois.GF(3**2, display="poly")
>>> GF.units
GF([     1,      2,       ,    + 1,    + 2,      2, 2 + 1, 2 + 2],
   order=3^2)
```

Documentation

- Reworked API reference using Sphinx Immaterial's `python-apigen`. (#370)
- Shortened website URLs to use directories. https://galois.readthedocs.io/en/v0.0.30/getting-started.html is converted to https://galois.readthedocs.io/en/v0.0.31/getting-started/. (#370)

Contributors

- Matt Hostetter (@mhostetter)

3.32.19 v0.0.32

Released July 28, 2022

Breaking changes

- Changed "quadratic residue" language in Galois fields to "square". This seems to be more canonical. Quadratic residue connotes quadratic residue modulo p, which is a square in $\mathrm{GF}(p)$. However, a quadratic residue modulo p^m is *not* a square in $\mathrm{GF}(p^m)$. Hopefully the "square" language is more clear. (#392)
  - Renamed `FieldArray.is_quadratic_residue` to `FieldArray.is_square`.
  - Renamed `FieldArray.quadratic_residues` to `FieldArray.squares`.
  - Renamed `FieldArray.quadratic_non_residues` to `FieldArray.non_squares`.

Changes

- Added support for Numba 0.56.x. (#389)
- Added general logarithm base any primitive element in `FieldArray.log()`. (#385)

```
>>> GF = galois.GF(3**5)
>>> x = GF.Random(10, low=1); x
GF([215, 176,  52,  20, 236,  48, 217, 131,  13,  57], order=3^5)
>>> beta = GF.primitive_elements[-1]; beta
GF(242, order=3^5)
>>> i = x.log(beta); i
array([171, 240, 109,  65, 162,  57,  34, 166,  72,  56])
>>> np.array_equal(beta ** i, x)
True
```

- Added Pollard-ρ discrete logarithm for certain $\mathrm{GF}(2^m)$ fields. The algorithm is only applicable to fields whose multiplicative group has prime order. It has complexity $O(\sqrt{n})$ compared to $O(n)$ for the brute-force algorithm. In this example, Pollard-ρ is **1650%** faster than brute force. (#385)

```
In [3]: GF = galois.GF(2**19, compile="jit-calculate")

In [4]: galois.is_prime(GF.order - 1)
Out[4]: True

In [5]: x = GF.Random(100, low=1, seed=1)

# v0.0.31
In [6]: %timeit np.log(x)
80.3 ms ± 55.8 µs per loop (mean ± std. dev. of 7 runs, 1 loop each)

# v0.0.32
In [6]: %timeit np.log(x)
4.59 ms ± 90.5 µs per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

- Added Pohlig-Hellman discrete logarithm to replace the brute-force search. It has complexity $O(\sum_{i=1}^{r} e_i(\lg n + \sqrt{p_i}))$ compared to $O(n)$ for the brute-force algorithm. It is especially efficient for fields whose multiplicative group has smooth order. In this example with $p^m - 1$ smooth, Pohlig-Hellman is **~3,000,000%** faster than brute force. (#387)

```
In [3]: GF = galois.GF(491954233)

# The multiplicative group's order is smooth
In [4]: galois.factors(GF.order - 1)
Out[4]: ([2, 3, 7, 11, 19, 14011], [3, 1, 1, 1, 1, 1])

In [5]: x = GF.Random(1, low=1, seed=1)

# v0.0.31
In [6]: %timeit np.log(x)
1.82 s ± 2.95 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

# v0.0.32
In [6]: %timeit np.log(x)
61.3 µs ± 14.6 µs per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

- Added Itoh-Tsujii inversion algorithm for extension fields, which is **35%** faster than inversion with Fermat's Little Theorem. (#383)

```
In [3]: GF = galois.GF(109987**4)

In [4]: x = GF.Random(100, low=1, seed=1)

# v0.0.31
In [5]: %timeit np.reciprocal(x)
646 ms ± 834 µs per loop (mean ± std. dev. of 7 runs, 1 loop each)

# v0.0.32
In [5]: %timeit np.reciprocal(x)
479 ms ± 26.9 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

- Fixed a bug where `FieldArray` subclasses and instances could not be pickled. (#393)

Contributors

- Matt Hostetter (@mhostetter)

3.32.20 v0.0.33

Released August 26, 2022

Breaking changes

- Modified FEC `encode()`, `detect()`, and `decode()` methods to always return `FieldArray` instances, not `np.ndarray`. (#397)
  - Invoke `.view(np.ndarray)` on the output to convert it back to a NumPy array, if needed.

Changes

- Added support for `ArrayLike` inputs to FEC `encode()`, `detect()`, and `decode()` methods. (#397)
- Modified library packaging to use `pyproject.toml` and a `src/` source code folder. (#404)

Contributors

- Matt Hostetter (@mhostetter)

3.33 Index

PYTHON MODULE INDEX

g

INDEX