CONTENTS

1 Installation 3
   1.1 Install with pip 3
   1.2 Install for development 3
   1.3 Install for development with min dependencies 3

2 Versioning 5

3 Basic Usage 7
   3.1 Galois field arrays 7
   3.2 Polynomials over Galois fields 12
   3.3 Forward error correction codes 13

4 Tutorials 15
   4.1 Intro to Galois Fields: Prime Fields 15
   4.2 Intro to Galois Fields: Extension Fields 21
   4.3 Constructing Galois field array classes 33
   4.4 Array creation 35
   4.5 Galois field array arithmetic 38
   4.6 Extremely large fields 41

5 Performance Testing 45
   5.1 Performance compared with native NumPy 45
   5.2 Benchmarking 48

6 Development 59
   6.1 Lint the package 59
   6.2 Run the unit tests 59
   6.3 Build the documentation 60

7 API Reference v0.0.21 61
   7.1 Galois Fields 61
   7.2 Polynomials over Galois Fields 131
   7.3 Forward Error Correcting Codes 166
   7.4 Linear Sequences 192
   7.5 Number Theory 199
   7.6 Integer Factorization 210
   7.7 Primes 219
   7.8 Numpy Examples 227

8 Acknowledgements 249
9 Citation

10 Release Notes

10.1 v0.0.21 ............................................................. 253
10.2 v0.0.20 ............................................................. 253
10.3 v0.0.19 ............................................................. 254
10.4 v0.0.18 ............................................................. 254
10.5 v0.0.17 ............................................................. 256
10.6 v0.0.16 ............................................................. 257
10.7 v0.0.15 ............................................................. 257
10.8 v0.0.14 ............................................................. 258

11 Indices and tables

Python Module Index ................................................ 261
Index ........................................................................ 263
• Supports all Galois fields GF($p^m$), even arbitrarily-large fields!
• **Faster** than native NumPy! GF(x) * GF(y) is faster than (x * y) % p for GF(p)
• Seamless integration with NumPy – normal NumPy functions work on Galois field arrays
• Linear algebra on Galois field matrices using normal np.linalg functions
• Functions to generate irreducible, primitive, and Conway polynomials
• Polynomials over Galois fields with `galois.Poly`
• Forward error correction codes with `galois.BCH` and `galois.ReedSolomon`
• Fibonacci and Galois linear feedback shift registers with `galois.LFSR`, both binary and p-ary
• Various number theoretic functions
• Integer factorization and accompanying algorithms
• Prime number generation and primality testing
1.1 Install with pip

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

$ python3 -m pip install galois

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

1.2 Install for development

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

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

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

1.3 Install for development with min dependencies

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

Listing 1: requirements-min.txt

1. numpy==1.17.3
2. numba==0.53

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

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

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

(continues on next page)
Then, installing the galois package won't upgrade the dependencies since their versions are satisfactory.

```
$ python3 -m pip install -e galois
```
This project uses semantic versioning. Releases are versioned `major.minor.patch`. Major releases introduce API-changing features. Minor releases add features and are backwards compatible with other releases in `major.x.x`. Patch releases fix bugs in a minor release and are backwards compatible with other releases in `major.minor.x`.

Releases before `1.0.0` are alpha and beta releases. Alpha releases are `0.0.alpha`. There is no API compatibility guarantee for them. They can be thought of as `0.0.alpha-major`. Beta releases are `0.beta.x` and are API compatible. They can be thought of as `0.beta-major.beta-minor`. 
The main idea of the \texttt{galois} package can be summarized as follows. The user creates a “Galois field array class” using \texttt{GF = galois.GF(p**m)}. A Galois field array class GF is a subclass of \texttt{numpy.ndarray} and its constructor \texttt{x = GF(array_like)} mimics the call signature of \texttt{numpy.array()}. A Galois field array \texttt{x} is operated on like any other NumPy array, but all arithmetic is performed in \texttt{GF(p^m)} not \texttt{Z} or \texttt{R}.

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

\section{Galois field arrays}

\subsection{Class construction}

Galois field array classes are created using the \texttt{galois.GF()} class factory function.

\begin{verbatim}
In [1]: import numpy as np
In [2]: import galois
In [3]: GF256 = galois.GF(2**8)
In [4]: print(GF256)
<class 'numpy.ndarray over GF(2^8)'>
\end{verbatim}

These classes are subclasses of \texttt{galois.FieldArray} (which itself subclasses \texttt{numpy.ndarray}) and have \texttt{galois.FieldClass} as their metaclass.

\begin{verbatim}
In [5]: isinstance(GF256, galois.FieldClass)
Out[5]: True
In [6]: issubclass(GF256, galois.FieldArray)
Out[6]: True
In [7]: issubclass(GF256, np.ndarray)
Out[7]: True
\end{verbatim}

A Galois field array class contains attributes relating to its Galois field and methods to modify how the field is calculated or displayed. See the attributes and methods in \texttt{galois.FieldClass}. 

\texttt{galois} package content is an example of a table that provides a summary of the relationship between NumPy and Galois field arithmetic. This table highlights how the galois package extends NumPy's functionality to include Galois field operations, making it easier for users to work with finite fields in their computations.
# Summarizes some properties of the Galois field

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

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

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

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

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

## 3.1.2 Array creation

Galois field arrays can be created from existing NumPy arrays.

In [12]: array = np.random.randint(0, GF256.order, 10, dtype=int); array
Out[12]: array([ 91, 203, 196, 251, 118, 153, 41, 28, 133, 152])

In [13]: GF256(array)
Out[13]: GF([ 91, 203, 196, 251, 118, 153, 41, 28, 133, 152], order=2^8)

# Or view an existing numpy array as a Galois field array -- no copy is performed
In [14]: array.view(GF256)
Out[14]: GF([ 91, 203, 196, 251, 118, 153, 41, 28, 133, 152], order=2^8)

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

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

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

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

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


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

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

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

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

# Temporarily set the display mode to represent GF(2^8) field elements as powers of the primitive element
In [21]: with GF256.display("power"):
   ....:   print(a)
   ....:
GF([191, 0, 1, 194], order=2^8)

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

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

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

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

```
In [23]: x = GF256.Random((2,5)); x
Out[23]:
GF([[155, 190, 192, 29, 9],
     [206, 231, 185, 155, 57]], order=2^8)

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

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

3.1.4 Linear algebra

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

- \text{np.dot()}, \text{np.vdot()}, \text{np.inner()}, \text{np.outer()}, \text{np.matmul()}, \text{np.linalg.matrix_power()}
- \text{np.linalg.det()}, \text{np.linalg.matrix_rank()}, \text{np.trace()}
- \text{np.linalg.solve()}, \text{np.linalg.inv()}

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

(continues on next page)
# Ensure A is invertible

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

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

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

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

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

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

### 3.1.5 NumPy ufunc methods

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

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

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

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

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

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

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

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

Polynomials can be created by specifying the polynomial coefficients as either a `galois.FieldArray` or an “array-like” object with the `field` keyword argument.

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

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

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

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

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

```python
In [40]: with GF256.display("poly"):  
   ....:     print(p)
   ....: Poly((^7 + ^6 + ^5 + 1)x^3 + (^4 + 1)x + (^5 + 1), GF(2^8))
```

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


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

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

Polynomial arithmetic is performed using Python operators.

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

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

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

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

(continues on next page)
Polynomials over Galois fields can be evaluated at scalars or arrays of field elements.

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

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

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

# Evaluate the polynomial at an array of values
In [51]: p(x)
Out[51]: GF([[161, 67, 64, 90, 108],
          [141, 129, 242,  1, 63]], order=2^8)
```

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

```
# Notice the irreducible polynomial is over GF(2), not GF(2^8)
In [52]: p = GF256.irreducible_poly; p
Out[52]: Poly(x^8 + x^4 + x^3 + x^2 + 1, GF(2))

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

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

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

### 3.3 Forward error correction codes

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

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

In [57]: import galois
```
A message can be either a 1-D vector or a 2-D matrix of messages. Shortened codes are also supported. See the docs for more details.

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

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

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

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

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

In [64]: bch.decode(C)
Out[64]: GF([[1, 0, 1, 1, 0, 0, 0],
            [1, 0, 0, 0, 0, 1, 1]], order=2)
```
4.1 Intro to Galois Fields: Prime Fields

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

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

4.1.1 Elements

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

Using the \textit{galois} package, a Galois field array class is created using the class factory \textit{galois.GF}.

\begin{verbatim}
In [1]: GF7 = galois.GF(7); GF7
Out[1]: <class 'numpy.ndarray over GF(7)'>

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

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

\begin{verbatim}
In [3]: GF7.Elements()
Out[3]: GF([0, 1, 2, 3, 4, 5, 6], order=7)
\end{verbatim}

This array should be read as “a Galois field array \([0, 1, 2, 3, 4, 5, 6]\) over the finite field with order 7”.
### 4.1.2 Arithmetic mod p

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

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

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

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

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

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

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

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

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

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

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

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

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

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>
Division in GF(\(p\)) is a little more difficult. Division can’t be as simple as taking \(x/y \pmod{p}\) because many integer divisions do not result in integers. The division of \(x/y = z\) can be reformulated as the question “what \(z\) multiplied by \(y\) results in \(x\)?”. This is an equivalent problem to “what \(z\) multiplied by \(y\) results in 1?”, where \(z\) is the multiplicative inverse of \(y\).

To find the multiplicative inverse of \(y\), one can simply perform trial multiplication until the result of 1 is found. For instance, suppose \(y = 4\) in GF(7). We can multiply 4 by every element in the field until the product is 1 and we’ll find that \(4^{-1} = 2\) in GF(7), namely \(2 \times 4 = 1\) in GF(7).
In [15]: y = GF7(4); y
Out[15]: GF(4, order=7)

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

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

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

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

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

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

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

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

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

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

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

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

<table>
<thead>
<tr>
<th>x / y</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>4</td>
<td>5</td>
<td>2</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>5</td>
<td>1</td>
<td>6</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>2</td>
<td>6</td>
<td>1</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>6</td>
<td>4</td>
<td>3</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>3</td>
<td>2</td>
<td>5</td>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>

(continues on next page)
4.1.3 Primitive elements

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

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

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

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

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

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

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

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

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

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

<table>
<thead>
<tr>
<th>Power</th>
<th>Polynomial</th>
<th>Vector</th>
<th>Integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>[0]</td>
<td>0</td>
</tr>
<tr>
<td>(3^0)</td>
<td>1</td>
<td>[1]</td>
<td>1</td>
</tr>
<tr>
<td>(3^1)</td>
<td>3</td>
<td>[3]</td>
<td>3</td>
</tr>
<tr>
<td>(3^2)</td>
<td>2</td>
<td>[2]</td>
<td>2</td>
</tr>
<tr>
<td>(3^3)</td>
<td>6</td>
<td>[6]</td>
<td>6</td>
</tr>
<tr>
<td>(3^4)</td>
<td>4</td>
<td>[4]</td>
<td>4</td>
</tr>
</tbody>
</table>

(continues on next page)
There are multiple primitive elements of a given field. In the case of GF(7), 3 and 5 are primitive elements.

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

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

<table>
<thead>
<tr>
<th>Power</th>
<th>Polynomial</th>
<th>Vector</th>
<th>Integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>0^5</td>
<td>5</td>
<td>[5]</td>
<td>5</td>
</tr>
<tr>
<td>5^0</td>
<td>1</td>
<td>[1]</td>
<td>1</td>
</tr>
<tr>
<td>5^1</td>
<td>5</td>
<td>[5]</td>
<td>5</td>
</tr>
<tr>
<td>5^2</td>
<td>4</td>
<td>[4]</td>
<td>4</td>
</tr>
<tr>
<td>5^3</td>
<td>6</td>
<td>[6]</td>
<td>6</td>
</tr>
<tr>
<td>5^4</td>
<td>2</td>
<td>[2]</td>
<td>2</td>
</tr>
<tr>
<td>5^5</td>
<td>3</td>
<td>[3]</td>
<td>3</td>
</tr>
</tbody>
</table>
```

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

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

<table>
<thead>
<tr>
<th>Power</th>
<th>Polynomial</th>
<th>Vector</th>
<th>Integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>0^5</td>
<td>5</td>
<td>[5]</td>
<td>5</td>
</tr>
<tr>
<td>2^0</td>
<td>1</td>
<td>[1]</td>
<td>1</td>
</tr>
<tr>
<td>2^1</td>
<td>2</td>
<td>[2]</td>
<td>2</td>
</tr>
<tr>
<td>2^2</td>
<td>4</td>
<td>[4]</td>
<td>4</td>
</tr>
<tr>
<td>2^3</td>
<td>1</td>
<td>[1]</td>
<td>1</td>
</tr>
<tr>
<td>2^4</td>
<td>2</td>
<td>[2]</td>
<td>2</td>
</tr>
<tr>
<td>2^5</td>
<td>4</td>
<td>[4]</td>
<td>4</td>
</tr>
</tbody>
</table>
```
4.2 Intro to Galois Fields: Extension Fields

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

4.2.1 Elements

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

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

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

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

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

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

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

```plaintext
In [4]: print(GF9.repr_table())
    Power | Polynomial | Vector | Integer
    0     | 0          | [0, 0] | 0
    x^0   | 1          | [0, 1] | 1
    x^1   | x          | [1, 0] | 3
    x^2   | x + 1      | [1, 1] | 4
    x^3   | 2x + 1     | [2, 1] | 7
    x^4   | 2          | [0, 2] | 2
```

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

With \textit{galois}, we can represent a single Galois field element using \texttt{GF9(int)} or \texttt{GF9(string)}.

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

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

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

In addition to scalars, these conversions work for arrays.

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

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

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

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

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

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

# Permanently set the display mode to the polynomial representation
In [13]: GF9.display("poly"); x
Out[13]: GF([0, 1, 2, \text{^1}, \text{^2}, \text{^2}, \text{^2 + 1}, \text{^2 + 2}], order=3^2)
```
# Reset the display mode to the integer representation
In [14]: GF9.display(); x
Out[14]: GF([0, 1, 2, 3, 4, 5, 6, 7, 8], order=3^2)

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

## 4.2.2 Arithmetic mod \( p(x) \)

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

When constructing an extension field, if an explicit irreducible polynomial is not specified, a default is chosen. The default polynomial is a Conway polynomial which is irreducible and primitive, see \texttt{galois.conway\_poly()} for more information.

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

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

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

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

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

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

In [21]: a.integer

(continues on next page)
Out[21]: 5

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

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

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

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

We can do the equivalent calculation directly in the field GF(3^2).

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

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

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

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

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

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

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

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

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

As before, let’s compute this polynomial product explicitly first.
In [35]: b = galois.Poly([1, 1], field=GF3); b
Out[35]: Poly(x + 1, GF(3))

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

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

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

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

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

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

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

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

Now the entire multiplication table can be shown for completeness.

In [43]: with GF9.display("poly"):
    ....:   print(GF9.arithmetic_table("*"))
    ....:
    x * y 0 1 2 | +1 | +2 | 2 | 2+1 | 2+2
0 0 0 0 0 0 0 0 0 0
1 0 1 2 | +1 | +2 | 2 | 2+1 | 2+2
2 0 2 1 | 2 | 2+2 | 2+1 | 2+2 | 2+2 | +1
0 | 2 | +1 | 2+1 | 1 | 2+2 | 2 | +2
+1 0 | +1 | 2+2 | 2+1 | 2 | +2 | 2 | 1
+2 0 | +2 | 2+1 | 1 | 2+2 | 2 | +1 | 2
2 0 | 2 | 2+2 | +2 | 2 | +1 | 1 | 2+1
2+1 0 | 2+1 | +2 | 2 | 2 | +1 | 1 | 2+2
(continues on next page)
Division, as in $\text{GF}(p)$, is a little more difficult. Fortunately the Extended Euclidean Algorithm, which was used in prime fields on integers, can be used for extension fields on polynomials. Given two polynomials $a$ and $b$, the Extended Euclidean Algorithm finds the polynomials $x$ and $y$ such that $xa + yb = \gcd(a, b)$. This algorithm is implemented in `galois.egcd()`.

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

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

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

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

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

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

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

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

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

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

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

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

```python
In [52]: with GF9.display("poly"):
     ....:     print(GF9.arithmetic_table("/"))
     ....:
x / y  1  2  |  +1  |  +2  |  2  |  2+1  |  2+2
```
4.2.3 Primitive elements

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

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

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

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

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

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

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

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

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

<table>
<thead>
<tr>
<th>Power</th>
<th>Polynomial</th>
<th>Vector</th>
<th>Integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>[0, 0]</td>
<td>0</td>
</tr>
<tr>
<td>$x^0$</td>
<td>1</td>
<td>[0, 1]</td>
<td>1</td>
</tr>
</tbody>
</table>

(continues on next page)
And here is the representation table using a different generator $g = 2x + 1$.

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

<table>
<thead>
<tr>
<th>Power</th>
<th>Polynomial</th>
<th>Vector</th>
<th>Integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>[0, 0]</td>
<td>0</td>
</tr>
<tr>
<td>(2x + 1)^0</td>
<td>1</td>
<td>[0, 1]</td>
<td>1</td>
</tr>
<tr>
<td>(2x + 1)^1</td>
<td>2x + 1</td>
<td>[2, 1]</td>
<td>7</td>
</tr>
<tr>
<td>(2x + 1)^2</td>
<td>2x + 2</td>
<td>[2, 2]</td>
<td>8</td>
</tr>
<tr>
<td>(2x + 1)^3</td>
<td>x</td>
<td>[1, 0]</td>
<td>3</td>
</tr>
<tr>
<td>(2x + 1)^4</td>
<td>2</td>
<td>[0, 2]</td>
<td>2</td>
</tr>
<tr>
<td>(2x + 1)^5</td>
<td>x + 2</td>
<td>[1, 2]</td>
<td>5</td>
</tr>
<tr>
<td>(2x + 1)^6</td>
<td>x + 1</td>
<td>[1, 1]</td>
<td>4</td>
</tr>
<tr>
<td>(2x + 1)^7</td>
<td>2x</td>
<td>[2, 0]</td>
<td>6</td>
</tr>
</tbody>
</table>
```

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

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

<table>
<thead>
<tr>
<th>Power</th>
<th>Polynomial</th>
<th>Vector</th>
<th>Integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>[0, 0]</td>
<td>0</td>
</tr>
<tr>
<td>(x + 1)^0</td>
<td>1</td>
<td>[0, 1]</td>
<td>1</td>
</tr>
<tr>
<td>(x + 1)^1</td>
<td>x + 1</td>
<td>[1, 1]</td>
<td>4</td>
</tr>
</tbody>
</table>
```

(continues on next page)
4.2.4 Primitive polynomials

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

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

\begin{center}
\begin{tabular}{|c|c|c|c|}
\hline
\((x + 1)^2\) & 2 & \([0, 2]\) & 2 \\
\hline
\((x + 1)^3\) & \(2x + 2\) & \([2, 2]\) & 8 \\
\hline
\((x + 1)^4\) & 1 & \([0, 1]\) & 1 \\
\hline
\((x + 1)^5\) & \(x + 1\) & \([1, 1]\) & 4 \\
\hline
\((x + 1)^6\) & 2 & \([0, 2]\) & 2 \\
\hline
\((x + 1)^7\) & \(2x + 2\) & \([2, 2]\) & 8 \\
\hline
\end{tabular}
\end{center}

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

\begin{center}
\begin{Verbatim}
In [60]: GF16 = galois.GF(2**4)
In [61]: print(GF16.properties)
GF(2^4):
    characteristic: 2
    degree: 4
    order: 16
    irreducible_poly: x^4 + x + 1
    is_primitive_poly: True
    primitive_element: x
\end{Verbatim}
\end{center}

\begin{center}
\begin{Verbatim}
In [62]: p = GF16.irreducible_poly; p
Out[62]: Poly(x^4 + x + 1, GF(2))
In [63]: galois.is_irreducible(p)
Out[63]: True
In [64]: galois.is_primitive(p)
Out[64]: True

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

Since the irreducible polynomial is primitive, we write the field elements in polynomial basis with indeterminate \(\alpha\) instead of \(x\), where \(\alpha\) represents the primitive element of \(\text{GF}(p^m)\). For powers of \(\alpha\) less than 4, it can be seen that \(\alpha = x\), \(\alpha^2 = x^2\), and \(\alpha^3 = x^3\).
Extension fields do not need to be constructed from primitive polynomials, however. The polynomial \( p(x) = x^4 + x^3 + x^2 + x + 1 \) is irreducible, but not primitive. This polynomial can define arithmetic in \( GF(2^4) \). The two fields (the first defined by a primitive polynomial and the second defined by a non-primitive polynomial) are isomorphic to one another.
Notice the primitive element of $\mathbb{GF}(2^4)$ with irreducible polynomial $p(x) = x^4 + x^3 + x^2 + x + 1$ does not have $x + 1$ as root in $\mathbb{GF}(2^4)$.

# Evaluate the irreducible polynomial over $\mathbb{GF}(2^4)$ at the primitive element

In [73]: p(GF16_v2.primitive_element, field=GF16_v2)
Out[73]: GF(6, order=2^4)

As can be seen in the representation table, for powers of $\alpha$ less than 4, $\alpha \not= x$, $\alpha^2 \not= x^2$, and $\alpha^3 \not= x^3$. Therefore the polynomial indeterminate used is $x$ to distinguish it from $\alpha$, the primitive element.
4.3 Constructing Galois field array classes

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

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

The Galois field `GF(2)` is already constructed in `galois`. It can be accessed by `galois.GF2`.

```
In [1]: GF2 = galois.GF2
In [2]: print(GF2)
<class 'numpy.ndarray over GF(2)'>
In [3]: issubclass(GF2, np.ndarray)
Out[3]: True
In [4]: issubclass(GF2, galois.FieldArray)
Out[4]: True
In [5]: issubclass(type(GF2), galois.FieldClass)
Out[5]: True
In [6]: print(GF2.properties)
GF(2):
    characteristic: 2
    degree: 1
    order: 2
    irreducible_poly: x + 1
    is_primitive_poly: True
    primitive_element: 1
```

GF($2^m$) fields, where $m$ is a positive integer, can be constructed using the class factory `galois.GF()`.

```
In [7]: GF8 = galois.GF(2**3)
In [8]: print(GF8)
<class 'numpy.ndarray over GF(2^3)'>
```

| $(x + 1)^{11}$ | $x^3 + x + 1$ | $[1, 0, 1, 1]$ | 11 |
| $(x + 1)^{12}$ | $x$ | $[0, 0, 1, 0]$ | 2 |
| $(x + 1)^{13}$ | $x^2 + x$ | $[0, 1, 1, 0]$ | 6 |
| $(x + 1)^{14}$ | $x^3 + x$ | $[1, 0, 1, 0]$ | 10 |
In [9]: `issubclass(GF8, np.ndarray)`
Out[9]: True

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

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

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

GF(p) fields, where p is prime, can be constructed using the class factory `galois.GF()`.

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

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

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

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

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

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

4.4.1 Explicit construction

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

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

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

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

4.4.2 View casting

```
# View cast an existing array to a Galois field array (no copy operation)
In [4]: y = x_np.view(GF7); y
Out[4]: GF([3, 0, 2, 2, 0, 1, 3, 0, 6, 5], order=7)
```

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

```
In [5]: x_np
Out[5]: array([3, 0, 2, 2, 0, 1, 3, 0, 6, 5])

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

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

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


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

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

In [11]: GF256.Elements()
Out[11]:
GF([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13,
     14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27,
     28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41,
     42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55,
     56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69,
     70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83,
     84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97,
     98, 99,100,101,102,103,104,105,106,107,108,109,110,111,
     112,113,114,115,116,117,118,119,120,121,122,123,124,125,
     126,127,128,129,130,131,132,133,134,135,136,137,138,139,
     140,141,142,143,144,145,146,147,148,149,150,151,152,153,
     154,155,156,157,158,159,160,161,162,163,164,165,166,167,
     168,169,170,171,172,173,174,175,176,177,178,179,180,181,
     182,183,184,185,186,187,188,189,190,191,192,193,194,195,
     196,197,198,199,200,201,202,203,204,205,206,207,208,209,
     210,211,212,213,214,215,216,217,218,219,220,221,222,223,
     224,225,226,227,228,229,230,231,232,233,234,235,236,237,
     238,239,240,241,242,243,244,245,246,247,248,249,250,251,
     252,253,254,255], order=2^8)
```

### 4.4.4 Array dtypes

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

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

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

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

# Type cast an existing Galois field array to a different dtype
In [15]: a = a.astype(np.int16); a
Out[15]: GF([1, 2, 4, 5, 0, 6, 5, 2, 3, 5], order=7)
```
A specific dtype can be chosen by providing the dtype keyword argument during array creation.

```python
# Explicitly create a Galois field array with a specific dtype
In [17]: b = GF.Random(10, dtype=np.int16); b
Out[17]: GF([0, 1, 6, 4, 2, 2, 3, 3, 0, 6], order=7)
In [18]: b.dtype
Out[18]: dtype('int16')
```

### 4.4.5 Field element display modes

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

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

```python
In [19]: GF = galois.GF(2**3)
In [20]: a = GF.Random(10)
# The default mode represents the field elements as integers
In [21]: a
Out[21]: GF([4, 1, 3, 1, 0, 2, 6, 3, 7, 5], order=2^3)
# The display mode can be set to "poly" mode
In [22]: GF.display("poly"); a
Out[22]: GF([^2, 1, + 1, 0, , ^2 + , + 1, ^2 + + 1, ^2 + 1],
       order=2^3)
# The display mode can be set to "power" mode
In [23]: GF.display("power"); a
Out[23]: GF([^2, 1, ^3, 1, 0, , ^4, ^3, ^5, ^6], order=2^3)
# Reset the display mode to the default
In [24]: GF.display(); a
Out[24]: GF([4, 1, 3, 1, 0, 2, 6, 3, 7, 5], order=2^3)
```

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

```python
# The original display mode
In [25]: print(a)
GF([4, 1, 3, 1, 0, 2, 6, 3, 7, 5], order=2^3)
# The new display context
In [26]: with GF.display("poly"):
```
4.5 Galois field array arithmetic

4.5.1 Addition, subtraction, multiplication, division

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

---

4.5 Galois field array arithmetic

4.5.1 Addition, subtraction, multiplication, division

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

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

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

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

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

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

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

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

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

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

```
In [10]: X, Y = np.meshgrid(GF7.Elements(), GF7.Elements(), indexing="ij")
In [11]: X + Y
Out[11]:
GF([[0, 1, 2, 3, 4, 5, 6],
    [1, 0, 6, 5, 4, 3, 2],
    [2, 1, 0, 6, 5, 4, 3],
    [3, 2, 1, 0, 6, 5, 4],
    [4, 3, 2, 1, 0, 6, 5],
    [5, 4, 3, 2, 1, 0, 6],
    [6, 5, 4, 3, 2, 1, 0]], order=7)

In [12]: X - Y
Out[12]:
GF([[0, 6, 5, 4, 3, 2, 1],
    [1, 0, 6, 5, 4, 3, 2],
    [2, 1, 0, 6, 5, 4, 3],
    [3, 2, 1, 0, 6, 5, 4],
    [4, 3, 2, 1, 0, 6, 5],
    [5, 4, 3, 2, 1, 0, 6],
    [6, 5, 4, 3, 2, 1, 0]], order=7)

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

In [14]: X, Y = np.meshgrid(GF7.Elements(), GF7.Elements()[1:], indexing="ij")
In [15]: X / Y
Out[15]:
GF([[0, 0, 0, 0, 0, 0],
    [1, 4, 5, 2, 3, 6],
    [2, 1, 3, 4, 6, 5],
    [3, 5, 1, 6, 2, 4],
    [4, 2, 6, 1, 5, 3],
    [5, 6, 4, 3, 1, 2],
    [6, 3, 2, 5, 4, 1]], order=7)
```
4.5.2 Scalar multiplication

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

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

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

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

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

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

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

In prime fields \( GF(p) \), multiplication and scalar multiplication are equivalent.

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

In [21]: a = GF7.Random(10); a
Out[21]: GF([4, 5, 2, 4, 5, 2, 2, 4, 3, 0], order=7)

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

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

4.5.3 Exponentiation

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

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

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

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

(continues on next page)
4.5.4 Logarithm

```python
In [28]: GF7 = galois.GF(7)
In [29]: print(GF7)
<class 'numpy.ndarray over GF(7)'>

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

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

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

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

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

4.6 Extremely large fields

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

4.6.1 Large GF(p) fields

```python
In [1]: prime = 36893488147419103183
In [2]: galois.is_prime(prime)
Out[2]: True

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

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

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

(continues on next page)
4.6.2 Large GF(2^m) fields

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

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

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

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

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

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

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

In [15]: b
4.6. Extremely large fields

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

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

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

# Reset the display mode
In[18]: GF.display()
Out[18]: <galois._fields._class.DisplayContext at 0x7f5c76d56908>
5.1 Performance compared with native NumPy

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

5.1.1 Lookup performance

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

```python
In [1]: import numpy as np
In [2]: import galois
In [3]: GF = galois.GF(31)
In [4]: GF.ufunc_mode
Out[4]: 'jit-lookup'
In [5]: a = GF.Random(10_000, dtype=int)
In [6]: b = GF.Random(10_000, dtype=int)
In [7]: %timeit a * b
79.7 µs ± 1 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
In [8]: aa, bb = a.view(np.ndarray), b.view(np.ndarray)
# Equivalent calculation of a * b using native numpy implementation
In [9]: %timeit (aa * bb) % GF.order
96.6 µs ± 2.4 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

The galois ufunc runtime has a floor, however. This is due to a requirement to view the output array and convert its dtype with astype(). For example, for small array sizes NumPy is faster than galois because it doesn’t need to do these conversions.
In [4]: a = GF.Random(10, dtype=int)
In [5]: b = GF.Random(10, dtype=int)
In [6]: %timeit a * b
   45.1 µs ± 1.82 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
In [7]: aa, bb = a.view(np.ndarray), b.view(np.ndarray)

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

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

In [10]: a = GF.Random(10_000_000, dtype=int)
In [11]: b = GF.Random(10_000_000, dtype=int)
In [12]: %timeit a * b
   59.8 ms ± 1.64 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
In [13]: aa, bb = a.view(np.ndarray), b.view(np.ndarray)

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

5.1.2 Calculation performance

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

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

In [1]: import numpy as np
In [2]: import galois
In [3]: GF = galois.GF(2097169)
In [4]: GF.ufunc_mode
Out[4]: 'jit-calculate'
In [5]: a = GF.Random(10_000, dtype=int)
In [6]: b = GF.Random(10_000, dtype=int)
In [7]: %timeit a * b
   68.2 µs ± 2.09 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
And again, the runtime comparison with NumPy improves with large $N$ because the time of viewing and type converting the output is small compared to the computation time. `galois` achieves better performance than NumPy because the multiplication and modulo operations are compiled together into one ufunc rather than two.

5.1.3 Linear algebra performance

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

For extension fields $\text{GF}(p^m)$, the performance of `galois` is close to native NumPy linear algebra (about 10x slower). However, for extension fields, each multiplication operation is equivalently a convolution (polynomial multiplication) of two $m$-length arrays and polynomial remainder division with the irreducible polynomial. So it’s not an apples-to-apples comparison.
Below is a comparison of `galois` computing the correct matrix multiplication over \(GF(2^8)\) and NumPy computing a normal integer matrix multiplication (which is not the correct result!). This comparison is just for a performance reference.

```python
In [1]: import numpy as np
In [2]: import galois
In [3]: GF = galois.GF(2**8)
In [4]: A = GF.Random((100,100), dtype=int)
In [5]: B = GF.Random((100,100), dtype=int)
In [6]: %timeit A @ B
   7.13 ms ± 114 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
In [7]: AA, BB = A.view(np.ndarray), B.view(np.ndarray)
   # Native numpy matrix multiplication, which doesn't produce the correct result!!
In [8]: %timeit AA @ BB
   651 µs ± 12.4 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

### 5.2 Benchmarking

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

#### 5.2.1 Running a benchmark test

To run a benchmark, invoke `pytest` on the `benchmarks/` folder or a specific test set (e.g., `benchmarks/test_field_arithmetic.py`). It is also advised to pass extra arguments to format the display.

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

(continues on next page)
(continued from previous page)

--- of all tests durations ---

====================================================================================
sum of all tests durations
====================================================================================

25.80s

----------------------- benchmark "GF(2) Array Arithmetic: shape=(100_000,),
                   ufunc_mode='jit-calculate'": 8 tests ------------------------

<table>
<thead>
<tr>
<th>Name</th>
<th>Min (time in us)</th>
<th>Max (time in us)</th>
<th>Mean (time in us)</th>
</tr>
</thead>
<tbody>
<tr>
<td>test_add</td>
<td>54.9031 (1.04)</td>
<td>238.1941 (1.19)</td>
<td>62.2950</td>
</tr>
<tr>
<td>test_additive_inverse</td>
<td>52.8959 (1.0)</td>
<td>237.0442 (1.18)</td>
<td>58.6939</td>
</tr>
<tr>
<td>test_divide</td>
<td>207.9280 (3.93)</td>
<td>636.3150 (3.17)</td>
<td>234.0875</td>
</tr>
<tr>
<td>test_multiplicative_inverse</td>
<td>188.9290 (3.57)</td>
<td>652.8331 (3.25)</td>
<td>229.7088</td>
</tr>
<tr>
<td>test_multiply</td>
<td>54.6200 (1.03)</td>
<td>228.7410 (1.14)</td>
<td>61.9206</td>
</tr>
<tr>
<td>test_power</td>
<td>229.3210 (4.34)</td>
<td>284.5279 (1.42)</td>
<td>246.6029</td>
</tr>
<tr>
<td>test_scalar_multiply</td>
<td>1,561.2941 (29.52)</td>
<td>3,148.2361 (15.67)</td>
<td>2,058.2764</td>
</tr>
<tr>
<td>test_subtract</td>
<td>54.5362 (1.03)</td>
<td>228.7410 (1.14)</td>
<td>61.9206</td>
</tr>
</tbody>
</table>

---------------------- benchmark "GF(257) Array Arithmetic: shape=(100_000,),
                     ufunc_mode='jit-calculate'": 8 tests -----------------------

<table>
<thead>
<tr>
<th>Name</th>
<th>Min (time in us)</th>
<th>Max (time in us)</th>
<th>Mean (time in us)</th>
</tr>
</thead>
<tbody>
<tr>
<td>test_add</td>
<td>161.1579 (1.22)</td>
<td>583.2699 (1.20)</td>
<td>197.0225</td>
</tr>
<tr>
<td>test_additive_inverse</td>
<td>132.4308 (1.0)</td>
<td>546.5120 (1.12)</td>
<td>166.7050</td>
</tr>
<tr>
<td>test_divide</td>
<td>6,306.7721 (47.62)</td>
<td>7,277.9991 (14.94)</td>
<td>6,658.3270</td>
</tr>
<tr>
<td>test_multiplicative_inverse</td>
<td>6,299.4179 (47.57)</td>
<td>6,477.2971 (13.29)</td>
<td>6,392.0670</td>
</tr>
<tr>
<td>test_multiply</td>
<td>337.8210 (2.55)</td>
<td>487.2240 (1.0)</td>
<td>362.2038</td>
</tr>
<tr>
<td>test_power</td>
<td>333.8210 (2.55)</td>
<td>487.2240 (1.0)</td>
<td>362.2038</td>
</tr>
<tr>
<td>test_scalar_multiply</td>
<td>5,026.9060 (37.96)</td>
<td>5,523.7100 (11.34)</td>
<td>5,273.6145</td>
</tr>
<tr>
<td>test_subtract</td>
<td>5,026.9060 (37.96)</td>
<td>5,523.7100 (11.34)</td>
<td>5,273.6145</td>
</tr>
</tbody>
</table>

(continues on next page)

5.2. Benchmarking
### Benchmarking Results

#### GF(257) Array Arithmetic: shape=(100_000,), ufunc_mode='jit-lookup'

<table>
<thead>
<tr>
<th>Name</th>
<th>Min</th>
<th>Max</th>
<th>Mean</th>
<th>StdDev</th>
<th>Median</th>
</tr>
</thead>
<tbody>
<tr>
<td>test_add</td>
<td>175.6609</td>
<td>217.3430</td>
<td>182.2442</td>
<td>12.6704</td>
<td>177.3606</td>
</tr>
<tr>
<td>test_additive_inverse</td>
<td>142.2670</td>
<td>178.2598</td>
<td>152.1027</td>
<td>13.3374</td>
<td>144.3694</td>
</tr>
<tr>
<td>test_divide</td>
<td>414.2381</td>
<td>543.3671</td>
<td>473.2308</td>
<td>50.1936</td>
<td>475.6054</td>
</tr>
<tr>
<td>test_multiplicative_inverse</td>
<td>336.2088</td>
<td>459.0931</td>
<td>363.2939</td>
<td>40.4252</td>
<td>342.7650</td>
</tr>
<tr>
<td>test_multiply</td>
<td>319.1240</td>
<td>322.0430</td>
<td>322.5294</td>
<td>33.0754</td>
<td>339.3393</td>
</tr>
<tr>
<td>test_power</td>
<td>446.4111</td>
<td>505.2469</td>
<td>461.2746</td>
<td>19.2285</td>
<td>449.4530</td>
</tr>
<tr>
<td>test_scalar_multiply</td>
<td>1,037.2710</td>
<td>2,068.3082</td>
<td>1,197.2376</td>
<td>185.2372</td>
<td>1,137.1800</td>
</tr>
<tr>
<td>test_subtract</td>
<td>176.0179</td>
<td>177.3450</td>
<td>177.2260</td>
<td>11.6174</td>
<td>177.3606</td>
</tr>
</tbody>
</table>

#### GF(2^8) Array Arithmetic: shape=(100_000,), ufunc_mode='jit-calculate'

<table>
<thead>
<tr>
<th>Name</th>
<th>Min</th>
<th>Max</th>
<th>Mean</th>
<th>StdDev</th>
<th>Median</th>
</tr>
</thead>
<tbody>
<tr>
<td>test_add</td>
<td>62.4850</td>
<td>232.4961</td>
<td>73.3393</td>
<td>12.2285</td>
<td>192.2285</td>
</tr>
<tr>
<td>test_additive_inverse</td>
<td>142.2670</td>
<td>303.9362</td>
<td>185.7432</td>
<td>23.9378</td>
<td>224.7432</td>
</tr>
<tr>
<td>test_divide</td>
<td>31357.7619</td>
<td>33928.1799</td>
<td>32024.5656</td>
<td>1071.7328</td>
<td>31594.5051</td>
</tr>
<tr>
<td>test_multiplicative_inverse</td>
<td>16130.9310</td>
<td>22589.3550</td>
<td>18422.6860</td>
<td>2467.6099</td>
<td>17968.8521</td>
</tr>
<tr>
<td>test_multiply</td>
<td>1726.0939</td>
<td>2139.3239</td>
<td>1832.1717</td>
<td>172.9170</td>
<td>1769.2349</td>
</tr>
<tr>
<td>test_power</td>
<td>15418.5530</td>
<td>19151.2981</td>
<td>17459.3566</td>
<td>15418.5530</td>
<td>19151.2981</td>
</tr>
<tr>
<td>test_scalar_multiply</td>
<td>1424.5091</td>
<td>3249.2320</td>
<td>1838.1388</td>
<td>3249.2320</td>
<td>3249.2320</td>
</tr>
<tr>
<td>test_subtract</td>
<td>1673 (24.98)</td>
<td>1729.170 (8.99)</td>
<td>1769.2349 (26.58)</td>
<td>1673 (24.98)</td>
<td>1729.170 (8.99)</td>
</tr>
</tbody>
</table>
### Benchmarking "GF(2^8) Array Arithmetic: shape=(100_000,), ufunc_mode='jit-lookup'": 8 tests

<table>
<thead>
<tr>
<th>Name</th>
<th>(time in us)</th>
<th>StdDev</th>
<th>Min</th>
<th>Max</th>
<th>Mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>test_add</td>
<td>66.6410 (1.09)</td>
<td>307.5062 (1.25)</td>
<td>92.5952</td>
<td></td>
<td></td>
</tr>
<tr>
<td>test_additive_inverse</td>
<td>61.1460 (1.0)</td>
<td>288.4439 (1.17)</td>
<td>73.9261</td>
<td></td>
<td></td>
</tr>
<tr>
<td>test_divide</td>
<td>381.8041 (6.24)</td>
<td>642.9779 (2.61)</td>
<td>486.1582</td>
<td></td>
<td></td>
</tr>
<tr>
<td>test_multiplicative_inverse</td>
<td>308.8908 (5.05)</td>
<td>450.9010 (1.83)</td>
<td>343.5373</td>
<td></td>
<td></td>
</tr>
<tr>
<td>test_multiply</td>
<td>454.8710 (7.44)</td>
<td>614.1132 (2.49)</td>
<td>502.7098</td>
<td></td>
<td></td>
</tr>
<tr>
<td>test_power</td>
<td>1,337.2621 (21.87)</td>
<td>3,347.7580 (13.58)</td>
<td>1,646.4154</td>
<td></td>
<td></td>
</tr>
<tr>
<td>test_subtract</td>
<td>63.4668 (1.04)</td>
<td>246.5090 (1.0)</td>
<td>76.8375</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Benchmarking "GF(3^5) Array Arithmetic: shape=(1_000,), ufunc_mode='jit-calculate'": 8 tests

<table>
<thead>
<tr>
<th>Name</th>
<th>(time in us)</th>
<th>StdDev</th>
<th>Min</th>
<th>Max</th>
<th>Mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>test_add</td>
<td>600.3019 (1.34)</td>
<td>875.1580 (1.42)</td>
<td>669.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>test_additive_inverse</td>
<td>447.7361 (1.0)</td>
<td>618.4638 (1.0)</td>
<td>478.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>test_divide</td>
<td>20,522.7621 (45.84)</td>
<td>26,730.4152 (43.22)</td>
<td>23,591.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>test_multiplicative_inverse</td>
<td>18,970.5859 (42.37)</td>
<td>29,393.6098 (47.53)</td>
<td>22,934.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>test_multiply</td>
<td>1,429.8242 (3.19)</td>
<td>1,609.7031 (2.60)</td>
<td>1,518.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>test_power</td>
<td>1,319.3421 (2.95)</td>
<td>3,319.4488 (5.37)</td>
<td>1,828.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>test_subtract</td>
<td>594.8299 (1.33)</td>
<td>726.0030 (1.17)</td>
<td>614.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Name</th>
<th>(time in us)</th>
<th>StdDev</th>
<th>Min</th>
<th>Max</th>
<th>Mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>test_add</td>
<td>600.3019 (1.34)</td>
<td>875.1580 (1.42)</td>
<td>669.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>test_additive_inverse</td>
<td>447.7361 (1.0)</td>
<td>618.4638 (1.0)</td>
<td>478.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>test_divide</td>
<td>20,522.7621 (45.84)</td>
<td>26,730.4152 (43.22)</td>
<td>23,591.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>test_multiplicative_inverse</td>
<td>18,970.5859 (42.37)</td>
<td>29,393.6098 (47.53)</td>
<td>22,934.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>test_multiply</td>
<td>1,429.8242 (3.19)</td>
<td>1,609.7031 (2.60)</td>
<td>1,518.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>test_power</td>
<td>1,319.3421 (2.95)</td>
<td>3,319.4488 (5.37)</td>
<td>1,828.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>test_subtract</td>
<td>594.8299 (1.33)</td>
<td>726.0030 (1.17)</td>
<td>614.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

5.2. Benchmarking

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

Legend:
Outliers: 1 Standard Deviation from Mean; 1.5 IQR (InterQuartile Range) from 1st and 3rd Quartile.
OPS: Operations Per Second, computed as 1 / Mean

5.2.2 Comparing with previous benchmarks

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

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

```bash
$ git checkout branch
$ pytest benchmarks/test_field_arithmetic.py --benchmark-columns=min,max,mean,stddev,median --benchmark-sort=name --benchmark-compare=0001_master
```

Comparing against benchmarks from: Linux-CPython-3.8-64bit/0001_master.on

<table>
<thead>
<tr>
<th>test session starts</th>
</tr>
</thead>
<tbody>
<tr>
<td>session ends</td>
</tr>
</tbody>
</table>

platform linux -- Python 3.8.10, pytest-6.0.1, py-1.9.0, pluggy-0.13.1
benchmark: 3.4.1 (defaults: timer=time.perf_counter disable_gc=False min_rounds=5 min_time=0.000005 max_time=1.0 calibration_precision=10 warmup=False warmup_iterations=100000)

(continues on next page)
rootdir: /home/matt/repos/galois, configfile: setup.cfg
plugins: benchmark-3.4.1, extra-durations-0.1.3, anyio-2.1.0
collected 56 items

benchmarks/test_field_arithmetic.py ....................................................
→ ...
→ [100%]

====================================================================
sum of all tests durations.
=====================================================================
23.72s

----------------------------- benchmark "GF(2) Array Arithmetic: shape=(100_000,), ufunc_
→ mode='jit-calculate": 16 tests -----------------------------

<table>
<thead>
<tr>
<th>Name (time in us)</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mean</td>
<td>StdDev</td>
</tr>
<tr>
<td>test_add (0001_master)</td>
<td>53.6300 (1.04)</td>
<td>570.4910 (3.85)</td>
</tr>
<tr>
<td></td>
<td>74.1800 (1.31)</td>
<td>66.1639 (1.23)</td>
</tr>
<tr>
<td>test_add (NOW)</td>
<td>53.8658 (1.05)</td>
<td>148.1280 (1.0)</td>
</tr>
<tr>
<td></td>
<td>56.6440 (1.0)</td>
<td>55.1331 (1.03)</td>
</tr>
<tr>
<td>test_additive_inverse (0001_master)</td>
<td>51.5240 (1.0)</td>
<td>200.4388 (1.35)</td>
</tr>
<tr>
<td></td>
<td>62.2888 (1.10)</td>
<td>55.8060 (1.04)</td>
</tr>
<tr>
<td>test_additive_inverse (NOW)</td>
<td>51.7170 (1.00)</td>
<td>176.7289 (1.19)</td>
</tr>
<tr>
<td></td>
<td>57.3724 (1.01)</td>
<td>53.6300 (1.0)</td>
</tr>
<tr>
<td>test_divide (0001_master)</td>
<td>206.0279 (4.00)</td>
<td>823.5171 (5.56)</td>
</tr>
<tr>
<td></td>
<td>265.5769 (4.69)</td>
<td>239.2294 (4.46)</td>
</tr>
<tr>
<td>test_divide (NOW)</td>
<td>205.6009 (3.99)</td>
<td>698.4309 (4.72)</td>
</tr>
<tr>
<td></td>
<td>225.6514 (3.98)</td>
<td>212.8950 (3.97)</td>
</tr>
<tr>
<td>test_multiplicative_inverse (0001_master)</td>
<td>186.9558 (3.63)</td>
<td>566.1491 (3.82)</td>
</tr>
<tr>
<td></td>
<td>220.2184 (3.89)</td>
<td>198.8420 (3.71)</td>
</tr>
<tr>
<td>test_multiplicative_inverse (NOW)</td>
<td>187.0971 (3.58)</td>
<td>464.8890 (3.14)</td>
</tr>
<tr>
<td></td>
<td>263.0407 (3.58)</td>
<td>193.4364 (3.61)</td>
</tr>
<tr>
<td>test_multiply (0001_master)</td>
<td>56.0421 (1.09)</td>
<td>248.2592 (1.68)</td>
</tr>
<tr>
<td></td>
<td>81.5853 (1.44)</td>
<td>68.5630 (1.28)</td>
</tr>
<tr>
<td>test_multiply (NOW)</td>
<td>53.5559 (1.04)</td>
<td>283.5400 (1.91)</td>
</tr>
<tr>
<td></td>
<td>59.3049 (1.05)</td>
<td>54.8789 (1.02)</td>
</tr>
<tr>
<td>test_power (0001_master)</td>
<td>234.1510 (4.54)</td>
<td>330.9071 (2.23)</td>
</tr>
<tr>
<td></td>
<td>272.5068 (4.81)</td>
<td>274.3008 (5.11)</td>
</tr>
<tr>
<td>test_power (NOW)</td>
<td>230.1259 (4.47)</td>
<td>405.7041 (2.74)</td>
</tr>
<tr>
<td></td>
<td>277.7140 (4.90)</td>
<td>236.8400 (4.42)</td>
</tr>
<tr>
<td>test_scalar_multiply (0001_master)</td>
<td>1,345.2091 (26.11)</td>
<td>2,872.6910 (19.39)</td>
</tr>
<tr>
<td></td>
<td>1,627.9802 (28.74)</td>
<td>1,575.4123 (29.38)</td>
</tr>
<tr>
<td>test_scalar_multiply (NOW)</td>
<td>1,277.7401 (24.80)</td>
<td>2,155.3310 (14.55)</td>
</tr>
<tr>
<td></td>
<td>1,377.6186 (24.32)</td>
<td>1,313.8375 (24.50)</td>
</tr>
<tr>
<td>test_subtract (0001_master)</td>
<td>60.0331 (1.17)</td>
<td>300.8121 (2.03)</td>
</tr>
<tr>
<td></td>
<td>83.1536 (1.47)</td>
<td>69.4490 (1.29)</td>
</tr>
<tr>
<td>test_subtract (NOW)</td>
<td>53.4630 (1.04)</td>
<td>185.5700 (1.25)</td>
</tr>
<tr>
<td></td>
<td>59.3924 (1.05)</td>
<td>54.9241 (1.02)</td>
</tr>
</tbody>
</table>

5.2. Benchmarking
--- benchmark "GF(257) Array Arithmetic: shape=(100_000,), ufunc_mode='jit-calculate'": 16 tests ---

<table>
<thead>
<tr>
<th>Name</th>
<th>(time in us)</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mean</td>
<td>StdDev</td>
<td></td>
</tr>
<tr>
<td>test_add (0001_master)</td>
<td>175.3040</td>
<td>(1.37)</td>
<td>635.8509 (1.89)</td>
</tr>
<tr>
<td></td>
<td>197.5639</td>
<td>(1.37)</td>
<td>186.2240 (1.37)</td>
</tr>
<tr>
<td>test_add (NOW)</td>
<td>158.3470</td>
<td>(1.23)</td>
<td>636.1029 (1.89)</td>
</tr>
<tr>
<td>test_additive_inverse (0001_master)</td>
<td>176.8827</td>
<td>32.4656 (2.82)</td>
<td>67.7470 (1.23)</td>
</tr>
<tr>
<td>test_additive_inverse (NOW)</td>
<td>170.0383</td>
<td>38.9894 (3.38)</td>
<td>153.7615 (1.13)</td>
</tr>
<tr>
<td>test_divide (0001_master)</td>
<td>139.4011</td>
<td>170.0383 (1.18)</td>
<td>38.9894 (3.38)</td>
</tr>
<tr>
<td>test_divide (NOW)</td>
<td>139.4011</td>
<td>170.0383 (1.18)</td>
<td>38.9894 (3.38)</td>
</tr>
<tr>
<td>test_divide (0001_master)</td>
<td>6,233.1830</td>
<td>6,233.1830 (48.59)</td>
<td>6,527.6541 (19.42)</td>
</tr>
<tr>
<td>test_divide (NOW)</td>
<td>6,233.1830</td>
<td>6,233.1830 (48.59)</td>
<td>6,527.6541 (19.42)</td>
</tr>
<tr>
<td>test_multiplicative_inverse (0001_master)</td>
<td>5,915.1719</td>
<td>5,915.1719 (46.11)</td>
<td>6,276.5831 (18.68)</td>
</tr>
<tr>
<td>test_multiplicative_inverse (NOW)</td>
<td>5,915.1719</td>
<td>5,915.1719 (46.11)</td>
<td>6,276.5831 (18.68)</td>
</tr>
<tr>
<td>test_multiplicative_inverse (0001_master)</td>
<td>6,073.0814</td>
<td>6,073.0814 (41.97)</td>
<td>174.3328 (15.13)</td>
</tr>
<tr>
<td>test_multiplicative_inverse (NOW)</td>
<td>6,073.0814</td>
<td>6,073.0814 (41.97)</td>
<td>174.3328 (15.13)</td>
</tr>
<tr>
<td>test_multiply (0001_master)</td>
<td>294.6239</td>
<td>294.6239 (2.30)</td>
<td>300.5223 (2.08)</td>
</tr>
<tr>
<td>test_multiply (NOW)</td>
<td>294.6239</td>
<td>294.6239 (2.30)</td>
<td>300.5223 (2.08)</td>
</tr>
<tr>
<td>test_multiply (0001_master)</td>
<td>294.6239</td>
<td>294.6239 (2.30)</td>
<td>300.5223 (2.08)</td>
</tr>
<tr>
<td>test_multiply (NOW)</td>
<td>294.6239</td>
<td>294.6239 (2.30)</td>
<td>300.5223 (2.08)</td>
</tr>
<tr>
<td>test_power (0001_master)</td>
<td>5,487.4821</td>
<td>5,487.4821 (42.78)</td>
<td>7,486.4800 (45.22)</td>
</tr>
<tr>
<td>test_power (NOW)</td>
<td>5,487.4821</td>
<td>5,487.4821 (42.78)</td>
<td>7,486.4800 (45.22)</td>
</tr>
<tr>
<td>test_power (0001_master)</td>
<td>5,624.7412</td>
<td>126.0851 (10.94)</td>
<td>5,624.7412 (38.87)</td>
</tr>
<tr>
<td>test_power (NOW)</td>
<td>5,624.7412</td>
<td>126.0851 (10.94)</td>
<td>5,624.7412 (38.87)</td>
</tr>
<tr>
<td>test_scalar_multiply (0001_master)</td>
<td>4,765.4652</td>
<td>4,765.4652 (37.15)</td>
<td>5,108.4792 (15.20)</td>
</tr>
<tr>
<td>test_scalar_multiply (NOW)</td>
<td>4,765.4652</td>
<td>4,765.4652 (37.15)</td>
<td>5,108.4792 (15.20)</td>
</tr>
<tr>
<td>test_scalar_multiply (0001_master)</td>
<td>4,997.8213</td>
<td>126.2470 (10.95)</td>
<td>4,997.8213 (33.92)</td>
</tr>
<tr>
<td>test_scalar_multiply (NOW)</td>
<td>4,997.8213</td>
<td>126.2470 (10.95)</td>
<td>4,997.8213 (33.92)</td>
</tr>
<tr>
<td>test_subtract (0001_master)</td>
<td>1,030.8721</td>
<td>1,030.8721 (8.05)</td>
<td>1,030.8721 (8.05)</td>
</tr>
<tr>
<td>test_subtract (NOW)</td>
<td>1,030.8721</td>
<td>1,030.8721 (8.05)</td>
<td>1,030.8721 (8.05)</td>
</tr>
<tr>
<td>test_subtract (0001_master)</td>
<td>1,164.7107</td>
<td>198.8079 (17.25)</td>
<td>1,164.7107 (8.05)</td>
</tr>
<tr>
<td>test_subtract (NOW)</td>
<td>1,164.7107</td>
<td>198.8079 (17.25)</td>
<td>1,164.7107 (8.05)</td>
</tr>
<tr>
<td>test_subtract (0001_master)</td>
<td>1,144.9685</td>
<td>132.6795 (11.51)</td>
<td>1,144.9685 (7.91)</td>
</tr>
<tr>
<td>test_subtract (NOW)</td>
<td>1,144.9685</td>
<td>132.6795 (11.51)</td>
<td>1,144.9685 (7.91)</td>
</tr>
<tr>
<td>test_subtract (0001_master)</td>
<td>177.7399</td>
<td>31.2034 (2.71)</td>
<td>177.7399 (1.23)</td>
</tr>
<tr>
<td>test_subtract (NOW)</td>
<td>177.7399</td>
<td>31.2034 (2.71)</td>
<td>177.7399 (1.23)</td>
</tr>
</tbody>
</table>

--- benchmark "GF(257) Array Arithmetic: shape=(100_000,), ufunc_mode='jit-lookup'": 16 tests ---

<table>
<thead>
<tr>
<th>Name</th>
<th>(time in us)</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mean</td>
<td>StdDev</td>
<td></td>
</tr>
<tr>
<td>test_add (0001_master)</td>
<td>177.8340</td>
<td>(1.28)</td>
<td>227.8131 (1.34)</td>
</tr>
<tr>
<td></td>
<td>184.8953</td>
<td>(1.29)</td>
<td>179.1745 (1.28)</td>
</tr>
<tr>
<td>test_add (NOW)</td>
<td>188.0738</td>
<td>(1.36)</td>
<td>321.0511 (1.88)</td>
</tr>
<tr>
<td>test_multiply (0001_master)</td>
<td>226.4540</td>
<td>50.5466 (5.24)</td>
<td>195.8210 (1.40)</td>
</tr>
<tr>
<td>test_multiply (NOW)</td>
<td>226.4540</td>
<td>50.5466 (5.24)</td>
<td>195.8210 (1.40)</td>
</tr>
</tbody>
</table>
### 5.2. Benchmarking

---

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Min (time in us)</th>
<th>Mean (time in us)</th>
<th>StdDev (time in us)</th>
<th>Max (time in us)</th>
</tr>
</thead>
<tbody>
<tr>
<td>test_add</td>
<td>149.0880 (1.04)</td>
<td>159.0611 (1.13)</td>
<td>143.6260 (1.03)</td>
<td>170.4299 (1.0)</td>
</tr>
<tr>
<td>test_add (NOW)</td>
<td>130.8538 (1.0)</td>
<td>138.5838 (1.0)</td>
<td>139.7599 (1.0)</td>
<td>171.5040 (1.0)</td>
</tr>
<tr>
<td>test_additive_inverse</td>
<td>143.1082 (1.0)</td>
<td>94.6383 (1.0)</td>
<td>143.6260 (1.03)</td>
<td>149.0880 (1.04)</td>
</tr>
<tr>
<td>test_additive_inverse (NOW)</td>
<td>130.8538 (1.0)</td>
<td>138.5838 (1.0)</td>
<td>139.7599 (1.0)</td>
<td>171.5040 (1.0)</td>
</tr>
<tr>
<td>test_divide</td>
<td>407.8521 (2.85)</td>
<td>31.3674 (3.25)</td>
<td>393.6000 (2.82)</td>
<td>472.9840 (2.78)</td>
</tr>
<tr>
<td>test_divide (NOW)</td>
<td>368.6091 (2.66)</td>
<td>31.3674 (3.25)</td>
<td>393.6000 (2.82)</td>
<td>469.5239 (2.75)</td>
</tr>
<tr>
<td>test_multiplicative_inverse</td>
<td>397.4139 (2.78)</td>
<td>31.3674 (3.25)</td>
<td>393.6000 (2.82)</td>
<td>472.9840 (2.78)</td>
</tr>
<tr>
<td>test_multiplicative_inverse (NOW)</td>
<td>368.6091 (2.66)</td>
<td>31.3674 (3.25)</td>
<td>393.6000 (2.82)</td>
<td>469.5239 (2.75)</td>
</tr>
<tr>
<td>test_multiply</td>
<td>322.5519 (2.25)</td>
<td>18.2331 (1.89)</td>
<td>315.9530 (2.26)</td>
<td>322.5519 (2.25)</td>
</tr>
<tr>
<td>test_multiply (NOW)</td>
<td>283.0350 (2.04)</td>
<td>18.2331 (1.89)</td>
<td>315.9530 (2.26)</td>
<td>322.5519 (2.25)</td>
</tr>
<tr>
<td>test_power</td>
<td>492.7848 (3.44)</td>
<td>28.5527 (2.96)</td>
<td>509.1929 (3.64)</td>
<td>509.1929 (3.64)</td>
</tr>
<tr>
<td>test_power (NOW)</td>
<td>466.9611 (3.22)</td>
<td>28.5527 (2.96)</td>
<td>509.1929 (3.64)</td>
<td>509.1929 (3.64)</td>
</tr>
<tr>
<td>test_scalar_multiply</td>
<td>1,034.2200 (7.46)</td>
<td>303.5041 (1.36)</td>
<td>2,384.6650 (13.99)</td>
<td>2,384.6650 (13.99)</td>
</tr>
<tr>
<td>test_scalar_multiply (NOW)</td>
<td>1,047.1060 (7.56)</td>
<td>303.5041 (1.36)</td>
<td>2,384.6650 (13.99)</td>
<td>2,384.6650 (13.99)</td>
</tr>
<tr>
<td>test_subtract</td>
<td>20,575.8919 (345.65)</td>
<td>22,262.2380 (99.77)</td>
<td>20,575.8919 (345.65)</td>
<td>22,262.2380 (99.77)</td>
</tr>
<tr>
<td>test_subtract (NOW)</td>
<td>20,864.8040 (350.51)</td>
<td>22,262.2380 (99.77)</td>
<td>20,575.8919 (345.65)</td>
<td>22,262.2380 (99.77)</td>
</tr>
<tr>
<td>test_multiplicative_inverse</td>
<td>15,077.0759 (253.28)</td>
<td>15,525.0910 (69.57)</td>
<td>15,077.0759 (253.28)</td>
<td>15,525.0910 (69.57)</td>
</tr>
<tr>
<td>test_multiplicative_inverse (NOW)</td>
<td>15,077.0759 (253.28)</td>
<td>15,525.0910 (69.57)</td>
<td>15,077.0759 (253.28)</td>
<td>15,525.0910 (69.57)</td>
</tr>
<tr>
<td>Benchmark</td>
<td>Time (us)</td>
<td>Min</td>
<td>Max</td>
<td></td>
</tr>
<tr>
<td>-------------------------------</td>
<td>-----------------</td>
<td>--------</td>
<td>--------</td>
<td></td>
</tr>
<tr>
<td>Name</td>
<td>Time (us)</td>
<td>Mean</td>
<td>StdDev</td>
<td></td>
</tr>
<tr>
<td>test_add (0001_master)</td>
<td>62.1150 (1.04)</td>
<td>62.1150</td>
<td>183.1381 (1.0)</td>
<td></td>
</tr>
<tr>
<td>test_add (NOW)</td>
<td>73.7056 (1.08)</td>
<td>73.7056</td>
<td>19.0953 (1.73)</td>
<td></td>
</tr>
<tr>
<td>test_additive_inverse (0001_master)</td>
<td>63.4750 (1.06)</td>
<td>63.4750</td>
<td>306.6689 (1.67)</td>
<td></td>
</tr>
<tr>
<td>test_additive_inverse (NOW)</td>
<td>83.7360 (1.23)</td>
<td>83.7360</td>
<td>27.5077 (2.49)</td>
<td></td>
</tr>
<tr>
<td>test_divide (0001_master)</td>
<td>59.8810 (1.0)</td>
<td>59.8810</td>
<td>240.5860 (1.31)</td>
<td></td>
</tr>
<tr>
<td>test_divide (NOW)</td>
<td>404.0765 (5.94)</td>
<td>404.0765</td>
<td>20.4260 (1.85)</td>
<td></td>
</tr>
<tr>
<td>test_multiplicative_inverse (0001_master)</td>
<td>370.7750 (8.56)</td>
<td>370.7750</td>
<td>614.1500 (3.35)</td>
<td></td>
</tr>
<tr>
<td>test_multiplicative_inverse (NOW)</td>
<td>465.4766 (6.84)</td>
<td>465.4766</td>
<td>108.3985 (9.82)</td>
<td></td>
</tr>
<tr>
<td>test_power (0001_master)</td>
<td>317.9003 (4.67)</td>
<td>317.9003</td>
<td>12.0781 (1.09)</td>
<td></td>
</tr>
<tr>
<td>test_power (NOW)</td>
<td>376.0530 (5.53)</td>
<td>376.0530</td>
<td>67.0493 (6.07)</td>
<td></td>
</tr>
<tr>
<td>test_scalar_multiply (0001_master)</td>
<td>313.4176 (4.61)</td>
<td>313.4176</td>
<td>25.9667 (2.35)</td>
<td></td>
</tr>
<tr>
<td>test_scalar_multiply (NOW)</td>
<td>546.0020 (8.03)</td>
<td>546.0020</td>
<td>42.9520 (3.89)</td>
<td></td>
</tr>
<tr>
<td>test_subtract (0001_master)</td>
<td>314.1120 (20.60)</td>
<td>314.1120</td>
<td>1,747.94 (15.83)</td>
<td></td>
</tr>
<tr>
<td>test_subtract (NOW)</td>
<td>1,382.8612 (20.33)</td>
<td>1,382.8612</td>
<td>1,778.0490 (29.92)</td>
<td></td>
</tr>
</tbody>
</table>

(continues on next page)
### Benchmarking Results

#### Benchmark: GF(3^5) Array Arithmetic: shape=(1_000,), \texttt{ufunc_mode}= \texttt{\textquoteleft jit-calculate\textquoteright}:

<table>
<thead>
<tr>
<th>Name (time in us)</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>mean</td>
<td>577.8901 (1.29)</td>
<td>669.9329 (1.15)</td>
</tr>
<tr>
<td>stddev</td>
<td>29.3569 (1.20)</td>
<td>581.5226 (1.28)</td>
</tr>
<tr>
<td>median</td>
<td>596.5873 (1.22)</td>
<td>564.5840 (1.12)</td>
</tr>
</tbody>
</table>

#### Benchmark: GF(3^5) Array Arithmetic: shape=(1_000,), \texttt{ufunc_mode}= \texttt{\textquoteleft jit-lookup\textquoteright}:

<table>
<thead>
<tr>
<th>Name (time in us)</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>mean</td>
<td>23,700.1861 (52.26)</td>
<td>27,859.3549 (47.71)</td>
</tr>
<tr>
<td>stddev</td>
<td>1,936.6428 (79.20)</td>
<td>27,384.5338 (60.25)</td>
</tr>
<tr>
<td>median</td>
<td>25,382.3458 (51.91)</td>
<td>25,000.1471 (55.00)</td>
</tr>
</tbody>
</table>

---

### 5.2. Benchmarking

(continues on next page)
<table>
<thead>
<tr>
<th>Test Function</th>
<th>Mean</th>
<th>Standard Deviation</th>
<th>OPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>test_add (0001_master)</td>
<td>49.7401</td>
<td>(1.17)</td>
<td>76.4418 (1.15) 57.</td>
</tr>
<tr>
<td>test_add (NOW)</td>
<td>47.1489</td>
<td>(1.11)</td>
<td>72.2532 (1.09) 54.</td>
</tr>
<tr>
<td>test_additive_inverse (0001_master)</td>
<td>43.5489</td>
<td>(1.03)</td>
<td>108.0472 (1.62) 66.</td>
</tr>
<tr>
<td>test_additive_inverse (NOW)</td>
<td>42.7819</td>
<td>(1.01)</td>
<td>69.3691 (1.04) 48.</td>
</tr>
<tr>
<td>test_divide (0001_master)</td>
<td>43.5489</td>
<td>(1.02)</td>
<td>66.0670 (1.42) 46.</td>
</tr>
<tr>
<td>test_divide (NOW)</td>
<td>42.7819</td>
<td>(1.05)</td>
<td>69.3691 (1.04) 48.</td>
</tr>
<tr>
<td>test_multiplicative_inverse (0001_master)</td>
<td>42.4671</td>
<td>(1.0)</td>
<td>66.5220 (1.0) 46.</td>
</tr>
<tr>
<td>test_multiplicative_inverse (NOW)</td>
<td>44.0120</td>
<td>(1.04)</td>
<td>68.5172 (1.03) 48.</td>
</tr>
<tr>
<td>test_multiply (0001_master)</td>
<td>44.4860</td>
<td>(1.05)</td>
<td>67.9540 (1.02) 48.</td>
</tr>
<tr>
<td>test_multiply (NOW)</td>
<td>43.8211</td>
<td>(1.03)</td>
<td>67.5640 (1.02) 48.</td>
</tr>
<tr>
<td>test_power (0001_master)</td>
<td>65.6650</td>
<td>(1.55)</td>
<td>97.5579 (1.47) 73.</td>
</tr>
<tr>
<td>test_power (NOW)</td>
<td>65.0741</td>
<td>(1.52)</td>
<td>95.6031 (1.44) 71.</td>
</tr>
<tr>
<td>test_scalar_multiply (0001_master)</td>
<td>68.4580</td>
<td>(1.61)</td>
<td>300.1180 (4.51) 78.</td>
</tr>
<tr>
<td>test_scalar_multiply (NOW)</td>
<td>69.1570</td>
<td>(1.67)</td>
<td>256.4839 (3.86) 77.</td>
</tr>
<tr>
<td>test_subtract (0001_master)</td>
<td>49.5000</td>
<td>(1.17)</td>
<td>76.2481 (1.15) 61.</td>
</tr>
<tr>
<td>test_subtract (NOW)</td>
<td>47.5000</td>
<td>(1.12)</td>
<td>88.6670 (1.33) 65.</td>
</tr>
</tbody>
</table>

Legend:
Outliers: 1 Standard Deviation from Mean; 1.5 IQR (InterQuartile Range) from 1st and 3rd Quartile.

OPS: Operations Per Second, computed as 1 / Mean

56 passed in 24.48s.
For users who would like to actively develop with *galois*, these sections may prove helpful.

### 6.1 Lint the package

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

Listing 1: requirements-lint.txt

```python
pylint
```

Install the linter dependencies.

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

Run the linter.

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

### 6.2 Run the unit tests

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

Listing 2: requirements-test.txt

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

Install the test dependencies.

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

Run the unit tests.

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

6.3 Build the documentation

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

```python
1  sphinx>=3
2  recommonmark>=0.5
3  sphinx_rtd_theme>=0.5
4  readthedocs-sphinx-ext>=1.1
5  ipykernel
6  pandoc
7  numpy
```

Install the documentation dependencies.

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

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

```
$ sphinx-build -b html -v docs/build/
```
7.1 Galois Fields

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

7.1.1 Galois field class creation

Class factory functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>GF(order[, irreducible_poly, ...])</code></td>
<td>Factory function to construct a Galois field array class for $\mathbb{GF}(p^m)$.</td>
</tr>
<tr>
<td><code>Field(order[, irreducible_poly, ...])</code></td>
<td>Alias of <code>galois.GF()</code>.</td>
</tr>
</tbody>
</table>

`galois.GF`

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

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

Parameters

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

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

- **verify**(bool, optional) – Indicates whether to verify that the specified irreducible polynomial is in fact irreducible and whether the specified primitive element is in fact a generator of the multiplicative group. The default is `True`. For large fields and irreducible polynomials that are already known to be irreducible (which may take a long time to verify), this argument can be set to `False`. If the default irreducible polynomial and primitive element are used, no verification is performed because the defaults are guaranteed to be irreducible and a multiplicative generator, respectively.

- **compile**(str, optional) – The ufunc calculation mode. This can be modified after class construction with the `galois.FieldClass.compile()` method.
- **None** (default): For newly-created classes, `None` corresponds to "auto". For Galois field array classes of this type that were previously created, `None` does not modify the current ufunc compilation mode.
- **"auto"**: Selects “jit-lookup” for fields with order less than $2^{20}$, “jit-calculate” for larger fields, and “python-calculate” for fields whose elements cannot be represented with `numpy.int64`.
- **"jit-lookup"**: JIT compiles arithmetic ufuncs to use Zech log, log, and anti-log lookup tables for efficient computation. In the few cases where explicit calculation is faster than table lookup, explicit calculation is used.
- **"jit-calculate"**: JIT compiles arithmetic ufuncs to use explicit calculation. The “jit-calculate” mode is designed for large fields that cannot or should not store lookup tables in RAM. Generally, the “jit-calculate” mode is slower than “jit-lookup”.
- **"python-calculate"**: Uses pure-python ufuncs with explicit calculation. This is reserved for fields whose elements cannot be represented with `numpy.int64` and instead use `numpy.object_` with python `int` (which has arbitrary precision).

- **display**(str, optional) – The field element display representation. This can be modified after class construction with the `galois.FieldClass.display()` method.
- **None** (default): For newly-created classes, `None` corresponds to the integer representation ("int"). For Galois field array classes of this type that were previously created, `None` does not modify the current display mode.
- **"int"**: The element displayed as the integer representation of the polynomial. For example, $2x^2 + x + 2$ is an element of GF($3^3$) and is equivalent to the integer $23 = 2\cdot3^2 + 3 + 2$. 


- "poly": The element as a polynomial over GF(p) of degree less than m. For example, 
  \(2x^2 + x + 2\) is an element of GF(3^2).

- "power": The element as a power of the primitive element, see `galois.FieldClass.
  primitive_element`. For example, 
  \(2x^2 + x + 2 = \alpha^5\) in GF(3^2) with irreducible
  polynomial \(x^3 + 2x + 1\) and primitive element \(\alpha = x\).

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

**Return type** `galois.FieldClass`

**Notes**

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

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

**Examples**

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

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

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

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

(continues on next page)
order: 3
irreducible_poly: x + 1
is_primitive_poly: True
primitive_element: 2

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

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

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

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

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

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

# Construct a very large GF(2^m) class
In [8]: GF2m = galois.GF(2**100); print(GF2m.properties)
GF(2^100):
    characteristic: 2
degree: 100
order: 12676506002822940149670320376
irreducible_poly: x^100 + x^57 + x^56 + x^55 + x^52 + x^48 + x^47 + x^46 + x^45 + x^44 + x^43 + x^41 + x^37 + x^36 + x^35 + x^34 + x^31 + x^30 + x^27 + x^25 + x^24 + x^22 + x^20 + x^19 + x^16 + x^15 + x^11 + x^9 + x^8 + x^6 + x^5 + x^3 + 1
is_primitive_poly: True
primitive_element: x

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

# Construct a very large GF(p) class
In [10]: GFp = galois.GF(36893488147419103183); print(GFp.properties)

(continues on next page)
GF(36893488147419103183):
  characteristic: 36893488147419103183
  degree: 1
  order: 36893488147419103183
  irreducible_poly: x + 36893488147419103180
  is_primitive_poly: True
  primitive_element: 3

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

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

In [12]: GF = galois.GF(2**8)
In [13]: GF.Random()
Out[13]: GF(158, order=2^8)
In [14]: GF = galois.GF(2**8, display="poly")
In [15]: GF.Random()
Out[15]: GF(^6 + ^5 + ^4 + + 1, order=2^8)

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

In [16]: poly1 = galois.Poly([1, 0, 0, 0, 1, 1, 0, 1, 1])
In [17]: poly2 = poly1.integer
In [18]: galois.GF(2**8, irreducible_poly=poly1) is galois.GF(2**8, irreducible_poly=poly2)
Out[18]: True

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

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

Alias of `galois.GF()`.

### Abstract base classes

<table>
<thead>
<tr>
<th><code>FieldArray</code>&lt;br&gt;(array[, dtype, copy, order, ndmin])</th>
<th>An array over GF($p^n$).</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>FieldClass</code>(name, bases, namespace, **kwargs)</td>
<td>Defines a metaclass for all <code>galois.FieldArray</code> classes.</td>
</tr>
</tbody>
</table>

**galois.FieldArray**

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

An array over GF($p^n$).

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

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

```
In [1]: GF = galois.GF(7)
In [2]: issubclass(GF, galois.FieldArray)
Out[2]: True

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

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

### Notes

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

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

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

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

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

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

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

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

Galois field arrays can be created from existing numpy arrays.

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

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

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

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

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

# A scalar GF(2^8) element from its polynomial representation
In [13]: GF256("x^5 + x^2 + 1")
Out[13]: GF(37, order=2^8)
```
# A GF(2^8) array from a list of elements in their integer representation
In [14]: GF256([[142, 27], [92, 253]])
Out[14]: GF([[142, 27],
         [ 92, 253]], order=2^8)

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

There’s also an alternate constructor `Vector()` (and accompanying `vector()` method) to convert an array of coefficients over GF(p) with last dimension m into Galois field elements in GF(p^m).

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

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

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

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

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

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

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

Constructors

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>__init__</code>(array[, dtype, copy, order, ndmin])</td>
<td>Creates an array over GF(p^m).</td>
</tr>
<tr>
<td><code>Elements</code>(dtype)</td>
<td>Creates a 1-D Galois field array of the field’s elements {0, ..., p^m - 1}.</td>
</tr>
<tr>
<td><code>Identity</code>(size[, dtype])</td>
<td>Creates an n x n Galois field identity matrix.</td>
</tr>
<tr>
<td><code>Ones</code>(shape[, dtype])</td>
<td>Creates a Galois field array with all ones.</td>
</tr>
<tr>
<td><code>Random</code>(shape[, low, high, dtype])</td>
<td>Creates a Galois field array with random field elements.</td>
</tr>
</tbody>
</table>

continues on next page
Table 3 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Range()</code></td>
<td>Creates a 1-D Galois field array with a range of field elements.</td>
</tr>
<tr>
<td><code>Vandermonde()</code></td>
<td>Creates an ( m \times n ) Vandermonde matrix of ( a \in GF(p^m) ).</td>
</tr>
<tr>
<td><code>Vector()</code></td>
<td>Creates a Galois field array over ( GF(p^m) ) from length-( m ) vectors over the prime subfield ( GF(p) ).</td>
</tr>
<tr>
<td><code>Zeros()</code></td>
<td>Creates a Galois field array with all zeros.</td>
</tr>
</tbody>
</table>

**Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>lu_decompose()</code></td>
<td>Decomposes the input array into the product of lower and upper triangular matrices.</td>
</tr>
<tr>
<td><code>lup_decompose()</code></td>
<td>Decomposes the input array into the product of lower and upper triangular matrices using partial pivoting.</td>
</tr>
<tr>
<td><code>row_reduce()</code></td>
<td>Performs Gaussian elimination on the matrix to achieve reduced row echelon form.</td>
</tr>
<tr>
<td><code>vector()</code></td>
<td>Converts the Galois field array over ( GF(p^m) ) to length-( m ) vectors over the prime subfield ( GF(p) ).</td>
</tr>
</tbody>
</table>

**Special Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>__add__</code>(other)</td>
<td>Adds two Galois field arrays element-wise.</td>
</tr>
<tr>
<td><code>__divmod__</code>(other)</td>
<td>Divides two Galois field arrays element-wise and returns the quotient and remainder.</td>
</tr>
<tr>
<td><code>__floordiv__</code>(other)</td>
<td>Divides two Galois field arrays element-wise.</td>
</tr>
<tr>
<td><code>__mod__</code>(other)</td>
<td>Divides two Galois field arrays element-wise and returns the remainder.</td>
</tr>
<tr>
<td><code>__mul__</code>(other)</td>
<td>Multiplies two Galois field arrays element-wise.</td>
</tr>
<tr>
<td><code>__pow__</code>(other)</td>
<td>Exponentiates a Galois field array element-wise.</td>
</tr>
<tr>
<td><code>__sub__</code>(other)</td>
<td>Subtracts two Galois field arrays element-wise.</td>
</tr>
<tr>
<td><code>__truediv__</code>(other)</td>
<td>Divides two Galois field arrays element-wise.</td>
</tr>
</tbody>
</table>

**classmethod Elements** *(dtype=\(None\))*

Creates a 1-D Galois field array of the field’s elements \( \{0, \ldots, p^m - 1\} \).

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

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

**Return type**  \(galois.FieldArray\)
Examples

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

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

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

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

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

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

classmethod Identity(size, dtype=None)

Creates an \( n \times n \) Galois field identity matrix.

Parameters

- **size** (*int*) – The size \( n \) along one axis of the matrix. The resulting array has shape \((size, size)\).

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

Returns

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

Return type

`galois.FieldArray`
Examples

```python
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)
```
**Returns**  A Galois field array of random field elements.

**Return type**  `galois.FieldArray`

**Examples**

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

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

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

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

**Parameters**

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

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

**Return type**  `galois.FieldArray`

**Examples**

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

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

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

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

**Parameters**

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

**Returns**  The \( m \times n \) Vandermonde matrix.

**Return type**  `galois.FieldArray`
Examples

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

In [2]: a = GF.primitive_element

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

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

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

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

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

Parameters

- `array (array_like)` – The input array with field elements in GF($p$) to be converted to a Galois field array in GF($p^m$). The last dimension of the input array must be $m$. An input array with shape (n1, n2, m) has output shape (n1, n2). By convention, the vectors are ordered from highest degree to 0-th degree.
- `dtype (numpy.dtype, optional)` – The numpy.dtype of the array elements. The default is None which represents the smallest unsigned dtype for this class, i.e. the first element in galois.FieldClass.dtypes.

Returns  A Galois field array over GF($p^m$).

Return type  galois.FieldArray
```

Examples

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

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

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

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

(continues on next page)
classmethod Zeros(shape, dtype=None)

Creates a Galois field array with all zeros.

Parameters
- shape (int, tuple) – A numpy-compliant shape tuple, see numpy.ndarray.shape. An empty tuple () represents a scalar. A single integer or 1-tuple, e.g. N or (N,), represents the size of a 1-D array. A 2-tuple, e.g. (N, N), represents a 2-D array with each element indicating the size in each dimension.
- dtype (numpy.dtype, optional) – The numpy.dtype of the array elements. The default is None which represents the smallest unsigned dtype for this class, i.e. the first element in galois.FieldClass.dtypes.

Returns A Galois field array of zeros.

Return type galois.FieldArray

Examples

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

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

__add__(other)

Adds two Galois field arrays element-wise.

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

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

Returns The Galois field array self + other.

Return type galois.FieldArray
Examples

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

```
__divmod__(other)
Divides two Galois field arrays element-wise and returns the quotient and remainder.

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

Parameters

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

Returns

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

Examples

```python
In [1]: GF = galois.GF(7)
In [2]: a = GF.Random((2,5)); a
Out[2]:
GF([[5, 6, 1, 4, 0],
    [2, 0, 1, 0, 4]], order=7)
In [3]: b = GF.Random(5, low=1); b
Out[3]: GF([1, 3, 3, 5, 3], order=7)
In [4]: q, r = divmod(a, b)
In [5]: q, r
Out[5]:
(GF([[5, 2, 5, 5, 0],
    [2, 0, 5, 0, 6]], order=7),
GF([[0, 0, 5, 0, 6],
    [0, 0, 0, 0, 0]], order=7))
In [6]: b^nq + r
Out[6]:
```
(continues on next page)
Galois

(continued from previous page)

\[
\begin{align*}
\text{GF}([5, 6, 1, 4, 0], \\
[2, 0, 1, 0, 4]), \text{order}=7
\end{align*}
\]

__floordiv__(other)
Divides two Galois field arrays element-wise.

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

**Parameters**

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

**Returns**

The Galois field array self // other.

**Return type**

galois.FieldArray

**Examples**

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

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

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

In [4]: a // b
Out[4]:
\[
\text{GF}([5, 5, 5, 5, 5], \\
[5, 1, 1, 2, 4]), \text{order}=7
\]

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

Creates an array over GF(p^m).

**Parameters**

- **array** (int, str, tuple, list, numpy.ndarray, galois.FieldArray) – The input array-like object to be converted to a Galois field array. See the examples section for demonstrations of array creation using each input type. See galois.FieldClass.display() and galois.FieldClass.display_mode for a description of the “integer” and “polynomial” representation of Galois field elements.
  - int: A single integer, which is the “integer representation” of a Galois field element, creates a 0-D array.
  - str: A single string, which is the “polynomial representation” of a Galois field element, creates a 0-D array.
  - tuple, list: A list or tuple (or nested lists/tuples) of ints or strings (which can be mix-and-matched) creates an array of Galois field elements from their integer or polynomial representations. * numpy.ndarray, galois.FieldArray: An array of ints creates a copy of the array over this specific field.
• **dtype** *(numpy.dtype, optional)* – The `numpy.dtype` of the array elements. The default is `None` which represents the smallest unsigned dtype for this class, i.e. the first element in `galois.FieldClass.dtypes`.

• **copy** *(bool, optional)* – The `copy` keyword argument from `numpy.array()`. The default is `True` which makes a copy of the input array.

• **order** *(str, optional)* – The `order` keyword argument from `numpy.array()`. Valid values are "K" (default), "A", "C", or "F".

• **ndmin** *(int, optional)* – The `ndmin` keyword argument from `numpy.array()`. The minimum number of dimensions of the output. The default is 0.

**Returns** An array over \(GF(p^m)\).

**Return type** `galois.FieldArray`

```python
__mod__(other)
```

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

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

**Parameters**

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

**Returns** The Galois field array \(self \% other\).

**Return type** `galois.FieldArray`

**Examples**

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

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

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

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

```python
__mul__(other)
```

Multiplies two Galois field arrays element-wise.

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

**Warning:** When both multiplicands are `galois.FieldArray`, that indicates a Galois field multiplication. When one multiplicand is an integer or integer `numpy.ndarray`, that indicates a scalar multiplication (repeated addition). Galois field multiplication and scalar multiplication are equivalent in prime fields, but not in extension fields.
Parameters `other` (*numpy.ndarray*, *galois.FieldArray*) – A *numpy.ndarray* of integers for scalar multiplication or a *galois.FieldArray* of Galois field elements for finite field multiplication.

Returns The Galois field array `self * other`.

Return type *galois.FieldArray*

Examples

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

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

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

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

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

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

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

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

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

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

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

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

`__pow__`(other)

Exponentiates a Galois field array element-wise.

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

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

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

---

78 Chapter 7. API Reference v0.0.21
Return type  \texttt{galois.FieldArray}

\textbf{Examples}

\begin{verbatim}
In [1]: GF = galois.GF(7)

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

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

In [4]: a ** b
Out[4]:
GF([[2, 1, 1, 1, 6],
     [1, 1, 1, 4, 1]], order=7)
\end{verbatim}

\textbf{\_\_sub\_\_}(other)
Subtracts two Galois field arrays element-wise.

\textbf{Broadcasting rules apply. Both arrays must be over the same Galois field.}

\textbf{Parameters}  \texttt{other (galois.FieldArray)} – The other Galois field array.

\textbf{Returns}  The Galois field array \texttt{self - other}.

\textbf{Return type}  \texttt{galois.FieldArray}

\textbf{Examples}

\begin{verbatim}
In [1]: GF = galois.GF(7)

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

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

In [4]: a - b
Out[4]:
GF([[1, 6, 2, 6, 1],
     [6, 3, 5, 4, 0]], order=7)
\end{verbatim}

\textbf{\_\_truediv\_\_}(other)
Divides two Galois field arrays element-wise.

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

\textbf{Parameters}  \texttt{other (galois.FieldArray)} – The other Galois field array.
**Returns** The Galois field array `self / other`.

**Return type** `galois.FieldArray`

### Examples

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

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

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

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

### lu_decompose()

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

**Returns**

- `galois.FieldArray` – The lower triangular matrix.

### Examples

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

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

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

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

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

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

## lup_decompose()

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

Returns

- `galois.FieldArray` – The lower triangular matrix.
- `galois.FieldArray` – The permutation matrix.

## Examples

```
In [1]: GF = galois.GF(5)
In [2]: A = GF([[1, 3, 2, 0], [3, 4, 2, 3], [0, 2, 1, 4], [4, 3, 3, 1]])
In [3]: L, U, P = A.lup_decompose()
In [4]: L
Out[4]:
GF([[1, 0, 0, 0],
    [0, 1, 0, 0],
    [3, 0, 1, 0],
    [4, 3, 2, 1]], order=5)
In [5]: U
Out[5]:
GF([[1, 3, 2, 0],
    [0, 2, 1, 4],
    [0, 0, 1, 3],
    [0, 0, 0, 3]], order=5)
In [6]: P
Out[6]:
GF([[1, 0, 0, 0],
    [0, 0, 1, 0],
    [0, 1, 0, 0],
    [0, 0, 0, 1]], order=5)
```

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

## row_reduce(ncols=None)

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

Row reduction operations

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

**Parameters** 

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

**Returns** 

The reduced row echelon form of the input array.

**Return type** 

`galois.FieldArray`

## Examples

```python
In [1]: GF = galois.GF(31)
In [2]: A = GF.Random((4,4)); A
Out[2]:
GF([[ 4, 28, 7, 21],
    [12, 26, 11, 21],
    [12, 26, 17, 18],
    [30, 27, 25, 17]], order=31)
In [3]: A.row_reduce()
Out[3]:
GF([[1, 0, 0, 0],
    [0, 1, 0, 0],
    [0, 0, 1, 0],
    [0, 0, 0, 1]], order=31)
In [4]: np.linalg.matrix_rank(A)
Out[4]: 4
```

One column is a linear combination of another.

```python
In [5]: GF = galois.GF(31)
In [6]: A = GF.Random((4,4)); A
Out[6]:
GF([[13, 20, 16, 0],
    [ 3, 21, 18, 19],
    [ 6, 4, 4, 22],
    [11, 3, 12, 14]], order=31)
In [7]: A[:,2] = A[:,1] * GF(17); A
Out[7]:
GF([[13, 20, 30, 0],
    [ 3, 21, 16, 19],
    [ 6, 4, 6, 22],
    [11, 3, 20, 14]], order=31)
In [8]: A.row_reduce()
Out[8]:
GF([[1, 0, 0, 0],
    [0, 1, 17, 0],
    ...]
```

(continues on next page)
One row is a linear combination of another.

```python
In [10]: GF = galois.GF(31)
In [11]: A = GF.Random((4,4)); A
Out[11]:
GF([[ 6, 18, 12, 4],
    [12, 25, 4, 9],
    [20, 10, 8, 21],
    [11, 14, 16, 30]], order=31)
In [12]: A[3,:] = A[2,:] * GF(8); A
Out[12]:
GF([[ 6, 18, 12, 4],
    [12, 25, 4, 9],
    [20, 10, 8, 21],
    [ 5, 18, 2, 13]], order=31)
In [13]: A.row_reduce()
Out[13]:
GF([[1, 0, 0, 9],
    [0, 1, 0, 6],
    [0, 0, 1, 23],
    [0, 0, 0, 0]], order=31)
In [14]: np.linalg.matrix_rank(A)
Out[14]: 3
```

The `vector()` function converts the Galois field array over \( \text{GF}(p^m) \) to length-\( m \) vectors over the prime subfield \( \text{GF}(p) \).

This function is the inverse operation of the `Vector()` constructor. For an array with shape \((n_1, n_2)\), the output shape is \((n_1, n_2, m)\). By convention, the vectors are ordered from highest degree to 0-th degree.

```
vector(dtype=None)
```

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

**Parameters**

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

**Returns**

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

**Return type**

- `galois.FieldArray`
Examples

```python
In [1]: GF = galois.GF(2**6)
In [2]: a = GF.Random(3); a
Out[2]: GF([25, 13, 25], order=2^6)
In [3]: with GF.display("poly"):  
   ...:   print(a)
   ...:
GF([^4 + ^3 + 1, ^3 + ^2 + 1, ^4 + ^3 + 1], order=2^6)
In [4]: vec = a.vector(); vec
Out[4]: GF([[0, 1, 1, 0, 0, 1],
          [0, 0, 1, 1, 0, 1],
          [0, 1, 1, 0, 0, 1]], order=2)
In [5]: GF.Vector(vec)
Out[5]: GF([25, 13, 25], order=2^6)
```

galois.FieldClass

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

Defines a metaclass for all galois.FieldArray classes.

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

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

```python
In [1]: GF = galois.GF(7)
In [2]: isinstance(GF, galois.FieldClass)
Out[2]: True
```

Constructors

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

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>arithmetic_table(operation[, x, y])</code></td>
<td>Generates the specified arithmetic table for the Galois field.</td>
</tr>
<tr>
<td><code>compile(mode)</code></td>
<td>Recompiles the just-in-time compiled numba ufuncs for a new calculation mode.</td>
</tr>
<tr>
<td><code>display([mode])</code></td>
<td>Sets the display mode for all Galois field arrays of this type.</td>
</tr>
<tr>
<td><code>repr_table([primitive_element, sort])</code></td>
<td>Generates a field element representation table comparing the power, polynomial, vector, and integer representations.</td>
</tr>
</tbody>
</table>

Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>characteristic</code></td>
<td>The prime characteristic $p$ of the Galois field $GF(p^m)$.</td>
</tr>
<tr>
<td><code>default_ufunc_mode</code></td>
<td>The default ufunc arithmetic mode for this Galois field.</td>
</tr>
<tr>
<td><code>degree</code></td>
<td>The prime characteristic’s degree $m$ of the Galois field $GF(p^m)$.</td>
</tr>
<tr>
<td><code>display_mode</code></td>
<td>The representation of Galois field elements, either &quot;int&quot;, &quot;poly&quot;, or &quot;power&quot;.</td>
</tr>
<tr>
<td><code>dtypes</code></td>
<td>List of valid integer <code>numpy.dtype</code> values that are compatible with this Galois field.</td>
</tr>
<tr>
<td><code>irreducible_poly</code></td>
<td>The irreducible polynomial $f(x)$ of the Galois field $GF(p^m)$.</td>
</tr>
<tr>
<td><code>is_extension_field</code></td>
<td>Indicates if the field’s order is a prime power.</td>
</tr>
<tr>
<td><code>is_prime_field</code></td>
<td>Indicates if the field’s order is prime.</td>
</tr>
<tr>
<td><code>is_primitive_poly</code></td>
<td>Indicates whether the <code>irreducible_poly</code> is a primitive polynomial.</td>
</tr>
<tr>
<td><code>name</code></td>
<td>The Galois field name.</td>
</tr>
<tr>
<td><code>order</code></td>
<td>The order $p^m$ of the Galois field $GF(p^m)$.</td>
</tr>
<tr>
<td><code>prime_subfield</code></td>
<td>The prime subfield $GF(p)$ of the extension field $GF(p^m)$.</td>
</tr>
<tr>
<td><code>primitive_element</code></td>
<td>A primitive element $\alpha$ of the Galois field $GF(p^m)$.</td>
</tr>
<tr>
<td><code>primitive_elements</code></td>
<td>All primitive elements $\alpha$ of the Galois field $GF(p^m)$.</td>
</tr>
<tr>
<td><code>properties</code></td>
<td>A formatted string displaying relevant properties of the Galois field.</td>
</tr>
<tr>
<td><code>ufunc_mode</code></td>
<td>The mode for ufunc compilation, either &quot;jit-lookup&quot;, &quot;jit-calculate&quot;, or &quot;python-calculate&quot;.</td>
</tr>
<tr>
<td><code>ufunc_modes</code></td>
<td>All supported ufunc modes for this Galois field array class.</td>
</tr>
</tbody>
</table>

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

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

Generates the specified arithmetic table for the Galois field.

**Parameters**

- `operation (str)` – The arithmetic operation, either "+", "-", "/", or ".".
• \( x \) (\texttt{galois.FieldArray}, \textit{optional}) – Optionally specify the \( x \) values for the arithmetic table. The default is \texttt{None} which represents \( \{0,\ldots,p^m-1\} \).

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

\textbf{Returns} \quad \text{A UTF-8 formatted arithmetic table.}

\textbf{Return type} \quad \texttt{str}

\textbf{Examples}

\begin{verbatim}
In [1]: GF = galois.GF(3**2)

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

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

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

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

```
x + y 0 | 1 | 2 | | + 1 | + 2 | 2 | 2 + 1 | 2 + \omega
     0 0 | 1 | 2 | | + 1 | + 2 | 2 | 2 + 1 | 2 + \omega
     1 1 | 2 | 0 | | + 1 | + 2 | | 2 + 1 | 2 + 2 | 2 + \omega
     2 2 | 0 | 1 | | + 2 | | 1 | 2 + 2 | 2 | 2 + \omega
```
\end{verbatim}
In [5]: GF.display("power");

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

<table>
<thead>
<tr>
<th>x + y</th>
<th>0</th>
<th>1</th>
<th>^2</th>
<th>^3</th>
<th>^4</th>
<th>^5</th>
<th>^6</th>
<th>^7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>^2</td>
<td>^3</td>
<td>^4</td>
<td>^5</td>
<td>^6</td>
<td>^7</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>^4</td>
<td>^2</td>
<td>^7</td>
<td>^6</td>
<td>0</td>
<td>^3</td>
<td>^5</td>
</tr>
<tr>
<td></td>
<td>^2</td>
<td>^5</td>
<td>^3</td>
<td>1</td>
<td>^7</td>
<td>0</td>
<td>^4</td>
<td>^6</td>
</tr>
<tr>
<td>^2</td>
<td>^4</td>
<td>^7</td>
<td>^3</td>
<td>^6</td>
<td>^4</td>
<td>1</td>
<td>0</td>
<td>^5</td>
</tr>
<tr>
<td>^3</td>
<td>^3</td>
<td>^6</td>
<td>1</td>
<td>^4</td>
<td>^7</td>
<td>^5</td>
<td>^2</td>
<td>0</td>
</tr>
<tr>
<td>^4</td>
<td>^4</td>
<td>0</td>
<td>^7</td>
<td>^5</td>
<td>1</td>
<td>^6</td>
<td>^3</td>
<td>^2</td>
</tr>
<tr>
<td>^5</td>
<td>^5</td>
<td>^3</td>
<td>0</td>
<td>^1</td>
<td>^2</td>
<td>^6</td>
<td>^7</td>
<td>^4</td>
</tr>
<tr>
<td>^6</td>
<td>^6</td>
<td>^5</td>
<td>^4</td>
<td>0</td>
<td>^3</td>
<td>^7</td>
<td>^2</td>
<td>1</td>
</tr>
<tr>
<td>^7</td>
<td>^7</td>
<td>^6</td>
<td>^5</td>
<td>0</td>
<td>^2</td>
<td>^4</td>
<td>1</td>
<td>^3</td>
</tr>
</tbody>
</table>

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

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

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

(continues on next page)
Out[9]: GF([2 + 1, + 1, 2 + 2], order=3^2)

In [10]: print(GF.arithmetic_table("+", x=x, y=y))
x + y 2 + 1 | + 1 | 2 + 2
2 + 1 + 2 | 2 |

+ 2 0 | 2 | 1
2 + 1 | 1 | + 2
0 2 + 1 | + 1 | 2 + 2

In [11]: GF.display();

`compile(mode)`
Recompile the just-in-time compiled numba ufuncs for a new calculation mode.
This function updates `ufunc_mode`.

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

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

`display(mode='int')`
Sets the display mode for all Galois field arrays of this type.
The display mode can be set to either the integer representation, polynomial representation, or power representation. This function updates `display_mode`.

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

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

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

Chapter 7. API Reference v0.0.21
• "poly": The element as a polynomial over GF(p) of degree less than m. For example, 
2x^2 + x + 2 is an element of GF(3^3).

• "power": The element as a power of the primitive element, see FieldClass.
  primitive_element. For example, 2x^2 + x + 2 = α^5 in GF(3^3) with irreducible
  polynomial x^3 + 2x + 1 and primitive element α = x.

Examples

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

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

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

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

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

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

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

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

For the polynomial representation, when the primitive element is α = x in GF(p)[x] the polynomial indeterminate used is α.

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

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

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

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

```
In [14]: GF = galois.GF(2**8, irreducible_poly=galois.Poly.Degrees([8, 4, 3, 1, 0]))
In [15]: print(GF.properties)
GF(2^8):
    characteristic: 2
    degree: 8
    order: 256
    irreducible_poly: x^8 + x^4 + x^3 + x + 1
    is_primitive_poly: False
    primitive_element: x + 1
In [16]: a = GF.Random()
```

```
In [17]: print(GF.display_mode, a)
int GF(62, order=2^8)
In [18]: with GF.display("poly"):
    ....:    print(GF.display_mode, a)
    ....:    poly GF(x^5 + x^4 + x^3 + x^2 + x, order=2^8)
In [19]: with GF.display("power"):
    ....:    print(GF.display_mode, a)
    ....:    power GF(^218, order=2^8)
# The display mode is reset after exiting the context manager
In [20]: print(GF.display_mode, a)
int GF(62, order=2^8)
```
Parameters

- **primitive_element** (galois.FieldArray, optional) – The primitive element to use for the power representation. The default is None which uses the field’s default primitive element, primitive_element.

- **sort** (str, optional) – The sorting method for the table, either "power" (default), "poly", "vector", or "int". Sorting by "power" will order the rows of the table by ascending powers of the primitive element. Sorting by any of the others will order the rows in lexicographically-increasing polynomial/vector order, which is equivalent to ascending order of the integer representation.

Returns

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

Return type

str

Examples

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

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

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

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

<table>
<thead>
<tr>
<th>Power</th>
<th>Polynomial</th>
<th>Vector</th>
<th>Integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>[0, 0, 0, 0]</td>
<td>0</td>
</tr>
<tr>
<td>( x^0 )</td>
<td>1</td>
<td>[0, 0, 0, 1]</td>
<td>1</td>
</tr>
<tr>
<td>( x^1 )</td>
<td>x</td>
<td>[0, 0, 1, 0]</td>
<td>2</td>
</tr>
<tr>
<td>( x^2 )</td>
<td>( x^2 )</td>
<td>[0, 1, 0, 0]</td>
<td>4</td>
</tr>
<tr>
<td>( x^3 )</td>
<td>( x^3 )</td>
<td>[1, 0, 0, 0]</td>
<td>8</td>
</tr>
<tr>
<td>( x^4 )</td>
<td>( x + 1 )</td>
<td>[0, 0, 1, 1]</td>
<td>3</td>
</tr>
<tr>
<td>( x^5 )</td>
<td>( x^2 + x )</td>
<td>[0, 1, 1, 0]</td>
<td>6</td>
</tr>
<tr>
<td>( x^6 )</td>
<td>( x^3 + x^2 )</td>
<td>[1, 1, 0, 0]</td>
<td>12</td>
</tr>
<tr>
<td>( x^7 )</td>
<td>( x^3 + x + 1 )</td>
<td>[1, 0, 1, 1]</td>
<td>11</td>
</tr>
</tbody>
</table>

(continues on next page)
Generate a representation table for \( \text{GF}(2^4) \) using a different primitive element \( x^3 + x^2 + x \). Notice, \( \text{ord}(x^3 + x^2 + x) = 15 \).

\begin{verbatim}
In [4]: alpha = GF.primitive_elements[-1]

In [5]: print(GF.repr_table(alpha))
\end{verbatim}

\begin{tabular}{|c|c|c|c|}
\hline
Power & Polynomial & Vector & Integer \\
\hline
0 & 0 & [0, 0, 0, 0] & 0 \\
\hline
\((x^3 + x^2 + x)^0\) & 1 & [0, 0, 0, 1] & 1 \\
\hline
\((x^3 + x^2 + x)^1\) & \(x^3 + x^2 + x\) & [1, 1, 1, 0] & 14 \\
\hline
\((x^3 + x^2 + x)^2\) & \(x^3 + x + 1\) & [1, 0, 1, 1] & 11 \\
\hline
\((x^3 + x^2 + x)^3\) & \(x^3\) & [1, 0, 0, 0] & 8 \\
\hline
\((x^3 + x^2 + x)^4\) & \(x^3 + 1\) & [1, 0, 0, 1] & 9 \\
\hline
\((x^3 + x^2 + x)^5\) & \(x^2 + x + 1\) & [0, 1, 1, 1] & 7 \\
\hline
\((x^3 + x^2 + x)^6\) & \(x^3 + x^2\) & [1, 1, 0, 0] & 12 \\
\hline
\((x^3 + x^2 + x)^7\) & \(x^2\) & [0, 1, 0, 0] & 4 \\
\hline
\((x^3 + x^2 + x)^8\) & \(x^3 + x^2 + 1\) & [1, 1, 0, 1] & 13 \\
\hline
\((x^3 + x^2 + x)^9\) & \(x^3 + x\) & [1, 0, 1, 0] & 10 \\
\hline
\((x^3 + x^2 + x)^{10}\) & \(x^2 + x\) & [0, 1, 1, 0] & 6 \\
\hline
\((x^3 + x^2 + x)^{11}\) & \(x\) & [0, 0, 1, 0] & 2 \\
\hline
\((x^3 + x^2 + x)^{12}\) & \(x^3 + x^2 + x + 1\) & [1, 1, 1, 1] & 15 \\
\hline
\end{tabular}
(continued from previous page)

\[
\begin{array}{c|c|c|c}
(x^3 + x^2 + x)^{13} & x^2 + 1 & [0, 1, 0, 1] & 5 \\
(x^3 + x^2 + x)^{14} & x + 1 & [0, 0, 1, 1] & 3 \\
\end{array}
\]

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

**In [6]:** \(\text{beta} = \text{GF}("x^3 + x^2")\)

**In [7]:** \(\text{print(GF.repr_table(beta))}\)

<table>
<thead>
<tr>
<th>Power</th>
<th>Polynomial</th>
<th>Vector</th>
<th>Integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>[0, 0, 0, 0]</td>
<td>0</td>
</tr>
<tr>
<td>((x^3 + x^2)^0)</td>
<td>1</td>
<td>[0, 0, 0, 1]</td>
<td>1</td>
</tr>
<tr>
<td>((x^3 + x^2)^1)</td>
<td>(x^3 + x^2)</td>
<td>[1, 1, 0, 0]</td>
<td>12</td>
</tr>
<tr>
<td>((x^3 + x^2)^2)</td>
<td>(x^3 + x^2 + x + 1)</td>
<td>[1, 1, 1, 1]</td>
<td>15</td>
</tr>
<tr>
<td>((x^3 + x^2)^3)</td>
<td>(x^3)</td>
<td>[0, 0, 0, 0]</td>
<td>8</td>
</tr>
<tr>
<td>((x^3 + x^2)^4)</td>
<td>(x^3 + x)</td>
<td>[1, 0, 1, 0]</td>
<td>10</td>
</tr>
<tr>
<td>((x^3 + x^2)^5)</td>
<td>1</td>
<td>[0, 0, 0, 1]</td>
<td>1</td>
</tr>
<tr>
<td>((x^3 + x^2)^6)</td>
<td>(x^3 + x^2)</td>
<td>[1, 1, 0, 0]</td>
<td>12</td>
</tr>
<tr>
<td>((x^3 + x^2)^7)</td>
<td>(x^3 + x^2 + x + 1)</td>
<td>[1, 1, 1, 1]</td>
<td>15</td>
</tr>
<tr>
<td>((x^3 + x^2)^8)</td>
<td>(x^3)</td>
<td>[1, 0, 0, 0]</td>
<td>8</td>
</tr>
<tr>
<td>((x^3 + x^2)^9)</td>
<td>(x^3 + x)</td>
<td>[1, 0, 1, 0]</td>
<td>10</td>
</tr>
<tr>
<td>((x^3 + x^2)^{10})</td>
<td>1</td>
<td>[0, 0, 0, 1]</td>
<td>1</td>
</tr>
<tr>
<td>((x^3 + x^2)^{11})</td>
<td>(x^3 + x^2)</td>
<td>[1, 1, 0, 0]</td>
<td>12</td>
</tr>
<tr>
<td>((x^3 + x^2)^{12})</td>
<td>(x^3 + x^2 + x + 1)</td>
<td>[1, 1, 1, 1]</td>
<td>15</td>
</tr>
<tr>
<td>((x^3 + x^2)^{13})</td>
<td>(x^3)</td>
<td>[1, 0, 0, 0]</td>
<td>8</td>
</tr>
<tr>
<td>((x^3 + x^2)^{14})</td>
<td>(x^3 + x)</td>
<td>[1, 0, 1, 0]</td>
<td>10</td>
</tr>
</tbody>
</table>

**property characteristic**

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

```
In [1]: GF = galois.GF(2**8, display="poly")

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

In [3]: a = GF.Random(low=1); a
Out[3]: GF(^4 + ^3 + ^2 + , order=2^8)

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

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

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

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

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

**property default_ufunc_mode**

The default ufunc arithmetic mode for this Galois field.

Examples

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

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

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

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

**property degree**

The prime characteristic’s degree \( m \) of the Galois field \( GF(p^m) \). The degree is a positive integer.
Examples

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

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

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

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

Type int

property display_mode

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

Examples

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

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

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

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

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

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

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

# The display mode is reset after exiting the context manager
But when the primitive element is $\alpha \neq x$ in $\text{GF}(p)[x]$, the polynomial indeterminate used is $x$.

```python
In [8]: GF = galois.GF(2**8, irreducible_poly=galois.Poly.Degrees([8,4,3,1,0]))
In [9]: print(GF.properties)
GF(2^8):
    characteristic: 2
    degree: 8
    order: 256
    irreducible_poly: x^8 + x^4 + x^3 + x + 1
    is_primitive_poly: False
    primitive_element: x + 1
In [10]: a = GF.Random()
In [11]: print(GF.display_mode, a)
int GF(249, order=2^8)
In [12]: with GF.display("poly"):
    ....:    print(GF.display_mode, a)
    ....:
    poly GF(x^7 + x^6 + x^5 + x^4 + x^3 + 1, order=2^8)
In [13]: with GF.display("power"):
    ....:    print(GF.display_mode, a)
    ....:
    power GF(^99, order=2^8)
# The display mode is reset after exiting the context manager
In [14]: print(GF.display_mode, a)
int GF(249, order=2^8)
```

The power representation displays elements as powers of $\alpha$ the primitive element, see `FieldClass.primitive_element`.

```python
In [15]: with GF.display("power"):
    ....:    print(GF.display_mode, a)
    ....:
    power GF(^99, order=2^8)
# The display mode is reset after exiting the context manager
In [16]: print(GF.display_mode, a)
int GF(249, order=2^8)
```

**Type**  
`str`

**property dtypes**

List of valid integer `numpy.dtype` values that are compatible with this Galois field. Creating an array with an unsupported dtype will throw a `TypeError` exception.
Examples

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

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

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

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

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

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

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

**Type** list

**property irreducible_poly**

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

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

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

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

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

Type `galois.Poly`

**property is_extension_field**
Indicates if the field’s order is a prime power.

Examples

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

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

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

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

Type `bool`

**property is_prime_field**
Indicates if the field’s order is prime.

Examples

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

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

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

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

Type  bool

property is_primitive_poly
Indicates whether the irreducible_poly is a primitive polynomial. If so, \( x \) is a primitive element of the Galois field.

Examples

In [1]: GF = galois.GF(2**8, display="poly")
In [2]: GF.irreducible_poly
Out[2]: Poly(x^8 + x^4 + x^3 + x^2 + 1, GF(2))
In [3]: GF.primitive_element
Out[3]: GF(x + 1, order=2^8)

# The irreducible polynomial is a primitive polynomial if the primitive element is a root
In [4]: GF.irreducible_poly(GF.primitive_element, field=GF)
Out[4]: GF(0, order=2^8)
In [5]: GF.is_primitive_poly
Out[5]: True

Here is an example using the GF(2^8) field from AES, which does not use a primitive polynomial.

In [6]: GF = galois.GF(2**8, irreducible_poly=galois.Poly.Degrees([8,4,3,1,0]), display="poly")
In [7]: GF.irreducible_poly
Out[7]: Poly(x^8 + x^4 + x^3 + x + 1, GF(2))
In [8]: GF.primitive_element
Out[8]: GF(x + 1, order=2^8)

# The irreducible polynomial is a primitive polynomial if the primitive element is a root
In [9]: GF.irreducible_poly(GF.primitive_element, field=GF)
Out[9]: GF(x^2 + x, order=2^8)
In [10]: GF.is_primitive_poly
Out[10]: False

Type  bool

property name
The Galois field name.
Examples

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

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

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

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

**Type** `str`

**property order**
The order $p^m$ of the Galois field $GF(p^m)$. The order of the field is also equal to the field’s size.

Examples

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

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

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

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

**Type** `int`

**property prime_subfield**
The prime subfield $GF(p)$ of the extension field $GF(p^m)$.

Examples

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

In [2]: print(galois.GF(2**8).prime_subfield.properties)
```

(continues on next page)
GF(2):
    characteristic: 2
    degree: 1
    order: 2
    irreducible_poly: x + 1
    is_primitive_poly: True
    primitive_element: 1

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

In [4]: print(galois.GF(7**5).prime_subfield.properties)
GF(7):
    characteristic: 7
    degree: 1
    order: 7
    irreducible_poly: x + 4
    is_primitive_poly: True
    primitive_element: 3

Type `galois.FieldClass`

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

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

Examples

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

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

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

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

Type `galois.FieldArray`
**property primitive_elements**

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

**Examples**

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

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

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

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

**property properties**

A formatted string displaying relevant properties of the Galois field.

**Examples**

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

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

(continues on next page)
primitive_element: x

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

In [4]: GF = galois.GF(7**5); print(GF.properties)
GF(7^5):
    characteristic: 7
    degree: 5
    order: 16807
    irreducible_poly: x^5 + x + 4
    is_primitive_poly: True
    primitive_element: x

Type str

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

Examples

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

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

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

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

Type str

property ufunc_modes
All supported ufunc modes for this Galois field array class.
Examples

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

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

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

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

Type  list

Pre-made Galois field classes

| `GF2(array[, dtype, copy, order, ndmin])` | An array over GF(2). |

**galois.GF2**

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

An array over GF(2).

This class is a pre-generated `galois.FieldArray` subclass generated with `galois.GF(2)` and is included in the API for convenience. See `galois.FieldArray` and `galois.FieldClass` for more complete documentation and examples.

Examples

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

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

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

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

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

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

(continues on next page)
irreducible_poly: x + 1
is_primitive_poly: True
primitive_element: 1

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

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

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

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

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

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

classmethod Elements(dtype=None)

Creates a 1-D Galois field array of the field’s elements \{0, ..., p^n - 1\}.

**Parameters**

dtype (*numpy.dtype*, *optional*) – The `numpy.dtype` of the array elements.

The default is `None` which represents the smallest unsigned dtype for this class, i.e. the first

**Returns**

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

**Return type**

`galois.FieldArray`

**Examples**

In [10]: GF = galois.GF(2**4)

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

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

# Permanently set the display mode to "poly"
In [12]: GF.display("poly");
In [13]: GF.Elements()
Out[13]:
GF([0, 1, , + 1, ^2, ^2 + 1, ^2 + , ^2 + + 1, ^3, ^3 + 1,
     ^3 + , ^3 + + 1, ^3 + ^2, ^3 + ^2 + 1, ^3 + ^2 + ,
     ^3 + ^2 + + 1], order=2^4)

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

# Reset the display mode to "int"
In [15]: GF.display();

classmethod Identity(size, dtype=None)

Creates an \( n \times n \) Galois field identity matrix.

Parameters

- **size** (*int*) – The size \( n \) along one axis of the matrix. The resulting array has shape \((size, size)\).

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

Returns

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

Return type *galois.FieldArray*

Examples

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

In [17]: GF.Identity(4)
Out[17]:
GF([[1, 0, 0, 0],
    [0, 1, 0, 0],
    [0, 0, 1, 0],
    [0, 0, 0, 1]], order=31)

classmethod Ones(shape, dtype=None)

Creates a Galois field array with all ones.

Parameters

- **shape** (*int, tuple*) – A numpy-compliant shape tuple, see *numpy.ndarray.shape*. An empty tuple () represents a scalar. A single integer or 1-tuple, e.g. \( N \) or \((N,)\), represents the size of a 1-D array. A 2-tuple, e.g. \((M,N)\), represents a 2-D array with each element indicating the size in each dimension.

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

**Return type**  `galois.FieldArray`

**Examples**

```python
In [18]: GF = galois.GF(31)
In [19]: GF.Oones((2,5))
Out[19]:
GF([[1, 1, 1, 1, 1],
     [1, 1, 1, 1, 1]], order=31)
```

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

Creates a Galois field array with random field elements.

**Parameters**

- **shape** (*int, tuple*) – A numpy-compliant shape tuple, see `numpy.ndarray.shape`. An empty tuple () represents a scalar. A single integer or 1-tuple, e.g. N or (N,), represents the size of a 1-D array. A 2-tuple, e.g. (M,N), represents a 2-D array with each element indicating the size in each dimension.

- **low** (*int, optional*) – The lowest value (inclusive) of a random field element in its integer representation. The default is 0.

- **high** (*int, optional*) – The highest value (exclusive) of a random field element in its integer representation. The default is None which represents the field’s order \(p^m\).

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

**Returns**  A Galois field array of random field elements.

**Return type**  `galois.FieldArray`

**Examples**

```python
In [20]: GF = galois.GF(31)
In [21]: GF.Random((2,5))
Out[21]:
GF([[17, 29, 20, 28,  6],
     [14, 18, 11, 21, 26]], order=31)
```

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

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

**Parameters**

- **start** (*int*) – The starting Galois field value (inclusive) in its integer representation.

- **stop** (*int*) – The stopping Galois field value (exclusive) in its integer representation.

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

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

**Return type** *galois.FieldArray*

### Examples

In [22]: GF = galois.GF(31)
In [23]: GF.Range(10, 20)
Out[23]: GF([10, 11, 12, 13, 14, 15, 16, 17, 18, 19], order=31)

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

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

**Parameters**

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

**Returns** The \( m \times n \) Vandermonde matrix.

**Return type** *galois.FieldArray*

### Examples

In [24]: GF = galois.GF(2**3)
In [25]: a = GF.primitive_element
In [26]: V = GF.Vandermonde(a, 7, 7)
In [27]: with GF.display("power"):
   ....:     print(V)
   ....:
GF([[1, 1, 1, 1, 1, 1, 1],
    [1, 1^2, 1^3, 1^4, 1^5, 1^6],
    [1, 1^2, 1^4, 1^6, 1^3, 1^5],
    [1, 1^3, 1^6, 1^2, 1^5, 1^4],
    [1, 1^4, 1^6, 1^2, 1^3, 1^5],
    [1, 1^5, 1^3, 1^6, 1^4, 1^2],
    [1, 1^6, 1^5, 1^4, 1^3, 1^2]], order=2^3)

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

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

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

- **array (array_like)** – The input array with field elements in \( \text{GF}(p) \) to be converted to a Galois field array in \( \text{GF}(p^m) \). The last dimension of the input array must be \( m \). An input array with shape \((n_1, n_2, m)\) has output shape \((n_1, n_2)\). By convention, the vectors are ordered from highest degree to 0-th degree.

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

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

**Return type**  `galois.FieldArray`

**Examples**

```python
In [28]: GF = galois.GF(2**6)
In [29]: vec = galois.GF2.Random((3,6)); vec
Out[29]: GF([[1, 1, 0, 1, 1, 0],
         [1, 0, 1, 0, 1, 1],
         [1, 1, 0, 0, 0, 0]], order=2)
In [30]: a = GF.Vector(vec); a
Out[30]: GF([54, 43, 48], order=2^6)
In [31]: with GF.display("poly"):
    ....:     print(a)
    ....:
GF([^5 + ^4 + ^2 + , ^5 + ^3 + + 1, ^5 + ^4], order=2^6)
In [32]: a.vector()
Out[32]: GF([[1, 1, 0, 1, 1, 0],
         [1, 0, 1, 0, 1, 1],
         [1, 1, 0, 0, 0, 0]], order=2)
```

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

Creates a Galois field array with all zeros.

**Parameters**

- **shape (int, tuple)** – A numpy-compliant `shape` tuple, see `numpy.ndarray.shape`. An empty tuple () represents a scalar. A single integer or 1-tuple, e.g. \( N \) or \( (N,) \), represents the size of a 1-D array. A 2-tuple, e.g. \( (M, N) \), represents a 2-D array with each element indicating the size in each dimension.

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

**Returns**  A Galois field array of zeros.

**Return type**  `galois.FieldArray`
Examples

```plaintext
In [33]: GF = galois.GF(31)
In [34]: GF.Zeros((2,5))
Out[34]:
GF([[0, 0, 0, 0, 0],
     [0, 0, 0, 0, 0]], order=31)
```

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

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

Parameters

- **array** *(int, str, tuple, list, numpy.ndarray, galois.FieldArray)* – The input array-like object to be converted to a Galois field array. See the examples section for demonstrations of array creation using each input type. See see `galois.FieldClass.display()` and `galois.FieldClass.display_mode` for a description of the “integer” and “polynomial” representation of Galois field elements.
  - int: A single integer, which is the “integer representation” of a Galois field element, creates a 0-D array.
  - str: A single string, which is the “polynomial representation” of a Galois field element, creates a 0-D array.
  - tuple, list: A list or tuple (or nested lists/tuples) of ints or strings (which can be mix-and-matched) creates an array of Galois field elements from their integer or polynomial representations. *numpy.ndarray, galois.FieldArray*: An array of ints creates a copy of the array over this specific field.
- **dtype** *(numpy.dtype, optional)* – The `numpy.dtype` of the array elements. The default is `None` which represents the smallest unsigned dtype for this class, i.e. the first element in `galois.FieldClass.dtypes`.
- **copy** *(bool, optional)* – The copy keyword argument from `numpy.array()`. The default is `True` which makes a copy of the input array.
- **order** *(str, optional)* – The order keyword argument from `numpy.array()`. Valid values are “K” (default), “A”, “C”, or “F”.
- **ndmin** *(int, optional)* – The ndmin keyword argument from `numpy.array()`. The minimum number of dimensions of the output. The default is 0.

Returns

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

Return type  
galois.FieldArray
7.1.2 Prime field functions

Primitive roots

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>primitive_root(n[, start, stop, reverse])</code></td>
<td>Finds the smallest primitive root modulo $n$.</td>
</tr>
<tr>
<td><code>primitive_roots(n[, start, stop, reverse])</code></td>
<td>Finds all primitive roots modulo $n$.</td>
</tr>
<tr>
<td><code>is_primitive_root(g, n)</code></td>
<td>Determines if $g$ is a primitive root modulo $n$.</td>
</tr>
</tbody>
</table>

**galois.primitive_root**

galois.primitive_root($n$, $start=1$, $stop=None$, $reverse=False$)

Finds the smallest primitive root modulo $n$.

**Parameters**

- $n$ (**int**) – A positive integer.
- $start$ (**int**, **optional**) – Starting value (inclusive) in the search for a primitive root. The default is 1. The resulting primitive root, if found, will be $start \leq g < stop$.
- $stop$ (**int**, **optional**) – Stopping value (exclusive) in the search for a primitive root. The default is `None` which corresponds to $n$. The resulting primitive root, if found, will be $start \leq g < stop$.
- $reverse$ (**bool**, **optional**) – Search for a primitive root in reverse order, i.e. find the largest primitive root first. Default is `False`.

**Returns**
The smallest primitive root modulo $n$. Returns `None` if no primitive roots exist.

**Return type**
**int**

**Notes**

$g$ is a primitive root if the totatives of $n$, the positive integers $1 \leq a < n$ that are coprime with $n$, can be generated by powers of $g$. Alternatively said, $g$ is a primitive root modulo $n$ if and only if $g$ is a generator of the multiplicative group of integers modulo $n$, $(\mathbb{Z}/n\mathbb{Z})^\times = \{g^0, g^1, g^2, \ldots, g^{\phi(n)-1}\}$ where $\phi(n)$ is order of the group. If $(\mathbb{Z}/n\mathbb{Z})^\times$ is cyclic, the number of primitive roots modulo $n$ is given by $\phi(\phi(n))$.

**References**

Examples

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

# n is of type 2*p^k, which is cyclic
In [1]: n = 14

In [2]: galois.is_cyclic(n)
Out[2]: True

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

# Euler's totient function counts the "totatives", positive integers coprime with n
In [4]: phi = galois.euler_phi(n); phi
Out[4]: 6

In [5]: len(Znx) == phi
Out[5]: True

# The primitive roots are the elements in Znx that multiplicatively generate the group
In [6]: for a in Znx:
    ...:     span = set([pow(a, i, n) for i in range(1, phi + 1)])
    ...:     primitive_root = span == Znx
    ...:     print("Element: {:<2d}, Span: {:<20}, Primitive root: {}".format(a,
                       str(span), primitive_root))
    ...
Element:  1, Span: {1}, Primitive root: False
Element:  3, Span: {1, 3, 5, 9, 11, 13}, Primitive root: True
Element:  5, Span: {1, 3, 5, 9, 11, 13}, Primitive root: True
Element:  9, Span: {9, 11, 1}, Primitive root: False
Element: 11, Span: {9, 11, 1}, Primitive root: False
Element: 13, Span: {1, 13}, Primitive root: False

# Find the smallest primitive root
In [7]: galois.primitive_root(n)
Out[7]: 3

# Find all primitive roots
In [8]: roots = galois.primitive_roots(n); roots
Out[8]: [3, 5]

# Euler's totient function (() counts the primitive roots of n
In [9]: len(roots) == galois.euler_phi(phi)
Out[9]: True

A counterexample is \(n = 15 = 3 \cdot 5\), which doesn’t fit the condition for cyclicness. 
\((\mathbb{Z}/15\mathbb{Z})^\times = \{1, 2, 4, 7, 8, 11, 13, 14\}\).

# n is of type p1^k1 * p2^k2, which is not cyclic
In [10]: n = 15

(continues on next page)
In [11]: galois.is_cyclic(n)
Out[11]: False

# The congruence class coprime with n
In [12]: Znx = set([a for a in range(1, n) if math.gcd(n, a) == 1]); Znx
Out[12]: {1, 2, 4, 7, 8, 11, 13, 14}

# Euler's totient function counts the "totatives", positive integers coprime with n
In [13]: phi = galois.euler_phi(n); phi
Out[13]: 8
In [14]: len(Znx) == phi
Out[14]: True

# The primitive roots are the elements in Znx that multiplicatively generate the group
In [15]: for a in Znx:
   ....:     span = set([pow(a, i, n) for i in range(1, phi + 1)])
   ....:     primitive_root = span == Znx
   ....:     print("Element: {:2d}, Span: {:<13}, Primitive root: {}").format(a,
   ....:                        str(span), primitive_root))
   ....:
Element:  1, Span: {1} , Primitive root: False
Element:  2, Span: {8, 1, 2, 4} , Primitive root: False
Element:  4, Span: {1, 4} , Primitive root: False
Element:  7, Span: {1, 4, 13, 7}, Primitive root: False
Element:  8, Span: {8, 1, 2, 4} , Primitive root: False
Element: 11, Span: {1, 11} , Primitive root: False
Element: 13, Span: {1, 4, 13, 7}, Primitive root: False
Element: 14, Span: {1, 14} , Primitive root: False

# Find the smallest primitive root
In [16]: galois.primitive_root(n)

# Find all primitive roots
In [17]: roots = galois.primitive_roots(n); roots
Out[17]: []

# Note the max order of any element is 4, not 8, which is Carmichael's lambda function
In [18]: galois.carmichael_lambda(n)
Out[18]: 4

The algorithm is also efficient for very large n.

In [19]: n = 1000000000000000035000061
In [20]: galois.primitive_root(n)
Out[20]: 7

7.1. Galois Fields
galois

**galois.primitive_roots**

`galois.primitive_roots(n, start=1, stop=None, reverse=False)`

Finds all primitive roots modulo `n`.

**Parameters**

- `n` *(int)* – A positive integer.
- `start` *(int, optional)* – Starting value (inclusive) in the search for a primitive root. The default is 1. The resulting primitive roots, if found, will be `start <= x < stop`.
- `stop` *(int, optional)* – Stopping value (exclusive) in the search for a primitive root. The default is `None` which corresponds to `n`. The resulting primitive roots, if found, will be `start <= x < stop`.
- `reverse` *(bool, optional)* – List all primitive roots in descending order, i.e. largest to smallest. Default is `False`.

**Returns**

All the positive primitive `n`-th roots of unity, `x`.

**Return type** *list*

**Notes**

`g` is a primitive root if the totatives of `n`, the positive integers `1 <= a < n` that are coprime with `n`, can be generated by powers of `g`. Alternatively said, `g` is a primitive root modulo `n` if and only if `g` is a generator of the multiplicative group of integers modulo `n`, `\((\mathbb{Z}/n\mathbb{Z})^\times = \{g^0, g^1, g^2, \ldots, g^{\phi(n)-1}\}\)` where `\phi(n)` is order of the group. If `\((\mathbb{Z}/n\mathbb{Z})^\times\)` is cyclic, the number of primitive roots modulo `n` is given by `\phi(\phi(n))`.

**References**


**Examples**

The elements of `\((\mathbb{Z}/n\mathbb{Z})^\times\)` are the positive integers less than `n` that are coprime with `n`. For example, `\((\mathbb{Z}/14\mathbb{Z})^\times = \{1, 3, 5, 9, 11, 13\}\)`.  

```python
# n is of type 2^p*k, which is cyclic
In [1]: n = 14

In [2]: galois.is_cyclic(n)
Out[2]: True

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

# Euler's totient function counts the "totatives", positive integers coprime with n
In [4]: phi = galois.euler_phi(n); phi
Out[4]: 6
```
In [5]: \texttt{len(Znx) == phi}
Out[5]: True

# The primitive roots are the elements in Znx that multiplicatively generate the group
In [6]: \texttt{for a in Znx:}
   ...
   \texttt{span = set([pow(a, i, n) for i in range(1, phi + 1)])}
   ...
   \texttt{primitive_root = span == Znx}
   ...
   \texttt{print("Element: {:2d}, Span: {:<20}, Primitive root: {}".format(a, str(span), primitive_root))}
   ...

Element: 1, Span: \{1\}, Primitive root: False
Element: 3, Span: \{1, 3, 5, 9, 11, 13\}, Primitive root: True
Element: 5, Span: \{1, 3, 5, 9, 11, 13\}, Primitive root: True
Element: 9, Span: \{9, 11, 1\}, Primitive root: False
Element: 11, Span: \{9, 11, 1\}, Primitive root: False
Element: 13, Span: \{1, 13\}, Primitive root: False

# Find the smallest primitive root
In [7]: \texttt{galois.primitive_root(n)}
Out[7]: 3

# Find all primitive roots
In [8]: \texttt{roots = galois.primitive_roots(n); roots}
Out[8]: [3, 5]

# Euler's totient function \((\phi(n))\) counts the primitive roots of \(n\)
In [9]: \texttt{len(roots) == galois.euler_phi(phi)}
Out[9]: True

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

# \(n\) is of type \(p^1k^1 \cdot p^2k^2\), which is not cyclic
In [10]: \texttt{n = 15}

In [11]: \texttt{galois.is_cyclic(n)}
Out[11]: False

# The congruence class coprime with \(n\)
In [12]: \texttt{Znx = set([a for a in range(1, n) if math.gcd(n, a) == 1]); Znx}
Out[12]: \{1, 2, 4, 7, 8, 11, 13, 14\}

# Euler's totient function counts the "totatives", positive integers coprime with \(n\)
In [13]: \texttt{phi = galois.euler_phi(n); phi}
Out[13]: 8

In [14]: \texttt{len(Znx) == phi}
Out[14]: True

# The primitive roots are the elements in Znx that multiplicatively generate the group

(continues on next page)
In [15]: for a in Znx:
    ...:    span = set([pow(a, i, n) for i in range(1, phi + 1)])
    ...:    primitive_root = span == Znx
    ...:    print("Element: {:2d}, Span: {:<13}, Primitive root: {}".format(a,
    ...:                        str(span), primitive_root))
    ...:
Element: 1, Span: {1} , Primitive root: False
Element: 2, Span: {8, 1, 2, 4} , Primitive root: False
Element: 4, Span: {1, 4} , Primitive root: False
Element: 7, Span: {1, 4, 13, 7}, Primitive root: False
Element: 8, Span: {8, 1, 2, 4} , Primitive root: False
Element: 11, Span: {1, 11} , Primitive root: False
Element: 13, Span: {1, 4, 13, 7}, Primitive root: False
Element: 14, Span: {1, 14} , Primitive root: False

# Find the smallest primitive root
In [16]: galois.primitive_root(n)

# Find all primitive roots
In [17]: roots = galois.primitive_roots(n); roots
Out[17]: []

# Note the max order of any element is 4, not 8, which is Carmichael's lambda function
In [18]: galois.carmichael_lambda(n)
Out[18]: 4

**galois.is_primitive_root**

galois.is_primitive_root(g, n)

Determines if $g$ is a primitive root modulo $n$.

**Parameters**

- $g$ (int) – A positive integer that may be a primitive root modulo $n$.
- $n$ (int) – A positive integer.

**Returns** True if $g$ is a primitive root modulo $n$.

**Return type** bool

**Notes**

$g$ is a primitive root if the totatives of $n$, the positive integers $1 \leq a < n$ that are coprime with $n$, can be generated by powers of $g$. Alternatively said, $g$ is a primitive root modulo $n$ if and only if $g$ is a generator of the multiplicative group of integers modulo $n$, $(\mathbb{Z}/n\mathbb{Z})^\times = \{g^0, g^1, g^2, \ldots, g^{\phi(n)-1}\}$ where $\phi(n)$ is order of the group. If $(\mathbb{Z}/n\mathbb{Z})^\times$ is cyclic, the number of primitive roots modulo $n$ is given by $\phi(\phi(n))$. 
Examples

```
In [1]: galois.is_primitive_root(2, 7)
Out[1]: False

In [2]: galois.is_primitive_root(3, 7)
Out[2]: True

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

7.1.3 Extension field functions

Irreducible polynomials

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>irreducible_poly</code></td>
<td>Returns a monic irreducible polynomial ( f(x) ) over ( \text{GF}(q) ) with degree ( m ).</td>
</tr>
<tr>
<td><code>irreducible_polys</code></td>
<td>Returns all monic irreducible polynomials ( f(x) ) over ( \text{GF}(q) ) with degree ( m ).</td>
</tr>
<tr>
<td><code>is_irreducible</code></td>
<td>Determines whether the polynomial ( f(x) ) over ( \text{GF}(p^m) ) is irreducible.</td>
</tr>
</tbody>
</table>

**galois.irreducible_poly**

```
galois.irreducible_poly(order, degree[, method='min'])
```

Returns a monic irreducible polynomial \( f(x) \) over \( \text{GF}(q) \) with degree \( m \).

**Parameters**

- `order` *(int)* – The prime power \( q \) of the field \( \text{GF}(q) \) that the polynomial is over.
- `degree` *(int)* – The degree \( m \) of the desired irreducible polynomial.
- `method` *(str, optional)* – The search method for finding the irreducible polynomial.
  - "min" (default): Returns the lexicographically-minimal monic irreducible polynomial.
  - "max": Returns the lexicographically-maximal monic irreducible polynomial.
  - "random": Returns a randomly generated degree-\( m \) monic irreducible polynomial.

**Returns** The degree-\( m \) monic irreducible polynomial over \( \text{GF}(q) \).

**Return type** `galois.Poly`
Notes

If $f(x)$ is an irreducible polynomial over $\mathbb{GF}(q)$ and $\alpha \in \mathbb{GF}(q) \setminus \{0\}$, then $\alpha \cdot f(x)$ is also irreducible. In addition to other applications, $f(x)$ produces the field extension $\mathbb{GF}(q^m)$ of $\mathbb{GF}(q)$.

Examples

The lexicographically-minimal, monic irreducible polynomial over $\mathbb{GF}(7)$ with degree 5.

```python
In [1]: p = galois.irreducible_poly(7, 5); p
Out[1]: Poly(x^5 + x + 3, GF(7))
In [2]: galois.is_irreducible(p)
Out[2]: True
```

Irreducible polynomials scaled by non-zero field elements are also irreducible.

```python
In [3]: GF = galois.GF(7)
In [4]: galois.is_irreducible(p * GF(3))
Out[4]: True
```

A random, monic irreducible polynomial over $\mathbb{GF}(7^2)$ with degree 3.

```python
In [5]: p = galois.irreducible_poly(7**2, 3, method="random"); p
Out[5]: Poly(x^3 + 38x^2 + 31x + 13, GF(7^2))
In [6]: galois.is_irreducible(p)
Out[6]: True
```

galois.irreducible_polys

galois.irreducible_polys(order, degree)

Returns all monic irreducible polynomials $f(x)$ over $\mathbb{GF}(q)$ with degree $m$.

Parameters

- **order** (*int*) – The prime power order $q$ of the field $\mathbb{GF}(q)$ that the polynomial is over.
- **degree** (*int*) – The degree $m$ of the desired irreducible polynomial.

Returns All degree-$m$ monic irreducible polynomials over $\mathbb{GF}(q)$.

Return type  list
Notes

If $f(x)$ is an irreducible polynomial over $\mathbb{GF}(q)$ and $\alpha \in \mathbb{GF}(q) \setminus \{0\}$, then $\alpha \cdot f(x)$ is also irreducible. In addition to other applications, $f(x)$ produces the field extension $\mathbb{GF}(q^m)$ of $\mathbb{GF}(q)$.

Examples

All monic irreducible polynomials over $\mathbb{GF}(2)$ with degree 5.

```python
In [1]: galois.irreducible_polys(2, 5)
Out[1]:
[Poly(x^5 + x^2 + 1, GF(2)),
 Poly(x^5 + x^3 + 1, GF(2)),
 Poly(x^5 + x^3 + x^2 + x + 1, GF(2)),
 Poly(x^5 + x^4 + x^2 + x + 1, GF(2)),
 Poly(x^5 + x^4 + x^3 + x + 1, GF(2)),
 Poly(x^5 + x^4 + x^3 + x^2 + 1, GF(2))]
```

All monic irreducible polynomials over $\mathbb{GF}(3^2)$ with degree 2.

```python
In [2]: galois.irreducible_polys(3**2, 2)
Out[2]:
[Poly(x^2 + 3, GF(3^2)),
 Poly(x^2 + 5, GF(3^2)),
 Poly(x^2 + 6, GF(3^2)),
 Poly(x^2 + 7, GF(3^2)),
 Poly(x^2 + x + 3, GF(3^2)),
 Poly(x^2 + x + 4, GF(3^2)),
 Poly(x^2 + x + 7, GF(3^2)),
 Poly(x^2 + x + 8, GF(3^2)),
 Poly(x^2 + 2x + 3, GF(3^2)),
 Poly(x^2 + 2x + 4, GF(3^2)),
 Poly(x^2 + 2x + 7, GF(3^2)),
 Poly(x^2 + 2x + 8, GF(3^2)),
 Poly(x^2 + 3x + 1, GF(3^2)),
 Poly(x^2 + 3x + 2, GF(3^2)),
 Poly(x^2 + 3x + 6, GF(3^2)),
 Poly(x^2 + 3x + 7, GF(3^2)),
 Poly(x^2 + 4x + 4, GF(3^2)),
 Poly(x^2 + 4x + 5, GF(3^2)),
 Poly(x^2 + 4x + 6, GF(3^2)),
 Poly(x^2 + 4x + 8, GF(3^2)),
 Poly(x^2 + 5x + 1, GF(3^2)),
 Poly(x^2 + 5x + 2, GF(3^2)),
 Poly(x^2 + 5x + 3, GF(3^2)),
 Poly(x^2 + 5x + 5, GF(3^2)),
 Poly(x^2 + 5x + 6, GF(3^2)),
 Poly(x^2 + 5x + 7, GF(3^2)),
 Poly(x^2 + 5x + 8, GF(3^2)),
 Poly(x^2 + 6x + 1, GF(3^2)),
 Poly(x^2 + 6x + 2, GF(3^2)),
 Poly(x^2 + 6x + 6, GF(3^2)),
 Poly(x^2 + 6x + 7, GF(3^2)),
 Poly(x^2 + 7x + 1, GF(3^2)),
 Poly(x^2 + 7x + 2, GF(3^2)),
 Poly(x^2 + 7x + 3, GF(3^2)),
 Poly(x^2 + 7x + 6, GF(3^2)),
 Poly(x^2 + 7x + 7, GF(3^2)),
 Poly(x^2 + 7x + 8, GF(3^2)),
 Poly(x^2 + 8x + 1, GF(3^2)),
 Poly(x^2 + 8x + 2, GF(3^2)),
 Poly(x^2 + 8x + 3, GF(3^2)),
 Poly(x^2 + 8x + 5, GF(3^2)),
 Poly(x^2 + 8x + 6, GF(3^2)),
 Poly(x^2 + 8x + 7, GF(3^2)),
 Poly(x^2 + 8x + 8, GF(3^2))]
```
```python
Poly(x^2 + 7x + 5, GF(3^2)),
Poly(x^2 + 8x + 4, GF(3^2)),
Poly(x^2 + 8x + 5, GF(3^2)),
Poly(x^2 + 8x + 6, GF(3^2)),
Poly(x^2 + 8x + 8, GF(3^2))]
```

galois.is_irreducible
galois.is_irreducible(poly)

Determines whether the polynomial \( f(x) \) over \( GF(p^m) \) is irreducible.

**Parameters**

- **poly** (galois.Poly) – A polynomial \( f(x) \) over \( GF(p^m) \).

**Returns**

- **True** if the polynomial is irreducible.

**Return type**

- bool

**Notes**

A polynomial \( f(x) \in GF(p^m)[x] \) is **reducible** over \( GF(p^m) \) if it can be represented as \( f(x) = g(x)h(x) \) for some \( g(x), h(x) \in GF(p^m)[x] \) 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 \( GF(q) \) for prime power \( q \) is irreducible if and only if \( f(x) \mid (x^{q^m} - x) \) and \( \gcd(f(x), x^{q^m} - x) = 1 \) for \( 1 \leq i \leq k \), where \( m_i = m/p_i \) for the \( k \) prime divisors \( p_i \) of \( m \).

**References**

- Section 4.5.1 from https://cacr.uwaterloo.ca/hac/about/chap4.pdf

**Examples**

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

# f(x) has no roots in GF(2), a necessary but not sufficient condition of being irreducible
in [2]: f.roots()
Out[2]: GF([], order=2)

in [3]: galois.is_irreducible(f)
Out[3]: True
```
In [4]: g = galois.irreducible_poly(2**4, 2, method="random"); g
Out[4]: Poly(x^2 + 2x + 2, GF(2^4))

In [5]: h = galois.irreducible_poly(2**4, 3, method="random"); h
Out[5]: Poly(x^3 + 4x^2 + 3x + 1, GF(2^4))

In [6]: f = g * h; f
Out[6]: Poly(x^5 + 6x^4 + 9x^3 + 15x^2 + 4x + 2, GF(2^4))

In [7]: galois.is_irreducible(f)
Out[7]: False

### Primitive polynomials

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>primitive_poly</td>
<td>Returns a monic primitive polynomial $f(x)$ over GF($q$) with degree $m$.</td>
</tr>
<tr>
<td>primitive_polys</td>
<td>Returns all monic primitive polynomials $f(x)$ over GF($q$) with degree $m$.</td>
</tr>
<tr>
<td>conway_poly</td>
<td>Returns the Conway polynomial $C_{p,m}(x)$ over GF($p$) with degree $m$.</td>
</tr>
<tr>
<td>matlab_primitive_poly</td>
<td>Returns Matlab’s default primitive polynomial $f(x)$ over GF($p$) with degree $m$.</td>
</tr>
<tr>
<td>is_primitive</td>
<td>Determines whether the polynomial $f(x)$ over GF($q$) is primitive.</td>
</tr>
</tbody>
</table>

#### galois.primitive_poly

**galois.primitive_poly**(order, degree, method='min')

Returns a monic primitive polynomial $f(x)$ over GF($q$) with degree $m$.

**Parameters**

- **order** (*int*) – The prime power order $q$ of the field GF($q$) that the polynomial is over.
- **degree** (*int*) – The degree $m$ of the desired primitive polynomial.
- **method** (*str, optional*) – The search method for finding the primitive polynomial.
  
  - "min" (default): Returns the lexicographically-minimal monic primitive polynomial.
  
  - "max": Returns the lexicographically-maximal monic primitive polynomial.
  
  - "random": Returns a randomly generated degree-$m$ monic primitive polynomial.

**Returns** The degree-$m$ monic primitive polynomial over GF($q$).

**Return type** *galois.Poly*
Notes

In addition to other applications, $f(x)$ produces the field extension $GF(q^m)$ of $GF(q)$. Since $f(x)$ is primitive, $x$ is a primitive element $\alpha$ of $GF(q^m)$ such that $GF(q^m) = \{0, 1, \alpha, \alpha^2, \ldots, \alpha^{q^m-2}\}$.

Examples

Notice `galois.primitive_poly()` returns the lexicographically-minimal primitive polynomial, whereas `galois.conway_poly()` returns the lexicographically-minimal primitive polynomial that is consistent with smaller Conway polynomials, which is not necessarily the same.

```
In [1]: galois.primitive_poly(2, 4)
Out[1]: Poly(x^4 + x + 1, GF(2))

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

```
In [3]: galois.primitive_poly(7, 10)
Out[3]: Poly(x^10 + 5x^2 + x + 5, GF(7))

In [4]: galois.conway_poly(7, 10)
Out[4]: Poly(x^10 + x^6 + x^5 + 4x^4 + x^3 + 2x^2 + 3x + 3, GF(7))
```

galois.primitive_polys

galois.primitive_polys(order, degree)

Returns all monic primitive polynomials $f(x)$ over $GF(q)$ with degree $m$.

Parameters

- `order (int)`: The prime order $q$ of the field $GF(q)$ that the polynomial is over.
- `degree (int)`: The degree $m$ of the desired primitive polynomial.

Returns All degree-$m$ monic primitive polynomials over $GF(q)$.

Return type list

Notes

In addition to other applications, $f(x)$ produces the field extension $GF(q^m)$ of $GF(q)$. Since $f(x)$ is primitive, $x$ is a primitive element $\alpha$ of $GF(q^m)$ such that $GF(q^m) = \{0, 1, \alpha, \alpha^2, \ldots, \alpha^{q^m-2}\}$.

Examples

All monic primitive polynomials over $GF(2)$ with degree 5.

```
In [1]: galois.primitive_polys(2, 5)
Out[1]:
[Poly(x^5 + x^2 + 1, GF(2)),
 Poly(x^5 + x^3 + 1, GF(2)),
 Poly(x^5 + x^3 + x^2 + x + 1, GF(2)),
 Poly(x^5 + x^3 + x^2 + 1, GF(2)),
 Poly(x^5 + x^3 + x^2 + x + 1, GF(2))]
```
Poly(x^5 + x^4 + x^2 + x + 1, GF(2)),
Poly(x^5 + x^4 + x^3 + x + 1, GF(2)),
Poly(x^5 + x^4 + x^3 + x^2 + 1, GF(2))

All monic primitive polynomials over GF(3^2) with degree 2.

In [2]: galois.primitive_polys(3**2, 2)
Out[2]:
[Poly(x^2 + x + 3, GF(3^2)),
Poly(x^2 + x + 7, GF(3^2)),
Poly(x^2 + 2x + 3, GF(3^2)),
Poly(x^2 + 2x + 7, GF(3^2)),
Poly(x^2 + 3x + 6, GF(3^2)),
Poly(x^2 + 3x + 7, GF(3^2)),
Poly(x^2 + 4x + 5, GF(3^2)),
Poly(x^2 + 4x + 6, GF(3^2)),
Poly(x^2 + 5x + 3, GF(3^2)),
Poly(x^2 + 5x + 5, GF(3^2)),
Poly(x^2 + 6x + 6, GF(3^2)),
Poly(x^2 + 6x + 7, GF(3^2)),
Poly(x^2 + 7x + 3, GF(3^2)),
Poly(x^2 + 7x + 5, GF(3^2)),
Poly(x^2 + 8x + 5, GF(3^2)),
Poly(x^2 + 8x + 6, GF(3^2))]

galois.conway_poly

galois.conway_poly(characteristic, degree)
    Returns the Conway polynomial \(C_{p,m}(x)\) over GF\(p\) with degree \(m\).

Parameters
    * characteristic (int) – The prime characteristic \(p\) of the field GF\(p\) that the polynomial
      is over.
    * degree (int) – The degree \(m\) of the Conway polynomial.

Returns
    The degree-\(m\) Conway polynomial \(C_{p,m}(x)\) over GF\(p\).

Return type galois.Poly

Raises LookupError – If the Conway polynomial \(C_{p,m}(x)\) is not found in Frank Luebeck’s database.

Notes

A Conway polynomial is a an irreducible and primitive polynomial over GF\(p\) that provides a standard representation of 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.

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

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

Notice `galois.primitive_poly()` returns the lexicographically-minimal primitive polynomial, where `galois.conway_poly()` returns the lexicographically-minimal primitive polynomial that is consistent with smaller Conway polynomials, which is not necessarily the same.

```
In [1]: galois.primitive_poly(2, 4)
Out[1]: Poly(x^4 + x + 1, GF(2))

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

```
In [3]: galois.primitive_poly(7, 10)
Out[3]: Poly(x^10 + 5x^2 + x + 5, GF(7))

In [4]: galois.conway_poly(7, 10)
Out[4]: Poly(x^10 + x^6 + x^5 + 4x^4 + x^3 + 2x^2 + 3x + 3, GF(7))
```

galois.matlab_primitive_poly

`galois.matlab_primitive_poly(characteristic, degree)`

Returns Matlab’s default primitive polynomial $f(x)$ over GF(p) with degree $m$.

Parameters

- **characteristic** (`int`) – The prime characteristic $p$ of the field 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 GF($p$).

Return type `galois.Poly`

Notes

This function returns the same result as Matlab’s `gfprimdf(m, p)`. Matlab uses the primitive polynomial with minimum terms (equivalent to `galois.primitive_poly(p, m, method="min-terms")`) as the default... mostly. There are three notable exceptions:

1. GF(2$^7$) uses $x^7 + x^3 + 1$, not $x^7 + x + 1$.
2. GF(2$^{14}$) uses $x^{14} + x^{10} + x^9 + x + 1$, not $x^{14} + x^5 + x^3 + x + 1$.
3. GF(2$^{16}$) uses $x^{16} + x^{12} + x^3 + x + 1$, not $x^{16} + x^5 + x^3 + x^2 + 1$. 
galois

References

• S. Lin and D. Costello. Error Control Coding. Table 2.7.

**Warning:** This has been tested for all the GF($2^m$) fields for $2 \leq m \leq 16$ (Matlab doesn’t support larger than 16). And it has been spot-checked for GF($p^m$). There may exist other exceptions. Please submit a GitHub issue if you discover one.

Examples

```python
In [1]: galois.primitive_poly(2, 6)
Out[1]: Poly(x^6 + x + 1, GF(2))

In [2]: galois.matlab_primitive_poly(2, 6)
Out[2]: Poly(x^6 + x + 1, GF(2))

In [3]: galois.primitive_poly(2, 7)
Out[3]: Poly(x^7 + x + 1, GF(2))

In [4]: galois.matlab_primitive_poly(2, 7)
Out[4]: Poly(x^7 + x^3 + 1, GF(2))
```

galois.is_primitive

galois.is_primitive(poly)

Determines whether the polynomial $f(x)$ over GF($q$) is primitive.

A degree-$m$ polynomial $f(x)$ over GF($q$) is *primitive* if it is irreducible and $f(x) \mid (x^k - 1)$ for $k = q^m - 1$ and no $k$ less than $q^m - 1$.

**Parameters**

- poly (galois.Poly) – A degree-$m$ polynomial $f(x)$ over GF($q$).

**Returns**

- True if the polynomial is primitive.

**Return type**

bool

References

• Algorithm 4.77 from https://cacr.uwaterloo.ca/hac/about/chap4.pdf

Examples

All Conway polynomials are primitive.

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

In [2]: galois.is_primitive(f)
Out[2]: True
```

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

In [4]: galois.is_primitive(f)
Out[4]: True

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

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

In [6]: galois.is_primitive(f)
Out[6]: False

**Primitive elements**

<table>
<thead>
<tr>
<th>primitive_element(irreducible_poly[, start,...])</th>
<th>Finds the smallest primitive element (g(x)) of the Galois field (\text{GF}(p^m)) with degree-(m) irreducible polynomial (f(x)) over (\text{GF}(p)).</th>
</tr>
</thead>
<tbody>
<tr>
<td>primitive_elements(irreducible_poly[, ...])</td>
<td>Finds all primitive elements (g(x)) of the Galois field (\text{GF}(p^m)) with degree-(m) irreducible polynomial (f(x)) over (\text{GF}(p)).</td>
</tr>
<tr>
<td>is_primitive_element(element, irreducible_poly)</td>
<td>Determines if (g(x)) is a primitive element of the Galois field (\text{GF}(p^m)) with degree-(m) irreducible polynomial (f(x)) over (\text{GF}(p)).</td>
</tr>
</tbody>
</table>

**galois.primitive_element**

galois.primitive_element(irreducible_poly, start=None, stop=None, reverse=False)

Finds the smallest primitive element \(g(x)\) of the Galois field \(\text{GF}(p^m)\) with degree-\(m\) irreducible polynomial \(f(x)\) over \(\text{GF}(p)\).

**Parameters**

- **irreducible_poly** (galois.Poly) – The degree-\(m\) irreducible polynomial \(f(x)\) over \(\text{GF}(p)\) that defines the extension field \(\text{GF}(p^m)\).

- **start** (int, optional) – Starting value (inclusive, integer representation of the polynomial) in the search for a primitive element \(g(x)\) of \(\text{GF}(p^m)\). The default is None which represents \(p\), which corresponds to \(g(x) = x\) over \(\text{GF}(p)\).

- **stop** (int, optional) – Stopping value (exclusive, integer representation of the polynomial) in the search for a primitive element \(g(x)\) of \(\text{GF}(p^m)\). The default is None which represents \(p^m\), which corresponds to \(g(x) = x^m\) over \(\text{GF}(p)\).

- **reverse** (bool, optional) – Search for a primitive element in reverse order, i.e. find the largest primitive element first. Default is False.

**Returns** A primitive element of \(\text{GF}(p^m)\) with irreducible polynomial \(f(x)\). The primitive element \(g(x)\) is a polynomial over \(\text{GF}(p)\) with degree less than \(m\).

**Return type** galois.Poly
Examples

In [1]: GF = galois.GF(3)
In [2]: f = galois.Poly([1,1,2], field=GF); f
Out[2]: Poly(x^2 + x + 2, GF(3))
In [3]: galois.is_irreducible(f)
Out[3]: True
In [4]: galois.is_primitive(f)
Out[4]: True
In [5]: galois.primitive_element(f)
Out[5]: Poly(x, GF(3))

In [6]: GF = galois.GF(3)
In [7]: f = galois.Poly([1,0,1], field=GF); f
Out[7]: Poly(x^2 + 1, GF(3))
In [8]: galois.is_irreducible(f)
Out[8]: True
In [9]: galois.is_primitive(f)
Out[9]: False
In [10]: galois.primitive_element(f)
Out[10]: Poly(x + 1, GF(3))

galois.primitive_elements

galois.primitive_elements(irreducible_poly, start=None, stop=None, reverse=False)
Finds all primitive elements \(g(x)\) of the Galois field \(GF(p^m)\) with degree-\(m\) irreducible polynomial \(f(x)\) over \(GF(p)\).

The number of primitive elements of \(GF(p^m)\) is \(\phi(p^m - 1)\), where \(\phi(n)\) is the Euler totient function. See :obj:`galois.euler_phi`.

Parameters

- **irreducible_poly** (galois.Poly) – The degree-\(m\) irreducible polynomial \(f(x)\) over \(GF(p)\) that defines the extension field \(GF(p^m)\).
- **start** (int, optional) – Starting value (inclusive, integer representation of the polynomial) in the search for primitive elements \(g(x)\) of \(GF(p^m)\). The default is None which represents \(p\), which corresponds to \(g(x) = x\) over \(GF(p)\).
- **stop** (int, optional) – Stopping value (exclusive, integer representation of the polynomial) in the search for primitive elements \(g(x)\) of \(GF(p^m)\). The default is None which represents \(p^m\), which corresponds to \(g(x) = x^m\) over \(GF(p)\).
- **reverse** (bool, optional) – Search for primitive elements in reverse order, i.e. largest to smallest. Default is False.
**Returns**  List of all primitive elements of $\text{GF}(p^m)$ with irreducible polynomial $f(x)$. Each primitive element $g(x)$ is a polynomial over $\text{GF}(p)$ with degree less than $m$.

**Return type**  list

**Examples**

```
In [1]: GF = galois.GF(3)
In [2]: f = galois.Poly([1,1,2], field=GF); f
Out[2]: Poly(x^2 + x + 2, GF(3))
In [3]: galois.is_irreducible(f)
Out[3]: True
In [4]: galois.is_primitive(f)
Out[4]: True
In [5]: g = galois.primitive_elements(f); g
Out[5]: [Poly(x, GF(3)), Poly(x + 1, GF(3)), Poly(2x, GF(3)), Poly(2x + 2, GF(3))]
In [6]: len(g) == galois.euler_phi(3**2 - 1)
Out[6]: True

In [7]: GF = galois.GF(3)
In [8]: f = galois.Poly([1,0,1], field=GF); f
Out[8]: Poly(x^2 + 1, GF(3))
In [9]: galois.is_irreducible(f)
Out[9]: True
In [10]: galois.is_primitive(f)
Out[10]: False
In [11]: g = galois.primitive_elements(f); g
Out[11]: [Poly(x + 1, GF(3)), Poly(x + 2, GF(3)), Poly(2x + 1, GF(3)), Poly(2x + 2, GF(3))]
In [12]: len(g) == galois.euler_phi(3**2 - 1)
Out[12]: True
```
galois.is_primitive_element

**galois.is_primitive_element(element, irreducible_poly)**

Determines if \( g(x) \) is a primitive element of the Galois field \( GF(p^m) \) with degree-\( m \) irreducible polynomial \( f(x) \) over \( GF(p) \).

**Parameters**

- **element** (*galois.Poly*) – An element \( g(x) \) of \( GF(p^m) \) as a polynomial over \( GF(p) \) with degree less than \( m \).
- **irreducible_poly** (*galois.Poly*) – The degree-\( m \) irreducible polynomial \( f(x) \) over \( GF(p) \) that defines the extension field \( GF(p^m) \).

**Returns**

True if \( g(x) \) is a primitive element of \( GF(p^m) \) with irreducible polynomial \( f(x) \).

**Return type** bool

**Notes**

The number of primitive elements of \( GF(p^m) \) is \( \phi(p^m - 1) \), where \( \phi(n) \) is the Euler totient function, see `galois.euler_phi()`.

**Examples**

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

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

In [3]: galois.is_irreducible(f)
Out[3]: True

In [4]: galois.is_primitive(f)
Out[4]: True

In [5]: g = galois.Poly.Identity(GF); g
Out[5]: Poly(x, GF(3))

In [6]: galois.is_primitive_element(g, f)
Out[6]: True

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

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

In [9]: galois.is_irreducible(f)
Out[9]: True

In [10]: galois.is_primitive(f)
Out[10]: False

In [11]: g = galois.Poly.Identity(GF); g
```
Minimal polynomials

`minimal_poly(element)` Computes the minimal polynomial $m_e(x) \in \mathbb{F}_p[x]$ of a Galois field element $e \in \mathbb{F}_{p^m}$.

**galois.minimal_poly**

`galois.minimal_poly(element)`

Computes the minimal polynomial $m_e(x) \in \mathbb{F}_p[x]$ of a Galois field element $e \in \mathbb{F}_{p^m}$.

The **minimal polynomial** of a Galois field element $e \in \mathbb{F}_{p^m}$ is the polynomial of minimal degree over $\mathbb{F}_p$ for which $e$ is a root when evaluated in $\mathbb{F}_{p^m}$. Namely, $m_e(x) \in \mathbb{F}_p[x]$ is a polynomial of the form $m_e(x) = \prod_{i=0}^{m-1} (x - e^i) \in \mathbb{F}_p[x]$ for which $m_e(e) = 0$ over $\mathbb{F}_{p^m}$.

**Parameters**

- `element` *(galois.FieldArray)* — Any element $e$ of the Galois field $\mathbb{F}_{p^m}$. This must be a 0-D array.

**Returns**

- The minimal polynomial $m_e(x)$ over $\mathbb{F}_p$ of the element $e$.

**Return type** `galois.Poly`

**Examples**

```python
gf = galois.GF(2**4)
e = gf.primitive_element;
e
m_e = galois.minimal_poly(e);
m_e
Out[3]: Poly(x^4 + x + 1, GF(2))
```

For a given element $e$, the minimal polynomials of $e$ and all its conjugates are the same.

```python
conjugates = np.unique(e**(2**np.arange(0, 4)));
for conjugate in conjugates:
    print(galois.minimal_poly(conjugate))
```

```
Poly(x^4 + x + 1, GF(2))
```

(continues on next page)
Not all elements of GF($2^4$) have minimal polynomials with degree-4.

```python
In [7]: e = GF.primitive_element**5; e
Out[7]: GF(6, order=2^4)

# The conjugates of e
In [8]: conjugates = np.unique(e**(2**np.arange(0, 4))); conjugates
Out[8]: GF([6, 7], order=2^4)

In [9]: for conjugate in conjugates:
   ...:     print(galois.minimal_poly(conjugate))
   ...:
Poly(x^2 + x + 1, GF(2))
Poly(x^2 + x + 1, GF(2))
```

In prime fields, the minimal polynomial of $e$ is simply $m_e(x) = x - e$.

```python
In [10]: GF = galois.GF(7)

In [11]: e = GF(3); e
Out[11]: GF(3, order=7)

In [12]: m_e = galois.minimal_poly(e); m_e
Out[12]: Poly(x + 4, GF(7))

In [13]: m_e(e)
Out[13]: GF(0, order=7)
```

### 7.2 Polynomials over Galois Fields

This section contains classes and functions for creating polynomials over Galois fields.

#### 7.2.1 Polynomial classes

- **Poly(coeffs, field, order)**: Create a polynomial $f(x)$ over GF($p^m$).
galois

galois.Poly

class galois.Poly(coeffs, field=None, order='desc')
Create a polynomial \( f(x) \) over \( GF(p^m) \).

The polynomial \( f(x) = a_dx^d + a_{d-1}x^{d-1} + \cdots + a_1x + a_0 \) has coefficients \( \{a_d, a_{d-1}, \ldots, a_1, a_0\} \) in \( GF(p^m) \).

Parameters

- **coeffs** (tuple, list, numpy.ndarray, galois.FieldArray) – The polynomial coefficients \( \{a_d, a_{d-1}, \ldots, a_1, a_0\} \) with type galois.FieldArray. Alternatively, an iterable tuple, list, or numpy.ndarray may be provided and the Galois field domain is taken from the field keyword argument.

- **field** (galois.FieldClass, optional) – The Galois field \( GF(p^m) \) the polynomial is over.
  - None (default): If the coefficients are a galois.FieldArray, they won’t be modified. If the coefficients are not explicitly in a Galois field, they are assumed to be from \( GF(2) \) and are converted using galois.GF2(coeffs).
  - galois.FieldClass: The coefficients are explicitly converted to this Galois field field(coeffs).

- **order** (str, optional) – The interpretation of the coefficient degrees.
  - "desc" (default): The first element of coeffs is the highest degree coefficient, i.e. \( \{a_d, a_{d-1}, \ldots, a_1, a_0\} \).
  - "asc": The first element of coeffs is the lowest degree coefficient, i.e. \( \{a_0, a_1, \ldots, a_{d-1}, a_d\} \).

Returns

The polynomial \( f(x) \).

Return type  galois.Poly

Examples

Create a polynomial over \( GF(2) \).

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

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

Create a polynomial over \( GF(2^8) \).

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

In [4]: galois.Poly([124,0,223,0,0,15], field=GF)
Out[4]: Poly(124x^5 + 223x^3 + 15, GF(2^8))

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

Polynomial arithmetic using binary operators.
In [6]: a = galois.Poly([117,0,63,37], field=GF); a
Out[6]: Poly(117x^3 + 63x + 37, GF(2^8))

In [7]: b = galois.Poly([224,0,21], field=GF); b
Out[7]: Poly(224x^2 + 21, GF(2^8))

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

In [9]: a - b
Out[9]: Poly(117x^3 + 224x^2 + 63x + 48, GF(2^8))

# Compute the quotient of the polynomial division
In [10]: a / b
Out[10]: Poly(202x, GF(2^8))

# True division and floor division are equivalent
In [11]: a / b == a // b
Out[11]: True

# Compute the remainder of the polynomial division
In [12]: a % b
Out[12]: Poly(198x + 37, GF(2^8))

# Compute both the quotient and remainder in one pass
In [13]: divmod(a, b)
Out[13]: (Poly(202x, GF(2^8)), Poly(198x + 37, GF(2^8)))

Constructors

__init__()

**Degrees**(degrees[, coeffs, field]) Constructs a polynomial over \( GF(p^m) \) from its non-zero degrees.

**Identity**(field) Constructs the polynomial \( f(x) = x \) over \( GF(p^m) \).

**Integer**(integer[, field]) Constructs a polynomial over \( GF(p^m) \) from its integer representation.

**One**(field) Constructs the polynomial \( f(x) = 1 \) over \( GF(p^m) \).

**Random**(degree[, field]) Constructs a random polynomial over \( GF(p^m) \) with degree \( d \).

**Roots**(roots[, multiplicities, field]) Constructs a monic polynomial over \( GF(p^m) \) from its roots.

**String**(string[, field]) Constructs a polynomial over \( GF(p^m) \) from its string representation.

**Zero**(field) Constructs the polynomial \( f(x) = 0 \) over \( GF(p^m) \).
Methods

*derivative*(\([k]\)) Computes the \(k\)-th formal derivative \(\frac{d^k}{dx^k} f(x)\) of the polynomial \(f(x)\).

*reverse*() Returns the \(d\)-th reversal \(x^d f\left(\frac{1}{x}\right)\) of the polynomial \(f(x)\) with degree \(d\).

*roots*([multiplicity]) Calculates the roots \(r\) of the polynomial \(f(x)\), such that \(f(r) = 0\).

Special Methods

*add*(*other*) Adds two polynomials.

*divmod*(*other*) Divides two polynomials and returns the quotient and remainder.

*floordiv*(*other*) Divides two polynomials and returns the quotient.

*mod*(*other*) Divides two polynomials and returns the remainder.

*mul*(*other*) Multiplies two polynomials.

*pow*(*other*) Exponentiates the polynomial to an integer power.

*sub*(*other*) Subtracts two polynomials.

*truediv*(*other*) Divides two polynomials and returns the quotient.

Attributes

*coeffs* The coefficients of the polynomial in degree-descending order.

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

*degrees* An array of the polynomial degrees in degree-descending order.

*field* The Galois field array class to which the coefficients belong.

*integer* The integer representation of the polynomial.

*nonzero_coeffs* The non-zero coefficients of the polynomial in degree-descending order.

*nonzero_degrees* An array of the polynomial degrees that have non-zero coefficients, in degree-descending order.

*string* The string representation of the polynomial, without specifying the Galois field.

**classmethod Degrees**(*degrees*, *coeffs=None*, *field=None*) Constructs a polynomial over \(GF(p^m)\) from its non-zero degrees.

**Parameters**


- *coeffs* (*tuple, list, numpy.ndarray, galois.FieldArray, optional*) – The corresponding non-zero polynomial coefficients with type *galois.FieldArray*. Alternatively, an iterable *tuple, list, or numpy.ndarray* may be provided and the Galois field
domain is taken from the field keyword argument. The default is None which corresponds to all ones.

- **field** (*galois.FieldClass, optional*) – The Galois field \( \text{GF}(p^m) \) the polynomial is over.
  - None (default): If the coefficients are a *galois.FieldArray*, they won't be modified. If the coefficients are not explicitly in a Galois field, they are assumed to be from \( \text{GF}(2) \) and are converted using *galois.GF2(coeffs)*.
  - *galois.FieldClass*: The coefficients are explicitly converted to this Galois field *field(coeffs)*.

Returns The polynomial \( f(x) \).

Return type *galois.Poly*

Examples

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

```
In [2]: GF = galois.GF(2**8)
In [3]: galois.Poly.Degrees([3,1,0], coeffs=[251,73,185], field=GF)
Out[3]: Poly(251x^3 + 73x + 185, GF(2^8))
```

classmethod **Identity**(field=<class 'numpy.ndarray over GF(2)'>)

Constructs the polynomial \( f(x) = x \) over \( \text{GF}(p^m) \).

Parameters

- **field** (*galois.FieldClass, optional*) – The Galois field \( \text{GF}(p^m) \) the polynomial is over. The default is *galois.GF2*.

Returns The polynomial \( f(x) = x \).

Return type *galois.Poly*

Examples

Construct the identity polynomial over \( \text{GF}(2) \).

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

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

```
In [2]: GF = galois.GF(2**8)
In [3]: galois.Poly.Identity(field=GF)
Out[3]: Poly(x, GF(2^8))
```

classmethod **Integer**(integer, field=<class 'numpy.ndarray over GF(2)'>)

Constructs a polynomial over \( \text{GF}(p^m) \) from its integer representation.
Parameters

- **integer** (*int*) – The integer representation of the polynomial \( f(x) \).
- **field** (*galois.FieldClass*, *optional*) – The Galois field \( \text{GF}(p^m) \) the polynomial is over. The default is `galois.GF2`.

Returns

The polynomial \( f(x) \).

Return type

`galois.Poly`

Notes

The integer value \( i \) represents the polynomial

\[
  f(x) = a_dx^d + a_{d-1}x^{d-1} + \cdots + a_1x + a_0 \text{ over the field } \text{GF}(p^m) \text{ if } i = a_d(p^m)^d + a_{d-1}(p^m)^{d-1} + \cdots + a_1(p^m) + a_0 \text{ using integer arithmetic, not finite field arithmetic.}
\]

Said differently, if the polynomial coefficients \( \{a_d, a_{d-1}, \ldots, a_1, a_0\} \) are considered as the “digits” of a radix-\( p^m \) value, the polynomial’s integer representation is the decimal value (radix-10).

Examples

Construct a polynomial over \( \text{GF}(2) \) from its integer representation.

```
In [1]: galois.Poly.Integer(5)
Out[1]: Poly(x^2 + 1, GF(2))
```

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

```
In [2]: GF = galois.GF(2**8)
In [3]: galois.Poly.Integer(13*256**3 + 117, field=GF)
Out[3]: Poly(13x^3 + 117, GF(2^8))
```

classmethod `One(field=<class 'numpy.ndarray over GF(2)'>)`

Constructs the polynomial \( f(x) = 1 \) over \( \text{GF}(p^m) \).

Parameters

- **field** (*galois.FieldClass*, *optional*) – The Galois field \( \text{GF}(p^m) \) the polynomial is over. The default is `galois.GF2`.

Returns

The polynomial \( f(x) = 1 \).

Return type

`galois.Poly`

Examples

Construct the one polynomial over \( \text{GF}(2) \).

```
In [1]: galois.Poly.One()
Out[1]: Poly(1, GF(2))
```

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

```
In [1]: galois.Poly.One()
Out[1]: Poly(1, GF(2^8))
```
In [2]: GF = galois.GF(2**8)
In [3]: galois.Poly.One(field=GF)
Out[3]: Poly(1, GF(2^8))

```
classmethod Random(degree, field=<class 'numpy.ndarray over GF(2)'>)
    Constructs a random polynomial over GF(p^m) with degree d.

    Parameters
    • degree (int) – The degree of the polynomial.
    • field (galois.FieldClass, optional) – The Galois field GF(p^m) the polynomial is
      over. The default is galois.GF2.

    Returns  The polynomial f(x).
    Return type  galois.Poly
```

Examples

Construct a random degree-5 polynomial over GF(2).

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

Construct a random degree-5 polynomial over GF(2^8).

```
In [2]: GF = galois.GF(2**8)
In [3]: galois.Poly.Random(5, field=GF)
Out[3]: Poly(151x^5 + 70x^4 + 89x^3 + 180x^2 + 92x + 137, GF(2^8))
```

```
classmethod Roots(roots, multiplicities=None, field=None)
    Constructs a monic polynomial over GF(p^m) from its roots.

    Parameters
    • roots (tuple, list, numpy.ndarray, galois.FieldArray) – The roots of the de-
      sired polynomial with type galois.FieldArray. Alternatively, an iterable tuple, list,
      or numpy.ndarray may be provided and the Galois field domain is taken from the
      field keyword argument.
    • multiplicities (tuple, list, numpy.ndarray, optional) – The correspond-
      ing root multiplicities. The default is None which corresponds to all ones, i.e. [1,]
      *len(roots).
    • field (galois.FieldClass, optional) – The Galois field GF(p^m) the polynomial is
      over.
      – None (default): If the roots are a galois.FieldArray, they won’t be modified. If the
        roots are not explicitly in a Galois field, they are assumed to be from GF(2) and are
        converted using galois.GF2(roots).
      – galois.FieldClass: The roots are explicitly converted to this Galois field
        field(roots).

    Returns  The polynomial f(x).
    Return type  galois.Poly
```

7.2. Polynomials over Galois Fields
Notes

The polynomial \( f(x) \) with \( k \) roots \( \{r_1, r_2, \ldots, r_k\} \) with multiplicities \( \{m_1, m_2, \ldots, m_k\} \) is

\[
f(x) = (x - r_1)^{m_1}(x - r_2)^{m_2} \cdots (x - r_k)^{m_k}
\]

with degree \( d = \sum_{i=1}^{k} m_i \).

Examples

Construct a polynomial over \( \text{GF}(2) \) from a list of its roots.

```python
In [1]: roots = [0, 0, 1]
In [2]: p = galois.Poly.Roots(roots); p
Out[2]: Poly(x^3 + x^2, GF(2))
# Evaluate the polynomial at its roots
In [3]: p(roots)
Out[3]: GF([0, 0, 0], order=2)
```

Construct a polynomial over \( \text{GF}(2^8) \) from a list of its roots with specific multiplicities.

```python
In [4]: GF = galois.GF(2**8)
In [5]: roots = [121, 198, 225]
In [6]: multiplicities = [1, 2, 1]
In [7]: p = galois.Poly.Roots(roots, multiplicities=multiplicities, field=GF); p
Out[7]: Poly(x^4 + 152x^3 + 85x^2 + 223x + 147, GF(2^8))
# Evaluate the polynomial at its roots
In [8]: p(roots)
Out[8]: GF([0, 0, 0], order=2^8)
```

**classmethod String**(string,
field=<class 'numpy.ndarray over GF(2)'>)

Constructs a polynomial over \( \text{GF}(p^m) \) from its string representation.

**Parameters**

- **string** *(str)* – The string representation of the polynomial \( f(x) \).
- **field** *(galois.FieldClass, optional)* – The Galois field \( \text{GF}(p^m) \) the polynomial is over. The default is \( galois.GF2 \).

**Returns** The polynomial \( f(x) \).

**Return type** *galois.Poly*
Notes

The string parsing rules include:

• Either ^ or ** may be used for indicating the polynomial degrees. For example, "13x^3 + 117" or "13x**3 + 117".

• Multiplication operators * may be used between coefficients and the polynomial indeterminate x, but are not required. For example, "13x^3 + 117" or "13*x^3 + 117".

• Polynomial coefficients of 1 may be specified or omitted. For example, "x^3 + 117" or "1*x^3 + 117".

• The polynomial indeterminate can be any single character, but must be consistent. For example, "13x^3 + 117" or "13y^3 + 117".

• Spaces are not required between terms. For example, "13x^3 + 117" or "13x^3+117".

• Any combination of the above rules is acceptable.

Examples

Construct a polynomial over GF(2) from its string representation.

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

Construct a polynomial over GF(2^8) from its string representation.

```
In [2]: GF = galois.GF(2**8)
In [3]: galois.Poly.String("13x^3 + 117", field=GF)
Out[3]: Poly(13x^3 + 117, GF(2^8))
```

```
classmethod Zero(field=<class 'numpy.ndarray' over GF(2)>)
```

Constructs the polynomial \( f(x) = 0 \) over GF\( (p^m) \).

Parameters field (galois.FieldClass, optional) – The Galois field GF\( (p^m) \) the polynomial is over. The default is galois.GF2.

Returns The polynomial \( f(x) = 0 \).

Return type galois.Poly

Examples

Construct the zero polynomial over GF(2).

```
In [1]: galois.Poly.Zero()
Out[1]: Poly(0, GF(2))
```

Construct the zero polynomial over GF\( (2^8) \).

```
In [2]: GF = galois.GF(2**8)
In [3]: galois.Poly.Zero(field=GF)
Out[3]: Poly(0, GF(2^8))
```

7.2. Polynomials over Galois Fields
__add__(other)
Adds two polynomials.

Parameters

other (galois.Poly) – The polynomial \(b(x)\).

Returns

The polynomial \(c(x) = a(x) + b(x)\).

Return type

galois.Poly

Examples

```python
In [1]: a = galois.Poly.Random(5); a
Out[1]: Poly(x^5 + x^2, GF(2))

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

In [3]: a + b
Out[3]: Poly(x^5 + x^3 + x + 1, GF(2))
```

__divmod__(other)
Divides two polynomials and returns the quotient and remainder.

Parameters

other (galois.Poly) – The polynomial \(b(x)\).

Returns

- \(q\) \(\in\) galois.Poly such that \(a(x) = b(x)q(x) + r(x)\).
- \(r\) \(\in\) galois.Poly such that \(a(x) = b(x)q(x) + r(x)\).

Examples

```python
In [1]: a = galois.Poly.Random(5); a
Out[1]: Poly(x^5 + x + 1, GF(2))

In [2]: b = galois.Poly.Random(3); b
Out[2]: Poly(x^3 + x^2 + x, GF(2))

In [3]: q, r = divmod(a, b)
In [4]: q, r
Out[4]: (Poly(x^2 + x, GF(2)), Poly(x^2 + x + 1, GF(2)))

In [5]: b*q + r
Out[5]: Poly(x^5 + x + 1, GF(2))
```

__floordiv__(other)
Divides two polynomials and returns the quotient.

True division and floor division are equivalent.

Parameters

other (galois.Poly) – The polynomial \(b(x)\).

Returns

The quotient polynomial \(q(x)\) such that \(a(x) = b(x)q(x) + r(x)\).

Return type

galois.Poly
Examples

In [1]: a = galois.Poly.Random(5); a
Out[1]: Poly(x^5 + x^4 + x^3 + x + 1, GF(2))

In [2]: b = galois.Poly.Random(3); b
Out[2]: Poly(x^3 + x^2 + x, GF(2))

In [3]: divmod(a, b)
Out[3]: (Poly(x^2, GF(2)), Poly(x + 1, GF(2)))

In [4]: a // b
Out[4]: Poly(x^2, GF(2))

__init__()

__mod__(other)
Divides two polynomials and returns the remainder.

Parameters other (galois.Poly) – The polynomial b(x).

Returns The remainder polynomial r(x) such that a(x) = b(x)q(x) + r(x).

Return type galois.Poly

Examples

In [1]: a = galois.Poly.Random(5); a
Out[1]: Poly(x^5 + x^4 + x^3 + x + 1, GF(2))

In [2]: b = galois.Poly.Random(3); b
Out[2]: Poly(x^3 + x^2 + x, GF(2))

In [3]: divmod(a, b)
Out[3]: (Poly(x^2, GF(2)), Poly(x + 1, GF(2)))

In [4]: a % b
Out[4]: Poly(x + 1, GF(2))

__mul__(other)
Multiplies two polynomials.

Parameters other (galois.Poly) – The polynomial b(x).

Returns The polynomial c(x) = a(x)b(x).

Return type galois.Poly
Examples

```
In [1]: a = galois.Poly.Random(5); a
Out[1]: Poly(x^5 + x^4 + x^2 + x + 1, GF(2))

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

In [3]: a * b
Out[3]: Poly(x^8 + x^7 + x^3 + x^2 + x + 1, GF(2))
```

`__pow__(other)`

Exponentiates the polynomial to an integer power.

**Parameters**

- **other** (*int*) — The non-negative integer exponent.

**Returns**

The polynomial $a(x)^{\text{other}}$.

**Return type**

`galois.Poly`

Examples

```
In [1]: a = galois.Poly.Random(5); a
Out[1]: Poly(x^5 + x^3 + x^2 + x + 1, GF(2))

In [2]: a**3
Out[2]: Poly(x^15 + x^13 + x^12 + x^10 + x^8 + x^7 + x^5 + x + 1, GF(2))

In [3]: a * a * a
Out[3]: Poly(x^15 + x^13 + x^12 + x^10 + x^8 + x^7 + x^5 + x + 1, GF(2))
```

`__sub__(other)`

Subtracts two polynomials.

**Parameters**

- **other** (*galois.Poly*) — The polynomial $b(x)$.

**Returns**

The polynomial $c(x) = a(x) - b(x)$.

**Return type**

`galois.Poly`

Examples

```
In [1]: a = galois.Poly.Random(5); a
Out[1]: Poly(x^5 + x^4 + x^3 + 1, GF(2))

In [2]: b = galois.Poly.Random(3); b
Out[2]: Poly(x^3, GF(2))

In [3]: a - b
Out[3]: Poly(x^5 + x^4 + 1, GF(2))
```

`__truediv__(other)`

Divides two polynomials and returns the quotient.

True division and floor division are equivalent.
Parameters `other` (`galois.Poly`) – The polynomial $b(x)$.

Returns The quotient polynomial $q(x)$ such that $a(x) = b(x)q(x) + r(x)$.

Return type `galois.Poly`

Examples

```python
In [1]: a = galois.Poly.Random(5); a
Out[1]: Poly(x^5 + x^2, GF(2))

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

In [3]: divmod(a, b)
Out[3]: (Poly(x^2 + x + 1, GF(2)), Poly(x^2 + x + 1, GF(2)))

In [4]: a / b
Out[4]: Poly(x^2 + x + 1, GF(2))
```

derivative($k=1$)

Computes the $k$-th formal derivative $\frac{d^k}{dx^k} f(x)$ of the polynomial $f(x)$.

Parameters `k` (`int`, optional) – The number of derivatives to compute. 1 corresponds to $p'(x)$, 2 corresponds to $p''(x)$, etc. The default is 1.

Returns The $k$-th formal derivative of the polynomial $f(x)$.

Return type `galois.Poly`

Notes

For the polynomial

$$f(x) = a_dx^d + a_{d-1}x^{d-1} + \cdots + a_1x + a_0$$

the first formal derivative is defined as

$$f'(x) = (d) \cdot a_dx^{d-1} + (d-1) \cdot a_{d-1}x^{d-2} + \cdots + (2) \cdot a_2x + a_1$$

where $\cdot$ represents scalar multiplication (repeated addition), not finite field multiplication. For example, $3 \cdot a = a + a + a$.

References


7.2. Polynomials over Galois Fields
Examples

Compute the derivatives of a polynomial over GF(2).

```
In [1]: p = galois.Poly.Random(7); p
Out[1]: Poly(x^7 + x^6, GF(2))

In [2]: p.derivative()
Out[2]: Poly(x^6, GF(2))

# k derivatives of a polynomial where k is the Galois field’s characteristic → will always result in 0
In [3]: p.derivative(2)
Out[3]: Poly(0, GF(2))
```

Compute the derivatives of a polynomial over GF(7).

```
In [4]: GF = galois.GF(7)

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

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

In [7]: p.derivative(2)
Out[7]: Poly(3x^9 + 6x^8 + 2x^7 + 5x^4 + 6x^3 + 4x + 1, GF(7))

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

# k derivatives of a polynomial where k is the Galois field’s characteristic → will always result in 0
In [9]: p.derivative(7)
Out[9]: Poly(0, GF(7))
```

Compute the derivatives of a polynomial over GF(2^8).

```
In [10]: GF = galois.GF(2**8)

In [11]: p = galois.Poly.Random(7, field=GF); p
Out[11]: Poly(85x^7 + 123x^6 + 167x^5 + 195x^4 + 118x^3 + 60x^2 + 66x + 82, GF(2^8))

In [12]: p.derivative()
Out[12]: Poly(85x^6 + 167x^5 + 118x^2 + 66, GF(2^8))

# k derivatives of a polynomial where k is the Galois field’s characteristic → will always result in 0
In [13]: p.derivative(2)
Out[13]: Poly(0, GF(2^8))
```

`reverse()`

Returns the \( d \)-th reversal \( x^d f\left(\frac{1}{x}\right) \) of the polynomial \( f(x) \) with degree \( d \).
Returns The \( n \)-th reversal \( x^n f(\frac{1}{x}) \).

Return type \( \text{galois.Poly} \)

Notes

For a polynomial \( f(x) = a_dx^d + a_{d-1}x^{d-1} + \cdots + a_1x + a_0 \) with degree \( d \), the \( d \)-th reversal is equivalent to reversing the coefficients.

\[
\text{rev}_d f(x) = x^d f(x^{-1}) = a_0x^d + a_1x^{d-1} + \cdots + a_{d-1}x + a_d
\]

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))

\( \text{roots}(\text{multiplicity}=\text{False}) \)

Calculates the roots \( r \) of the polynomial \( f(x) \), such that \( f(r) = 0 \).

Parameters multiplicity (bool, optional) – Optionally return the multiplicity of each root. The default is False which only returns the unique roots.

Returns

- \( \text{galois.FieldArray} \) – Galois field array of roots of \( f(x) \). The roots are ordered in increasing order.
- \( \text{np.ndarray} \) – The multiplicity of each root, only returned if multiplicity=True.

Notes

This implementation uses Chien’s search to find the roots \( \{r_1, r_2, \ldots, r_k\} \) of the degree-\( d \) polynomial

\[
f(x) = a_dx^d + a_{d-1}x^{d-1} + \cdots + a_1x + a_0,
\]

where \( k \leq d \). Then, \( f(x) \) can be factored as

\[
f(x) = (x - r_1)^{m_1}(x - r_2)^{m_2} \cdots (x - r_k)^{m_k},
\]

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 \( \text{GF}(p^m) = \{0, 1, \alpha, \alpha^2, \ldots, \alpha^{p^m-2}\} \), where \( \alpha \) is a primitive element of \( \text{GF}(p^m) \).

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 \( \text{GF}(p^m) \) are powers of \( \alpha \). The following equations calculate \( f(\alpha^i) \), where \( \alpha^i \) is a root of \( f(x) \) if \( f(\alpha^i) = 0 \).

\[
f(\alpha^i) = a_d(\alpha^i)^d + a_{d-1}(\alpha^i)^{d-1} + \cdots + a_1(\alpha^i) + a_0
\]

\[
f(\alpha^i) \triangleq \lambda_{i,d} + \lambda_{i,d-1} + \cdots + \lambda_{i,1} + \lambda_{i,0}
\]

\[
f(\alpha^i) = \sum_{j=0}^{d} \lambda_{i,j}
\]
The next power of $\alpha$ can be easily calculated from the previous calculation.

\[
f(\alpha^{i+1}) = a_d(\alpha^{i+1})^d + a_{d-1}(\alpha^{i+1})^{d-1} + \cdots + a_1(\alpha^{i+1}) + a_0
\]

\[
f(\alpha^{i+1}) = a_d(\alpha)^d + a_{d-1}(\alpha)^{d-1} + \cdots + a_1(\alpha) + a_0
\]

\[
f(\alpha^{i+1}) = \lambda_{i,d}\alpha^d + \lambda_{i,d-1}\alpha^{d-1} + \cdots + \lambda_{i,1}\alpha + \lambda_{i,0}
\]

\[
f(\alpha^{i+1}) = \sum_{j=0}^{d} \lambda_{i,j}\alpha^j
\]

References


Examples

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

```python
In [1]: p = galois.Poly.Roots([0,]*7 + [1,]*13); p
Out[1]: Poly(x^20 + x^19 + x^16 + x^15 + x^12 + x^11 + x^8 + x^7, GF(2))

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

In [3]: p.roots(multiplicity=True)
Out[3]: (GF([0, 1], order=2), array([ 7, 13]))
```

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

```python
In [4]: GF = galois.GF(2**8)

In [5]: p = galois.Poly.Roots([18,]*7 + [155,]*13 + [227,]*9, field=GF); p
Out[5]: Poly(x^29 + 106x^28 + 27x^27 + 155x^26 + 230x^25 + 38x^24 + 78x^23 + 8x^22 + 46x^21 + 210x^20 + 248x^19 + 214x^18 + 172x^17 + 152x^16 + 82x^15 + 237x^14 + 172x^13 + 230x^12 + 141x^11 + 63x^10 + 103x^9 + 167x^8 + 199x^7 + 127x^6 +
   \rightarrow 254x^5 + 95x^4 + 93x^3 + 3x^2 + 4x + 208, GF(2^8))

In [6]: p.roots()
Out[6]: GF([ 18, 155, 227], order=2^8)

In [7]: p.roots(multiplicity=True)
Out[7]: (GF([ 18, 155, 227], order=2^8), array([ 7, 13, 9]))
```

**property `coeffs`**

The coefficients of the polynomial in degree-descending order. The entries of `degrees` are paired with `coeffs`. 
Examples

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

In [2]: p = galois.Poly([3, 0, 5, 2], field=GF); p
Out[2]: Poly(3x^3 + 5x + 2, GF(7))

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

Type `galois.FieldArray`

**property degree**

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

Examples

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

In [2]: p = galois.Poly([3, 0, 5, 2], field=GF); p
Out[2]: Poly(3x^3 + 5x + 2, GF(7))

In [3]: p.degree
Out[3]: 3
```

Type `int`

**property degrees**

An array of the polynomial degrees in degree-descending order. The entries of `degrees` are paired with `coeffs`.

Examples

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

In [2]: p = galois.Poly([3, 0, 5, 2], field=GF); p
Out[2]: Poly(3x^3 + 5x + 2, GF(7))

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

Type `numpy.ndarray`

**property field**

The Galois field array class to which the coefficients belong.
Examples

```
In [1]: a = galois.Poly.Random(5); a
Out[1]: Poly(x^5 + 1, GF(2))

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

In [3]: GF = galois.GF(2**8)
In [4]: b = galois.Poly.Random(5, field=GF); b
Out[4]: Poly(96x^5 + 6x^4 + 141x^3 + 179x^2 + 30x + 129, GF(2^8))
In [5]: b.field
Out[5]: <class 'numpy.ndarray over GF(2^8)'>
```

**Type**  
`galois.FieldClass`

**property integer**

The integer representation of the polynomial. For the polynomial \( f(x) = a_d x^d + a_{d-1} x^{d-1} + \cdots + a_1 x + a_0 \) over the field \( \text{GF}(p^m) \), 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, if the polynomial coefficients \( \{a_d, a_{d-1}, \ldots, a_1, a_0\} \) are considered as the “digits” of a radix-\( p^m \) value, the polynomial’s integer representation is the decimal value (radix-10).

**Examples**

```
In [1]: GF = galois.GF(7)
In [2]: p = galois.Poly([3, 0, 5, 2], field=GF); p
Out[2]: Poly(3x^3 + 5x + 2, GF(7))
In [3]: p.integer
Out[3]: 1066
In [4]: p.integer == 3*7**3 + 5*7**1 + 2*7**0
Out[4]: True
```

**Type**  
`int`

**property nonzero_coeffs**

The non-zero coefficients of the polynomial in degree-descending order. The entries of `nonzero_degrees` are paired with `nonzero_coeffs`. 

---

148  Chapter 7. API Reference v0.0.21
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** nonzero_coeffs

An array of the polynomial degrees that have non-zero coefficients, in degree-descending order. The entries of nonzero_degrees are paired with nonzero_coeffs.

Examples

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

**property** string

The string representation of the polynomial, without specifying the Galois field.

Examples

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

**Type** str
### 7.2.2 Special polynomials

#### Irreducible polynomials

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>irreducible_poly(order, degree[, method])</code></td>
<td>Returns a monic irreducible polynomial $f(x)$ over $\text{GF}(q)$ with degree $m$.</td>
</tr>
<tr>
<td><code>irreducible_polys(order, degree)</code></td>
<td>Returns all monic irreducible polynomials $f(x)$ over $\text{GF}(q)$ with degree $m$.</td>
</tr>
</tbody>
</table>

#### Primitive polynomials

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>primitive_poly(order, degree[, method])</code></td>
<td>Returns a monic primitive polynomial $f(x)$ over $\text{GF}(q)$ with degree $m$.</td>
</tr>
<tr>
<td><code>primitive_polys(order, degree)</code></td>
<td>Returns all monic primitive polynomials $f(x)$ over $\text{GF}(q)$ with degree $m$.</td>
</tr>
<tr>
<td><code>conway_poly(characteristic, degree)</code></td>
<td>Returns the Conway polynomial $C_{p,m}(x)$ over $\text{GF}(p)$ with degree $m$.</td>
</tr>
<tr>
<td><code>matlabPrimitive_poly(characteristic, degree)</code></td>
<td>Returns Matlab’s default primitive polynomial $f(x)$ over $\text{GF}(p)$ with degree $m$.</td>
</tr>
</tbody>
</table>

#### Minimal polynomials

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>minimal_poly(element)</code></td>
<td>Computes the minimal polynomial $m_e(x) \in \text{GF}(p^m)[x]$ of a Galois field element $e \in \text{GF}(p^m)$.</td>
</tr>
</tbody>
</table>

### 7.2.3 Polynomial functions

#### Divisibility

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>gcd(a, b)</code></td>
<td>Finds the greatest common divisor of $a$ and $b$.</td>
</tr>
<tr>
<td><code>egcd(a, b)</code></td>
<td>Finds the multiplicands of $a$ and $b$ such that $as + bt = \gcd(a, b)$.</td>
</tr>
<tr>
<td><code>lcm(*values)</code></td>
<td>Computes the least common multiple of the arguments.</td>
</tr>
<tr>
<td><code>prod(*values)</code></td>
<td>Computes the product of the arguments.</td>
</tr>
<tr>
<td><code>are_coprime(*values)</code></td>
<td>Determines if the arguments are pairwise coprime.</td>
</tr>
</tbody>
</table>

**galois.gcd**

**Syntax**

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

Finds the greatest common divisor of $a$ and $b$.

**Parameters**

- `a (int, galois.Poly)` – The first integer or polynomial argument.
- `b (int, galois.Poly)` – The second integer or polynomial argument.

**Returns**

Greatest common divisor of $a$ and $b$. 
**Return type** int, galois.Poly

**Notes**

This function implements the Euclidean Algorithm.

**References**

- Algorithm 2.104 from https://cacr.uwaterloo.ca/hac/about/chap2.pdf
- Algorithm 2.218 from https://cacr.uwaterloo.ca/hac/about/chap2.pdf

**Examples**

Compute the GCD of two integers.

```python
In [1]: galois.gcd(12, 16)
Out[1]: 4
```

Compute the GCD of two polynomials.

```python
In [2]: GF = galois.GF(7)
In [3]: p1 = galois.irreducible_poly(7, 1); p1
Out[3]: Poly(x, GF(7))
In [4]: p2 = galois.irreducible_poly(7, 2); p2
Out[4]: Poly(x^2 + 1, GF(7))
In [5]: p3 = galois.irreducible_poly(7, 3); p3
Out[5]: Poly(x^3 + 2, GF(7))
In [6]: a = p1**2 * p2; a
Out[6]: Poly(x^4 + x^2, GF(7))
In [7]: b = p1 * p3; b
Out[7]: Poly(x^4 + 2x, GF(7))
In [8]: gcd = galois.gcd(a, b); gcd
Out[8]: Poly(x, GF(7))
```

**galois.egcd**

`galois.egcd(a, b)`

Finds the multiplicands of `a` and `b` such that `as + bt = gcd(a, b)`.

**Parameters**

- `a` (int, galois.Poly) – The first integer or polynomial argument.
- `b` (int, galois.Poly) – The second integer or polynomial argument.

**Returns**

7.2. Polynomials over Galois Fields 151
• `int, galois.Poly` – Greatest common divisor of `a` and `b`.
• `int, galois.Poly` – The multiplicand `s` of `a`, such that `as + bt = gcd(a, b)`.
• `int, galois.Poly` – The multiplicand `t` of `b`, such that `as + bt = gcd(a, b)`.

Notes

This function implements the Extended Euclidean Algorithm.

References

• Algorithm 2.107 from https://cacr.uwaterloo.ca/hac/about/chap2.pdf
• Algorithm 2.221 from https://cacr.uwaterloo.ca/hac/about/chap2.pdf
• T. Moon, “Error Correction Coding”, Section 5.2.2: The Euclidean Algorithm and Euclidean Domains, p. 181

Examples

Compute the extended GCD of two integers.

```
In [1]: a, b = 12, 16
In [2]: gcd, s, t = galois.egcd(a, b)
In [3]: gcd, s, t
Out[3]: (4, -1, 1)
In [4]: a*s + b*t == gcd
Out[4]: True
```

Compute the extended GCD of two polynomials.

```
In [5]: GF = galois.GF(7)
In [6]: p1 = galois.irreducible_poly(7, 1); p1
Out[6]: Poly(x, GF(7))
In [7]: p2 = galois.irreducible_poly(7, 2); p2
Out[7]: Poly(x^2 + 1, GF(7))
In [8]: p3 = galois.irreducible_poly(7, 3); p3
Out[8]: Poly(x^3 + 2, GF(7))
In [9]: a = p1**2 * p2; a
Out[9]: Poly(x^4 + x^2, GF(7))
In [10]: b = p1 * p3; b
Out[10]: Poly(x^4 + 2x, GF(7))
In [11]: gcd, s, t = galois.egcd(a, b)
```
galois

```python
In [12]: gcd, s, t
Out[12]: (Poly(x, GF(7)), Poly(2x^2 + 4x + 1, GF(7)), Poly(5x^2 + 3x + 4, GF(7)))
In [13]: a*s + b*t == gcd
Out[13]: True
```

galois.lcm

```
galois.lcm(*values)

Computes the least common multiple of the arguments.

Parameters *values (int, galois.Poly) – Each argument must be an integer or polynomial.

Returns The least common multiple of the arguments.

Return type int, galois.Poly
```

Examples

Compute the LCM of three integers.

```
In [1]: galois.lcm(2, 4, 14)
Out[1]: 28
```

Compute the LCM of three polynomials.

```
In [2]: GF = galois.GF(7)
In [3]: p1 = galois.irreducible_poly(7, 1); p1
Out[3]: Poly(x, GF(7))
In [4]: p2 = galois.irreducible_poly(7, 2); p2
Out[4]: Poly(x^2 + 1, GF(7))
In [5]: p3 = galois.irreducible_poly(7, 3); p3
Out[5]: Poly(x^3 + 2, GF(7))
In [6]: a = p1**2 * p2; a
Out[6]: Poly(x^4 + x^2, GF(7))
In [7]: b = p1 * p3; b
Out[7]: Poly(x^4 + 2x, GF(7))
In [8]: c = p2 * p3; c
Out[8]: Poly(x^5 + x^3 + 2x^2 + 2, GF(7))
In [9]: galois.lcm(a, b, c)
Out[9]: Poly(x^7 + x^5 + 2x^4 + 2x^2, GF(7))
In [10]: p1**2 * p2 * p3
Out[10]: Poly(x^7 + x^5 + 2x^4 + 2x^2, GF(7))
```

7.2. Polynomials over Galois Fields

153
galois.prod

**galois.prod(**values**)

Computes the product of the arguments.

**Parameters**

*values (int, galois.Poly)* – Each argument must be an integer or polynomial.

**Returns**

The product of the arguments.

**Return type**

int, galois.Poly

**Examples**

Compute the product of three integers.

```
In [1]: galois.prod(2, 4, 14)
Out[1]: 112
```

Compute the product of three polynomials.

```
In [2]: GF = galois.GF(7)
In [3]: a = galois.Poly.Random(2, field=GF)
In [4]: b = galois.Poly.Random(3, field=GF)
In [5]: c = galois.Poly.Random(4, field=GF)
In [6]: galois.prod(a, b, c)
Out[6]: Poly(4x^9 + 6x^8 + 3x^7 + x^5 + 5x^4 + 6x^3 + x^2 + 6x, GF(7))
In [7]: a * b * c
Out[7]: Poly(4x^9 + 6x^8 + 3x^7 + x^5 + 5x^4 + 6x^3 + x^2 + 6x, GF(7))
```

---

galois.are_coprime

**galois.are_coprime(**values**)

Determines if the arguments are pairwise coprime.

**Parameters**

*values (int, galois.Poly)* – Each argument must be an integer or polynomial.

**Returns**

True if the arguments are pairwise coprime.

**Return type**

bool
Notes

A set of integers or polynomials are pairwise coprime if their LCM is equal to their product.

Examples

Determine if a set of integers are pairwise coprime.

```
In [1]: galois.are_coprime(3, 4, 5)
Out[1]: True

In [2]: galois.are_coprime(3, 7, 9, 11)
Out[2]: False
```

Determine if a set of polynomials are pairwise coprime.

```
In [3]: GF = galois.GF(7)
In [4]: p1 = galois.irreducible_poly(7, 1); p1
Out[4]: Poly(x, GF(7))
In [5]: p2 = galois.irreducible_poly(7, 2); p2
Out[5]: Poly(x^2 + 1, GF(7))
In [6]: p3 = galois.irreducible_poly(7, 3); p3
Out[6]: Poly(x^3 + 2, GF(7))
In [7]: galois.are_coprime(p1, p2, p3)
Out[7]: True
In [8]: galois.are_coprime(p1*p2, p2, p3)
Out[8]: False
```

Congruences

\[
pow(base, exponent, modulus) \quad \text{Efficiently performs modular exponentiation.}
\]
\[
crt(remainders, moduli) \quad \text{Solves the simultaneous system of congruences for } x.
\]

**galois.pow**

\[
galois.pow(base, exponent, modulus)
\]

Efficiently performs modular exponentiation.

Parameters

- **base (int, galois.Poly)** – The integer or polynomial base \( a \).
- **exponent (int)** – The non-negative integer exponent \( k \).
- **modulus (int, galois.Poly)** – The integer or polynomial modulus \( m \).

Returns

The modular exponentiation \( a^k \mod m \).
**Return type**  int, galois.Poly

**Notes**
This function implements the Square-and-Multiply Algorithm. The algorithm is more efficient than exponentiating first and then reducing modulo $m$, especially for very large exponents. Instead, this algorithm repeatedly squares $a$, reducing modulo $m$ at each step.

**References**
- Algorithm 2.143 from https://cacr.uwaterloo.ca/hac/about/chap2.pdf
- Algorithm 2.227 from https://cacr.uwaterloo.ca/hac/about/chap2.pdf

**Examples**
Compute the modular exponentiation of an integer.

```
In [1]: galois.pow(3, 100, 7)
Out[1]: 4

In [2]: 3**100 % 7
Out[2]: 4
```

Compute the modular exponentiation of a polynomial.

```
In [3]: GF = galois.GF(7)
In [4]: a = galois.Poly.Random(3, field=GF)
In [5]: m = galois.Poly.Random(10, field=GF)
In [6]: galois.pow(a, 100, m)
Out[6]: Poly(6x^9 + 3x^8 + 2x^7 + 3x^6 + x^5 + 5x^4 + 5x^3 + 6x^2 + 4, GF(7))

In [7]: a**100 % m
Out[7]: Poly(6x^9 + 3x^8 + 2x^7 + 3x^6 + x^5 + 5x^4 + 5x^3 + 6x^2 + 4, GF(7))
```

**galois.crt**

galois.crt(remainders, moduli)
Solves the simultaneous system of congruences for $x$.

**Parameters**
- *remainders* (tuple, list) – The integer or polynomial remainders $a_i$.
- *moduli* (tuple, list) – The integer or polynomial moduli $m_i$.

**Returns** The simultaneous solution $x$ to the system of congruences.

**Return type** int
Notes

This function implements the Chinese Remainder Theorem.

\[ x \equiv a_1 \pmod{m_1} \]
\[ x \equiv a_2 \pmod{m_2} \]
\[ x \equiv \ldots \]
\[ x \equiv a_n \pmod{m_n} \]

References

- Section 14.5 from https://cacr.uwaterloo.ca/hac/about/chap14.pdf

Examples

Solve a system of integer congruences.

```
In [1]: a = [0, 3, 4]
In [2]: m = [3, 4, 5]
In [3]: x = galois.crt(a, m); x
Out[3]: 39
In [4]: for i in range(len(a)):
    ...:     ai = x % m[i]
    ...:     print(f"x = {ai} (mod {m[i]}), Valid congruence: {ai == a[i]}")
39 = 0 (mod 3), Valid congruence: True
39 = 3 (mod 4), Valid congruence: True
39 = 4 (mod 5), Valid congruence: True
```

Solve a system of polynomial congruences.

```
In [5]: GF = galois.GF(7)
In [6]: x_truth = galois.Poly.Random(10, field=GF); x_truth
Out[6]: Poly(x^10 + 2x^9 + 5x^8 + 4x^7 + 3x^6 + 5x^5 + 2x^4 + 5x^3 + 6x^2 + 6, →GF(7))
In [7]: m = [galois.irreducible_poly(7, 2), galois.irreducible_poly(7, 3), galois.irreducible_poly(7, 4)]; m
Out[7]: [Poly(x^2 + 1, GF(7)), Poly(x^3 + 2, GF(7)), Poly(x^4 + x + 1, GF(7))]
In [8]: a = [x_truth % mi for mi in m]; a
Out[8]: [Poly(5x + 3, GF(7)), Poly(2x^2 + 4x + 6, GF(7)), Poly(2x^3 + x^2 + x + 5, GF(7))]
In [9]: x = galois.crt(a, m); x
Out[9]: Poly(4x^8 + 6x^7 + 3x^6 + 4x^3 + x + 2, GF(7))
```

(continues on next page)
In [10]: for i in range(len(a)):
....:     ai = x % m[i]
....:     print(f"x = {ai} (mod {m[i]}), Valid congruence: {ai == a[i]}")
....:
Poly(4x^8 + 6x^7 + 3x^6 + 4x^3 + x + 2, GF(7)) = Poly(5x + 3, GF(7)) (mod Poly(x^2 + 1, GF(7))), Valid congruence: True
Poly(4x^8 + 6x^7 + 3x^6 + 4x^3 + x + 2, GF(7)) = Poly(2x^2 + 4x + 6, GF(7)) (mod Poly(x^3 + 2, GF(7))), Valid congruence: True
Poly(4x^8 + 6x^7 + 3x^6 + 4x^3 + x + 2, GF(7)) = Poly(2x^3 + x^2 + x + 5, GF(7)) (mod Poly(x^4 + x + 1, GF(7))), Valid congruence: True

Polynomial factorization

<table>
<thead>
<tr>
<th><strong>factors(value)</strong></th>
<th>Computes the prime factors of a positive integer or the irreducible factors of a non-constant, monic polynomial.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>square_free_factorization(poly)</strong></td>
<td>Factors the monic polynomial ( f(x) ) into a product of square-free polynomials.</td>
</tr>
<tr>
<td><strong>distinct_degree_factorization(poly)</strong></td>
<td>Factors the monic, square-free polynomial ( f(x) ) into a product of polynomials whose irreducible factors all have the same degree.</td>
</tr>
<tr>
<td><strong>equal_degree_factorization(poly, degree)</strong></td>
<td>Factors the monic, square-free polynomial ( f(x) ) of degree ( rd ) into a product of ( r ) irreducible factors with degree ( d ).</td>
</tr>
</tbody>
</table>

**galois.factors**

galois.factors(value)
Computes the prime factors of a positive integer or the irreducible factors of a non-constant, monic polynomial.

**Parameters**

- **value** (*int, galois.Poly*) – A positive integer \( n \) or a non-constant, monic polynomial \( f(x) \).

**Returns**

- **list** – Sorted list of prime factors \( \{p_1, p_2, \ldots, p_k\} \) of \( n \) with \( p_1 < p_2 < \cdots < p_k \) or irreducible factors \( \{g_1(x), g_2(x), \ldots, g_k(x)\} \) of \( f(x) \) sorted in lexicographically-increasing order.
- **list** – List of corresponding multiplicities \( \{e_1, e_2, \ldots, e_k\} \).
Notes

Integer Factorization
This function factors a positive integer \( n \) into its \( k \) prime factors such that \( n = p_1^{e_1} p_2^{e_2} \ldots p_k^{e_k} \).

Steps:
1. Test if \( n \) is prime. If so, return \([n], [1]\).
2. Test if \( n \) is a perfect power, such that \( n = x^k \). If so, prime factor \( x \) and multiply the exponents by \( k \).
3. Use trial division with a list of primes up to \( 10^6 \). If no residual factors, return the discovered prime factors.
4. Use Pollard’s Rho algorithm to find a non-trivial factor of the residual. Continue until all are found.

Polynomial Factorization
This function factors a monic polynomial \( f(x) \) into its \( k \) irreducible factors such that \( f(x) = g_1(x)^{e_1} g_2(x)^{e_2} \ldots g_k(x)^{e_k} \).

Steps:
1. Apply the Square-Free Factorization algorithm to factor the monic polynomial into square-free polynomials.
2. Apply the Distinct-Degree Factorization algorithm to factor each square-free polynomial into a product of factors with the same degree.
3. Apply the Equal-Degree Factorization algorithm to factor the product of factors of equal degree into their irreducible factors.

References

- Section 2.1 from https://people.csail.mit.edu/dmoshkov/courses/codes/poly-factorization.pdf

Examples

Factor a positive integer.

\[
\begin{align*}
\textbf{In [1]:} & \quad \text{galois.factors(120)} \\
\text{Out[1]:} & \quad ([2, 3, 5], [3, 1, 1])
\end{align*}
\]

Factor a polynomial over GF(3).

\[
\begin{align*}
\textbf{In [2]:} & \quad \text{GF = galois.GF(3)} \\
\text{In [3]:} & \quad g1, g2, g3 = galois.irreducible_poly(3, 3), galois.irreducible_poly(3, 4), \ldots \\
& \quad \text{galois.irreducible_poly(3, 5)} \\
\text{In [4]:} & \quad g1, g2, g3 \\
\text{Out[4]:} & \quad (\text{Poly}(x^3 + 2x + 1, GF(3)), \\
& \quad \text{Poly}(x^4 + x + 2, GF(3)), \\
& \quad \text{Poly}(x^5 + 2x + 1, GF(3))) \\
\text{In [5]:} & \quad e1, e2, e3 = 5, 4, 3
\end{align*}
\]

(continues on next page)
# Construct the composite polynomial

```python
In [6]: f = g1**e1 * g2**e2 * g3**e3

In [7]: galois.factors(f)
Out[7]:
([Poly(x^3 + 2x + 1, GF(3)), Poly(x^4 + x + 2, GF(3)), Poly(x^5 + 2x + 1, GF(3))], [5, 4, 3])
```

### galois.square_free_factorization

```python
galois.square_free_factorization(poly)
```

Factors the monic polynomial $f(x)$ into a product of square-free polynomials.

**Parameters**

- `poly` *(galois.Poly)* – A non-constant, monic polynomial $f(x)$ over $GF(p^m)$.

**Returns**

- `list` – The list of non-constant, square-free polynomials $h_i(x)$ in the factorization.
- `list` – The list of corresponding multiplicities $i$.

**Notes**

The Square-Free Factorization algorithm factors $f(x)$ into a product of $m$ square-free polynomials $h_j(x)$ with multiplicity $j$.

$$f(x) = \prod_{j=1}^{m} h_j(x)^i$$

Some $h_j(x) = 1$, but those polynomials are not returned by this function.

A complete polynomial factorization is implemented in *galois.factors()*.

**References**

- Section 2.1 from [https://people.csail.mit.edu/dmoshkov/courses/codes/poly-factorization.pdf](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 $GF(5)$. Each polynomial $x$, $x^3 + 2x + 4$, and $x^2 + 4x + 1$ are all irreducible over $GF(5)$.

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

In [2]: a = galois.Poly([1,0], field=GF); a, galois.is_irreducible(a)
Out[2]: (Poly(x, GF(5)), True)
```
In [3]: b = galois.Poly([1,0,2,4], field=GF); b, galois.is_irreducible(b)
Out[3]: (Poly(x^3 + 2x + 4, GF(5)), True)

In [4]: c = galois.Poly([1,4,1], field=GF); c, galois.is_irreducible(c)
Out[4]: (Poly(x^2 + 4x + 1, GF(5)), True)

In [5]: f = a * b * c**3; f
Out[5]: Poly(x^10 + 2x^9 + 3x^8 + x^7 + x^6 + 2x^5 + 3x^3 + 4x, GF(5))

The square-free factorization is \(\{x(x^3 + 2x + 4), x^2 + 4x + 1\}\) with multiplicities \(\{1,3\}\).

In [6]: galois.square_free_factorization(f)
Out[6]: ([Poly(x^4 + 2x^2 + 4x, GF(5)), Poly(x^2 + 4x + 1, GF(5))], [1, 3])

In [7]: [a*b, c], [1, 3]
Out[7]: ([Poly(x^4 + 2x^2 + 4x, GF(5)), Poly(x^2 + 4x + 1, GF(5))], [1, 3])

---

galois.distinct_degree_factorization

**galois.distinct_degree_factorization**(poly)

Factors the monic, square-free polynomial \(f(x)\) into a product of polynomials whose irreducible factors all have the same degree.

**Parameters**

- poly (galois.Poly) – A monic, square-free polynomial \(f(x)\) over GF\(\left(p^m\right)\).

**Returns**

- list – The list of polynomials \(f_i(x)\) whose irreducible factors all have degree \(i\).
- list – The list of corresponding distinct degrees \(i\).

**Notes**

The Distinct-Degree Factorization algorithm factors a square-free polynomial \(f(x)\) with degree \(d\) into a product of \(d\) polynomials \(f_i(x)\), where \(f_i(x)\) is the product of all irreducible factors of \(f(x)\) with degree \(i\).

\[
f(x) = \prod_{i=1}^{d} f_i(x)
\]

For example, suppose \(f(x) = x(x+1)(x^2+x+1)(x^3+x+1)(x^3+x^2+1)\) over GF\(\left(2\right)\), then the distinct-degree factorization is

\[
\begin{align*}
f_1(x) &= x(x+1) = x^2 + x \\
f_2(x) &= x^2 + x + 1 \\
f_3(x) &= (x^3 + x + 1)(x^3 + x^2 + 1) = x^6 + x^5 + x^4 + x^3 + x^2 + x + 1 \\
f_i(x) &= 1 \text{ for } i = 4, \ldots, 10.
\end{align*}
\]

Some \(f_i(x) = 1\), but those polynomials are not returned by this function. In this example, the function returns \(\{f_1(x), f_2(x), f_3(x)\}\) and \(\{1,2,3\}\).

The Distinct-Degree Factorization algorithm is often applied after the Square-Free Factorization algorithm, see `galois.square_free_factorization()`. A complete polynomial factorization is implemented in `galois.factors()`.


References

- 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, galois.is_irreducible(a)
Out[1]: (Poly(x, GF(2)), True)

In [2]: b = galois.Poly([1,1]); b, galois.is_irreducible(b)
Out[2]: (Poly(x + 1, GF(2)), True)

In [3]: c = galois.Poly([1,1,1]); c, galois.is_irreducible(c)
Out[3]: (Poly(x^2 + x + 1, GF(2)), True)

In [4]: d = galois.Poly([1,0,1,1]); d, galois.is_irreducible(d)
Out[4]: (Poly(x^3 + x + 1, GF(2)), True)

In [5]: e = galois.Poly([1,1,0,1]); e, galois.is_irreducible(e)
Out[5]: (Poly(x^3 + x^2 + 1, GF(2)), True)

In [6]: f = a * b * c * d * e; f
Out[6]: Poly(x^10 + x^9 + x^8 + x^3 + x^2 + x, GF(2))
```

The distinct-degree factorization is \( \{x(x + 1), x^2 + x + 1, (x^3 + x + 1)(x^3 + x^2 + 1)\} \) whose irreducible factors have degrees \{1, 2, 3\}.

```
In [7]: galois.distinct_degree_factorization(f)
Out[7]: ([Poly(x^2 + x, GF(2)), Poly(x^2 + x + 1, GF(2)), Poly(x^6 + x^5 + x^4 + x^3 + x^2 + x + 1, GF(2))], [1, 2, 3])

In [8]: [a*b, c, d*e], [1, 2, 3]
Out[8]: ([Poly(x^2 + x, GF(2)), Poly(x^2 + x + 1, GF(2)), Poly(x^6 + x^5 + x^4 + x^3 + x^2 + x + 1, GF(2))], [1, 2, 3])
```
**galois.equal_degree_factorization**

`galois.equal_degree_factorization(poly, degree)`

Factors the monic, square-free polynomial \( f(x) \) of degree \( rd \) into a product of \( r \) irreducible factors with degree \( d \).

**Parameters**

- **poly** ([`galois.Poly`]) – A monic, square-free polynomial \( f(x) \) over \( GF(p^m) \).
- **degree** ([`int`]) – The degree \( d \) of each irreducible factor of \( f(x) \).

**Returns**

The list of \( r \) irreducible factors \( \{ g_1(x), \ldots, g_r(x) \} \) in lexicographically-increasing order.

**Return type** list

**Notes**

The Equal-Degree Factorization algorithm factors a square-free polynomial \( f(x) \) with degree \( rd \) into a product of \( r \) irreducible polynomials each with degree \( d \). This function implements the Cantor-Zassenhaus algorithm, which is probabilistic.

The Equal-Degree Factorization algorithm is often applied after the Distinct-Degree Factorization algorithm, see `galois.distinct_degree_factorization()`. A complete polynomial factorization is implemented in `galois.factors()`.

**References**

- Section 2.3 from https://people.csail.mit.edu/dmoshkov/courses/codes/poly-factorization.pdf
- Section 1 from https://www.csa.iisc.ac.in/~chandan/courses/CNT/notes/lec8.pdf

**Examples**

Factor a product of degree-1 irreducible polynomials over \( GF(2) \).

```
In [1]: a = galois.Poly([1,0]); a, galois.is_irreducible(a)
Out[1]: (Poly(x, GF(2)), True)

In [2]: b = galois.Poly([1,1]); b, galois.is_irreducible(b)
Out[2]: (Poly(x + 1, GF(2)), True)

In [3]: f = a * b; f
Out[3]: Poly(x^2 + x, GF(2))

In [4]: galois.equal_degree_factorization(f, 1)
Out[4]: [Poly(x, GF(2)), Poly(x + 1, GF(2))]
```

Factor a product of degree-3 irreducible polynomials over \( GF(5) \).

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

In [6]: a = galois.Poly([1,0,2,1], field=GF); a, galois.is_irreducible(a)
Out[6]: (Poly(x^3 + 2x + 1, GF(5)), True)
```
Polynomial tests

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>is_monic(poly)</code></td>
<td>Determines whether the polynomial is monic, i.e., having leading coefficient equal to 1.</td>
</tr>
<tr>
<td><code>is_irreducible(poly)</code></td>
<td>Determines whether the polynomial $f(x)$ over $GF(p^m)$ is irreducible.</td>
</tr>
<tr>
<td><code>is_primitive(poly)</code></td>
<td>Determines whether the polynomial $f(x)$ over $GF(q)$ is primitive.</td>
</tr>
<tr>
<td><code>is_square_free(value)</code></td>
<td>Determines if the positive integer or the non-constant, monic polynomial is square-free.</td>
</tr>
</tbody>
</table>

**galois.is_monic**

`galois.is_monic(poly)`

Determines whether the polynomial is monic, i.e., having leading coefficient equal to 1.

**Parameters**

- `poly (galois.Poly)` – A polynomial over a Galois field.

**Returns**

- `True` if the polynomial is monic.

**Return type**

`bool`

**Examples**

```
In [1]: GF = galois.GF(7)
In [2]: p = galois.Poly([1,0,4,5], field=GF); p
Out[2]: Poly(x^3 + 4x + 5, GF(7))
In [3]: galois.is_monic(p)
Out[3]: True
In [4]: p = galois.Poly([3,0,4,5], field=GF); p
Out[4]: Poly(3x^3 + 4x + 5, GF(7))
In [5]: galois.is_monic(p)
Out[5]: False
```
### galois.is_square_free

#### galois.is_square_free(value)
Determines if the positive integer or the non-constant, monic polynomial is square-free.

- **Parameters**
  - `value` (int, galois.Poly) – A positive integer `n` or a non-constant, monic polynomial `f(x)`.

- **Returns**
  - `True` if the integer or polynomial is square-free.

- **Return type**
  - bool

#### Notes
A square-free integer `n` is divisible by no perfect squares. As a consequence, the prime factorization of a square-free integer `n` is

\[ n = \prod_{i=1}^{k} p_i^{e_i} = \prod_{i=1}^{k} p_i. \]

Similarly, a square-free polynomial `f(x)` has no irreducible factors with multiplicity greater than one. Therefore, its canonical factorization is

\[ f(x) = \prod_{i=1}^{k} g_i(x)^{e_i} = \prod_{i=1}^{k} g_i(x). \]

#### Examples

Determine if an integer is square-free.

```
In [1]: galois.is_square_free(10)
Out[1]: True

In [2]: galois.is_square_free(16)
Out[2]: False
```

Determine if a polynomial is square-free over GF(3).

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

In [4]: g3 = galois.irreducible_poly(3, 3); g3
Out[4]: Poly(x^3 + 2x + 1, GF(3))

In [5]: g4 = galois.irreducible_poly(3, 4); g4
Out[5]: Poly(x^4 + x + 2, GF(3))

In [6]: galois.is_square_free(g3 * g4)
Out[6]: True

In [7]: galois.is_square_free(g3**2 * g4)
Out[7]: False
```
7.3 Forward Error Correcting Codes

This section contains classes and functions for constructing forward error correction codes.

7.3.1 FEC classes

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BCH(n, k[, ...])</td>
<td>A primitive, narrow-sense binary BCH(n, k) code.</td>
</tr>
<tr>
<td>ReedSolomon(n, k[, ...])</td>
<td>A general RS(n, k) code.</td>
</tr>
</tbody>
</table>

galois.BCH

class galois.BCH(n, k, primitive_poly=None, primitive_element=None, systematic=True)
A primitive, narrow-sense binary BCH(n, k) code.

A BCH(n, k) code is a [n, k, d]2 linear block code with codeword size n, message size k, minimum distance d, and symbols taken from an alphabet of size 2.

To create the shortened BCH(n − s, k − s) code, construct the full-sized BCH(n, k) code and then pass k − s bits into encode() and n − s bits into decode(). Shortened codes are only applicable for systematic codes.

Examples

Construct the BCH code.

```python
In [1]: galois.bch_valid_codes(15)
Out[1]: [(15, 11, 1), (15, 7, 2), (15, 5, 3), (15, 1, 7)]

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

 Encode a message.

```python
In [3]: m = galois.GF2.Random(bch.k); m
Out[3]: GF([1, 1, 1, 1, 1, 1, 1], order=2)

In [4]: c = bch.encode(m); c
Out[4]: GF([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1], order=2)

Corrupt the codeword and decode the message.

```python
# Corrupt the first bit in the codeword
In [5]: c[0] ^= 1

In [6]: dec_m = bch.decode(c); dec_m
Out[6]: GF([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1], order=2)

In [7]: np.array_equal(dec_m, m)
Out[7]: True
```
In this section, we will discuss the Galois field arithmetic and its applications in coding theory. The Galois field, denoted as \( \mathbb{GF}(2^m) \), is a finite field with \( 2^m \) elements. This field is constructed by taking the residue classes of polynomials under modulo addition and multiplication.

### Constructors

**`__init__`**

\[ \text{Constructs a primitive, narrow-sense binary } \ \text{BCH}(n, k) \text{ code.} \]

**Parameters**

- \( n \) (int) – The codeword size \( n \), must be \( n = 2^m - 1 \).
- \( k \) (int) – The message size \( k \).
- \( \text{primitive\_poly} \) (galois.Poly, optional) – Optionally specify the \( g(x) \) whose roots are \( \alpha, \alpha^2, \ldots, \alpha^{2t} \).
- \( \text{systematic} \) (bool, default=True) – Indicates if the code is configured to return codewords in systematic form.

### Methods

#### `decode` (codeword[, errors])

Decodes the BCH codeword \( c \) into the message \( m \).  

#### `detect` (codeword)

Detects if errors are present in the BCH codeword \( c \).  

#### `encode` (message[, parity_only])

Encodes the message \( m \) into the BCH codeword \( c \).  

### Attributes

- \( G \) (galois.Poly) – The generator matrix \( G \) with shape \((k, n)\).
- \( H \) (galois.Poly) – The parity-check matrix \( H \) with shape \((2t, n)\).
- \( d \) (int) – The design distance \( d \) of the \([n, k, d]\) code.
- \( \text{field} \) (galois.GF) – The Galois field \( \mathbb{GF}(2^m) \) that defines the BCH code.
- \( \text{generator\_poly} \) (galois.Poly) – The generator polynomial \( g(x) \) whose roots are \( \alpha, \alpha^2, \ldots, \alpha^{2t} \).
- \( \text{is\_narrow\_sense} \) (bool) – Indicates if the BCH code is narrow sense, meaning the roots of the generator polynomial are consecutive powers of \( \alpha \) starting at 1, i.e. \( \alpha, \alpha^2, \ldots, \alpha^{2t} \).
- \( \text{is\_primitive} \) (bool) – Indicates if the BCH code is primitive, meaning \( n = 2^m - 1 \).
- \( \text{roots} \) (list) – The \( 2t \) roots of the generator polynomial.
- \( \text{systematic} \) (bool) – Indicates if the code is configured to return codewords in systematic form.
- \( t \) (int) – The error-correcting capability of the code.

#### `__init__` (n, k[, primitive\_poly, ...])

Constructs a primitive, narrow-sense binary \( \text{BCH}(n, k) \) code.

Parameters

- \( n \) (int) – The codeword size \( n \), must be \( n = 2^m - 1 \).
- \( k \) (int) – The message size \( k \).
- \( \text{primitive\_poly} \) (galois.Poly, optional) – Optionally specify the \( g(x) \) whose roots are \( \alpha, \alpha^2, \ldots, \alpha^{2t} \).
- \( \text{systematic} \) (bool, default=True) – Indicates if the code is configured to return codewords in systematic form.
Matlab’s default, see `galois.matlab_primitive_poly()`. Matlab tends to use the lexicographically-minimal primitive polynomial as a default instead of the Conway polynomial.

- **primitive_element** (int, `galois.Poly`, optional) – Optionally specify the primitive element \( \alpha \) whose powers are roots of the generator polynomial \( g(x) \). The default is `None` which uses the lexicographically-minimal primitive element in \( \text{GF}(2^m) \), see `galois.primitive_element()`.

- **systematic** (bool, optional) – Optionally specify if the encoding should be systematic, meaning the codeword is the message with parity appended. The default is `True`.

Returns A primitive, narrow-sense binary BCH \((n, k)\) code object.

Return type `galois.BCH`

`decode(codeword, errors=False)`

Decodes the BCH codeword \( c \) into the message \( m \).

Parameters

- **codeword** (numpy.ndarray, `galois.FieldArray`) – The codeword as either a \( n \)-length vector or \((N, n)\) matrix, where \( N \) is the number of codewords. For systematic codes, codeword lengths less than \( n \) may be provided for shortened codewords.

- **errors** (bool, optional) – Optionally specify whether to return the number of corrected errors.

Returns

- **numpy.ndarray, galois.FieldArray** – The decoded message as either a \( k \)-length vector or \((N, k)\) matrix.

- **int, np.ndarray** – Optional return argument of the number of corrected bit errors as either a scalar or \( n \)-length vector. Valid number of corrections are in \([0, t]\). If a codeword has too many errors and cannot be corrected, `-1` will be returned.

Notes

The codeword vector \( c \) is defined as \( c = [c_{n-1}, \ldots, c_1, c_0] \in \text{GF}(2)^n \), which corresponds to the codeword polynomial \( c(x) = c_{n-1}x^{n-1} + \cdots + c_1x + c_0 \). The message vector \( m \) is defined as \( m = [m_{k-1}, \ldots, m_1, m_0] \in \text{GF}(2)^k \), which corresponds to the message polynomial \( m(x) = m_{k-1}x^{k-1} + \cdots + m_1x + m_0 \).

In decoding, the syndrome vector \( s \) is computed by \( s = cH^T \), where \( H \) is the parity-check matrix. The equivalent polynomial operation is \( s(x) = c(x) \mod g(x) \). A syndrome of zeros indicates the received codeword is a valid codeword and there are no errors. If the syndrome is non-zero, the decoder will find an error-locator polynomial \( \sigma(x) \) and the corresponding error locations and values.

For the shortened BCH \((n - s, k - s)\) code (only applicable for systematic codes), pass \( n - s \) bits into `decode()` to return the \( k - s \)-bit message.
Examples

Decode a single codeword.

In [1]: bch = galois.BCH(15, 7)

In [2]: m = galois.GF2.Random(bch.k); m
Out[2]: GF([0, 0, 0, 1, 0, 0], order=2)

In [3]: c = bch.encode(m); c
Out[3]: GF([0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0], order=2)

# Corrupt the first bit in the codeword
In [4]: c[0] ^= 1

In [5]: dec_m = bch.decode(c); dec_m
Out[5]: GF([0, 0, 0, 1, 0, 0], order=2)

In [6]: np.array_equal(dec_m, m)
Out[6]: True

# Instruct the decoder to return the number of corrected bit errors
In [7]: dec_m, N = bch.decode(c, errors=True); dec_m, N
Out[7]: (GF([0, 0, 0, 1, 0, 0], order=2), 1)

In [8]: np.array_equal(dec_m, m)
Out[8]: True

Decode a single, shortened codeword.

In [9]: m = galois.GF2.Random(bch.k - 3); m
Out[9]: GF([0, 0, 1], order=2)

In [10]: c = bch.encode(m); c
Out[10]: GF([0, 0, 0, 1, 0, 1, 0, 1, 0], order=2)

# Corrupt the first bit in the codeword
In [11]: c[0] ^= 1

In [12]: dec_m = bch.decode(c); dec_m
Out[12]: GF([0, 0, 0, 1, 0, 0], order=2)

In [13]: np.array_equal(dec_m, m)
Out[13]: True

Decode a matrix of codewords.

In [14]: m = galois.GF2.Random((5, bch.k)); m
Out[14]: GF([[1, 1, 1, 1, 0, 1, 1],
            [0, 0, 0, 1, 0, 1, 0],
            [0, 0, 1, 1, 0, 1, 1],
            [1, 0, 1, 0, 1, 0, 0],
            [1, 1, 0, 0, 0, 0, 1]], order=2)
In [15]:  c = bch.encode(m); c
Out[15]:
GF([[1, 1, 1, 1, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 1],
    [0, 0, 0, 0, 1, 0, 1, 0, 0, 1, 1, 0, 1, 1, 1],
    [1, 0, 1, 0, 1, 0, 0, 0, 0, 1, 1, 0, 1, 1, 1],
    [1, 1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 1, 1, 0, 1]], order=2)

# Corrupt the first bit in each codeword
In [16]:  c[:,0] ^= 1

In [17]:  dec_m = bch.decode(c); dec_m
Out[17]:
GF([[1, 1, 1, 1, 0, 1, 1],
    [0, 0, 0, 0, 1, 0, 1],
    [0, 0, 1, 1, 0, 1, 1],
    [1, 0, 1, 0, 1, 0, 0],
    [1, 1, 0, 0, 0, 0, 1]], order=2)

In [18]:  np.array_equal(dec_m, m)
Out[18]: True

# Instruct the decoder to return the number of corrected bit errors
In [19]:  dec_m, N = bch.decode(c, errors=True); dec_m, N
Out[19]:
(GF([[1, 1, 1, 1, 0, 1, 1],
    [0, 0, 0, 0, 1, 0, 1],
    [0, 0, 1, 1, 0, 1, 1],
    [1, 0, 1, 0, 1, 0, 0],
    [1, 1, 0, 0, 0, 0, 1]], order=2),
array([[1, 1, 1, 1]]))

In [20]:  np.array_equal(dec_m, m)
Out[20]: True

detect(codeword)

Detects if errors are present in the BCH codeword c.

The \([n, k, d]_2\) BCH code has \(d_{\text{min}} \geq d\) minimum distance. It can detect up to \(d_{\text{min}} - 1\) errors.

Parameters  

codeword (numpy.ndarray, galois.FieldArray) – The codeword as either a \(n\)-length vector or \((N, n)\) matrix, where \(N\) is the number of codewords. For systematic codes, codeword lengths less than \(n\) may be provided for shortened codewords.

Returns  

A boolean scalar or array indicating if errors were detected in the corresponding codeword True or not False.

Return type  

bool, numpy.ndarray
Examples

Detect errors in a valid codeword.

In [1]: bch = galois.BCH(15, 7)
   # The minimum distance of the code
In [2]: bch.d
Out[2]: 5
In [3]: m = galois.GF2.Random(bch.k); m
Out[3]: GF([1, 0, 0, 1, 1, 0, 1], order=2)
In [4]: c = bch.encode(m); c
Out[4]: GF([1, 0, 0, 1, 1, 0, 1, 1, 1, 0, 0, 0, 1, 0], order=2)
In [5]: bch.detect(c)
Out[5]: False

Detect $d_{\text{min}} - 1$ errors in a received codeword.

# Corrupt the first $d - 1$ bits in the codeword
In [6]: c[0:bch.d - 1] ^= 1
In [7]: bch.detect(c)
Out[7]: True

**encode**(*message, parity_only=False*)

Encodes the message $m$ into the BCH codeword $c$.

**Parameters**

- *message* (*numpy.ndarray, galois.FieldArray*) – The message as either a $k$-length vector or $(N, k)$ matrix, where $N$ is the number of messages. For systematic codes, message lengths less than $k$ may be provided to produce shortened codewords.

- *parity_only* (*bool, optional*) – Optionally specify whether to return only the parity bits. This only applies to systematic codes. The default is False.

**Returns**

The codeword as either a $n$-length vector or $(N, n)$ matrix. The return type matches the message type. If parity_only=True, the parity bits are returned as either a $n - k$-length vector or $(N, n - k)$ matrix.

**Return type**

*numpy.ndarray, galois.FieldArray*

**Notes**

The message vector $m$ is defined as $m = [m_{k-1}, \ldots, m_1, m_0] \in \text{GF}(2)^k$, which corresponds to the message polynomial $m(x) = m_{k-1}x^{k-1} + \cdots + m_1x + m_0$. The codeword vector $c$ is defined as $c = [c_{n-1}, \ldots, c_1, c_0] \in \text{GF}(2)^n$, which corresponds to the codeword polynomial $c(x) = c_{n-1}x^{n-1} + \cdots + c_1x + c_0$.

The codeword vector is computed from the message vector by $c = mG$, where $G$ is the generator matrix. The equivalent polynomial operation is $c(x) = m(x)g(x)$. For systematic codes, $G = [I \mid P]$ such that $c = [m \mid p]$. And in polynomial form, $p(x) = -(m(x)x^{n-k} \mod g(x))$ with $c(x) = m(x)x^{n-k} + p(x)$. For systematic and non-systematic codes, each codeword is a multiple of the generator polynomial, i.e. $g(x) \mid c(x)$.

7.3. Forward Error Correcting Codes
For the shortened BCH \((n - s, k - s)\) code (only applicable for systematic codes), pass \(k - s\) bits into `encode()` to return the \(n - s\)-bit codeword.

### Examples

Encode a single codeword.

```python
In [1]: bch = galois.BCH(15, 7)

In [2]: m = galois.GF2.Random(bch.k); m
Out[2]: GF([0, 0, 1, 1, 0, 0, 1], order=2)

In [3]: c = bch.encode(m); c
Out[3]: GF([0, 0, 1, 1, 0, 0, 1, 1, 1, 1, 1, 0, 1, 1, 0], order=2)

In [4]: p = bch.encode(m, parity_only=True); p
Out[4]: GF([1, 1, 1, 1, 0, 1, 1, 0], order=2)
```

Encode a single, shortened codeword.

```python
In [5]: m = galois.GF2.Random(bch.k - 3); m
Out[5]: GF([0, 1, 1, 0], order=2)

In [6]: c = bch.encode(m); c
Out[6]: GF([0, 1, 1, 0, 1, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0], order=2)
```

Encode a matrix of codewords.

```python
In [7]: m = galois.GF2.Random((5, bch.k)); m
Out[7]:
GF([[1, 1, 1, 0, 1, 1, 0],
    [0, 0, 1, 0, 1, 1, 0],
    [1, 0, 0, 0, 0, 0, 0],
    [1, 1, 0, 0, 0, 0, 1],
    [0, 1, 0, 0, 0, 0, 1]], order=2)

In [8]: c = bch.encode(m); c
Out[8]:
GF([[1, 1, 1, 0, 1, 1, 0, 1, 1, 0, 0, 1, 1, 1, 1],
    [0, 0, 1, 0, 1, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0],
    [1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0],
    [1, 1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0],
    [0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 1, 0, 1, 0]], order=2)

In [9]: p = bch.encode(m, parity_only=True); p
Out[9]:
GF([[0, 0, 1, 1, 0, 0, 1, 1],
    [1, 0, 1, 0, 1, 1, 1, 1],
    [1, 1, 0, 1, 0, 0, 1, 1, 0, 1, 0, 0, 0, 0, 0],
    [0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1]], order=2)
```

property \(G\)

The generator matrix \(G\) with shape \((k, n)\).
Examples

```
In [1]: bch = galois.BCH(15, 7); bch
Out[1]: <BCH Code: [15, 7, 5] over GF(2)>

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

Type `galois.GF2`

property H

The parity-check matrix $H$ with shape $(2t, n)$.

Examples

```
In [1]: bch = galois.BCH(15, 7); bch
Out[1]: <BCH Code: [15, 7, 5] over GF(2)>

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

Type `galois.FieldArray`

property d

The design distance $d$ of the $[n, k, d]_2$ code. The minimum distance of a BCH code may be greater than the design distance, $d_{\text{min}} \geq d$.

Examples

```
In [1]: bch = galois.BCH(15, 7); bch
Out[1]: <BCH Code: [15, 7, 5] over GF(2)>

In [2]: bch.d
Out[2]: 5
```

Type `int`

7.3. Forward Error Correcting Codes
**property field**

The Galois field \( GF(2^m) \) that defines the BCH code.

**Examples**

```python
In [1]: bch = galois.BCH(15, 7); bch
Out[1]: <BCH Code: [15, 7, 5] over GF(2)>

In [2]: bch.field
Out[2]: <class 'numpy.ndarray over GF(2^4)'>

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

Type `galois.FieldClass`

**property generator_poly**

The generator polynomial \( g(x) \) whose roots are `roots`.

**Examples**

```python
In [1]: bch = galois.BCH(15, 7); bch
Out[1]: <BCH Code: [15, 7, 5] over GF(2)>

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

# Evaluate the generator polynomial at its roots in GF(2^m)
In [3]: bch.generator_poly(bch.roots, field=bch.field)
Out[3]: GF([0, 0, 0, 0], order=2^4)
```

Type `galois.Poly`

**property is_narrow_sense**

Indicates if the BCH code is narrow sense, meaning the roots of the generator polynomial are consecutive powers of \( \alpha \) starting at 1, i.e. \( \alpha, \alpha^2, \ldots, \alpha^{2t} \).
Examples

```python
In [1]: bch = galois.BCH(15, 7); bch
Out[1]: <BCH Code: [15, 7, 5] over GF(2)>

In [2]: bch.is_narrow_sense
Out[2]: True

In [3]: bch.roots
Out[3]: GF([2, 4, 8, 3], order=2^4)

In [4]: bch.field.primitive_element**(np.arange(1, 2*bch.t + 1))
Out[4]: GF([2, 4, 8, 3], order=2^4)
```

**property is_primitive**
Indicates if the BCH code is primitive, meaning \(n = 2^m - 1\).

Examples

```python
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
```

**property k**
The message size \(k\) of the \([n, k, d]_2\) code

Examples

```python
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
```

**property n**
The codeword size \(n\) of the \([n, k, d]_2\) code
Examples

```
In [1]: bch = galois.BCH(15, 7); bch
Out[1]: <BCH Code: [15, 7, 5] over GF(2)>

In [2]: bch.n
Out[2]: 15
```

**Type** int

**property roots**
The $2t$ roots of the generator polynomial. These are consecutive powers of $\alpha$, specifically $\alpha, \alpha^2, \ldots, \alpha^{2t}$.

```
In [1]: bch = galois.BCH(15, 7); bch
Out[1]: <BCH Code: [15, 7, 5] over GF(2)>

In [2]: bch.roots
Out[2]: GF([2, 4, 8, 3], order=2^4)

# Evaluate the generator polynomial at its roots in GF(2^m)
In [3]: bch.generator_poly(bch.roots, field=bch.field)
Out[3]: GF([0, 0, 0, 0], order=2^4)
```

**Type** galois.FieldArray

**property systematic**
Indicates if the code is configured to return codewords in systematic form.

```
In [1]: bch = galois.BCH(15, 7); bch
Out[1]: <BCH Code: [15, 7, 5] over GF(2)>

In [2]: bch.systematic
Out[2]: True
```

**Type** bool

**property t**
The error-correcting capability of the code. The code can correct $t$ bit errors in a codeword.
Examples

```python
In [1]: bch = galois.BCH(15, 7); bch
Out[1]: <BCH Code: [15, 7, 5] over GF(2)>

In [2]: bch.t
Out[2]: 2
```

**Type** int

**galois.ReedSolomon**

```python
class galois.ReedSolomon(n, k, c=1, primitive_poly=None, primitive_element=None, systematic=True)

A general RS\((n, k)\) code.

A RS\((n, k)\) code is a \([n, k, d]_q\) linear block code with codeword size \(n\), message size \(k\), minimum distance \(d\), and symbols taken from an alphabet of size \(q\) (a prime power).

To create the shortened RS\((n - s, k - s)\) code, construct the full-sized RS\((n, k)\) code and then pass \(k - s\) symbols into `encode()` and \(n - s\) symbols into `decode()`.
```

**Examples**

Construct the Reed-Solomon code.

```python
In [1]: rs = galois.ReedSolomon(15, 9)

In [2]: GF = rs.field
```

Encode a message.

```python
In [3]: m = GF.Random(rs.k); m
Out[3]: GF([15, 2, 5, 5, 13, 9, 6, 6, 4], order=2^4)

In [4]: c = rs.encode(m); c
Out[4]: GF([15, 2, 5, 5, 13, 9, 6, 6, 4, 7, 2, 6, 10, 7, 7], order=2^4)
```

Corrupt the codeword and decode the message.

```python
# Corrupt the first symbol in the codeword
In [5]: c[0] ^= 13

In [6]: dec_m = rs.decode(c); dec_m
Out[6]: GF([15, 2, 5, 5, 13, 9, 6, 6, 4], order=2^4)

In [7]: np.array_equal(dec_m, m)
Out[7]: True

# Instruct the decoder to return the number of corrected symbol errors
In [8]: dec_m, N = rs.decode(c, errors=True); dec_m, N
```

(continues on next page)
Out[8]: (GF([15, 2, 5, 5, 13, 9, 6, 6, 4], order=2^4), 1)
In [9]: np.array_equal(dec_m, m)
Out[9]: True

Constructors

__init__(n, k[, c, primitive_poly, ...]) Constructs a general RS(n, k) code.

Methods

decode(codeword[, errors]) Decodes the Reed-Solomon codeword c into the message m.
detect(codeword) Detects if errors are present in the Reed-Solomon codeword c.
encode(message[, parity_only]) Encodes the message m into the Reed-Solomon codeword c.

Attributes

G The generator matrix G with shape (k, n).
H The parity-check matrix H with shape (2t, n).
c The degree of the first consecutive root.
d The design distance d of the [n, k, d]q code.
field The Galois field GF(q) that defines the Reed-Solomon code.
generator_poly The generator polynomial g(x) whose roots are roots.
is_narrow_sense Indicates if the Reed-Solomon code is narrow sense, meaning the roots of the generator polynomial are consecutive powers of α starting at 1, i.e. α, α^2, ..., α^{2^t-1}.
k The message size k of the [n, k, d]q code.
n The codeword size n of the [n, k, d]q code.
roots The 2t roots of the generator polynomial.
systematic Indicates if the code is configured to return codewords in systematic form.
t The error-correcting capability of the code.

__init__(n, k, c=1, primitive_poly=None, primitive_element=None, systematic=True) Constructs a general RS(n, k) code.

Parameters

- n (int) – The codeword size n, must be n = q – 1 where q is a prime power.
- k (int) – The message size k. The error-correcting capability t is defined by n – k = 2t.
- **c** *(int, optional)* – The first consecutive power of \( \alpha \). The default is 1.

- **primitive_poly** *(galois.Poly, optional)* – Optionally specify the primitive polynomial that defines the extension field \( \text{GF}(q) \). The default is None which uses Mat- 
  lab’s default, see *galois.matlab_primitive_poly()*.

- **primitive_element** *(int, galois.Poly, optional)* – Optionally specify the prime-
  tive element \( \alpha \) of \( \text{GF}(q) \) whose powers are roots of the generator polynomial \( g(x) \). The default is None which uses the lexicographically-minimal primitive element in \( \text{GF}(q) \), see *galois.primitive_element()*.

- **systematic** *(bool, optional)* – Optionally specify if the encoding should be sys-
  tematic, meaning the codeword is the message with parity appended. The default is True.

Returns A general RS\((n, k)\) code object.

Return type galois.ReedSolomon

decode(codeword, errors=False)

Decodes the Reed-Solomon codeword \( c \) into the message \( m \).

Parameters

- **codeword** *(numpy.ndarray, galois.FieldArray)* – The codeword as either a \( n \)-
  length vector or \((N, n)\) matrix, where \( N \) is the number of codewords. For systematic codes,
  codeword lengths less than \( n \) may be provided for shortened codewords.

- **errors** *(bool, optional)* – Optionally specify whether to return the number of cor-
  rected errors.

Returns

- **numpy.ndarray, galois.FieldArray** – The decoded message as either a \( k \)-length vector or
  \((N, k)\) matrix.

- **int, np.ndarray** – Optional return argument of the number of corrected symbol errors as
  either a scalar or \( n \)-length vector. Valid number of corrections are in \([0, t]\) if a codeword
  has too many errors and cannot be corrected, -1 will be returned.

Notes

The codeword vector \( c \) is defined as \( c = [c_{n-1}, \ldots, c_1, c_0] \in \text{GF}(q)^n \), which corresponds to the codeword polynomial \( c(x) = c_{n-1}x^{n-1} + \cdots + c_1x + c_0 \). The message vector \( m \) is defined as \( m = [m_{k-1}, \ldots, m_1, m_0] \in \text{GF}(q)^k \), which corresponds to the message polynomial \( m(x) = m_{k-1}x^{k-1} + \cdots + m_1x + m_0 \).

In decoding, the syndrome vector \( s \) is computed by \( s = cH^T \), where \( H \) is the parity-check matrix. The equivalent polynomial operation is \( s(x) = c(x) \mod g(x) \). A syndrome of zeros indicates the received codeword is a valid codeword and there are no errors. If the syndrome is non-zero, the decoder will find an error-locator polynomial \( \sigma(x) \) and the corresponding error locations and values.

For the shortened RS\((n-s, k-s)\) code (only applicable for systematic codes), pass \( n-s \) symbols into *decode()* to return the \( k-s \)-symbol message.
Examples

Decode a single codeword.

```python
In [1]: rs = galois.ReedSolomon(15, 9)
In [2]: GF = rs.field
In [3]: m = GF.Random(rs.k); m
Out[3]: GF([11, 14, 12, 1, 4, 8, 5, 9, 1], order=2^4)
In [4]: c = rs.encode(m); c
Out[4]: GF([11, 14, 12, 1, 4, 8, 5, 9, 1, 8, 3, 1, 0, 2, 14], order=2^4)
# Corrupt the first symbol in the codeword
In [5]: c[0] += GF(13)
In [6]: dec_m = rs.decode(c); dec_m
Out[6]: GF([11, 14, 12, 1, 4, 8, 5, 9, 1], order=2^4)
In [7]: np.array_equal(dec_m, m)
Out[7]: True
# Instruct the decoder to return the number of corrected symbol errors
In [8]: dec_m, N = rs.decode(c, errors=True); dec_m, N
Out[8]: (GF([11, 14, 12, 1, 4, 8, 5, 9, 1], order=2^4), 1)
In [9]: np.array_equal(dec_m, m)
Out[9]: True
```

Decode a single, shortened codeword.

```python
In [10]: m = GF.Random(rs.k - 4); m
Out[10]: GF([ 7, 14, 5, 7, 3], order=2^4)
In [11]: c = rs.encode(m); c
Out[11]: GF([ 7, 14, 5, 7, 3, 2, 8, 4, 0, 11, 5], order=2^4)
# Corrupt the first symbol in the codeword
In [12]: c[0] += GF(13)
In [13]: dec_m = rs.decode(c); dec_m
Out[13]: GF([ 7, 14, 5, 7, 3], order=2^4)
In [14]: np.array_equal(dec_m, m)
Out[14]: True
```

Decode a matrix of codewords.

```python
In [15]: m = GF.Random((5, rs.k)); m
Out[15]: GF([[15, 1, 12, 14, 12, 7, 1, 8, 10],
          [11, 9, 7, 14, 15, 11, 13, 0, 10],
          [10, 0, 9, 14, 15, 2, 13, 11, 1],
          [15, 8, 12, 14, 12, 11, 0, 10, 9],
          [11, 13, 10, 12, 14, 15, 7, 0, 1]], order=2^4)
```

[12, 4, 13, 4, 6, 15, 15, 9, 0],
[ 9, 5, 6, 2, 8, 8, 2, 0, 10],
[12, 15, 4, 7, 4, 10, 14, 7, 8],
[ 8, 0, 7, 3, 5, 1, 7, 9, 2]], order=2^4)

In [16]: c = rs.encode(m); c
Out[16]:
GF([[15, 1, 12, 14, 12, 7, 1, 8, 10, 0, 14, 6, 2, 2, 10],
[12, 4, 13, 4, 6, 15, 15, 9, 0, 13, 9, 11, 0, 6, 1],
[ 9, 5, 6, 2, 8, 8, 2, 0, 10, 8, 14, 4, 10, 13, 6],
[12, 15, 4, 7, 4, 10, 14, 7, 8, 9, 6, 7, 13, 1, 9],
[ 8, 0, 7, 3, 5, 1, 7, 9, 2]], order=2^4)

# Corrupt the first symbol in each codeword
In [17]: c[:,0] += GF(13)
In [18]:
dec_m = rs.decode(c); dec_m
Out[18]:
GF([[15, 1, 12, 14, 12, 7, 1, 8, 10],
[12, 4, 13, 4, 6, 15, 15, 9, 0],
[ 9, 5, 6, 2, 8, 8, 2, 0, 10],
[12, 15, 4, 7, 4, 10, 14, 7, 8],
[ 8, 0, 7, 3, 5, 1, 7, 9, 2]], order=2^4)

In [19]: np.array_equal(dec_m, m)
Out[19]: True

# Instruct the decoder to return the number of corrected symbol errors
In [20]: dec_m, N = rs.decode(c, errors=True); dec_m, N
Out[20]:
(GF([[15, 1, 12, 14, 12, 7, 1, 8, 10],
[12, 4, 13, 4, 6, 15, 15, 9, 0],
[ 9, 5, 6, 2, 8, 8, 2, 0, 10],
[12, 15, 4, 7, 4, 10, 14, 7, 8],
[ 8, 0, 7, 3, 5, 1, 7, 9, 2]], order=2^4),
array([1, 1, 1, 1, 1]))

In [21]: np.array_equal(dec_m, m)
Out[21]: True

detect(codeword)

Detects if errors are present in the Reed-Solomon codeword c.

The $[n, k, d]_q$ Reed-Solomon code has $d_{\min} = d$ minimum distance. It can detect up to $d_{\min} - 1$ errors.

Parameters

- **codeword** (numpy.ndarray, galois.FieldArray) – The codeword as either an $n$-length vector or $(N, n)$ matrix, where $N$ is the number of codewords. For systematic codes, codeword lengths less than $n$ may be provided for shortened codewords.

Returns

A boolean scalar or array indicating if errors were detected in the corresponding codeword

Return type

bool, numpy.ndarray

7.3. Forward Error Correcting Codes
**Examples**

Detect errors in a valid codeword.

```python
In [1]: rs = galois.ReedSolomon(15, 9)
In [2]: GF = rs.field
# The minimum distance of the code
In [3]: rs.d
Out[3]: 7
In [4]: m = GF.Random(rs.k); m
Out[4]: GF([15, 14, 13, 14, 14, 10, 15, 7, 0], order=2^4)
In [5]: c = rs.encode(m); c
Out[5]: GF([15, 14, 13, 14, 14, 10, 15, 7, 0, 13, 8, 11, 6, 13, 8], order=2^4)
In [6]: rs.detect(c)
Out[6]: False
```

Detect $d_{\text{min}} - 1$ errors in a received codeword.

```python
# Corrupt the first `$d - 1$` symbols in the codeword
In [7]: c[0:rs.d - 1] += GF(13)
In [8]: rs.detect(c)
Out[8]: True
```

```python
**encode**(message, parity_only=False)
```

Encodes the message $m$ into the Reed-Solomon codeword $c$.

**Parameters**

- **message** *(numpy.ndarray, galois.FieldArray)* – The message as either a $k$-length vector or $(N, k)$ matrix, where $N$ is the number of messages. For systematic codes, message lengths less than $k$ may be provided to produce shortened codewords.

- **parity_only** *(bool, optional)* – Optionally specify whether to return only the parity symbols. This only applies to systematic codes. The default is `False`.

**Returns** The codeword as either a $n$-length vector or $(N, n)$ matrix. The return type matches the message type. If `parity_only=True`, the parity symbols are returned as either a $n-k$-length vector or $(N, n - k)$ matrix.

**Return type** `numpy.ndarray, galois.FieldArray`
Notes

The message vector \( m \) is defined as \( m = [m_{k-1}, \ldots, m_1, m_0] \in \text{GF}(q)^k \), which corresponds to the message polynomial \( m(x) = m_{k-1}x^{k-1} + \cdots + m_1x + m_0 \). The codeword vector \( c \) is defined as \( c = [c_{n-1}, \ldots, c_1, c_0] \in \text{GF}(q)^n \), which corresponds to the codeword polynomial \( c(x) = c_{n-1}x^{n-1} + \cdots + c_1x + c_0 \).

The codeword vector is computed from the message vector by \( c = mG \), where \( G \) is the generator matrix. The equivalent polynomial operation is \( c(x) = m(x)g(x) \). For systematic codes, \( G = [I \mid P] \) such that \( c = [m \mid p] \). And in polynomial form, \( p(x) = -(m(x)x^{n-k} \mod g(x)) \) with \( c(x) = m(x)x^{n-k} + p(x) \). For systematic and non-systematic codes, each codeword is a multiple of the generator polynomial, i.e. \( g(x) \mid c(x) \).

For the shortened RS\((n - s, k - s)\) code (only applicable for systematic codes), pass \( k - s \) symbols into \texttt{encode()} to return the \( n - s \)-symbol codeword.

Examples

Encode a single codeword.

```python
In [1]: rs = galois.ReedSolomon(15, 9)
In [2]: GF = rs.field
In [3]: m = GF.Random(rs.k); m
Out[3]: GF([0, 5, 7, 7, 1, 11, 7, 12, 5], order=2^4)
In [4]: c = rs.encode(m); c
Out[4]: GF([0, 5, 7, 7, 1, 11, 7, 12, 5, 7, 5, 4, 14, 2, 3], order=2^4)
In [5]: p = rs.encode(m, parity_only=True); p
Out[5]: GF([7, 5, 4, 14, 2, 3], order=2^4)
```

Encode a single, shortened codeword.

```python
In [6]: m = GF.Random(rs.k - 4); m
Out[6]: GF([11, 3, 14, 0, 14], order=2^4)
In [7]: c = rs.encode(m); c
Out[7]: GF([11, 3, 14, 0, 14, 2, 3, 6, 8, 12, 15], order=2^4)
```

Encode a matrix of codewords.

```python
In [8]: m = GF.Random((5, rs.k)); m
Out[8]: GF([[6, 10, 15, 10, 3, 12, 15, 10, 5],
           [2, 1, 4, 0, 0, 1, 7, 3, 1],
           [6, 10, 5, 13, 13, 10, 6, 1, 11],
           [11, 7, 2, 6, 4, 11, 3, 2, 3],
           [3, 15, 6, 6, 8, 15, 14, 6, 2]], order=2^4)
In [9]: c = rs.encode(m); c
```

(continues on next page)
property \( G \)

The generator matrix \( G \) with shape \((k, n)\).

Examples

```
In [1]: rs = galois.ReedSolomon(15, 9); rs
Out[1]: <Reed-Solomon Code: [15, 9, 7] over GF(2^4)>
```

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

Type \( galois.FieldArray \)

property \( H \)

The parity-check matrix \( H \) with shape \((2t, n)\).
Examples

```python
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, 12,  4, 13,  10,  6,  2, 15,  5,  3,  1],
    [ 7,  6,  1,  7,  6,  1,  7,  6,  1,  7,  6,  1,  7,  6,  1],
    [10,  8, 15, 12,  1, 10,  8, 15, 12,  1, 10,  8, 15, 12,  1]],
    order=2^4)
```

Type `galois.FieldArray`

property `c`
The degree of the first consecutive root.

Examples

```python
In [1]: rs = galois.ReedSolomon(15, 9); rs
Out[1]: <Reed-Solomon Code: [15, 9, 7] over GF(2^4)>

In [2]: rs.c
Out[2]: 1
```

Type `int`

property `d`
The design distance $d$ of the $[n, k, d]_q$ code. The minimum distance of a Reed-Solomon code is exactly equal to the design distance, $d_{min} = d$.

Examples

```python
In [1]: rs = galois.ReedSolomon(15, 9); rs
Out[1]: <Reed-Solomon Code: [15, 9, 7] over GF(2^4)>

In [2]: rs.d
Out[2]: 7
```

Type `int`

property `field`
The Galois field $GF(q)$ that defines the Reed-Solomon code.
Examples

```python
In [1]: rs = galois.ReedSolomon(15, 9); rs
Out[1]: <Reed-Solomon Code: [15, 9, 7] over GF(2^4)>

In [2]: rs.field
Out[2]: <class 'numpy.ndarray over GF(2^4)'>

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

Type `galois.FieldClass`

**property generator_poly**
The generator polynomial \( g(x) \) whose roots are `roots`.

Examples

```python
In [1]: rs = galois.ReedSolomon(15, 9); rs
Out[1]: <Reed-Solomon Code: [15, 9, 7] over GF(2^4)>

In [2]: rs.generator_poly
Out[2]: Poly(x^6 + 7x^5 + 9x^4 + 3x^3 + 12x^2 + 10x + 12, GF(2^4))

# Evaluate the generator polynomial at its roots
In [3]: rs.generator_poly(rs.roots)
Out[3]: GF([0, 0, 0, 0, 0, 0], order=2^4)
```

Type `galois.Poly`

**property is_narrow_sense**
Indicates if the Reed-Solomon code is narrow sense, meaning the roots of the generator polynomial are consecutive powers of \( \alpha \) starting at 1, i.e. \( \alpha, \alpha^2, \ldots, \alpha^{2^t-1} \).

Examples

```python
In [1]: rs = galois.ReedSolomon(15, 9); rs
Out[1]: <Reed-Solomon Code: [15, 9, 7] over GF(2^4)>

In [2]: rs.is_narrow_sense
Out[2]: True

In [3]: rs.roots
Out[3]: GF([ 2, 4, 8, 3, 6, 12], order=2^4)
```
In [4]: rs.field.primitive_element**(np.arange(1, 2*rs.t + 1))
Out[4]: GF([ 2, 4, 8, 3, 6, 12], order=2^4)

Type bool

property k
The message size $k$ of the $[n, k, d]_q$ code.

Examples

In [1]: rs = galois.ReedSolomon(15, 9); rs
Out[1]: <Reed-Solomon Code: [15, 9, 7] over GF(2^4)>

In [2]: rs.k
Out[2]: 9

Type int

property n
The codeword size $n$ of the $[n, k, d]_q$ code.

Examples

In [1]: rs = galois.ReedSolomon(15, 9); rs
Out[1]: <Reed-Solomon Code: [15, 9, 7] over GF(2^4)>

In [2]: rs.n
Out[2]: 15

Type int

property roots
The $2t$ roots of the generator polynomial. These are consecutive powers of $\alpha$, specifically $\alpha^c, \alpha^{c+1}, \ldots, \alpha^{c+2t-1}$.

Examples

In [1]: rs = galois.ReedSolomon(15, 9); rs
Out[1]: <Reed-Solomon Code: [15, 9, 7] over GF(2^4)>

In [2]: rs.roots
Out[2]: GF([ 2, 4, 8, 3, 6, 12], order=2^4)

# Evaluate the generator polynomial at its roots
In [3]: rs.generator_poly(rs.roots)
Out[3]: GF([0, 0, 0, 0, 0, 0], order=2^4)

7.3. Forward Error Correcting Codes
Type `galois.FieldArray`

property systematic
Indicates if the code is configured to return codewords in systematic form.

Examples

```
In [1]: rs = galois.ReedSolomon(15, 9); rs
Out[1]: <Reed-Solomon Code: [15, 9, 7] over GF(2^4)>

In [2]: rs.systematic
Out[2]: True
```

Type `bool`

property t
The error-correcting capability of the code. The code can correct \( t \) symbol errors in a codeword.

Examples

```
In [1]: rs = galois.ReedSolomon(15, 9); rs
Out[1]: <Reed-Solomon Code: [15, 9, 7] over GF(2^4)>

In [2]: rs.t
Out[2]: 3
```

Type `int`

7.3.2 Linear block code functions

<table>
<thead>
<tr>
<th>function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>generator_to_parity_check_matrix(G)</code></td>
<td>Converts the generator matrix ( G ) of a linear ([n, k] ) code into its parity-check matrix ( H ).</td>
</tr>
<tr>
<td><code>parity_check_to_generator_matrix(H)</code></td>
<td>Converts the parity-check matrix ( H ) of a linear ([n, k] ) code into its generator matrix ( G ).</td>
</tr>
</tbody>
</table>

**galois.generator_to_parity_check_matrix**

```
galois.generator_to_parity_check_matrix(G)
```

Converts the generator matrix \( G \) of a linear \([n, k] \) code into its parity-check matrix \( H \).

The generator and parity-check matrices satisfy the equations \( GH^T = 0 \).

**Parameters**

- \( G \) (`galois.FieldArray`) – The \((k, n)\) generator matrix \( G \) in systematic form \( G = [I_{k,k} \mid P_{k,n-k}] \).

**Returns**

- The \((n - k, n)\) parity-check matrix \( H = [-P_{k,n-k}^T \mid I_{n-k,n-k}] \).

**Return type** `galois.FieldArray`
galois

## Examples

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

In [2]: G = galois.poly_to_generator_matrix(7, g); G
Out[2]:
GF([[1, 0, 0, 1, 1, 0, 1],
     [0, 1, 0, 1, 0, 0, 0],
     [0, 0, 1, 0, 0, 0, 1],
     [0, 0, 0, 0, 0, 1, 0],
     [0, 0, 0, 0, 0, 0, 1]], order=2)

In [3]: H = galois.generator_to_parity_check_matrix(G); H
Out[3]:
GF([[1, 1, 1, 0, 0, 0, 0],
     [0, 1, 1, 0, 0, 0, 0],
     [1, 0, 0, 1, 0, 0, 0]], order=2)

In [4]: G @ H.T
Out[4]:
GF([[0, 0, 0, 0, 0, 0, 0],
     [0, 0, 0, 0, 0, 0, 0],
     [0, 0, 0, 0, 0, 0, 0],
     [0, 0, 0, 0, 0, 0, 0],
     [0, 0, 0, 0, 0, 0, 0],
     [0, 0, 0, 0, 0, 0, 0],
     [0, 0, 0, 0, 0, 0, 0]], order=2)
```

### galois.parity_check_to_generator_matrix

**galois.parity_check_to_generator_matrix(H)**

Converts the parity-check matrix \( \mathbf{H} \) of a linear \([n, k]\) code into its generator matrix \( \mathbf{G} \).

The generator and parity-check matrices satisfy the equations \( \mathbf{GH}^T = 0 \).

**Parameters**

- \( \mathbf{H} \) (galois.FieldArray) – The \((n - k, n)\) parity-check matrix \( \mathbf{G} \) in systematic form

\[
\mathbf{H} = [-\mathbf{P}^T_{k, n-k} \mid \mathbf{I}_{n-k, n-k}].
\]

**Returns**

The \((k, n)\) generator matrix \( \mathbf{G} = [\mathbf{I}_{k, k} \mid \mathbf{P}_{k, n-k}] \).

**Return type**

`galois.FieldArray`

## Examples

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

In [2]: G = galois.poly_to_generator_matrix(7, g); G
Out[2]:
GF([[1, 0, 0, 0, 1, 0, 1],
     [0, 1, 0, 0, 1, 1, 1],
     [0, 0, 1, 0, 1, 1, 0],
     [0, 0, 0, 1, 0, 1, 1]], order=2)

In [3]: H = galois.generator_to_parity_check_matrix(G); H
Out[3]:
GF([[1, 1, 1, 0, 1, 0, 0],
     [0, 1, 1, 1, 0, 1, 0],
     [1, 1, 0, 1, 0, 0, 1]], order=2)
```

(continues on next page)
7.3.3 Cyclic code functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>bch_valid_codes(n[, t_min])</code></td>
<td>Returns a list of ((n, k, t)) tuples of valid primitive binary BCH codes.</td>
</tr>
<tr>
<td><code>poly_to_generator_matrix(n, generator_poly)</code></td>
<td>Converts the generator polynomial (g(x)) into the generator matrix (G) for an ([n,k]) cyclic code.</td>
</tr>
<tr>
<td><code>roots_to_parity_check_matrix(n, roots)</code></td>
<td>Converts the generator polynomial roots into the parity-check matrix (H) for an ([n,k]) cyclic code.</td>
</tr>
</tbody>
</table>

**galois.bch_valid_codes**

```
galois.bch_valid_codes(n, t_min=1)
```

Returns a list of \((n, k, t)\) tuples of valid primitive binary BCH codes.

A BCH code with parameters \((n, k, t)\) is represented as a \([n, k, d]_2\) linear block code with \(d = 2t + 1\).

Parameters

- \(n\) (int) – The codeword size \(n\), must be \(n = 2^m - 1\).
- \(t\_min\) (int, optional) – The minimum error-correcting capability. The default is 1.

Returns

A list of \((n, k, t)\) tuples of valid primitive BCH codes.

Return type list
References


Examples

```python
In [1]: galois.bch_valid_codes(31)
Out[1]: [(31, 26, 1), (31, 21, 2), (31, 16, 3), (31, 11, 5), (31, 6, 7), (31, 1, →15)]

In [2]: galois.bch_valid_codes(31, t_min=3)
Out[2]: [(31, 16, 3), (31, 11, 5), (31, 6, 7), (31, 1, 15)]
```

`galois.poly_to_generator_matrix`

`galois.poly_to_generator_matrix(n, generator_poly, systematic=True)`

Converts the generator polynomial \( g(x) \) into the generator matrix \( G \) for an \([n, k]\) cyclic code.

Parameters

- \( n \) (int) – The codeword size \( n \).
- \( generator\_poly \) (galois.Poly) – The generator polynomial \( g(x) \).
- \( systematic \) (bool, optional) – Optionally specify if the encoding should be systematic, meaning the codeword is the message with parity appended. The default is True.

Returns

The \((k, n)\) generator matrix \( G \), such that given a message \( m \), a codeword is defined by \( c = mG \).

Return type `galois.FieldArray`

Examples

Compute the generator matrix for the Hamming(7, 4) code.

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

In [2]: galois.poly_to_generator_matrix(7, g, systematic=False)
Out[2]: GF([[1, 0, 1, 1, 0, 0, 0],
          [0, 1, 0, 1, 1, 0, 0],
          [0, 0, 1, 0, 1, 1, 0],
          [0, 0, 0, 1, 0, 1, 1]], order=2)

In [3]: galois.poly_to_generator_matrix(7, g, systematic=True)
Out[3]: GF([[1, 0, 0, 1, 0, 0, 1],
          [0, 1, 0, 1, 1, 1, 1],
          [0, 0, 1, 0, 1, 1, 0],
          [0, 0, 0, 1, 0, 1, 1]], order=2)
```
galois.roots_to_parity_check_matrix

Converts the generator polynomial roots into the parity-check matrix $H$ for an $[n, k]$ cyclic code.

**Parameters**

- `n` (*int*) – The codeword size $n$.
- `roots` (*galois.FieldArray*) – The $2t$ roots of the generator polynomial $g(x)$.

**Returns**

The $(2t, n)$ parity-check matrix $H$, such that given a codeword $c$, the syndrome is defined by $s = cH^T$.

**Return type** *galois.FieldArray*

**Examples**

Compute the parity-check matrix for the RS(15, 9) code.

```python
In [1]: GF = galois.GF(2**4)
In [2]: alpha = GF.primitive_element
In [3]: t = 3
In [4]: roots = alpha**np.arange(1, 2*t + 1); roots
Out[4]: GF([ 2, 4, 8, 3, 6, 12], order=2^4)
In [5]: g = galois.Poly.Roots(roots); g
Out[5]: Poly(x^6 + 7x^5 + 9x^4 + 3x^3 + 12x^2 + 10x + 12, GF(2^4))
In [6]: galois.roots_to_parity_check_matrix(15, roots)
Out[6]:
GF([[ 9, 13, 15, 14, 7, 10, 5, 11, 12, 6, 3, 8, 4, 2, 1],
    [13, 14, 10, 11, 6, 8, 2, 9, 15, 7, 5, 12, 3, 4, 1],
    [15, 10, 12, 8, 1, 15, 10, 12, 8, 1, 15, 10, 12, 8, 1],
    [14, 11, 8, 9, 7, 12, 4, 13, 10, 6, 2, 15, 5, 3, 1],
    [ 7, 6, 1, 7, 6, 1, 7, 6, 1, 7, 6, 1, 7, 6, 1],
    [10, 8, 15, 12, 1, 10, 8, 15, 12, 1, 10, 8, 15, 12, 1]],
   order=2^4)
```

### 7.4 Linear Sequences

This section contains classes and functions for creating and analyzing linear sequences.
7.4.1 Linear-feedback shift registers

**LFSR**(poly[, state, config]) A linear-feedback shift register (LFSR).

galois.LFSR

class galois.LFSR(poly, state=1, config='fibonacci')

A linear-feedback shift register (LFSR).

This class implements an LFSR in either the Fibonacci or Galois configuration. An LFSR is defined by its generator polynomial \( g(x) = g_n x^n + \cdots + g_1 x + g_0 \) and initial state vector \( s = [s_{n-1}, \ldots, s_1, s_0] \). Below are diagrams for a degree-3 LFSR in the Fibonacci and Galois configuration. The generator polynomial is \( g(x) = g_3 x^3 + g_2 x^2 + g_1 x + g_0 \) and state vector is \( s = [s_2, s_1, s_0] \).

Listing 1: Fibonacci LFSR Configuration

```
g0  g1  g2  g3
g  |   |   |   |
  s2 --- s1 --- s0 --- y[n]
```

In the Fibonacci configuration, at time instant \( i \) the next \( n - 1 \) outputs are the current state reversed, that is \([y_i, y_{i+1}, \ldots, y_{i+n-1}] = [s_0, s_1, \ldots, s_{n-1}] \). And the \( n \)-th output is a linear combination of the current state and the generator polynomial \( y_{i+n} = (g_n s_0 + g_{n-1} s_1 + \cdots + g_1 s_{n-1}) g_0 \).

Listing 2: Galois LFSR Configuration

```
g0  g1  g2  g3
g  |   |   |   |
  s0 --- s1 --- s2 --- y[n]
```

In the Galois configuration, the next output is \( y = s_{n-1} \) and the next state is computed by \( s_k = s_{n-1} g_n g_k + s_{k-1} \). In the case of \( s_0 \) there is no previous state added.

References

- https://core.ac.uk/download/pdf/288371609.pdf
- https://jhafranco.com/2014/02/15/n-ary-m-sequence-generator-in-python/
Constructors

__init__(poly[, state, config]) Constructs a linear-feedback shift register.

Methods

reset() Resets the LFSR state to the initial state.
step([steps]) Steps the LFSR and produces steps output symbols.

Attributes

config The LFSR configuration, either "fibonacci" or "galois".
field The Galois field that defines the LFSR arithmetic.
initial_state The initial state vector $s = [s_{n-1}, \ldots, s_1, s_0]$.
poly The generator polynomial $g(x) = g_n x^n + \cdots + g_1 x + g_0$.
state The current state vector $s = [s_{n-1}, \ldots, s_1, s_0]$.

__init__(poly, state=1, config='fibonacci')
Constructs a linear-feedback shift register.

Parameters

- **poly** (galois.Poly) – The generator polynomial $g(x) = g_n x^n + \cdots + g_1 x + g_0$.
- **state** (int, tuple, list, numpy.ndarray, galois.FieldArray, optional) – The initial state vector $s = [s_{n-1}, \ldots, s_1, s_0]$. If specified as an integer, then $s_{n-1}$ is interpreted as the MSB and $s_0$ as the LSB. The default is 1 which corresponds to $s = [0, \ldots, 0, 1]$.
- **config** (str, optional) – A string indicating the LFSR feedback configuration, either "fibonacci" (default) or "galois".

Returns A linear-feedback shift register object.

Return type galois.LFSR

reset() Resets the LFSR state to the initial state.

Examples

```
In [1]: lfsr = galois.LFSR(galois.primitive_poly(2, 4)); lfsr
Out[1]: <Fibonacci LFSR: poly=Poly(x^4 + x + 1, GF(2))>

In [2]: lfsr.state
Out[2]: GF([0, 0, 0, 1], order=2)

In [3]: lfsr.step(10)
Out[3]: GF([1, 0, 0, 1, 1, 1, 1, 0, 1, 0], order=2)
```

(continues on next page)
In [4]: lfsr.state
Out[4]: GF([0, 1, 1, 0], order=2)

In [5]: lfsr.reset()

In [6]: lfsr.state
Out[6]: GF([0, 0, 0, 1], order=2)

step(steps=1)
Steps the LFSR and produces steps output symbols.

Parameters steps (int, optional) -- The number of output symbols to produce. The default is 1.

Returns An array of output symbols of type field with size steps.

Return type galois.FieldArray

Examples
Step the LFSR one output at a time.

In [1]: lfsr = galois.LFSR(galois.primitive_poly(2, 4)); lfsr
Out[1]: <Fibonacci LFSR: poly=Polynomial(2, 4)>

In [2]: lfsr.state
Out[2]: GF([0, 0, 0, 1], order=2)

In [3]: lfsr.state, lfsr.step()
Out[3]: (GF([0, 0, 0, 1], order=2), GF(1, order=2))

In [4]: lfsr.state, lfsr.step()
Out[4]: (GF([1, 0, 0, 0], order=2), GF(0, order=2))

In [5]: lfsr.state, lfsr.step()
Out[5]: (GF([1, 1, 0, 0], order=2), GF(0, order=2))

In [6]: lfsr.state, lfsr.step()
Out[6]: (GF([1, 1, 1, 0], order=2), GF(0, order=2))

Step the LFSR 10 steps.

In [7]: lfsr.reset()

In [8]: lfsr.step(10)
Out[8]: GF([1, 0, 0, 0, 1, 0, 1, 1, 1, 0, 1], order=2)

property config
The LFSR configuration, either "fibonacci" or "galois". See the Notes section of LFSR for descriptions of the two configurations.
Examples

```
In [1]: lfsr = galois.LFSR(galois.primitive_poly(2, 4)); lfsr
Out[1]: <Fibonacci LFSR: poly=Poly(x^4 + x + 1, GF(2))>

In [2]: lfsr.config
Out[2]: 'fibonacci'

In [3]: lfsr = galois.LFSR(galois.primitive_poly(2, 4), config="galois"); lfsr
Out[3]: <Galois LFSR: poly=Poly(x^4 + x + 1, GF(2))>

In [4]: lfsr.config
Out[4]: 'galois'
```

**Type**  
str

**property field**  
The Galois field that defines the LFSR arithmetic. The generator polynomial \(g(x)\) is over this field and the state vector contains values in this field.

Examples

```
In [1]: lfsr = galois.LFSR(galois.primitive_poly(2, 4)); lfsr
Out[1]: <Fibonacci LFSR: poly=Poly(x^4 + x + 1, GF(2))>

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

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

**Type**  
galois.FieldClass

**property initial_state**  
The initial state vector \(s = [s_{n-1}, \ldots, s_1, s_0]\).
Examples

```
In [1]: lfsr = galois.LFSR(galois.primitive_poly(2, 4)); lfsr
Out[1]: <Fibonacci LFSR: poly=Poly(x^4 + x + 1, GF(2))>

In [2]: lfsr.initial_state
Out[2]: GF([0, 0, 0, 1], order=2)
```

Type `galois.FieldArray`

**property poly**
The generator polynomial \( g(x) = g_n x^n + \cdots + g_1 x + g_0 \).  

Examples

```
In [1]: lfsr = galois.LFSR(galois.primitive_poly(2, 4)); lfsr
Out[1]: <Fibonacci LFSR: poly=Poly(x^4 + x + 1, GF(2))>

In [2]: lfsr.poly
Out[2]: Poly(x^4 + x + 1, GF(2))
```

Type `galois.Poly`

**property state**
The current state vector \( s = [s_{n-1}, \ldots, s_1, s_0] \).

Examples

```
In [1]: lfsr = galois.LFSR(galois.primitive_poly(2, 4)); lfsr
Out[1]: <Fibonacci LFSR: poly=Poly(x^4 + x + 1, GF(2))>

In [2]: lfsr.state
Out[2]: GF([0, 0, 0, 1], order=2)

In [3]: lfsr.step(10)
Out[3]: GF([1, 0, 0, 0, 1, 1, 1, 1, 0, 1], order=2)

In [4]: lfsr.state
Out[4]: GF([0, 1, 1, 0], order=2)
```

Type `galois.FieldArray`
7.4.2 Sequence analysis functions

\texttt{berlekamp\_massey(sequence[, config, state])} Finds the minimum-degree polynomial \(c(x)\) that produces the sequence in \(\text{GF}(p^m)\).

\texttt{galois.berlekamp\_massey}

\texttt{galois.berlekamp\_massey(sequence, config='fibonacci', state=False)}
Finds the minimum-degree polynomial \(c(x)\) that produces the sequence in \(\text{GF}(p^m)\).

This function implements the Berlekamp-Massey algorithm.

Parameters

- \texttt{sequence (galois.FieldArray)} – A 1-D sequence of Galois field elements in \(\text{GF}(p^m)\).
- \texttt{config (str, optional)} – A string indicating the LFSR feedback configuration for the returned connection polynomial, either "fibonacci" (default) or "galois". See the LFSR configurations in \texttt{galois.LFSR}. The LFSR configuration will indicate if the connection polynomial coefficients should be reversed or not.
- \texttt{state (bool, optional)} – Indicates whether to return the LFSR initial state such that the output is the input sequence. The default is \texttt{False}.

Returns

- \texttt{galois.Poly} – The minimum-degree polynomial \(c(x) \in \text{GF}(p^m)[x]\) that produces the input sequence.
- \texttt{galois.FieldArray} – The initial state of the LFSR such that the output will generate the input sequence. Only returned if \texttt{state=True}.

References


Examples

The Berlekamp-Massey algorithm requires \(2n\) output symbols to determine the \(n\)-degree minimum connection polynomial.

\begin{verbatim}
In [1]: g = galois.conway_poly(2, 8); g
Out[1]: Poly(x^8 + x^4 + x^3 + x^2 + 1, GF(2))

In [2]: lfsr = galois.LFSR(g, state=1); lfsr
Out[2]: <Fibonacci LFSR: poly=Poly(x^8 + x^4 + x^3 + x^2 + 1, GF(2))>

In [3]: s = lfsr.step(16); s
Out[3]: GF([1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 1, 1], order=2)

In [4]: galois.berlekamp_massey(s)
Out[4]: Poly(x^8 + x^4 + x^3 + x^2 + 1, GF(2))
\end{verbatim}
7.5 Number Theory

This section contains functions for performing modular arithmetic and other number theoretic routines.

7.5.1 Divisibility

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcd(a, b)</td>
<td>Finds the greatest common divisor of $a$ and $b$.</td>
</tr>
<tr>
<td>egcd(a, b)</td>
<td>Finds the multiplicands of $a$ and $b$ such that $as + bt = \gcd(a, b)$.</td>
</tr>
<tr>
<td>lcm(*values)</td>
<td>Computes the least common multiple of the arguments.</td>
</tr>
<tr>
<td>prod(*values)</td>
<td>Computes the product of the arguments.</td>
</tr>
<tr>
<td>euler_phi(n)</td>
<td>Counts the positive integers (totatives) in $[1, n]$ that are coprime to $n$.</td>
</tr>
<tr>
<td>totatives(n)</td>
<td>Returns the positive integers (totatives) in $[1, n]$ that are coprime to $n$.</td>
</tr>
<tr>
<td>are_coprime(*values)</td>
<td>Determines if the arguments are pairwise coprime.</td>
</tr>
</tbody>
</table>

**galois.euler_phi**

galois.euler_phi($n$) Counts the positive integers (totatives) in $[1, n]$ that are coprime to $n$.

**Parameters**

- $n$ (int) – A positive integer.

**Returns**

The number of totatives that are coprime to $n$.

**Return type**

int

**Notes**

This function implements the Euler totient function

$$\phi(n) = n \prod_{p \mid n} \left(1 - \frac{1}{p}\right) = \prod_{i=1}^{k} p_i^{e_i - 1} (p_i - 1)$$

for prime $p$ and the prime factorization $n = p_1^{e_1} \cdots p_k^{e_k}$.

**References**

- Section 2.4.1 from [https://cacr.uwaterloo.ca/hac/about/chap2.pdf](https://cacr.uwaterloo.ca/hac/about/chap2.pdf)
- [https://oeis.org/A000010](https://oeis.org/A000010)
Examples

In [1]: n = 20

In [2]: phi = galois.euler_phi(n); phi
Out[2]: 8

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

# The number of totatives is phi
In [4]: len(totatives) == phi
Out[4]: True

# For prime n, \((n) = n - 1\)
In [5]: galois.euler_phi(13)
Out[5]: 12

**galois.totatives**

galois.totatives\((n)\)

Returns the positive integers (totatives) in \([1, n]\) that are coprime to \(n\).

The totatives of \(n\) form the multiplicative group \((\mathbb{Z}/n\mathbb{Z})^∞\).

**Parameters**

- \(n\) \((\textit{int})\) – A positive integer.

**Returns**

The totatives of \(n\).

**Return type**

list

**References**

- Section 2.4.3 from https://cacr.uwaterloo.ca/hac/about/chap2.pdf
- https://oeis.org/A000010

Examples

In [1]: n = 20

In [2]: totatives = galois.totatives(n); totatives
Out[2]: [1, 3, 7, 9, 11, 13, 17, 19]

In [3]: phi = galois.euler_phi(n); phi
Out[3]: 8

In [4]: len(totatives) == phi
Out[4]: True
### 7.5.2 Congruences

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pow(base, exponent, modulus)</code></td>
<td>Efficiently performs modular exponentiation.</td>
</tr>
<tr>
<td><code>crt(reminders, moduli)</code></td>
<td>Solves the simultaneous system of congruences for ( x ).</td>
</tr>
<tr>
<td><code>primitive_root(n[, start, stop, reverse])</code></td>
<td>Finds the smallest primitive root modulo ( n ).</td>
</tr>
<tr>
<td><code>primitive_roots(n[, start, stop, reverse])</code></td>
<td>Finds all primitive roots modulo ( n ).</td>
</tr>
<tr>
<td><code>carmichael_lambda(n)</code></td>
<td>Finds the smallest positive integer ( m ) such that ( a^m \equiv 1 \pmod{m} ) for every integer ( a ) in ([1, n]) that is coprime to ( n ).</td>
</tr>
<tr>
<td><code>legendre_symbol(a, p)</code></td>
<td>Computes the Legendre symbol ( \left( \frac{a}{p} \right) ).</td>
</tr>
<tr>
<td><code>jacobi_symbol(a, n)</code></td>
<td>Computes the Jacobi symbol ( \left( \frac{a}{n} \right) ).</td>
</tr>
<tr>
<td><code>kronecker_symbol(a, n)</code></td>
<td>Computes the Kronecker symbol ( \left( \frac{a}{n} \right) ).</td>
</tr>
<tr>
<td><code>is_primitive_root(g, n)</code></td>
<td>Determines if ( g ) is a primitive root modulo ( n ).</td>
</tr>
<tr>
<td><code>is_cyclic(n)</code></td>
<td>Determines whether the multiplicative group ((\mathbb{Z}/n\mathbb{Z})^*) is cyclic.</td>
</tr>
</tbody>
</table>

#### galois.carmichael_lambda

**galois.carmichael_lambda\( (n) \)**

Finds the smallest positive integer \( m \) such that \( a^m \equiv 1 \pmod{m} \) for every integer \( a \) in \([1, n]\) that is coprime to \( n \).

This function implements the Carmichael function \( \lambda(n) \).

**Parameters**

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

**Returns**

The smallest positive integer \( m \) such that \( a^m \equiv 1 \pmod{m} \) for every integer \( a \) in \([1, n]\) that is coprime to \( n \).

**Return type**

\( int \)

#### References

- [https://oeis.org/A002322](https://oeis.org/A002322)

#### Examples

The Carmichael lambda function and Euler totient function are often equal. However, there are notable exceptions.

```
In [1]: [galois.euler_phi(n) for n in range(1, 20)]
Out[1]: [1, 1, 2, 2, 4, 2, 6, 4, 6, 4, 10, 4, 12, 6, 8, 8, 16, 6, 18]

In [2]: [galois.carmichael_lambda(n) for n in range(1, 20)]
Out[2]: [1, 1, 2, 2, 4, 2, 6, 2, 6, 4, 10, 2, 12, 6, 4, 4, 16, 6, 18]
```

For prime \( n \), \( \phi(n) = \lambda(n) = n - 1 \). And for most composite \( n \), \( \phi(n) = \lambda(n) \) \( < n - 1 \).

```
In [3]: n = 9

In [4]: phi = galois.euler_phi(n); phi
Out[4]: 6
```
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, 4, 7, 1] (mod 9)
   Power 3: [1, 8, 1, 8, 1, 8] (mod 9)
   Power 4: [1, 7, 4, 4, 7, 1] (mod 9)
   Power 5: [1, 5, 7, 2, 4, 8] (mod 9)
   Power 6: [1, 1, 1, 1, 1, 1] (mod 9)

In [8]: galois.is_cyclic(n)
Out[8]: True

When $\phi(n) \neq \lambda(n)$, the multiplicative group $(\mathbb{Z}/n\mathbb{Z})^\times$ is not cyclic. See `galois.is_cyclic()`.

In [9]: n = 8

In [10]: phi = galois.euler_phi(n); phi
Out[10]: 4

In [11]: lambda_ = galois.carmichael_lambda(n); lambda_
Out[11]: 2

In [12]: totatives = galois.totatives(n); totatives
Out[12]: [1, 3, 5, 7]

In [13]: for power in range(1, phi + 1):
   ....:     y = [pow(a, power, n) for a in totatives]
   ....:     print("Power {}: {} (mod {})").format(power, y, n))
   ....:
   Power 1: [1, 3, 5, 7] (mod 8)
   Power 2: [1, 1, 1, 1] (mod 8)
   Power 3: [1, 3, 5, 7] (mod 8)
   Power 4: [1, 1, 1, 1] (mod 8)

In [14]: galois.is_cyclic(n)
Out[14]: False
**galois.legendre_symbol**

`galois.legendre_symbol(a, p)`  
Computes the Legendre symbol \( \left( \frac{a}{p} \right) \).

**Parameters**

- `a (int)` – An integer.
- `p (int)` – An odd prime \( p \geq 3 \).

**Returns** The Legendre symbol \( \left( \frac{a}{p} \right) \) with value in \{0, 1, -1\}.

**Return type** int

**Notes**

The Legendre symbol is useful for determining if \( a \) is a quadratic residue modulo \( p \), namely \( a \in \mathbb{Q}_p \). A quadratic residue \( a \) modulo \( p \) satisfies \( x^2 \equiv a \pmod{p} \) for some \( x \).

\[
\left( \frac{a}{p} \right) = \begin{cases} 
0, & p \mid a \\
1, & a \in \mathbb{Q}_p \\
-1, & a \in \overline{\mathbb{Q}_p} 
\end{cases}
\]

**References**

- Algorithm 2.149 from https://cacr.uwaterloo.ca/hac/about/chap2.pdf

**Examples**

The quadratic residues modulo 7 are \( \mathbb{Q}_7 = \{1, 2, 4\} \). The quadratic non-residues modulo 7 are \( \overline{\mathbb{Q}_7} = \{3, 5, 6\} \).

```
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

**galois.jacobi_symbol**

galois.jacobi_symbol(a, n)

Computes the Jacobi symbol \( \left( \frac{a}{n} \right) \).

**Parameters**

- **a (int)** – An integer.
- **n (int)** – An odd integer \( n \geq 3 \).

**Returns** The Jacobi symbol \( \left( \frac{a}{n} \right) \) with value in \{0, 1, -1\}.

**Return type** int

**Notes**

The Jacobi symbol extends the Legendre symbol for odd \( n \geq 3 \). Unlike the Legendre symbol, \( \left( \frac{a}{n} \right) = 1 \) does not imply \( a \) is a quadratic residue modulo \( n \). However, all \( a \in \mathbb{Q}_n \) have \( \left( \frac{a}{n} \right) = 1 \).

**References**

- Algorithm 2.149 from https://cacr.uwaterloo.ca/hac/about/chap2.pdf

**Examples**

The quadratic residues modulo 9 are \( \mathbb{Q}_9 = \{1, 4, 7\} \) and these all satisfy \( \left( \frac{a}{9} \right) = 1 \). The quadratic non-residues modulo 9 are \( \mathbb{Q}_9^c = \{2, 3, 5, 6, 8\} \), but notice \( \{2, 5, 8\} \) also satisfy \( \left( \frac{a}{9} \right) = 1 \). The set of integers \( \{3, 6\} \) not coprime to 9 satisfies \( \left( \frac{a}{9} \right) = 0 \).

```
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"\(\frac{a}{9}\) = {galois.jacobi_symbol(a, 9)}")
   ....:     (0 / 9) = 0
   ....:     (1 / 9) = 1
   ....:     (2 / 9) = 1
   ....:     (3 / 9) = 0
   ....:     (4 / 9) = 1
   ....:     (5 / 9) = 1
   ....:     (6 / 9) = 0
   ....:     (7 / 9) = 1
   ....:     (8 / 9) = 1
```
galois.kronecker_symbol

galois.kronecker_symbol(a, n)
Computes the Kronecker symbol \( \left( \frac{a}{n} \right) \).

The Kronecker symbol extends the Jacobi symbol for all \( n \).

Parameters

- **a** (int) – An integer.
- **n** (int) – An integer.

Returns The Kronecker symbol \( \left( \frac{a}{n} \right) \) with value in \{0, -1, 1\}.

Return type int

References

- Algorithm 2.149 from https://cacr.uwaterloo.ca/hac/about/chap2.pdf

galois.is_cyclic

galois.is_cyclic(n)
Determines whether the multiplicative group \( (\mathbb{Z}/n\mathbb{Z})^\times \) is cyclic.

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

Returns True if the multiplicative group \( (\mathbb{Z}/n\mathbb{Z})^\times \) is cyclic.

Return type bool

Notes

The multiplicative group \( (\mathbb{Z}/n\mathbb{Z})^\times \) is the set of positive integers \( 1 \leq a < n \) that are coprime with \( n \). \( (\mathbb{Z}/n\mathbb{Z})^\times \) being cyclic means that some primitive root of \( n \), or generator, \( g \) can generate the group \( \{g^0, g^1, g^2, \ldots, g^{\phi(n)-1}\} \), where \( \phi(n) \) is Euler’s totient function and calculates the order of the group. If \( (\mathbb{Z}/n\mathbb{Z})^\times \) is cyclic, the number of primitive roots is found by \( \phi(\phi(n)) \).

\( (\mathbb{Z}/n\mathbb{Z})^\times \) is cyclic if and only if \( n \) is 2, 4, \( p^k \), or \( 2p^k \), where \( p \) is an odd prime and \( k \) is a positive integer.

Examples

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

```python
# n is of type 2*p^e, which is cyclic
In [1]: n = 14

In [2]: galois.is_cyclic(n)
Out[2]: True

In [3]: Znx = set(galois.totatives(n)); Znx
Out[3]: {1, 3, 5, 9, 11, 13}
```

In [4]: phi = galois.euler_phi(n); phi
Out[4]: 6

In [5]: len(Znx) == phi
Out[5]: True

# The primitive roots are the elements in Znx that multiplicatively generate the group
In [6]: for a in Znx:
   ...:     span = set([pow(a, i, n) for i in range(1, phi + 1)])
   ...:     primitive_root = galois.is_primitive_root(a, n)
   ...:     print("Element: {:2d}, Span: {:<20}, Primitive root: {}".format(a, 
   ...:     str(span), primitive_root))
   ...
Element:  1, Span: {1} , Primitive root: False
Element:  3, Span: {1, 3, 5, 9, 11, 13}, Primitive root: True
Element:  5, Span: {1, 3, 5, 9, 11, 13}, Primitive root: True
Element:  9, Span: {9, 11, 1} , Primitive root: False
Element: 11, Span: {9, 11, 1} , Primitive root: False
Element: 13, Span: {1, 13} , Primitive root: False

# Find the smallest primitive root
In [7]: galois.primitive_root(n)
Out[7]: 3

# Find all primitive roots
In [8]: roots = galois.primitive_roots(n); roots
Out[8]: [3, 5]

# Euler's totient function ( \( \phi(n) \) ) counts the primitive roots of n
In [9]: len(roots) == galois.euler_phi(phi)
Out[9]: True

A counterexample is \( n = 15 = 3 \cdot 5 \), which doesn't fit the condition for cyclicness. \( (\mathbb{Z}/15\mathbb{Z})^\times = \{1, 2, 4, 7, 8, 11, 13, 14\} \). Since the group is not cyclic, it has no primitive roots.

# n is of type \( p1^{e1} \cdot p2^{e2} \), which is not cyclic
In [10]: n = 15

In [11]: galois.is_cyclic(n)
Out[11]: False

In [12]: Znx = set(galois.totatives(n)); Znx
Out[12]: {1, 2, 4, 7, 8, 11, 13, 14}

In [13]: phi = galois.euler_phi(n); phi
Out[13]: 8

In [14]: len(Znx) == phi
Out[14]: True

# The primitive roots are the elements in Znx that multiplicatively generate the group
(continues on next page)
In [15]: for a in Znx:
    .....:  span = set([pow(a, i, n) for i in range(1, phi + 1)])
    .....:  primitive_root = galois.is_primitive_root(a, n)
    .....:  print("Element: {:2d}, Span: {:<13}, Primitive root: {}".format(a, str(span), primitive_root))
    .....:
Element:  1, Span: {1} , Primitive root: False
Element:  2, Span: {8, 1, 2, 4} , Primitive root: False
Element:  4, Span: {1, 4} , Primitive root: False
Element:  7, Span: {1, 4, 13, 7}, Primitive root: False
Element:  8, Span: {8, 1, 2, 4} , Primitive root: False
Element: 11, Span: {1, 11} , Primitive root: False
Element: 13, Span: {1, 4, 13, 7}, Primitive root: False
Element: 14, Span: {1, 14} , Primitive root: False

# Find the smallest primitive root
In [16]: galois.primitive_root(n)

# Find all primitive roots
In [17]: roots = galois.primitive_roots(n); roots
Out[17]: []

# Note the max order of any element is 4, not 8, which is Carmichael's lambda function
In [18]: galois.carmichael_lambda(n)
Out[18]: 4

For prime \( n \), a primitive root modulo \( n \) is also a primitive element of the Galois field \( \text{GF}(n) \). A primitive element is a generator of the multiplicative group \( \text{GF}(p)^\times = \{1, 2, \ldots, p - 1\} = \{g^0, g^1, g^2, \ldots, g^{\phi(n)-1}\} \).

# n is of type p, which is cyclic
In [19]: n = 7

In [20]: galois.is_cyclic(n)
Out[20]: True

In [21]: Znx = set(galois.totatives(n)); Znx
Out[21]: {1, 2, 3, 4, 5, 6}

In [22]: phi = galois.euler_phi(n); phi
Out[22]: 6

In [23]: len(Znx) == phi
Out[23]: True

# The primitive roots are the elements in Znx that multiplicatively generate the group
In [24]: for a in Znx:
    .....:  span = set([pow(a, i, n) for i in range(1, phi + 1)])
    .....:  primitive_root = galois.is_primitive_root(a, n)
    .....:  print("Element: {:2d}, Span: {:<18}, Primitive root: {}".format(a, str(span), primitive_root))
### 7.5.3 Integer arithmetic

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>isqrt(n)</code></td>
<td>Computes ( x = \lfloor \sqrt{n} \rfloor ) such that ( x^2 \leq n &lt; (x + 1)^2 ).</td>
</tr>
<tr>
<td><code>iroot(n, k)</code></td>
<td>Computes ( x = \lfloor n^{\frac{1}{k}} \rfloor ) such that ( x^k \leq n &lt; (x + 1)^k ).</td>
</tr>
<tr>
<td><code>ilog(n, b)</code></td>
<td>Computes ( x = \lfloor \log_b(n) \rfloor ) such that ( b^x \leq n &lt; b^{x+1} ).</td>
</tr>
</tbody>
</table>

**galois.isqrt**

`galois.isqrt(n)`

Computes \( x = \lfloor \sqrt{n} \rfloor \) such that \( x^2 \leq n < (x + 1)^2 \).

**Note:** This function is included for Python versions before 3.8. For Python 3.8 and later, this function calls `math.isqrt()` from the standard library.

**Parameters**

- \( n \) (int) – A non-negative integer.

**Returns**

The integer square root of \( n \).

**Return type**

int
Examples

In [1]: n = 1000
In [2]: x = galois.isqrt(n); x
Out[2]: 31
In [3]: print(f"{x}**2 \leq n \lt {(x + 1)}**2")
961 \leq 1000 \lt 1024

galois.iroot

**galois.iroot***(n, k)**

Computes *x* = \( \lfloor n^{\frac{1}{k}} \rfloor \) such that \( x^k \leq n < (x + 1)^k \).

**Parameters**

- **n** *(int)* – A non-negative integer.
- **k** *(int)* – The root *k*, must be at least 2.

**Returns** The integer *k*-th root of *n*.

**Return type** int

Examples

In [1]: n = 1000
In [2]: x = galois.iroot(n, 5); x
Out[2]: 3
In [3]: print(f"{x}**5 \leq n \lt {(x + 1)}**5")
243 \leq 1000 \lt 1024

galois.ilog

**galois.ilog**(n, b)

Computes *x* = \( \lfloor \log_b(n) \rfloor \) such that \( b^x \leq n < b^{x+1} \).

**Parameters**

- **n** *(int)* – A positive integer.
- **b** *(int)* – The logarithm base *b*.

**Returns** The integer logarithm base *b* of *n*.

**Return type** int

7.5. Number Theory
Examples

```python
In [1]: n = 1000
In [2]: x = galois.ilog(n, 5); x
Out[2]: 4
In [3]: print(f"{5**x} <= {n} < {5**(x + 1)}")
625 <= 1000 < 3125
```

### 7.6 Integer Factorization

This section contains functions for factoring integers and analyzing their properties.

#### 7.6.1 Prime factorization

<table>
<thead>
<tr>
<th><code>factors(value)</code></th>
<th>Computes the prime factors of a positive integer or the irreducible factors of a non-constant, monic polynomial.</th>
</tr>
</thead>
</table>

#### 7.6.2 Composite factorization

<table>
<thead>
<tr>
<th><code>divisors(n)</code></th>
<th>Computes all positive integer divisors $d$ of the integer $n$ such that $d \mid n$.</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>divisor_sigma(n[, k])</code></td>
<td>Returns the sum of $k$-th powers of the positive divisors of $n$.</td>
</tr>
</tbody>
</table>

**galois.divisors**

`galois.divisors(n)` Computes all positive integer divisors $d$ of the integer $n$ such that $d \mid n$.

- **Parameters** 
  - `n (int)` – Any integer.
- **Returns** 
  - Sorted list of positive integer divisors $d$.
- **Return type** 
  - list

**Notes**

`galois.divisors()` find all positive integer divisors or factors of $n$, where `galois.factors()` only finds the prime factors of $n$. 
Examples

```
In [1]: galois.divisors(0)
Out[1]: []

In [2]: galois.divisors(1)
Out[2]: [1]

In [3]: galois.divisors(24)
Out[3]: [1, 2, 3, 4, 6, 8, 12, 24]

In [4]: galois.divisors(-24)
Out[4]: [1, 2, 3, 4, 6, 8, 12, 24]

In [5]: galois.factors(24)
Out[5]: ([2, 3], [3, 1])
```

galois.divisor_sigma

galois.divisor_sigma(n, k=1)

Returns the sum of $k$-th powers of the positive divisors of $n$.

Parameters

- `n (int)` – Any integer.
- `k (int, optional)` – The degree of the positive divisors. The default is 1 which corresponds to $\sigma_1(n)$ which is the sum of positive divisors.

Returns

The sum of divisors function $\sigma_k(n)$.

Return type  int

Notes

This function implements the $\sigma_k(n)$ function. It is defined as:

$$\sigma_k(n) = \sum_{d \mid n} d^k$$

Examples

```
In [1]: galois.divisors(9)
Out[1]: [1, 3, 9]

In [2]: galois.divisor_sigma(9, k=0)
Out[2]: 3

In [3]: galois.divisor_sigma(9, k=1)
Out[3]: 13

In [4]: galois.divisor_sigma(9, k=2)
Out[4]: 91
```
7.6.3 Specific factorization algorithms

### perfect_power(n)
Returns the integer base $c > 1$ and exponent $e > 1$ of $n = c^e$ if $n$ is a perfect power.

### trial_division(n, B)
Finds all the prime factors $p_i^{e_i}$ of $n$ for $p_i \leq B$.

### pollard_p1(n, B[, B2])
Attempts to find a non-trivial factor of $n$ if it has a prime factor $p$ such that $p - 1$ is $B$-smooth.

### pollard_rho(n[, c])
Attempts to find a non-trivial factor of $n$ using cycle detection.

---

**galois.perfect_power**

**galois.perfect_power(n)**
Returns the integer base $c > 1$ and exponent $e > 1$ of $n = c^e$ if $n$ is a perfect power.

**Parameters**
- **n** *(int)* – A positive integer $n > 1$.

**Returns**
- None is $n$ is not a perfect power. Otherwise, $(c, e)$ such that $n = c^e$. $c$ may be composite.

**Return type**
- None, tuple

**Examples**

```python
# Primes are not perfect powers
In [1]: galois.perfect_power(5)

# Products of primes are not perfect powers
In [2]: galois.perfect_power(2*3)

# Products of prime powers were the GCD of the exponents is 1 are not perfect powers
In [3]: galois.perfect_power(2 * 3 * 5**3)

# Products of prime powers were the GCD of the exponents is > 1 are perfect powers
In [4]: galois.perfect_power(2**2 * 3**2 * 5**4)
Out[4]: (150, 2)

In [5]: galois.perfect_power(36)
Out[5]: (6, 2)

In [6]: galois.perfect_power(125)
Out[6]: (5, 3)
```
galois.trial_division

**galois.trial_division**(*n, B=None*)

Finds all the prime factors \( p^e_i \) of \( n \) for \( p_i \leq B \).

The trial division factorization will find all prime factors \( p^e_i \leq B \) such that \( n \) factors as \( n = p^e_1 \cdots p^e_k n_r \) where \( n_r \) is a residual factor (which may be composite).

**Parameters**

- **n** (*int*) – A positive integer.
- **B** (*int, optional*) – The max divisor in the trial division. The default is `None` which corresponds to \( B = \sqrt{n} \). If \( B > \sqrt{n} \), the algorithm will only search up to \( \sqrt{n} \), since a prime factor of \( n \) cannot be larger than \( \sqrt{n} \).

**Returns**

- **list** – The discovered prime factors \( \{p_1, \ldots, p_k\} \).
- **list** – The corresponding prime exponents \( \{e_1, \ldots, e_k\} \).
- **int** – The residual factor \( n_r \).

**Examples**

```
In [1]: n = 2**4 * 17**3 * 113 * 15013
In [2]: galois.trial_division(n)
Out[2]: ((2, 17, 113, 15013), [4, 3, 1, 1], 1)
In [3]: galois.trial_division(n, B=500)
Out[3]: ((2, 17, 113), [4, 3, 1], 15013)
In [4]: galois.trial_division(n, B=100)
Out[4]: ((2, 17), [4, 3], 1696469)
```

**galois.pollard_p1**

**galois.pollard_p1**(*n, B, B2=None*)

Attempts to find a non-trivial factor of \( n \) if it has a prime factor \( p \) such that \( p - 1 \) is \( B \)-smooth.

For a given odd composite \( n \) with a prime factor \( p \), Pollard’s \( p - 1 \) algorithm can discover a non-trivial factor of \( n \) if \( p - 1 \) is \( B \)-smooth. Specifically, the prime factorization must satisfy \( p - 1 = p^e_1 \cdots p^e_k \) with each \( p_i \leq B \).

A extension of Pollard’s \( p - 1 \) algorithm allows a prime factor \( p \) to be \( B \)-smooth with the exception of one prime factor \( B < p_{k+1} \leq B_2 \). In this case, the prime factorization is \( p - 1 = p^e_1 \cdots p^e_k p_{k+1} \). Often \( B_2 \) is chosen such that \( B_2 \gg B \).

**Parameters**

- **n** (*int*) – An odd composite integer \( n > 2 \) that is not a prime power.
- **B** (*int*) – The smoothness bound \( B > 2 \).
- **B2** (*int, optional*) – The smoothness bound \( B_2 \) for the optional second step of the algorithm. The default is `None`, which will not perform the second step.

**Returns** A non-trivial factor of \( n \), if found. `None` if not found.
Return type  None, int

References
- Section 3.2.3 from https://cacr.uwaterloo.ca/hac/about/chap3.pdf

Examples
Here, \( n = pq \) where \( p - 1 \) is 1039-smooth and \( q - 1 \) is 17-smooth.

\[
\begin{align*}
\text{In [1]: } & \quad p, q = 1458757, 1326001 \\
\text{In [2]: } & \quad \text{galois.factors}(p - 1) \\
& \quad \text{Out[2]: } ([2, 3, 13, 1039], [2, 3, 1, 1]) \\
\text{In [3]: } & \quad \text{galois.factors}(q - 1) \\
& \quad \text{Out[3]: } ([2, 3, 5, 13, 17], [4, 1, 3, 1, 1])
\end{align*}
\]

Searching with \( B = 15 \) will not recover a prime factor.

\[
\begin{align*}
\text{In [4]: } & \quad \text{galois.pollard_p1}(p q, 15) \\
\text{Out[4]: } & \quad \text{1326001}
\end{align*}
\]

Searching with \( B = 17 \) will recover the prime factor \( q \).

\[
\begin{align*}
\text{In [5]: } & \quad \text{galois.pollard_p1}(p q, 17) \\
& \quad \text{Out[5]: } \text{1326001}
\end{align*}
\]

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 \( 15 < 17 \leq 100 \).

\[
\begin{align*}
\text{In [6]: } & \quad \text{galois.pollard_p1}(p q, 15, B2=100) \\
& \quad \text{Out[6]: } \text{1326001}
\end{align*}
\]

Pollard’s \( p - 1 \) algorithm may return a composite factor.

\[
\begin{align*}
\text{In [7]: } & \quad n = 2133861346249 \\
\text{In [8]: } & \quad \text{galois.factors}(n) \\
& \quad \text{Out[8]: } ([37, 41, 5471, 257107], [1, 1, 1, 1]) \\
\text{In [9]: } & \quad \text{galois.pollard_p1}(n, 10) \\
& \quad \text{Out[9]: } 1517 \\
\text{In [10]: } & \quad 37*41 \\
& \quad \text{Out[10]: } 1517
\end{align*}
\]
galois.pollard_rho

\texttt{galois.pollard_rho}(n, c=1)

Attempts to find a non-trivial factor of \( n \) using cycle detection.

Pollard’s \( \rho \) algorithm seeks to find a non-trivial factor of \( n \) by finding a cycle in a sequence of integers \( x_0, x_1, \ldots \) defined by \( x_i = f(x_{i-1}) = x_i^2 + 1 \pmod{p} \) where \( p \) is an unknown small prime factor of \( n \). This happens when \( x_m \equiv x_{2m} \pmod{p} \). Because \( p \) is unknown, this is accomplished by computing the sequence modulo \( n \) and looking for \( \gcd(x_m - x_{2m}, n) > 1 \).

**Parameters**

- \texttt{n (int)} – An odd composite integer \( n > 2 \) that is not a prime power.
- \texttt{c (int, optional)} – The constant offset in the function \( f(x) = x^2 + c \pmod{n} \). The default is 1. A requirement of the algorithm is that \( c \not\in \{0, -2\} \).

**Returns** A non-trivial factor \( m \) of \( n \), if found. \texttt{None} if not found.

**Return type** \texttt{None, int}

**References**

- Section 3.2.2 from https://cacr.uwaterloo.ca/hac/about/chap3.pdf

**Examples**

Pollard’s \( \rho \) is especially good at finding small factors.

```python
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.

```python
In [3]: n = 1182640843 * 1716279751

In [4]: galois.pollard_rho(n)
Out[4]: 1716279751
```

### 7.6.4 Integer tests

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{is_prime(n)}</td>
<td>Determines if ( n ) is prime.</td>
</tr>
<tr>
<td>\texttt{is_prime_power(n)}</td>
<td>Determines if ( n ) is a prime power ( n = p^k ) for prime ( p ) and ( k \geq 1 ).</td>
</tr>
<tr>
<td>\texttt{is_composite(n)}</td>
<td>Determines if ( n ) is composite.</td>
</tr>
<tr>
<td>\texttt{is_perfect_power(n)}</td>
<td>Determines if ( n ) is a perfect power ( n = x^k ) for ( x &gt; 0 ) and ( k \geq 2 ).</td>
</tr>
<tr>
<td>\texttt{is_square_free(value)}</td>
<td>Determines if the positive integer or the non-constant, monic polynomial is square-free.</td>
</tr>
<tr>
<td>\texttt{is_smooth(n, B)}</td>
<td>Determines if the positive integer ( n ) is ( B )-smooth.</td>
</tr>
<tr>
<td>\texttt{is_powersmooth(n, B)}</td>
<td>Determines if the positive integer ( n ) is ( B )-powersmooth.</td>
</tr>
</tbody>
</table>
**galois.is_prime**

**galois.is_prime(n)**
Determines if \( n \) is prime.

- **Parameters** \( n \) (int) – A positive integer.
- **Returns** True if the integer \( n \) is prime.
- **Return type** bool

**Notes**

This algorithm will first run Fermat’s primality test to check \( n \) for compositeness, see `galois.fermat_primality_test()`. If it determines \( n \) is composite, the function will quickly return. If Fermat’s primality test returns True, then \( n \) could be prime or pseudoprime. If so, then the algorithm will run 10 rounds of Miller-Rabin’s primality test, see `galois.miller_rabin_primality_test()`. With this many rounds, a result of True should have high probability of \( n \) being a true prime, not a pseudoprime.

**Examples**

```
In [1]: galois.is_prime(13)
Out[1]: True

In [2]: galois.is_prime(15)
Out[2]: False
```

The algorithm is also efficient on very large \( n \).

```
In [3]: galois.is_prime(1000000000000000035000061)
Out[3]: True
```

**galois.is_prime_power**

**galois.is_prime_power(n)**
Determines if \( n \) is a prime power \( n = p^k \) for prime \( p \) and \( k \geq 1 \).

- **Parameters** \( n \) (int) – A positive integer.
- **Returns** True if the integer \( n \) is a prime power.
- **Return type** bool

**Notes**

There is some controversy over whether 1 is a prime power \( p^0 \). Since 1 is the 0-th power of all primes, it is often regarded not as a prime power. This function returns False for 1.
Examples

```
In [1]: galois.is_prime_power(8)
Out[1]: True

In [2]: galois.is_prime_power(6)
Out[2]: False

In [3]: galois.is_prime_power(1)
Out[3]: False
```

galois.is_composite

galois.is_composite(n)
Determines if \( n \) is composite.

Parameters
- \( n \) (int) – A positive integer.

Returns
- True if the integer \( n \) is composite.

Return type
- bool

Examples

```
In [1]: galois.is_composite(13)
Out[1]: False

In [2]: galois.is_composite(15)
Out[2]: True
```

galois.is_perfect_power

galois.is_perfect_power(n)
Determines if \( n \) is a perfect power \( n = x^k \) for \( x > 0 \) and \( k \geq 2 \).

Parameters
- \( n \) (int) – A positive integer.

Returns
- True if the integer \( n \) is a perfect power.

Return type
- bool

Examples

```
In [1]: galois.is_perfect_power(8)
Out[1]: True

In [2]: galois.is_perfect_power(16)
Out[2]: True

In [3]: galois.is_perfect_power(20)
Out[3]: False
```
\textbf{galois.is_smooth}

\texttt{galois.is_smooth}(n, B)

Determines if the positive integer \( n \) is \( B \)-smooth.

\begin{itemize}
  \item \texttt{n (int)} – A positive integer.
  \item \texttt{B (int)} – The smoothness bound \( B \geq 2 \).
\end{itemize}

\textbf{Returns} True if \( n \) is \( B \)-smooth.

\textbf{Return type} \texttt{bool}

\textbf{Notes}

An integer \( n \) with prime factorization \( n = p_1^{e_1} \ldots p_k^{e_k} \) is \( B \)-smooth if \( p_k \leq B \). The 2-smooth numbers are the powers of 2. The 5-smooth numbers are known as regular numbers. The 7-smooth numbers are known as humble numbers or highly composite numbers.

\textbf{Examples}

\begin{verbatim}
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
\end{verbatim}

\textbf{galois.is_powersmooth}

\texttt{galois.is_powersmooth}(n, B)

Determines if the positive integer \( n \) is \( B \)-powersmooth.

\begin{itemize}
  \item \texttt{n (int)} – A positive integer.
  \item \texttt{B (int)} – The smoothness bound \( B \geq 2 \).
\end{itemize}

\textbf{Returns} True if \( n \) is \( B \)-powersmooth.

\textbf{Return type} \texttt{bool}
Notes

An integer \( n \) with prime factorization \( n = p_1^{e_1} \cdots p_k^{e_k} \) is \( B \)-powersmooth if \( p_i^{e_i} \leq B \) for \( 1 \leq i \leq k \).

Examples

Comparison of \( B \)-smooth and \( B \)-powersmooth. Necessarily, any \( n \) that is \( B \)-powersmooth must be \( B \)-smooth.

```python
In [1]: galois.is_smooth(2**4 * 3**2 * 5, 5)
Out[1]: True

In [2]: galois.is_powersmooth(2**4 * 3**2 * 5, 5)
Out[2]: False
```

7.7 Primes

This section contains functions for generating primes and analyzing primality.

7.7.1 Prime number generation

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>primes(n)</code></td>
<td>Returns all primes ( p ) for ( p \leq n ).</td>
</tr>
<tr>
<td><code>kth_prime(k)</code></td>
<td>Returns the ( k )-th prime.</td>
</tr>
<tr>
<td><code>prev_prime(n)</code></td>
<td>Returns the nearest prime ( p ), such that ( p \leq n ).</td>
</tr>
<tr>
<td><code>next_prime(n)</code></td>
<td>Returns the nearest prime ( p ), such that ( p &gt; n ).</td>
</tr>
<tr>
<td><code>random_prime(bits)</code></td>
<td>Returns a random prime ( p ) with ( b ) bits, such that ( 2^b \leq p &lt; 2^{b+1} ).</td>
</tr>
<tr>
<td><code>mersenne_exponents([n])</code></td>
<td>Returns all known Mersenne exponents ( e ) for ( e \leq n ).</td>
</tr>
<tr>
<td><code>mersenne_primes([n])</code></td>
<td>Returns all known Mersenne primes ( p ) for ( p \leq 2^n - 1 ).</td>
</tr>
</tbody>
</table>

```
galois.primes(n)

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.

Return type
- list
```
Notes
This function implements the Sieve of Eratosthenes to efficiently find the primes.

References
- https://oeis.org/A000040

Examples
```
In [1]: galois.primes(19)
Out[1]: [2, 3, 5, 7, 11, 13, 17, 19]

In [2]: galois.primes(20)
Out[2]: [2, 3, 5, 7, 11, 13, 17, 19]
```

galois.kth_prime

galois.kth_prime(k)
Returns the \(k\)-th prime.

Parameters
- \(k\) (int) – The prime index (1-indexed), where \(k = \{1, 2, 3, 4, \ldots\}\) for primes \(p = \{2, 3, 5, 7, \ldots\}\).

Returns
- The \(k\)-th prime.

Return type
- int

Examples
```
In [1]: galois.kth_prime(1)
Out[1]: 2

In [2]: galois.kth_prime(3)
Out[2]: 5

In [3]: galois.kth_prime(1000)
Out[3]: 7919
```

galois.prev_prime

galois.prev_prime(n)
Returns the nearest prime \(p\), such that \(p \leq n\).

Parameters
- \(n\) (int) – An integer.

Returns
- The nearest prime \(p \leq n\). If \(n < 2\), the function returns None.

Return type
- int
Examples

```python
In [1]: galois.prev_prime(13)
Out[1]: 13

In [2]: galois.prev_prime(15)
Out[2]: 13
```

**galois.next_prime**

galois.next_prime\((n)\)

Returns the nearest prime \(p\), such that \(p > n\).

- **Parameters** \(n\) (int) – An integer.
- **Returns** The nearest prime \(p > n\).
- **Return type** int

Examples

```python
In [1]: galois.next_prime(13)
Out[1]: 17

In [2]: galois.next_prime(15)
Out[2]: 17
```

**galois.random_prime**

galois.random_prime\((b)\)

Returns a random prime \(p\) with \(b\) bits, such that \(2^b \leq p < 2^{b+1}\).

This function randomly generates integers with \(b\) bits and uses the primality tests in `galois.is_prime()` to determine if \(p\) is prime.

- **Parameters** \(b\) (int) – The number of bits in the prime \(p\).
- **Returns** A random prime in \(2^b \leq p < 2^{b+1}\).
- **Return type** int

References

- https://en.wikipedia.org/wiki/Prime_number_theorem
Examples

Generate a random 1024-bit prime.

```
In [1]: p = galois.random_prime(1024); p
Out[1]:
302711861554170192179474787927862719156474620525867658512247564508724574369810999060567097656663068263171308881483830680302388827755716669799997493009638780318383161757492245312105505676375017778713275030196788866826746785182636880483511655527448746530195412433916327042116757316105007696887330341949675226854564791261007349076703205099125403932151776703687166010929011916252533566687600011051106220281329523836695973462572911797058011347969829757071750663178955070325400441259176733602981223944173232

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

```bash
$ openssl prime
23686178792697382206996886087214592029752524078026392358936844479667423570833116126506927877731514D68EDB7C650F1FF713531A1A43255A4BE6D6EE1FDBD96F4EB32757C1B1BAF16A5933E24D45FAD6C6A814F3C8C14F3C
→ (23686178792697382206996886087214592029752524078026392358936844479667423570833116126506927877731514D68EDB7C650F1FF713531A1A43255A4BE6D6EE1FDBD96F4EB32757C1B1BAF16A5933E24D45FAD6C6A814F3C8C14F3C)
→ is prime
```

galois.mersenne_exponents

```
galois.mersenne_exponents(n=None)

Returns all known Mersenne exponents \( e \) for \( e \leq n \).

A Mersenne exponent \( e \) is an exponent of 2 such that \( 2^e - 1 \) is prime.

Parameters
- \( n \) (int, optional) – The max exponent of 2. The default is None which returns all known Mersenne exponents.

Returns
- The list of Mersenne exponents \( e \) for \( e \leq n \).

Return type
- list
```

References

- https://oeis.org/A000043

Examples

```
# List all Mersenne exponents for Mersenne primes up to 2000 bits
In [1]: e = galois.mersenne_exponents(2000); e
Out[1]: [2, 3, 5, 7, 13, 17, 19, 31, 61, 89, 107, 127, 521, 607, 1279]

# Select one Merseene exponent and compute its Mersenne prime
In [2]: p = 2**e[-1] - 1; p
Out[2]:
104079321946643990819252403273640855386152622472667048053191123504036080596733602981223944173232

In [3]: galois.is_prime(p)
Out[3]: True
```
galois.mersenne_primes

`galois.mersenne_primes(n=None)`

Returns all known Mersenne primes $p$ for $p \leq 2^n - 1$.

Mersenne primes are primes that are one less than a power of 2.

**Parameters**

- `n (int, optional)` – The max power of 2. The default is `None` which returns all known Mersenne exponents.

**Returns**

The list of known Mersenne primes $p$ for $p \leq 2^n - 1$.

**Return type**

`list`

**References**

- [https://oeis.org/A000668](https://oeis.org/A000668)

**Examples**

```python
# 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,
  16225927682921336391578010288127,
  170141183460469231731687303715884105727,
  ...
  68647976601306097149819007990813932172694353001433054093944634591855431833976560521122559640661454...
  ...
  53113792816767098689588206552468627329593117727031923199444138200052655986085224273916255022652292...
  ...
  1040793219466439908192524032736408553861526224726670480531911235040360805967336029801223944173232...

In [2]: galois.is_prime(p[-1])
Out[2]: True
```
## 7.7.2 Primality tests

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>is_prime(n)</code></td>
<td>Determines if ( n ) is prime.</td>
</tr>
<tr>
<td><code>is_prime_power(n)</code></td>
<td>Determines if ( n ) is a prime power ( n = p^k ) for prime ( p ) and ( k \geq 1 ).</td>
</tr>
<tr>
<td><code>is_perfect_power(n)</code></td>
<td>Determines if ( n ) is a perfect power ( n = x^k ) for ( x &gt; 0 ) and ( k \geq 2 ).</td>
</tr>
<tr>
<td><code>is_composite(n)</code></td>
<td>Determines if ( n ) is composite.</td>
</tr>
<tr>
<td><code>is_square_free(value)</code></td>
<td>Determines if the positive integer or the non-constant, monic polynomial is square-free.</td>
</tr>
<tr>
<td><code>is_smooth(n, B)</code></td>
<td>Determines if the positive integer ( n ) is ( B )-smooth.</td>
</tr>
<tr>
<td><code>is_powersmooth(n, B)</code></td>
<td>Determines if the positive integer ( n ) is ( B )-powersmooth.</td>
</tr>
</tbody>
</table>

## 7.7.3 Specific primality tests

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fermat_primality_test(n[, a, rounds])</code></td>
<td>Determines if ( n ) is composite using Fermat’s primality test.</td>
</tr>
<tr>
<td><code>miller_rabin_primality_test(n[, a, rounds])</code></td>
<td>Determines if ( n ) is composite using the Miller-Rabin primality test.</td>
</tr>
</tbody>
</table>

### galois.fermat_primality_test

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

Determine if \( n \) is composite using Fermat’s primality test.

**Parameters**

- **n (int)** – An odd integer \( n \geq 3 \).
- **a (int, optional)** – An integer in \( 2 \leq a \leq n - 2 \). The default is `None` which selects a random \( a \).
- **rounds (int, optional)** – The number of iterations attempting to detect \( n \) as composite. Additional rounds will choose a new \( a \). The default is 1.

**Returns**

- **False** if \( n \) is shown to be composite. **True** if \( n \) is probable prime.

**Return type** **bool**

### Notes

Fermat’s theorem says that for prime \( p \) and \( 1 \leq a \leq p - 1 \), the congruence \( a^{p-1} \equiv 1 \pmod{p} \) holds. Fermat’s primality test of \( n \) computes \( a^{n-1} \pmod{n} \) for some \( 1 \leq a \leq n - 1 \). If \( a \) is such that \( a^{n-1} \not\equiv 1 \pmod{p} \), 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]
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, False, False]
```

And the pseudoprimes base 3 from A005935.

```
# List of some Fermat pseudoprimes to base 3
In [8]: pseudoprimes = [2465, 7381, 16531]
In [9]: [galois.is_prime(p) for p in pseudoprimes]
Out[9]: [False, False, False]

# The pseudoprimes base 3 satisfy 3^(p-1) = 1 (mod p)
In [10]: [galois.fermat_primality_test(p, a=3) for p in pseudoprimes]
Out[10]: [True, True, True]

# But they may not satisfy a^(p-1) = 1 (mod p) for other a
In [11]: [galois.fermat_primality_test(p) for p in pseudoprimes]
Out[11]: [True, False, False]
```
**galois.miller_rabin_primality_test**

Determines if \( n \) is composite using the Miller-Rabin primality test.

**Parameters**
- **n** (int) – An odd integer \( n \geq 3 \).
- **a** (int, optional) – An integer in \( 2 \leq a \leq n - 2 \). The default is 2.
- **rounds** (int, optional) – The number of iterations attempting to detect \( n \) as composite. Additional rounds will choose consecutive primes for \( a \).

**Returns**
- False if \( n \) is shown to be composite.
- True if \( n \) is probable prime.

**Return type**
- bool

**Notes**

The Miller-Rabin primality test is based on the fact that for odd \( n \) with factorization \( n = 2^s r \) for odd \( r \) and integer \( a \) such that \( \gcd(a, n) = 1 \), then either \( a^r \equiv 1 \pmod{n} \) or \( a^{2^j r} \equiv -1 \pmod{n} \) for some \( j \) in \( 0 \leq j \leq s - 1 \).

In the Miller-Rabin primality test, if \( a^r \not\equiv 1 \pmod{n} \) and \( a^{2^j r} \not\equiv -1 \pmod{n} \) for all \( j \) in \( 0 \leq j \leq s - 1 \), then \( a \) is called a strong witness to the compositeness of \( n \). If not, namely \( a^r \equiv 1 \pmod{n} \) or \( a^{2^j r} \equiv -1 \pmod{n} \) for any \( j \) in \( 0 \leq j \leq s - 1 \), then \( a \) is called a strong liar to the primality of \( n \) and \( n \) is called a strong pseudoprime to the base \( a \).

Since \( a = \{1, n - 1\} \) are strong liars for all composite \( n \), it is common to reduce the range of possible \( a \) to \( 2 \leq a \leq n - 2 \).

For composite odd \( n \), the probability that the Miller-Rabin test declares it a probable prime is less than \( \left(\frac{1}{4}\right)^t \), 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://cacr.uwaterloo.ca/hac/about/chap4.pdf)
- [https://math.dartmouth.edu/~carlp/PDF/paper25.pdf](https://math.dartmouth.edu/~carlp/PDF/paper25.pdf)

**Examples**

The Miller-Rabin primality test will never mark a true prime as composite.

```python
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 = [9,10,12,16,17,22,29,38,53,69,74,75,79,81,82]

In [5]: witnesses = [a for a in range(2, 90) if a not in strong_liars]

# All strong liars falsely assert that 91 is prime
In [6]: [galois.miller_rabin_primality_test(91, a=a) for a in strong_liars] ==
   
   →[True,]*len(strong_liars)
Out[6]: True

# All other a are witnesses to the compositeness of 91
In [7]: [galois.miller_rabin_primality_test(91, a=a) for a in witnesses] == [False,
   
   →]*len(witnesses)
Out[7]: True

7.8 Numpy Examples

This section contains examples of some numpy functions when called on Galois field arrays. Many more functions are supported, just not explicitly documented here.

7.8.1 General

<table>
<thead>
<tr>
<th>np.copy(a)</th>
<th>Returns a copy of a given Galois field array.</th>
</tr>
</thead>
<tbody>
<tr>
<td>np.concatenate(arrays[, axis])</td>
<td>Concatenates the input arrays along the given axis.</td>
</tr>
<tr>
<td>np.insert(array, object, values[, axis])</td>
<td>Inserts values along the given axis.</td>
</tr>
</tbody>
</table>

np.copy

np.copy(a)

Returns a copy of a given Galois field array.


**Warning:** This function returns an numpy.ndarray, not an instance of the subclass. To return a copy of the subclass, pass subok=True (for numpy version 1.19 and above) or use a.copy().

Examples

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

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

# NOTE: b is an ndarray
In [3]: b = np.copy(a); b
Out[3]: array([3, 7, 7, 4, 3], dtype=uint8)
In [4]: type(b)
Out[4]: numpy.ndarray

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

# b is unmodified
In [6]: b
Out[6]: array([3, 7, 7, 4, 3], dtype=uint8)

In [7]: a.copy()
Out[7]: GF([0, 7, 7, 4, 3], order=2^3)

np.concatenate

np.concatenate(arrays, axis=0)

Concatenates the input arrays along the given axis.


Examples

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

In [2]: A = GF.Random((2,2)); A
Out[2]:
GF([[3, 2],
    [5, 0]], order=2^3)

In [3]: B = GF.Random((2,2)); B
Out[3]:
GF([[2, 5],
    [5, 6]], order=2^3)

In [4]: np.concatenate((A,B), axis=0)
Out[4]:
GF([[3, 2, 2, 5],
    [5, 0, 5, 6]], order=2^3)

In [5]: np.concatenate((A,B), axis=1)
Out[5]:
GF([[3, 2, 2, 5],
    [5, 0, 5, 6]], order=2^3)
np.insert

np.insert(array, object, values, axis=None)

Inserts values along the given axis.


Examples

In [1]: GF = galois.GF(2**3)
In [2]: x = GF.Random(5); x
Out[2]: GF([7, 6, 1, 3, 2], order=2^3)
In [3]: np.insert(x, 1, [0,1,2,3])
Out[3]: GF([7, 0, 1, 2, 3, 6, 1, 3, 2], order=2^3)

7.8.2 Arithmetic

| np.add(x, y) | Adds two Galois field arrays element-wise. |
| np.subtract(x, y) | Subtracts two Galois field arrays element-wise. |
| np.multiply(x, y) | Multiplies two Galois field arrays element-wise. |
| np.divide(x, y) | Divides two Galois field arrays element-wise. |
| np.negative(x) | Returns the element-wise additive inverse of a Galois field array. |
| np.reciprocal(x) | Returns the element-wise multiplicative inverse of a Galois field array. |
| np.power(x, y) | Exponentiates a Galois field array element-wise. |
| np.square(x) | Squares a Galois field array element-wise. |
| np.log(x) | Computes the logarithm (base GF.primitive_element) of a Galois field array element-wise. |
| np.matmul(a, b) | Returns the matrix multiplication of two Galois field arrays. |

np.add

np.add(x, y)

Adds two Galois field arrays element-wise.
References


Examples

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

In [2]: x = GF.Random(10); x
Out[2]: GF([13, 27, 21, 9, 3, 15, 0, 14, 19, 26], order=31)

In [3]: y = GF.Random(10); y
Out[3]: GF([12, 1, 22, 5, 29, 5, 23, 8, 28, 21], order=31)

In [4]: np.add(x, y)
Out[4]: GF([25, 28, 12, 14, 1, 20, 23, 22, 16, 16], order=31)

In [5]: x + y
Out[5]: GF([25, 28, 12, 14, 1, 20, 23, 22, 16, 16], order=31)
```

`np.subtract`

`np.subtract(x, y)`

Subtracts two Galois field arrays element-wise.

References


Examples

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

In [2]: x = GF.Random(10); x
Out[2]: GF([29, 11, 8, 2, 11, 1, 12, 0, 22, 20], order=31)

In [3]: y = GF.Random(10); y
Out[3]: GF([7, 21, 2, 19, 24, 16, 18, 9, 19, 4], order=31)

In [4]: np.subtract(x, y)
Out[4]: GF([22, 21, 6, 14, 18, 16, 25, 22, 3, 16], order=31)

In [5]: x - y
Out[5]: GF([22, 21, 6, 14, 18, 16, 25, 22, 3, 16], order=31)
```
np.multiply

np.multiply(x, y)

Multiplies two Galois field arrays element-wise.

References


Examples

Multiplying two Galois field arrays results in field multiplication.

```
In [1]: GF = galois.GF(31)
In [2]: x = GF.Random(10); x
Out[2]: GF([ 0, 21, 16, 16, 15, 28, 13, 8, 29, 27], order=31)
In [3]: y = GF.Random(10); y
Out[3]: GF([ 2, 26, 25, 3, 17, 14, 29, 5, 8, 9], order=31)
In [4]: np.multiply(x, y)
Out[4]: GF([ 0, 19, 28, 17, 7, 20, 5, 9, 15, 26], order=31)
In [5]: x * y
Out[5]: GF([ 0, 19, 28, 17, 7, 20, 5, 9, 15, 26], order=31)
```

Multiplying a Galois field array with an integer results in scalar multiplication.

```
In [6]: GF = galois.GF(31)
In [7]: x = GF.Random(10); x
Out[7]: GF([ 2, 29, 0, 6, 4, 25, 1, 13, 2, 11], order=31)
In [8]: np.multiply(x, 3)
Out[8]: GF([ 6, 25, 0, 18, 12, 13, 3, 8, 6, 2], order=31)
In [9]: x * 3
Out[9]: GF([ 6, 25, 0, 18, 12, 13, 3, 8, 6, 2], order=31)
```

```
In [10]: print(GF.properties)
GF(31):
    characteristic: 31
    degree: 1
    order: 31
    irreducible_poly: x + 28
    is_primitive_poly: True
    primitive_element: 3

# Adding `characteristic` copies of any element always results in zero
In [11]: x * GF.characteristic
Out[11]: GF([0, 0, 0, 0, 0, 0, 0, 0, 0, 0], order=31)
```
np.divide

np.divide(x, y)
Divides two Galois field arrays element-wise.

References


Examples

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

In [2]: x = GF.Random(10); x
Out[2]: GF([ 7, 22, 29, 3, 3, 10, 2, 17, 3, 30], order=31)

In [3]: y = GF.Random(10, low=1); y
Out[3]: GF([20, 1, 16, 7, 27, 16, 2, 3, 15, 5], order=31)

In [4]: z = np.divide(x, y); z
Out[4]: GF([ 5, 22, 27, 27, 7, 20, 1, 16, 25, 6], order=31)

In [5]: y * z
Out[5]: GF([ 7, 22, 29, 3, 3, 10, 2, 17, 3, 30], order=31)

In [6]: np.true_divide(x, y)
Out[6]: GF([ 5, 22, 27, 27, 7, 20, 1, 16, 25, 6], order=31)

In [7]: x / y
Out[7]: GF([ 5, 22, 27, 27, 7, 20, 1, 16, 25, 6], order=31)

In [8]: np.floor_divide(x, y)
Out[8]: GF([ 5, 22, 27, 27, 7, 20, 1, 16, 25, 6], order=31)

In [9]: x // y
Out[9]: GF([ 5, 22, 27, 27, 7, 20, 1, 16, 25, 6], order=31)

np.negative

np.negative(x)
Returns the element-wise additive inverse of a Galois field array.
References


Examples

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

In [2]: x = GF.Random(10); x
Out[2]: GF([12, 11, 5, 28, 30, 15, 13, 22, 7, 20], order=31)

In [3]: y = np.negative(x); y
Out[3]: GF([19, 20, 26, 3, 1, 16, 18, 9, 24, 11], order=31)

In [4]: x + y
Out[4]: GF([0, 0, 0, 0, 0, 0, 0, 0, 0, 0], order=31)

In [5]: -x
Out[5]: GF([19, 20, 26, 3, 1, 16, 18, 9, 24, 11], order=31)

In [6]: -1*x
Out[6]: GF([19, 20, 26, 3, 1, 16, 18, 9, 24, 11], order=31)
```

np.reciprocal

np.reciprocal(x)

Returns the element-wise multiplicative inverse of a Galois field array.

References


Examples

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

In [2]: x = GF.Random(5, low=1); x
Out[2]: GF([29, 10, 19, 30, 18], order=31)

In [3]: y = np.reciprocal(x); y
Out[3]: GF([15, 28, 18, 30, 19], order=31)

In [4]: x * y
Out[4]: GF([1, 1, 1, 1, 1], order=31)

In [5]: x ** -1
Out[5]: GF([15, 28, 18, 30, 19], order=31)

In [6]: GF(1) / x
```

(continues on next page)
np.power

np.power(x, y)

Exponentiates a Galois field array element-wise.

References


Examples

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

In [2]: x = GF.Random(10); x
Out[2]: GF([24, 2, 20, 13, 26, 14, 17, 29, 16, 6], order=31)

In [3]: np.power(x, 3)
Out[3]: GF([29, 8, 2, 27, 30, 16, 15, 23, 4, 30], order=31)

In [4]: x ** 3
Out[4]: GF([29, 8, 2, 27, 30, 16, 15, 23, 4, 30], order=31)

In [5]: x * x * x
Out[5]: GF([29, 8, 2, 27, 30, 16, 15, 23, 4, 30], order=31)

In [6]: x = GF.Random(10, low=1); x
Out[6]: GF([9, 22, 13, 23, 5, 11, 30, 21, 17, 17], order=31)

In [7]: y = np.random.randint(-10, 10, 10); y
Out[7]: array([-1, 3, -6, -8, 7, 0, 8, -1, 8, 8])

In [8]: np.power(x, y)
Out[8]: GF([7, 15, 2, 2, 5, 1, 1, 3, 18, 18], order=31)

In [9]: x ** y
Out[9]: GF([7, 15, 2, 2, 5, 1, 1, 3, 18, 18], order=31)
np.square

np.square(x)
Squares a Galois field array element-wise.

References

Examples

In [1]: GF = galois.GF(31)
In [2]: x = GF.Random(10); x
Out[2]: GF([13, 10, 8, 26, 8, 11, 16, 5, 22, 18], order=31)
In [3]: np.square(x)
Out[3]: GF([14, 7, 2, 25, 2, 28, 8, 25, 19, 14], order=31)
In [4]: x ** 2
Out[4]: GF([14, 7, 2, 25, 2, 28, 8, 25, 19, 14], order=31)
In [5]: x * x
Out[5]: GF([14, 7, 2, 25, 2, 28, 8, 25, 19, 14], order=31)

np.log

np.log(x)
Computes the logarithm (base GF.primitive_element) of a Galois field array element-wise.

Calling np.log() implicitly uses base galois.FieldClass.primitive_element. See galois.FieldArray.log() for logarithm with arbitrary base.

References

Examples

In [1]: GF = galois.GF(31)
In [2]: alpha = GF.primitive_element; alpha
Out[2]: GF(3, order=31)
In [3]: x = GF.Random(10, low=1); x
Out[3]: GF([14, 8, 23, 13, 24, 24, 23, 26, 22, 25], order=31)
In [4]: y = np.log(x); y
Out[4]: array([22, 12, 27, 11, 13, 13, 27, 5, 17, 10])

(continues on next page)
In [5]: alpha ** y
Out[5]: GF([14, 8, 23, 13, 24, 24, 23, 26, 22, 25], order=31)

np.matmul

np.matmul(a, b)

Returns the matrix multiplication of two Galois field arrays.

References


Examples

In [1]: GF = galois.GF(31)
In [2]: A = GF.Random((3,3)); A
Out[2]:
GF([[27, 8, 17],
    [30, 27, 27],
    [ 7, 15,  1]], order=31)
In [3]: B = GF.Random((3,3)); B
Out[3]:
GF([[16, 0, 0],
    [11, 10, 23],
    [29, 23, 18]], order=31)
In [4]: np.matmul(A, B)
Out[4]:
GF([[21, 6, 25],
    [10, 23, 22],
    [27, 18, 22]], order=31)
In [5]: A @ B
Out[5]:
GF([[21, 6, 25],
    [10, 23, 22],
    [27, 18, 22]], order=31)
## 7.8.3 Advanced Arithmetic

`np.convolve(a, b)`  
Convolves the input arrays.

### Examples

```
In [1]: GF = galois.GF(31)
In [2]: a = GF.Random(10)
In [3]: b = GF.Random(10)

In [4]: np.convolve(a, b)
Out[4]:
GF([3, 18, 1, 8, 14, 13, 29, 25, 17, 29, 23, 2, 2, 12, 20, 5, 6,
     3, 10], order=31)

# Equivalent implementation with native numpy
In [5]: np.convolve(a.view(np.ndarray).astype(int), b.view(np.ndarray).astype(int))
    →% 31
Out[5]:
array([3, 18, 1, 8, 14, 13, 29, 25, 17, 29, 23, 2, 2, 12, 20, 5, 6,
      3, 10])
```

```
In [6]: GF = galois.GF(2**8)
In [7]: a = GF.Random(10)
In [8]: b = GF.Random(10)

In [9]: np.convolve(a, b)
Out[9]:
GF([197, 176, 120, 213, 244, 48, 52, 59, 134, 244, 62, 70, 214, 71,
    222, 101, 126, 171, 219], order=2^8)
```
7.8.4 Linear Algebra

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>np.dot(a, b)</code></td>
<td>Returns the dot product of two Galois field arrays.</td>
</tr>
<tr>
<td><code>np.vdot(a, b)</code></td>
<td>Returns the dot product of two Galois field vectors.</td>
</tr>
<tr>
<td><code>np.inner(a, b)</code></td>
<td>Returns the inner product of two Galois field arrays.</td>
</tr>
<tr>
<td><code>np.outer(a, b)</code></td>
<td>Returns the outer product of two Galois field arrays.</td>
</tr>
<tr>
<td><code>np.matmul(a, b)</code></td>
<td>Returns the matrix multiplication of two Galois field arrays.</td>
</tr>
<tr>
<td><code>np.linalg.matrix_power(x)</code></td>
<td>Raises a square Galois field matrix to an integer power.</td>
</tr>
<tr>
<td><code>np.linalg.det(A)</code></td>
<td>Computes the determinant of the matrix.</td>
</tr>
<tr>
<td><code>np.linalg.matrix_rank(x)</code></td>
<td>Returns the rank of a Galois field matrix.</td>
</tr>
<tr>
<td><code>np.trace(x)</code></td>
<td>Returns the sum along the diagonal of a Galois field array.</td>
</tr>
<tr>
<td><code>np.linalg.solve(x)</code></td>
<td>Solves the system of linear equations.</td>
</tr>
<tr>
<td><code>np.linalg.inv(A)</code></td>
<td>Computes the inverse of the matrix.</td>
</tr>
</tbody>
</table>

**np.dot**

\[ \text{np.dot}(a, b) \]

Returns the dot product of two Galois field arrays.

**References**


**Examples**

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

In [2]: a = GF.Random(3); a
Out[2]: GF([29, 15, 20], order=31)

In [3]: b = GF.Random(3); b
Out[3]: GF([30, 5, 21], order=31)

In [4]: np.dot(a, b)
Out[4]: GF(1, order=31)

In [5]: A = GF.Random((3,3)); A
Out[5]:
GF([ [ 8, 11, 4],
    [ 5, 14, 20],
    [27, 17, 29]], order=31)

In [6]: B = GF.Random((3,3)); B
Out[6]:
GF([ [ 1, 0, 19],
    [28, 29, 25],
    [22, 3, 26]], order=31)
```
np.dot

np.vdot(a, b)
Returns the dot product of two Galois field vectors.

References


Examples

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

In [2]: a = GF.Random(3); a
Out[2]: GF([14, 2, 24], order=31)

In [3]: b = GF.Random(3); b
Out[3]: GF([13, 19, 21], order=31)

In [4]: np.vdot(a, b)
Out[4]: GF(11, order=31)

In [5]: A = GF.Random((3,3)); A
Out[5]: GF([[ 4, 13, 13],
          [10, 7, 22],
          [16, 10, 16]], order=31)

In [6]: B = GF.Random((3,3)); B
Out[6]: GF([[ 1, 9, 8],
          [9, 30, 28],
          [29, 12, 7]], order=31)

In [7]: np.vdot(A, B)
Out[7]: GF(8, order=31)
np.inner

np.inner(a, b)

Returns the inner product of two Galois field arrays.

References


Examples

```python
In [1]: GF = galois.GF(31)
In [2]: a = GF.Random(3); a
Out[2]: GF([12, 3, 19], order=31)
In [3]: b = GF.Random(3); b
Out[3]: GF([20, 20, 10], order=31)
In [4]: np.inner(a, b)
Out[4]: GF(25, order=31)
```

```python
In [5]: A = GF.Random((3,3)); A
Out[5]: GF([[27, 10, 15],
            [28, 3, 13],
            [1, 18, 13]], order=31)
In [6]: B = GF.Random((3,3)); B
Out[6]: GF([[25, 5, 23],
            [2, 20, 11],
            [25, 30, 25]], order=31)
In [7]: np.inner(A, B)
Out[7]: GF([[16, 16, 17],
            [22, 11, 30],
            [11, 9, 22]], order=31)
```

np.outer

np.outer(a, b)

Returns the outer product of two Galois field arrays.
References


Examples

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

In [2]: a = GF.Random(3); a
Out[2]: GF([11, 12, 19], order=31)

In [3]: b = GF.Random(3); b
Out[3]: GF([14, 23, 19], order=31)

In [4]: np.outer(a, b)
Out[4]:
GF([[30, 5, 23],
    [13, 28, 11],
    [18, 3, 20]], order=31)
```

`np.linalg.matrix_power`

`np.linalg.matrix_power(x)`

Raises a square Galois field matrix to an integer power.

References


Examples

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

In [2]: A = GF.Random((3,3)); A
Out[2]: GF([[21, 26, 5],
    [16, 19, 13],
    [16, 7, 2]], order=31)

In [3]: np.linalg.matrix_power(A, 3)
Out[3]:
GF([[12, 1, 5],
    [20, 29, 17],
    [2, 5, 25]], order=31)

In [4]: A @ A @ A
Out[4]:
GF([[12, 1, 5],
    [20, 29, 17],
    [2, 5, 25]], order=31)
```
In [5]: GF = galois.GF(31)

# Ensure A is full rank and invertible
In [6]: while True:
    ...:     A = GF.Random((3,3))
    ...:     if np.linalg.matrix_rank(A) == 3:
    ...:         break
    ...:

In [7]: A
Out[7]:
GF([[28, 5, 19],
     [16, 6, 11],
     [12, 10, 23]], order=31)

In [8]: np.linalg.matrix_power(A, -3)
Out[8]:
GF([[18, 0, 5],
     [ 8, 20, 13],
     [19, 7, 10]], order=31)

In [9]: A_inv = np.linalg.inv(A)

In [10]: A_inv @ A_inv @ A_inv
Out[10]:
GF([[18, 0, 5],
     [ 8, 20, 13],
     [19, 7, 10]], order=31)

np.linalg.det

np.linalg.det(A)
Computes the determinant of the matrix.

References


Examples

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

In [2]: A = GF.Random((2,2)); A
Out[2]:
GF([[22, 23],
     [18, 17]], order=31)

In [3]: np.linalg.det(A)
Out[3]: GF(22, order=31)
In [4]: A[0,0]*A[1,1] - A[0,1]*A[1,0]
Out[4]: GF(2, order=31)

np.linalg.matrix_rank

np.linalg.matrix_rank(x)
Returns the rank of a Galois field matrix.

References


Examples

In [1]: GF = galois.GF(31)
In [2]: A = GF.Identity(4); A
Out[2]:
GF([[1, 0, 0, 0],
    [0, 1, 0, 0],
    [0, 0, 1, 0],
    [0, 0, 0, 1]], order=31)
In [3]: np.linalg.matrix_rank(A)
Out[3]: 4

One column is a linear combination of another.

In [4]: GF = galois.GF(31)
In [5]: A = GF.Random((4,4)); A
Out[5]:
GF([[4, 0, 3, 4],
    [14, 28, 19, 16],
    [8, 21, 26, 20],
    [0, 17, 12, 2]], order=31)
In [6]: A[:,2] = A[:,1] * GF(17); A
Out[6]:
GF([[4, 0, 0, 4],
    [14, 28, 11, 16],
    [8, 21, 16, 20],
    [0, 17, 10, 2]], order=31)
In [7]: np.linalg.matrix_rank(A)
Out[7]: 3

One row is a linear combination of another.

7.8. Numpy Examples
In [8]: GF = galois.GF(31)

In [9]: A = GF.Random((4,4)); A
Out[9]:
GF([[19, 6, 3, 6],
    [ 5, 23, 28, 4],
    [25, 11, 20, 27],
    [18, 6, 0, 12]], order=31)

In [10]: A[3,:] = A[2,:] * GF(8); A
Out[10]:
GF([[19, 6, 3, 6],
    [ 5, 23, 28, 4],
    [25, 11, 20, 27],
    [14, 26, 5, 30]], order=31)

In [11]: np.linalg.matrix_rank(A)
Out[11]: 3

np.trace

np.trace(x)

Returns the sum along the diagonal of a Galois field array.

References


Examples

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

In [2]: A = GF.Random((5,6)); A
Out[2]:
GF([[11, 8, 8, 13, 27, 1],
    [26, 16, 17, 13, 25, 2],
    [26, 8, 23, 12, 7, 4],
    [14, 28, 23, 21, 19, 19],
    [30, 12, 5, 28, 21, 22]], order=31)

In [3]: np.trace(A)
Out[3]: GF(30, order=31)

Out[4]: GF(30, order=31)

In [5]: np.trace(A, offset=1)
Out[5]: GF(16, order=31)
Out[6]: GF(16, order=31)

np.linalg.solve

np.linalg.solve(x)
Solves the system of linear equations.

References


Examples

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

# Ensure A is full rank and invertible
In [2]: while True:
    ...:     A = GF.Random((4,4))
    ...:     if np.linalg.matrix_rank(A) == 4:
    ...:         break
    ...:

In [3]: A
Out[3]:
GF([[23, 17, 10, 9],
    [20, 29, 9, 11],
    [ 2, 8, 1, 21],
    [13, 5, 8, 0]], order=31)

In [4]: b = GF.Random(4); b
Out[4]: GF([ 7, 19, 3, 22], order=31)

In [5]: x = np.linalg.solve(A, b); x
Out[5]: GF([14, 3, 13, 0], order=31)

In [6]: A @ x
Out[6]: GF([ 7, 19, 3, 22], order=31)

In [7]: GF = galois.GF(31)

# Ensure A is full rank and invertible
In [8]: while True:
    ...:     A = GF.Random((4,4))
    ...:     if np.linalg.matrix_rank(A) == 4:
    ...:         break
    ...:

In [9]: A
```python
In [10]: B = GF.Random((4,2)); B
Out[10]:
GF([[ 4, 13],
     [ 8, 16],
     [ 7,  4],
     [18, 25]], order=31)
```

```python
In [11]: X = np.linalg.solve(A, B); X
Out[11]:
GF([[13, 13],
    [ 4,  1],
    [27, 10],
    [30,  3]], order=31)
```

```python
In [12]: A @ X
Out[12]:
GF([[ 4, 13],
     [ 8, 16],
     [ 7,  4],
     [18, 25]], order=31)
```

**np.linalg.inv**

np.linalg.inv(A)

Computes the inverse of the matrix.

**References**


**Examples**

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

# Ensure A is full rank and invertible
In [2]: while True:
   ...:     A = GF.Random((3,3))
   ...:     if np.linalg.matrix_rank(A) == 3:
   ...:         break
   ...:
In [3]: A
Out[3]:
```

\begin{Verbatim}
GF([10, 0, 11],
     [23, 30, 15],
     [13, 30, 15]), order=31)

In [4]: A_inv = np.linalg.inv(A); A_inv
Out[4]:
GF([[ 0, 28, 3],
    [ 7, 16, 14],
    [17, 14, 17]], order=31)

In [5]: A_inv @ A
Out[5]:
GF([[1, 0, 0],
    [0, 1, 0],
    [0, 0, 1]], order=31)
\end{Verbatim}
ACKNOWLEDGEMENTS

• This library is an extension of, and completely dependent on, NumPy.
• We heavily rely on Numba and its just-in-time compiler for optimizing performance of Galois field arithmetic.
• We use Frank Luebeck’s compilation of Conway polynomials.
• We also use Wolfram’s compilation of primitive polynomials.
• We extensively use SageMath for generating test vectors.
• We also use Octave for generating test vectors.

Many thanks!
If this library was useful to you in your research, please cite us. Following the GitHub citation standards, here is the recommended citation.

Bibtex:
```
@misc{Hostetter_Galois_2020,
    title = {{Galois: A performant NumPy extension for Galois fields}},
    author = {Hostetter, Matt},
    month = {11},
    year = {2020},
    url = {https://github.com/mhostetter/galois},
}
```

APA:
```
```
RELEASE NOTES

10.1 v0.0.21

10.1.1 Changes

• Fix docstrings and code completion for Python classes that weren’t rendering correctly in an IDE.
• Various documentation improvements.

10.1.2 Contributors

• Matt Hostetter (@mhostetter)

10.2 v0.0.20

10.2.1 Breaking Changes

• Move poly_gcd() functionality into gcd().
• Move poly_egcd() functionality into egcd().
• Move poly_factors() functionality into factors().

10.2.2 Changes

• Fix polynomial factorization algorithms. Previously only parital factorization was implemented.
• Support generating and testing irreducible and primitive polynomials over extension fields.
• Support polynomial input to is_square_free().
• Minor documentation improvements.
• Pin Numba dependency to <0.54
10.2.3 Contributors

- Matt Hostetter (@mhostetter)

10.3 v0.0.19

10.3.1 Breaking Changes

- Remove unnecessary is_field() function. Use isinstance(x, galois.FieldClass) or isinstance(x, galois.FieldArray) instead.
- Remove log_naive() function. Might be re-added later through np.log() on a multiplicative group array.
- Rename mode kwarg in galois.GF() to compile.
- Revert np.copy() override that always returns a subclass. Now, by default it does not return a subclass. To return a Galois field array, use x.copy() or np.copy(x, subok=True) instead.

10.3.2 Changes

- Improve documentation.
- Improve unit test coverage.
- Add benchmarking tests.
- Add initial LFSR implementation.
- Add display kwarg to galois.GF() class factory to set the display mode at class-creation time.
- Add Poly.reverse() method.
- Allow polynomial strings as input to galois.GF(). For example, galois.GF(2**4, irreducible_poly="x^4 + x + 1").
- Enable np.divmod() and np.remainder() on Galois field arrays. The remainder is always zero, though.
- Fix bug in bch_valid_codes() where repetition codes weren’t included.
- Various minor bug fixes.

10.3.3 Contributors

- Matt Hostetter (@mhostetter)

10.4 v0.0.18

10.4.1 Breaking Changes

- Make API more consistent with software like Matlab and Wolfram:
  - Rename galois.prime_factors() to galois.factors().
  - Rename galois.gcd() to galois.egcd() and add galois.gcd() for conventional GCD.
- Rename `galois.poly_gcd()` to `galois.poly_egcd()` and add `galois.poly_gcd()` for conventional GCD.
- Rename `galois.euler_totient()` to `galois.euler_phi()`.
- Rename `galois.carmichael()` to `galois.carmichael_lambda()`.
- Rename `galois.is_prime_fermat()` to `galois.fermat_primality_test()`.
- Rename `galois.is_prime_miller_rabin()` to `galois.miller_rabin_primality_test()`.

- Rename polynomial search method keyword argument values from ["smallest", "largest", "random"] to ["min", "max", "random"].

### 10.4.2 Changes

- Clean up `galois` API and `dir()` so only public classes and functions are displayed.
- Speed-up `galois.is_primitive()` test and search for primitive polynomials in `galois.primitive_poly()`.
- Speed-up `galois.is_smooth()`.
- Add Reed-Solomon codes in `galois.ReedSolomon`.
- Add shortened BCH and Reed-Solomon codes.
- Add error detection for BCH and Reed-Solomon with the `detect()` method.
- Add general cyclic linear block code functions.
- Add Matlab default primitive polynomial with `galois.matlab_primitive_poly()`.
- Add number theoretic functions:
  - Add `galois.legendre_symbol()`, `galois.jacobi_symbol()`, `galois.kronecker_symbol()`.
  - Add `galois.divisors()`, `galois.divisor_sigma()`.
  - Add `galois.is_composite()`, `galois.is_prime_power()`, `galois.is_perfect_power()`, `galois.is_square_free()`, `galois.is_powersmooth()`.
  - Add `galois.are_coprime()`.
- Clean up integer factorization algorithms and add some to public API:
  - Add `galois.withdraw()`.
- Fix minor bugs in BCH codes.
10.4.3 Contributors

- Matt Hostetter (@mhostetter)

10.5 v0.0.17

10.5.1 Breaking Changes

- Rename FieldMeta to FieldClass.
- Remove target keyword from FieldClass.compile() until there is better support for GPUs.
- Consolidate verify_irreducible and verify_primitive keyword arguments into verify for the galois.GF() class factory function.
- Remove group arrays until there is more complete support.

10.5.2 Changes

- Speed-up Galois field class creation time.
- Speed-up JIT compilation time by caching functions.
- Speed-up Poly.roots() by JIT compiling it.
- Add BCH codes with galois.BCH.
- Add ability to generate irreducible polynomials with irreducible_poly() and irreducible_polys().
- Add ability to generate primitive polynomials with primitive_poly() and primitive_polys().
- Add computation of the minimal polynomial of an element of an extension field with minimal_poly().
- Add display of arithmetic tables with FieldClass.arithmetic_table().
- Add display of field element representation table with FieldClass.repr_table().
- Add Berlekamp-Massey algorithm in berlekamp_massey().
- Enable ipython tab-completion of Galois field classes.
- Cleanup API reference page.
- Add introduction to Galois fields tutorials.
- Fix bug in isPrimitive() where some reducible polynomials were marked irreducible.
- Fix bug in integer<->polynomial conversions for large binary polynomials.
- Fix bug in “power” display mode of 0.
- Other minor bug fixes.
10.5.3 Contributors

- Dominik Wernberger (@Werni2A)
- Matt Hostetter (@mhostetter)

10.6 v0.0.16

10.6.1 Changes

- Add Field() alias of GF() class factory.
- Add finite groups modulo n with Group() class factory.
- Add is_group(), is_field(), is_prime_field(), is_extension_field().
- Add polynomial constructor Poly.String().
- Add polynomial factorization in poly_factors().
- Add np.vdot() support.
- Fix PyPI packaging issue from v0.0.15.
- Fix bug in creation of 0-degree polynomials.
- Fix bug in poly_gcd() not returning monic GCD polynomials.

10.6.2 Contributors

- Matt Hostetter (@mhostetter)

10.7 v0.0.15

10.7.1 Breaking Changes

- Rename poly_exp_mod() to poly_pow() to mimic the native pow() function.
- Rename fermat_primality_test() to is_prime_fermat().
- Rename miller_rabin_primality_test() to is_prime_miller_rabin().

10.7.2 Changes

- Massive linear algebra speed-ups. (See #88)
- Massive polynomial speed-ups. (See #88)
- Various Galois field performance enhancements. (See #92)
- Support np.convolve() for two Galois field arrays.
- Allow polynomial arithmetic with Galois field scalars (of the same field). (See #99), e.g.
>>> GF = galois.GF(3)

>>> p = galois.Poly([1,2,0], field=GF)
Poly(x^2 + 2x, GF(3))

>>> p * GF(2)
Poly(2x^2 + x, GF(3))

- Allow creation of 0-degree polynomials from integers. (See #99), e.g.

>>> p = galois.Poly(1)
Poly(1, GF(2))

- Add the four Oakley fields from RFC 2409.
- Speed-up unit tests.
- Restructure API reference.

### 10.7.3 Contributors

- Matt Hostetter (@mhostetter)

### 10.8 v0.0.14

#### 10.8.1 Breaking Changes

- Rename GFArray.Eye() to GFArray.Identity().
- Rename chinese_remainder_theorem() to crt().

#### 10.8.2 Changes

- Lots of performance improvements.
- Additional linear algebra support.
- Various bug fixes.

#### 10.8.3 Contributors

- Baalateja Kataru (@BK-Modding)
- Matt Hostetter (@mhostetter)
CHAPTER
ELEVEN

INDICES AND TABLES

- genindex
- modindex
- search
g

galois, 61
Symbols

__add__() (galois.FieldArray method), 74
__add__() (galois.Poly method), 139
__divmod__() (galois.FieldArray method), 75
__divmod__() (galois.Poly method), 140
__floordiv__() (galois.FieldArray method), 76
__floordiv__() (galois.Poly method), 140
__init__() (galois.BCH method), 167
__init__() (galois.FieldArray method), 76
__init__() (galois.FieldClass method), 85
__init__() (galois.GF2 method), 110
__init__() (galois.LFSR method), 194
__init__() (galois.Poly method), 141
__init__() (galois.ReedSolomon method), 178
__mod__() (galois.FieldArray method), 77
__mod__() (galois.Poly method), 141
__mul__() (galois.FieldArray method), 77
__mul__() (galois.Poly method), 141
__pow__() (galois.FieldArray method), 78
__pow__() (galois.Poly method), 142
__sub__() (galois.FieldArray method), 79
__sub__() (galois.Poly method), 142
__truediv__() (galois.FieldArray method), 79
__truediv__() (galois.Poly method), 142

A

add() (in module np), 229
are_coprime() (in module galois), 154
arithmetic_table() (galois.FieldClass method), 85

B

BCH (class in galois), 166
bch_valid_codes() (in module galois), 190
berlekamp_massey() (in module galois), 198

C

c (galois.ReedSolomon property), 185
carmichael_lambda() (in module galois), 201
characteristic (galois.FieldClass property), 93
coeffs (galois.Poly property), 146
compile() (galois.FieldClass method), 88
concatenate() (in module np), 228
config (galois.LFSR property), 195
convolve() (in module np), 237
conway_poly() (in module galois), 123
copy() (in module np), 227
crt() (in module galois), 156

D

d (galois.BCH property), 173
d (galois.ReedSolomon property), 185
decode() (galois.BCH method), 168
decode() (galois.ReedSolomon method), 179
default_ufunc_mode (galois.FieldClass property), 94
degree (galois.FieldClass property), 94
degree (galois.Poly property), 147
degrees (galois.Poly property), 147
Degrees() (galois.Poly class method), 134
derivative() (galois.Poly method), 143
det() (in module np.linalg), 242
detect() (galois.BCH method), 170
detect() (galois.ReedSolomon method), 181
display() (galois.FieldClass method), 88
display_mode (galois.FieldClass property), 95
distinct_degree_factorization() (in module galois), 161
divide() (in module np), 232
divisor_sigma() (in module galois), 211
divisors() (in module galois), 210
dot() (in module np), 238
dtypes (galois.FieldClass property), 96

e

egcd() (in module galois), 151
Elements() (galois.FieldClass method), 69
Elements() (galois.GF2 method), 105
decode() (galois.BCH method), 171
decode() (galois.ReedSolomon method), 182
equal_degree_factorization() (in module galois), 163
euler_phi() (in module galois), 199
F
factors() (in module galois), 158
fermat_primality_test() (in module galois), 224
field (galois.BCH property), 173
field (galois.LFSR property), 196
field (galois.Poly property), 147
field (galois.ReedSolomon property), 185
Field() (in module galois), 66
FieldArray (class in galois), 66
FieldClass (class in galois), 84
G
G (galois.BCH property), 172
G (galois.ReedSolomon property), 184
galois
    module, 61
gcd() (in module galois), 150
generator_poly (galois.BCH property), 174
generator_poly (galois.ReedSolomon property), 186
generator_to_parity_check_matrix() (in module galois), 188
GF() (in module galois), 61
GF2 (class in galois), 104
H
H (galois.BCH property), 173
H (galois.ReedSolomon property), 184
I
Identity() (galois.FieldArray class method), 70
Identity() (galois.GF2 class method), 106
Identity() (galois.Poly class method), 135
ilog() (in module galois), 209
initial_state (galois.LFSR property), 196
inner() (in module np), 240
insert() (in module np), 229
integer (galois.Poly property), 148
Integer() (galois.Poly class method), 135
inv() (in module np.linalg), 246
iroot() (in module galois), 209
irreducible_poly (galois.FieldClass property), 97
irreducible_poly() (in module galois), 117
irreducible_polys() (in module galois), 118
is_composite() (in module galois), 217
is_cyclic() (in module galois), 205
is_extension_field (galois.FieldClass property), 98
is_irreducible() (in module galois), 120
is_monic() (in module galois), 164
is_narrow_sense (galois.BCH property), 174
is_narrow_sense (galois.ReedSolomon property), 186
is_perfect_power() (in module galois), 217
is_powersmooth() (in module galois), 218
is_prime() (in module galois), 216
is_prime_field (galois.FieldClass property), 98
is_prime_power() (in module galois), 216
is_primitive (galois.BCH property), 175
is_primitive() (in module galois), 125
is_primitive_element() (in module galois), 129
is_primitive_poly (galois.FieldClass property), 99
is_primitive_root() (in module galois), 116
is_smooth() (in module galois), 218
is_square_free() (in module galois), 165
isqrt() (in module galois), 208
J
jacobi_symbol() (in module galois), 204
K
k (galois.BCH property), 175
k (galois.ReedSolomon property), 187
kronecker_symbol() (in module galois), 205
kth_prime() (in module galois), 220
L
lcm() (in module galois), 153
legendre_symbol() (in module galois), 203
LFSR (class in galois), 193
log() (in module np), 235
lu_decompose() (galois.FieldArray method), 80
lup_decompose() (galois.FieldArray method), 81
M
matlab_primitive_poly() (in module galois), 124
matmul() (in module np), 236
matrix_power() (in module np.linalg), 241
matrix_rank() (in module np.linalg), 243
mersenne_exponents() (in module galois), 222
mersenne_primes() (in module galois), 223
miller_rabin_primality_test() (in module galois), 226
minimal_poly() (in module galois), 130
module
galois, 61
multiply() (in module np), 231
N
n (galois.BCH property), 175
n (galois.ReedSolomon property), 187
name (galois.FieldClass property), 99
negative() (in module np), 232
next_prime() (in module galois), 221
nonzero_coeffs (galois.Poly property), 148
nonzero_degrees (galois.Poly property), 149
O
One() (galois.Poly class method), 136