## Roadmap

1. Getting Started ................................................. 7
2. Array Classes .................................................. 10
3. Compilation Modes ............................................. 12
4. Element Representation ........................................ 14
5. Array Creation .................................................. 20
6. Array Arithmetic ................................................ 32
7. Polynomials ..................................................... 46
8. Polynomial Arithmetic .......................................... 50
9. Intro to Prime Fields ........................................... 55
10. Intro to Extension Fields ....................................... 65
11. Prime Fields ................................................... 81
12. Binary Extension Fields ........................................ 84
13. Benchmarks .................................................... 87
14. Installation ..................................................... 91
15. Linter ........................................................ 92
16. Unit Tests ..................................................... 93
17. Documentation .................................................. 95
18. galois ........................................................ 95
19. Versioning ..................................................... 320
20. v0.0.31 ..................................................... 321
21. v0.0.30 ..................................................... 322
22. v0.0.29 ..................................................... 323
23. v0.0.28 ..................................................... 325
24. v0.0.27 ..................................................... 326
25. v0.0.26 ..................................................... 328
26. v0.0.25 ..................................................... 331
27. v0.0.24 ..................................................... 333
28. v0.0.23 ..................................................... 334
29. v0.0.22 ..................................................... 335
30. v0.0.21 ..................................................... 336
31. v0.0.20 ..................................................... 336
32. v0.0.19 ..................................................... 337
33. v0.0.18 ..................................................... 337
34. v0.0.17 ..................................................... 339
35. v0.0.16 ..................................................... 340
The `galois` library is a Python 3 package that extends NumPy arrays to operate over finite fields.

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

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

**Disclaimer**

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

- Supports all Galois fields $GF(p^n)$, even arbitrarily-large fields!
- **Faster** than native NumPy! $GF(x) \ast GF(y)$ is faster than $(x \ast y) \mod p$ for $GF(p)$.
- Seamless integration with NumPy – normal NumPy functions work on `FieldArray` instances.
- Linear algebra over finite fields using normal `numpy.linalg` functions.
- Linear transforms over finite fields, such as the FFT with `numpy.fft.fft()` and the NTT with `ntt()`.
- Functions to generate irreducible, primitive, and Conway polynomials.
- Univariate polynomials over finite fields with `Poly`.
- Forward error correction codes with `BCH` and `ReedSolomon`.
- Fibonacci and Galois linear-feedback shift registers over any finite field with `FLFSR` and `GLFSR`.
- Various number theoretic functions.
- Integer factorization and accompanying algorithms.
- Prime number generation and primality testing.
CHAPTER
ONE

ROADMAP

• Elliptic curves over finite fields
• Galois ring arrays
• GPU support
ACKNOWLEDGEMENTS

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

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

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

This library would not be possible without all of the other libraries mentioned. Thank you to all their developers!
If this library was useful to you in your research, please cite us. Following the GitHub citation standards, here is the recommended citation.

**BibTeX**

```bibtex
@software{Hostetter_Galois_2020,
    title = {{Galois: A performant NumPy extension for Galois fields}},
    author = {Hostetter, Matt},
    month = {11},
    year = {2020},
    url = {https://github.com/mhostetter/galois},
}
```

**APA**


3.1 Getting Started

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

3.1.1 Install the package

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

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

Import the *galois* package in Python.

```
In [1]: import galois

In [2]: galois.__version__
Out[2]: '0.0.31'
```
3.1.2 Create a FieldArray subclass

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

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

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

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

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

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

See `Array Classes` for more details.

3.1.3 Create two FieldArray instances

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

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

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

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

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

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

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

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

See `Array Creation` for more details.
3.1.4 Change the element representation

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

Set the display mode by passing the display keyword argument to $GF()$ or by calling the display() classmethod. Choose whichever element representation is most convenient for you.

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

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

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

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

See Element Representation for more details.

3.1.5 Perform array arithmetic

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

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

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

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

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

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

More complicated arithmetic, like square root and logarithm base $\alpha$, are also supported.

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

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

See Array Arithmetic for more details.
### 3.2 Array Classes

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

#### 3.2.1 Array subclasses

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

#### 3.2.2 FieldArray subclasses

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

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

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

```
In [3]: issubclass(GF, galois.FieldArray)
Out[3]: True
In [4]: issubclass(GF, galois.Array)
Out[4]: True
In [5]: issubclass(GF, np.ndarray)
Out[5]: True
```

**Class singletons**

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

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

```
In [6]: galois.GF(3**5) is galois.GF(3**5)
Out[6]: True
```

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

However, the field $GF(3^5)$ defined with irreducible polynomial $x^5 + x^2 + x + 2$, while isomorphic to the first field, has different arithmetic. As such, $GF()$ returns a unique *FieldArray* subclass.
In [7]: galois.GF(3**5) is galois.GF(3**5, irreducible_poly="x^5 + x^2 + x + 2")
Out[7]: False

Methods and properties

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

In [8]: GF.irreducible_poly
Out[8]: Poly(x^5 + 2x + 1, GF(3))

3.2.3 FieldArray instances

A FieldArray instance is created using GF’s constructor.

In [9]: x = GF([23, 78, 163, 124])
In [10]: x
Out[10]: GF([ 23, 78, 163, 124], order=3^5)

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

In [11]: isinstance(x, GF)
Out[11]: True
In [12]: isinstance(x, galois.FieldArray)
Out[12]: True
In [13]: isinstance(x, galois.Array)
Out[13]: True
In [14]: isinstance(x, np.ndarray)
Out[14]: True

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

In [15]: type(x) is GF
Out[15]: True

Constructors

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

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

In [16]: GF.Random((2, 3))
Out[16]: GF([[155, 116, 46],
       [ 72, 114, 230]], order=3^5)
Or, create an identity matrix using \texttt{Identity()}.

\begin{verbatim}
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=3^5)
\end{verbatim}

**Methods**

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

\begin{verbatim}
In [18]: x.multiplicative_order()
Out[18]: array([242, 11, 242, 242])
\end{verbatim}

### 3.3 Compilation Modes

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

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

\begin{verbatim}
In [1]: GF = galois.GF(3**5)
In [2]: GF.ufunc_modes
Out[2]: ['jit-lookup', 'jit-calculate']
\end{verbatim}

Large finite fields, which have \texttt{numpy.object} data types, use "python-calculate" which utilizes non-compiled, pure-Python ufuncs.

\begin{verbatim}
In [3]: GF = galois.GF(2**100)
In [4]: GF.ufunc_modes
Out[4]: ['python-calculate']
\end{verbatim}

#### 3.3.1 Lookup tables

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

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

\[ x = \alpha^i \]

\[ \log_\alpha(x) = i \]
With these lookup tables, many arithmetic operations are simplified. For instance, multiplication of two finite field elements is reduced to three lookups and one integer addition.

\[ x \cdot y = \alpha^m \cdot \alpha^n = \alpha^{m+n} \]

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

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

\[ Z(i) = \log_{\alpha}(1 + \alpha^i) \]

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

\[ x + y = \alpha^m + \alpha^n = \alpha^m(1 + \alpha^{n-m}) = \alpha^m \cdot \alpha^{Z(n-m)} = \alpha^{m+Z(n-m)} \]

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

```python
In [5]: GF = galois.GF(3**5)
In [6]: GF.ufunc_mode
Out[6]: 'jit-lookup'
```

### 3.3.2 Explicit calculation

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

```python
In [7]: GF = galois.GF(2**24)
In [8]: GF.ufunc_mode
Out[8]: 'jit-calculate'
```

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

```python
In [9]: GF = galois.GF(2**24, compile="jit-lookup")
In [10]: GF.ufunc_mode
Out[10]: 'jit-lookup'
```
3.3.3 Python explicit calculation

Large finite fields cannot use JIT compiled ufuncs. This is because they cannot use NumPy integer data types. This is either because the order of the field or an intermediate arithmetic result is larger than the max value of `numpy.int64`. These finite fields use the `numpy.object_` data type and have ufunc compilation mode "python-calculate". This mode does not compile the Python functions, but rather converts them into Python ufuncs using `numpy.frompyfunc()`. The lack of JIT compilation allows the ufuncs to operate on Python integers, which have unlimited size. This does come with a performance penalty, however.

```
In [11]: GF = galois.GF(2**100)
In [12]: GF.ufunc_mode
Out[12]: 'python-calculate'
```

3.3.4 Recompile the ufuncs

The compilation mode may be explicitly set during creation of the `FieldArray` subclass using the `compile` keyword argument to `GF()`. Here, the `FieldArray` subclass for GF(3\(^5\)) would normally select "jit-lookup" as its default compilation mode. However, we can intentionally choose explicit calculation.

```
In [13]: GF = galois.GF(3**5, compile="jit-calculate")
In [14]: GF.ufunc_mode
Out[14]: 'jit-calculate'
```

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

```
In [15]: GF.compile("jit-lookup")
In [16]: GF.ufunc_mode
Out[16]: 'jit-lookup'
```

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

3.4 Element Representation

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

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

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

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

```python
In [1]: GF = galois.GF(3**5, display="poly")
In [2]: x = GF([17, 4])
In [3]: x
Out[3]: GF([^2 + 2 + 2, + 1], order=3^5)
In [4]: print(x)
[^2 + 2 + 2, + 1]
```

---

**Note:** Notice `__repr__()` displays `GF([...], order=p^m)` where `__str__()` only displays ` [... ]`. This is designed to be consistent with NumPy's use of `repr()` and `str()`.

The current display mode is accessed with the `display_mode` class property.

```python
In [5]: GF.display_mode
Out[5]: 'poly'
```

The display mode can be temporarily changed using the `display()` classmethod as a context manager.

```python
# Inside the context manager, x prints using the power representation
In [6]: with GF.display("power"):
   ...:     print(x)
   ...:
[^222, ^69]
# Outside the context manager, x prints using the previous representation
In [7]: print(x)
[^2 + 2 + 2, + 1]
```

The display mode can be permanently changed using the `display()` method.

```python
# The old polynomial display mode
In [8]: x
Out[8]: GF([^2 + 2 + 2, + 1], order=3^5)
In [9]: GF.display("int");
# The new integer display mode
In [10]: x
Out[10]: GF([17, 4], order=3^5)
```

---

3.4. Element Representation
3.4.2 Integer representation

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

In prime fields, the integer representation is simply the integer element in \(\{0, 1, \ldots, p - 1\}\).

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

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

```
In [13]: GF = galois.GF(3**5)
In [14]: GF(17)
Out[14]: GF(17, order=3^5)
In [15]: GF("^2 + 2 + 2")
Out[15]: GF(17, order=3^5)
```

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

3.4.3 Polynomial representation

The polynomial display mode displays all finite field elements as polynomials over their prime subfield with degree less than \(m\).

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

```
In [17]: GF = galois.GF(31, display="poly")
In [18]: GF(11)
Out[18]: GF(11, order=31)
```

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

```
In [19]: GF = galois.GF(3**5, display="poly")
In [20]: GF(17)
Out[20]: GF(^2 + 2 + 2, order=3^5)
In [21]: GF("^2 + 2 + 2")
Out[21]: GF(^2 + 2 + 2, order=3^5)
```

# Integer/polynomial equivalence
```
In [22]: p = 3; p**2 + 2*p + 2 == 17
Out[22]: True
```
Tip: Use `set_printoptions()` to display the polynomial coefficients in degree-ascending order. Use `numpy.set_printoptions()` to increase the line width to display large arrays more clearly. See NumPy print options for more details.

### 3.4.4 Power representation

The power display mode represents the elements as powers of the finite field’s primitive element $\alpha$.

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

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

```
In [23]: GF = galois.GF(31, display="power")
In [24]: GF(11)
Out[24]: GF(11, order=31)

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

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

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

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

```
In [28]: GF = galois.GF(3**5, display="power")
In [29]: GF(17)
Out[29]: GF(17, order=3^5)

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

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

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

The vector representation, while not a proper display mode of `display()`, represents finite field elements as vectors of their polynomial coefficients.

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

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

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

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

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

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

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

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

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

3.4.6 NumPy print options

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

For example, below is a 5 × 5 matrix over GF(3⁵) displayed in the polynomial representation. With the default line width, the array is quite difficult to read.

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

In [40]: x = GF.Random((5, 5)); x
Out[40]: GF([[^4 + 2^3 + 2^2 + 2 + 1, ^4 + 2^3 + 2^2 + 1, ^4 + 2^3 + 2 + 1, ^4 + 2^3 + 2 + 1, ^4 + 2^3 + 2 + 1],
                [2^3 + ^2, 2^3 + ^2, 2^3 + ^2, 2^3 + ^2, 2^3 + ^2],
                [2^3 + ^2 + + 2, 2^3 + ^2 + + 2, 2^3 + ^2 + + 2, 2^3 + ^2 + + 2, 2^3 + ^2 + + 2],
                [2^3 + ^2, 2^3 + ^2, 2^3 + ^2, 2^3 + ^2, 2^3 + ^2],
                [2^3 + ^2, 2^3 + ^2, 2^3 + ^2, 2^3 + ^2, 2^3 + ^2]])
```

(continues on next page)
^4 + 2^3 + ^2 + 2, 2^4,
^4 + 2^3],
[ 2^4 + 2^2, 2^4 + 2^3 + ^2 + + 1,
2^4 + ^3 + , 2^3 + 2^2 + 2 + 2,
^3 + ^2 + 1], order=3^5)

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

3.4.7 Representation comparisons

For any finite field, each of the four representations can be easily compared using the `repr_table()` classmethod.

```
In [43]: GF = galois.GF(3**3)
In [44]: 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]</td>
<td>0</td>
</tr>
<tr>
<td>x^1</td>
<td>x</td>
<td>[0, 1, 0]</td>
<td>3</td>
</tr>
<tr>
<td>x^2</td>
<td>x^2</td>
<td>[1, 0, 0]</td>
<td>9</td>
</tr>
<tr>
<td>x^3</td>
<td>x + 2</td>
<td>[0, 1, 2]</td>
<td>5</td>
</tr>
<tr>
<td>x^4</td>
<td>x^2 + 2x</td>
<td>[1, 2, 0]</td>
<td>15</td>
</tr>
<tr>
<td>x^5</td>
<td>2x^2 + x + 2</td>
<td>[2, 1, 2]</td>
<td>23</td>
</tr>
<tr>
<td>x^6</td>
<td>x^2 + x + 1</td>
<td>[1, 1, 1]</td>
<td>13</td>
</tr>
<tr>
<td>x^7</td>
<td>x^2 + 2x + 2</td>
<td>[1, 2, 2]</td>
<td>17</td>
</tr>
<tr>
<td>x^8</td>
<td>2x^2 + 2</td>
<td>[2, 0, 2]</td>
<td>20</td>
</tr>
<tr>
<td>x^9</td>
<td>x + 1</td>
<td>[0, 1, 1]</td>
<td>4</td>
</tr>
<tr>
<td>x^10</td>
<td>x^2 + x</td>
<td>[1, 1, 0]</td>
<td>12</td>
</tr>
<tr>
<td>x^11</td>
<td>x^2 + x + 2</td>
<td>[1, 1, 2]</td>
<td>14</td>
</tr>
<tr>
<td>x^12</td>
<td>x^2 + 2</td>
<td>[1, 0, 2]</td>
<td>11</td>
</tr>
<tr>
<td>x^13</td>
<td>2</td>
<td>[0, 0, 2]</td>
<td>2</td>
</tr>
<tr>
<td>x^14</td>
<td>2x</td>
<td>[0, 2, 0]</td>
<td>6</td>
</tr>
<tr>
<td>x^15</td>
<td>2x^2</td>
<td>[2, 0, 0]</td>
<td>18</td>
</tr>
<tr>
<td>x^16</td>
<td>2x + 1</td>
<td>[0, 2, 1]</td>
<td>7</td>
</tr>
</tbody>
</table>
```
3.5 Array Creation

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

**Integer**

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

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

**Polynomial**

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

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

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

### 3.5.1 Create a scalar

A single finite field element (a scalar) is a 0-D `FieldArray`. They are created by passing a single `ElementLike` object to `GF`'s constructor.

**Integer**

```
In [7]: a = GF(17); a
Out[7]: GF(17, order=3^5)

In [8]: a = GF("x^2 + 2x + 2"); a
Out[8]: GF(17, order=3^5)

In [9]: a.ndim
Out[9]: 0
```

**Polynomial**

```
In [10]: a = GF(17); a
Out[10]: GF(x^2 + 2x + 2, order=3^5)

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

In [12]: a.ndim
Out[12]: 0
```
Power

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

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

In [15]: a.ndim
Out[15]: 0
```

3.5.2 Create a new array

Array-Like objects

A `FieldArray` can be created from various `ArrayLike` objects.

Integer

```
In [16]: GF([17, 4, 148, 205])
Out[16]: GF([17, 4, 148, 205], order=3^5)

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

Polynomial

```
In [18]: GF([17, 4, 148, 205])
Out[18]:
  GF([\^2 + 2 + 2, \^4 + 2^3 + \^2 + + 1, 2^4 + \^3 + \^2 + 2 + 1], order=3^5)

In [19]: GF([["x^2 + 2x + 2", 4], ["x^4 + 2x^3 + x^2 + x + 1", 205]])
Out[19]:
  GF([[\^2 + 2 + 2, \^4 + 2^3 + \^2 + + 1, 2^4 + \^3 + \^2 + 2 + 1]], order=3^5)
```
Power

```
In [20]: GF([17, 4, 148, 205])
Out[20]: GF([^222, ^69, ^54, ^24], order=3^5)

In [21]: GF([["x^2 + 2x + 2", 4], ["x^4 + 2x^3 + x^2 + x + 1", 205]])
Out[21]:
GF([[^222, ^69],
    [ ^54, ^24]], order=3^5)
```

Polynomial coefficients

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

Integer

```
In [22]: GF.Vector([[0, 0, 1, 2, 2], [0, 0, 0, 1, 1]])
Out[22]: GF([17, 4], order=3^5)
```

Polynomial

```
In [23]: GF.Vector([[0, 0, 1, 2, 2], [0, 0, 0, 1, 1]])
Out[23]: GF([x^2 + 2 + 2, + 1], order=3^5)
```

Power

```
In [24]: GF.Vector([[0, 0, 1, 2, 2], [0, 0, 0, 1, 1]])
Out[24]: GF([^222, ^69], order=3^5)
```

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

Integer

```
In [25]: GF([17, 4]).vector()
Out[25]:
GF([[0, 0, 1, 2, 2],
    [0, 0, 0, 1, 1]], order=3)
```
Galois

Polynomial

```
In [26]: GF([17, 4]).vector()
Out[26]:
GF([[0, 0, 1, 2, 2],
    [0, 0, 0, 1, 1]], order=3)
```

Power

```
In [27]: GF([17, 4]).vector()
Out[27]:
GF([[0, 0, 1, 2, 2],
    [0, 0, 0, 1, 1]], order=3)
```

Primitive element powers

A `FieldArray` can also be created from the powers of a primitive element $\alpha$.

Integer

```
In [28]: alpha = GF.primitive_element; alpha
Out[28]: GF(3, order=3^5)

In [29]: powers = np.array([222, 69, 54, 24]); powers
Out[29]: array([222, 69, 54, 24])

In [30]: alpha ** powers
Out[30]: GF([ 17, 4, 148, 205], order=3^5)
```

Polynomial

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

In [32]: powers = np.array([222, 69, 54, 24]); powers
Out[32]: array([222, 69, 54, 24])

In [33]: alpha ** powers
Out[33]: GF([ ^2 + 2 + 2, + 1,
    ^4 + 2^3 + ^2 + + 1, 2^4 + ^3 + ^2 + 2 + 1], order=3^5)
```
Power

```
In [34]: alpha = GF.primitive_element; alpha
Out[34]: GF(, order=3^5)

In [35]: powers = np.array([222, 69, 54, 24]); powers
Out[35]: array([222, 69, 54, 24])

In [36]: alpha ** powers
Out[36]: GF([^222, ^69, ^54, ^24], order=3^5)
```

NumPy array

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

Integer

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

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

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

Polynomial

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

In [41]: x = GF(x_np); x
Out[41]: GF([2^4 + ^3 + 2^2 + 2, 2^4 + ^3 + 2^2 + 2 + 1, 2^4 + ^3 + 2^2 + 2], order=3^5)

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

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

In [44]: x = GF(x_np); x
Out[44]: GF([183, 9, 69, 153, 58], order=3^5)

# Modifying x does not modify x_np
In [45]: x[0] = 0; x_np
Out[45]: array([213, 167, 4, 214, 209])
```

3.5.3 View an existing array

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

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

Integer

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

In [47]: x = x_np.view(GF); x
Out[47]: GF([213, 167, 4, 214, 209], order=3^5)

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

Polynomial

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

In [50]: x = x_np.view(GF); x
Out[50]: GF([2^4 + 3 + 2^2 + 2, 2^4 + 2, 2^4 + 1, 2^4 + 3 + 2^2 + 2 + 1, 2^4 + 3 + 2^2 + 2], order=3^5)

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

```python
In [52]: x_np = np.array([213, 167, 4, 214, 209], dtype=int); x_np
Out[52]: array([213, 167,  4, 214, 209])

In [53]: x = x_np.view(GF); x
Out[53]: GF([183,  9, 69, 153, 58], order=3^5)

# Modifying x does modify x_np!
In [54]: x[0] = 0; x_np
Out[54]: array([ 0, 167,  4, 214, 209])
```

3.5.4 Classmethods

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

Constant arrays

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

Integer

```python
In [55]: GF.Zeros(4)
Out[55]: GF([0, 0, 0, 0], order=3^5)

In [56]: GF.Ones(4)
Out[56]: GF([1, 1, 1, 1], order=3^5)
```

Polynomial

```python
In [57]: GF.Zeros(4)
Out[57]: GF([0, 0, 0, 0], order=3^5)

In [58]: GF.Ones(4)
Out[58]: GF([1, 1, 1, 1], order=3^5)
```

Power

```python
In [59]: GF.Zeros(4)
Out[59]: GF([0, 0, 0, 0], order=3^5)

In [60]: GF.Ones(4)
Out[60]: GF([1, 1, 1, 1], order=3^5)
```

Note: There is no numpy.empty() equivalent. This is because FieldArray instances must have values in $[0,p^m)$. Empty NumPy arrays have whatever values are currently in memory, and therefore would fail those bounds checks.

3.5. Array Creation 27
Ordered arrays

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

**Integer**

```
In [61]: GF.Range(10, 20)
Out[61]: GF([10, 11, 12, 13, 14, 15, 16, 17, 18, 19], order=3^5)

In [62]: GF.Range(10, 20, 2)
Out[62]: GF([10, 12, 14, 16, 18], order=3^5)
```

**Polynomial**

```
In [63]: GF.Range(10, 20)
Out[63]: GF([^2 + 1, ^2 + 2, ^2 + 1, ^2 + 2, ^2 + 2, ^2 + 2 + 1, ^2 + 2 + 2, 2^2, 2^2 + 1],
            order=3^5)

In [64]: GF.Range(10, 20, 2)
Out[64]: GF([^2 + 1, ^2 + 2, ^2 + 2 + 2, ^2 + 2 + 1, 2^2],
            order=3^5)
```

**Power**

```
In [65]: GF.Range(10, 20)
Out[65]: GF([^46, ^74, ^70, ^10, ^209, ^6, ^138, ^222, ^123, ^195],
            order=3^5)

In [66]: GF.Range(10, 20, 2)
Out[66]: GF([^46, ^70, ^209, ^138, ^123], order=3^5)
```

Random arrays

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

```python
In [67]: GF.Random(4, seed=1234)
Out[67]: GF([176, 177, 172, 237], order=3^5)
```

```python
In [68]: GF.Random(4, low=10, high=20, seed=5678)
Out[68]: GF([11, 18, 12, 10], order=3^5)
```

**Polynomial**

```python
In [69]: GF.Random(4, seed=1234)
Out[69]: GF([2^4 + 2^2 + + 2, 2^4 + 2^2 + 2, 
                 2^4 + 2^2 + 1, 2^4 + 2^3 + 2^2 + ], order=3^5)
```

```python
In [70]: GF.Random(4, low=10, high=20, seed=5678)
Out[70]: GF([2^2 + 2, 2^2, 2^2 + , 2^2 + 1], order=3^5)
```

**Power**

```python
In [71]: GF.Random(4, seed=1234)
Out[71]: GF([114, 78, 206, 96], order=3^2)
```

```python
In [72]: GF.Random(4, low=10, high=20, seed=5678)
Out[72]: GF([74, 123, 70, 46], order=3^5)
```

### 3.5.5 Class properties

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

**Integer**

```python
In [73]: GF = galois.GF(3**2)

In [74]: GF.elements
Out[74]: GF([0, 1, 2, 3, 4, 5, 6, 7, 8], order=3^2)

In [75]: GF.units
Out[75]: GF([1, 2, 3, 4, 5, 6, 7, 8], order=3^2)

In [76]: GF.primitive_elements
Out[76]: GF([3, 5, 6, 7], order=3^2)

In [77]: GF.quadratic_residues
Out[77]: GF([0, 1, 2, 4, 8], order=3^2)
```
Polynomial

```
In [78]: GF = galois.GF(3**2, display="poly")

In [79]: GF.elements
Out[79]:
GF([ 0, 1, 2, , + 1, + 2, 2, 2 + 1, 2 + 2], order=3^2)

In [80]: GF.units
Out[80]:
GF([ 1, 2, , + 1, + 2, 2, 2 + 1, 2 + 2], order=3^2)

In [81]: GF.primitive_elements
Out[81]: GF([ , + 2, 2, 2 + 1], order=3^2)

In [82]: GF.quadratic_residues
Out[82]: GF([ 0, 1, + 1, 2 + 2], order=3^2)
```

Power

```
In [83]: GF = galois.GF(3**2, display="power")

In [84]: GF.elements
Out[84]: GF([ 0, 1, 2, , ^4, ^2, ^7, ^5, ^3, ^6], order=3^2)

In [85]: GF.units
Out[85]: GF([ 1, ^4, , ^2, ^7, ^5, ^3, ^6], order=3^2)

In [86]: GF.primitive_elements
Out[86]: GF([ ^7, ^5, ^3], order=3^2)

In [87]: GF.quadratic_residues
Out[87]: GF([ 0, 1, ^4, ^2, ^6], order=3^2)
```
### 3.5.6 Data types

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

#### Valid data types

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

```python
In [88]: GF = galois.GF(2**4)
In [89]: GF.dtypes
Out[89]: [numpy.uint8, numpy.uint16, numpy.uint32, numpy.int8, numpy.int16, numpy.int32, numpy.int64]
```

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

```python
In [90]: GF = galois.GF(2**10)
In [91]: GF.dtypes
Out[91]: [numpy.uint16, numpy.uint32, numpy.int16, numpy.int32, numpy.int64]
```

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

```python
In [92]: GF = galois.GF(2**100)
In [93]: GF.dtypes
Out[93]: [numpy.object_]
```

#### Default data type

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

```python
In [94]: GF = galois.GF(2**10)
In [95]: GF.dtypes
Out[95]: [numpy.uint16, numpy.uint32, numpy.int16, numpy.int32, numpy.int64]

In [96]: x = GF.Random(4); x
Out[96]: GF([720, 136, 864, 945], order=2^10)

In [97]: x.dtype
Out[97]: dtype('uint16')
```
In [98]: GF = galois.GF(2**100)

In [99]: GF.dtypes
Out[99]: [numpy.object_

In [100]: x = GF.Random(4); x
Out[100]: GF([ 745408149171652856989398618398, 124985248415171096458661289532, 324148594782345344794031417768, 1252336439199145065212215187911], order=2^100)

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

Changing data types

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

In [102]: GF = galois.GF(2**10)

In [103]: x = GF([273, 388, 124, 400], dtype=np.uint32); x
Out[103]: GF([273, 388, 124, 400], order=2^10)

In [104]: x.dtype
Out[104]: dtype('uint32')

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

In [105]: x.dtype
Out[105]: dtype('uint32')

In [106]: x = x.astype(np.int64)

In [107]: x.dtype
Out[107]: dtype('int64')

3.6 Array Arithmetic

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

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

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

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

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

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

Expand any section for more details.

Addition: \( x + y == \text{np.add}(x, y) \)

```
In [4]: x + y  
Out[4]: GF([ 81,  7, 215, 58], order=3^5)

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

Additive inverse: \(-x == \text{np.negative}(x)\)

```
In [6]: -x  
Out[6]: GF([ 98, 14, 206, 62], order=3^5)

In [7]: np.negative(x)  
Out[7]: GF([ 98, 14, 206, 62], order=3^5)
```

Any array added to its additive inverse results in zero.

```
In [8]: x + np.negative(x)  
Out[8]: GF([0, 0, 0, 0], order=3^5)
```

Subtraction: \( x - y == \text{np.subtract}(x, y) \)

```
In [9]: x - y  
Out[9]: GF([17, 16, 18, 4], order=3^5)

In [10]: np.subtract(x, y)  
Out[10]: GF([17, 16, 18, 4], order=3^5)
```

Multiplication: \( x \ast y == \text{np.multiply}(x, y) \)

```
In [11]: x \ast y  
Out[11]: GF([ 41, 225, 106, 123], order=3^5)

In [12]: np.multiply(x, y)  
Out[12]: GF([ 41, 225, 106, 123], order=3^5)
```

Scalar multiplication: \( x \ast z == \text{np.multiply}(x, z) \)

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

```
In [13]: x \ast 4  
Out[13]: GF([184, 25, 157, 31], order=3^5)

In [14]: np.multiply(x, 4)  
Out[14]: GF([184, 25, 157, 31], order=3^5)
```
In finite fields $GF(p^m)$, the characteristic $p$ is the smallest value when multiplied by any non-zero field element that results in 0.

```python
In [16]: p = GF.characteristic; p
Out[16]: 3
```

```python
In [17]: x * p
Out[17]: GF([0, 0, 0, 0], order=3^5)
```

Multiplicative inverse: $y^{-1} = \text{np.reciprocal}(y)$

```python
In [18]: y ** -1 == np.reciprocal(y)
Out[18]: True
```

```python
In [19]: GF(1) / y
Out[19]: GF([71, 217, 213, 235], order=3^5)
```

```python
In [20]: np.reciprocal(y)
Out[20]: GF([71, 217, 213, 235], order=3^5)
```

Any array multiplied by its multiplicative inverse results in one.

```python
In [21]: y * np.reciprocal(y)
Out[21]: GF([1, 1, 1, 1], order=3^5)
```

Division: $x / y = x \div y = \text{np.divide}(x, y)$

```python
In [22]: x / y
Out[22]: GF([237, 56, 122, 126], order=3^5)
```

```python
In [23]: x // y
Out[23]: GF([237, 56, 122, 126], order=3^5)
```

```python
In [24]: np.divide(x, y)
Out[24]: GF([237, 56, 122, 126], order=3^5)
```

Remainder: $x \% y = \text{np.remainder}(x, y)$

```python
In [25]: x % y
Out[25]: GF([0, 0, 0, 0], order=3^5)
```

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

Divmod: $\text{divmod}(x, y) = \text{np.divmod}(x, y)$

```python
In [27]: x / y, x % y
Out[27]: (GF([237, 56, 122, 126], order=3^5), GF([0, 0, 0, 0], order=3^5))
```

```python
In [28]: divmod(x, y)
```
3.6. Array Arithmetic

Out[28]: (GF([237, 56, 122, 126], order=3^5), GF([0, 0, 0, 0], order=3^5))

In [29]: np.divmod(x, y)
Out[29]: (GF([237, 56, 122, 126], order=3^5), GF([0, 0, 0, 0], order=3^5))

In [30]: q, r = divmod(x, y)
In [31]: q*y + r == x
Out[31]: array([True, True, True, True])

Exponentiation: x ** z == np.power(x, z)

In [32]: x ** 3
Out[32]: GF([175, 76, 218, 192], order=3^5)
In [33]: np.power(x, 3)
Out[33]: GF([175, 76, 218, 192], order=3^5)
In [34]: x * x * x
Out[34]: GF([175, 76, 218, 192], order=3^5)

Square root: np.sqrt(x)

# Ensure the elements of x have square roots
In [35]: x.is_quadratic_residue()
Out[35]: array([True, True, True, True])

In [36]: z = np.sqrt(x); z
Out[36]: GF([102, 109, 14, 111], order=3^5)
In [37]: z ** 2 == x
Out[37]: array([True, True, True, True])

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

Compute the logarithm base \( \alpha \), the primitive element of the field.

In [38]: z = np.log(y); z
Out[38]: array([60, 2, 59, 3])
In [39]: alpha = GF.primitive_element; alpha
Out[39]: GF(3, order=3^5)
In [40]: alpha ** z == y
Out[40]: array([True, True, True, True])

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

In [41]: beta = GF.primitive_elements[-1]; beta
Out[41]: GF(242, order=3^5)
In [42]: z = y.log(beta); z
Out[42]: array([190, 208, 207, 191])

(continues on next page)
3.6.2 Ufunc methods

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

Expand any section for more details.

reduce()

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

```python
In [44]: np.add.reduce(x)
Out[44]: GF(7, order=3^5)

Out[45]: GF(7, order=3^5)
```

```python
In [46]: np.multiply.reduce(x)
Out[46]: GF(105, order=3^5)

Out[47]: GF(105, order=3^5)
```

accumulate()

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

```python
In [48]: np.add.accumulate(x)
Out[48]: GF([184, 173, 57, 7], order=3^5)

In [49]: GF([x[0], x[0] + x[1], x[0] + x[1] + x[2], x[0] + x[1] + x[2] + x[3]])
Out[49]: GF([184, 173, 57, 7], order=3^5)
```

```python
In [50]: np.multiply.accumulate(x)
Out[50]: GF([184, 9, 211, 105], order=3^5)

In [51]: GF([x[0], x[0] * x[1], x[0] * x[1] * x[2], x[0] * x[1] * x[2] * x[3]])
Out[51]: GF([184, 9, 211, 105], order=3^5)
```

reduceat()

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

```python
In [52]: np.add.reduceat(x, [0, 3])
Out[52]: GF([57, 31], order=3^5)

In [53]: GF([x[0] + x[1] + x[2], x[3]])
Out[53]: GF([57, 31], order=3^5)
```
In [54]: np.multiply.reduceat(x, [0, 3])
Out[54]: GF([211, 31], order=3^5)

In [55]: GF([x[0] * x[1] * x[2], x[3]])
Out[55]: GF([211, 31], order=3^5)

outer()

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

In [56]: np.add.outer(x, y)
Out[56]: GF([[ 81, 166, 80, 211],
       [165, 7, 155, 52],
       [ 54, 139, 215, 103],
       [198, 40, 89, 58]], order=3^5)

In [57]: np.multiply.outer(x, y)
Out[57]: GF([[ 41, 192, 93, 97],
       [ 91, 225, 193, 196],
       [ 80, 211, 106, 145],
       [149, 41, 129, 123]], order=3^5)

at()

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

In [58]: z = x.copy()

# Negate indices 0 and 1 in-place
In [59]: np.negative.at(x, [0, 1]); x
Out[59]: GF([ 98, 14, 157, 31], order=3^5)

In [60]: z[0:1] *= -1; z
Out[60]: GF([ 98, 25, 157, 31], order=3^5)

3.6.3 Advanced arithmetic

Convolution: np.convolve(x, y)

In [61]: np.convolve(x, y)
Out[61]: GF([ 79, 80, 5, 79, 167, 166, 123], order=3^5)

FFT: np.fft.fft(x)

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

In [62]: GF = galois.GF(7**5)

In [63]: n = 6

(continues on next page)
3.6.4 Linear algebra

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

Expand any section for more details.

Dot product: np.dot(a, b)

In [74]: GF = galois.GF(31)

In [75]: a = GF([29, 0, 2, 21]); a
Out[75]: GF([29, 0, 2, 21], order=31)
In [76]: b = GF([23, 5, 15, 12]); b
Out[76]: GF([23, 5, 15, 12], order=31)

In [77]: np.dot(a, b)
Out[77]: GF(19, order=31)

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

In [78]: GF = galois.GF(31)
In [79]: a = GF([29, 0, 2, 21]); a
Out[79]: GF([29, 0, 2, 21], order=31)
In [80]: b = GF([23, 5, 15, 12]); b
Out[80]: GF([23, 5, 15, 12], order=31)
In [81]: np.vdot(a, b)
Out[81]: GF(19, order=31)

Inner product: np.inner(a, b)

In [82]: GF = galois.GF(31)
In [83]: a = GF([29, 0, 2, 21]); a
Out[83]: GF([29, 0, 2, 21], order=31)
In [84]: b = GF([23, 5, 15, 12]); b
Out[84]: GF([23, 5, 15, 12], order=31)
In [85]: np.inner(a, b)
Out[85]: GF(19, order=31)

Outer product: np.outer(a, b)

In [86]: GF = galois.GF(31)
In [87]: a = GF([29, 0, 2, 21]); a
Out[87]: GF([29, 0, 2, 21], order=31)
In [88]: b = GF([23, 5, 15, 12]); b
Out[88]: GF([23, 5, 15, 12], order=31)
In [89]: np.outer(a, b)
Out[89]: GF([[16, 21, 1, 7],
        [0, 0, 0, 0],
        [15, 10, 30, 24],
        [18, 12, 5, 4]], order=31)

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

In [90]: GF = galois.GF(31)
In [91]: A = GF([[17, 25, 18, 8], [7, 9, 21, 15], [6, 16, 6, 30]]); A
Out[91]:
GF([[17, 25, 18, 8],
    [ 7, 9, 21, 15],
    [ 6, 16, 6, 30]], order=31)

In [92]: B = GF([[8, 18], [22, 0], [7, 8], [20, 24]]); B
Out[92]:
GF([[ 8, 18],
    [22, 0],
    [ 7, 8],
    [20, 24]], order=31)

In [93]: A @ B
Out[93]:
GF([[11, 22],
    [19, 3],
    [19, 8]], order=31)

In [94]: np.matmul(A, B)
Out[94]:
GF([[11, 22],
    [19, 3],
    [19, 8]], order=31)

Matrix exponentiation: np.linalg.matrix_power(A, z)

In [95]: GF = galois.GF(31)

In [96]: A = GF([[14, 1, 5], [3, 23, 6], [24, 27, 4]]); A
Out[96]:
GF([[14, 1, 5],
    [ 3, 23, 6],
    [24, 27, 4]], order=31)

In [97]: np.linalg.matrix_power(A, 3)
Out[97]:
GF([[ 1, 16, 4],
    [11, 9, 9],
    [ 8, 24, 29]], order=31)

In [98]: A @ A @ A
Out[98]:
GF([[ 1, 16, 4],
    [11, 9, 9],
    [ 8, 24, 29]], order=31)

Matrix determinant: np.linalg.det(A)

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

In [100]: A = GF([[23, 11, 3, 3], [13, 6, 16, 4], [12, 10, 5, 3], [17, 23, 15, 28]]); A
Out[100]:
GF([[23, 11, 3, 3],
    [13, 6, 16, 4],
    [12, 10, 5, 3],
    [17, 23, 15, 28]], order=31)
Out[100]:
GF([[23, 11, 3, 3],
    [13, 6, 16, 4],
    [12, 10, 5, 3],
    [17, 23, 15, 28]], order=31)

In [101]: np.linalg.det(A)
Out[101]: GF(0, order=31)

Matrix rank: np.linalg.matrix_rank(A, z)

In [102]: GF = galois.GF(31)
In [103]: A = GF([[23, 11, 3, 3], [13, 6, 16, 4], [12, 10, 5, 3], [17, 23, 15, 28]]); A
Out[103]: GF([[23, 11, 3, 3],
        [13, 6, 16, 4],
        [12, 10, 5, 3],
        [17, 23, 15, 28]], order=31)

In [104]: np.linalg.matrix_rank(A)
Out[104]: 3

In [105]: A.row_reduce()
Out[105]:
GF([[ 1, 0, 0, 11],
    [ 0, 1, 0, 25],
    [ 0, 0, 1, 11],
    [ 0, 0, 0, 0]], order=31)

Matrix trace: np.trace(A)

In [106]: GF = galois.GF(31)
In [107]: A = GF([[23, 11, 3, 3], [13, 6, 16, 4], [12, 10, 5, 3], [17, 23, 15, 28]]); A
Out[107]: GF([[23, 11, 3, 3],
        [13, 6, 16, 4],
        [12, 10, 5, 3],
        [17, 23, 15, 28]], order=31)

In [108]: np.trace(A)
Out[108]: GF(0, order=31)

Out[109]: GF(0, order=31)

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

In [110]: GF = galois.GF(31)
In [111]: A = GF([[14, 21, 14, 28], [24, 22, 23, 23], [16, 30, 26, 18], [4, 23, 18, 3]]); A
(continues on next page)
Matrix inverse: np.linalg.inv(A)

In [115]: GF = galois.GF(31)
In [116]: A = GF([[14, 21, 14, 28], [24, 22, 23, 23], [16, 30, 26, 18], [4, 23, 18, 3]]); A
Out[116]:
GF([[14, 21, 14, 28],
    [24, 22, 23, 23],
    [16, 30, 26, 18],
    [ 4, 23, 18, 3]], order=31)
In [117]: A_inv = np.linalg.inv(A); A_inv
Out[117]:
GF([[27, 17, 9, 8],
    [20, 21, 12, 4],
    [30, 10, 23, 22],
    [13, 25, 6, 13]], order=31)
In [118]: A @ A_inv
Out[118]:
GF([[1, 0, 0, 0],
    [0, 1, 0, 0],
    [0, 0, 1, 0],
    [0, 0, 0, 1]], order=31)

3.6.5 Additional linear algebra

Below are additional linear algebra routines provided for FieldArray arrays/matrices that are not included in NumPy.

Row space: A.row_space()
\begin{align*}
[13, \ 6, \ 16, \ 4], \\
[12, \ 10, \ 5, \ 3], \\
[17, \ 23, \ 15, \ 28], \text{order}=31
\end{align*}

**In [121]:** A.row_space()

**Out[121]:**
\begin{align*}
\text{GF}([[\ 1, \ 0, \ 0, \ 0], \\
[\ 0, \ 1, \ 0, \ 11], \\
[\ 0, \ 0, \ 1, \ 5]], \text{order}=31)
\end{align*}

See `row_space()` for more details.

Column space: A.column_space()

**In [122]:** GF = galois.GF(31)

**In [123]:** A = GF([[23, 11, 3, 3], [13, 6, 16, 4], [12, 10, 5, 3], [17, 23, 15, 28]]); A

**Out[123]:**
\begin{align*}
\text{GF}([[23, \ 11, \ 3, \ 3], \\
[13, \ 6, \ 16, \ 4], \\
[12, \ 10, \ 5, \ 3], \\
[17, \ 23, \ 15, \ 28]], \text{order}=31)
\end{align*}

**In [124]:** A.column_space()

**Out[124]:**
\begin{align*}
\text{GF}([[\ 1, \ 0, \ 0, \ 0], \\
[\ 0, \ 1, \ 0, \ 11], \\
[\ 0, \ 0, \ 1, \ 5]], \text{order}=31)
\end{align*}

See `column_space()` for more details.

Left null space: A.left_null_space()

**In [125]:** GF = galois.GF(31)

**In [126]:** A = GF([[23, 11, 3, 3], [13, 6, 16, 4], [12, 10, 5, 3], [17, 23, 15, 28]]); A

**Out[126]:**
\begin{align*}
\text{GF}([[23, \ 11, \ 3, \ 3], \\
[13, \ 6, \ 16, \ 4], \\
[12, \ 10, \ 5, \ 3], \\
[17, \ 23, \ 15, \ 28]], \text{order}=31)
\end{align*}

**In [127]:** A.left_null_space()

**Out[127]:** GF([[ 0, \ 1, \ 23, \ 14]], \text{order}=31)

See `left_null_space()` for more details.

Null space: A.null_space()

**In [128]:** GF = galois.GF(31)

**In [129]:** A = GF([[23, 11, 3, 3], [13, 6, 16, 4], [12, 10, 5, 3], [17, 23, 15, 28]]); A

**Out[129]:**
\begin{align*}
\text{GF}([[23, \ 11, \ 3, \ 3], \\
[13, \ 6, \ 16, \ 4], \\
[12, \ 10, \ 5, \ 3], \\
[17, \ 23, \ 15, \ 28]], \text{order}=31)
\end{align*}


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

```
In [130]: A.null_space()
Out[130]: GF([[ 1, 22,  1, 14]], order=31)
```

See `null_space()` for more details.

Gaussian elimination: `A.row_reduce()`

```
```
```
In [131]: GF = galois.GF(31)
```
```
In [132]: A = GF([[23, 11, 3, 3], [13, 6, 16, 4], [12, 10, 5, 3], [17, 23, 15, 28]]); A
Out[132]:
GF([[23, 11, 3, 3],
 [13, 6, 16, 4],
 [12, 10, 5, 3],
 [17, 23, 15, 28]], order=31)
```
```
In [133]: A.row_reduce()
Out[133]:
GF([[ 1, 0, 0, 11],
 [ 0, 1, 0, 25],
 [ 0, 0, 1, 11],
 [ 0, 0, 0, 0]], order=31)
```

See `row_reduce()` for more details.

LU decomposition: `A.lu_decompose()`

```
```
```
In [134]: GF = galois.GF(31)
```
```
In [135]: A = GF([[4, 1, 24], [7, 6, 1], [11, 20, 2]]); A
Out[135]:
GF([[ 4,  1, 24],
 [ 7,  6,  1],
 [11, 20,  2]], order=31)
```
```
In [136]: L, U = A.lu_decompose()
In [137]: L
Out[137]:
GF([[ 1, 0, 0],
 [25, 1, 0],
 [26, 15, 1]], order=31)
```
```
In [138]: U
Out[138]:
GF([[ 4, 1, 24],
 [ 0, 12, 21],
 [ 0,  0, 24]], order=31)
```

(continues on next page)
See `lu_decompose()` for more details.

PLU decomposition: `A.plu_decompose()`

```python
In [140]: GF = galois.GF(31)
In [141]: A = GF([[15, 4, 11], [7, 6, 1], [11, 20, 2]]); A
Out[141]:
GF([[15, 4, 11],
    [ 7, 6, 1],
    [11, 20, 2]], order=31)
In [142]: P, L, U = A.plu_decompose()
In [143]: P
Out[143]:
GF([[1, 0, 0],
    [0, 0, 1],
    [0, 1, 0]], order=31)
In [144]: L
Out[144]:
GF([[ 1,  0,  0],
    [ 9,  1,  0],
    [17,  0,  1]], order=31)
In [145]: U
Out[145]:
GF([[15,  4, 11],
    [ 0, 15, 27],
    [ 0,  0,  0]], order=31)
In [146]: np.array_equal(P @ L @ U, A)
Out[146]: True
```

See `plu_decompose()` for more details.
### 3.7 Polynomials

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

#### 3.7.1 Create a polynomial

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

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

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

```python
In [3]: coeffs = GF([1, 0, 0, 55, 23]); coeffs
Out[3]: GF([ 1, 0, 0, 55, 23], order=2^8)
In [4]: galois.Poly(coeffs)
Out[4]: Poly(x^4 + 55x + 23, GF(2^8))
```

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

#### 3.7.2 Element representation

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

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

# Display f(x) using the default integer representation
In [6]: f = galois.Poly([13, 0, 4, 2], field=GF); print(f)
13x^3 + 4x + 2

# Display f(x) using the polynomial representation
In [7]: GF.display("poly"); print(f)
(\^2 + + 1)x^3 + ( + 1)x + 2

# Display f(x) using the power representation
In [8]: GF.display("power"); print(f)
(^10)x^3 + (^69)x + ^121

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

See [Element Representation](#) for more details.
3.7.3 Alternate constructors

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

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

```
In [10]: galois.Poly.Degrees([1000, 1], coeffs=[1, 179], field=GF)
Out[10]: Poly(x^1000 + 179x, GF(3^5))
```

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

**Integer**

```
In [11]: galois.Poly.Int(268, field=GF)
Out[11]: Poly(x + 25, GF(3^5))
```

**Binary string**

```
In [12]: galois.Poly.Int(int("0b1011", 2))
Out[12]: Poly(x^3 + x + 1, GF(2))
```

**Octal string**

```
In [13]: galois.Poly.Int(int("0o5034", 8), field=galois.GF(2**3))
Out[13]: Poly(5x^3 + 3x + 4, GF(2^3))
```

**Hex string**

```
In [14]: galois.Poly.Int(int("0xf700a275", 16), field=galois.GF(2**8))
Out[14]: Poly(247x^3 + 162x + 117, GF(2^8))
```

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

```
In [15]: galois.Poly.Str("x^5 + 143", field=GF)
Out[15]: Poly(x^5 + 143, GF(3^5))
```

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

```
In [16]: f = galois.Poly.Roots([137, 22, 51], field=GF); f
Out[16]: Poly(x^3 + 180x^2 + 19x + 58, GF(3^5))

In [17]: f.roots()
Out[17]: GF([ 22, 51, 137], order=3^5)
```

The `Zero()`, `One()`, and `Identity()` classmethods create common, simple polynomials. They are included for convenience.
Random polynomials of a given degree are easily created with `Random()`.

```python
In [21]: galois.Poly.Random(4, field=GF)
Out[21]: Poly(113x^4 + 195x^3 + 179x^2 + 2x + 217, GF(3^5))
```

### 3.7.4 Methods

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

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

```python
In [22]: GF = galois.GF(7)

In [23]: f = galois.Poly([1, 0, 5, 2, 3], field=GF); f
Out[23]: Poly(x^4 + 5x^2 + 2x + 3, GF(7))

In [24]: f.derivative()
Out[24]: Poly(4x^3 + 3x + 2, GF(7))
```

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

```python
In [25]: f.roots()
Out[25]: GF([5, 6], order=7)
```

### 3.7.5 Properties

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

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

```python
In [26]: GF = galois.GF(7)

In [27]: f = galois.Poly([1, 0, 3], field=GF); f
Out[27]: Poly(x^2 + 3, GF(7))

In [28]: f.nonzero_degrees
Out[28]: array([2, 0])

In [29]: f.nonzero_coeffs
Out[29]: GF([1, 3], order=7)
```

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

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

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

```
In [38]: galois.irreducible_poly(3, 3)
Out[38]: Poly(x^3 + 2x + 1, GF(3))

In [39]: list(galois.irreducible_polys(3, 3))
Out[39]:
[Poly(x^3 + 2x + 1, GF(3))],
Poly(x^3 + 2x + 2, GF(3))],
Poly(x^3 + x^2 + 2, GF(3))]
```
Find one or all primitive polynomials with \texttt{primitive\_poly()} and \texttt{primitive\_polys()}.

\begin{verbatim}
In [40]: galois.primitive_poly(3, 3)
Out[40]: Poly(x^3 + 2x + 1, GF(3))

In [41]: list(galois.primitive_polys(3, 3))
Out[41]:
[Poly(x^3 + 2x + 1, GF(3)),
 Poly(x^3 + x^2 + 2x + 1, GF(3)),
 Poly(x^3 + 2x^2 + 1, GF(3)),
 Poly(x^3 + 2x^2 + x + 1, GF(3)),
 Poly(x^3 + 2x^2 + 2x + 2, GF(3))]
\end{verbatim}

Find the Conway polynomial using \texttt{conway\_poly()}.

\begin{verbatim}
In [42]: galois.conway_poly(3, 3)
Out[42]: Poly(x^3 + 2x + 1, GF(3))
\end{verbatim}

### 3.8 Polynomial Arithmetic

#### 3.8.1 Standard arithmetic

After creating a \textit{polynomial over a finite field}, nearly any polynomial arithmetic operation can be performed using Python operators.

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

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

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

In [3]: g = galois.Poly([2, 1, 3], field=GF); g
Out[3]: Poly(2x^2 + x + 3, GF(7))
\end{verbatim}

Expand any section for more details.

Addition: \( f + g \)

Add two polynomials.

\begin{verbatim}
In [4]: f + g
Out[4]: Poly(x^3 + 2x^2 + 5x + 6, GF(7))
\end{verbatim}

Add a polynomial and a finite field scalar. The scalar is treated as a 0-degree polynomial.
In [5]: f + GF(3)
Out[5]: Poly(x^3 + 4x + 6, GF(7))

In [6]: GF(3) + f
Out[6]: Poly(x^3 + 4x + 6, GF(7))

Additive inverse: -f

In [7]: -f
Out[7]: Poly(6x^3 + 3x + 4, GF(7))

Any polynomial added to its additive inverse results in zero.

In [8]: f + -f
Out[8]: Poly(0, GF(7))

Subtraction: f - g

Subtract one polynomial from another.

In [9]: f - g
Out[9]: Poly(x^3 + 5x^2 + 3x, GF(7))

Subtract finite field scalar from a polynomial, or vice versa. The scalar is treated as a 0-degree polynomial.

In [10]: f - GF(3)
Out[10]: Poly(x^3 + 4x, GF(7))

In [11]: GF(3) - f
Out[11]: Poly(6x^3 + 3x, GF(7))

Multiplication: f * g

Multiply two polynomials.

In [12]: f * g
Out[12]: Poly(2x^5 + x^4 + 4x^3 + 3x^2 + x + 2, GF(7))

Multiply a polynomial and a finite field scalar. The scalar is treated as a 0-degree polynomial.

In [13]: f * GF(3)
Out[13]: Poly(3x^3 + 5x + 2, GF(7))

In [14]: GF(3) * f
Out[14]: Poly(3x^3 + 5x + 2, GF(7))

Scalar multiplication: f * 3

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

In [15]: f * 4
Out[15]: Poly(4x^3 + 2x + 5, GF(7))

In [16]: f + f + f + f
Out[16]: Poly(4x^3 + 2x + 5, GF(7))

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

```
In [17]: p = GF.characteristic; p
Out[17]: 7

In [18]: f * p
Out[18]: Poly(0, GF(7))
```

Division: $f \div g$

Divide one polynomial by another. Floor division is supported. True division is not supported since fractional polynomials are not currently supported.

```
In [19]: f \div g
Out[19]: Poly(4x + 5, GF(7))
```

Divide a polynomial by a finite field scalar, or vice versa. The scalar is treated as a 0-degree polynomial.

```
In [20]: f \div GF(3)
Out[20]: Poly(5x^3 + 6x + 1, GF(7))

In [21]: GF(3) \div g
Out[21]: Poly(0, GF(7))
```

Remainder: $f \mod g$

Divide one polynomial by another and keep the remainder.

```
In [22]: f \mod g
Out[22]: Poly(x + 2, GF(7))
```

Divide a polynomial by a finite field scalar, or vice versa, and keep the remainder. The scalar is treated as a 0-degree polynomial.

```
In [23]: f \mod GF(3)
Out[23]: Poly(0, GF(7))

In [24]: GF(3) \mod g
Out[24]: Poly(3, GF(7))
```

Divmod: \texttt{divmod}(f, g)

Divide one polynomial by another and return the quotient and remainder.

```
In [25]: divmod(f, g)
Out[25]: (Poly(4x + 5, GF(7)), Poly(x + 2, GF(7)))
```

Divide a polynomial by a finite field scalar, or vice versa, and keep the remainder. The scalar is treated as a 0-degree polynomial.

```
In [26]: divmod(f, GF(3))
Out[26]: (Poly(5x^3 + 6x + 1, GF(7)), Poly(0, GF(7)))

In [27]: divmod(GF(3), g)
Out[27]: (Poly(0, GF(7)), Poly(3, GF(7)))
```
Exponentiation: \( f \; ** \; 3 \)
Exponentiate a polynomial to a non-negative exponent.

<table>
<thead>
<tr>
<th>In [28]:</th>
<th>f ** 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Out[28]:</td>
<td>Poly(x^9 + 5x^7 + 2x^6 + 6x^5 + 2x^4 + 4x^2 + 3x + 6, GF(7))</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>In [29]:</th>
<th>pow(f, 3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Out[29]:</td>
<td>Poly(x^9 + 5x^7 + 2x^6 + 6x^5 + 2x^4 + 4x^2 + 3x + 6, GF(7))</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>In [30]:</th>
<th>f * f * f</th>
</tr>
</thead>
<tbody>
<tr>
<td>Out[30]:</td>
<td>Poly(x^9 + 5x^7 + 2x^6 + 6x^5 + 2x^4 + 4x^2 + 3x + 6, GF(7))</td>
</tr>
</tbody>
</table>

Modular exponentiation: \( \text{pow}(f, 123456789, g) \)
Exponentiate a polynomial to a non-negative exponent and reduce modulo another polynomial. This performs efficient modular exponentiation.

```python
# Efficiently computes (f ** 123456789) % g
In [31]: pow(f, 123456789, g)
Out[31]: Poly(x + 2, GF(7))
```

### 3.8.2 Special arithmetic

Polynomial objects also work on several special arithmetic operations. Below are some examples.

<table>
<thead>
<tr>
<th>In [32]:</th>
<th>GF = galois.GF(31)</th>
</tr>
</thead>
<tbody>
<tr>
<td>In [33]:</td>
<td>f = galois.Poly([1, 30, 0, 26, 6], field=GF); f</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>Out[33]:</td>
<td>Poly(x^4 + 30x^3 + 26x + 6, GF(31))</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>In [34]:</th>
<th>g = galois.Poly([4, 17, 3], field=GF); g</th>
</tr>
</thead>
<tbody>
<tr>
<td>Out[34]:</td>
<td>Poly(4x^2 + 17x + 3, GF(31))</td>
</tr>
</tbody>
</table>

Compute the polynomial greatest common divisor using \( \text{gcd}() \) and \( \text{egcd}() \).

<table>
<thead>
<tr>
<th>In [35]:</th>
<th>galois.gcd(f, g)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Out[35]:</td>
<td>Poly(1, GF(31))</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>In [36]:</th>
<th>galois.egcd(f, g)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Out[36]:</td>
<td>(Poly(1, GF(31)), Poly(14x + 13, GF(31)), Poly(12x^2 + 19x^2 + 22x + 26, GF(31)))</td>
</tr>
</tbody>
</table>

Factor a polynomial into its irreducible polynomial factors using \( \text{factors}() \).

<table>
<thead>
<tr>
<th>In [37]:</th>
<th>galois.factors(f)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Out[37]:</td>
<td>([Poly(x^2 + 5, GF(31)), Poly(x^2 + 30x + 26, GF(31))], [1, 1])</td>
</tr>
</tbody>
</table>
3.8.3 Polynomial evaluation

Polynomials are evaluated by invoking `__call__()`. They can be evaluated at scalars.

```
In [38]: GF = galois.GF(31)
In [39]: f = galois.Poly([1, 0, 0, 15], field=GF); f
Out[39]: Poly(x^3 + 15, GF(31))
In [40]: f(26)
Out[40]: GF(14, order=31)
```

# The equivalent field calculation
```
In [41]: GF(26)**3 + GF(15)
Out[41]: GF(14, order=31)
```

Or they can also be evaluated at arrays element-wise.

```
In [42]: x = GF([26, 13, 24, 4])
# Evaluate f(x) element-wise at a 1-D array
In [43]: f(x)
Out[43]: GF([14, 11, 13, 17], order=31)
In [44]: X = GF([[26, 13], [24, 4]])
# Evaluate f(x) element-wise at a 2-D array
In [45]: f(X)
Out[45]: GF([[2, 20],
        [25, 23]], order=31)
```

Or they can also be evaluated at square matrices. Note, this is different than element-wise array evaluation. Here, the square matrix indeterminate is exponentiated using matrix multiplication. So $f(x) = x^3$ evaluated at the square matrix $X$ equals $X \times X \times X$.

```
In [46]: f
Out[46]: Poly(x^3 + 15, GF(31))
# Evaluate f(x) at the 2-D square matrix
In [47]: f(X, elementwise=False)
Out[47]: GF([[ 2, 20],
        [25, 23]], order=31)
# The equivalent matrix operation
In [48]: np.linalg.matrix_power(X, 3) + GF(15)*GF.Identity(X.shape[0])
Out[48]: GF([[ 2, 20],
        [25, 23]], order=31)
```
3.9 Intro to Prime Fields

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

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.

3.9.1 Prime field

In this tutorial, we will consider the prime field \( GF(7) \). Using the galois library, the FieldArray subclass GF7 is created using the class factory \( GF() \).

Integer

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

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

Power

```
In [3]: GF7 = galois.GF(7, display="power")
```

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

Note: In this tutorial, we use the display="int" default to display the elements. However, sometimes it is useful to view elements in their power representation \( \{0, 1, \alpha, \alpha^2, ..., \alpha^{p^m-2}\} \). Switch the display between these two representations using the tabbed sections. Note, the polynomial representation is not shown because it is identical to the integer representation for prime fields.


3.9.2 Elements

The elements of the finite field \( GF(p) \) are naturally represented as the integers \( \{0, 1, \ldots, p - 1\} \).
The elements of the finite field are retrieved in a 1-D array using the \texttt{Elements()} method.

**Integer**

\[
\begin{align*}
\text{In [5]: } & \text{GF7.elements} \\
\text{Out[5]: } & \text{GF([0, 1, 2, 3, 4, 5, 6], order=7)}
\end{align*}
\]

**Power**

\[
\begin{align*}
\text{In [6]: } & \text{GF7.elements} \\
\text{Out[6]: } & \text{GF([ 0, 1, ^2, , ^4, ^5, ^3], order=7)}
\end{align*}
\]

3.9.3 Arithmetic

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

In this tutorial, consider two field elements \( a = 3 \) and \( b = 5 \). We will use \textit{galois} to perform explicit modular integer arithmetic and then prime field arithmetic.

Here are \( a \) and \( b \) represented as Python integers.

\[
\begin{align*}
\text{In [7]: } & \text{a_int = 3} \\
\text{In [8]: } & \text{b_int = 5} \\
\text{In [9]: } & \text{p = GF7.characteristic; p} \\
\text{Out[9]: } & 7
\end{align*}
\]

Here are \( a \) and \( b \) represented as prime field elements. See \textit{Array Creation} for more details.

**Integer**

\[
\begin{align*}
\text{In [10]: } & \text{a = GF7(3); a} \\
\text{Out[10]: } & \text{GF(3, order=7)} \\
\text{In [11]: } & \text{b = GF7(5); b} \\
\text{Out[11]: } & \text{GF(5, order=7)}
\end{align*}
\]
**Power**

In [12]: a = GF7(3); a
Out[12]: GF(3, order=7)

In [13]: b = GF7(5); b
Out[13]: GF(5, order=7)

**Addition**

We can see that $3 + 5 \equiv 1 \pmod{7}$. So accordingly, $3 + 5 = 1$ in GF(7).

**Integer**

In [14]: (a_int + b_int) % p
Out[14]: 1

In [15]: a + b
Out[15]: GF(1, order=7)

**Power**

In [16]: (a_int + b_int) % p
Out[16]: 1

In [17]: a + b
Out[17]: GF(1, order=7)

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

**Integer**

In [18]: print(GF7.arithmetic_table("+"))
x + y | 0 1 2 3 4 5 6
-------|---------------------
 0 | 0 1 2 3 4 5 6
 1 | 1 2 3 4 5 6 0
 2 | 2 3 4 5 6 0 1
 3 | 3 4 5 6 0 1 2
 4 | 4 5 6 0 1 2 3
 5 | 5 6 0 1 2 3 4
 6 | 6 0 1 2 3 4 5
Power

In [19]: print(GF7.arithmetic_table("+"))

x + y | 0 1 ^2 ^3 ^4 ^5
-------|---------------------
  0 | 0 1 ^2 ^3 ^4 ^5
  1 | 1 ^2 ^4 0 ^5 ^3
 ^2 | ^2 ^5 ^4 1 ^3 0
 ^3 | ^3 0 ^2 1 ^5 ^4
 ^4 | ^4 ^5 0 ^3 1 ^2
 ^5 | ^5 ^3 1 0 ^4 ^2

Subtraction

As with addition, we can see that $3 - 5 \equiv 5 \pmod{7}$. So accordingly, $3 - 5 = 5$ in GF(7).

Integer

In [20]: (a_int - b_int) % p
Out[20]: 5

In [21]: a - b
Out[21]: GF(5, order=7)

Power

In [22]: (a_int - b_int) % p
Out[22]: 5

In [23]: a - b
Out[23]: GF(^5, order=7)

Here is the subtraction table for completeness.

Integer

In [24]: print(GF7.arithmetic_table("-"))

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

58 Chapter 3. Citation
Power

```
In [25]: print(GF7.arithmetic_table("-"))
x - y | 0 1  \^2 \^3 \^4 \^5
------|-----------------------------------------------
  0 | 0 \^3 \^4 \^5 1 \^2
  1 | 1 0 \^5 \^3 \^2 \^4
\^2 | \^2 0 1 \^4 \^3 \^5
\^3 | \^3 \^5 \^4 0 \^2 1
\^4 | \^4 1 \^2 \^5 0 \^3
\^5 | \^5 \^4 \^2 \^3 1 0
```

Multiplication

Similarly, we can see that $3 \cdot 5 \equiv 1 \pmod{7}$. So accordingly, $3 \cdot 5 = 1$ in $\text{GF}(7)$.

Integer

```
In [26]: (a_int * b_int) % p
Out[26]: 1

In [27]: a * b
Out[27]: GF(1, order=7)
```

Power

```
In [28]: (a_int * b_int) % p
Out[28]: 1

In [29]: a * b
Out[29]: GF(1, order=7)
```

Here is the multiplication table for completeness.

Integer

```
In [30]: print(GF7.arithmetic_table("*"))
x * y | 0 1 2 3 4 5 6
------|---------------------
  0 | 0 0 0 0 0 0 0
  1 | 0 1 2 3 4 5 6
  2 | 0 2 4 6 1 3 5
  3 | 0 3 6 2 5 1 4
  4 | 0 4 1 5 2 6 3
  5 | 0 5 3 1 6 4 2
  6 | 0 6 5 4 3 2 1
```

3.9. Intro to Prime Fields
Power

```
In [31]: print(GF7.arithmetic_table("*"))
```

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

Multiplicative inverse

Division in GF(p) is a little more difficult. Division can’t be as simple as taking \( a/b \) (mod \( p \)) because many integer divisions do not result in integers! The division \( a/b \) can be reformulated into \( ab^{-1} \), where \( b^{-1} \) is the multiplicative inverse of \( b \). Let’s first learn the multiplicative inverse before returning to division.

Euclid discovered an efficient algorithm to solve the Bézout Identity, which is used to find the multiplicative inverse. It is now called the Extended Euclidean Algorithm. Given two integers \( x \) and \( y \), the Extended Euclidean Algorithm finds the integers \( s \) and \( t \) such that \( xs + yt = \gcd(x, y) \). This algorithm is implemented in \texttt{egcd()}.

If \( x = 5 \) is a field element of GF(7) and \( y = 7 \) is the prime characteristic, then \( s = x^{-1} \) in GF(7). Note, the GCD will always be 1 because \( y \) is prime.

```
# Returns (gcd, s, t)
In [32]: galois.egcd(b_int, p)
Out[32]: (1, 3, -2)
```

The \texttt{galois} library uses the Extended Euclidean Algorithm to compute multiplicative inverses (and division) in prime fields. The inverse of 5 in GF(7) can be easily computed in the following way.

Integer

```
In [33]: b ** -1
Out[33]: GF(3, order=7)
```

```
In [34]: np.reciprocal(b)
Out[34]: GF(3, order=7)
```

Power

```
In [35]: b ** -1
Out[35]: GF(3, order=7)
```

```
In [36]: np.reciprocal(b)
Out[36]: GF(3, order=7)
```
Division

Now let’s return to division in finite fields. As mentioned earlier, \( a/b \) is equivalent to \( ab^{-1} \), and we have already learned multiplication and multiplicative inversion in finite fields.

To compute \( 3/5 \) in GF(7), we can equivalently compute \( 3 \cdot 5^{-1} \) in GF(7).

**Integer**

```
In [37]: _, b_inv_int, _ = galois.egcd(b_int, p)
In [38]: (a_int * b_inv_int) % p
Out[38]: 2
In [39]: a * b**-1
Out[39]: GF(2, order=7)
In [40]: a / b
Out[40]: GF(2, order=7)
```

**Power**

```
In [41]: _, b_inv_int, _ = galois.egcd(b_int, p)
In [42]: (a_int * b_inv_int) % p
Out[42]: 2
In [43]: a * b**-1
Out[43]: GF(^2, order=7)
In [44]: a / b
Out[44]: GF(^2, order=7)
```

Here is the division table for completeness. Notice that division is not defined for \( y = 0 \).

**Integer**

```
In [45]: print(GF7.arithmetic_table("/"))
x / y | 1 2 3 4 5 6
--------|------------------
0 | 0 0 0 0 0 0
1 | 1 4 5 2 3 6
2 | 2 1 3 4 6 5
3 | 3 5 1 6 2 4
4 | 4 2 6 1 5 3
5 | 5 6 4 3 1 2
6 | 6 3 2 5 4 1
```
3.9.4 Primitive elements

A property of finite fields is that some elements produce the non-zero elements of the field by their powers. A primitive element $g$ of $\text{GF}(p)$ is an element such that $\text{GF}(p) = \{0, 1, g, g^2, \ldots, g^{p-2}\}$. The non-zero elements $\{1, g, g^2, \ldots, g^{p-2}\}$ form the cyclic multiplicative group $\text{GF}(p)^\times$. A primitive element has multiplicative order $\text{ord}(g) = p - 1$.

In prime fields $\text{GF}(p)$, the generators or primitive elements of $\text{GF}(p)$ are primitive roots mod $p$.

**Primitive roots mod $p$**

An integer $g$ is a primitive root mod $p$ if every number coprime to $p$ can be represented as a power of $g$ mod $p$. Namely, every $a$ coprime to $p$ can be represented as $g^k \equiv a \pmod{p}$ for some $k$. In prime fields, since $p$ is prime, every integer $1 \leq a < p$ is coprime to $p$.

Finding primitive roots mod $p$ is implemented in `primitive_root()` and `primitive_roots()`.

```
In [47]: galois.primitive_root(7)
Out[47]: 3
```

**A primitive element**

In `galois`, a primitive element of a finite field is provided by the `primitive_element` class property.

**Integer**

```
In [48]: print(GF7.properties)
Galois Field:
    name: GF(7)
    characteristic: 7
    degree: 1
    order: 7
    irreducible_poly: x + 4
    is_primitive_poly: True
    primitive_element: 3
```
In [49]: g = GF7.primitive_element; g
Out[49]: GF(3, order=7)

Power

In [50]: print(GF7.properties)
Galois Field:
   name: GF(7)
   characteristic: 7
   degree: 1
   order: 7
   irreducible_poly: x + 4
   isPrimitive_poly: True
   primitive_element: 3

In [51]: g = GF7.primitive_element; g
Out[51]: GF(, order=7)

The galois package allows you to easily display all powers of an element and their equivalent polynomial, vector, and integer representations using repr_table(). Let’s ignore the polynomial and vector representations for now. They will become useful for extension fields.

Here is the representation table using the default generator \( g = 3 \). Notice its multiplicative order is \( p - 1 \).

In [52]: g.multiplicative_order()
Out[52]: 6

In [53]: print(GF7.repr_table())

<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>
<tr>
<td>3^5</td>
<td>5</td>
<td>[5]</td>
<td>5</td>
</tr>
</tbody>
</table>
Other primitive elements

There are multiple primitive elements of any finite field. All primitive elements are provided in the `primitive_elements` class property.

Integer

```
In [54]: list(galois.primitive_roots(7))
Out[54]: [3, 5]

In [55]: GF7.primitive_elements
Out[55]: GF([3, 5], order=7)

In [56]: g = GF7(5); g
Out[56]: GF(5, order=7)
```

Power

```
In [57]: list(galois.primitive_roots(7))
Out[57]: [3, 5]

In [58]: GF7.primitive_elements
Out[58]: GF([ , ^5], order=7)

In [59]: g = GF7(5); g
Out[59]: GF(^5, order=7)
```

This means that 3 and 5 generate the multiplicative group $\mathbb{GF}(7)^\times$. We can examine this by viewing the representation table using different generators.

Here is the representation table using a different generator $g = 5$. Notice it also has multiplicative order $p - 1$.

```
In [60]: g.multiplicative_order()
Out[60]: 6

In [61]: print(GF7.repr_table(g))
```

<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>$5^0$</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>
Non-primitive elements

All other elements of the field cannot generate the multiplicative group. They have multiplicative orders less than $p - 1$. For example, the element $e = 2$ is not a primitive element.

<table>
<thead>
<tr>
<th>Integer</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>In [62]:</strong> e = GF7(2); e</td>
</tr>
<tr>
<td><strong>Out[62]:</strong> GF(2, order=7)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Power</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>In [63]:</strong> e = GF7(2); e</td>
</tr>
<tr>
<td><strong>Out[63]:</strong> GF(2, order=7)</td>
</tr>
</tbody>
</table>

It has ord($e$) = 3. Notice elements 3, 5, and 6 are not represented by the powers of $e$.

<table>
<thead>
<tr>
<th>Integer</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>In [64]:</strong> e.multiplicative_order()</td>
</tr>
<tr>
<td><strong>Out[64]:</strong> 3</td>
</tr>
</tbody>
</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>$2^0$</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>

3.10 Intro to Extension Fields

As discussed in the *Intro to Prime Fields* tutorial, a finite field is a finite set that is closed under addition, subtraction, multiplication, and division. Galois proved that finite fields exist only when their order (or size of the set) is a prime power $p^m$.

When the order is prime, the arithmetic is mostly computed using integer arithmetic modulo $p$. When the order is a prime power, namely extension fields $GF(p^m)$, the arithmetic is mostly computed using polynomial arithmetic modulo the irreducible polynomial $f(x)$.  

3.10. Intro to Extension Fields 65
3.10.1 Extension field

In this tutorial, we will consider the extension field $\text{GF}(3^2)$. Using the \texttt{galois} library, the \texttt{FieldArray} subclass \texttt{GF9} is created using the class factory \texttt{GF()}.

**Integer**

```
In [1]: GF9 = galois.GF(3**2)
In [2]: print(GF9.properties)
Galois Field:
   name: GF(3^2)
   characteristic: 3
   degree: 2
   order: 9
   irreducible_poly: x^2 + 2x + 2
   is_primitive_poly: True
   primitive_element: x
```

**Polynomial**

```
In [3]: GF9 = galois.GF(3**2, display="poly")
In [4]: print(GF9.properties)
Galois Field:
   name: GF(3^2)
   characteristic: 3
   degree: 2
   order: 9
   irreducible_poly: x^2 + 2x + 2
   is_primitive_poly: True
   primitive_element: x
```

**Power**

```
In [5]: GF9 = galois.GF(3**2, display="power")
In [6]: print(GF9.properties)
Galois Field:
   name: GF(3^2)
   characteristic: 3
   degree: 2
   order: 9
   irreducible_poly: x^2 + 2x + 2
   is_primitive_poly: True
   primitive_element: x
```

**Note:** In this tutorial, we use \texttt{display="poly"} to display the elements in their polynomial form. Although, it is common to use the default integer representation $\{0, 1, \ldots, p^m - 1\}$ to display the arrays more compactly. Switch the
display between the three representations using the tabbed sections.
See Element Representation for more details.

### 3.10.2 Elements

The elements of \( \mathbb{GF}(p^m) \) are polynomials over \( \mathbb{GF}(p) \) with degree less than \( m \). Formally, they are all polynomials \( a_{m-1}x^{m-1} + \cdots + a_1x + a_0 \in \mathbb{GF}(p)[x] \). There are exactly \( p^m \) elements.

The elements of the finite field are retrieved in a 1-D array using the `Elements()` classmethod.

#### Integer

In [7]: GF9.elements

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

#### Polynomial

In [8]: GF9.elements

Out[8]: GF([0, 1, ^2, ^1, ^2, +1, ^2, 2, 2 + 2], order=3^2)

#### Power

In [9]: GF9.elements

Out[9]: GF([0, 1, ^4, , ^2, ^7, ^5, ^3, ^6], order=3^2)

### 3.10.3 Irreducible polynomial

Every extension field must be defined with respect to an irreducible polynomial \( f(x) \). This polynomial defines the arithmetic of the field.

When creating a FieldArray subclass in galois, if an irreducible polynomial is not explicitly specified, a default is chosen. The default is the Conway polynomial \( C_{p,m}(x) \), which is irreducible and primitive. See conway_poly() for more information.

Notice \( f(x) \) is over \( \mathbb{GF}(3) \) with degree 2.

In [10]: f = GF9.irreducible_poly; f

Out[10]: Poly(x^2 + 2x + 2, GF(3))

Also note, when factored, \( f(x) \) has no irreducible factors other than itself – an analogue of a prime number.

In [11]: f.is_irreducible()

Out[11]: True

In [12]: f.factors()

Out[12]: ([Poly(x^2 + 2x + 2, GF(3))], [1])
3.10.4 Arithmetic

Addition, subtraction, and multiplication in \(\mathbb{GF}(p^m)\) with irreducible polynomial \(f(x)\) is equivalent to polynomial addition, subtraction, and multiplication over \(\mathbb{GF}(p)\) reduced modulo \(f(x)\). Mathematically speaking, this is the polynomial ring \(\mathbb{GF}(p)[x]/f(x)\).

In this tutorial, consider two field elements \(a = x + 2\) and \(b = x + 1\). We will use \textit{galois} to perform explicit polynomial calculations and then extension field arithmetic.

Here are \(a\) and \(b\) represented using \textit{Poly} objects.

```
In [13]: GF3 = galois.GF(3)
In [14]: a_poly = galois.Poly([1, 2], field=GF3); a_poly
Out[14]: Poly(x + 2, GF(3))
In [15]: b_poly = galois.Poly([1, 1], field=GF3); b_poly
Out[15]: Poly(x + 1, GF(3))
```

Here are \(a\) and \(b\) represented as extension field elements. Extension field elements can be specified as integers or polynomial strings. See \textit{Array Creation} for more details.

**Integer**

```
In [16]: a = GF9("x + 2"); a
Out[16]: GF(5, order=3^2)
In [17]: b = GF9("x + 1"); b
Out[17]: GF(4, order=3^2)
```

**Polynomial**

```
In [18]: a = GF9("x + 2"); a
Out[18]: GF( + 2, order=3^2)
In [19]: b = GF9("x + 1"); b
Out[19]: GF( + 1, order=3^2)
```

**Power**

```
In [20]: a = GF9("x + 2"); a
Out[20]: GF(^7, order=3^2)
In [21]: b = GF9("x + 1"); b
Out[21]: GF(^2, order=3^2)
```
Addition

In polynomial addition, the polynomial coefficients add degree-wise in $\text{GF}(p)$. Addition of polynomials with degree less than $m$ will never result in a polynomial of degree $m$ or greater. Therefore, it is unnecessary to reduce modulo the degree-$m$ polynomial $f(x)$, since the quotient will always be zero.

We can see that $a + b = (1 + 1)x + (2 + 1) = 2x$.

**Integer**

```python
In [22]: a_poly + b_poly
Out[22]: Poly(2x, GF(3))

In [23]: a + b
Out[23]: GF(6, order=3^2)
```

**Polynomial**

```python
In [24]: a_poly + b_poly
Out[24]: Poly(2x, GF(3))

In [25]: a + b
Out[25]: GF(2, order=3^2)
```

**Power**

```python
In [26]: a_poly + b_poly
Out[26]: Poly(2x, GF(3))

In [27]: a + b
Out[27]: GF(^5, order=3^2)
```

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

**Integer**

```python
In [28]: print(GF9.arithmetic_table("+"))
x + y | 0 1 2 3 4 5 6 7 8
-------|---------------------------
  0 | 0 1 2 3 4 5 6 7 8
  1 | 1 2 0 4 5 3 7 8 6
  2 | 2 0 1 5 3 4 8 6 7
  3 | 3 4 5 6 7 8 0 1 2
  4 | 4 5 3 7 8 6 1 2 0
  5 | 5 3 4 8 6 7 2 0 1
  6 | 6 7 8 0 1 2 3 4 5
```

(continues on next page)
Polynomial

In [29]: print(GF9.arithmetic_table("+"))

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

Power

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

| x + y | 0  | 1  | ^2 | ^3 | ^4 | ^5 | ^6 | ^7 |
|-------|====|====|====|====|====|====|====|====|
| 0     | 0  | 1  | ^2 | ^3 | ^4 | ^5 | ^6 | ^7 |
| 1     | 1  | ^4 | ^2 | ^7 | ^6 | 0  | ^3 | ^5 |
| ^2    | ^2 | ^5 | ^3 | 1  | ^7 | 0  | ^4 | ^6 |
| ^3    | ^3 | ^6 | 1  | ^4 | ^7 | ^5 | ^2 | 0  |
| ^4    | ^4 | 0  | ^7 | ^5 | 1  | ^6 | ^3 | ^2 |
| ^5    | ^5 | ^3 | 0  | 1  | ^2 | ^6 | ^7 | ^4 |
| ^6    | ^6 | ^5 | ^4 | 0  | ^3 | ^7 | ^2 | 1  |
| ^7    | ^7 | ^6 | ^5 | 0  | ^2 | ^4 | 1  | ^3 |
Subtraction

Subtraction, like addition, is performed on coefficients degree-wise and will never result in a polynomial with greater degree.

We can see that \( a - b = (1 - 1)x + (2 - 1) = 1 \).

Integer

```python
In [31]: a_poly - b_poly
Out[31]: Poly(1, GF(3))

In [32]: a - b
Out[32]: GF(1, order=3^2)
```

Polynomial

```python
In [33]: a_poly - b_poly
Out[33]: Poly(1, GF(3))

In [34]: a - b
Out[34]: GF(1, order=3^2)
```

Power

```python
In [35]: a_poly - b_poly
Out[35]: Poly(1, GF(3))

In [36]: a - b
Out[36]: GF(1, order=3^2)
```

Here is the entire subtraction table for completeness.

Integer

```python
In [37]: print(GF9.arithmetic_table("-"))
x - y | 0 1 2 3 4 5 6 7 8
--------|---------------------------
  0 | 0 2 1 6 8 7 3 5 4
  1 | 1 0 2 7 6 8 4 3 5
  2 | 2 1 0 8 7 6 5 4 3
  3 | 3 5 4 0 2 1 6 8 7
  4 | 4 3 5 1 0 2 7 6 8
  5 | 5 4 3 2 1 0 8 7 6
  6 | 6 8 7 3 5 4 0 2 1
  7 | 7 6 8 4 3 5 1 0 2
  8 | 8 7 6 5 4 3 2 1 0
```
Polynomial

In [38]: print(GF9.arithmetic_table("-"))

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

Power

In [39]: print(GF9.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>^4</td>
<td>^5</td>
<td>^6</td>
<td>^7</td>
<td>1</td>
<td>^2</td>
<td>^3</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>^3</td>
<td>^5</td>
<td>^4</td>
<td>^2</td>
<td>^7</td>
<td>^6</td>
</tr>
<tr>
<td>^2</td>
<td>^7</td>
<td>0</td>
<td>^4</td>
<td>^6</td>
<td>^2</td>
<td>^5</td>
<td>^3</td>
<td>1</td>
</tr>
<tr>
<td>^3</td>
<td>^2</td>
<td>1</td>
<td>0</td>
<td>^5</td>
<td>^7</td>
<td>^3</td>
<td>^6</td>
<td>^4</td>
</tr>
<tr>
<td>^4</td>
<td>^3</td>
<td>^5</td>
<td>^2</td>
<td>0</td>
<td>^6</td>
<td>1</td>
<td>^4</td>
<td>^7</td>
</tr>
<tr>
<td>^5</td>
<td>^4</td>
<td>1</td>
<td>^6</td>
<td>^3</td>
<td>^2</td>
<td>0</td>
<td>^7</td>
<td>^5</td>
</tr>
<tr>
<td>^6</td>
<td>^5</td>
<td>^6</td>
<td>^7</td>
<td>^4</td>
<td>^3</td>
<td>0</td>
<td>1</td>
<td>^2</td>
</tr>
<tr>
<td>^7</td>
<td>^6</td>
<td>^3</td>
<td>^7</td>
<td>^2</td>
<td>1</td>
<td>^5</td>
<td>^4</td>
<td>0</td>
</tr>
</tbody>
</table>

Multiplication

Multiplication of polynomials with degree less than $m$, however, will often result in a polynomial of degree $m$ or greater. Therefore, it is necessary to reduce the result modulo $f(x)$.

First compute $ab = (x+2)(x+1) = x^2 + 2$. Notice that $x^2 + 2$ has degree 2, but the elements of GF(3$^2$) can have degree at most 1. Therefore, reduction modulo $f(x)$ is required. After remainder division, we see that $ab \equiv x \mod f(x)$.

Integer

# Note the degree is greater than 1
In [40]: a_poly * b_poly
Out[40]: Poly(x^2 + 2, GF(3))

In [41]: (a_poly * b_poly) % f
Out[41]: Poly(x, GF(3))

In [42]: a * b
Out[42]: GF(3, order=3^2)
Polynomial

```
# Note the degree is greater than 1
In [43]: a_poly * b_poly
Out[43]: Poly(x^2 + 2, GF(3))

In [44]: (a_poly * b_poly) % f
Out[44]: Poly(x, GF(3))

In [45]: a * b
Out[45]: GF(, order=3^2)
```

Power

```
# Note the degree is greater than 1
In [46]: a_poly * b_poly
Out[46]: Poly(x^2 + 2, GF(3))

In [47]: (a_poly * b_poly) % f
Out[47]: Poly(x, GF(3))

In [48]: a * b
Out[48]: GF(, order=3^2)
```

Here is the entire multiplication table for completeness.

Integer

```
In [49]: print(GF9.arithmetic_table("*

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

```
In [50]: print(GF9.arithmetic_table("*"))

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

Power

```
In [51]: print(GF9.arithmetic_table("^"))

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

Multiplicative inverse

As with prime fields, the division $a(x)/b(x)$ is reformulated into $a(x)b(x)^{-1}$. So, first we must compute the multiplicative inverse $b^{-1}$ before continuing onto division.

The Extended Euclidean Algorithm, which was used in prime fields on integers, can be used for extension fields on polynomials. Given two polynomials $a(x)$ and $b(x)$, the Extended Euclidean Algorithm finds the polynomials $s(x)$ and $t(x)$ such that $a(x)s(x) + b(x)t(x) = \text{gcd}(a(x), b(x))$. This algorithm is implemented in `egcd()`.

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

```
# Returns (gcd, s, t)
In [52]: galois.egcd(b_poly, f)
Out[52]: (Poly(1, GF(3)), Poly(2x + 2, GF(3)), Poly(1, GF(3)))
```

The `galois` library uses the Extended Euclidean Algorithm to compute multiplicative inverses (and division) in extension fields. The inverse of $x + 1$ in GF$(3^2)$ can be easily computed in the following way.
3.10. Intro to Extension Fields 75
Polynomial

In [63]: _, b_inv_poly, _ = galois.egcd(b_poly, f)

In [64]: (a_poly * b_inv_poly) % f
Out[64]: Poly(2x, GF(3))

In [65]: a * b**-1
Out[65]: GF(2, order=3^2)

In [66]: a / b
Out[66]: GF(2, order=3^2)

Power

In [67]: _, b_inv_poly, _ = galois.egcd(b_poly, f)

In [68]: (a_poly * b_inv_poly) % f
Out[68]: Poly(2x, GF(3))

In [69]: a * b**-1
Out[69]: GF(^5, order=3^2)

In [70]: a / b
Out[70]: GF(^5, order=3^2)

Here is the division table for completeness. Notice that division is not defined for \( y = 0 \).

Integer

In [71]: print(GF9.arithmetic_table("/"))
x / y | 1 2 3 4 5 6 7 8
------|------------------------
 0 | 0 0 0 0 0 0 0 0
 1 | 1 2 5 8 3 7 6 4
 2 | 2 1 7 4 6 5 3 8
 3 | 3 6 1 5 4 2 8 7
 4 | 4 8 3 1 7 6 5 2
 5 | 5 7 8 6 1 4 2 3
 6 | 6 3 2 7 8 1 4 5
 7 | 7 5 4 3 2 8 1 6
 8 | 8 4 6 2 5 3 7 1
Polynomial

\begin{verbatim}
In [72]: print(GF9.arithmetic_table("/"))
x / y | 1 2 + 1 + 2 2 2 + 1 2 + 2
-------|-------------------------------------------------------------------------------------------------
0 | 0 0 0 0 0 0 0 0
1 | 1 2 + 2 2 + 2 2 + 1 2 + 1
2 | 2 1 2 + 1 + 1 2 + 2 2 + 2
+ 1 | + 1 2 + 2 1 2 + 1 2 + 2 2
+ 2 | + 2 2 + 1 2 + 2 2 1 + 1 2
2 | 2 2 2 + 1 2 + 2 1 + 1 + 2
2 + 1 | 2 + 1 + 2 + 1 2 2 + 2 1 2
2 + 2 | 2 + 2 + 1 2 2 + 2 2 + 1 1
\end{verbatim}

Power

\begin{verbatim}
In [73]: print(GF9.arithmetic_table("/"))
x / y | 1 ^2 ^3 ^4 ^5 ^6 ^7
------|----------------------------------------
0 | 0 0 0 0 0 0 0 0
1 | 1 ^7 ^6 ^5 ^4 ^3 ^2
^2 | ^2 1 ^7 ^6 ^5 ^4 ^3
^3 | ^3 ^2 1 ^7 ^6 ^5 ^4
^4 | ^4 ^3 ^2 1 ^7 ^6 ^5
^5 | ^5 ^4 ^3 ^2 1 ^7 ^6
^6 | ^6 ^5 ^4 ^3 ^2 1 ^7
^7 | ^7 ^6 ^5 ^4 ^3 ^2 1
\end{verbatim}

3.10.5 Primitive elements

A property of finite fields is that some elements produce the non-zero elements of the field by their powers. A primitive element \( g \) of \( GF(p^m) \) is an element such that \( GF(p^m) = \{0, 1, g, g^2, \ldots, g^{p^m-2}\} \). The non-zero elements \( \{1, g, g^2, \ldots, g^{p^m-2}\} \) form the cyclic multiplicative group \( GF(p^m)^\times \). A primitive element has multiplicative order \( \text{ord}(g) = p^m - 1 \).

A primitive element

In \textit{galois}, a primitive element of a finite field is provided by the \texttt{primitive_element} class property.
The `galois` package allows you to easily display all powers of an element and their equivalent polynomial, vector, and integer representations using `repr_table()`.

Here is the representation table using the default generator $g = x$. Notice its multiplicative order is $p^m - 1$. 

```python
In [74]: print(GF9.properties)
Galois Field:
   name: GF(3^2)
   characteristic: 3
   degree: 2
   order: 9
   irreducible_poly: x^2 + 2x + 2
   is_primitive_poly: True
   primitive_element: x

In [75]: g = GF9.primitive_element; g
Out[75]: GF(3, order=3^2)

In [76]: print(GF9.properties)
Galois Field:
   name: GF(3^2)
   characteristic: 3
   degree: 2
   order: 9
   irreducible_poly: x^2 + 2x + 2
   is_primitive_poly: True
   primitive_element: x

In [77]: g = GF9.primitive_element; g
Out[77]: GF(3, order=3^2)

In [78]: print(GF9.properties)
Galois Field:
   name: GF(3^2)
   characteristic: 3
   degree: 2
   order: 9
   irreducible_poly: x^2 + 2x + 2
   is_primitive_poly: True
   primitive_element: x

In [79]: g = GF9.primitive_element; g
Out[79]: GF(3, order=3^2)
```
Other primitive elements

There are multiple primitive elements of any finite field. All primitive elements are provided in the `primitive_elements` class property.

Integer

```python
In [82]: GF9.primitive_elements
Out[82]: GF([3, 5, 6, 7], order=3^2)
```

```python
In [83]: g = GF9("2x + 1"); g
Out[83]: GF(7, order=3^2)
```

Polynomial

```python
In [84]: GF9.primitive_elements
Out[84]: GF([1, 2, 2 + 1, 2], order=3^2)
```

```python
In [85]: g = GF9("2x + 1"); g
Out[85]: GF(2 + 1, order=3^2)
```

Power

```python
In [86]: GF9.primitive_elements
Out[86]: GF([1, 7, 5, 3], order=3^2)
```

```python
In [87]: g = GF9("2x + 1"); g
Out[87]: GF(3, order=3^2)
```

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

Here is the representation table using a different generator \(g = 2x + 1\). Notice it also has multiplicative order \(p^m - 1\),
In [88]: g.multiplicative_order()
Out[88]: 8

In [89]: print(GF9.repr_table(g))

<table>
<thead>
<tr>
<th>Power</th>
<th>Polynomial</th>
<th>Vector</th>
<th>Integer</th>
</tr>
</thead>
<tbody>
<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>

Non-primitive elements

All other elements of the field cannot generate the multiplicative group. They have multiplicative orders less than \( p^m - 1 \).

For example, the element \( e = x + 1 \) is not a primitive element. It has \( \text{ord}(e) = 4 \). Notice elements \( x, x + 2, 2x, \) and \( 2x + 1 \) are not represented by the powers of \( e \).

Integer

In [90]: e = GF9("x + 1"); e
Out[90]: GF(4, order=3^2)

Polynomial

In [91]: e = GF9("x + 1"); e
Out[91]: GF(+1, order=3^2)

Power

In [92]: e = GF9("x + 1"); e
Out[92]: GF(4^2, order=3^2)

In [93]: e.multiplicative_order()
Out[93]: 4

In [94]: print(GF9.repr_table(e))

<table>
<thead>
<tr>
<th>Power</th>
<th>Polynomial</th>
<th>Vector</th>
<th>Integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x + 1)^0</td>
<td>0</td>
<td>[0, 0]</td>
<td>0</td>
</tr>
<tr>
<td>(x + 1)^1</td>
<td>1</td>
<td>[0, 1]</td>
<td>1</td>
</tr>
<tr>
<td>(x + 1)^2</td>
<td>x + 1</td>
<td>[1, 1]</td>
<td>4</td>
</tr>
</tbody>
</table>
3.11 Prime Fields

This page compares the performance of \textit{galois} to native NumPy when performing finite field multiplication in GF(p). Native NumPy can perform finite field multiplication in GF(p) because prime fields are very simple. Multiplication is simply $xy \mod p$.

3.11.1 Lookup table performance

This section tests \textit{galois} when using the "jit-lookup" compilation mode. For finite fields with order less than or equal to $2^{20}$, \textit{galois} uses lookup tables by default for efficient arithmetic.

Below are examples computing 10 million multiplications in the prime field GF(31).

```python
In [1]: import galois
In [2]: GF = galois.GF(31)
In [3]: GF.ufunc_mode
Out[3]: 'jit-lookup'
In [4]: a = GF.Random(10_000_000, seed=1, dtype=int)
In [5]: b = GF.Random(10_000_000, seed=2, dtype=int)
# Invoke the ufunc once to JIT compile it, if necessary
In [6]: a * b
Out[6]: GF([ 9, 27, 7, ..., 14, 21, 15], order=31)
In [7]: %timeit a * b
36 ms ± 1.07 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

The equivalent operation using native NumPy ufuncs is ~1.8x slower.

```python
In [8]: import numpy as np
In [9]: aa, bb = a.view(np.ndarray), b.view(np.ndarray)
In [10]: %timeit (aa * bb) % GF.order
65.3 ms ± 1.26 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```
3.11.2 Explicit calculation performance

This section tests \textit{galois} when using the "jit-calculate" compilation mode. For finite fields with order greater than $2^{20}$, \textit{galois} will use explicit arithmetic calculation by default rather than lookup tables. \textit{Even in these cases, galois} is faster than NumPy!

Below are examples computing 10 million multiplications in the prime field $GF(2097169)$.

\begin{verbatim}
In [1]: import galois

In [2]: GF = galois.GF(2097169)

In [3]: GF.ufunc_mode
Out[3]: 'jit-calculate'

In [4]: a = GF.Random(10_000_000, seed=1, dtype=int)

In [5]: b = GF.Random(10_000_000, seed=2, dtype=int)

# Invoke the ufunc once to JIT compile it, if necessary

In [6]: a * b
Out[6]: GF([1879104, 1566761, 967164, ..., 744769, 975853, 1142138], order=2097169)

In [7]: %timeit a * b
32.7 ms ± 1.44 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)

The equivalent operation using native NumPy ufuncs is ~2.5x slower.

In [8]: import numpy as np

In [9]: aa, bb = a.view(np.ndarray), b.view(np.ndarray)

In [10]: %timeit (aa * bb) % GF.order
78.8 ms ± 1.6 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
\end{verbatim}

3.11.3 Runtime floor

The \textit{galois} ufunc runtime has a floor, however. This is due to a requirement of the ufuncs to \texttt{.view()} the output array and convert its dtype with \texttt{.astype()}. Also the \textit{galois} ufuncs must perform input verification that NumPy ufuncs don't.

For example, for small array sizes, NumPy is faster than \textit{galois}. This is true whether using lookup tables or explicit calculation.

\begin{verbatim}
In [1]: import galois

In [2]: GF = galois.GF(2097169)

In [3]: GF.ufunc_mode
Out[3]: 'jit-calculate'

In [4]: a = GF.Random(10, seed=1, dtype=int)
\end{verbatim}

(continues on next page)
In [5]: b = GF.Random(10, seed=2, dtype=int)
   # Invoke the ufunc once to JIT compile it, if necessary
In [6]: a * b
Out[6]:
GF([1879104, 1566761, 967164, 1403108, 100593, 595358, 852783,
   1035698, 1207498, 989189], order=2097169)

In [7]: %timeit a * b
   7.62 µs ± 390 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)

The equivalent operation using native NumPy ufuncs is ~6x faster. However, in absolute terms, the difference is only ~6 µs.

In [8]: import numpy as np
In [9]: aa, bb = a.view(np.ndarray), b.view(np.ndarray)
In [10]: %timeit (aa * bb) % GF.order
   1.29 µs ± 12.6 ns per loop (mean ± std. dev. of 7 runs, 1,000,000 loops each)

3.11.4 Linear algebra performance

Linear algebra performance in prime fields is comparable to the native NumPy implementations, which use BLAS/LAPACK. This is because galois uses the native NumPy ufuncs when possible.

If overflow is prevented, dot products in GF(p) can be computed by first computing the dot product in Z and then reducing modulo p. In this way, the efficient BLAS/LAPACK implementations are used to keep finite field linear algebra fast, whenever possible.

Below are examples computing the matrix multiplication of two 100 × 100 matrices in the prime field GF(2097169).

In [1]: import galois
In [2]: GF = galois.GF(2097169)
In [3]: GF.ufunc_mode
Out[3]: 'jit-calculate'
In [4]: A = GF.Random((100,100), seed=1, dtype=int)
In [5]: B = GF.Random((100,100), seed=2, dtype=int)
   # Invoke the ufunc once to JIT compile it, if necessary
In [6]: A @ B
Out[6]:
GF([[1147163,  59466, 1841183, ..., 667877, 2084618,  79166],
   [ 306714, 1380503, 810935, ..., 1932687, 1690697,  329837],
   [ 325274, 575543, 1327001, ..., 167724, 422518,  696986],
   ..., [ 862992, 1143160, 588384, ...,  668891, 1285421, 1196448],
   [1026856, 1413416, 1844802, ...,  38844, 1643604, 10409],...,
In [7]: %timeit A @ B
708 µs ± 1.48 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)

The equivalent operation using native NumPy ufuncs is slightly faster. This is because galois has some internal overhead before invoking the same NumPy calculation.

In [8]: import numpy as np
In [9]: AA, BB = A.view(np.ndarray), B.view(np.ndarray)
In [10]: %timeit (AA @ BB) % GF.order
682 µs ± 11.1 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)

### 3.12 Binary Extension Fields

This page compares the performance of galois performing finite field multiplication in GF($2^m$) with native NumPy performing only modular multiplication.

Native NumPy cannot easily perform finite field multiplication in GF($2^m$) because it involves polynomial multiplication (convolution) followed by reducing modulo the irreducible polynomial. To make a similar comparison, NumPy will perform integer multiplication followed by integer remainder division.

**Important:** Native NumPy is not computing the correct result! This is not a fair fight!

These are not fair comparisons because NumPy is not computing the correct product. However, they are included here to provide a performance reference point with native NumPy.

#### 3.12.1 Lookup table performance

This section tests galois when using the "jit-lookup" compilation mode. For finite fields with order less than or equal to $2^{20}$, galois uses lookup tables by default for efficient arithmetic.

Below are examples computing 10 million multiplications in the binary extension field GF($2^8$).

In [1]: import galois
In [2]: GF = galois.GF(2**8)
In [3]: GF.ufunc_mode
Out[3]: 'jit-lookup'
In [4]: a = GF.Random(10_000_000, seed=1, dtype=int)
In [5]: b = GF.Random(10_000_000, seed=2, dtype=int)
# Invoke the ufunc once to JIT compile it, if necessary
In [6]: a * b
Out[6]: GF([181, 92, 148, ..., 255, 220, 153], order=2^8)

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

NumPy, even when computing the incorrect result, is ~1.9x slower than *galois*. This is because *galois* is using lookup tables instead of explicitly performing the polynomial multiplication and division.

In [8]: import numpy as np
In [9]: aa, bb = a.view(np.ndarray), b.view(np.ndarray)
In [10]: pp = int(GF.irreducible_poly)
   # This does not produce the correct result!
In [11]: %timeit (aa * bb) % pp
   64 ms ± 747 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)

3.12.2 Explicit calculation performance

This section tests *galois* when using the "jit-calculate" compilation mode. For finite fields with order greater than $2^{20}$, *galois* will use explicit arithmetic calculation by default rather than lookup tables.

Below are examples computing 10 million multiplications in the binary extension field $GF(2^{32})$.

In [1]: import galois
In [2]: GF = galois.GF(2**32)
In [3]: GF.ufunc_mode
Out[3]: 'jit-calculate'
In [4]: a = GF.Random(10_000_000, seed=1, dtype=int)
In [5]: b = GF.Random(10_000_000, seed=2, dtype=int)
   # Invoke the ufunc once to JIT compile it, if necessary
In [6]: a * b
Out[6]: GF([1174047800, 3249326965, 3196014003, ..., 3195457330, 100242821,
                  338589759], order=2^32)
In [7]: %timeit a * b
   386 ms ± 14 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

The *galois* library when using explicit calculation is only ~3.9x slower than native NumPy, which isn’t even computing the correct product.

In [8]: import numpy as np
In [9]: aa, bb = a.view(np.ndarray), b.view(np.ndarray)
(continues on next page)
In 

```python
In [10]: pp = int(GF.irreducible_poly)

# This does not produce the correct result!
In [11]: %timeit (aa * bb) % pp
100 ms ± 718 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

### 3.12.3 Linear algebra performance

Linear algebra performance in extension fields is definitely slower than native NumPy. This is because, unlike with prime fields, it is not possible to use the BLAS/LAPACK implementations. Instead, entirely new JIT-compiled ufuncs are generated, which are not as optimized for parallelism or hardware acceleration as BLAS/LAPACK.

Below are examples computing the matrix multiplication of two $100 \times 100$ matrices in the binary extension field $GF(2^{32})$.

```python
In [1]: import galois

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

In [3]: GF.ufunc_mode
Out[3]: 'jit-calculate'

In [4]: A = GF.Random((100,100), seed=1, dtype=int)

In [5]: B = GF.Random((100,100), seed=2, dtype=int)

# Invoke the ufunc once to JIT compile it, if necessary
In [6]: A @ B
Out[6]: GF([4203877556, 3977035749, 2623937858, ..., 3721257849, 4250999056, 4026271867],
    [3120760606, 1017695431, 1111117124, ..., 2988805996, 1734614583],
    [2508826906, 2800993411, 1720697782, ..., 3858180318, 2521070820, 3906771227],
    ...,
    [ 624580545, 984724090, 3969931498, ..., 1692192269, 473079794, 1029376699],
    [1232183301, 209395954, 2659712274, ..., 2967695343, 2747874320, 1249453570],
    [3938433735, 828783569, 3286222384, ..., 3669775257, 33626526, 4278384359]], order=2^32)

In [7]: %timeit A @ B
45.1 ms ± 264 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

The `galois` library is about ~65x slower than native NumPy (which isn’t computing the correct product).

```python
In [8]: import numpy as np
```
3.13 Benchmarks

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

3.13.1 Install dependencies

First, pytest and pytest-benchmark must be installed on your system. Easily install them by installing the development dependencies.

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

3.13.2 Create a benchmark

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

```
$ python3 -m pytest benchmarks/test_field_arithmetic.py --benchmark-columns=min,max,mean,stddev,median --benchmark-sort=name
```

**Test session**
```
... [100%]
```

**Name (time in us) Min Max Mean StdDev Median**
```
+-------------------------+-------------------+-------------------+-------------------+-------------------+
| test_add                | 16.3810 (1.21)    | 218.3280 (1.22)   | 18.9455 (1.17)    | 5.4959 (1.07)     |
|                         | 17                | 17.3620 (1.22)    |                  |                   |
|                         |                   |                   |                  |                   |
```

(continues on next page)
<table>
<thead>
<tr>
<th>Name</th>
<th>Min</th>
<th>Max</th>
<th>Mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>test_add</td>
<td>78.2860 (1.37)</td>
<td>311.2620 (1.40)</td>
<td>87.9984</td>
</tr>
<tr>
<td>test_additive_inverse</td>
<td>57.0860 (1.0)</td>
<td>281.9070 (1.27)</td>
<td>69.7403</td>
</tr>
<tr>
<td>test_divide</td>
<td>3,274.0860 (57.35)</td>
<td>3,351.6220 (15.09)</td>
<td>3,309.5920</td>
</tr>
<tr>
<td>test_multiplicative_inverse</td>
<td>3,455.1620 (56.85)</td>
<td>4,295.9590 (19.34)</td>
<td>3,350.8016</td>
</tr>
<tr>
<td>test_multiply</td>
<td>197.1090 (3.45)</td>
<td>305.5620 (1.38)</td>
<td>218.1805</td>
</tr>
<tr>
<td>test_power</td>
<td>3,270.7210 (57.29)</td>
<td>3,520.5480 (15.85)</td>
<td>3,349.1942</td>
</tr>
<tr>
<td>test_scalar_multiply</td>
<td>544.0880 (9.53)</td>
<td>1,182.1140 (5.32)</td>
<td>575.6227</td>
</tr>
<tr>
<td>test_subtract</td>
<td>77.6160 (1.36)</td>
<td>222.1760 (1.0)</td>
<td>88.3242</td>
</tr>
<tr>
<td>test_additive_inverse</td>
<td>5.9249 (1.16)</td>
<td>14.2670 (1.0)</td>
<td>9.6105</td>
</tr>
<tr>
<td>test_divide</td>
<td>132.0870 (9.72)</td>
<td>191.0680 (1.07)</td>
<td>169.2386</td>
</tr>
<tr>
<td>test_multiplicative_inverse</td>
<td>91.4410 (6.73)</td>
<td>179.0050 (1.0)</td>
<td>111.1525</td>
</tr>
<tr>
<td>test_multiply</td>
<td>16.0400 (1.18)</td>
<td>22.4038 (4.57)</td>
<td>18.3296</td>
</tr>
<tr>
<td>test_power</td>
<td>150.2410 (11.06)</td>
<td>212.2870 (1.19)</td>
<td>181.6396</td>
</tr>
<tr>
<td>test_scalar_multiply</td>
<td>543.3970 (40.00)</td>
<td>714.2870 (3.99)</td>
<td>626.2968</td>
</tr>
<tr>
<td>test_subtract</td>
<td>16.3110 (1.20)</td>
<td>2,233.8710 (12.48)</td>
<td>19.2938</td>
</tr>
<tr>
<td>test_additive_inverse</td>
<td>13.5850 (1.0)</td>
<td>206.5360 (1.15)</td>
<td>16.1445</td>
</tr>
<tr>
<td>test_divide</td>
<td>132.0870 (9.72)</td>
<td>191.0680 (1.07)</td>
<td>149.6357</td>
</tr>
<tr>
<td>test_multiplicative_inverse</td>
<td>91.4410 (6.73)</td>
<td>179.0050 (1.0)</td>
<td>102.6590</td>
</tr>
<tr>
<td>test_multiply</td>
<td>16.0400 (1.18)</td>
<td>22.4038 (4.57)</td>
<td>18.3296</td>
</tr>
<tr>
<td>test_power</td>
<td>150.2410 (11.06)</td>
<td>212.2870 (1.19)</td>
<td>168.8103</td>
</tr>
<tr>
<td>test_scalar_multiply</td>
<td>543.3970 (40.00)</td>
<td>714.2870 (3.99)</td>
<td>562.2968</td>
</tr>
<tr>
<td>test_subtract</td>
<td>16.3110 (1.20)</td>
<td>2,233.8710 (12.48)</td>
<td>19.2938</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Min</th>
<th>Max</th>
<th>Mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>test_add</td>
<td>79.0550 (1.37)</td>
<td>393.6670 (2.39)</td>
<td>86.7954</td>
</tr>
<tr>
<td>test_additive_inverse</td>
<td>12.6945 (1.0)</td>
<td>81.4630 (1.34)</td>
<td>69.0218</td>
</tr>
<tr>
<td>test_divide</td>
<td>21.7213 (1.71)</td>
<td>60.6330 (1.0)</td>
<td>34.5755</td>
</tr>
<tr>
<td>test_multiplicative_inverse</td>
<td>57.9080 (1.0)</td>
<td>164.6380 (1.0)</td>
<td>91.8218</td>
</tr>
<tr>
<td>test_multiply</td>
<td>16.0400 (1.18)</td>
<td>22.4038 (4.57)</td>
<td>18.3296</td>
</tr>
<tr>
<td>test_power</td>
<td>150.2410 (11.06)</td>
<td>212.2870 (1.19)</td>
<td>168.8103</td>
</tr>
<tr>
<td>test_scalar_multiply</td>
<td>543.3970 (40.00)</td>
<td>714.2870 (3.99)</td>
<td>562.2968</td>
</tr>
<tr>
<td>test_subtract</td>
<td>16.3110 (1.20)</td>
<td>2,233.8710 (12.48)</td>
<td>19.2938</td>
</tr>
</tbody>
</table>

(continues on next page)
### 3.13. Benchmarks

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Min</th>
<th>Max</th>
<th>Mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>test_divide</td>
<td>228.7890 (3.95)</td>
<td>280.8050 (1.71)</td>
<td>243.1431 (3.52)</td>
</tr>
<tr>
<td>test_multiplicative_inverse</td>
<td>263.8140 (4.56)</td>
<td>348.4620 (2.12)</td>
<td>290.6663 (4.21)</td>
</tr>
<tr>
<td>test_multiply</td>
<td>193.5820 (3.34)</td>
<td>475.2490 (2.89)</td>
<td>216.4317 (3.14)</td>
</tr>
<tr>
<td>test_power</td>
<td>311.6030 (5.38)</td>
<td>389.2180 (2.36)</td>
<td>328.9333 (4.77)</td>
</tr>
<tr>
<td>test_scalar_multiply</td>
<td>539.7710 (9.32)</td>
<td>973.1410 (5.91)</td>
<td>573.4538 (8.31)</td>
</tr>
<tr>
<td>test_subtract</td>
<td>80.3500 (1.39)</td>
<td>270.0450 (1.64)</td>
<td>97.6062 (1.47)</td>
</tr>
</tbody>
</table>

---

### 3.13. Benchmarks (continued)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Min</th>
<th>Max</th>
<th>Mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>test_add</td>
<td>16.6110 (1.21)</td>
<td>218.1990 (1.09)</td>
<td>19.0428 (1.0)</td>
</tr>
<tr>
<td>test_additive_inverse</td>
<td>13.6750 (1.0)</td>
<td>200.7150 (1.0)</td>
<td>16.0465 (1.0)</td>
</tr>
<tr>
<td>test_divide</td>
<td>13,280.4310 (971.15)</td>
<td>13,367.6440 (66.60)</td>
<td>13,340.0968 (831.34)</td>
</tr>
<tr>
<td>test_multiplicative_inverse</td>
<td>11,842.1600 (865.97)</td>
<td>15,404.4870 (76.75)</td>
<td>12,129.1417 (755.88)</td>
</tr>
<tr>
<td>test_multiply</td>
<td>1,079.0300 (78.91)</td>
<td>1,137.0780 (5.67)</td>
<td>1,098.1473 (68.44)</td>
</tr>
<tr>
<td>test_power</td>
<td>12,832.8340 (938.41)</td>
<td>13,115.7640 (65.35)</td>
<td>12,942.1951 (806.54)</td>
</tr>
<tr>
<td>test_scalar_multiply</td>
<td>883.2930 (64.59)</td>
<td>1,192.1310 (5.94)</td>
<td>928.3991 (57.86)</td>
</tr>
<tr>
<td>test_subtract</td>
<td>16.6210 (1.22)</td>
<td>1,334.7780 (6.65)</td>
<td>19.7528 (1.23)</td>
</tr>
</tbody>
</table>

---

### 3.13. Benchmarks (continued)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Min</th>
<th>Max</th>
<th>Mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>test_add</td>
<td>16.0900 (1.23)</td>
<td>277.5990 (3.89)</td>
<td>18.8739 (1.24)</td>
</tr>
<tr>
<td>test_additive_inverse</td>
<td>13.1050 (1.0)</td>
<td>71.3340 (1.0)</td>
<td>15.1649 (1.0)</td>
</tr>
<tr>
<td>test_divide</td>
<td>215.6730 (16.46)</td>
<td>271.6490 (3.81)</td>
<td>233.7595 (15.41)</td>
</tr>
</tbody>
</table>

*(continues on next page)*
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Name</th>
<th>Min</th>
<th>Max</th>
<th>Mean</th>
<th>StdDev</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>test_add</td>
<td>313.477</td>
<td>358.230</td>
<td>328.456</td>
<td>12.133</td>
</tr>
<tr>
<td></td>
<td>test_additive_inverse</td>
<td>153.698</td>
<td>226.655</td>
<td>177.913</td>
<td>19.689</td>
</tr>
<tr>
<td></td>
<td>test_divide</td>
<td>222.346</td>
<td>284.713</td>
<td>235.549</td>
<td>15.518</td>
</tr>
<tr>
<td></td>
<td>test_multiplicative_inverse</td>
<td>165.460</td>
<td>241.201</td>
<td>186.593</td>
<td>23.519</td>
</tr>
<tr>
<td></td>
<td>test_multiply</td>
<td>202.169</td>
<td>327.162</td>
<td>231.310</td>
<td>30.287</td>
</tr>
<tr>
<td></td>
<td>test_power</td>
<td>361.526</td>
<td>447.006</td>
<td>385.759</td>
<td>28.706</td>
</tr>
<tr>
<td></td>
<td>test_scalar_multiply</td>
<td>850.281</td>
<td>1,128.30</td>
<td>884.649</td>
<td>84.649</td>
</tr>
<tr>
<td></td>
<td>test_subtract</td>
<td>16.040</td>
<td>83.546</td>
<td>18.269</td>
<td>4.490</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Name</th>
<th>Min</th>
<th>Max</th>
<th>Mean</th>
<th>StdDev</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>test_add</td>
<td>876.931</td>
<td>1,635.9</td>
<td>936.241</td>
<td>163.584</td>
</tr>
<tr>
<td></td>
<td>test_additive_inverse</td>
<td>557.644</td>
<td>1,945.0</td>
<td>632.242</td>
<td>258.070</td>
</tr>
<tr>
<td></td>
<td>test_divide</td>
<td>756.546</td>
<td>1,014.9</td>
<td>792.178</td>
<td>79.217</td>
</tr>
<tr>
<td></td>
<td>test_multiplicative_inverse</td>
<td>82,011</td>
<td>83,817</td>
<td>82,897</td>
<td>82.897</td>
</tr>
<tr>
<td></td>
<td>test_multiply</td>
<td>82,011</td>
<td>83,817</td>
<td>82,897</td>
<td>82.897</td>
</tr>
<tr>
<td></td>
<td>test_power</td>
<td>82,011</td>
<td>83,817</td>
<td>82,897</td>
<td>82.897</td>
</tr>
<tr>
<td></td>
<td>test_scalar_multiply</td>
<td>82,011</td>
<td>83,817</td>
<td>82,897</td>
<td>82.897</td>
</tr>
<tr>
<td></td>
<td>test_subtract</td>
<td>82,011</td>
<td>83,817</td>
<td>82,897</td>
<td>82.897</td>
</tr>
</tbody>
</table>
### 3.13.3 Compare with a previous benchmark

If you would like to compare the performance impact of a branch, first run a benchmark on master using the `--benchmark-save` option. This will save the file `.benchmarks/0001_master.json`.

```
$ git checkout master
$ python3 -m pytest benchmarks/test_field_arithmetic.py --benchmark-save=master --
```

Next, run a benchmark on the branch under test while comparing against the benchmark from master.

```
$ git checkout branch
$ python3 -m pytest benchmarks/test_field_arithmetic.py --benchmark-compare=0001_master --
```

Or, save a benchmark run from branch and compare it explicitly against the one from master. This benchmark run will save the file `.benchmarks/0001_branch.json`.

```
$ git checkout branch
$ python3 -m pytest benchmarks/test_field_arithmetic.py --benchmark-save=branch --
$ python3 -m pytest-benchmark compare 0001_master 0001_branch
```

### 3.14 Installation

#### 3.14.1 Install from PyPI

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

```
$ python3 -m pip install galois
```
3.14.2 Install from GitHub

The latest code on master can be installed using pip in this way.

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

Or from a specific branch.

```
$ python3 -m pip install git+https://github.com/mhostetter/galois.git@branch
```

3.14.3 Editable install from local folder

To actively develop the library, it is beneficial to pip install the library in an editable fashion from a local folder. This allows any changes in the current directory to be immediately seen upon the next `import galois`.

Clone the repo wherever you’d like.

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

Install the local folder using the -e or --editable flag.

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

3.14.4 Install the dev dependencies

The development dependencies include packages for linting and unit testing. These dependencies are stored in `requirements-dev.txt`.

```
1  pylint>=2.14
2  pytest
3  pytest-cov
4  pytest-benchmark
```

Install the development dependencies by passing the -r switch to pip install.

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

3.15 Linter

The galois library uses `pylint` for static analysis and code formatting.
3.15.1 Install

First, pylint needs to be installed on your system. Easily install it by installing the development dependencies.

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

3.15.2 Configuration

Various nuisance pylint warnings are added to an ignore list in `setup.cfg`.

```
[pylint]
disable =
    line-too-long,
    too-many-lines,
    # ...
```

3.15.3 Run from the command line

Run the linter manually from the command line by passing in the `setup.cfg` file as the `pylint` configuration file.

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

3.15.4 Run from VS Code

Included is a VS Code configuration file `.vscode/settings.json`. This instructs VS Code about how to invoke pylint. VS Code will run the linter as you view and edit files.

3.16 Unit Tests

The `galois` library uses `pytest` for unit testing.

3.16.1 Install

First, pytest needs to be installed on your system. Easily install it by installing the development dependencies.

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

3.16.2 Run from the command line

Execute all of the unit tests manually from the command line.

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

Or only run a specific test file.

```
$ python3 -m pytest tests/test_math.py
```
Or only run a specific unit test.

$ python3 -m pytest tests/test_math.py::test_gcd

### 3.16.3 Run from VS Code

Included is a VS Code configuration file `.vscode/settings.json`. This instructs VS Code about how to invoke pytest. VS Code’s integrated test infrastructure will locate the tests and allow you to run or debug any test.

### 3.16.4 Test vectors

Test vectors are generated by third-party tools and stored in `.pkl` files. Most test vectors are stored in these folders:

- `tests/data/`
- `tests/fields/data/GF(*)/`
- `tests/polys/data/GF(*)/`

The scripts that generate the test vectors are:

- `scripts/generate_int_test_vectors.py`
- `scripts/generate_field_test_vectors.py`

The two primary third-party tools are Sage and SymPy.

#### Install Sage

$ sudo apt install sagemath

#### Install SymPy

$ python3 -m pip install sympy

#### Generate test vectors

To re-generate the test vectors locally, run:

$ python3 scripts/generate_int_test_vectors.py
$ python3 scripts/generate_field_test_vectors.py

The scripts use random number generator seeds to generate reproducible test vectors. To generate different test vectors, modify the seeds. It’s also easy to increase the number of test cases for any individual test.
3.17 Documentation

The `galois` documentation is generated with Sphinx and the Sphinx Immaterial theme.

3.17.1 Install

The documentation dependencies are stored in `docs/requirements.txt`.

```
sphinx==5
git+https://github.com/jbms/sphinx-immaterial@05e70669e8f42bdd1dafa87c2a5f1879b7ba73d25a
myst-parser
sphinx-design
sphinxcontrib-details-directive
ipykernel
numpy
```

Install the documentation dependencies by passing the `-r` switch to `pip install`.

```
$ python3 -m pip install -r docs/requirements.txt
```

3.17.2 Build the docs

The documentation is built by running the `sphinx-build` command.

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

The HTML output is located in `docs/build/`. The home page is `docs/build/index.html`.

3.18 galois

3.18.1 Arrays

```
class galois.Array(numpy.ndarray)
    An abstract ndarray subclass over a Galois field or Galois ring.

galois.typing.ArrayLike
    A Union representing objects that can be coerced into a Galois field array.

galois.typing.DTypeLike
    A Union representing objects that can be coerced into a NumPy data type.

galois.typing.ElementLike
    A Union representing objects that can be coerced into a Galois field element.

galois.typing.IterableLike
    A Union representing iterable objects that can be coerced into a Galois field array.

galois.typing.ShapeLike
    A Union representing objects that can be coerced into a NumPy shape tuple.
```
class `galois.Array`(
    `numpy.ndarray`
)
An abstract `ndarray` subclass over a Galois field or Galois ring.

**Important:** `Array` is an abstract base class for `FieldArray` and cannot be instantiated directly.

### Constructors

**classmethod** `Identity`(`size: int, ...`) → `Array`
Creates an $n \times n$ identity matrix.

**classmethod** `Ones`(`shape: ShapeLike, ...`) → `Array`
Creates an array of all ones.

**classmethod** `Random`(`shape: ShapeLike = (), low, high`) → `Array`
Creates an array with random elements.

**classmethod** `Range`(`start: ElementLike, stop, ...`) → `Array`
Creates a 1-D array with a range of elements.

**classmethod** `Zeros`(`shape: ShapeLike, ...`) → `Array`
Creates an array of all zeros.

**classmethod** `galois.Array.Identity`(`size: int, dtype: DTypeLike | None = None`) → `Array`
Creates an $n \times n$ identity matrix.

**Parameters**

- **size:** `int`
  The size $n$ along one dimension of the identity matrix.

- **dtype:** `DTypeLike | None = None`
  The `numpy.dtype` of the array elements. The default is `None` which represents the smallest unsigned data type for this `Array` subclass (the first element in `dtypes`).

**Returns**

A 2-D identity matrix with shape `(size, size)`.

**classmethod** `galois.Array.Ones`(`shape: ShapeLike, dtype: DTypeLike | None = None`) → `Array`
Creates an array of all ones.

**Parameters**

- **shape:** `ShapeLike`
  A NumPy-compliant shape tuple.

- **dtype:** `DTypeLike | None = None`
  The `numpy.dtype` of the array elements. The default is `None` which represents the smallest unsigned data type for this `Array` subclass (the first element in `dtypes`).

**Returns**

An array of ones.

**classmethod** `galois.Array.Random`(`shape: ShapeLike = (), low: ElementLike = 0, high: ElementLike | None = None, seed: int | Generator | None = None, dtype: DTypeLike | None = None`) → `Array`
Creates an array with random elements.
Parameters

shape: ShapeLike = ()
A NumPy-compliant shape tuple. The default is () which represents a scalar.

low: ElementLike = 0
The smallest element (inclusive). The default is 0.

high: ElementLike | None = None
The largest element (exclusive). The default is None which represents order.

seed: int | Generator | None = None
Non-negative integer used to initialize the PRNG. The default is None which means that unpredictable entropy will be pulled from the OS to be used as the seed. A numpy.random.Generator can also be passed.

dtype: DtypeLike | None = None
The numpy.dtype of the array elements. The default is None which represents the smallest unsigned data type for this Array subclass (the first element in dtypes).

Returns
An array of random elements.

classmethod galois.Array.Range(start: ElementLike, stop: ElementLike, step: int = 1, dtype: DtypeLike | None = None) → Array

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

Parameters

start: ElementLike
The starting element (inclusive).

stop: ElementLike
The stopping element (exclusive).

step: int = 1
The increment between elements. The default is 1.

dtype: DtypeLike | None = None
The numpy.dtype of the array elements. The default is None which represents the smallest unsigned data type for this Array subclass (the first element in dtypes).

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

classmethod galois.Array.Zeros(shape: ShapeLike, dtype: DtypeLike | None = None) → Array

Creates an array of all zeros.

Parameters

shape: ShapeLike
A NumPy-compliant shape tuple.

dtype: DtypeLike | None = None
The numpy.dtype of the array elements. The default is None which represents the smallest unsigned data type for this Array subclass (the first element in dtypes).

Returns
An array of zeros.
Methods

**classmethod compile(mode)**
Recompile the just-in-time compiled ufuncs for a new calculation mode.

**classmethod display(...) → Generator[None, None, None]**
Sets the display mode for all arrays from this FieldArray subclass.

**classmethod galois.Array.compile(mode: 'auto' | 'jit-lookup' | 'jit-calculate' | 'python-calculate')**
Recompile the just-in-time compiled ufuncs for a new calculation mode.

This function updates `ufunc_mode`.

**Parameters**

**mode: 'auto' | 'jit-lookup' | 'jit-calculate' | 'python-calculate'**
The ufunc calculation mode.

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

**classmethod galois.Array.display(mode: 'int' | 'poly' | 'power' = 'int') → Generator[None, None, None]**
Sets the display mode for all arrays from this FieldArray subclass.

The display mode can be set to either the integer representation, polynomial representation, or power representation. See Element Representation for a further discussion.

This function updates `display_mode`.

**Warning:** For the power representation, `numpy.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: 'int' | 'poly' | 'power' = 'int'**
The field element representation.

- "int": Sets the display mode to the integer representation.
- "poly": Sets the display mode to the polynomial representation.
- "power": Sets the display mode to the power representation.
Returns

A context manager for use in a `with` statement. If permanently setting the display mode, disregard the return value.

Examples

The default display mode is the integer representation.

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

Permanently set the display mode by calling `display()`.

**Polynomial**

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

**Power**

```
In [5]: GF.display("power");
In [6]: x
Out[6]: GF([ 0, 1, ^4, , ^2, ^7, ^5, ^3, ^6], order=3^2)
```

Temporarily modify the display mode by using `display()` as a context manager.

**Polynomial**

```
In [7]: print(x)
[0, 1, 2, 3, 4, 5, 6, 7, 8]
In [8]: with GF.display("poly"): ...
   print(x)
   ...
[ 0, 1, 2, , +1, +2, 2, 2 +1, 2 +2]
# Outside the context manager, the display mode reverts to its previous value
In [9]: print(x)
[0, 1, 2, 3, 4, 5, 6, 7, 8]
```
Power

```
In [10]: print(x)
[0, 1, 2, 3, 4, 5, 6, 7, 8]

In [11]: with GF.display("power"):
    ....:    print(x)
    ....:
[ 0, 1, ^4, , ^2, ^7, ^5, ^3, ^6]

# Outside the context manager, the display mode reverts to its previous value
In [12]: print(x)
[0, 1, 2, 3, 4, 5, 6, 7, 8]
```

Properties

class property characteristic: int
   The characteristic \( p \) of the Galois field \( GF(p^m) \) or \( p^n \) of the Galois ring \( GR(p^n, m) \).

class property default_ufunc_mode: 'jit-lookup' | 'jit-calculate' | 'python-calculate'
   The default compilation mode of the Galois field or Galois ring.

class property degree: int
   The degree \( m \) of the Galois field \( GF(p^m) \) or Galois ring \( GR(p^n, m) \).

class property display_mode: 'int' | 'poly' | 'power'
   The current element representation of the Galois field or Galois ring.

class property dtypes: List[dtype]
   List of valid integer numpy.dtype values that are compatible with this Galois field or Galois ring.

class property elements: Array
   All elements of the Galois field or Galois ring.

class property irreducible_poly: Poly
   The irreducible polynomial of the Galois field or Galois ring.

class property name: str
   The name of the Galois field or Galois ring.

class property order: int
   The order \( p^m \) of the Galois field \( GF(p^m) \) or \( p^n \) of the Galois ring \( GR(p^n, m) \).

class property primitive_element: Array
   A primitive element of the Galois field or Galois ring.

class property ufunc_mode: 'jit-lookup' | 'jit-calculate' | 'python-calculate'
   The current compilation mode of the Galois field or Galois ring.

class property ufunc_modes: List[str]
   All supported compilation modes of the Galois field or Galois ring.

class property units: Array
   All units of the Galois field or Galois ring.
class property galois.Array.characteristic : int
The characteristic $p$ of the Galois field $GF(p^m)$ or $p^e$ of the Galois ring $GR(p^e, m)$.

class property galois.Array.default_ufunc_mode : ['jit-lookup', 'jit-calculate', 'python-calculate']
The default compilation mode of the Galois field or Galois ring.

class property galois.Array.degree : int
The degree $m$ of the Galois field $GF(p^m)$ or Galois ring $GR(p^e, m)$.

class property galois.Array.display_mode : ['int', 'poly', 'power']
The current element representation of the Galois field or Galois ring.

class property galois.Array.dtypes : List[dtype]
List of valid integer numpy.dtype values that are compatible with this Galois field or Galois ring.

class property galois.Array.elements : Array
All elements of the Galois field or Galois ring.

class property galois.Array.irreducible_poly : Poly
The irreducible polynomial of the Galois field or Galois ring.

class property galois.Array.name : str
The name of the Galois field or Galois ring.

class property galois.Array.order : int
The order $p^m$ of the Galois field $GF(p^m)$ or $p^e^m$ of the Galois ring $GR(p^e, m)$.

class property galois.Array.primitive_element : Array
A primitive element of the Galois field or Galois ring.

class property galois.Array.ufunc_mode : ['jit-lookup', 'jit-calculate', 'python-calculate']
The current compilation mode of the Galois field or Galois ring.

class property galois.Array.ufunc_modes : List[str]
All supported compilation modes of the Galois field or Galois ring.

class property galois.Array.units : Array
All units of the Galois field or Galois ring.

galois.typing.ArrayLike
A Union representing objects that can be coerced into a Galois field array.

Union

- IterableLike: A recursive iterable of iterables of elements.

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

# Mix and match integers and strings
In [3]: GF([["x^2 + 2x + 2", 4], ["x^4 + 2x^3 + x^2 + x + 1", 205]])
Out[3]:
```
(continues on next page)
• **ndarray**: A NumPy array of integers, representing finite field elements in their *integer representation*.

```python
In [4]: x = np.array([[17, 4], [148, 205]]); x
Out[4]:
array([[ 17,  4],
       [148, 205]])

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

• **Array**: A previously-created *Array* object. No coercion is necessary.

### Alias

Alias of `Union[Sequence[Union[int, str, Array]], Sequence[IterableLike], ndarray, Array]`

galois.typing.DTypeLike

A *Union* representing objects that can be coerced into a NumPy data type.

### Union

• **numpy.integer**: A fixed-width NumPy integer data type.

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

In [2]: x = GF.Random(4, dtype=np.uint16); x.dtype
Out[2]:
dtype('uint16')

In [3]: x = GF.Random(4, dtype=np.int32); x.dtype
Out[3]:
dtype('int32')
```

• **int**: The system default integer.

```python
In [4]: x = GF.Random(4, dtype=int); x.dtype
Out[4]:
dtype('int64')
```

• **str**: The string that can be coerced with *numpy.dtype*.

```python
In [5]: x = GF.Random(4, dtype="uint16"); x.dtype
Out[5]:
dtype('uint16')

In [6]: x = GF.Random(4, dtype="int32"); x.dtype
Out[6]:
dtype('int32')
```

• **object**: A Python object data type. This applies to non-compiled fields.
In [7]: GF = galois.GF(2**100)

In [8]: x = GF.Random(4, dtype=object); x.dtype
Out[8]: dtype('O')

Alias

alias of Union[integer, int, str, object]
galois.typing.ElementLike
A Union representing objects that can be coerced into a Galois field element.
Scalars are 0-D Array objects.

Union

- int: A finite field element in its integer representation.

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

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

- str: A finite field element in its polynomial representation. Many string conventions are accepted, including: with/without *, with/without spaces, ^ or **, any indeterminate variable, increasing/decreasing degrees, etc. Or any combination of the above.

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

# Add explicit * for multiplication
In [4]: GF("x^2 + 2*x + 2")
Out[4]: GF(17, order=3^5)

# No spaces
In [5]: GF("x^2+2x+2")
Out[5]: GF(17, order=3^5)

# ** instead of ^
In [6]: GF("x**2 + 2x + 2")
Out[6]: GF(17, order=3^5)

# Different indeterminate
In [7]: GF("^2 + 2 + 2")
Out[7]: GF(17, order=3^5)

# Ascending degrees
In [8]: GF("2 + 2x + x^2")
Out[8]: GF(17, order=3^5)

- Array: A previously-created scalar Array object. No coercion is necessary.
**Galois**

**Alias**

alias of `Union[int, str, Array]`

galois.typing.IterableLike

A `Union` representing iterable objects that can be coerced into a Galois field array.

**Union**

- **Sequence** `[ElementLike]`: An iterable of elements.

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

In [2]: GF([17, 4])
Out[2]: GF([17, 4], order=3^5)

# Mix and match integers and strings
In [3]: GF([17, "x + 1"])
Out[3]: GF([17, 4], order=3^5)
```

- **Sequence** `[IterableLike]`: A recursive iterable of iterables of elements.

```
In [4]: GF = galois.GF(3**5)

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

# Mix and match integers and strings
In [6]: GF(["x^2 + 2x + 2", 4], ["x^4 + 2x^3 + x^2 + x + 1", 205]])
Out[6]: GF([[17, 4],
          [148, 205]], order=3^5)
```

**Alias**

alias of `Union[Sequence[Union[int, str, Array]], Sequence[IterableLike]]`

galois.typing.ShapeLike

A `Union` representing objects that can be coerced into a NumPy `shape` tuple.

**Union**

- **int**: The size of a 1-D array.

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

In [2]: x = GF.Random(4); x
Out[2]: GF([239, 86, 174, 21], order=3^5)
```
In [3]: x.shape
Out[3]: (4,)

- **Sequence[int]**: An iterable of integer dimensions. Tuples or lists are allowed. An empty iterable, () or [], represents a 0-D array (scalar).

In [4]: x = GF.Random((2, 3)); x
Out[4]:
GF([[177, 61, 231],
    [ 35, 145, 110]], order=3^5)

In [5]: x.shape
Out[5]: (2, 3)

In [6]: x = GF.Random([2, 3, 4]); x
Out[6]:
GF([[[ 39, 239, 203, 190],
       [112, 161, 158, 242],
       [ 30, 228, 154,  95]],
     [[[ 69, 127, 207, 206],
       [ 10, 193, 41, 124],
       [212, 169, 6,  19]], order=3^5])

In [7]: x.shape
Out[7]: (2, 3, 4)

In [8]: x = GF.Random(()); x
Out[8]: GF(134, order=3^5)

In [9]: x.shape
Out[9]: ()

**Alias**

alias of Union[int, Sequence[int]]

### 3.18.2 Galois fields

class galois.FieldArray(galois.Array)
An abstract ndarray subclass over GF($p^m$).

class galois.GF2(galois.FieldArray)
A FieldArray subclass over GF(2).
galois.Field(order: int, ...) \rightarrow Type[FieldArray]
Alias of GF().
galois.GF(order: int, ...) \rightarrow Type[FieldArray]
Creates a FieldArray subclass for GF($p^m$).
class galois.FieldArray(galois.Array)
    An abstract ndarray subclass over GF($p^m$).

Important: FieldArray is an abstract base class and cannot be instantiated directly. Instead, FieldArray subclasses are created using the class factory GF().

Examples

Create a FieldArray subclass over GF($3^5$) using the class factory GF().

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

Create a FieldArray instance using GF's constructor.

```
In [4]: x = GF([44, 236, 206, 138]); x
Out[4]: GF([ 44, 236, 206, 138], order=3^5)
In [5]: isinstance(x, GF)
Out[5]: True
```

Constructors

FieldArray(x: ElementLike | ArrayLike, ...)
    Creates an array over GF($p^m$).

classmethod Identity(size: int, ...) → FieldArray
    Creates an $n \times n$ identity matrix.

classmethod Ones(shape: ShapeLike, ...) → FieldArray
    Creates an array of all ones.

classmethod Random(shape: ShapeLike = (), ...) → FieldArray
    Creates an array with random elements.

classmethod Range(start: ElementLike, stop, ...) → FieldArray
    Creates a 1-D array with a range of elements.
classmethod Vandermonde(element: ElementLike, ...) → FieldArray

Creates an \( m \times n \) Vandermonde matrix of \( a \in GF(q) \).

classmethod Vector(array: ArrayLike, ...) → FieldArray

Creates an array over \( GF(p^m) \) from length-\( m \) vectors over the prime subfield \( GF(p) \).

classmethod Zeros(shape: ShapeLike, ...) → FieldArray

Creates an array of all zeros.

galois.FieldArray(x: ElementLike | ArrayLike, dtype: DTypeLike | None = None, copy: bool = True, order: 'K' | 'A' | 'C' | 'F' = 'K', ndmin: int = 0)

Creates an array over \( GF(p^m) \).

Parameters

- x: ElementLike | ArrayLike
  A finite field scalar or array.

- dtype: DTypeLike | None = None
  The numpy.dtype of the array elements. The default is None which represents the smallest unsigned data type for this FieldArray subclass (the first element in dtypes).

- copy: bool = True
  The copy keyword argument from numpy.array(). The default is True.

- order: 'K' | 'A' | 'C' | 'F' = 'K'
  The order keyword argument from numpy.array(). The default is "K".

- ndmin: int = 0
  The ndmin keyword argument from numpy.array(). The default is 0.

classmethod galois.FieldArray.Identity(size: int, dtype: DTypeLike | None = None) → FieldArray

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

Parameters

- size: int
  The size \( n \) along one dimension of the identity matrix.

- dtype: DTypeLike | None = None
  The numpy.dtype of the array elements. The default is None which represents the smallest unsigned data type for this FieldArray subclass (the first element in dtypes).

Returns

A 2-D identity matrix with shape (size, size).

Examples

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

In [2]: GF.Identity(4)
Out[2]:
GF([[1, 0, 0, 0],
    [0, 1, 0, 0],
    [0, 0, 1, 0],
    [0, 0, 0, 1]], order=31)
```
classmethod  galois.FieldArray.ones(shape: ShapeLike, dtype: DTypeLike | None = None) → FieldArray

Creates an array of all ones.

Parameters

shape: ShapeLike
A NumPy-compliant shape tuple.

dtype: DTypeLike | None = None
The numpy.dtype of the array elements. The default is None which represents the smallest unsigned data type for this FieldArray subclass (the first element in dtypes).

Returns
An array of ones.

Examples

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

classmethod  galois.FieldArray.random(shape: ShapeLike = (), low: ElementLike = 0, high: ElementLike | None = None, seed: int | Generator | None = None, dtype: DTypeLike | None = None) → FieldArray

Creates an array with random elements.

Parameters

shape: ShapeLike = ()
A NumPy-compliant shape tuple. The default is () which represents a scalar.

low: ElementLike = 0
The smallest element (inclusive). The default is 0.

high: ElementLike | None = None
The largest element (exclusive). The default is None which represents order.

seed: int | Generator | None = None
Non-negative integer used to initialize the PRNG. The default is None which means that unpredictable entropy will be pulled from the OS to be used as the seed. A numpy.random.Generator can also be passed.

dtype: DTypeLike | None = None
The numpy.dtype of the array elements. The default is None which represents the smallest unsigned data type for this FieldArray subclass (the first element in dtypes).

Returns
An array of random elements.
Examples

Generate a random matrix with an unpredictable seed.

```
In [1]: GF = galois.GF(31)
In [2]: GF.Random((2, 5))
Out[2]:
GF([[25, 12, 22, 30, 8],
     [12, 11, 14, 16, 0]], order=31)
```

Generate a random array with a specified seed. This produces repeatable outputs.

```
In [3]: GF.Random(10, seed=123456789)
Out[3]: GF([ 7, 29, 20, 27, 18, 5, 2, 0, 24, 24], order=31)
In [4]: GF.Random(10, seed=123456789)
Out[4]: GF([ 7, 29, 20, 27, 18, 5, 2, 0, 24, 24], order=31)
```

Generate a group of random arrays using a single global seed.

```
In [5]: rng = np.random.default_rng(123456789)
In [6]: GF.Random(10, seed=rng)
Out[6]: GF([ 7, 29, 20, 27, 18, 5, 2, 0, 24, 24], order=31)
In [7]: GF.Random(10, seed=rng)
Out[7]: GF([20, 15, 3, 28, 22, 0, 5, 10, 1, 0], order=31)
```

classmethod galois.FieldArray.Range(start: ElementLike, stop: ElementLike, step: int = 1, dtype: DTypeLike | None = None) → FieldArray

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

Parameters

- **start**: `ElementLike`
  The starting element (inclusive).

- **stop**: `ElementLike`
  The stopping element (exclusive).

- **step**: `int = 1`
  The increment between elements. The default is 1.

- **dtype**: `DTypeLike | None = None`
  The `numpy.dtype` of the array elements. The default is `None` which represents the smallest unsigned data type for this `FieldArray` subclass (the first element in `dtypes`).

Returns

A 1-D array of a range of elements.
Examples

For prime fields, the increment is simply a finite field element, since all elements are integers.

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

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

In [3]: GF.Range(10, 20, 2)
Out[3]: GF([10, 12, 14, 16, 18], order=31)
```

For extension fields, the increment is the integer increment between finite field elements in their integer representation.

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

In [5]: GF.Range(10, 20)
Out[5]: GF([\texttt{^2} + 1, \texttt{^2} + 2, \texttt{^2} + , \texttt{^2} + + 1, \texttt{^2} + + 2,
                \texttt{^2} + 2, \texttt{^2} + 2 + 1, \texttt{^2} + 2 + 2, \texttt{2^2}, \texttt{2^2} + 1],
                \text{order=3^3})

In [6]: GF.Range(10, 20, 2)
Out[6]: GF([\texttt{^2} + 1, \texttt{^2} + , \texttt{^2} + + 2, \texttt{^2} + 2 + 1, \texttt{2^2}],
                \text{order=3^3})
```

classmethod galois.FieldArray.Vandermonde(element: ElementLike, rows: int, cols: int, dtype: DTypeLike | None = None) → FieldArray

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

Parameters

- **element**: ElementLike
  An element $a$ of $\text{GF}(q)$.

- **rows**: int
  The number of rows $m$ in the Vandermonde matrix.

- **cols**: int
  The number of columns $n$ in the Vandermonde matrix.

- **dtype**: DTypeLike | None = None
  The numpy dtype of the array elements. The default is None which represents the smallest unsigned data type for this FieldArray subclass (the first element in dtypes).

Returns

A $m \times n$ Vandermonde matrix.
Examples

```plaintext
In [1]: GF = galois.GF(2**3, display="power")

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

In [3]: V = GF.Vandermonde(a, 7, 7); V
Out[3]: GF([[ 1, 1, 1, 1, 1, 1, 1],
         [ 1, ^2, ^3, ^4, ^5, ^6, ],
         [ 1, ^2, ^4, ^6, ^3, ^5 ],
         [ 1, ^3, ^6, ^2, ^5, ^4 ],
         [ 1, ^4, ^5, ^2, ^6, ^3 ],
         [ 1, ^5, ^3, ^6, ^4, ^2 ],
         [ 1, ^6, ^5, ^4, ^3, ^2 ]], order=2^3)
```

classmethod `galois.FieldArray.Vector`(array: ArrayLike, dtype: DTypeLike | None = None) → FieldArray

Creates an 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: ArrayLike
   An array over GF($p$) with last dimension $m$. An array with shape (n1, n2, m) has output
   shape (n1, n2). By convention, the vectors are ordered from degree $m - 1$ to degree 0.

dtype: DTypeLike | None = None
   The numpy.dtype of the array elements. The default is None which represents the smallest
   unsigned data type for this FieldArray subclass (the first element in dtypes).

Returns
   An array over GF($p^m$).

Examples

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

In [2]: a = GF.Vector([[1, 0, 2], [0, 2, 1]]); a
Out[2]: GF([2 + 2, 2 + 1], order=3^3)

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

classmethod `galois.FieldArray.Zeros`(shape: ShapeLike, dtype: DTypeLike | None = None) → FieldArray

Creates an array of all zeros.

Parameters

shape: ShapeLike
   A NumPy-compliant shape tuple.
**dtype: DTypeLike | None = None**

The numpy `dtype` of the array elements. The default is `None` which represents the smallest unsigned data type for this `FieldArray` subclass (the first element in `dtypes`).

**Returns**

An array of zeros.

**Examples**

```python
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)
```

**String representation**

```python
__repr__() → str
```

Displays the array specifying the class and finite field order.

```python
__str__() → str
```

Displays the array without specifying the class or finite field order.

```python
class property properties : str
```

A formatted string of relevant properties of the Galois field.

```python
galois.FieldArray.__repr__() → str
```

Displays the array specifying the class and finite field order.

This function prepends `GF( and appends ` order=p^m`).

**Examples**

**Integer**

```python
In [1]: GF = galois.GF(3**2)
In [2]: x = GF([4, 2, 7, 5])
In [3]: x
Out[3]: GF([4, 2, 7, 5], order=3^2)
```
Polynomial

In [4]: GF = galois.GF(3**2, display="poly")
In [5]: x = GF([4, 2, 7, 5])
In [6]: x  
Out[6]: GF([ + 1, 2, 2 + 1, + 2], order=3^2)

Power

In [7]: GF = galois.GF(3**2, display="power")
In [8]: x = GF([4, 2, 7, 5])
In [9]: x  
Out[9]: GF([^2, ^4, ^3, ^7], order=3^2)

galois.FieldArray.__str__() → str
Displays the array without specifying the class or finite field order.
This function does not prepend GF( and or append , order=p^m).

Examples

Integer

In [1]: GF = galois.GF(3**2)
In [2]: x = GF([4, 2, 7, 5])
In [3]: print(x)
[4, 2, 7, 5]

Polynomial

In [4]: GF = galois.GF(3**2, display="poly")
In [5]: x = GF([4, 2, 7, 5])
In [6]: print(x)
[ + 1, 2, 2 + 1, + 2]
Power

```python
In [7]: GF = galois.GF(3**2, display="power")
In [8]: x = GF([4, 2, 7, 5])
In [9]: print(x)
[^2, ^4, ^3, ^7]
```

class property galois.FieldArray.properties : str
A formatted string of relevant properties of the Galois field.

Examples

```python
In [1]: GF = galois.GF(7**5)
In [2]: print(GF.properties)
Galois Field:
   name: GF(7^5)
   characteristic: 7
   degree: 5
   order: 16807
   irreducible_poly: x^5 + x + 4
   is_primitive_poly: True
   primitive_element: x
```

Methods

additive_order() → integer | ndarray
Computes the additive order of each element in $x$.

classmethod arithmetic_table(operation,...) → str
Generates the specified arithmetic table for the finite field.

characteristic_poly() → Poly
Computes the characteristic polynomial of a finite field element $a$ or a square matrix $A$.

column_space() → FieldArray
Computes the column space of the matrix $A$.

classmethod compile(mode)
Recompile the just-in-time compiled ufuncs for a new calculation mode.

classmethod display(...) → Generator[None, None, None]
Sets the display mode for all arrays from this FieldArray subclass.

field_norm() → FieldArray
Computes the field norm $N_{L/K}(x)$ of the elements of $x$.

field_trace() → FieldArray
Computes the field trace $\text{Tr}_{L/K}(x)$ of the elements of $x$. 

**is_quadratic_residue()** → bool | ndarray
Determines if the elements of \( x \) are quadratic residues in the finite field.

**left_null_space()** → FieldArray
Computes the left null space of the matrix \( A \).

**log** (base: ElementLike | ArrayLike | None = None) → ndarray
Computes the logarithm of the array \( x \) base \( \beta \).

**lu_decompose()** → Tuple[FieldArray, FieldArray]
Decomposes the input array into the product of lower and upper triangular matrices.

**minimal_poly()** → Poly
Computes the minimal polynomial of a finite field element \( a \).

**multiplicative_order()** → integer | ndarray
Computes the multiplicative order \( \text{ord}(x) \) of each element in \( x \).

**null_space()** → FieldArray
Computes the null space of the matrix \( A \).

**plu_decompose()** → Tuple[FieldArray, FieldArray, FieldArray]
Decomposes the input array into the product of lower and upper triangular matrices using partial pivoting.

**classmethod primitive_root_of_unity**(n: int) → FieldArray
Finds a primitive \( n \)-th root of unity in the finite field.

**classmethod primitive_roots_of_unity**(n: int) → FieldArray
Finds all primitive \( n \)-th roots of unity in the finite field.

**classmethod repr_table**(…) → str
Generates a finite field element representation table comparing the power, polynomial, vector, and integer representations.

**row_reduce**(ncols: int | None = None) → FieldArray
Performs Gaussian elimination on the matrix to achieve reduced row echelon form (RREF).

**row_space()** → FieldArray
Computes the row space of the matrix \( A \).

**vector**(dtype: DtypeLike | None = None) → FieldArray
Converts an array over \( \text{GF}(p^m) \) to length-\( m \) vectors over the prime subfield \( \text{GF}(p) \).

**galois.FieldArray.additive_order()** → integer | ndarray
Computes the additive order of each element in \( x \).

**Returns**
An integer array of the additive order of each element in \( x \). The return value is a single integer if the input array \( x \) is a scalar.
Notes

The additive order $a$ of $x$ in $GF(p^m)$ is the smallest integer $a$ such that $xa = 0$. With the exception of 0, the additive order of every element is the finite field’s characteristic.

Examples

Compute the additive order of each element of $GF(3^2)$.

```python
In [1]: GF = galois.GF(3**2, display="poly")
In [2]: x = GF.elements; x
Out[2]: GF([ 0, 1, 2, 1, 2, 2, 2, 2, 2], order=3^2)
In [3]: order = x.additive_order(); order
Out[3]: array([1, 3, 3, 3, 3, 3, 3, 3, 3])
In [4]: x * order
Out[4]: GF([0, 0, 0, 0, 0, 0, 0, 0, 0], order=3^2)
```

classmethod galois.FieldArray.arithmetic_table(operation: '+', '-', '*', '/', x: FieldArray | None = None, y: FieldArray | None = None) → str

Generates the specified arithmetic table for the finite field.

Parameters

- **operation**: '+', '-', '*', '/'
  - The arithmetic operation.
- **x**: `FieldArray | None = None`
  - Optionally specify the $x$ values for the arithmetic table. The default is `None` which represents $\{0, \ldots, p^m - 1\}$.
- **y**: `FieldArray | None = None`
  - Optionally specify the $y$ values for the arithmetic table. The default is `None` which represents $\{0, \ldots, p^m - 1\}$ for addition, subtraction, and multiplication and $\{1, \ldots, p^m - 1\}$ for division.

Returns

- A string representation of the arithmetic table.

Examples

Arithmetic tables can be displayed using any element representation.
Galois Integer

```python
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
```

Galois Polynomial

```python
In [3]: GF = galois.GF(3**2, display="poly")
In [4]: print(GF.arithmetic_table("+"))
x + y | 0 1 2 + 1 + 2 2 2 + 1 2 + 2
-------|------------------------------------------------------------------------
0 | 0 1 2 + 1 + 2 2 2 + 1 2 + 2
1 | 1 2 0 + 1 + 2 2 + 1 2 + 2 2
2 | 2 0 1 + 1 + 2 2 + 2 + 2 2 + 2 + 1
1 + 1 + 2 2 2 + 1 2 + 2 0 1 2
+ 1 | + 1 + 2 2 + 1 2 + 2 2 1 2 0
+ 2 | + 2 + 1 2 + 2 2 2 + 1 2 0 1
2 | 2 2 + 1 2 + 2 0 1 2 + 1 2 2 + 2 + 1 2 + 2
2 + 1 | 2 + 1 2 + 2 2 1 2 0 + 1 + 2
2 + 2 | 2 + 2 2 2 + 1 2 0 1 + 2 + 1
```

Galois Power

```python
In [5]: GF = galois.GF(3**2, display="power")
In [6]: print(GF.arithmetic_table("+"))
x + y | 0 1 ^2 ^3 ^4 ^5 ^6 ^7
------|---------------------------------------------
0 | 0 1 ^2 ^3 ^4 ^5 ^6 ^7
1 | 1 ^4 ^2 ^7 ^6 0 ^3 ^5
| ^2 ^5 ^3 1 ^7 0 ^4 ^6
^2 | ^2 ^7 ^3 ^6 ^4 1 0 ^5
^3 | ^3 ^6 1 ^4 ^7 ^5 ^2 0
^4 | ^4 0 ^7 ^5 1 ^6 ^3 ^2
^5 | ^5 ^3 0 1 ^2 ^6 ^7 ^4
^6 | ^6 ^5 ^4 0 ^3 ^7 ^2 1
^7 | ^7 ^6 ^5 0 ^2 ^4 1 ^3
```
An arithmetic table may also be constructed from arbitrary $x$ and $y$.

**Integer**

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

In [8]: x = GF([7, 2, 8]); x
Out[8]: GF([7, 2, 8], order=3^2)

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

In [10]: print(GF.arithmetic_table("+", x=x, y=y))
x + y | 1 4 5 3
------|------------
7     | 8 2 0 1
2     | 0 3 4 5
8     | 6 0 1 2
```

**Polynomial**

```python
In [11]: GF = galois.GF(3**2, display="poly")

In [12]: x = GF([7, 2, 8]); x
Out[12]: GF([2 + 1, 2, 2 + 2], order=3^2)

In [13]: y = GF([1, 4, 5, 3]); y
Out[13]: GF([1, + 1, + 2], order=3^2)

In [14]: print(GF.arithmetic_table("+", x=x, y=y))
x + y | 1 + 1 + 2
------|-----------------------------
2 + 1 | 2 + 2 2 0 1
2     | 0 + 1 + 2
2 + 2 | 2 0 1 2
```

**Power**

```python
In [15]: GF = galois.GF(3**2, display="power")

In [16]: x = GF([7, 2, 8]); x
Out[16]: GF([^3, ^4, ^6], order=3^2)

In [17]: y = GF([1, 4, 5, 3]); y
Out[17]: GF([1, ^2, ^7, ], order=3^2)

In [18]: print(GF.arithmetic_table("+", x=x, y=y))
x + y | 1 ^2 ^7
------|---------------------
(continues on next page)
galois.FieldArray.characteristic_poly() → Poly
Computes the characteristic polynomial of a finite field element \( a \) or a square matrix \( A \).

**Important:** This function may only be invoked on a single finite field element (scalar 0-D array) or a square \( n \times n \) matrix (2-D array).

**Returns**
For scalar inputs, the degree-\( m \) characteristic polynomial \( c_a(x) \) of \( a \) over \( \text{GF}(p) \). For square \( n \times n \) matrix inputs, the degree-\( n \) characteristic polynomial \( c_A(x) \) of \( A \) over \( \text{GF}(p^m) \).

**Notes**
An element \( a \) of \( \text{GF}(p^m) \) has characteristic polynomial \( c_a(x) \) over \( \text{GF}(p) \). The characteristic polynomial when evaluated in \( \text{GF}(p^m) \) annihilates \( a \), that is \( c_a(a) = 0 \). In prime fields \( \text{GF}(p) \), the characteristic polynomial of \( a \) is simply \( c_a(x) = x - a \).

An \( n \times n \) matrix \( A \) has characteristic polynomial \( c_A(x) = \det(x I - A) \) over \( \text{GF}(p^m) \). The constant coefficient of the characteristic polynomial is \( \det(-A) \). The \( x^{n-1} \) coefficient of the characteristic polynomial is \( -\text{Tr}(A) \). The characteristic polynomial annihilates \( A \), that is \( c_A(A) = 0 \).

**References**

**Examples**
The characteristic polynomial of the element \( a \).

```python
In [1]: GF = galois.GF(3**5)
In [2]: a = GF.Random(); a
Out[2]: GF(180, order=3^5)
In [3]: poly = a.characteristic_poly(); poly
Out[3]: Poly(x^5 + x^4 + x^3 + 2x^2 + x + 1, GF(3))
```

# The characteristic polynomial annihilates \( a 
In [4]: poly(a, field=GF)
Out[4]: GF(0, order=3^5)
```

The characteristic polynomial of the square matrix \( A \).

```python
In [5]: GF = galois.GF(3**5)
In [6]: A = GF.Random((3,3)); A
```

(continues on next page)
The column space of the matrix $A$ is defined by all linear combinations of the columns of $A$. The column space has at most dimension $\min(m, n)$.

The column space has properties $\mathcal{C}(A) = \mathcal{R}(A^T)$ and $\dim(\mathcal{C}(A)) + \dim(\mathcal{N}(A)) = n$.

Examples

The column space() method defines basis vectors (its rows) that span the column space of $A$. 

\begin{verbatim}
In [1]: m, n = 3, 5
In [2]: GF = galois.GF(31)
In [3]: A = GF.Random((m, n)); A
Out[3]:
GF([[13, 19, 3, 0, 26],
    [13, 8, 18, 22, 9],
    [5, 5, 1, 29, 1]], order=31)
\end{verbatim}
In [4]: C = A.column_space(); C
Out[4]:
GF([[1, 0, 0],
    [0, 1, 0],
    [0, 0, 1]], order=31)

The dimension of the column space and null space sum to $n$.

In [5]: N = A.null_space(); N
Out[5]:
GF([[ 1, 0, 19, 19, 14],
    [ 0, 1, 14, 28, 6]], order=31)

In [6]: C.shape[0] + N.shape[0] == n
Out[6]: True

\texttt{galois.FieldArray.field_norm()} → FieldArray

\texttt{Computes the field norm $N_{L/K}(x)$ of the elements of $x$.}

\textbf{Returns}

The field norm of $x$ in the prime subfield $GF(p)$.

\textbf{Notes}

The \texttt{self} array $x$ is over the extension field $L = GF(p^m)$. The field norm of $x$ is over the subfield $K = GF(p)$. In other words, $N_{L/K}(x) : L \rightarrow K$.

For finite fields, since $L$ is a Galois extension of $K$, the field norm of $x$ is defined as a product of the Galois conjugates of $x$.

$$N_{L/K}(x) = \prod_{i=0}^{m-1} x^{p^i} = x^{(p^m-1)/(p-1)}$$

\textbf{References}

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

\textbf{Examples}

Compute the field norm of the elements of $GF(3^2)$.

In [1]: GF = galois.GF(3**2, display="poly")

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

In [3]: y = x.field_norm(); y
Out[3]: GF([0, 1, 2, 1, 2, 2, 2, 1], order=3)
galois.FieldArray.field_trace() → FieldArray

Computes the field trace $\text{Tr}_{L/K}(x)$ of the elements of $x$.

Returns

The field trace of $x$ in the prime subfield $\text{GF}(p)$.

Notes

The self array $x$ is over the extension field $L = \text{GF}(p^m)$. The field trace of $x$ is over the subfield $K = \text{GF}(p)$. In other words, $\text{Tr}_{L/K}(x) : L \to K$.

For finite fields, since $L$ is a Galois extension of $K$, the field trace of $x$ is defined as a sum of the Galois conjugates of $x$.

$$\text{Tr}_{L/K}(x) = \sum_{i=0}^{m-1} x^{p^i}$$

References

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

Examples

Compute the field trace of the elements of $\text{GF}(3^2)$.

```
In [1]: GF = galois.GF(3**2, display="poly")

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

In [3]: y = x.field_trace(); y
Out[3]: GF([0, 1, 2, 2, 2, 1, 0, 0], order=3)
```

galois.FieldArray.is_quadratic_residue() → bool | ndarray

Determines if the elements of $x$ are quadratic residues in the finite field.

Returns

A boolean array indicating if each element in $x$ is a quadratic residue. The return value is a single boolean if the input array $x$ is a scalar.

See also:

quadratic_residues, quadratic_non_residues
Notes

An element $x$ in $\text{GF}(p^m)$ is a quadratic residue if there exists a $y$ such that $y^2 = x$ in the field.

In fields with characteristic 2, every element is a quadratic residue. In fields with characteristic greater than 2, exactly half of the nonzero elements are quadratic residues (and they have two unique square roots).

References

- Section 3.5.1 from https://cacr.uwaterloo.ca/hac/about/chap3.pdf.

Examples

Since $\text{GF}(2^3)$ has characteristic 2, every element has a square root.

```python
In [1]: GF = galois.GF(2**3, display="poly")
In [2]: x = GF.elements; x
Out[2]: GF([ 0, 1, , + 1, ^2, 
^2 + 1, ^2 + , ^2 + + 1], order=2^3)
In [3]: x.is_quadratic_residue()
Out[3]: array([ True, True, True, True, True, True, True, True])
```

In $\text{GF}(11)$, the characteristic is greater than 2 so only half of the elements have square roots.

```python
In [4]: GF = galois.GF(11)
In [5]: x = GF.elements; x
Out[5]: GF([ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10], order=11)
In [6]: x.is_quadratic_residue()
Out[6]: array([ True, True, False, True, True, True, False, False, False, True, False])
```

galois.FieldArray.left_null_space() → FieldArray

Computes the left null space of the matrix $A$.

**Returns**

The left null space basis matrix. The rows of the basis matrix are the basis vectors that span the left null space. The number of rows of the basis matrix is the dimension of the left null space.
Notes

Given an $m \times n$ matrix $A$ over $\text{GF}(q)$, the left null space of $A$ is the vector space $\{x \in \text{GF}(q)^m\}$ that annihilates the rows of $A$, i.e. $xA = 0$.

The left null space has properties $\mathcal{LN}(A) = \mathcal{N}(A^T)$ and $\dim(\mathcal{R}(A)) + \dim(\mathcal{LN}(A)) = m$.

Examples

The `left_null_space()` method defines basis vectors (its rows) that span the left null space of $A$.

```python
In [1]: m, n = 5, 3
In [2]: GF = galois.GF(31)
In [3]: A = GF.Random((m, n)); A
Out[3]: GF([[21, 28, 4],
          [23, 26, 13],
          [26, 24, 25],
          [14, 8, 3],
          [24, 29, 13]], order=31)

In [4]: LN = A.left_null_space(); LN
Out[4]: GF([[ 1, 0, 21, 16, 20],
          [ 0, 1, 22, 28, 17]], order=31)
```

The left null space is the set of vectors that sum the rows to 0.

```python
In [5]: LN @ A
Out[5]: GF([[ 0, 0, 0],
          [ 0, 0, 0]], order=31)
```

The dimension of the row space and left null space sum to $m$.

```python
In [6]: R = A.row_space(); R
Out[6]: GF([[1, 0, 0],
          [0, 1, 0],
          [0, 0, 1]], order=31)

In [7]: R.shape[0] + LN.shape[0] == m
Out[7]: True
```

galois.FieldArray.log(base: ElementLike | ArrayLike | None = None) → ndarray

Computes the logarithm of the array $x$ base $\beta$.

Important: If the Galois field is configured to use lookup tables, `ufunc_mode == "jit-lookup"`, and this function is invoked with a base different from `primitive_element`, then explicit calculation will be used.
Parameters

```
base: ElementLike | ArrayLike | None = None
```
A primitive element(s) $\beta$ of the finite field that is the base of the logarithm. The default is
```
None
```
which uses `primitive_element`.

Returns

An integer array $i$ of powers of $\beta$ such that $\beta^i = x$. The return array shape obeys NumPy
broadcasting rules.

Examples

Compute the logarithm of $x$ with default base $\alpha$, which is the specified primitive element of the field.

```
In [1]: GF = galois.GF(3**5, display="poly")
In [2]: alpha = GF.primitive_element; alpha
Out[2]: GF(, order=3^5)
In [3]: x = GF.Random(10, low=1); x
Out[3]: GF([ 2^4 + ^2 + 1, 2^4 + ^3 + ^2 + 2, 2^4 + 2^3 + 2^2 + ^2, ^4 + 2^3 + ^2 + 2, 2^2 + ^2, ^3 + ^2, 2^4 + 2], order=3^5)
In [4]: i = x.log(); i
Out[4]: array([206, 33, 96, 219, 2, 221, 101, 219, 71, 87])
In [5]: np.array_equal(alpha ** i, x)
Out[5]: True
```

With the default argument, `numpy.log()` and `log()` are equivalent.

```
In [6]: np.array_equal(np.log(x), x.log())
Out[6]: True
```

Compute the logarithm of $x$ with a different base $\beta$, which is another primitive element of the field.

```
In [7]: beta = GF.primitive_elements[-1]; beta
Out[7]: GF(2^4 + 2^3 + 2^2 + 2 + 2, order=3^5)
In [8]: i = x.log(beta); i
Out[8]: array([128, 165, 62, 149, 208, 115, 219, 36, 3, 215])
In [9]: np.array_equal(beta ** i, x)
Out[9]: True
```

Compute the logarithm of a single finite field element base all of the primitive elements of the field.

```
In [10]: x = GF.Random(low=1); x
Out[10]: GF(2^2 + + 1, order=3^5)
```
In [11]: bases = GF.primitive_elements

In [12]: i = x.log(bases); i
Out[12]: array([101, 19, 117, 111, 3, 89, 37, 1, 191, 17, 119, 195, 87,
        39, 239, 7, 153, 103, 73, 203, 205, 49, 93, 81, 225, 173,
        133, 139, 107, 177, 15, 163, 167, 201, 159, 199, 145, 71, 59,
        197, 95, 155, 135, 189, 161, 193, 131, 45, 147, 5, 25,
        91, 9, 67, 217, 175, 211, 63, 215, 127, 129, 185, 23, 53,
        221, 181, 227, 237, 141, 29, 13, 65, 207, 229, 105, 137, 123,
        171, 85, 183, 169, 97, 179, 83, 241, 41, 149, 113, 151, 69,
        223, 43, 27, 75, 31, 235, 115, 51, 157, 35, 57, 109, 61,
        21, 233, 79, 47, 125, 219])

In [13]: np.all(bases ** i == x)
Out[13]: True
In [6]: np.array_equal(A, L @ U)
Out[6]: True

galois.FieldArray.minimal_poly() \rightarrow \text{Poly}
Computes the minimal polynomial of a finite field element \(a\).

**Important:** This function may only be invoked on a single finite field element (scalar 0-D array).

**Returns**
For scalar inputs, the minimal polynomial \(m_a(x)\) of \(a\) over \(GF(p)\).

**Notes**
An element \(a\) of \(GF(p^m)\) has minimal polynomial \(m_a(x)\) over \(GF(p)\). The minimal polynomial when evaluated in \(GF(p^m)\) annihilates \(a\), that is \(m_a(a) = 0\). The minimal polynomial always divides the characteristic polynomial. In prime fields \(GF(p)\), the minimal polynomial of \(a\) is simply \(m_a(x) = x - a\).

**References**

**Examples**
The minimal polynomial of the element \(a\).

In [1]: GF = galois.GF(3**5)
In [2]: a = GF.Random(); a
Out[2]: GF(14, order=3^5)
In [3]: poly = a.minimal_poly(); poly
Out[3]: Poly(x^5 + 2x^4 + 2x^3 + 2x^2 + 1, GF(3))
# The minimal polynomial annihilates a
In [4]: poly(a, field=GF)
Out[4]: GF(0, order=3^5)
# The minimal polynomial always divides the characteristic polynomial
In [5]: divmod(a.characteristic_poly(), poly)
Out[5]: (Poly(1, GF(3)), Poly(0, GF(3)))

galois.FieldArray.multiplicative_order() \rightarrow \text{integer | ndarray}
Computes the multiplicative order \(ord(x)\) of each element in \(x\).
**Returns**

An integer array of the multiplicative order of each element in \(x\). The return value is a single integer if the input array \(x\) is a scalar.

**Raises**

`ArithmeticError` – If zero is provided as an input. The multiplicative order of 0 is not defined. There is no power of 0 that ever results in 1.

**Notes**

The multiplicative order \(\text{ord}(x) = a\) of \(x\) in \(\text{GF}(p^m)\) is the smallest power \(a\) such that \(x^a = 1\). If \(a = p^m - 1\), \(a\) is said to be a generator of the multiplicative group \(\text{GF}(p^m)^\times\).

Note, `multiplicative_order()` should not be confused with `order`. The former returns the multiplicative order of `FieldArray` elements. The latter is a property of the field, namely the finite field’s order or size.

**Examples**

Compute the multiplicative order of each non-zero element of \(\text{GF}(3^2)\).

```
In [1]: GF = galois.GF(3**2, display="poly")
In [2]: x = GF.units; x
Out[2]: GF([ 1, 2, , + 1, + 2, 2, 2 + 1, 2 + 2], order=3^2)
In [3]: order = x.multiplicative_order(); order
Out[3]: array([1, 2, 8, 4, 8, 8, 8, 4])
In [4]: x ** order
Out[4]: GF([1, 1, 1, 1, 1, 1, 1, 1], order=3^2)
```

The elements with \(\text{ord}(x) = 8\) are multiplicative generators of \(\text{GF}(3^2)^\times\), which are also called primitive elements.

```
In [5]: GF.primitive_elements
Out[5]: GF([ , + 2, 2, 2 + 1], order=3^2)
```
Notes

Given an $m \times n$ matrix $A$ over GF($q$), the null space of $A$ is the vector space $\{x \in GF(q)^n\}$ that annihilates the columns of $A$, i.e. $Ax = 0$.

The null space has properties $\mathcal{N}(A) = \mathcal{L}(A^T)$ and $\dim(\mathcal{C}(A)) + \dim(\mathcal{N}(A)) = n$.

Examples

The `null_space()` method defines basis vectors (its rows) that span the null space of $A$.

```
In [1]: m, n = 3, 5
In [2]: GF = galois.GF(31)
In [3]: A = GF.Random((m, n)); A
Out[3]: GF([[4, 13, 29, 2, 24],
    [7, 8, 19, 19, 16],
    [18, 6, 7, 23, 29]], order=31)
In [4]: N = A.null_space(); N
Out[4]: GF([[1, 0, 9, 28, 6],
    [0, 1, 17, 22, 21]], order=31)
```

The null space is the set of vectors that sum the columns to 0.

```
In [5]: A @ N.T
Out[5]: GF([[0, 0],
    [0, 0],
    [0, 0]], order=31)
```

The dimension of the column space and null space sum to $n$.

```
In [6]: C = A.column_space(); C
Out[6]: GF([[1, 0, 0],
    [0, 1, 0],
    [0, 0, 1]], order=31)
```

```
In [7]: C.shape[0] + N.shape[0] == n
Out[7]: True
```

galois.FieldArray.plu_decompose() → Tuple[FieldArray, FieldArray, FieldArray]

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

 Returns

 - $P$ – The column permutation matrix.
 - $L$ – The lower triangular matrix.
 - $U$ – The upper triangular matrix.
The PLU decomposition of $A$ is defined as $A = PLU$. This is equivalent to $P^T A = LU$.

Examples

```
In [1]: GF = galois.GF(31)
In [2]: A = GF([[0, 29, 2, 9], [20, 24, 5, 1], [2, 24, 1, 7]]); A
Out[2]:
GF([[ 0, 29, 2, 9],
    [20, 24, 5, 1],
    [ 2, 24, 1, 7]], order=31)
In [3]: P, L, U = A.plu_decompose()
In [4]: P
Out[4]:
GF([[0, 1, 0],
    [1, 0, 0],
    [0, 0, 1]], order=31)
In [5]: L
Out[5]:
GF([[1, 0, 0],
    [0, 1, 0],
    [28, 14, 1]], order=31)
In [6]: U
Out[6]:
GF([[20, 24, 5, 1],
    [0, 29, 2, 9],
    [0, 0, 19, 8]], order=31)
In [7]: np.array_equal(A, P @ L @ U)
Out[7]: True
In [8]: np.array_equal(P.T @ A, L @ U)
Out[8]: True
```
Notes

A primitive $n$-th root of unity $\omega_n$ is such that $\omega_n^n = 1$ and $\omega_n^k \neq 1$ for all $1 \leq kn$.

In $\text{GF}(p^m)$, a primitive $n$-th root of unity exists when $n$ divides $p^m - 1$. Then, the primitive root is $\omega_n = \alpha^{(p^m-1)/n}$ where $\alpha$ is a primitive element of the field.

Examples

In $\text{GF}(31)$, primitive roots exist for all divisors of 30.

```python
In [1]: GF = galois.GF(31)
In [2]: GF.primitive_root_of_unity(2)
Out[2]: GF(30, order=31)
In [3]: GF.primitive_root_of_unity(5)
Out[3]: GF(16, order=31)
In [4]: GF.primitive_root_of_unity(15)
Out[4]: GF(9, order=31)
```

However, they do not exist for $n$ that do not divide 30.

```python
In [5]: GF.primitive_root_of_unity(7)
---------------------------------------------------------------------------
ValueError Traceback (most recent call last)
Input In [5], in <cell line: 1>()
----> 1 GF.primitive_root_of_unity(7)
File ~/checkouts/readthedocs.org/user_builds/galois/checkouts/latest/galois/_fields/_array.py:1219, in FieldArray.primitive_root_of_unity(cls, n)
   1217     raise ValueError(f"Argument `n` must be in [1, \{cls.order\}], not \{n\}
   1218     if not (cls.order - 1) % n == 0:
-> 1219     raise ValueError(f"There are no primitive \{n\}-th roots of unity in
   1221     return cls.primitive_element ** ((cls.order - 1) // n)
ValueError: There are no primitive 7-th roots of unity in GF(31).
```

For $\omega_5$, one can see that $\omega_5^5 = 1$ and $\omega_5^k \neq 1$ for $1 \leq k5$.

```python
In [6]: root = GF.primitive_root_of_unity(5); root
Out[6]: GF(16, order=31)
In [7]: powers = np.arange(1, 5 + 1); powers
Out[7]: array([1, 2, 3, 4, 5])
In [8]: root ** powers
Out[8]: GF([16, 8, 4, 2, 1], order=31)
```

**classmethod** `galois.FieldArray.primitive_roots_of_unity(n: int) → FieldArray`

Finds all primitive $n$-th roots of unity in the finite field.
**Parameters**

- **n**: int  
The root of unity.

**Returns**

All primitive $n$-th roots of unity, a 1-D array. The roots are sorted in lexicographically-increasing order.

**Raises**

- **ValueError** – If no primitive $n$-th roots of unity exist. This happens when $n$ is not a divisor of $p^m - 1$.

**Notes**

A primitive $n$-th root of unity $\omega_n$ is such that $\omega_n^n = 1$ and $\omega_n^k \neq 1$ for all $1 \leq k < n$.

In $\text{GF}(p^m)$, a primitive $n$-th root of unity exists when $n$ divides $p^m - 1$. Then, the primitive root is $\omega_n = \alpha^{(p^m - 1)/n}$ where $\alpha$ is a primitive element of the field.

**Examples**

In $\text{GF}(31)$, primitive roots exist for all divisors of 30.

```
In [1]: GF = galois.GF(31)
In [2]: GF.primitive_roots_of_unity(2)
Out[2]: GF([30], order=31)
In [3]: GF.primitive_roots_of_unity(5)
Out[3]: GF([2, 4, 8, 16], order=31)
In [4]: GF.primitive_roots_of_unity(15)
Out[4]: GF([7, 9, 10, 14, 18, 19, 20, 28], order=31)
```

However, they do not exist for $n$ that do not divide 30.

```
In [5]: GF.primitive_roots_of_unity(7)
---------------------------------------------------------------------------
ValueError                                 Traceback (most recent call last)
Input In [5], in <cell line: 1>()
----> 1 GF.primitive_roots_of_unity(7)

File ~/checkouts/readthedocs.org/user_builds/galois/checkouts/latest/galois/_                                                               
→fields/_array.py:1283, in FieldArray.primitive_roots_of_unity(cls, n)
   1281     raise TypeError("Argument \`n` must be an int, not \`type(n)!r`.")
   1282 if not (cls.order - 1) % n == 0:
--> 1283     raise ValueError("There are no primitive \{n\}-th roots of unity in
→\{cls.name\}.")
   1285 roots = np.unique(cls.primitive_elements ** ((cls.order - 1) // n))
   1286 roots = np.sort(roots)

ValueError: There are no primitive 7-th roots of unity in GF(31).
```

For $\omega_5$, one can see that $\omega_5^5 = 1$ and $\omega_5^k \neq 1$ for $1 \leq k < 5$.
```python
In [6]: root = GF.primitive_roots_of_unity(5); root
Out[6]: GF([ 2, 4, 8, 16], order=31)

In [7]: powers = np.arange(1, 5 + 1); powers
Out[7]: array([1, 2, 3, 4, 5])

In [8]: np.power.outer(root, powers)
Out[8]: GF([[ 2, 4, 8, 16, 1],
        [ 4, 16, 2, 8, 1],
        [ 8, 2, 16, 4, 1],
        [16, 8, 4, 2, 1]], order=31)
```

classmethod galois.FieldArray.repr_table(element: ElementLike | None = None, sort: 'power' | 'poly' | 'vector' | 'int' = 'power') → str

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

**Parameters**

- element: ElementLike | None = None
  An element to use as the exponent base in the power representation. The default is None which corresponds to primitive_element.

- sort: 'power' | 'poly' | 'vector' | 'int' = 'power'
  The sorting method for the table. The default is "power". Sorting by "power" will order the rows of the table by ascending powers of element. Sorting by any of the others will order the rows in lexicographically-increasing polynomial/vector order, which is equivalent to ascending order of the integer representation.

**Returns**

A string representation of the table comparing the power, polynomial, vector, and integer representations of each field element.

**Examples**

Create a FieldArray subclass for GF(3^3).

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

**Galois Field:**
- name: GF(3^3)
- characteristic: 3
- degree: 3
- order: 27
- irreducible_poly: x^3 + 2x + 1
- is_primitive_poly: True
- primitive_element: x

Generate a representation table for GF(3^3). Since x^3 + 2x + 1 is a primitive polynomial, x is a primitive element of the field. Notice, \(\text{ord}(x) = 26\).
Generate a representation table for $GF(3^3)$ using a different primitive element $2x^2 + 2x + 2$. Notice, $\text{ord}(2x^2 + 2x + 2) = 26$.

In [4]: `GF("x").multiplicative_order()`  
Out[4]: 26
\[(2x^2 + 2x + 2)^{12} \quad 2x^2 + x + 1 \quad [2, 1, 1] \quad 22\]
\[(2x^2 + 2x + 2)^{13} \quad 2 \quad [0, 0, 2] \quad 2\]
\[(2x^2 + 2x + 2)^{14} \quad x^2 + x + 1 \quad [1, 1, 1] \quad 13\]
\[(2x^2 + 2x + 2)^{15} \quad 2x^2 + 1 \quad [2, 0, 1] \quad 19\]
\[(2x^2 + 2x + 2)^{16} \quad x^2 + x + 2 \quad [1, 2, 0] \quad 16\]
\[(2x^2 + 2x + 2)^{17} \quad 2x^2 + 2x \quad [2, 1, 0] \quad 14\]
\[(2x^2 + 2x + 2)^{18} \quad 2x + 1 \quad [0, 0, 2] \quad 8\]
\[(2x^2 + 2x + 2)^{19} \quad x + 1 \quad [0, 1, 1] \quad 4\]
\[(2x^2 + 2x + 2)^{20} \quad x^2 \quad [1, 0, 0] \quad 9\]
\[(2x^2 + 2x + 2)^{21} \quad 2x \quad [0, 2, 0] \quad 6\]
\[(2x^2 + 2x + 2)^{22} \quad 2x + 2 \quad [0, 2, 2] \quad 24\]
\[(2x^2 + 2x + 2)^{23} \quad 2x + 1 \quad [0, 2, 1] \quad 7\]
\[(2x^2 + 2x + 2)^{24} \quad x^2 + 2x \quad [1, 2, 1] \quad 16\]
\[(2x^2 + 2x + 2)^{25} \quad x^2 + 1 \quad [1, 0, 1] \quad 13\]

\textbf{In [6]:} GF("2x^2 + 2x + 2").multiplicative_order()
\textbf{Out[6]:} 26

Generate a representation table for GF(3^3) using non-primitive element \(x^2\). Notice, \(\text{ord}(x^2) = 13 \neq 26\).

\begin{center}
\begin{tabular}{llll}
\textbf{Power} & \textbf{Polynomial} & \textbf{Vector} & \textbf{Integer} \\
\hline
0 & 0 & [0, 0, 0] & 0 \\
(x^2)^0 & 1 & [0, 0, 1] & 1 \\
(x^2)^1 & x^2 & [1, 0, 0] & 9 \\
(x^2)^2 & x^2 + 2x & [1, 1, 0] & 15 \\
(x^2)^3 & x^2 + x + 1 & [1, 1, 1] & 13 \\
(x^2)^4 & 2x^2 + 2 & [2, 0, 2] & 20 \\
(x^2)^5 & x^2 + x & [1, 1, 0] & 12 \\
(x^2)^6 & x^2 + 2 & [1, 0, 2] & 11 \\
(x^2)^7 & 2x & [0, 2, 0] & 6 \\
(x^2)^8 & 2x + 1 & [0, 2, 1] & 7 \\
(x^2)^9 & x^2 + 2x + 1 & [1, 2, 1] & 16 \\
(x^2)^10 & 2x^2 + x + 1 & [2, 1, 0] & 22 \\
(x^2)^11 & 2x + 2 & [0, 2, 2] & 8 \\
(x^2)^12 & 2x^2 + 2x + 1 & [2, 2, 1] & 25 \\
(x^2)^13 & 1 & [0, 0, 1] & 1 \\
(x^2)^14 & x^2 & [1, 0, 0] & 9 \\
(x^2)^15 & x^2 + 2x & [1, 2, 0] & 15 \\
(x^2)^16 & x^2 + x + 1 & [1, 1, 1] & 13 \\
(x^2)^17 & 2x^2 + 2 & [2, 0, 2] & 20 \\
(x^2)^18 & x^2 + x & [1, 1, 0] & 12 \\
(x^2)^19 & x^2 + 2 & [1, 0, 2] & 11 \\
(x^2)^20 & 2x & [0, 2, 0] & 6 \\
(x^2)^21 & 2x + 1 & [0, 2, 1] & 7 \\
(x^2)^22 & x^2 + 2x + 1 & [1, 2, 1] & 16 \\
(x^2)^23 & 2x^2 + x + 1 & [2, 1, 1] & 22 \\
(x^2)^24 & 2x + 2 & [0, 2, 2] & 8 \\
(x^2)^25 & 2x^2 + 2x + 1 & [2, 2, 1] & 25 \\
\end{tabular}
\end{center}

\textbf{In [8]:} GF("x^2").multiplicative_order()
galois.FieldArray.row_reduce(ncols: int | None = None) → FieldArray

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

**Parameters**

- **ncols**: int | None = None
  - The number of columns to perform Gaussian elimination over. The default is None which represents the number of columns of the matrix.

**Returns**

The reduced row echelon form of the input matrix.

**Notes**

The elementary row operations in Gaussian elimination are:

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

**Examples**

Perform Gaussian elimination to get the reduced row echelon form of A.

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

In [2]: A = GF([[16, 12, 1, 25], [1, 10, 27, 29], [1, 0, 3, 19]]); A

Out[2]:
GF([[16, 12, 1, 25],
    [ 1, 10, 27, 29],
    [ 1, 0, 3, 19]], order=31)

In [3]: A.row_reduce()

Out[3]:
GF([[ 1, 0, 0, 11],
    [ 0, 1, 0, 7],
    [ 0, 0, 1, 13]], order=31)

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

Out[4]: 3
```

Or only perform Gaussian elimination over 2 columns.

```
In [5]: A.row_reduce(ncols=2)

Out[5]:
GF([[ 1, 0, 5, 14],
    [ 0, 1, 27, 17],
    [ 0, 0, 29, 5]], order=31)
```
**galois.FieldArray.row_space() → FieldArray**

Computes the row space of the matrix \( A \).

**Returns**

The row space basis matrix. The rows of the basis matrix are the basis vectors that span the row space. The number of rows of the basis matrix is the dimension of the row space.

**Notes**

Given an \( m \times n \) matrix \( A \) over \( GF(q) \), the row space of \( A \) is the vector space \( \{ x \in GF(q)^n \} \) defined by all linear combinations of the rows of \( A \). The row space has at most dimension \( \min(m, n) \).

The row space has properties \( \mathcal{R}(A) = \mathcal{C}(A^T) \) and \( \dim(\mathcal{R}(A)) + \dim(\mathcal{L}(A)) = m \).

**Examples**

The `row_space()` method defines basis vectors (its rows) that span the row space of \( A \).

```python
In [1]: m, n = 5, 3
In [2]: GF = galois.GF(31)
In [3]: A = GF.Random((m, n)); A
Out[3]:
GF([[30, 18, 24],
    [ 9, 6, 27],
    [24, 7, 8],
    [22, 9, 23],
    [ 2, 10, 10]], order=31)
In [4]: R = A.row_space(); R
Out[4]:
GF([[1, 0, 0],
    [0, 1, 0],
    [0, 0, 1]], order=31)
```

The dimension of the row space and left null space sum to \( m \).

```python
In [5]: LN = A.left_null_space(); LN
Out[5]:
GF([[1, 0, 18, 16, 27],
    [0, 1, 23, 19, 22]], order=31)
In [6]: R.shape[0] + LN.shape[0] == m
Out[6]: True
```

**galois.FieldArray.vector(dtype: DTypeLike | None = None) → FieldArray**

Converts an array over \( GF(p^m) \) to length-\( m \) vectors over the prime subfield \( GF(p) \).

This function is the inverse operation of the `Vector()` constructor. For an array with shape \((n1, n2)\), the output shape is \((n1, n2, m)\). By convention, the vectors are ordered from degree \( m - 1 \) to degree 0.

**Parameters**
**galois**

**dtype:** *DTypeLike* | *None = None*

The `numpy.dtype` of the array elements. The default is *None* which represents the smallest unsigned data type for this *FieldArray* subclass (the first element in *dtypes*).

**Returns**

An array over GF(p) with last dimension *m*.

**Examples**

```
In [1]: GF = galois.GF(3**3, display="poly")
In [2]: a = GF([11, 7]); a
Out[2]: GF([^2 + 2, 2 + 1], order=3^3)
In [3]: vec = a.vector(); vec
Out[3]: GF([[1, 0, 2],
        [0, 2, 1]], order=3)
In [4]: GF.Vector(vec)
Out[4]: GF([^2 + 2, 2 + 1], order=3^3)
```

**Properties**

**class property characteristic:** *int*

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

**class property default_ufunc_mode:** *'jit-lookup' | 'jit-calculate' | 'python-calculate'*

The default ufunc compilation mode for this *FieldArray* subclass. The ufuncs may be recompiled with `compile()`.

**class property degree:** *int*

The extension degree *m* of the Galois field GF(*p^m*). The degree is a positive integer.

**class property display_mode:** *'int' | 'poly' | 'power'*

The current finite field element representation. This can be changed with `display()`.

**class property dtypes:** *List*[dtype]

List of valid integer `numpy.dtype` values that are compatible with this finite field. Creating an array with an unsupported dtype will raise a *TypeError* exception.

**class property elements:** *FieldArray*

All of the finite field’s elements {0, ..., *p^m* − 1}.

**class property irreducible_poly:** *Poly*

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

**class property is_extension_field:** *bool*

Indicates if the finite field is an extension field. This is true when the field’s order is a prime power.
class property `is_prime_field`: bool
Indicates if the finite field is a prime field, not an extension field. This is true when the field’s order is prime.

class property `is_primitive_poly`: bool
Indicates whether the `irreducible_poly` is a primitive polynomial. If so, \( x \) is a primitive element of the finite field.

class property `name`: str
The finite field’s name as a string `GF(p)` or `GF(p^m)`.

class property `order`: int
The order \( p^m \) of the Galois field \( GF(p^m) \). The order of the field is equal to the field’s size.

class property `prime_subfield`: Type[FieldArray]
The prime subfield \( GF(p) \) of the extension field \( GF(p^m) \).

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

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

class property `quadratic_non_residues`: FieldArray
All quadratic non-residues in the Galois field.

class property `quadratic_residues`: FieldArray
All quadratic residues in the finite field.

class property `ufunc_mode`: ['jit-lookup', 'jit-calculate', 'python-calculate']
The current ufunc compilation mode for this `FieldArray` subclass. The ufuncs may be recompiled with `compile()`.

class property `ufunc_modes`: List[str]
All supported ufunc compilation modes for this `FieldArray` subclass.

class property `units`: FieldArray
All of the finite field’s units \( \{1, \ldots, p^m - 1\} \). A unit is an element with a multiplicative inverse.

class property `galois.FieldArray.characteristic`: int
The prime characteristic \( p \) of the Galois field \( GF(p^m) \). Adding \( p \) copies of any element will always result in 0.

Examples

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

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

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

(continues on next page)
class property galois.FieldArray.default_ufunc_mode: 'jit-lookup' | 'jit-calculate' | 'python-calculate'

The default ufunc compilation mode for this FieldArray subclass. The ufuncs may be recompiled with compile().

Examples

Fields with order less than $2^{20}$ are compiled, by default, using lookup tables for speed.

In [1]: galois.GF(65537).default_ufunc_mode
Out[1]: 'jit-lookup'

In [2]: galois.GF(2**16).default_ufunc_mode
Out[2]: 'jit-lookup'

Fields with order greater than $2^{20}$ are compiled, by default, using explicit calculation for memory savings. The field elements and arithmetic must still fit within numpy.int64.

In [3]: galois.GF(2147483647).default_ufunc_mode
Out[3]: 'jit-calculate'

In [4]: galois.GF(2**32).default_ufunc_mode
Out[4]: 'jit-calculate'

Fields whose elements and arithmetic cannot fit within numpy.int64 use pure-Python explicit calculation.

In [5]: galois.GF(36893488147419103183).default_ufunc_mode
Out[5]: 'python-calculate'

In [6]: galois.GF(2**100).default_ufunc_mode
Out[6]: 'python-calculate'

class property galois.FieldArray.degree: int

The extension 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

class property galois.FieldArray.display_mode: 'int' | 'poly' | 'power'
The current finite field element representation. This can be changed with display().

See Element Representation for a further discussion.

Examples

The default display mode is the integer representation.

In [1]: GF = galois.GF(3**2)
In [2]: x = GF.elements; x
Out[2]: GF([0, 1, 2, 3, 4, 5, 6, 7, 8], order=3^2)
In [3]: GF.display_mode
Out[3]: 'int'

Permanently modify the display mode by calling display().

In [4]: GF.display("poly");
In [5]: x
Out[5]: GF([0, 1, 2, +1, +2, 2, 2 +1, 2 +2], order=3^2)
In [6]: GF.display_mode
Out[6]: 'poly'

class property galois.FieldArray.dtypes: List[dtype]
List of valid integer numpy.dtype values that are compatible with this finite field. Creating an array with an unsupported dtype will raise a TypeError exception.

For finite fields whose elements cannot be represented with numpy.int64, the only valid data type is numpy.object_.

Examples

For small finite fields, all integer data types are acceptable, with the exception of numpy.uint64. This is because all arithmetic is done using numpy.int64.

In [1]: GF = galois.GF(31); GF.dtypes
Out[1]:
[numpy.uint8,
numpy.uint16,
numpy.uint32,
numpy.int8,
numpy.int16,
Some data types are too small for certain finite fields, such as `numpy.int16` for \( GF(7^5) \).

```python
In [2]: GF = galois.GF(7**5); GF.dtypes
Out[2]: [numpy.uint16, numpy.uint32, numpy.int16, numpy.int32, numpy.int64]
```

Large fields must use `numpy.object_` which uses Python `int` for its unlimited size.

```python
In [3]: GF = galois.GF(2**100); GF.dtypes
Out[3]: [numpy.object_]
```

```python
In [4]: GF = galois.GF(36893488147419103183); GF.dtypes
Out[4]: [numpy.object_]
```

class property galois.FieldArray.\texttt{elements}: FieldArray

All of the finite field’s elements \( \{0, \ldots, p^m - 1\} \).

**Examples**

All elements of the prime field \( GF(31) \) in increasing order.

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

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

All elements of the extension field \( GF(5^2) \) in lexicographically-increasing order.

```python
In [3]: GF = galois.GF(5**2, display="poly")

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

class property galois.FieldArray.\texttt{irreducible_poly}: Poly

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

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

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

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

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

class property `galois.FieldArray.is_extension_field`: bool

Indicates if the finite field is an extension field. This is true when the field’s order is a prime power.

Examples

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

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

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

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

class property `galois.FieldArray.is_prime_field`: bool

Indicates if the finite field is a prime field, not an extension field. This is true when the field’s order is prime.

Examples

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

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

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

In [4]: galois.GF(7**5).is_prime_field
Out[4]: False
```
**class property**  `galois.FieldArray.is_primitive_poly`: bool

Indicates whether the `irreducible_poly` is a primitive polynomial. If so, \( x \) is a primitive element of the finite field.

**Examples**

The default \( GF(2^8) \) field uses a primitive polynomial.

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

```python
In [3]: GF.is_primitive_poly
Out[3]: True
```

The \( GF(2^8) \) field from AES uses a non-primitive polynomial.

```python
In [4]: GF = galois.GF(2**8, irreducible_poly="x^8 + x^4 + x^3 + x + 1")
In [5]: print(GF.properties)
Galois Field:
   name: GF(2^8)
   characteristic: 2
   degree: 8
   order: 256
   irreducible_poly: x^8 + x^4 + x^3 + x + 1
   is_primitive_poly: False
   primitive_element: x + 1
```

```python
In [6]: GF.is_primitive_poly
Out[6]: False
```

**class property**  `galois.FieldArray.name`: str

The finite field’s name as a string \( GF(p) \) or \( GF(p^m) \).
Examples

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

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

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

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

class property galois.FieldArray.order: int
The order \( p^m \) of the Galois field \( \operatorname{GF}(p^m) \). The order of the field is equal to the field’s size.

Examples

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

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

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

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

class property galois.FieldArray.prime_subfield: Type[FieldArray]
The prime subfield \( \operatorname{GF}(p) \) of the extension field \( \operatorname{GF}(p^m) \).

Examples

In [1]: galois.GF(2).prime_subfield
Out[1]: galois.GF2

In [2]: galois.GF(2**8).prime_subfield
Out[2]: galois.GF2

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

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

class property galois.FieldArray.primitive_element: FieldArray
A primitive element \( \alpha \) of the Galois field \( \operatorname{GF}(p^m) \). A primitive element is a multiplicative generator of the field, such that \( \operatorname{GF}(p^m) = \{0, 1, \alpha, \alpha^2, \ldots, \alpha^{p^m-2}\} \).
A primitive element is a root of the primitive polynomial \( f(x) \), such that \( f(\alpha) = 0 \) over \( \text{GF}(p^m) \).

**Examples**

<table>
<thead>
<tr>
<th>In [1]:</th>
<th>galois.GF(2).primitive_element</th>
</tr>
</thead>
<tbody>
<tr>
<td>Out[1]:</td>
<td>GF(1, order=2)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>In [2]:</th>
<th>galois.GF(2**8).primitive_element</th>
</tr>
</thead>
<tbody>
<tr>
<td>Out[2]:</td>
<td>GF(2, order=2^8)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>In [3]:</th>
<th>galois.GF(31).primitive_element</th>
</tr>
</thead>
<tbody>
<tr>
<td>Out[3]:</td>
<td>GF(3, order=31)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>In [4]:</th>
<th>galois.GF(7**5).primitive_element</th>
</tr>
</thead>
<tbody>
<tr>
<td>Out[4]:</td>
<td>GF(7, order=7^5)</td>
</tr>
</tbody>
</table>

**class property** galois.FieldArray.primitive_elements: FieldArray

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**

<table>
<thead>
<tr>
<th>In [1]:</th>
<th>galois.GF(2).primitive_elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Out[1]:</td>
<td>GF([1], order=2)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>In [2]:</th>
<th>galois.GF(2**8).primitive_elements</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>In [3]:</th>
<th>galois.GF(31).primitive_elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Out[3]:</td>
<td>GF([3, 11, 12, 13, 17, 21, 22, 24], order=31)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>In [4]:</th>
<th>galois.GF(7**5).primitive_elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Out[4]:</td>
<td>GF([7, 8, 14, ..., 16797, 16798, 16803], order=7^5)</td>
</tr>
</tbody>
</table>

**class property** galois.FieldArray.quadratic_non_residues: FieldArray

All quadratic non-residues in the Galois field.

An element \( x \) in \( \text{GF}(p^m) \) is a quadratic non-residue if there does not exist a \( y \) such that \( y^2 = x \) in the field.

**See also:**

*is_quadratic_residue*
Examples

In fields with characteristic 2, no elements are quadratic non-residues.

```
In [1]: GF = galois.GF(2**4)
In [2]: GF.quadratic_non_residues
Out[2]: GF([], order=2^4)
```

In fields with characteristic greater than 2, exactly half of the nonzero elements are quadratic non-residues.

```
In [3]: GF = galois.GF(11)
In [4]: GF.quadratic_non_residues
Out[4]: GF([ 2, 6, 7, 8, 10], order=11)
```

class property  
galois.FieldArray.quadratic_residues: FieldArray
All quadratic residues in the finite field.

An element \( x \) in \( GF(p^m) \) is a quadratic residue if there exists a \( y \) such that \( y^2 = x \) in the field.

See also:

is_quadratic_residue

Examples

In fields with characteristic 2, every element is a quadratic residue.

```
In [1]: GF = galois.GF(2**4)
In [2]: x = GF.quadratic_residues; x
Out[2]: GF([ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15], order=2^4)
In [3]: r = np.sqrt(x); r
Out[3]: GF([ 0, 1, 5, 4, 2, 3, 7, 6, 10, 11, 15, 14, 8, 9, 13, 12], order=2^4)
In [4]: np.array_equal(r ** 2, x)
Out[4]: True
In [5]: np.array_equal((-r) ** 2, x)
Out[5]: True
```

In fields with characteristic greater than 2, exactly half of the nonzero elements are quadratic residues (and they have two unique square roots).

```
In [6]: GF = galois.GF(11)
In [7]: x = GF.quadratic_residues; x
Out[7]: GF([ 0, 1, 3, 4, 5, 9], order=11)
```

(continues on next page)
In [8]: r = np.sqrt(x); r
Out[8]: GF([0, 1, 5, 2, 4, 3], order=11)

In [9]: np.array_equal(r ** 2, x)
Out[9]: True

In [10]: np.array_equal((-r) ** 2, x)
Out[10]: True

class property galois.FieldArray.ufunc_mode: 'jit-lookup' | 'jit-calculate' | 'python-calculate'
The current ufunc compilation mode for this FieldArray subclass. The ufuncs may be recompiled with compile().

Examples
Fields with order less than $2^{20}$ are compiled, by default, using lookup tables for speed.

In [1]: galois.GF(65537).ufunc_mode
Out[1]: 'jit-lookup'

In [2]: galois.GF(2**16).ufunc_mode
Out[2]: 'jit-lookup'

Fields with order greater than $2^{20}$ are compiled, by default, using explicit calculation for memory savings. The field elements and arithmetic must still fit within numpy.int64.

In [3]: galois.GF(2147483647).ufunc_mode
Out[3]: 'jit-calculate'

In [4]: galois.GF(2**32).ufunc_mode
Out[4]: 'jit-calculate'

Fields whose elements and arithmetic cannot fit within numpy.int64 use pure-Python explicit calculation.

In [5]: galois.GF(36893488147419103183).ufunc_mode
Out[5]: 'python-calculate'

In [6]: galois.GF(2**100).ufunc_mode
Out[6]: 'python-calculate'

class property galois.FieldArray.ufunc_modes: List[str]
All supported ufunc compilation modes for this FieldArray subclass.
Examples

Fields whose elements and arithmetic can fit within `numpy.int64` can be JIT compiled to use either lookup tables or explicit calculation.

```python
In [1]: galois.GF(65537).ufunc_modes
Out[1]: ['jit-lookup', 'jit-calculate']

In [2]: galois.GF(2**32).ufunc_modes
Out[2]: ['jit-lookup', 'jit-calculate']
```

Fields whose elements and arithmetic cannot fit within `numpy.int64` may only use pure-Python explicit calculation.

```python
In [3]: galois.GF(36893488147419103183).ufunc_modes
Out[3]: ['python-calculate']

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

class property `galois.FieldArray.units`: `FieldArray`

All of the finite field’s units \{1, \ldots, p^m - 1\}. A unit is an element with a multiplicative inverse.

Examples

All units of the prime field \(GF(31)\) in increasing order.

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

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

All units of the extension field \(GF(5^2)\) in lexicographically-increasing order.

```python
In [3]: GF = galois.GF(5**2, display="poly")

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

class `galois.GF2` (`galois.FieldArray`)

A `FieldArray` subclass over \(GF(2)\).

**Important**: This class is a pre-generated `FieldArray` subclass generated with `galois.GF(2)` and is included in the API for convenience.
Examples

This class is equivalent, and in fact identical, to the `FieldArray` subclass returned from the class factory `GF()`.

```
In [1]: galois.GF2 is galois.GF(2)
Out[1]: True

In [2]: issubclass(galois.GF2, galois.FieldArray)
Out[2]: True

In [3]: print(galois.GF2.properties)
Galois Field:
   name: GF(2)
   characteristic: 2
   degree: 1
   order: 2
   irreducible_poly: x + 1
   is_primitive_poly: True
   primitive_element: 1
```

Create a `FieldArray` instance using `GF2`’s constructor.

```
In [4]: x = galois.GF2([1, 0, 1, 1]); x
Out[4]: GF([1, 0, 1, 1], order=2)

In [5]: isinstance(x, galois.GF2)
Out[5]: True
```

Constructors

- `GF2(x: ElementLike | ArrayLike, dtype: DTypeLike | None = None, ...) -> FieldArray`
  - Creates an array over GF($p^m$).
- `classmethod Identity(size: int, ...) -> FieldArray`
  - Creates an $n \times n$ identity matrix.
- `classmethod Ones(shape: ShapeLike, ...) -> FieldArray`
  - Creates an array of all ones.
- `classmethod Random(shape: ShapeLike = (0, ...)) -> FieldArray`
  - Creates an array with random elements.
- `classmethod Range(start: ElementLike, stop, ...) -> FieldArray`
  - Creates a 1-D array with a range of elements.
- `classmethod Vandermonde(element: ElementLike, ...) -> FieldArray`
  - Creates an $m \times n$ Vandermonde matrix of $a \in GF(q)$.
- `classmethod Vector(array: ArrayLike, ...) -> FieldArray`
  - Creates an array over GF($p^m$) from length-$m$ vectors over the prime subfield GF($p$).
- `classmethod Zeros(shape: ShapeLike, ...) -> FieldArray`
  - Creates an array of all zeros.
**String representation**

```python
__repr__() → str
```
Displays the array specifying the class and finite field order.

```python
__str__() → str
```
Displays the array without specifying the class or finite field order.

**class property properties : str**
A formatted string of relevant properties of the Galois field.

**Methods**

```python
additive_order() → integer | ndarray
```
Computes the additive order of each element in $x$.

```python
classmethod arithmetic_table(operation, ...) → str
```
Generates the specified arithmetic table for the finite field.

```python
characteristic_poly() → Poly
```
Computes the characteristic polynomial of a finite field element $a$ or a square matrix $A$.

```python
column_space() → FieldArray
```
Computes the column space of the matrix $A$.

```python
classmethod compile(mode)
```
Recompile the just-in-time compiled ufuncs for a new calculation mode.

```python
classmethod display(...) → Generator[None, None, None]
```
Sets the display mode for all arrays from this `FieldArray` subclass.

```python
field_norm() → FieldArray
```
Computes the field norm $N_{L/K}(x)$ of the elements of $x$.

```python
field_trace() → FieldArray
```
Computes the field trace $Tr_{L/K}(x)$ of the elements of $x$.

```python
is_quadratic_residue() → bool | ndarray
```
Determines if the elements of $x$ are quadratic residues in the finite field.

```python
left_null_space() → FieldArray
```
Computes the left null space of the matrix $A$.

```python
log(base: ElementLike | ArrayLike | None = None) → ndarray
```
Computes the logarithm of the array $x$ base $\beta$.

```python
lu_decompose() → Tuple[FieldArray, FieldArray]
```
Decomposes the input array into the product of lower and upper triangular matrices.

```python
minimal_poly() → Poly
```
Computes the minimal polynomial of a finite field element $a$.

```python
multiplicative_order() → integer | ndarray
```
Computes the multiplicative order $\text{ord}(x)$ of each element in $x$. 
null_space() → FieldArray
Computes the null space of the matrix A.

plu_decompose() → Tuple[FieldArray, FieldArray, FieldArray]
Decomposes the input array into the product of lower and upper triangular matrices using partial pivoting.

classmethod primitive_root_of_unity(n: int) → FieldArray
Finds a primitive n-th root of unity in the finite field.

classmethod primitive_roots_of_unity(n: int) → FieldArray
Finds all primitive n-th roots of unity in the finite field.
classmethod repr_table() → str
Generates a finite field element representation table comparing the power, polynomial, vector, and integer representations.

row_reduce(ncols: int | None = None) → FieldArray
Performs Gaussian elimination on the matrix to achieve reduced row echelon form (RREF).

row_space() → FieldArray
Computes the row space of the matrix A.

vector(dtype: DtypeLike | None = None) → FieldArray
Converts an array over \( \text{GF}(p^m) \) to length-\( m \) vectors over the prime subfield \( \text{GF}(p) \).

Properties

class property characteristic : int
The prime characteristic \( p \) of the Galois field \( \text{GF}(p^m) \). Adding \( p \) copies of any element will always result in 0.

class property default_ufunc_mode : ['jit-lookup' | 'jit-calculate' | 'python-calculate']
The default ufunc compilation mode for this FieldArray subclass. The ufuncs may be recompiled with compile().

class property degree : int
The extension degree \( m \) of the Galois field \( \text{GF}(p^m) \). The degree is a positive integer.

class property display_mode : ['int' | 'poly' | 'power']
The current finite field element representation. This can be changed with display().

class property dtypes : List[dtype]
List of valid integer numpy.dtype values that are compatible with this finite field. Creating an array with an unsupported dtype will raise a TypeError exception.

class property elements : FieldArray
All of the finite field’s elements \{0, \ldots, p^m - 1\}.

class property irreducible_poly : Poly
The irreducible polynomial \( f(x) \) of the Galois field \( \text{GF}(p^m) \). The irreducible polynomial is of degree \( m \) over \( \text{GF}(p) \).

class property is_extension_field : bool
Indicates if the finite field is an extension field. This is true when the field’s order is a prime power.
class property `is_prime_field`: bool
Indicates if the finite field is a prime field, not an extension field. This is true when the field’s order is prime.

class property `is_primitive_poly`: bool
Indicates whether the `irreducible_poly` is a primitive polynomial. If so, \( x \) is a primitive element of the finite field.

class property `name`: str
The finite field’s name as a string \( \text{GF}(p) \) or \( \text{GF}(p^m) \).

class property `order`: int
The order \( p^m \) of the Galois field \( \text{GF}(p^m) \). The order of the field is equal to the field’s size.

class property `prime_subfield`: Type[FieldArray]
The prime subfield \( \text{GF}(p) \) of the extension field \( \text{GF}(p^m) \).

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

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

class property `quadratic_non_residues`: FieldArray
All quadratic non-residues in the Galois field.

class property `quadratic_residues`: FieldArray
All quadratic residues in the finite field.

class property `ufunc_mode`: {'jit-lookup', 'jit-calculate', 'python-calculate'}
The current ufunc compilation mode for this FieldArray subclass. The ufuncs may be recompiled with `compile()`.

class property `ufunc_modes`: List[str]
All supported ufunc compilation modes for this FieldArray subclass.

class property `units`: FieldArray
All of the finite field’s units \( \{1, \ldots, p^m - 1\} \). A unit is an element with a multiplicative inverse.

galois.Field(order: int, irreducible_poly: PolyLike | None = None, primitive_element: int | PolyLike | None = None, verify: bool = True, compile: 'auto' | 'jit-lookup' | 'jit-calculate' | 'python-calculate' | None = None, display: 'int' | 'poly' | 'power' | None = None) → Type[FieldArray]
Alias of \( \text{GF}(.) \).

galois.GF(order: int, irreducible_poly: PolyLike | None = None, primitive_element: int | PolyLike | None = None, verify: bool = True, compile: 'auto' | 'jit-lookup' | 'jit-calculate' | 'python-calculate' | None = None, display: 'int' | 'poly' | 'power' | None = None) → Type[FieldArray]
Creates a FieldArray subclass for \( \text{GF}(p^m) \).

Parameters

order: int
The order \( p^m \) of the field \( \text{GF}(p^m) \). The order must be a prime power.
irreducible_poly: PolyLike | None = None

Optionally specify an irreducible polynomial of degree \( m \) over GF(\( p \)) that defines the finite field arithmetic. The default is None which uses the Conway polynomial \( C_{p,m} \), see `conway_poly()`.

primitive_element: int | PolyLike | None = None

Optionally specify a primitive element of the field. This value is used when building the exponential and logarithm lookup tables and as the base of `numpy.log`. A primitive element is a generator of the multiplicative group of the field.

For prime fields GF(\( p \)), the primitive element must be an integer and is a primitive root modulo \( p \). The default is None which uses `primitive_root()`.

For extension fields GF(\( p^m \)), the primitive element is a polynomial of degree less than \( m \) over GF(\( p \)). The default is None which uses `primitive_element()`.

verify: bool = True

Indicates whether to verify that the user-provided irreducible polynomial is in fact irreducible and that the user-provided primitive element is in fact a generator of the multiplicative group. The default is True.

For large fields and irreducible polynomials that are already known to be irreducible (which may take a while to verify), this argument may be set to False.

The default irreducible polynomial and primitive element are never verified because they are already known to be irreducible and a multiplicative generator, respectively.

compile: 'auto' | 'jit-lookup' | 'jit-calculate' | 'python-calculate' | None = None

The ufunc calculation mode. This can be modified after class construction with the `compile()` method. See Compilation Modes for a further discussion.

- None (default): For a newly-created FieldArray subclass, None corresponds to "auto". If the FieldArray subclass already exists, None does not modify its current compilation mode.
- "auto": Selects "jit-lookup" for fields with order less than \( 2^{20} \), "jit-calculate" for larger fields, and "python-calculate" for fields whose elements cannot be represented with `numpy.int64`.
- "jit-lookup": JIT compiles arithmetic ufuncs to use Zech log, log, and anti-log lookup tables for efficient computation. In the few cases where explicit calculation is faster than table lookup, explicit calculation is used.
- "jit-calculate": JIT compiles arithmetic ufuncs to use explicit calculation. The "jit-calculate" mode is designed for large fields that cannot or should not store lookup tables in RAM. Generally, the "jit-calculate" mode is slower than "jit-lookup".
- "python-calculate": Uses pure-Python ufuncs with explicit calculation. This is reserved for fields whose elements cannot be represented with `numpy.int64` and instead use `numpy.object_` with Python int (which has arbitrary precision).

display: 'int' | 'poly' | 'power' | None = None

The field element display representation. This can be modified after class construction with the `display()` method. See Element Representation for a further discussion.

- None (default): For a newly-created FieldArray subclass, None corresponds to "int". If the FieldArray subclass already exists, None does not modify its current display mode.
- "int": Sets the display mode to the integer representation.
• "poly": Sets the display mode to the polynomial representation.
• "power": Sets the display mode to the power representation.

Returns
A FieldArray subclass for GF($p^m$).

Notes
FieldArray subclasses of the same type (order, irreducible polynomial, and primitive element) are singletons. So, calling this class factory with arguments that correspond to the same subclass will return the same class object.

Examples
Create a FieldArray subclass for each type of finite field.

**GF(2)**
Construct the binary field.

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

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

**GF(p)**
Construct a prime field.

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

In [4]: print(GF.properties)
Galois Field:
   name: GF(31)
   characteristic: 31
   degree: 1
   order: 31
   irreducible_poly: x + 28
   is_primitive_poly: True
   primitive_element: 3
```
**GF(2^m)**

Construct a binary extension field. Notice the default irreducible polynomial is primitive and $x$ is a primitive element.

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

In [6]: print(GF.properties)
Galois Field:
   name: GF(2^8)
   characteristic: 2
   degree: 8
   order: 256
   irreducible_poly: x^8 + x^4 + x^3 + x^2 + 1
   is_primitive_poly: True
   primitive_element: x
```

**GF(p^m)**

Construct a prime extension field. Notice the default irreducible polynomial is primitive and $x$ is a primitive element.

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

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

Create a `FieldArray` subclass for extension fields and specify their irreducible polynomials.

**GF(2^m)**

Construct the GF(2^8) field that is used in AES. Notice the irreducible polynomial is not primitive and $x$ is not a primitive element.

```python
In [9]: GF = galois.GF(2**8, irreducible_poly="x^8 + x^4 + x^3 + x + 1")

In [10]: print(GF.properties)
Galois Field:
   name: GF(2^8)
   characteristic: 2
   degree: 8
   order: 256
   irreducible_poly: x^8 + x^4 + x^3 + x + 1
   is_primitive_poly: False
   primitive_element: x + 1
```
**GF(p^m)**

Construct GF(3^5) with an irreducible, but not primitive, polynomial. Notice that \( x \) is not a primitive element.

```python
In [11]: GF = galois.GF(3**5, irreducible_poly="x^5 + 2x + 2")

In [12]: print(GF.properties)
Galois Field:
   name: GF(3^5)
   characteristic: 3
   degree: 5
   order: 243
   irreducible_poly: x^5 + 2x + 2
   is_primitive_poly: False
   primitive_element: 2x
```

Finite fields with arbitrarily-large orders are supported.

**GF(p)**

Construct a large prime field.

```python
In [13]: GF = galois.GF(36893488147419103183)

In [14]: print(GF.properties)
Galois Field:
   name: GF(36893488147419103183)
   characteristic: 36893488147419103183
   degree: 1
   order: 36893488147419103183
   irreducible_poly: x + 36893488147419103180
   isPrimitive_poly: True
   primitive_element: 3
```

**GF(2^m)**

Construct a large binary extension field.

```python
In [15]: GF = galois.GF(2**100)

In [16]: print(GF.properties)
Galois Field:
   name: GF(2^100)
   characteristic: 2
   degree: 100
   order: 12676506002282294014967032005376
   irreducible_poly: x^100 + x^94 + x^93 + x^91 + x^90 + x^89 + x^88 + x^87 + x^86 + x^85 + x^84 + x^83 + x^82 + x^81 + x^80 + x^79 + x^78 + x^77 + x^76 + x^75 + x^74 + x^73 + x^72 + x^71 + x^70 + x^69 + x^68 + x^67 + x^66 + x^65 + x^64 + x^63 + x^62 + x^61 + x^60 + x^59 + x^58 + x^57 + x^56 + x^55 + x^54 + x^53 + x^52 + x^51 + x^50 + x^49 + x^48 + x^47 + x^46 + x^45 + x^44 + x^43 + x^42 + x^41 + x^40 + x^39 + x^38 + x^37 + x^36 + x^35 + x^34 + x^33 + x^32 + x^31 + x^30 + x^29 + x^28 + x^27 + x^26 + x^25 + x^24 + x^23 + x^22 + x^21 + x^20 + x^19 + x^18 + x^17 + x^16 + x^15 + x^14 + x^13 + x^12 + x^11 + x^10 + x^9 + x^8 + x^7 + x^6 + x^5 + x^4 + x^3 + x^2 + x^1 + x^0
   isPrimitive_poly: True
   primitive_element: x
```
GF(p^m)

Construct a large prime extension field.

```
In [17]: GF = galois.GF(109987**4)
In [18]: print(GF.properties)
Galois Field:
   name: GF(109987^4)
   characteristic: 109987
   degree: 4
   order: 146340800268433348561
   irreducible_poly: x^4 + 3x^2 + 100525x + 3
   is_primitive_poly: True
   primitive_element: x
```

**Primitive elements**

galois.\texttt{is\_primitive\_element}(element: PolyLike, ...) → bool

Determines if $g$ is a primitive element of the Galois field $GF(q^m)$ with degree-$m$ irreducible polynomial $f(x)$ over $GF(q)$.

galois.\texttt{primitive\_element}(irreducible\_poly: Poly, ...) → Poly

Finds a primitive element $g$ of the Galois field $GF(q^m)$ with degree-$m$ irreducible polynomial $f(x)$ over $GF(q)$.

galois.\texttt{primitive\_elements}(irreducible\_poly: Poly) → List[Poly]

Finds all primitive elements $g$ of the Galois field $GF(q^m)$ with degree-$m$ irreducible polynomial $f(x)$ over $GF(q)$.

galois.\texttt{is\_primitive\_element}(element: PolyLike, irreducible\_poly: Poly) → bool

Determines if $g$ is a primitive element of the Galois field $GF(q^m)$ with degree-$m$ irreducible polynomial $f(x)$ over $GF(q)$.

**Parameters**

- \texttt{element: PolyLike}
  An element $g$ of $GF(q^m)$ is a polynomial over $GF(q)$ with degree less than $m$.

- \texttt{irreducible\_poly: Poly}
  The degree-$m$ irreducible polynomial $f(x)$ over $GF(q)$ that defines the extension field $GF(q^m)$.

**Returns**

\texttt{True} if $g$ is a primitive element of $GF(q^m)$.

**See also:**

\texttt{primitive\_element, FieldArray.primitive\_element}
Examples

In the extension field \(GF(3^4)\), the element \(x + 2\) is a primitive element whose order is \(3^4 - 1 = 80\).

```
In [1]: GF = galois.GF(3**4)

In [2]: f = GF.irreducible_poly; f
Out[2]: Poly(x^4 + 2x^3 + 2, GF(3))

In [3]: galois.is_primitive_element("x + 2", f)
Out[3]: True

In [4]: GF("x + 2").multiplicative_order()
Out[4]: 80
```

However, the element \(x + 1\) is not a primitive element, as noted by its order being only 20.

```
In [5]: galois.is_primitive_element("x + 1", f)
Out[5]: False

In [6]: GF("x + 1").multiplicative_order()
Out[6]: 20
```

galois.primitive_element(irreducible_poly: Poly, method: 'min' | 'max' | 'random' = 'min') → Poly

Finds a primitive element \(g\) of the Galois field \(GF(q^m)\) with degree-\(m\) irreducible polynomial \(f(x)\) over \(GF(q)\).

Parameters

- **irreducible_poly**: Poly
  - The degree-\(m\) irreducible polynomial \(f(x)\) over \(GF(q)\) that defines the extension field \(GF(q^m)\).

- **method**: 'min' | 'max' | 'random' = 'min'
  - The search method for finding the primitive element.

Returns

A primitive element \(g\) of \(GF(q^m)\) with irreducible polynomial \(f(x)\). The primitive element \(g\) is a polynomial over \(GF(q)\) with degree less than \(m\).

See also:

- is_primitive_element, FieldArray.primitive_element

Examples

Construct the extension field \(GF(7^5)\).

```
In [1]: f = galois.irreducible_poly(7, 5, method="max"); f
Out[1]: Poly(x^5 + 6x^4 + 6x^3 + 6x^2 + 6x + 6, GF(7))

In [2]: GF = galois.GF(7**5, irreducible_poly=f, display="poly")

In [3]: print(GF.properties)
Galois Field:
  name: GF(7^5)
  characteristic: 7
```

(continues on next page)
degree: 5
order: 16007
irreducible_poly: \(x^5 + 6x^4 + 6x^3 + 6x^2 + 6x + 6\)
is_primitive_poly: False
primitive_element: \(x + 3\)

Find the smallest primitive element for the degree-5 extension of GF(7) with irreducible polynomial \(f(x)\).

```python
In [4]: g = galois.primitive_element(f); g
Out[4]: Poly(x + 3, GF(7))

# Convert the polynomial over GF(7) into an element of GF(7^5)
In [5]: g = GF(int(g)); g
Out[5]: GF(x + 3, order=7^5)

In [6]: g.multiplicative_order() == GF.order - 1
Out[6]: True
```

Find the largest primitive element for the degree-5 extension of GF(7) with irreducible polynomial \(f(x)\).

```python
In [7]: g = galois.primitive_element(f, method="max"); g
Out[7]: Poly(6x^4 + 6x^3 + 6x^2 + 6x + 3, GF(7))

# Convert the polynomial over GF(7) into an element of GF(7^5)
In [8]: g = GF(int(g)); g
Out[8]: GF(6x^4 + 6x^3 + 6x^2 + 6x + 3, order=7^5)

In [9]: g.multiplicative_order() == GF.order - 1
Out[9]: True
```

Find a random primitive element for the degree-5 extension of GF(7) with irreducible polynomial \(f(x)\).

```python
In [10]: g = galois.primitive_element(f, method="random"); g
Out[10]: Poly(4x^4 + 6x^3 + 6x + 1, GF(7))

# Convert the polynomial over GF(7) into an element of GF(7^5)
In [11]: g = GF(int(g)); g
Out[11]: GF(4x^4 + 6x^3 + 6x + 1, order=7^5)

In [12]: g.multiplicative_order() == GF.order - 1
Out[12]: True
```

galois.primitive_elements(irreducible_poly: Poly) → List[Poly]

Finds all primitive elements \(g\) of the Galois field GF(\(q^m\)) with degree-\(m\) irreducible polynomial \(f(x)\) over GF(\(q\)).

**Parameters**

- **irreducible_poly: Poly**
  
The degree-\(m\) irreducible polynomial \(f(x)\) over GF(\(q\)) that defines the extension field GF(\(q^m\)).

**Returns**

List of all primitive elements of GF(\(q^m\)) with irreducible polynomial \(f(x)\). Each primitive element \(g\) is a polynomial over GF(\(q\)) with degree less than \(m\).
See also:

`is_primitive_element`, `FieldArray.primitive_elements`

Notes

The number of primitive elements of $\mathbb{GF}(q^m)$ is $\phi(q^m - 1)$, where $\phi(n)$ is the Euler totient function. See `euler_phi`.

Examples

Construct the extension field $\mathbb{GF}(3^4)$.

```python
In [1]: f = galois.irreducible_poly(3, 4, method="max"); f
Out[1]: Poly(x^4 + 2x^3 + 2x^2 + x + 2, GF(3))

In [2]: GF = galois.GF(3**4, irreducible_poly=f, display="poly")
```

```python
In [3]: print(GF.properties)
Galois Field:
   name: GF(3^4)
   characteristic: 3
   degree: 4
   order: 81
   irreducible_poly: x^4 + 2x^3 + 2x^2 + x + 2
   is_primitive_poly: True
   primitive_element: x
```

Find all primitive elements for the degree-4 extension of $\mathbb{GF}(3)$.

```python
In [4]: g = galois.primitive_elements(f); g
Out[4]: [Poly(x, GF(3)), Poly(x + 1, GF(3)), Poly(2x, GF(3)), Poly(2x + 2, GF(3)), Poly(x^2 + 1, GF(3)), Poly(x^2 + 2x + 2, GF(3)), Poly(2x^2 + 2, GF(3)), Poly(2x^2 + x + 1, GF(3)), Poly(x^3, GF(3)), Poly(x^3 + 1, GF(3)), Poly(x^3 + x^2, GF(3)), Poly(x^3 + x^2 + 2, GF(3)), Poly(x^3 + x^2 + x, GF(3)), Poly(x^3 + x^2 + 2x, GF(3)), Poly(x^3 + x^2 + 2x + 2, GF(3)), Poly(x^3 + 2x^2, GF(3)), Poly(x^3 + 2x^2 + 2, GF(3)), Poly(x^3 + 2x^2 + x, GF(3)), Poly(x^3 + 2x^2 + 2x + 1, GF(3)), Poly(2x^3, GF(3)), Poly(x^3 + x^2 + 2 + x, GF(3)), Poly(x^3 + x^2 + 2x, GF(3)), Poly(x^3 + x^2 + 2x + 2, GF(3)), Poly(x^3 + 2x^2 + 2x, GF(3)), Poly(x^3 + 2x^2 + 2x + 1, GF(3)), Poly(2x^3, GF(3))], (continues on next page)
```
Poly(2x^3 + 2, GF(3)),
Poly(2x^3 + x^2, GF(3)),
Poly(2x^3 + x^2 + 1, GF(3)),
Poly(2x^3 + x^2 + x + 2, GF(3)),
Poly(2x^3 + x^2 + 2x, GF(3)),
Poly(2x^3 + x^2 + 2x + 2, GF(3)),
Poly(2x^3 + 2x^2, GF(3)),
Poly(2x^3 + 2x^2 + 1, GF(3)),
Poly(2x^3 + 2x^2 + x, GF(3)),
Poly(2x^3 + 2x^2 + x + 1, GF(3)),
Poly(2x^3 + 2x^2 + 2x, GF(3)),
Poly(2x^3 + 2x^2 + 2x + 2, GF(3))]

The number of primitive elements is given by $\phi(q^m - 1)$.

In [5]: phi = galois.euler_phi(3**4 - 1); phi
Out[5]: 32

In [6]: len(g) == phi
Out[6]: True

Shows that each primitive element has an order of $q^m - 1$.

# Convert the polynomials over GF(3) into elements of GF(3^4)
In [7]: g = GF([int(gi) for gi in g]); g
Out[7]: GF([ , + 1, 2,
2 + 2, ^2 + 1, ^2 + 2 + 2,
2^2 + 2, 2^2 + + 1, ^3,
^3 + 1,
^3 + ^2,
^3 + ^2 + 2,
^3 + ^2 + + 1,
^3 + 2^2 + + 1, ^3 + 2^2 + 2 + 1,
2^3 + ^2 + + 2, 2^3 + ^2 + 2 + 2,
2^3 + 2^2 + ^2 + 2,
2^3 + 2^2 + + 1, 2^3 + 2^2 + + 1, 2^3 + 2^2 + 2, 2^3 + 2^2 + + 1, 2^3 + 2^2 + + 1, 2^3 + 2^2 + + 1, 2^3 + 2^2 + + 1], order=3^4)

In [8]: np.all(g.multiplicative_order() == GF.order - 1)
Out[8]: True

### 3.18.3 Polynomials

class galois.Poly
A univariate polynomial $f(x)$ over GF($p^m$).

galois.typing.PolyLike
A Union representing objects that can be coerced into a polynomial.

class galois.Poly
A univariate polynomial $f(x)$ over GF($p^m$).
**Examples**

Create a polynomial over GF(2).

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

Create a polynomial over GF(3^5).

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

See *Polynomials* and *Polynomial Arithmetic* for more examples.

**Constructors**

```
galois.Poly(coeffs: ArrayLike, field: Type[Array] | None = None, ...) -> Poly
```

Creates a polynomial \( f(x) \) over GF\( (p^m) \).

```
Poly(coeffs: ArrayLike, field: Type[Array] | None = None, ...) -> Poly
```

The polynomial \( f(x) = \sum_{i=0}^{d} a_i x^i \) has coefficients \( \{a_d, a_{d-1}, \ldots, a_1, a_0\} \) in GF\( (p^m) \).

**Parameters**

- `coeffs`: ArrayLike
  - The polynomial coefficients \( \{a_d, a_{d-1}, \ldots, a_1, a_0\} \).
field: Type[Array] | None = None

The Galois field \( GF(p^m) \) the polynomial is over.

- **None** (default): If the coefficients are an `Array`, they won’t be modified. If the coefficients are not explicitly in a Galois field, they are assumed to be from \( GF(2) \) and are converted using `galois.GF2(coeffs)`.
- **Array subclass**: The coefficients are explicitly converted to this Galois field using `field(coeffs)`.

order: 'desc' | 'asc' = 'desc'

The interpretation of the coefficient degrees.

- "desc" (default): The first element of `coeffs` is the highest degree coefficient, i.e. \( \{a_d, a_{d-1}, \ldots, a_1, a_0\} \).
- "asc": The first element of `coeffs` is the lowest degree coefficient, i.e. \( \{a_0, a_1, \ldots, a_{d-1}, a_d\} \).

**classmethod galois.Poly.Degrees(degrees: Sequence[int] | ndarray, coeffs: ArrayLike | None = None, field: Type[Array] | None = None) \rightarrow Poly**

Constructs a polynomial over \( GF(p^m) \) from its non-zero degrees.

**Parameters**

- **degrees**: Sequence[int] | ndarray
  The polynomial degrees with non-zero coefficients.

- **coeffs**: ArrayLike | None = None
  The corresponding non-zero polynomial coefficients. The default is `None` which corresponds to all ones.

- **field**: Type[Array] | None = None
  The Galois field \( GF(p^m) \) the polynomial is over.

- **None** (default): If the coefficients are an `Array`, they won’t be modified. If the coefficients are not explicitly in a Galois field, they are assumed to be from \( GF(2) \) and are converted using `galois.GF2(coeffs)`.

- **Array subclass**: The coefficients are explicitly converted to this Galois field using `field(coeffs)`.

**Returns**

The polynomial \( f(x) \).

**Examples**

Construct a polynomial over \( GF(2) \) 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 \( GF(3^5) \) by specifying the degrees with non-zero coefficients and their coefficient values.

```
In [2]: GF = galois.GF(3**5)
```
Class method `galois.Poly.Identity(field: Type[Array]) | None = None) → Poly`

Constructs the polynomial $f(x) = x$ over $\text{GF}(p^m)$.

**Parameters**

- **field**: Type[Array] | None = None
  The Galois field $\text{GF}(p^m)$ the polynomial is over. The default is None which corresponds to GF2.

**Returns**

The polynomial $f(x) = x$.

**Examples**

Construct the identity polynomial over $\text{GF}(2)$.

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

Construct the identity polynomial over $\text{GF}(3^5)$.

```python
In [2]: GF = galois.GF(3**5)
In [3]: galois.Poly.Identity(GF)
Out[3]: Poly(x, GF(3^5))
```

Class method `galois.Poly.Int(integer: int, field: Type[Array]) | None = None) → Poly`

Constructs a polynomial over $\text{GF}(p^m)$ from its integer representation.

`Int()` and `__int__()` are inverse operations.

**Parameters**

- **integer**: int
  The integer representation of the polynomial $f(x)$.

- **field**: Type[Array] | None = None
  The Galois field $\text{GF}(p^m)$ the polynomial is over. The default is None which corresponds to GF2.

**Returns**

The polynomial $f(x)$. 

In [3]: galois.Poly.Degrees([3, 1, 0], coeffs=[214, 73, 185], field=GF)
Out[3]: Poly(214x^3 + 73x + 185, GF(3^5))
Examples

Construct a polynomial over \(GF(2)\) from its integer representation.

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

```
In [2]: int(f)
Out[2]: 5
```

Construct a polynomial over \(GF(3^5)\) from its integer representation.

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

```
In [4]: f = galois.Poly.Int(186535908, field=GF); f
Out[4]: Poly(13x^3 + 117, GF(3^5))
```

```
In [5]: int(f)
Out[5]: 186535908
```

# The polynomial/integer equivalence
```
In [6]: int(f) == 13*GF.order**3 + 117
Out[6]: True
```

Construct a polynomial over \(GF(2)\) from its binary string.

```
In [7]: f = galois.Poly.Int(int("0b1011", 2)); f
Out[7]: Poly(x^3 + x + 1, GF(2))
```

```
In [8]: bin(f)
Out[8]: '0b1011'
```

Construct a polynomial over \(GF(2^3)\) from its octal string.

```
In [9]: GF = galois.GF(2**3)
```

```
In [10]: f = galois.Poly.Int(int("0o5034", 8), field=GF); f
Out[10]: Poly(5x^3 + 3x + 4, GF(2^3))
```

```
In [11]: oct(f)
Out[11]: '0o5034'
```

Construct a polynomial over \(GF(2^8)\) from its hexadecimal string.

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

```
In [13]: f = galois.Poly.Int(int("0xf700a275", 16), field=GF); f
Out[13]: Poly(247x^3 + 162x + 117, GF(2^8))
```

```
In [14]: hex(f)
Out[14]: '0xf700a275'
```

classmethod galois.Poly.One(field: Type[Array] | None = None) → Poly

Constructs the polynomial \(f(x) = 1\) over \(GF(p^m)\).
Parameters

```python
galois.Poly.Random(degree: int, seed: int | Generator | None = None, field: Type[Array] | None = None) → Poly
```

Constructs a random polynomial over $\GF(p^m)$ with degree $d$.

Parameters

- **degree**: int
  
  The degree of the polynomial.

- **seed**: int | Generator | None = None
  
  Non-negative integer used to initialize the PRNG. The default is None which means that unpredictable entropy will be pulled from the OS to be used as the seed. A `numpy.random.Generator` can also be passed.

- **field**: Type[Array] | None = None
  
  The Galois field $\GF(p^m)$ the polynomial is over. The default is None which corresponds to $\GF(2)$.

Returns

The polynomial $f(x)$.

Examples

Construct a random degree-5 polynomial over $\GF(2)$.

```
In [1]: galois.Poly.Random(5)
Out[1]: Poly(x^5 + x^2, GF(2))
```

Construct a random degree-5 polynomial over $\GF(3^5)$ with a given seed. This produces repeatable results.

```
In [2]: GF = galois.GF(3**5)
In [3]: galois.Poly.Random(5, seed=123456789, field=GF)
```
Out[3]: Poly(56x^5 + 228x^4 + 157x^3 + 218x^2 + 148x + 43, GF(3^5))

In [4]: galois.Poly.Random(5, seed=123456789, field=GF)
Out[4]: Poly(56x^5 + 228x^4 + 157x^3 + 218x^2 + 148x + 43, GF(3^5))

Construct multiple polynomials with one global seed.

In [5]: rng = np.random.default_rng(123456789)
In [6]: galois.Poly.Random(5, seed=rng, field=GF)
Out[6]: Poly(56x^5 + 228x^4 + 157x^3 + 218x^2 + 148x + 43, GF(3^5))
In [7]: galois.Poly.Random(5, seed=rng, field=GF)
Out[7]: Poly(194x^5 + 195x^4 + 200x^3 + 141x^2 + 164x + 119, GF(3^5))

```
classmethod galois.Poly.Roots(roots: ArrayLike, multiplicities: Sequence[int] | ndarray | None = None, field: Type[Array] | None = None) → Poly
```

Constructs a monic polynomial over $\text{GF}(p^m)$ from its roots.

**Parameters**

- **roots**: `ArrayLike`
  The roots of the desired polynomial.

- **multiplicities**: `Sequence[int] | ndarray | None = None`
  The corresponding root multiplicities. The default is `None` which corresponds to all ones.

- **field**: `Type[Array] | None = None`
  The Galois field $\text{GF}(p^m)$ the polynomial is over.
  - `None` (default): If the roots are an `Array`, they won’t be modified. If the roots are not explicitly in a Galois field, they are assumed to be from $\text{GF}(2)$ and are converted using `galois.GF2(roots)`.
  - `Array` subclass: The roots are explicitly converted to this Galois field using `field(roots)`.

**Returns**

The polynomial $f(x)$.

**Notes**

The polynomial $f(x)$ with $k$ roots $\{r_1, r_2, \ldots, r_k\}$ with multiplicities $\{m_1, m_2, \ldots, m_k\}$ is

$$f(x) = (x - r_1)^{m_1} (x - r_2)^{m_2} \ldots (x - r_k)^{m_k}$$

$$= a_dx^d + a_{d-1}x^{d-1} + \cdots + a_1x + a_0$$

with degree $d = \sum_{i=1}^{k} m_i$. 
Examples

Construct a polynomial over \(\mathbb{GF}(2)\) from a list of its roots.

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

# Evaluate the polynomial at its roots
In [3]: f(roots)
Out[3]: GF([0, 0, 0], order=2)
```

Construct a polynomial over \(\mathbb{GF}(3^5)\) from a list of its roots with specific multiplicities.

```
In [4]: GF = galois.GF(3**5)
In [5]: roots = [121, 198, 225]
In [6]: f = galois.Poly.Roots(roots, multiplicities=[1, 2, 1], field=GF); f
Out[6]: Poly(x^4 + 215x^3 + 90x^2 + 183x + 119, GF(3^5))

# Evaluate the polynomial at its roots
In [7]: f(roots)
Out[7]: GF([0, 0, 0], order=3^5)
```

classmethod galois.Poly.Str(string: str, field: Type[Array] | None = None) → Poly

Constructs a polynomial over \(\mathbb{GF}(p^m)\) from its string representation.

\(\text{Str()}\) and \(\text{__str__()}\) are inverse operations.

Parameters

- **string**: str
  The string representation of the polynomial \(f(x)\).

- **field**: Type[Array] | None = None
  The Galois field \(\mathbb{GF}(p^m)\) the polynomial is over. The default is \(\text{None}\) which corresponds to \(\mathbb{GF}_2\).

Returns

The polynomial \(f(x)\).

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]: f = galois.Poly.Str("x^2 + 1"); f
Out[1]: Poly(x^2 + 1, GF(2))
```

```
In [2]: str(f)
Out[2]: 'x^2 + 1'
```

Construct a polynomial over GF(3^5) from its string representation.

```
In [3]: GF = galois.GF(3**5)
In [4]: f = galois.Poly.Str("13x^3 + 117", field=GF); f
Out[4]: Poly(13x^3 + 117, GF(3^5))
```

```
In [5]: str(f)
Out[5]: '13x^3 + 117'
```

**classmethod** `galois.Poly.Zero(field: Type[Array] | None = None) -> Poly`

Constructs the polynomial \( f(x) = 0 \) over GF\( (p^m) \).

**Parameters**

- **field**: Type[Array] | None = None
  The Galois field GF\( (p^m) \) the polynomial is over. The default is \( None \) which corresponds to GF2.

**Returns**

The polynomial \( f(x) = 0 \).

**Examples**

Construct the zero polynomial over GF(2).

```
In [1]: galois.Poly.Zero()
Out[1]: Poly(0, GF(2))
```

Construct the zero polynomial over GF(3^5).

```
In [2]: GF = galois.GF(3**5)
In [3]: galois.Poly.Zero(GF)
Out[3]: Poly(0, GF(3^5))
```
String representation

__repr__() → str
  A representation of the polynomial and the finite field it’s over.

__str__() → str
  The string representation of the polynomial, without specifying the finite field it’s over.

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

Examples

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

galois.Poly.__str__() → str
  The string representation of the polynomial, without specifying the finite field it’s over.
  Str() and __str__() are inverse operations.

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

Examples

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

_**call**_ (at: ElementLike | ArrayLike, ...) → Array

_**call**_ (at: Poly) → Poly

Evaluates the polynomial \( f(x) \) at \( x_0 \) or the polynomial composition \( f(g(x)) \).

_**eq**_ (other: PolyLike) → bool

Determines if two polynomials are equal.

_**int**_ () → int

The integer representation of the polynomial.

_**len**_ () → int

Returns the length of the coefficient array, which is equivalent to Poly.

```
galois.Poly._**call**_ (at: ElementLike | ArrayLike, field: Type[Array] | None = None, elementwise: bool = True) → Array
galois.Poly._**call**_ (at: Poly) → Poly
```

Evaluates the polynomial \( f(x) \) at \( x_0 \) or the polynomial composition \( f(g(x)) \).

**Parameters**

- **at: ElementLike | ArrayLike**
  - A finite field scalar or array \( x_0 \) to evaluate the polynomial at or the polynomial \( g(x) \) to evaluate the polynomial composition \( f(g(x)) \).

- **field: Type[Array] | None = None**
  - The Galois field to evaluate the polynomial over. The default is None which represents the polynomial’s current field, i.e. field.

- **elementwise: bool = True**
  - Indicates whether to evaluate \( x_0 \) element-wise. The default is True. If False (only valid for square matrices), the polynomial indeterminate \( x \) is exponentiated using matrix powers (repeated matrix multiplication).

**Returns**

The result of the polynomial evaluation \( f(x_0) \). The resulting array has the same shape as \( x_0 \). Or the polynomial composition \( f(g(x)) \).

**Examples**

Create a polynomial over \( GF(3^5) \).

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

In [2]: f = galois.Poly([37, 123, 0, 201], field=GF); f
Out[2]: Poly(37x^3 + 123x^2 + 201, GF(3^5))
```

Evaluate the polynomial element-wise at \( x_0 \).

```
In [3]: x0 = GF([185, 218, 84, 163])

In [4]: f(x0)
Out[4]: GF([ 33, 163, 146, 96], order=3^5)
```
# The equivalent calculation

In [5]: \( \text{GF}(37) \cdot x_0^3 + \text{GF}(123) \cdot x_0^2 + \text{GF}(201) \)

Out[5]: \( \text{GF}([33, 163, 146, 96], \text{order}=3^5) \)

Evaluate the polynomial at the square matrix \( X_0 \).

In [6]: \( X_0 = \text{GF}([\begin{bmatrix} 185 & 218 \\ 84 & 163 \end{bmatrix}]) \)

# This is performed element-wise. Notice the values are equal to the vector \( x_0 \).

In [7]: \( f(X_0) \)

Out[7]: \( \text{GF}([\begin{bmatrix} 33, 163 \\ 146, 96 \end{bmatrix}], \text{order}=3^5) \)

In [8]: \( f(X_0, \text{elementwise=False}) \)

Out[8]: \( \text{GF}([\begin{bmatrix} 103, 192 \\ 156, 10 \end{bmatrix}], \text{order}=3^5) \)

# The equivalent calculation

In [9]: \( \text{GF}(37) \cdot \text{np.linalg.matrix_power}(X_0, 3) + \text{GF}(123) \cdot \text{np.linalg.matrix_power}(X_0, 2) + \text{GF}(201) \cdot \text{GF}.\text{Identity}(2) \)

Out[9]: \( \text{GF}([\begin{bmatrix} 103, 192 \\ 156, 10 \end{bmatrix}], \text{order}=3^5) \)

Evaluate the polynomial \( f(x) \) at the polynomial \( g(x) \).

In [10]: \( g = \text{galois.Poly}([55, 0, 1], \text{field}=\text{GF}) \); \( g \)

Out[10]: \( \text{Poly}(55x^2 + 1, \text{GF}(3^5)) \)

In [11]: \( f(g) \)

Out[11]: \( \text{Poly}(77x^6 + 5x^4 + 104x^2 + 1, \text{GF}(3^5)) \)

# The equivalent calculation

In [12]: \( \text{GF}(37) \cdot g^3 + \text{GF}(123) \cdot g^2 + \text{GF}(201) \)

Out[12]: \( \text{Poly}(77x^6 + 5x^4 + 104x^2 + 1, \text{GF}(3^5)) \)

galois.Poly.__eq__(other: PolyLike) \rightarrow bool

Determines if two polynomials are equal.

Parameters

other: PolyLike

The polynomial to compare against.

Returns

True if the two polynomials have the same coefficients and are over the same finite field.
Examples

Compare two polynomials over the same field.

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

In [2]: b = galois.Poly([3, 0, 5], field=galois.GF(7)); b
Out[2]: Poly(3x^2 + 5, GF(7))

In [3]: a == b
Out[3]: True

# They are still two distinct objects, however
In [4]: a is b
Out[4]: False
```

Compare two polynomials with the same coefficients but over different fields.

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

In [6]: b = galois.Poly([3, 0, 5], field=galois.GF(7**2)); b
Out[6]: Poly(3x^2 + 5, GF(7^2))

In [7]: a == b
Out[7]: False
```

Comparison with PolyLike objects is allowed for convenience.

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

In [9]: a = galois.Poly([3, 0, 2], field=GF); a
Out[9]: Poly(3x^2 + 2, GF(7))

In [10]: a == GF([3, 0, 2])
Out[10]: True

In [11]: a == [3, 0, 2]
Out[11]: True

In [12]: a == "3x^2 + 2"
Out[12]: True

In [13]: a == 3*7**2 + 2
Out[13]: True
```

galois.Poly.__int__() → int
The integer representation of the polynomial.

Int() and __int__() are inverse operations.
Notes

For the polynomial $f(x) = a_dx^d + a_{d-1}x^{d-1} + \cdots + a_1x + a_0$ over the field $\mathbb{GF}(p^m)$, the integer representation is $i = a_d(p^m)^d + a_{d-1}(p^m)^{d-1} + \cdots + a_1(p^m) + a_0$ using integer arithmetic, not finite field arithmetic.

Said differently, the polynomial coefficients $\{a_d, a_{d-1}, \ldots, a_1, a_0\}$ are considered as the $d$ “digits” of a radix-$p^m$ value. The polynomial’s integer representation is that value in decimal (radix-10).

Examples

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

In [3]: int(f)
Out[3]: 1066

In [4]: int(f) == 3*GF.order**3 + 5*GF.order**1 + 2*GF.order**0
Out[4]: True
```

galois.Poly.__len__() → int

Returns the length of the coefficient array, which is equivalent to $\text{Poly}.\text{degree} + 1$.

Returns

The length of the coefficient array.

Examples

```python
In [1]: GF = galois.GF(3**5)
In [2]: f = galois.Poly([37, 123, 0, 201], field=GF); f
Out[2]: Poly(37x^3 + 123x^2 + 201, GF(3^5))

In [3]: f.coeffs
Out[3]: GF([ 37, 123, 0, 201], order=3^5)

In [4]: len(f)
Out[4]: 4

In [5]: f.degree + 1
Out[5]: 4
```
Methods

**coefficients** *(size: int | None = None, ... ) → Array*

Returns the polynomial coefficients in the order and size specified.

**derivative** *(k: int = 1) → Poly*

Computes the \( k \)-th formal derivative \( \frac{d^k}{dx^k} f(x) \) of the polynomial \( f(x) \).

**distinct_degree_factors** () → Tuple[List[Poly], List[int]]

Factors the monic, square-free polynomial \( f(x) \) into a product of polynomials whose irreducible factors all have the same degree.

**equal_degree_factors** *(degree: int) → List[Poly]*

Factors the monic, square-free polynomial \( f(x) \) of degree \( rd \) into a product of \( r \) irreducible factors with degree \( d \).

**factors** () → Tuple[List[Poly], List[int]]

Computes the irreducible factors of the non-constant, monic polynomial \( f(x) \).

**is_irreducible** () → bool

Determines whether the polynomial \( f(x) \) over GF(\( p^m \)) is irreducible.

**is_primitive** () → bool

Determines whether the polynomial \( f(x) \) over GF(\( q \)) is primitive.

**is_square_free** () → bool

Determines whether the polynomial \( f(x) \) over GF(\( q \)) is square-free.

**reverse** () → Poly

Returns the \( d \)-th reversal \( x^d f(\frac{1}{x}) \) of the polynomial \( f(x) \) with degree \( d \).

**roots** *(multiplicity: False = False) → Array*

**roots** *(multiplicity: True) → Tuple[Array, ndarray]*

Calculates the roots \( r \) of the polynomial \( f(x) \), such that \( f(r) = 0 \).

**square_free_factors** () → Tuple[List[Poly], List[int]]

Factors the monic polynomial \( f(x) \) into a product of square-free polynomials.

**galois.Poly.coefficients** *(size: int | None = None, order: 'desc' | 'asc' = 'desc') → Array*

Returns the polynomial coefficients in the order and size specified.

**Parameters**

- **size**: int | None = None
  
The fixed size of the coefficient array. Zeros will be added for higher-order terms. This value must be at least \( \text{degree} + 1 \) or a ValueError will be raised. The default is None which corresponds to \( \text{degree} + 1 \).

- **order**: 'desc' | 'asc' = 'desc'
  
The order of the coefficient degrees, either descending (default) or ascending.

**Returns**

An array of the polynomial coefficients with length size, either in descending order or ascending order.
Notes

This accessor is similar to the `coeffs` property, but it has more settings. By default, \( \text{Poly.coeffs} == \text{Poly.coefficients}() \).

Examples

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

Return the coefficients in ascending order.

In [5]: f.coeficients(order="asc")
Out[5]: GF([2, 5, 0, 3], order=7)

Return the coefficients in ascending order with size 8.

In [6]: f.coeficients(8, order="asc")
Out[6]: GF([2, 5, 0, 3, 0, 0, 0, 0], order=7)

galois.Poly.derivative(k: int = 1) \( \rightarrow \) Poly

Computes the \( k \)-th formal derivative \( \frac{d^k}{dx^k} f(x) \) of the polynomial \( f(x) \).

Parameters

- **k**: int = 1
  - The number of derivatives to compute. 1 corresponds to \( p'(x) \), 2 corresponds to \( p''(x) \), etc. The default is 1.

Returns

- The \( k \)-th formal derivative of the polynomial \( f(x) \).

Notes

For the polynomial

\[
f(x) = a_dx^d + a_{d-1}x^{d-1} + \cdots + a_1x + a_0
\]

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. The exponent that is “brought down” and multiplied by the coefficient is an integer, not a finite field element. For example, 3 \( \cdot \) a = a + a + a.
References


Examples

Compute the derivatives of a polynomial over GF(2).

```python
In [1]: f = galois.Poly.Random(7); f
Out[1]: Poly(x^7 + x^4 + x^2, GF(2))

In [2]: f.derivative()
Out[2]: Poly(x^6, GF(2))

# p derivatives of a polynomial, where p is the field's characteristic, will always result in 0
In [3]: f.derivative(GF.characteristic)
Out[3]: Poly(0, GF(2))
```

Compute the derivatives of a polynomial over GF(7).

```python
In [4]: GF = galois.GF(7)

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

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

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

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

# p derivatives of a polynomial, where p is the field's characteristic, will always result in 0
In [9]: f.derivative(GF.characteristic)
Out[9]: Poly(0, GF(7))
```

Compute the derivatives of a polynomial over GF(3^5).

```python
In [10]: GF = galois.GF(3**5)

In [11]: f = galois.Poly.Random(7, field=GF); f
Out[11]: Poly(54x^7 + 185x^6 + 215x^5 + 242x^4 + 80x^3 + 62x^2 + 189x + 130, GF(3^5))

In [12]: f.derivative()
Out[12]: Poly(54x^6 + 148x^4 + 242x^3 + 31x + 189, GF(3^5))
```

(continues on next page)
In [13]: f.derivative(2)
Out[13]: Poly(148x^3 + 31, GF(3^5))

# p derivatives of a polynomial, where p is the field's characteristic, will always result in 0
In [14]: f.derivative(GF.characteristic)
Out[14]: Poly(0, GF(3^5))

```
galois.Poly.distinct_degree_factors() \rightarrow \text{Tuple}[\text{List[Poly]}, \text{List[int]}]

Factors the monic, square-free polynomial \( f(x) \) into a product of polynomials whose irreducible factors all have the same degree.

Returns

• The list of polynomials \( f_i(x) \) whose irreducible factors all have degree \( i \).
• The list of corresponding distinct degrees \( i \).

Raises

ValueError – If \( f(x) \) is not monic, has degree 0, or is not square-free.
```

Notes

The Distinct-Degree Factorization algorithm factors a square-free polynomial \( f(x) \) with degree \( d \) into a product of \( d \) polynomials \( f_i(x) \), where \( f_i(x) \) is the product of all irreducible factors of \( f(x) \) with degree \( i \).

\[
f(x) = \prod_{i=1}^{d} f_i(x)
\]

For example, suppose \( f(x) = x(x + 1)(x^2 + x + 1)(x^3 + x + 1)(x^3 + x^2 + 1) \) over GF(2), 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 \texttt{square_free_factors()}. A complete polynomial factorization is implemented in \texttt{factors()}.

References

• Hachenberger, D. and Jungnickel, D. Topics in Galois Fields. Algorithm 6.2.2.
• Section 2.2 from https://people.csail.mit.edu/dmoshkov/courses/codes/poly-factorization.pdf
Examples

From the example in the notes, suppose \( f(x) = x(x+1)(x^2 + x + 1)(x^3 + x + 1)(x^3 + x^2 + 1) \) over GF(2).

```
In [1]: a = galois.Poly([1, 0]); a, a.is_irreducible()
Out[1]: (Poly(x, GF(2)), True)

In [2]: b = galois.Poly([1, 1]); b, b.is_irreducible()
Out[2]: (Poly(x + 1, GF(2)), True)

In [3]: c = galois.Poly([1, 1, 1]); c, c.is_irreducible()
Out[3]: (Poly(x^2 + x + 1, GF(2)), True)

In [4]: d = galois.Poly([1, 0, 1, 1]); d, d.is_irreducible()
Out[4]: (Poly(x^3 + x + 1, GF(2)), True)

In [5]: e = galois.Poly([1, 1, 0, 1]); e, e.is_irreducible()
Out[5]: (Poly(x^3 + x^2 + 1, GF(2)), True)

In [6]: f = a * b * c * d * e; f
Out[6]: Poly(x^10 + x^9 + x^8 + x^3 + x^2 + x, GF(2))
```

The distinct-degree factorization is \( \{x(x+1), x^2 + x + 1, (x^3 + x + 1)(x^3 + x^2 + 1)\} \) whose irreducible factors have degrees \{1, 2, 3\}.

```
In [7]: f.distinct_degree_factors()
Out[7]: ([Poly(x^2 + x, GF(2)), Poly(x^2 + x + 1, GF(2)), Poly(x^6 + x^5 + x^4 + x^3 + x^2 + x + 1, GF(2))], [1, 2, 3])

In [8]: [a*b, c, d*e], [1, 2, 3]
Out[8]: ([Poly(x^10 + x^9 + x^8 + x^3 + x^2 + x, GF(2))], [1, 2, 3])
```

galois.Poly.equal_degree_factors(degree: int) \( \rightarrow \) List[Poly]

Factors the monic, square-free polynomial \( f(x) \) of degree \( rd \) into a product of \( r \) irreducible factors with degree \( d \).

Parameters

- **degree**: int
  
The degree \( d \) of each irreducible factor of \( f(x) \).

Returns

The list of \( r \) irreducible factors \( \{g_1(x), \ldots, g_r(x)\} \) in lexicographically-increasing order.

Raises

- **ValueError** – If \( f(x) \) is not monic, has degree 0, or is not square-free.
Notes

The Equal-Degree Factorization algorithm factors a square-free polynomial \( f(x) \) with degree \( rd \) into a product of \( r \) irreducible polynomials each with degree \( d \). This function implements the Cantor-Zassenhaus algorithm, which is probabilistic.

The Equal-Degree Factorization algorithm is often applied after the Distinct-Degree Factorization algorithm, see \texttt{distinct_degree_factors()}\). A complete polynomial factorization is implemented in \texttt{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 \( GF(2) \).

```python
In [1]: a = galois.Poly([1, 0]); a, a.is_irreducible()
Out[1]: (Poly(x, GF(2)), True)

In [2]: b = galois.Poly([1, 1]); b, b.is_irreducible()
Out[2]: (Poly(x + 1, GF(2)), True)

In [3]: f = a * b; f
Out[3]: Poly(x^2 + x, GF(2))

In [4]: f.equal_degree_factors(1)
Out[4]: [Poly(x, GF(2)), Poly(x + 1, GF(2))]
```

Factor a product of degree-3 irreducible polynomials over \( GF(5) \).

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

In [6]: a = galois.Poly([1, 0, 2, 1], field=GF); a, a.is_irreducible()
Out[6]: (Poly(x^3 + 2x + 1, GF(5)), True)

In [7]: b = galois.Poly([1, 4, 4, 4], field=GF); b, b.is_irreducible()
Out[7]: (Poly(x^3 + 4x^2 + 4x + 4, GF(5)), True)

In [8]: f = a * b; f
Out[8]: Poly(x^6 + 4x^5 + x^4 + 3x^3 + 2x^2 + 2x + 4, GF(5))

In [9]: f.equal_degree_factors(3)
Out[9]: [Poly(x^3 + 2x + 1, GF(5)), Poly(x^3 + 4x^2 + 4x + 4, GF(5))]
```

galois.Poly.factors() \rightarrow \text{Tuple}[\text{List}[\text{Poly}], \text{List}[\text{int}]]

Computes the irreducible factors of the non-constant, monic polynomial \( f(x) \).

Returns

- Sorted list of irreducible factors \( \{g_1(x), g_2(x), \ldots, g_k(x)\} \) of \( f(x) \) sorted in lexicographically-increasing order.
• List of corresponding multiplicities \( \{e_1, e_2, \ldots, e_k\} \).

**Raises**

*ValueError* – If \( f(x) \) is not monic or has degree 0.

**Notes**

This function factors a monic polynomial \( f(x) \) into its \( k \) irreducible factors such that \( f(x) = g_1(x)^{e_1} g_2(x)^{e_2} \ldots g_k(x)^{e_k} \).

Steps:

1. Apply the Square-Free Factorization algorithm to factor the monic polynomial into square-free polynomials. See `square_free_factors()`.
2. Apply the Distinct-Degree Factorization algorithm to factor each square-free polynomial into a product of factors with the same degree. See `distinct_degree_factors()`.
3. Apply the Equal-Degree Factorization algorithm to factor the product of factors of equal degree into their irreducible factors. See `equal_degree_factors()`.

**References**

• Hachenberger, D. and Jungnickel, D. Topics in Galois Fields. Algorithm 6.1.7.
• Section 2.1 from [https://people.csail.mit.edu/dmoshkov/courses/codes/poly-factorization.pdf](https://people.csail.mit.edu/dmoshkov/courses/codes/poly-factorization.pdf)

**Examples**

Generate irreducible polynomials over GF(3).

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

In [2]: g1 = galois.irreducible_poly(3, 3); g1
Out[2]: Poly(x^3 + 2x + 1, GF(3))

In [3]: g2 = galois.irreducible_poly(3, 4); g2
Out[3]: Poly(x^4 + x + 2, GF(3))

In [4]: g3 = galois.irreducible_poly(3, 5); g3
Out[4]: Poly(x^5 + 2x + 1, GF(3))
```

Construct a composite polynomial.

```python
In [5]: e1, e2, e3 = 5, 4, 3

In [6]: f = g1**e1 * g2**e2 * g3**e3; f
Out[6]: Poly(x^46 + x^44 + 2x^43 + x^39 + 2x^38 + 2x^37 + 2x^36 + 2x^34
  + x^33 + 2x^32 + x^31 + 2x^30 + 2x^29 + 2x^28 + 2x^25 + 2x^24 + 2x^23 + x^20
  + x^19 + x^18 + x^15 + 2x^10 + 2x^8 + x^5 + x^4 + x^3 + 1, GF(3))
```

Factor the polynomial into its irreducible factors over GF(3).
```python
In [7]: f.factors()
Out[7]:
([Poly(x^3 + 2x + 1, GF(3)),
Poly(x^4 + x + 2, GF(3)),
Poly(x^5 + 2x + 1, GF(3))],
[5, 4, 3])
```

galois.Poly.is_irreducible() → bool

Determines whether the polynomial \( f(x) \) over GF\( (p^m) \) is irreducible.

**Returns**

- **True** if the polynomial is irreducible.

**Important:** This is a method, not a property, to indicate this test is computationally expensive.

**See also:**

- `irreducible_poly`, `irreducible_polys`

**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](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]: f.is_irreducible()
Out[3]: True

In [4]: g = galois.irreducible_poly(2**4, 2, method="random"); g
Out[4]: Poly(x^2 + 13x + 4, GF(2^4))

In [5]: h = galois.irreducible_poly(2**4, 3, method="random"); h
Out[5]: Poly(x^3 + 3x^2 + 3x + 4, GF(2^4))

In [6]: f = g * h; f
Out[6]: Poly(x^5 + 14x^4 + 3x^3 + 12x^2 + 13x + 3, GF(2^4))

In [7]: f.is_irreducible()
Out[7]: False

galois.Poly.is_primitive() \rightarrow \text{bool}

Determines whether the polynomial \( f(x) \) over \( GF(q) \) is primitive.

Returns

\text{True} if the polynomial is primitive.

Important: This is a method, not a property, to indicate this test is computationally expensive.

See also:

\text{primitive_poly, primitive_polys, conway_poly, matlabPrimitive_poly}

Notes

A degree-\( m \) polynomial \( f(x) \) over \( GF(q) \) is \textit{primitive} if it is irreducible and \( f(x) \mid (x^k - 1) \) for \( k = q^m - 1 \) and no \( k \) less than \( q^m - 1 \).

References

• Algorithm 4.77 from https://cacr.uwaterloo.ca/hac/about/chap4.pdf

Examples

All Conway polynomials are primitive.

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

In [2]: f.is_primitive()
Out[2]: True

In [3]: f = galois.conway_poly(3, 5); f
Out[3]: Poly(x^5 + 2x + 1, GF(3))
In [4]: f.is_primitive()
Out[4]: True

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

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

In [6]: f.is_irreducible()
Out[6]: True

In [7]: f.is_primitive()
Out[7]: False

galois.Poly.is_square_free() → bool

Determines whether the polynomial $f(x)$ over GF($q$) is square-free.

Returns

- True if the polynomial is square-free.

Important: This is a method, not a property, to indicate this test is computationally expensive.

Notes

A square-free polynomial $f(x)$ has no irreducible factors with multiplicity greater than one. Therefore, its canonical factorization is

$$f(x) = \prod_{i=1}^{k} g_i(x)^{e_i} = \prod_{i=1}^{k} g_i(x).$$

Examples

Generate irreducible polynomials over GF(3).

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

In [2]: f1 = galois.irreducible_poly(3, 3); f1
Out[2]: Poly(x^3 + 2x + 1, GF(3))

In [3]: f2 = galois.irreducible_poly(3, 4); f2
Out[3]: Poly(x^4 + x + 2, GF(3))

Determine if composite polynomials are square-free over GF(3).

In [4]: (f1 * f2).is_square_free()
Out[4]: True

In [5]: (f1**2 * f2).is_square_free()
Out[5]: False
galois.Poly.reverse() → Poly
Returns the $d$-th reversal $x^d f(x^{-1})$ of the polynomial $f(x)$ with degree $d$.

Notes
For a polynomial $f(x) = a_d x^d + a_{d-1} x^{d-1} + \cdots + a_1 x + a_0$ with degree $d$, the $d$-th reversal is equivalent to reversing the coefficients.
$$
rev_d f(x) = x^d f(x^{-1}) = a_0 x^d + a_1 x^{d-1} + \cdots + a_{d-1} x + a_d
$$

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)) |


galois.Poly.roots(multiplicity: False = False) → Array

galois.Poly.roots(multiplicity: True) → Tuple[Array, ndarray]
Calculates the roots $r$ of the polynomial $f(x)$, such that $f(r) = 0$.

Parameters

multiplicity: False = False
Optionally return the multiplicity of each root. The default is False which only returns the unique roots.

Returns

- An array of roots of $f(x)$. The roots are ordered in increasing order.
- The multiplicity of each root. This is only returned if multiplicity=True.

Notes

This implementation uses Chien’s search to find the roots $\{r_1, r_2, \ldots, r_k\}$ of the degree-$d$ polynomial
$$
f(x) = a_d x^d + a_{d-1} x^{d-1} + \cdots + a_1 x + a_0,
$$
where $k \leq d$. Then, $f(x)$ can be factored as
$$
f(x) = (x - r_1)^{m_1} (x - r_2)^{m_2} \cdots (x - r_k)^{m_k},
$$
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 $\text{GF}(p^m)$. 

186 Chapter 3. Citation
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 \( f(\alpha^i) = 0 \).

\[
f(\alpha^i) = a_d(\alpha^i)^d + a_{d-1}(\alpha^i)^{d-1} + \cdots + a_1(\alpha^i) + a_0
\]

\[
= \lambda_{i,d} + \lambda_{i,d-1} + \cdots + \lambda_{i,1} + \lambda_{i,0}
\]

\[
= \sum_{j=0}^{d} \lambda_{i,j}
\]

The next power of \( \alpha \) can be easily calculated from the previous calculation.

\[
f(\alpha^{i+1}) = a_d(\alpha^{i+1})^d + a_{d-1}(\alpha^{i+1})^{d-1} + \cdots + a_1(\alpha^{i+1}) + a_0
\]

\[
= a_d(\alpha^i)^d \alpha^d + a_{d-1}(\alpha^i)^{d-1} \alpha^{d-1} + \cdots + a_1(\alpha^i) \alpha + a_0
\]

\[
= \lambda_{i,d} \alpha^d + \lambda_{i,d-1} \alpha^{d-1} + \cdots + \lambda_{i,1} \alpha + \lambda_{i,0}
\]

\[
= \sum_{j=0}^{d} \lambda_{i,j} \alpha^j
\]

Examples

Find the roots of a polynomial over \( \text{GF}(2) \).

```
In [1]: f = galois.Poly.Roots([1, 0], multiplicities=[7, 3]); f
Out[1]: Poly(x^10 + x^9 + x^8 + x^7 + x^6 + x^5 + x^4 + x^3, GF(2))
```

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

```
In [3]: f.roots(multiplicity=True)
Out[3]: (GF([0, 1], order=2), array([3, 7]))
```

Find the roots of a polynomial over \( \text{GF}(3^5) \).

```
In [4]: GF = galois.GF(3**5)
```

```
In [5]: f = galois.Poly.Roots([18, 227, 153], multiplicities=[5, 7, 3],
-> field=GF); f
Out[5]: Poly(x^15 + 118x^14 + 172x^13 + 50x^12 + 204x^11 + 202x^10 + 141x^9 +
-> 153x^8 + 107x^7 + 187x^6 + 66x^5 + 221x^4 + 114x^3 + 121x^2 + 226x + 13, GF(3^5))
```

```
In [6]: f.roots()
Out[6]: GF([ 18, 153, 227], order=3^5)
```

```
In [7]: f.roots(multiplicity=True)
Out[7]: (GF([ 18, 153, 227], order=3^5), array([5, 3, 7]))
```

galois.Poly.square_free_factors() → Tuple[List[Poly], List[int]]
Factors the monic polynomial \( f(x) \) into a product of square-free polynomials.

**Returns**

- The list of non-constant, square-free polynomials \( h_j(x) \) in the factorization.
• The list of corresponding multiplicities \( j \).

**Raises**

- **ValueError** – If \( f(x) \) is not monic or has degree 0.

**Notes**

The Square-Free Factorization algorithm factors \( f(x) \) into a product of \( m \) square-free polynomials \( h_j(x) \) with multiplicity \( j \).

\[
f(x) = \prod_{j=1}^{m} h_j(x)^j
\]

Some \( h_j(x) = 1 \), but those polynomials are not returned by this function.

A complete polynomial factorization is implemented in \texttt{factors()}.

**References**

- Section 2.1 from https://people.csail.mit.edu/dmoshkov/courses/codes/poly-factorization.pdf

**Examples**

Suppose \( f(x) = x(x^3 + 2x + 4)(x^2 + 4x + 1)^3 \) over \( \text{GF}(5) \). Each polynomial \( x, x^3 + 2x + 4, \) and \( x^2 + 4x + 1 \) are all irreducible over \( \text{GF}(5) \).

```python
In [1]: GF = galois.GF(5)
In [2]: a = galois.Poly([1,0], field=GF); a, a.is_irreducible()
Out[2]: (Poly(x, GF(5)), True)
In [3]: b = galois.Poly([1,0,2,4], field=GF); b, b.is_irreducible()
Out[3]: (Poly(x^3 + 2x + 4, GF(5)), True)
In [4]: c = galois.Poly([1,4,1], field=GF); c, c.is_irreducible()
Out[4]: (Poly(x^2 + 4x + 1, GF(5)), True)
In [5]: f = a * b * c**3; f
Out[5]: Poly(x^10 + 2x^9 + 3x^8 + x^7 + x^6 + 2x^5 + 3x^3 + 4x, GF(5))
```

The square-free factorization is \( \{x(x^3 + 2x + 4), x^2 + 4x + 1\} \) with multiplicities \( \{1, 3\} \).

```python
In [6]: f.square_free_factors()
Out[6]: ([Poly(x^4 + 2x^2 + 4x, GF(5)), Poly(x^2 + 4x + 1, GF(5))], [1, 3])
In [7]: [a*b, c], [1, 3]
Out[7]: ([Poly(x^4 + 2x^2 + 4x, GF(5)), Poly(x^2 + 4x + 1, GF(5))], [1, 3])
```
Properties

**property** `coeffs`: `Array`

The coefficients of the polynomial in degree-descending order. The entries of `coeffs` are paired with `degrees`.

**property** `degree`: `int`

The degree of the polynomial. The degree of a polynomial is the highest degree with a non-zero coefficient.

**property** `degrees`: `ndarray`

An array of the polynomial degrees in descending order. The entries of `coeffs` are paired with `degrees`.

**property** `field`: `Type[Array]`

The `Array` subclass for the finite field the coefficients are over.

**property** `is_monic`: `bool`

Returns whether the polynomial is monic, meaning its highest-degree coefficient is one.

**property** `nonzero_coeffs`: `Array`

The non-zero coefficients of the polynomial in degree-descending order. The entries of `nonzero_coeffs` are paired with `nonzero_degrees`.

**property** `nonzero_degrees`: `ndarray`

An array of the polynomial degrees that have non-zero coefficients in descending order. The entries of `nonzero_coeffs` are paired with `nonzero_degrees`.

**property** `galois.Poly.coeffs`: `Array`

The coefficients of the polynomial in degree-descending order. The entries of `coeffs` are paired with `degrees`.

Examples

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

In [2]: p = galois.Poly([3, 0, 5, 2], field=GF); p
Out[2]: Poly(3x^3 + 5x + 2, GF(7))

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

**property** `galois.Poly.degree`: `int`

The degree of the polynomial. The degree of a polynomial is the highest degree with a non-zero coefficient.

Examples

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

In [2]: p = galois.Poly([3, 0, 5, 2], field=GF); p
Out[2]: Poly(3x^3 + 5x + 2, GF(7))

In [3]: p.degree
Out[3]: 3
```
**property** galois.Poly.degrees : ndarray
An array of the polynomial degrees in descending order. The entries of *coeffs* are paired with *degrees*.

**Examples**

```python
In [1]: GF = galois.GF(7)
In [2]: p = galois.Poly([3, 0, 5, 2], field=GF); p
Out[2]: Poly(3x^3 + 5x + 2, GF(7))
In [3]: p.degrees  
Out[3]: array([3, 2, 1, 0])
```

**property** galois.Poly.field : Type[Array]
The *Array* subclass for the finite field the coefficients are over.

**Examples**

```python
In [1]: a = galois.Poly.Random(5); a  
Out[1]: Poly(x^5 + x^3, GF(2))
In [2]: a.field  
Out[2]: galois.GF2
```

```python
In [3]: GF = galois.GF(2**8)
In [4]: b = galois.Poly.Random(5, field=GF); b  
Out[4]: Poly(172x^5 + 216x^4 + 111x^3 + 237x^2 + 75x + 141, GF(2^8))
In [5]: b.field  
Out[5]: galois.GF(2^8)
```

**property** galois.Poly.is_monic : bool
Returns whether the polynomial is monic, meaning its highest-degree coefficient is one.

**Examples**

A monic polynomial over GF(7).

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

A non-monic polynomial over GF(7).
In [4]: GF = galois.GF(7)

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

In [6]: p.is_monic
Out[6]: False

**property** galois.Poly.nonzero_coeffs: Array

The non-zero coefficients of the polynomial in degree-descending order. The entries of nonzero_coeffs are paired with nonzero_degrees.

**Examples**

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

In [2]: p = galois.Poly([3, 0, 5, 2], field=GF); p
Out[2]: Poly(3x^3 + 5x + 2, GF(7))

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

**property** galois.Poly.nonzero_degrees: ndarray

An array of the polynomial degrees that have non-zero coefficients in descending order. The entries of nonzero_coeffs are paired with nonzero_degrees.

**Examples**

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

In [2]: p = galois.Poly([3, 0, 5, 2], field=GF); p
Out[2]: Poly(3x^3 + 5x + 2, GF(7))

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

**galois.typing.PolyLike**

A Union representing objects that can be coerced into a polynomial.

**Union**

- **int**: A polynomial in its integer representation, see Int(). The Galois field must be known from context.

  # Known from context
  In [1]: GF = galois.GF(3)

  In [2]: galois.Poly.Int(19, field=GF)
  Out[2]: Poly(2x^2 + 1, GF(3))

- **str**: A polynomial in its string representation, see Str(). The Galois field must be known from context.
In 

\begin{verbatim}
[3]: galois.Poly.Str("2x^2 + 1", field=GF)

Out[3]: Poly(2x^2 + 1, GF(3))
\end{verbatim}

- **ArrayLike**: An array of polynomial coefficients in degree-descending order. If the coefficients are not `Array`, then the Galois field must be known from context.

In 

\begin{verbatim}
[4]: galois.Poly([2, 0, 1], field=GF)

Out[4]: Poly(2x^2 + 1, GF(3))
\end{verbatim}

In 

\begin{verbatim}
[5]: galois.Poly(GF([2, 0, 1]))

Out[5]: Poly(2x^2 + 1, GF(3))
\end{verbatim}

- **Poly**: A previously-created `Poly` object. No coercion is necessary.

**Alias**

alias of `Union[int, str, Sequence[Union[int, str, Array]], Sequence[IterableLike], ndarray, Array, Poly]`

**Important**: The *Number theory* section contains many functions that operate on polynomials.

**Irreducible polynomials**

\begin{verbatim}
galois.irreducible_poly(order: int, degree: int, ...) → Poly

Returns a monic irreducible polynomial \( f(x) \) over GF\( (q) \) with degree \( m \).
\end{verbatim}

\begin{verbatim}
galois.irreducible_polys(order: int, degree, ...) → Iterator[Polys]

Iterates through all monic irreducible polynomials \( f(x) \) over GF\( (q) \) with degree \( m \).
\end{verbatim}

\begin{verbatim}
galois.irreducible_poly(order: int, degree: int, method: 'min' | 'max' | 'random' = 'min') → Poly

Returns a monic irreducible polynomial \( f(x) \) over GF\( (q) \) with degree \( m \).
\end{verbatim}

**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 irreducible polynomial.

- **method**: `min` | `max` | `random` = `min`
  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 GF\( (q) \).

**See also**:

`Poly.is_irreducible`, `primitive_poly`, `conway_poly`
Notes
If \( f(x) \) is an irreducible polynomial over \( GF(q) \) and \( a \in GF(q) \setminus \{0\} \), then \( a \cdot f(x) \) is also irreducible.
In addition to other applications, \( f(x) \) produces the field extension \( GF(q^m) \) of \( GF(q) \).

Examples
Find the lexicographically minimal and maximal monic irreducible polynomial. Also find a random monic irreducible polynomial.

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

In [2]: galois.irreducible_poly(7, 3, method="max")
Out[2]: Poly(x^3 + 6x^2 + 6x + 4, GF(7))

In [3]: galois.irreducible_poly(7, 3, method="random")
Out[3]: Poly(x^3 + x + 6, GF(7))
```

Monic irreducible polynomials scaled by non-zero field elements (now non-monic) are also irreducible.

```
In [4]: GF = galois.GF(7)
In [5]: f = galois.irreducible_poly(7, 5, method="random"); f
Out[5]: Poly(x^5 + 2x^4 + 3x^3 + 4x^2 + 3x + 3, GF(7))

In [6]: f.is_irreducible()
Out[6]: True

In [7]: g = f * GF(3); g
Out[7]: Poly(3x^5 + 6x^4 + 2x^3 + 5x^2 + 2x + 2, GF(7))

In [8]: g.is_irreducible()
Out[8]: True
```

galois.irreducible_polys(order: int, degree: int, reverse: bool = False) → Iterator[Poly]
Iterates through all monic irreducible polynomials \( f(x) \) over \( GF(q) \) with degree \( m \).

Parameters
- \textbf{order: int}
  The prime power order \( q \) of the field \( GF(q) \) that the polynomial is over.
- \textbf{degree: int}
  The degree \( m \) of the desired irreducible polynomial.
- \textbf{reverse: bool = False}
  Indicates to return the irreducible polynomials from lexicographically maximal to minimal. The default is \textbf{False}.

Returns
An iterator over all degree-\( m \) monic irreducible polynomials over \( GF(q) \).

See also:
- \textit{Poly.is_irreducible}
- \textit{primitive_polys}
Notes

If \( f(x) \) is an irreducible polynomial over \( GF(q) \) and \( a \in GF(q) \setminus \{0\} \), then \( a \cdot f(x) \) is also irreducible.

In addition to other applications, \( f(x) \) produces the field extension \( GF(q^m) \) of \( GF(q) \).

Examples

Find all monic irreducible polynomials over \( GF(3) \) with degree 4. You may also use `tuple()` on the returned generator.

```
In [1]: list(galois.irreducible_polys(3, 4))
Out[1]:
[Poly(x^4 + x + 2, GF(3)),
 Poly(x^4 + 2x + 2, GF(3)),
 Poly(x^4 + x^2 + 2, GF(3)),
 Poly(x^4 + x^2 + x + 1, GF(3)),
 Poly(x^4 + 2x + 1, GF(3)),
 Poly(x^4 + x^3 + 2, GF(3)),
 Poly(x^4 + x^3 + 2x + 1, GF(3)),
 Poly(x^4 + x^3 + x^2 + 1, GF(3)),
 Poly(x^4 + x^3 + x^2 + x + 1, GF(3)),
 Poly(x^4 + x^3 + x^2 + 2x + 2, GF(3)),
 Poly(x^4 + x^3 + 2x^2 + 2x + 2, GF(3)),
 Poly(x^4 + x^3 + 2x + 1, GF(3)),
 Poly(x^4 + x^3 + x + 1, GF(3)),
 Poly(x^4 + x^3 + x^2 + 1, GF(3)),
 Poly(x^4 + 2x^3 + 2, GF(3)),
 Poly(x^4 + 2x^3 + x + 1, GF(3)),
 Poly(x^4 + 2x^3 + x^2 + 1, GF(3)),
 Poly(x^4 + 2x^3 + x^2 + 2x + 2, GF(3)),
 Poly(x^4 + 2x^3 + 2x^2 + x + 2, GF(3)),
 Poly(x^4 + 2x^3 + 2x^2 + x + 2, GF(3)),
 Poly(x^4 + 2x^3 + x^2 + 2x + 1, GF(3)),
 Poly(x^4 + 2x^3 + 2x^2 + x + 2, GF(3)),
 Poly(x^4 + 2x^3 + 2x^2 + x + 2, GF(3))]
```

Loop over all the polynomials in reversed order, only finding them as needed. The search cost for the polynomials that would have been found after the `break` condition is never incurred.

```
In [2]: for poly in galois.irreducible_polys(3, 4, reverse=True):
    ...
    if poly.coeffs[1] < 2:  # Early exit condition
        break
    ...
    print(poly)
    ...

x^4 + 2x^3 + 2x^2 + x + 2
x^4 + 2x^3 + x^2 + 2x + 1
x^4 + 2x^3 + x^2 + x + 2
x^4 + 2x^3 + x^2 + 1
x^4 + 2x^3 + x + 1
x^4 + 2x^3 + 2
```

Or, manually iterate over the generator.

```
In [3]: generator = galois.irreducible_polys(3, 4, reverse=True); generator
Out[3]: <generator object irreducible_polys at 0x7f6c1990b900>
In [4]: next(generator)
Out[4]: Poly(x^4 + 2x^3 + 2x^2 + x + 2, GF(3))
```

(continues on next page)
In [5]: `next(generator)`  
Out[5]: `Poly(x^4 + 2x^3 + x^2 + 2x + 1, GF(3))`  
In [6]: `next(generator)`  
Out[6]: `Poly(x^4 + 2x^3 + x^2 + x + 2, GF(3))`

**Primitive polynomials**

galois.conway_poly (characteristic: int, degree: int) → Poly  
Returns the Conway polynomial $C_{p,m}(x)$ over $\text{GF}(p)$ with degree $m$.

galois.matlab_primitive_poly (characteristic: int, degree) → Poly  
Returns Matlab’s default primitive polynomial $f(x)$ over $\text{GF}(p)$ with degree $m$.

galois.primitive_poly (order: int, degree: int, ...) → Poly  
Returns a monic primitive polynomial $f(x)$ over $\text{GF}(q)$ with degree $m$.

galois.primitive_polys (order: int, degree, ...) → Iterator[Poly]  
Iterates through all monic primitive polynomials $f(x)$ over $\text{GF}(q)$ with degree $m$.

galois.conway_poly (characteristic: int, degree: int) → Poly  
Returns the Conway polynomial $C_{p,m}(x)$ over $\text{GF}(p)$ with degree $m$.

**Parameters**

- characteristic: int  
The prime characteristic $p$ of the field $\text{GF}(p)$ 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 $\text{GF}(p)$.

**See also:**  
Poly.is_primitive, primitive_poly, matlab_primitive_poly

**Raises**

- LookupError – If the Conway polynomial $C_{p,m}(x)$ is not found in Frank Luebeck’s database.

**Notes**

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

The Conway polynomial $C_{p,m}(x)$ is defined as the lexicographically-minimal monic primitive polynomial of degree $m$ over $\text{GF}(p)$ 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 `primitive_poly()` returns the lexicographically-minimal primitive polynomial but `conway_poly()` returns the lexicographically-minimal primitive polynomial that is consistent with smaller Conway polynomials. This is sometimes the same polynomial.

```
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))
```

However, it is not always.

```
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(characteristic: int, degree: int) → Poly

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($p$).

**See also**:
`Poly.is_primitive, primitive_poly, conway_poly`

**Notes**

This function returns the same result as Matlab’s `gfprimdf(m, p)`. Matlab uses the 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 $x^{14} + x^{10} + x^6 + x + 1$, not $x^{14} + x^5 + x^3 + x + 1$.
3. GF($2^{16}$) uses $x^{16} + x^{12} + x^3 + x + 1$, not $x^{16} + x^5 + x^3 + x^2 + 1$.

**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.
References

• Lin, S. and Costello, D. Error Control Coding. Table 2.7.

Examples

```
In [1]: galois.primitive_poly(2, 6)
Out[1]: Poly(x^6 + x + 1, GF(2))

In [2]: galois.matlab_primitive_poly(2, 6)
Out[2]: Poly(x^6 + x + 1, GF(2))

In [3]: galois.primitive_poly(2, 7)
Out[3]: Poly(x^7 + x + 1, GF(2))

In [4]: galois.matlab_primitive_poly(2, 7)
Out[4]: Poly(x^7 + x^3 + 1, GF(2))
```

galois.primitive_poly(order: int, degree: int, method: 'min' | 'max' | 'random' = 'min') → Poly

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: 'min' | 'max' | 'random' = 'min'
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\( (q) \).

See also:
Poly.is_primitive, matlab_primitive_poly, conway_poly

Notes
If \( f(x) \) is a primitive polynomial over GF\( (q) \) and \( a \in GF(q) \setminus \{0\} \), then \( a \cdot f(x) \) is also primitive.

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

Find the lexicographically minimal and maximal monic primitive polynomial. Also find a random monic primitive polynomial.

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

In [2]: galois.primitive_poly(7, 3, method="max")
Out[2]: Poly(x^3 + 6x^2 + 6x + 4, GF(7))

In [3]: galois.primitive_poly(7, 3, method="random")
Out[3]: Poly(x^3 + 2x^2 + 6x + 2, GF(7))
```

Notice `primitive_poly()` returns the lexicographically-minimal primitive polynomial but `conway_poly()` returns the lexicographically-minimal primitive polynomial that is consistent with smaller Conway polynomials. This is sometimes the same polynomial.

```
In [4]: galois.primitive_poly(2, 4)
Out[4]: Poly(x^4 + x + 1, GF(2))

In [5]: galois.conway_poly(2, 4)
Out[5]: Poly(x^4 + x + 1, GF(2))
```

However, it is not always.

```
In [6]: galois.primitive_poly(7, 10)
Out[6]: Poly(x^10 + 5x^2 + x + 5, GF(7))

In [7]: galois.conway_poly(7, 10)
Out[7]: Poly(x^10 + x^6 + x^5 + 4x^4 + x^3 + 2x^2 + 3x + 3, GF(7))
```

Monic primitive polynomials scaled by non-zero field elements (now non-monic) are also primitive.

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

In [9]: f = galois.primitive_poly(7, 5, method="random"); f
Out[9]: Poly(x^5 + 2x^4 + x^2 + 5x + 2, GF(7))

In [10]: f.isPrimitive()
Out[10]: True

In [11]: g = f * GF(3); g
Out[11]: Poly(3x^5 + 6x^4 + 3x^2 + x + 6, GF(7))

In [12]: g.isPrimitive()
Out[12]: True
```

galois.primitive_polys(order: int, degree: int, reverse: bool = False) → Iterator[Poly]

Iterates through all monic primitive polynomials \( f(x) \) over \( \text{GF}(q) \) with degree \( m \).

Parameters

- **order**: int
  
  The prime power order \( q \) of the field \( \text{GF}(q) \) that the polynomial is over.
**degree**: int
   The degree \( m \) of the desired primitive polynomial.

**reverse**: bool = False
   Indicates to return the primitive polynomials from lexicographically maximal to minimal.
   The default is **False**.

**Returns**
   An iterator over all degree-\( m \) monic primitive polynomials over \( \text{GF}(q) \).

**See also:**
Poly.is_primitive, irreducible_polys

**Notes**

If \( f(x) \) is a primitive polynomial over \( \text{GF}(q) \) and \( a \in \text{GF}(q) \setminus \{0\} \), then \( a \cdot f(x) \) is also primitive.

In addition to other applications, \( f(x) \) produces the field extension \( \text{GF}(q^m) \) of \( \text{GF}(q) \). Since \( f(x) \) is primitive, \( x \) is a primitive element \( \alpha \) of \( \text{GF}(q^m) \) such that \( \text{GF}(q^m) = \{0, 1, \alpha, \alpha^2, \ldots, \alpha^{q^m-2}\} \).

**Examples**

Find all monic primitive polynomials over \( \text{GF}(3) \) with degree 4. You may also use `tuple()` on the returned generator.

```
In [1]: list(galois.primitive_polys(3, 4))
Out[1]:
[Poly(x^4 + x + 2, GF(3)),
 Poly(x^4 + 2x + 2, GF(3)),
 Poly(x^4 + x^3 + 2, GF(3)),
 Poly(x^4 + x^3 + x^2 + 2x + 2, GF(3)),
 Poly(x^4 + x^3 + 2x^2 + 2x + 2, GF(3)),
 Poly(x^4 + 2x^3 + 2, GF(3)),
 Poly(x^4 + 2x^3 + x^2 + x + 2, GF(3)),
 Poly(x^4 + 2x^3 + 2x^2 + x + 2, GF(3))]`
```

Loop over all the polynomials in reversed order, only finding them as needed. The search cost for the polynomials that would have been found after the `break` condition is never incurred.

```
In [2]: for poly in galois.primitive_polys(3, 4, reverse=True):
   ...:     if poly.coeffs[1] < 2:  # Early exit condition
   ...:         break
   ...:     print(poly)
   ....
x^4 + 2x^3 + 2x^2 + x + 2
x^4 + 2x^3 + x^2 + x + 2
x^4 + 2x^3 + 2
```

Or, manually iterate over the generator.

```
In [3]: generator = galois.primitive_polys(3, 4, reverse=True); generator
Out[3]: <generator object primitive_polys at 0x7f6c18dba740>
In [4]: next(generator)
```

(continues on next page)
Interpolating polynomials

galois.\texttt{lagrange\_poly}(x: \text{Array}, y: \text{Array}) \to \text{Poly}

Computes the Lagrange interpolating polynomial \( L(x) \) such that \( L(x_i) = y_i \).

galois.\texttt{lagrange\_poly}(x: \text{Array}, y: \text{Array}) \to \text{Poly}

Computes the Lagrange interpolating polynomial \( L(x) \) such that \( L(x_i) = y_i \).

\textbf{Parameters}

\begin{itemize}
\item \textit{x: Array} \\
An array of \( x_i \) values for the coordinates \((x_i, y_i)\). Must be 1-D. Must have no duplicate entries.
\item \textit{y: Array} \\
An array of \( y_i \) values for the coordinates \((x_i, y_i)\). Must be 1-D. Must be the same size as \textit{x}.
\end{itemize}

\textbf{Returns}

The Lagrange polynomial \( L(x) \).

\textbf{Notes}

The Lagrange interpolating polynomial is defined as

\[
L(x) = \sum_{j=0}^{k-1} y_j \ell_j(x)
\]

\[
\ell_j(x) = \prod_{0 \leq m < k \atop m \neq j} \frac{x - x_m}{x_j - x_m}
\]

It is the polynomial of minimal degree that satisfies \( L(x_i) = y_i \).

\textbf{References}

Examples

Create random \((x, y)\) pairs in \(\text{GF}(3^2)\).

\[
\text{In [1]: } \text{GF} = \text{galois.GF}(3**2) \\
\text{In [2]: } x = \text{GF}.\text{elements}; x \\
\text{Out[2]: } \text{GF}([0, 1, 2, 3, 4, 5, 6, 7, 8], \text{order}=3^2) \\
\text{In [3]: } y = \text{GF}.\text{Random}(x.\text{size}); y \\
\text{Out[3]: } \text{GF}([8, 6, 6, 6, 5, 0, 3, 0, 3], \text{order}=3^2)
\]

Find the Lagrange polynomial that interpolates the coordinates.

\[
\text{In [4]: } L = \text{galois}.\text{lagrange_poly}(x, y); L \\
\text{Out[4]: } \text{Poly}(5x^8 + 5x^6 + 6x^5 + x^4 + 8x^3 + 5x^2 + 7x + 8, \text{GF}(3^2))
\]

Show that the polynomial evaluated at \(x\) is \(y\).

\[
\text{In [5]: } \text{np.}\text{array_equal}(L(x), y) \\
\text{Out[5]: } \text{True}
\]

3.18.4 Forward error correction

class galois.BCH
A primitive, narrow-sense binary BCH\((n, k)\) code.

class galois.ReedSolomon
A general RS\((n, k)\) code.

galois.bch_valid_codes\((n: \text{int}, ...) \rightarrow \text{List[ Tuple[\text{int, int}]\)}}
Returns a list of \((n, k, t)\) tuples of valid primitive binary BCH codes.

galois.generator_to_parity_check_matrix\((G: \text{FieldArray}) \rightarrow \text{FieldArray}\)
Converts the generator matrix \(G\) of a linear \([n, k]\) code into its parity-check matrix \(H\).

galois.parity_check_to_generator_matrix\((H: \text{FieldArray}) \rightarrow \text{FieldArray}\)
Converts the parity-check matrix \(H\) of a linear \([n, k]\) code into its generator matrix \(G\).

galois.poly_to_generator_matrix\((n: \text{int}, ...) \rightarrow \text{FieldArray}\)
Converts the generator polynomial \(g(x)\) into the generator matrix \(G\) for an \([n, k]\) cyclic code.

galois.roots_to_parity_check_matrix\((n: \text{int}, \text{roots}) \rightarrow \text{FieldArray}\)
Converts the generator polynomial roots into the parity-check matrix \(H\) for an \([n, k]\) cyclic code.

class galois.BCH
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 \texttt{encode()} and \(n - s\) bits into \texttt{decode()} . Shortened codes are only applicable for systematic codes.
Examples

Construct the BCH code.

```
In [1]: galois.bch_valid_codes(15)
Out[1]: [(15, 11, 1), (15, 7, 2), (15, 5, 3), (15, 1, 7)]

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

Encode a message.

```
In [3]: m = galois.GF2.Random(bch.k); m
Out[3]: GF([1, 1, 1, 0, 0, 0, 1], order=2)

In [4]: c = bch.encode(m); c
Out[4]: GF([1, 1, 1, 0, 0, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1], order=2)
```

Corrupt the codeword and decode the message.

```
# Corrupt the first bit in the codeword
In [5]: c[0] ^= 1

In [6]: dec_m = bch.decode(c); dec_m
Out[6]: GF([1, 1, 1, 0, 0, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1], order=2)

In [7]: np.array_equal(dec_m, m)
Out[7]: True
```

```
# Instruct the decoder to return the number of corrected bit errors
In [8]: dec_m, N = bch.decode(c, errors=True); dec_m, N
Out[8]: (GF([1, 1, 1, 0, 0, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1], order=2), 1)

In [9]: np.array_equal(dec_m, m)
Out[9]: True
```

Constructors

```
BCH(n: int, k: int, primitive_poly: PolyLike | None = None, ...)

Constructs a primitive, narrow-sense binary BCH\((n, k)\) code.
```

```
galois.BCH(n: int, k: int, primitive_poly: PolyLike | None = None, primitive_element: PolyLike | None = None, systematic: bool = True)
```

Constructs a primitive, narrow-sense binary BCH\((n, k)\) code.

Parameters

- **n**: int
  - The codeword size \(n\), must be \(n = 2^m - 1\).

- **k**: int
  - The message size \(k\).
primitive_poly: PolyLike | None = None
Optionally specify the primitive polynomial that defines the extension field GF(2^m). The default is None which uses Matlab’s default, see matlab_primitive_poly().

primitive_element: PolyLike | None = None
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 GF(2^m), see primitive_element().

systematic: bool = True
Optionally specify if the encoding should be systematic, meaning the codeword is the message with parity appended. The default is True.

See also:
bch_valid_codes, primitive_poly, primitive_element

String representation

__repr__() → str
A terse representation of the BCH code.

__str__() → str
A formatted string with relevant properties of the BCH code.
galois.BCH.__repr__() → str
A terse representation of the BCH code.

Examples

```
In [1]: bch = galois.BCH(15, 7)
In [2]: bch
Out[2]: <BCH Code: [15, 7, 5] over GF(2)>
```

galois.BCH.__str__() → str
A formatted string with relevant properties of the BCH code.

Examples

```
In [1]: bch = galois.BCH(15, 7)
In [2]: print(bch)
BCH Code:
   [n, k, d]: [15, 7, 5]
   field: GF(2)
   generator_poly: x^8 + x^7 + x^6 + x^4 + 1
   is_primitive: True
   is_narrow_sense: True
   is_systematic: True
   t: 2
```
Methods

\begin{verbatim}
decode(codeword: ndarray | GF2, ...) → ndarray | GF2

Decodes the BCH codeword \( c \) into the message \( m \).

detect(codeword: ndarray | GF2) → bool | ndarray

Detects if errors are present in the BCH codeword \( c \).

encode(message: ndarray | GF2, ...) → ndarray | GF2

Encodes the message \( m \) into the BCH codeword \( c \).

galois.BCH.decode(codeword: ndarray | GF2, errors: False = False) → ndarray | GF2

Decodes the BCH codeword \( c \) into the message \( m \).

galois.BCH.decode(codeword: ndarray | GF2, errors: True) → Tuple[ndarray | GF2, integer | ndarray]

Decodes the BCH codeword \( c \) into the message \( m \).
\end{verbatim}

Parameters

\begin{itemize}
\item \textbf{codeword: ndarray | GF2}
  
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.

\item \textbf{errors: False = False}

   Optionally specify whether to return the number of corrected errors. The default is \textbf{False}.
\end{itemize}

Returns

\begin{itemize}
\item The decoded message as either a \( k \)-length vector or \((N, k)\) matrix.
\item 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.
\end{itemize}

Notes

The codeword vector \( c \) is defined as \( c = [c_{n-1}, \ldots, c_1, c_0] \in GF(2)^n \), 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 \( m = [m_{k-1}, \ldots, m_1, m_0] \in GF(2)^k \), 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 \texttt{decode()} to return the \( k - s \)-bit message.
Examples

Vector

Encode a single message using the BCH(15, 7) code.

```
In [1]: bch = galois.BCH(15, 7)
In [2]: GF = galois.GF(2)
In [3]: m = GF.Random(bch.k); m
Out[3]: GF([0, 0, 1, 1, 0, 0, 0], order=2)
In [4]: c = bch.encode(m); c
Out[4]: GF([0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 1], order=2)
```

Corrupt \( t \) bits of the codeword.

```
In [5]: bch.t
Out[5]: 2
In [6]: c[0:bch.t] ^= 1; c
Out[6]: GF([1, 1, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 1], order=2)
```

Decode the codeword and recover the message.

```
In [7]: d = bch.decode(c); d
Out[7]: GF([0, 0, 1, 1, 0, 0, 0], order=2)
In [8]: np.array_equal(d, m)
Out[8]: True
```

Decode the codeword, specifying the number of corrected errors, and recover the message.

```
In [9]: d, e = bch.decode(c, errors=True); d, e
Out[9]: (GF([0, 0, 1, 1, 0, 0, 0], order=2), 2)
In [10]: np.array_equal(d, m)
Out[10]: True
```

Vector (shortened)

Encode a single message using the shortened BCH(12, 4) code.

```
In [11]: bch = galois.BCH(15, 7)
In [12]: GF = galois.GF(2)
In [13]: m = GF.Random(bch.k - 3); m
Out[13]: GF([0, 1, 1, 0], order=2)
In [14]: c = bch.encode(m); c
Out[14]: GF([0, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1], order=2)
```
Corrupt $t$ bits of the codeword.

```
In [15]: bch.t
Out[15]: 2

In [16]: c[0:bch.t] ^= 1; c
Out[16]: GF([1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1], order=2)
```

Decode the codeword and recover the message.

```
In [17]: d = bch.decode(c); d
Out[17]: GF([0, 1, 1, 0], order=2)

In [18]: np.array_equal(d, m)
Out[18]: True
```

Decode the codeword, specifying the number of corrected errors, and recover the message.

```
In [19]: d, e = bch.decode(c, errors=True); d, e
Out[19]: (GF([0, 1, 1, 0], order=2), 2)

In [20]: np.array_equal(d, m)
Out[20]: True
```

**Matrix**

Encode a matrix of three messages using the BCH$(15, 7)$ code.

```
In [21]: bch = galois.BCH(15, 7)
In [22]: GF = galois.GF(2)
In [23]: m = GF.Random((3, bch.k)); m
Out[23]: GF([[0, 1, 0, 1, 0, 0, 1],
        [0, 1, 1, 0, 1, 1, 0],
        [1, 1, 0, 1, 0, 0, 1]], order=2)
In [24]: c = bch.encode(m); c
Out[24]: GF([[0, 1, 0, 1, 0, 0, 1, 1, 0, 1, 1, 1, 0, 0, 0],
         [0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1],
         [1, 1, 0, 1, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0]], order=2)
```

Corrupt the codeword. Add zero errors to the first codeword, one to the second, and two to the third.

```
In [25]: c[1,0:1] ^= 1
In [26]: c[2,0:2] ^= 1
In [27]: c
Out[27]: GF([[0, 1, 0, 1, 0, 0, 1, 1, 0, 1, 1, 1, 0, 0, 0],
         [0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1],
         [1, 1, 0, 1, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0]], order=2)
```
Decode the codeword and recover the message.

```python
In [28]: d = bch.decode(c); d
Out[28]:
GF([[0, 1, 0, 1, 0, 0, 1],
    [0, 1, 1, 0, 1, 1, 0],
    [1, 1, 0, 1, 0, 0, 1]], order=2)

In [29]: np.array_equal(d, m)
Out[29]: True
```

Decode the codeword, specifying the number of corrected errors, and recover the message.

```python
In [30]: d, e = bch.decode(c, errors=True); d, e
Out[30]:
(GF([[0, 1, 0, 1, 0, 0, 1],
    [0, 1, 1, 0, 1, 1, 0],
    [1, 1, 0, 1, 0, 0, 1]], order=2),
array([0, 1, 2]))

In [31]: np.array_equal(d, m)
Out[31]: True
```

**Matrix (shortened)**

Encode a matrix of three messages using the shortened BCH(12, 4) code.

```python
In [32]: bch = galois.BCH(15, 7)
In [33]: GF = galois.GF(2)
In [34]: m = GF.Random((3, bch.k - 3)); m
Out[34]:
GF([[1, 0, 0, 0],
    [1, 0, 0, 0],
    [0, 1, 1, 1]], order=2)

In [35]: c = bch.encode(m); c
Out[35]:
GF([[1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1],
    [1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1],
    [0, 1, 1, 1, 0, 1, 0, 0, 0, 1, 0, 0]], order=2)
```

Corrupt the codeword. Add zero errors to the first codeword, one to the second, and two to the third.

```python
In [36]: c[1,0:1] ^= 1
In [37]: c[2,0:2] ^= 1
```
Decode the codeword and recover the message.

```
In [38]: c
Out[38]:
GF([[1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1],
    [0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1],
    [1, 0, 1, 1, 0, 1, 0, 0, 0, 1, 0, 0]], order=2)
```

Decide the codeword and recover the message.

```
In [39]: d = bch.decode(c); d
Out[39]:
GF([[1, 0, 0, 0],
    [1, 0, 0, 0],
    [0, 1, 1, 1]], order=2)
```

```
In [40]: np.array_equal(d, m)
Out[40]: True
```

Decode the codeword, specifying the number of corrected errors, and recover the message.

```
In [41]: d, e = bch.decode(c, errors=True); d, e
Out[41]:
(GF([[1, 0, 0, 0],
    [1, 0, 0, 0],
    [0, 1, 1, 1]], order=2),
array([[0, 1, 2]]))
```

```
In [42]: np.array_equal(d, m)
Out[42]: True
```

\textbf{galois.BCH.detect} (codeword: ndarray | GF2) → bool | ndarray

Detects if errors are present in the BCH codeword \textit{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**: ndarray | GF2
  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

\textbf{True} or not \textbf{False}.
Examples

Vector

Encode a single message using the BCH(15, 7) code.

```python
In [1]: bch = galois.BCH(15, 7)
In [2]: GF = galois.GF(2)
In [3]: m = GF.Random(bch.k); m
Out[3]: GF([1, 1, 0, 1, 0, 1, 0], order=2)
In [4]: c = bch.encode(m); c
Out[4]: GF([1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 0, 1, 0], order=2)
```

Detect no errors in the valid codeword.

```python
In [5]: bch.detect(c)
Out[5]: False
```

Detect $d_{\text{min}} - 1$ errors in the codeword.

```python
In [6]: bch.d
Out[6]: 5
In [7]: c[0:bch.d - 1] ^= 1; c
Out[7]: GF([0, 0, 1, 0, 0, 1, 0, 1, 1, 1, 1, 0, 0, 1], order=2)
In [8]: bch.detect(c)
Out[8]: True
```

Vector (shortened)

Encode a single message using the shortened BCH(12, 4) code.

```python
In [9]: bch = galois.BCH(15, 7)
In [10]: GF = galois.GF(2)
In [11]: m = GF.Random(bch.k - 3); m
Out[11]: GF([1, 0, 0, 0], order=2)
In [12]: c = bch.encode(m); c
Out[12]: GF([1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1], order=2)
```

Detect no errors in the valid codeword.

```python
In [13]: bch.detect(c)
Out[13]: False
```

Detect $d_{\text{min}} - 1$ errors in the codeword.
Matrix

Encode a matrix of three messages using the BCH(15, 7) code.

In [17]: bch = galois.BCH(15, 7)
In [18]: GF = galois.GF(2)
In [19]: m = GF.Random((3, bch.k)); m
Out[19]: GF([[0, 1, 0, 0, 0, 1, 1],
            [0, 1, 0, 0, 1, 1, 0],
            [1, 0, 0, 1, 1, 0, 0]], order=2)
In [20]: c = bch.encode(m); c
Out[20]: GF([[0, 1, 0, 0, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 0],
            [0, 1, 0, 0, 1, 1, 0, 1, 1, 1, 0, 0, 0, 0, 1],
            [1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1]], order=2)

Detect no errors in the valid codewords.

In [21]: bch.detect(c)
Out[21]: array([False, False, False])

Detect one, two, and $d_{\text{min}} - 1$ errors in the codewords.

In [22]: bch.d
Out[22]: 5
In [23]: c[0,0:1] ^= 1
In [24]: c[1,0:2] ^= 1
In [25]: c[2, 0:bch.d - 1] ^= 1
In [26]: c
Out[26]: GF([[1, 1, 0, 0, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 0],
            [1, 0, 0, 0, 1, 1, 0, 1, 1, 1, 0, 0, 0, 0, 1],
            [0, 1, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1]], order=2)
In [27]: bch.detect(c)
Out[27]: array([ True,  True,  True])

Matrix (shortened)

Encode a matrix of three messages using the shortened BCH(12, 4) code.

In [28]: bch = galois.BCH(15, 7)
In [29]: GF = galois.GF(2)
In [30]: m = GF.Random((3, bch.k - 3)); m
Out[30]: GF([[1, 0, 0, 1],
              [0, 0, 1, 0],
              [0, 1, 0, 0]], order=2)
In [31]: c = bch.encode(m); c
Out[31]: GF([[1, 0, 0, 1, 1, 1, 0, 0, 1, 1, 0, 0],
              [0, 0, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1],
              [0, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1, 0]], order=2)
Detect no errors in the valid codewords.

In [32]: bch.detect(c)
Out[32]: array([False, False, False])
Detect one, two, and \(d_{\text{min}} - 1\) errors in the codewords.

In [33]: bch.d
Out[33]: 5
In [34]: c[0,0:1] ^= 1
In [35]: c[1,0:2] ^= 1
In [36]: c[2, 0:bch.d - 1] ^= 1
In [37]: c
Out[37]: GF([[0, 0, 0, 1, 1, 1, 0, 0, 1, 1, 0, 0],
              [1, 1, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1],
              [1, 0, 1, 1, 1, 1, 1, 0, 0, 1, 1, 0]], order=2)
In [38]: bch.detect(c)
Out[38]: array([ True,  True,  True])

galois.BCH.encode(message: ndarray | GF2, parity_only: bool = False) → ndarray | GF2

Encodes the message \(m\) into the BCH codeword \(c\).

Parameters
message: ndarray | GF2

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 = False

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.

Notes

The message vector $m$ is defined as $m = [m_{k-1}, \ldots, m_1, m_0] \in \GF(2)^k$, 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 $c = [c_{n-1}, \ldots, c_1, c_0] \in \GF(2)^n$, 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, $G = [I | P]$ such that $c = [m | p]$. 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. $g(x) | c(x)$.

For the shortened BCH$(n - s, k - s)$ code (only applicable for systematic codes), pass $k - s$ bits into `encode()` to return the $n - s$-bit codeword.

Examples

Vector

Encode a single message using the BCH$(15, 7)$ code.

```
In [1]: bch = galois.BCH(15, 7)
In [2]: GF = galois.GF(2)
In [3]: m = GF.Random(bch.k); m
Out[3]: GF([0, 0, 1, 0, 1, 0, 0], order=2)
In [4]: c = bch.encode(m); c
Out[4]: GF([0, 0, 1, 0, 1, 0, 0, 1, 1, 0, 1, 1, 1, 0, 0], order=2)
```

Compute the parity bits only.

```
In [5]: p = bch.encode(m, parity_only=True); p
Out[5]: GF([1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 0], order=2)
```
Vector (shortened)

Encode a single message using the shortened BCH(12, 4) code.

```
In [6]: bch = galois.BCH(15, 7)
In [7]: GF = galois.GF(2)
In [8]: m = GF.Random(bch.k - 3); m
Out[8]: GF([0, 0, 0, 1], order=2)
In [9]: c = bch.encode(m); c
Out[9]: GF([0, 0, 0, 1, 1, 1, 0, 1, 0, 0, 0, 1], order=2)

Compute the parity bits only.
```

```
In [10]: p = bch.encode(m, parity_only=True); p
Out[10]: GF([1, 1, 0, 1, 0, 0, 0, 1], order=2)
```

Matrix

Encode a matrix of three messages using the BCH(15, 7) code.

```
In [11]: bch = galois.BCH(15, 7)
In [12]: GF = galois.GF(2)
In [13]: m = GF.Random((3, bch.k)); m
Out[13]: GF([[0, 0, 1, 1, 1, 1, 1],
      [1, 1, 0, 1, 0, 1, 0],
      [1, 0, 0, 1, 1, 1, 1]], order=2)
In [14]: c = bch.encode(m); c
Out[14]: GF([[0, 0, 1, 1, 1, 1, 1, 0, 1, 1, 0, 0, 0, 1, 1],
      [1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 0, 0, 1, 0],
      [1, 0, 0, 1, 1, 1, 1, 1, 0, 1, 1, 0, 0, 0, 1]], order=2)

Compute the parity bits only.
```

```
In [15]: p = bch.encode(m, parity_only=True); p
Out[15]: GF([[0, 1, 1, 0, 0, 0, 1, 1],
      [1, 1, 1, 1, 0, 0, 1, 0],
      [1, 0, 1, 1, 0, 0, 0, 1]], order=2)
```
Encode a matrix of three messages using the shortened BCH(12, 4) code.

```
In [16]: bch = galois.BCH(15, 7)
In [17]: GF = galois.GF(2)
In [18]: m = GF.Random((3, bch.k - 3)); m
Out[18]:
GF([[1, 0, 0, 1],
    [1, 1, 1, 0],
    [1, 0, 0, 0]], order=2)
In [19]: c = bch.encode(m); c
Out[19]:
GF([[1, 0, 0, 1, 1, 1, 0, 0, 1, 1, 0, 0],
    [1, 1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0],
    [1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1]], order=2)
```

Compute the parity bits only.

```
In [20]: p = bch.encode(m, parity_only=True); p
Out[20]:
GF([[1, 1, 0, 0, 1, 1, 0, 0],
    [1, 0, 0, 0, 1, 0, 0, 0],
    [0, 0, 0, 1, 1, 1, 0, 1]], order=2)
```

**Properties**

- **property d : int**
  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, $d_{\text{min}} \geq d$.

- **property field : Type[FieldArray]**
  The FieldArray subclass for the GF($2^m$) field that defines the BCH code.

- **property G : GF2**
  The generator matrix $G$ with shape $(k, n)$.

- **property generator_poly : Poly**
  The generator polynomial $g(x)$ whose roots are roots.

- **property H : FieldArray**
  The parity-check matrix $H$ with shape $(2t, n)$.

- **property 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. $\alpha, \alpha^2, \ldots, \alpha^{2t}$.

- **property is_primitive : bool**
  Indicates if the BCH code is primitive, meaning $n = 2^m - 1$. 
property `is_systematic` : bool
Indicates if the code is configured to return codewords in systematic form.

property `k` : int
The message size \( k \) of the \([n, k, d]\) code

property `n` : int
The codeword size \( n \) of the \([n, k, d]\) code

property `roots` : FieldArray
The \(2t\) roots of the generator polynomial. These are consecutive powers of \(\alpha\), specifically \(\alpha, \alpha^2, \ldots, \alpha^{2t}\).

property `t` : int
The error-correcting capability of the code. The code can correct \( t \) bit errors in a codeword.

property `galois.BCH.d` : int
The design distance \( d \) of the \([n, k, d]\) 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
```

property `galois.BCH.field` : Type[FieldArray]
The FieldArray subclass for the \(GF(2^m)\) field that defines the BCH code.

Examples

```
In [1]: bch = galois.BCH(15, 7); bch
Out[1]: <BCH Code: [15, 7, 5] over GF(2)>

In [2]: bch.field
Out[2]: galois.GF(2^4)

In [3]: print(bch.field)
<class 'galois.GF(2^4)'>
```

property `galois.BCH.G` : GF2
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, 0, 1, 0, 0, 0],
     [0, 1, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0],
     [0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1],
     [0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 1, 1],
     [0, 0, 0, 0, 1, 0, 0, 1, 1, 0, 1, 0, 1, 1],
     [0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1],
     [0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 0, 0, 1]], order=2)

property galois.BCH.generator_poly: Poly

The generator polynomial \( g(x) \) whose roots are \( \text{roots} \).

Examples

In [1]: bch = galois.BCH(15, 7); bch
Out[1]: <BCH Code: [15, 7, 5] over GF(2)>

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

# Evaluate the generator polynomial at its roots in GF(2^m)
In [3]: bch.generator_poly(bch.roots, field=bch.field)
Out[3]: GF([0, 0, 0, 0], order=2^4)

property galois.BCH.H: FieldArray

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,  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)

property galois.BCH.is_narrow_sense: bool

Indicates if the BCH code is narrow sense, meaning the roots of the generator polynomial are consecutive powers of \( \alpha \) starting at 1, i.e. \( \alpha, \alpha^2, \ldots, \alpha^{2t} \).
Examples

```
In [1]: bch = galois.BCH(15, 7); bch
Out[1]: <BCH Code: [15, 7, 5] over GF(2)>

In [2]: bch.is_narrow_sense
Out[2]: True

In [3]: bch.roots
Out[3]: GF([2, 4, 8, 3], order=2^4)

In [4]: bch.field.primitive_element**(np.arange(1, 2*bch.t + 1))
Out[4]: GF([2, 4, 8, 3], order=2^4)
```

**property** galois.BCH.is_primitive: bool

Indicates if the BCH code is primitive, meaning $n = 2^m - 1$.

Examples

```
In [1]: bch = galois.BCH(15, 7); bch
Out[1]: <BCH Code: [15, 7, 5] over GF(2)>

In [2]: bch.is_primitive
Out[2]: True
```

**property** galois.BCH.is_systematic: bool

Indicates if the code is configured to return codewords in systematic form.

Examples

```
In [1]: bch = galois.BCH(15, 7); bch
Out[1]: <BCH Code: [15, 7, 5] over GF(2)>

In [2]: bch.is_systematic
Out[2]: True
```

**property** galois.BCH.k: int

The message size $k$ of the $[n, k, d]_2$ code

Examples

```
In [1]: bch = galois.BCH(15, 7); bch
Out[1]: <BCH Code: [15, 7, 5] over GF(2)>

In [2]: bch.k
Out[2]: 7
```

**property** galois.BCH.n: int

The codeword size $n$ 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.n
Out[2]: 15
```

**property** galois.BCH.roots : `FieldArray`

The $2t$ roots of the generator polynomial. These are consecutive powers of $\alpha$, specifically $\alpha, \alpha^2, \ldots, \alpha^{2t}$.

Examples

```python
In [1]: bch = galois.BCH(15, 7); bch
Out[1]: <BCH Code: [15, 7, 5] over GF(2)>

In [2]: bch.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)
```

**property** galois.BCH.t : `int`

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
```

class galois.ReedSolomon

A general RS$(n, k)$ code.

A RS$(n, k)$ code is a $[n, k, d]_q$ linear block code with codeword size $n$, message size $k$, minimum distance $d$, and symbols taken from an alphabet of size $q$ (a prime power).

To create the shortened RS$(n-s, k-s)$ code, construct the full-sized RS$(n, k)$ code and then pass $k-s$ symbols into `encode()` and $n-s$ symbols into `decode()`. Shortened codes are only applicable for systematic codes.
Examples

Construct the Reed-Solomon code.

```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([ 1, 14, 6, 0, 7, 5, 5, 6, 9], order=2^4)
In [4]: c = rs.encode(m); c
Out[4]: GF([ 1, 14, 6, 0, 7, 5, 5, 6, 9, 15, 15, 0, 3, 4, 12], 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([ 1, 14, 6, 0, 7, 5, 5, 6, 9], order=2^4)

In [7]: np.array_equal(dec_m, m)
Out[7]: True
```

```python
# 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([ 1, 14, 6, 0, 7, 5, 5, 6, 9], order=2^4), 1)

In [9]: np.array_equal(dec_m, m)
Out[9]: True
```

Constructors

**ReedSolomon**

```python
ReedSolomon(n: int, k: int, c: int = 1, ...)
```

Constructs a general RS\(n, k\) code.

```python
galois.ReedSolomon(n: int, k: int, c: int = 1, primitive_poly: PolyLike | None = None, primitive_element: PolyLike | None = None, systematic: bool = 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 = 1
  - The first consecutive power of \(\alpha\). The default is 1.
**primitive_poly:** *PolyLike | None = None*
Optionally specify the primitive polynomial that defines the extension field \( \text{GF}(q) \). The default is `None` which uses Matlab’s default, see `matlab_primitive_poly()`.

**primitive_element:** *PolyLike | None = None*
Optionally specify the primitive 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 `primitive_element()`.

**systematic:** *bool = True*
Optionally specify if the encoding should be systematic, meaning the codeword is the message with parity appended. The default is `True`.

See also:

`primitive_poly, primitive_element`

---

**String representation**

`__repr__()` → `str`
A terse representation of the Reed-Solomon code.

`__str__()` → `str`
A formatted string with relevant properties of the Reed-Solomon code.

```python
galois.ReedSolomon.__repr__() → str
A terse representation of the Reed-Solomon code.

Examples

```
In [1]: rs = galois.ReedSolomon(15, 9)
In [2]: rs
Out[2]: <Reed-Solomon Code: [15, 9, 7] over GF(2^4)>
```

```python
galois.ReedSolomon.__str__() → str
A formatted string with relevant properties of the Reed-Solomon code.

Examples

```
In [1]: rs = galois.ReedSolomon(15, 9)
In [2]: print(rs)
Reed-Solomon Code:
  [n, k, d]: [15, 9, 7]
  field: GF(2^4)
  generator_poly: x^6 + 7x^5 + 9x^4 + 3x^3 + 12x^2 + 10x + 12
  is_narrow_sense: True
  is_systematic: True
  t: 3
```

---
Methods

\texttt{decode(codeword: \texttt{ndarray} | \texttt{FieldArray}, ...) \rightarrow \texttt{ndarray} | \texttt{FieldArray}}

Decodes the Reed-Solomon codeword \( c \) into the message \( m \).

\texttt{detect(codeword: \texttt{ndarray} | \texttt{FieldArray}) \rightarrow \texttt{bool} | \texttt{ndarray}}

Detects if errors are present in the Reed-Solomon codeword \( c \).

\texttt{encode(message: \texttt{ndarray} | \texttt{FieldArray}, ...) \rightarrow \texttt{ndarray} | \texttt{FieldArray}}

Encodes the message \( m \) into the Reed-Solomon codeword \( c \).

\texttt{galois.ReedSolomon.decode(codeword: \texttt{ndarray} | \texttt{FieldArray}, errors: \texttt{False} = \texttt{False}) \rightarrow \texttt{ndarray} | \texttt{FieldArray}}

Decodes the Reed-Solomon codeword \( c \) into the message \( m \).

\texttt{galois.ReedSolomon.decode(codeword: \texttt{ndarray} | \texttt{FieldArray}, errors: \texttt{True}) \rightarrow Tuple[\texttt{ndarray} | \texttt{FieldArray}, integer | \texttt{ndarray}]} 

Decodes the Reed-Solomon codeword \( c \) into the message \( m \).

Parameters

- \textbf{codeword: \texttt{ndarray} | \texttt{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.

- \textbf{errors: \texttt{False} = \texttt{False}}
  
- \textbf{errors: \texttt{True}}
  
  Optionally specify whether to return the number of corrected errors. The default is \texttt{False}.

Returns

- The decoded message as either a \( k \)-length vector or \((N, k)\) matrix.

- 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 the codeword polynomial evaluated at each root of the generator polynomial, i.e. \( s = [c(\alpha^s), c(\alpha^{s+1}), \ldots, c(\alpha^{s+2t-1})] \). 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 \texttt{decode()} to return the \( k - s \)-symbol message.
Examples

Vector

Encode a single message using the RS(15,9) code.

```
In [1]: rs = galois.ReedSolomon(15, 9)
In [2]: GF = rs.field
In [3]: m = GF.Random(rs.k); m
Out[3]: GF([11, 12, 4, 0, 14, 7, 10, 8, 5], order=2^4)

In [4]: c = rs.encode(m); c
Out[4]: GF([11, 12, 4, 0, 14, 7, 10, 8, 5, 7, 7, 11, 4, 1, 13], order=2^4)
```

Corrupt \( t \) symbols of the codeword.

```
In [5]: e = GF.Random(rs.t, low=1); e
Out[5]: GF([13, 3, 2], order=2^4)

In [6]: c[0:rs.t] += e; c
Out[6]: GF([6, 15, 6, 0, 14, 7, 10, 8, 5, 7, 7, 11, 4, 1, 13], order=2^4)
```

Decode the codeword and recover the message.

```
In [7]: d = rs.decode(c); d
Out[7]: GF([11, 12, 4, 0, 14, 7, 10, 8, 5], order=2^4)
In [8]: np.array_equal(d, m)
Out[8]: True
```

Decode the codeword, specifying the number of corrected errors, and recover the message.

```
In [9]: d, e = rs.decode(c, errors=True); d, e
Out[9]: (GF([11, 12, 4, 0, 14, 7, 10, 8, 5], order=2^4), 3)
In [10]: np.array_equal(d, m)
Out[10]: True
```

Vector (shortened)

Encode a single message using the shortened RS(11,5) code.

```
In [11]: rs = galois.ReedSolomon(15, 9)
In [12]: GF = rs.field
In [13]: m = GF.Random(rs.k - 4); m
Out[13]: GF([7, 13, 0, 4, 0], order=2^4)
```
Corrupt $t$ symbols of the codeword.

```python
In [15]: e = GF.Random(rs.t, low=1); e
Out[15]: GF([ 7, 12, 1], order=2^4)

In [16]: c[0:rs.t] += e; c
Out[16]: GF([ 0, 1, 1, 4, 0, 12, 7, 9, 11, 5, 4], order=2^4)
```

Decode the codeword and recover the message.

```python
In [17]: d = rs.decode(c); d
Out[17]: GF([ 7, 13, 0, 4, 0], order=2^4)

In [18]: np.array_equal(d, m)
Out[18]: True
```

Decode the codeword, specifying the number of corrected errors, and recover the message.

```python
In [19]: d, e = rs.decode(c, errors=True); d, e
Out[19]: (GF([ 7, 13, 0, 4, 0], order=2^4), 3)

In [20]: np.array_equal(d, m)
Out[20]: True
```

**Matrix**

Encode a matrix of three messages using the RS(15, 9) code.

```python
In [21]: rs = galois.ReedSolomon(15, 9)
In [22]: GF = rs.field
In [23]: m = GF.Random((3, rs.k)); m
Out[23]: GF([[ 8, 9, 8, 8, 10, 13, 2, 11, 11, 15, 13, 5, 14, 15, 11],
          [ 1, 5, 11, 8, 5, 10, 9, 5, 1, 5, 4, 14, 6, 6, 6],
          [ 8, 13, 7, 3, 1, 14, 13, 12, 2]], order=2^4)

In [24]: c = rs.encode(m); c
Out[24]: GF([[ 8, 9, 8, 8, 10, 13, 2, 11, 11, 15, 13, 5, 14, 15, 11],
          [ 1, 5, 11, 8, 5, 10, 9, 5, 1, 5, 4, 14, 6, 6, 6],
          [ 8, 13, 7, 3, 1, 14, 13, 12, 2, 6, 7, 0, 4, 4, 14]], order=2^4)
```

Corrupt the codeword. Add one error to the first codeword, two to the second, and three to the third.

```python
In [25]: c[0,0:1] += GF.Random(1, low=1)
```
In [26]: c[1,0:2] += GF.Random(2, low=1)

In [27]: c[2,0:3] += GF.Random(3, low=1)

In [28]: c
Out[28]:
GF([[ 1, 9, 8, 8, 10, 13, 2, 11, 11, 15, 13, 5, 14, 15, 11],
    [ 6, 4, 11, 8, 5, 10, 9, 5, 1, 5, 4, 14, 6, 6, 6],
    [ 5, 14, 8, 3, 1, 14, 13, 12, 2, 6, 7, 0, 4, 4, 14]],
   order=2^4)

Decode the codeword and recover the message.

In [29]: d = rs.decode(c); d
Out[29]:
GF([[ 8, 9, 8, 8, 10, 13, 2, 11, 11],
    [ 1, 5, 11, 8, 5, 10, 9, 5, 1],
    [ 8, 13, 7, 3, 1, 14, 13, 12, 2]], order=2^4)

In [30]: np.array_equal(d, m)
Out[30]: True

Decode the codeword, specifying the number of corrected errors, and recover the message.

In [31]: d, e = rs.decode(c, errors=True); d, e
Out[31]:
(GF([[ 8, 9, 8, 8, 10, 13, 2, 11, 11],
    [ 1, 5, 11, 8, 5, 10, 9, 5, 1],
    [ 8, 13, 7, 3, 1, 14, 13, 12, 2]], order=2^4),
 array([1, 2, 3]))

In [32]: np.array_equal(d, m)
Out[32]: True

Matrix (shortened)

Encode a matrix of three messages using the shortened RS(11,5) code.

In [33]: rs = galois.ReedSolomon(15, 9)

In [34]: GF = rs.field

In [35]: m = GF.Random((3, rs.k - 4)); m
Out[35]:
GF([[ 3, 3, 13, 2, 11],
    [ 5, 5, 11, 14, 2],
    [13, 8, 4, 9, 1]], order=2^4)

In [36]: c = rs.encode(m); c
Out[36]:
GF([[ 3, 3, 13, 2, 11, 10, 8, 1, 14, 5, 13],
    [ 9, 5, 11, 9, 1, 2, 1, 14, 10, 8],
    [ 3, 8, 4, 9, 1, 14, 5, 13]], order=2^4)
Corrupt the codeword. Add one error to the first codeword, two to the second, and three to the third.

In [37]: \(c[0,0:1] += \text{GF.Random}(1, \text{low}=1)\)

In [38]: \(c[1,0:2] += \text{GF.Random}(2, \text{low}=1)\)

In [39]: \(c[2,0:3] += \text{GF.Random}(3, \text{low}=1)\)

In [40]: \(c\)

Out[40]:
\[
\text{GF([[ 6, 3, 13, 2, 11, 10, 8, 1, 14, 5, 13],}
\text{[ 8, 13, 11, 14, 2, 3, 10, 6, 9, 10, 9],}
\text{[15, 11, 3, 9, 1, 11, 2, 5, 1, 15, 3]], order=2^4)}
\]

Decode the codeword and recover the message.

In [41]: \(d = \text{rs.decode}(c)\); d

Out[41]:
\[
\text{GF([[ 3, 3, 13, 2, 11],}
\text{[ 5, 5, 11, 14, 2],}
\text{[13, 8, 4, 9, 1]], order=2^4)}
\]

In [42]: \(\text{np.array_equal}(d, m)\)

Out[42]: True

Decode the codeword, specifying the number of corrected errors, and recover the message.

In [43]: \(d, e = \text{rs.decode}(c, \text{errors=True})\); d, e

Out[43]:
\[
(GF([[ 3, 3, 13, 2, 11],}
\text{[ 5, 5, 11, 14, 2],}
\text{[13, 8, 4, 9, 1]], order=2^4)),
\text{array([1, 2, 3])})
\]

In [44]: \(\text{np.array_equal}(d, m)\)

Out[44]: True

galois.ReedSolomon.detect

\[\text{detect}(\text{codeword: ndarray | FieldArray}) \rightarrow \text{bool | ndarray}\]

Detects if errors are present in the Reed-Solomon codeword \(c\).

The \([n, k, d]_q\) Reed-Solomon code has \(d_{\text{min}} = d\) minimum distance. It can detect up to \(d_{\text{min}} - 1\) errors.

Parameters

- **codeword**: ndarray | 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**.
## Examples

### Vector

Encode a single message using the RS(15,9) code.

```
In [1]: rs = galois.ReedSolomon(15, 9)
In [2]: GF = rs.field
In [3]: m = GF.Random(rs.k); m
Out[3]: GF([ 5, 8, 12, 10, 11, 3, 12, 3, 11], order=2^4)
In [4]: c = rs.encode(m); c
Out[4]: GF([ 5, 8, 12, 10, 11, 3, 12, 3, 11, 9, 8, 15, 13, 1, 14], order=2^4)
```

Detect no errors in the valid codeword.

```
In [5]: rs.detect(c)
Out[5]: False
```

Detect $d_{\min} - 1$ errors in the codeword.

```
In [6]: rs.d
Out[6]: 7
In [7]: e = GF.Random(rs.d - 1, low=1); e
Out[7]: GF([ 7, 3, 12, 11, 8, 1], order=2^4)
In [8]: c[0:rs.d - 1] += e; c
Out[8]: GF([ 2, 11, 0, 1, 3, 2, 12, 3, 11, 9, 8, 15, 13, 1, 14], order=2^4)
In [9]: rs.detect(c)
Out[9]: True
```

### Vector (shortened)

Encode a single message using the shortened RS(11,5) code.

```
In [10]: rs = galois.ReedSolomon(15, 9)
In [11]: GF = rs.field
In [12]: m = GF.Random(rs.k - 4); m
Out[12]: GF([10, 2, 8, 12, 2], order=2^4)
In [13]: c = rs.encode(m); c
Out[13]: GF([10, 2, 8, 12, 2, 2, 13, 10, 15, 7, 12], order=2^4)
```

Detect no errors in the valid codeword.
Detect $d_{\text{min}} - 1$ errors in the codeword.

```
In [15]: rs.d
Out[15]: 7
```

```
In [16]: e = GF.Random(rs.d - 1, low=1); e
Out[16]: GF([12, 14, 5, 11, 8, 7], order=2^4)
```

```
In [17]: c[0:rs.d - 1] += e; c
Out[17]: GF([ 6, 12, 13, 7, 10, 5, 13, 10, 15, 7, 12], order=2^4)
```

```
In [18]: rs.detect(c)
Out[18]: True
```

**Matrix**

Encode a matrix of three messages using the RS(15, 9) code.

```
In [19]: rs = galois.ReedSolomon(15, 9)
```

```
In [20]: GF = rs.field

In [21]: m = GF.Random((3, rs.k)); m
Out[21]: GF([[10, 11, 4, 5, 14, 4, 10, 11, 6],
         [ 9, 13, 14, 15, 11, 7, 14, 7, 1],
         [ 1, 1, 9, 13, 1, 0, 3, 6, 7]], order=2^4)
```

```
In [22]: c = rs.encode(m); c
Out[22]: GF([[10, 11, 4, 5, 14, 4, 10, 11, 6, 13, 7, 13, 2, 2, 7],
         [ 9, 13, 14, 15, 11, 7, 14, 7, 1, 13, 13, 5, 0, 8, 14],
         [ 1, 1, 9, 13, 1, 0, 3, 6, 7, 10, 10, 0, 8, 9, 6]], order=2^4)
```

Detect no errors in the valid codewords.

```
In [23]: rs.detect(c)
Out[23]: array([False, False, False])
```

Detect one, two, and $d_{\text{min}} - 1$ errors in the codewords.

```
In [24]: rs.d
Out[24]: 7
```

```
In [25]: c[0,0:1] += GF.Random(1, low=1)
```

```
In [26]: c[1,0:2] += GF.Random(2, low=1)
```
Matrix (shortened)

Encode a matrix of three messages using the shortened RS(11, 5) code.

In [30]: rs = galois.ReedSolomon(15, 9)

In [31]: GF = rs.field

In [32]: m = GF.Random((3, rs.k - 4)); m
Out[32]: GF([[ 1, 4, 2, 11, 8],
          [ 6, 3, 10, 10, 9],
          [ 8, 10, 12, 8, 1]], order=2^4)

In [33]: c = rs.encode(m); c
Out[33]: GF([[ 1, 4, 2, 11, 8, 14, 13, 3, 2, 5, 15],
          [ 6, 3, 10, 10, 9, 4, 13, 14, 2, 9, 3],
          [ 8, 10, 12, 8, 1, 2, 13, 9, 3, 2, 3]], order=2^4)

Detect no errors in the valid codewords.

In [34]: rs.detect(c)
Out[34]: array([False, False, False])

Detect one, two, and $d_{\text{min}} - 1$ errors in the codewords.

In [35]: rs.d
Out[35]: 7

In [36]: c[0, 0:1] += GF.Random(1, low=1)

In [37]: c[1, 0:2] += GF.Random(2, low=1)

In [38]: c[2, 0:rs.d - 1] += GF.Random(rs.d - 1, low=1)

In [39]: c
Out[39]: GF([[ 5, 4, 2, 11, 8, 14, 13, 3, 2, 5, 15],
          [ 6, 3, 10, 10, 9, 4, 13, 14, 2, 9, 3],
          [ 8, 10, 12, 8, 1, 2, 13, 9, 3, 2, 3]], order=2^4)
[0, 9, 10, 10, 9, 4, 13, 14, 2, 9, 3],
[12, 0, 11, 6, 3, 13, 13, 9, 3, 2, 3]], order=2^4)

In [40]: rs.detect(c)
Out[40]: array([ True, True, True])

galois.ReedSolomon.encode(message: ndarray | FieldArray, parity_only: bool = False) → ndarray | FieldArray

Encodes the message \(m\) into the Reed-Solomon codeword \(c\).

Parameters

- **message**: ndarray | 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 = False
  - 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.

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 \(m(x) = m_{k-1}x^{k-1} + \cdots + m_1 x + m_0\). 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 \(c(x) = c_{n-1}x^{n-1} + \cdots + c_1 x + c_0\).

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 encode() to return the \(n-s\)-symbol codeword.

Examples

Vector

Encode a single message using the RS\((15, 9)\) code.

In [1]: rs = galois.ReedSolomon(15, 9)
In [2]: GF = rs.field
In [3]: m = GF.Random(rs.k); m

(continues on next page)
Out[3]: GF([11, 10, 12, 7, 1, 11, 7, 11, 10], order=2^4)

In [4]: c = rs.encode(m); c
Out[4]: GF([11, 10, 12, 7, 1, 11, 7, 11, 10, 6, 9, 14, 8, 10, 1], order=2^4)

Compute the parity symbols only.

In [5]: p = rs.encode(m, parity_only=True); p
Out[5]: GF([6, 9, 14, 8, 10, 1], order=2^4)

Vector (shortened)

Encode a single message using the shortened RS(11, 5) code.

In [6]: rs = galois.ReedSolomon(15, 9)

In [7]: GF = rs.field

In [8]: m = GF.Random(rs.k - 4); m
Out[8]: GF([3, 2, 13, 3, 5], order=2^4)

In [9]: c = rs.encode(m); c
Out[9]: GF([3, 2, 13, 3, 5, 12, 13, 4, 7, 3, 0], order=2^4)

Compute the parity symbols only.

In [10]: p = rs.encode(m, parity_only=True); p
Out[10]: GF([12, 13, 4, 7, 3, 0], order=2^4)

Matrix

Encode a matrix of three messages using the RS(15, 9) code.

In [11]: rs = galois.ReedSolomon(15, 9)

In [12]: GF = rs.field

In [13]: m = GF.Random((3, rs.k)); m
Out[13]: GF([[0, 10, 14, 11, 14, 1, 6, 15, 14],
          [9, 6, 2, 15, 1, 13, 9, 7, 0],
          [15, 2, 9, 2, 12, 4, 9, 14]], order=2^4)

In [14]: c = rs.encode(m); c
Out[14]: GF([[0, 10, 14, 11, 14, 1, 6, 15, 14, 4, 12, 6, 6, 12, 3],
          [9, 6, 2, 15, 1, 13, 9, 7, 0, 3, 12, 2, 4, 14, 12],
          [15, 2, 9, 2, 12, 4, 9, 14, 14, 2, 9, 8, 8, 3]], order=2^4)
Compute the parity symbols only.

```
In [15]: p = rs.encode(m, parity_only=True); p
Out[15]:
GF([[ 4, 12,  6,  6, 12,  3],
    [ 3, 12,  2,  4, 14, 12],
    [14,  2,  9,  8,  8,  3]], order=2^4)
```

Matrix (shortened)

Encode a matrix of three messages using the shortened RS(11, 5) code.

```
In [16]: rs = galois.ReedSolomon(15, 9)
In [17]: GF = rs.field
In [18]: m = GF.Random((3, rs.k - 4)); m
Out[18]:
GF([[ 8, 9, 10,  8,  5],
    [ 0, 3,  1,  1,  8],
    [10,  6, 10,  3, 11]], order=2^4)
In [19]: c = rs.encode(m); c
Out[19]:
GF([[ 8, 9, 10,  8, 12, 13],
    [ 0, 3,  1,  1,  8,  7],
    [10,  6, 10,  3, 14, 14]], order=2^4)
```

Compute the parity symbols only.

```
In [20]: p = rs.encode(m, parity_only=True); p
Out[20]:
GF([[14, 14, 15,  8, 12, 13],
    [15,  8,  7, 10, 12, 14],
    [ 1,  1,  7, 14, 14]], order=2^4)
```

Properties

- **property c**: int
  The degree of the first consecutive root.

- **property d**: int
  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, $d_{\text{min}} = d$.

- **property field**: Type[FieldArray]
  The FieldArray subclass for the GF$(q)$ field that defines the Reed-Solomon code.

- **property G**: FieldArray
  The generator matrix $G$ with shape $(k, n)$. 

3.18. galois
property generator_poly : Poly
        The generator polynomial \( g(x) \) whose roots are roots.

property H : FieldArray
        The parity-check matrix \( H \) with shape \((2t, n)\).

property is_narrow_sense : bool
        Indicates if the Reed-Solomon code is narrow sense, meaning the roots of the generator polynomial are
        consecutive powers of \( \alpha \) starting at 1, i.e. \( \alpha, \alpha^2, \ldots, \alpha^{2t-1} \).

property is_systematic : bool
        Indicates if the code is configured to return codewords in systematic form.

property k : int
        The message size \( k \) of the \([n, k, d]_q\) code.

property n : int
        The codeword size \( n \) of the \([n, k, d]_q\) code.

property roots : FieldArray
        The \( 2t \) roots of the generator polynomial. These are consecutive powers of \( \alpha \), specifically
        \( \alpha^c, \alpha^{c+1}, \ldots, \alpha^{c+2t-1} \).

property t : int
        The error-correcting capability of the code. The code can correct \( t \) symbol errors in a codeword.

property galois.ReedSolomon.c : int
        The degree of the first consecutive root.

Examples

In [1]: rs = galois.ReedSolomon(15, 9); rs
Out[1]: <Reed-Solomon Code: [15, 9, 7] over GF(2^4)>

In [2]: rs.c
Out[2]: 1

property galois.ReedSolomon.d : int
        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, \( d_{\text{min}} = d \).

Examples

In [1]: rs = galois.ReedSolomon(15, 9); rs
Out[1]: <Reed-Solomon Code: [15, 9, 7] over GF(2^4)>

In [2]: rs.d
Out[2]: 7

property galois.ReedSolomon.field : Type[FieldArray]
        The \( FieldArray \) subclass for the GF\( (q) \) field that defines the Reed-Solomon code.
Examples

```
In [1]: rs = galois.ReedSolomon(15, 9); rs
Out[1]: <Reed-Solomon Code: [15, 9, 7] over GF(2^4)>

In [2]: rs.field
Out[2]: galois.GF(2^4)

In [3]: print(rs.field)
<class 'galois.GF(2^4)'>
```

**property** `galois.ReedSolomon.G`: `FieldArray`

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, 2, 11, 10, 6, 15, 9],
    [0, 0, 0, 0, 0, 1, 0, 0, 0, 15, 9, 5, 8, 15, 2],
    [0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 7, 9, 3, 12, 10, 12]],
   order=2^4)
```

**property** `galois.ReedSolomon.generator_poly`: `Poly`

The generator polynomial $g(x)$ whose roots are `roots`.

Examples

```
In [1]: rs = galois.ReedSolomon(15, 9); rs
Out[1]: <Reed-Solomon Code: [15, 9, 7] over GF(2^4)>

In [2]: rs.generator_poly
Out[2]: Poly(x^6 + 7x^5 + 9x^4 + 3x^3 + 12x^2 + 10x + 12, GF(2^4))
```

Evaluate the generator polynomial at its roots.

```
In [3]: rs.generator_poly(rs.roots)
Out[3]: GF([0, 0, 0, 0, 0, 0], order=2^4)
```

**property** `galois.ReedSolomon.H`: `FieldArray`

The parity-check matrix $H$ with shape $(2t, n)$.
Examples

In [1]: rs = galois.ReedSolomon(15, 9); rs
Out[1]: <Reed-Solomon Code: [15, 9, 7] over GF(2^4)>

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

property galois.ReedSolomon.is_narrow_sense: bool
Indicates if the Reed-Solomon code is narrow sense, meaning the roots of the generator polynomial are consecutive powers of $\alpha$ starting at 1, i.e. $\alpha, \alpha^2, \ldots, \alpha^{2t-1}$.

Examples

In [1]: rs = galois.ReedSolomon(15, 9); rs
Out[1]: <Reed-Solomon Code: [15, 9, 7] over GF(2^4)>

In [2]: rs.is_narrow_sense
Out[2]: True

In [3]: rs.roots
Out[3]: GF([ 2, 4, 8, 3, 6, 12], order=2^4)

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)

property galois.ReedSolomon.is_systematic: bool
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.is_systematic
Out[2]: True

property galois.ReedSolomon.k: int
The message size $k$ of the $[n,k,d]_q$ 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.k
Out[2]: 9
```

**property** `galois.ReedSolomon.n`: int

The codeword size \( n \) of the \([n, k, d]\) 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.n
Out[2]: 15
```

**property** `galois.ReedSolomon.roots`: *FieldArray*

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

```python
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.

```python
In [3]: rs.generator_poly(rs.roots)
Out[3]: GF([0, 0, 0, 0, 0, 0], order=2^4)
```

**property** `galois.ReedSolomon.t`: int

The error-correcting capability of the code. The code can correct \( t \) symbol errors in a codeword.

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.t
Out[2]: 3
```

[galois.bch_valid_codes](n: int, t_min: int = 1) → List[Tuple[int, int, int]]

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 = 1
  The minimum error-correcting capability. The default is 1.

Returns

A list of $(n, k, t)$ tuples of valid primitive BCH codes.

See also:

BCH

References

- [https://link.springer.com/content/pdf/bbm%3A978-1-4899-2174-1%2F1.pdf](https://link.springer.com/content/pdf/bbm%3A978-1-4899-2174-1%2F1.pdf)

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.generator_to_parity_check_matrix(G: FieldArray) → FieldArray

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**: FieldArray
  The $(k, n)$ generator matrix $G$ in systematic form $G = [I_k | P_{k,n-k}]$.

Returns

- The $(n - k, n)$ parity-check matrix $H = [-P_{k,n-k}^T | I_{n-k,n-k}]$.

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
```
galois

```python
Out[3]:
GF([[1, 1, 1, 0, 1, 0, 0],
    [0, 1, 1, 1, 0, 1, 0],
    [1, 1, 0, 1, 0, 0, 1]], order=2)

In [4]: G @ H.T
Out[4]:
GF([[0, 0, 0],
    [0, 0, 0],
    [0, 0, 0],
    [0, 0, 0]], order=2)
```

```text
galois.parity_check_to_generator_matrix(H: FieldArray) → FieldArray
Converts the parity-check matrix H of a linear \([n, k]\) code into its generator matrix G.

The generator and parity-check matrices satisfy the equations \(GH^T = 0\).

Parameters

- **H**: FieldArray

  The \((n - k, n)\) parity-check matrix G in systematic form \(H = [-P_{k,n-k}^T | I_{n-k,n-k}]\).

Returns

- The \((k, n)\) generator matrix \(G = [I_{k,k} | P_{k,n-k}]\).

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)

In [4]: G2 = galois.parity_check_to_generator_matrix(H); G2
Out[4]:
GF([[1, 0, 0, 0, 1, 0, 1],
    [0, 1, 0, 0, 1, 1, 1],
    [0, 0, 1, 0, 1, 1, 0],
    [0, 0, 0, 1, 0, 1, 1]], order=2)

In [5]: G2 @ H.T
Out[5]:
```
```
galois

GF([[0, 0, 0],
     [0, 0, 0],
     [0, 0, 0],
     [0, 0, 0]], order=2)

galois.poly_to_generator_matrix(n: int, generator_poly: Poly, systematic: bool = True) → FieldArray
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**: Poly
  - The generator polynomial \( g(x) \).
- **systematic**: bool = True
  - 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 \( c = mG \).

Examples

Compute the generator matrix for the Hamming(7, 4) code.

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

In [2]: galois.poly_to_generator_matrix(7, g, systematic=False)
Out[2]:
GF([[1, 0, 1, 1, 0, 0, 0],
     [0, 1, 0, 1, 1, 0, 0],
     [0, 0, 1, 0, 1, 1, 0],
     [0, 0, 0, 1, 0, 1, 1]], order=2)

In [3]: galois.poly_to_generator_matrix(7, g, systematic=True)
Out[3]:
GF([[1, 0, 0, 0, 1, 0, 1],
     [0, 1, 0, 1, 1, 1, 0],
     [0, 0, 1, 0, 1, 1, 0],
     [0, 0, 0, 1, 0, 1, 1]], order=2)

galois.roots_to_parity_check_matrix(n: int, roots: FieldArray) → FieldArray
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**: 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
\[ s = cH^T. \]

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],
            [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)
```

### 3.18.5 Linear sequences

**class** `galois.FLFSR`
A Fibonacci linear-feedback shift register (LFSR).

**class** `galois.GLFSR`
A Galois linear-feedback shift register (LFSR).

`galois.berlekamp_massey`(sequence: FieldArray, ...) → Poly
`galois.berlekamp_massey`(sequence: FieldArray, output) → FLFSR
`galois.berlekamp_massey`(sequence: FieldArray, output) → GLFSR
Finds the minimal polynomial \(c(x)\) that produces the linear recurrent sequence \(y\).

**class** `galois.FLFSR`
A Fibonacci linear-feedback shift register (LFSR).
Notes

A Fibonacci LFSR is defined by its feedback polynomial \( f(x) \).

\[
f(x) = -c_0x^n - c_1x^{n-1} - \cdots - c_{n-2}x^2 - c_{n-1}x + 1 = x^n c(x^{-1})
\]

The feedback polynomial is the reciprocal of the characteristic polynomial \( c(x) \) of the linear recurrent sequence \( y \) produced by the Fibonacci LFSR.

\[
c(x) = x^n - c_{n-1}x^{n-1} - c_{n-2}x^{n-2} - \cdots - c_1x - c_0
\]

\[
y_t = c_{n-1}y_{t-1} + c_{n-2}y_{t-2} + \cdots + c_1y_{t-n+2} + c_0y_{t-n+1}
\]

Listing 1: Fibonacci LFSR Configuration

| +--------------+<-------------+<-------------+<-------------+<-------------+<-------------+<-------------+<-------------+<-------------+ |
| | ^ | c_n-1 | ^ | c_n-2 | ^ | c_1 | ^ | c_0 | |
| | +--------+ | +--------+ | | +--------+ | +--------+ |
| +--------+ | +--------+ |
| y[t+n-1] | y[t+n-2] | y[t+1] |

The shift register taps \( T \) are defined left-to-right as \( T = [T_0, T_1, \ldots, T_{n-2}, T_{n-1}] \). The state vector \( S \) is also defined left-to-right as \( S = [S_0, S_1, \ldots, S_{n-2}, S_{n-1}] \).

In the Fibonacci configuration, the shift register taps are \( T = [c_{n-1}, c_{n-2}, \ldots, c_1, c_0] \). Additionally, the state vector is equal to the next \( n \) outputs in reversed order, namely \( S = [y_{t+n-1}, y_{t+n-2}, \ldots, y_{t+2}, y_{t+1}] \).

References


See also:

berlekamp_massey

Examples

GF(2)

Create a Fibonacci LFSR from a degree-4 primitive characteristic polynomial over GF(2).

In [1]: c = galois.primitive_poly(2, 4); c
Out[1]: Poly(x^4 + x + 1, GF(2))

In [2]: lfsr = galois.FLFSR(c.reverse())

In [3]: print(lfsr)
Fibonacci LFSR:
   field: GF(2)
feedback_poly: \(x^4 + x^3 + 1\)
characteristic_poly: \(x^4 + x + 1\)
taps: [0, 0, 1, 1]
order: 4
state: [1, 1, 1, 1]
initial_state: [1, 1, 1, 1]

Step the Fibonacci LFSR and produce 10 output symbols.

```
In [4]: lfsr.state
Out[4]: GF([1, 1, 1, 1], order=2)

In [5]: lfsr.step(10)
Out[5]: GF([1, 1, 1, 1, 0, 0, 0, 1, 0, 0], order=2)

In [6]: lfsr.state
Out[6]: GF([1, 0, 1, 1], order=2)
```

GF(p)

Create a Fibonacci LFSR from a degree-4 primitive characteristic polynomial over GF(7).

```
In [7]: c = galois.primitive_poly(7, 4); c
Out[7]: Poly(x^4 + x^2 + 3x + 5, GF(7))

In [8]: lfsr = galois.FLFSR(c.reverse())

In [9]: print(lfsr)
Fibonacci LFSR:
  field: GF(7)
  feedback_poly: 5x^4 + 3x^3 + x^2 + 1
  characteristic_poly: x^4 + x^2 + 3x + 5
  taps: [0, 6, 4, 2]
  order: 4
  state: [1, 1, 1, 1]
  initial_state: [1, 1, 1, 1]

Step the Fibonacci LFSR and produce 10 output symbols.

```
In [10]: lfsr.state
Out[10]: GF([1, 1, 1, 1], order=7)

In [11]: lfsr.step(10)
Out[11]: GF([1, 1, 1, 1, 5, 5, 1, 3, 1, 4], order=7)

In [12]: lfsr.state
Out[12]: GF([5, 5, 6, 6], order=7)
```
Create a Fibonacci LFSR from a degree-4 primitive characteristic polynomial over $GF(2^3)$.

```
In [13]: c = galois.primitive_poly(2**3, 4); c
Out[13]: Poly(x^4 + x + 3, GF(2^3))

In [14]: lfsr = galois.FLFSR(c.reverse())

In [15]: print(lfsr)
Fibonacci LFSR:
   field: GF(2^3)
   feedback_poly: 3x^4 + x^3 + 1
   characteristic_poly: x^4 + x + 3
   taps: [0, 0, 1, 3]
   order: 4
   state: [1, 1, 1, 1]
   initial_state: [1, 1, 1, 1]
```

Step the Fibonacci LFSR and produce 10 output symbols.

```
In [16]: lfsr.state
Out[16]: GF([1, 1, 1, 1], order=2^3)

In [17]: lfsr.step(10)
Out[17]: GF([1, 1, 1, 1, 2, 2, 2, 1, 4, 4], order=2^3)

In [18]: lfsr.state
Out[18]: GF([0, 3, 7, 7], order=2^3)
```

Create a Fibonacci LFSR from a degree-4 primitive characteristic polynomial over $GF(3^3)$.

```
In [19]: c = galois.primitive_poly(3**3, 4); c
Out[19]: Poly(x^4 + x + 10, GF(3^3))

In [20]: lfsr = galois.FLFSR(c.reverse())

In [21]: print(lfsr)
Fibonacci LFSR:
   field: GF(3^3)
   feedback_poly: 10x^4 + x^3 + 1
   characteristic_poly: x^4 + x + 10
   taps: [0, 0, 2, 20]
   order: 4
   state: [1, 1, 1, 1]
   initial_state: [1, 1, 1, 1]
```

Step the Fibonacci LFSR and produce 10 output symbols.

```
In [22]: lfsr.state
Out[22]: GF([1, 1, 1, 1], order=3^3)
```
Constructors

**FLFSR** *(feedback\_poly: Poly, state: ArrayLike | None = None)*

Constructs a Fibonacci LFSR from its feedback polynomial $f(x)$.

classmethod **Taps**(taps: FieldArray,...) → FLFSR

Constructs a Fibonacci LFSR from its taps $T = [c_{n-1}, c_{n-2}, \ldots, c_1, c_0]$.

galois.FLFSR(feedback\_poly: Poly, state: ArrayLike | None = None)

Constructs a Fibonacci LFSR from its feedback polynomial $f(x)$.

Parameters

**feedback\_poly:** Poly  
The feedback polynomial $f(x) = -c_0x^n - c_1x^{n-1} - \cdots - c_{n-2}x^2 - c_{n-1}x + 1$.

**state:** ArrayLike | None = None  
The initial state vector $S = [S_0, S_1, \ldots, S_{n-2}, S_{n-1}]$. The default is None which corresponds to all ones.

See also:  
irreducible\_poly, primitive\_poly

Notes

A Fibonacci LFSR may be constructed from its characteristic polynomial $c(x)$ by passing in its reciprocal as the feedback polynomial. This is because $f(x) = x^n c(x^{-1})$.

classmethod galois.FLFSR.Taps(taps: FieldArray, state: ArrayLike | None = None) → FLFSR

Constructs a Fibonacci LFSR from its taps $T = [c_{n-1}, c_{n-2}, \ldots, c_1, c_0]$.

Parameters

**taps:** FieldArray  
The shift register taps $T = [c_{n-1}, c_{n-2}, \ldots, c_1, c_0]$.

**state:** ArrayLike | None = None  
The initial state vector $S = [S_0, S_1, \ldots, S_{n-2}, S_{n-1}]$. The default is None which corresponds to all ones.

Returns

A Fibonacci LFSR with taps $T = [c_{n-1}, c_{n-2}, \ldots, c_1, c_0]$.
Examples

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

In [2]: taps = -c.cofffs[1:]; taps
Out[2]: GF([0, 6, 4, 2], order=7)

In [3]: lfsr = galois.FLFSR.Taps(taps)

In [4]: print(lfsr)
Fibonacci LFSR:
    field: GF(7)
    feedback_poly: 5x^4 + 3x^3 + x^2 + 1
    characteristic_poly: x^4 + x^2 + 3x + 5
    taps: [0, 6, 4, 2]
    order: 4
    state: [1, 1, 1, 1]
    initial_state: [1, 1, 1, 1]
```

String representation

```
__repr__() → str
    A terse representation of the Fibonacci LFSR.

__str__() → str
    A formatted string of relevant properties of the Fibonacci LFSR.

galois.FLFSR.__repr__() → str
    A terse representation of the Fibonacci LFSR.
```

Examples

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

In [2]: lfsr = galois.FLFSR(c.reverse())

In [3]: lfsr
Out[3]: <Fibonacci LFSR: f(x) = 5x^4 + 3x^3 + x^2 + 1 over GF(7)>
```

galois.FLFSR.__str__() → str
    A formatted string of relevant properties of the Fibonacci LFSR.
Examples

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

In [2]: lfsr = galois.FLFSR(c.reverse())

In [3]: print(lfsr)
Fibonacci LFSR:
  field: GF(7)
  feedback_poly: 5x^4 + 3x^3 + x^2 + 1
  characteristic_poly: x^4 + x^2 + 3x + 5
  taps: [0, 6, 4, 2]
  order: 4
  state: [1, 1, 1, 1]
  initial_state: [1, 1, 1, 1]
```

Methods

- **reset** *(state: ArrayLike | None = None)*
  Resets the Fibonacci LFSR state to the specified state.

- **step** *(steps: int = 1) → FieldArray*
  Produces the next steps output symbols.

- **to_galois_lfsr() → GLFSR**
  Converts the Fibonacci LFSR to a Galois LFSR that produces the same output.

Examples

Initial state

Step the Fibonacci LFSR 10 steps to modify its state.

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

In [2]: lfsr = galois.FLFSR(c.reverse()); lfsr
Out[2]: <Fibonacci LFSR: f(x) = 5x^4 + 3x^3 + x^2 + 1 over GF(7)>

In [3]: lfsr.state
Out[3]: GF([1, 1, 1, 1], order=7)
```
```python
In [4]: lfsr.step(10)
Out[4]: GF([1, 1, 1, 1, 5, 5, 1, 3, 1, 4], order=7)

In [5]: lfsr.state
Out[5]: GF([5, 5, 6, 6], order=7)
```

Reset the Fibonacci LFSR state.

```python
In [6]: lfsr.reset()

In [7]: lfsr.state
Out[7]: GF([1, 1, 1, 1], order=7)
```

**Specific state**

Create an Fibonacci LFSR and view its initial state.

```python
In [8]: c = galois.primitive_poly(7, 4); c
Out[8]: Poly(x^4 + x^2 + 3x + 5, GF(7))

In [9]: lfsr = galois.FLFSR(c.reverse()); lfsr
Out[9]: <Fibonacci LFSR: f(x) = 5x^4 + 3x^3 + x^2 + 1 over GF(7)>

In [10]: lfsr.state
Out[10]: GF([1, 1, 1, 1], order=7)
```

Reset the Fibonacci LFSR state to a new state.

```python
In [11]: lfsr.reset([1, 2, 3, 4])

In [12]: lfsr.state
Out[12]: GF([1, 2, 3, 4], order=7)
```

galois.FLFSR.step(steps: int = 1) → FieldArray

Produces the next steps output symbols.

Negative values may be passed which reverses the direction of the shift registers and produces outputs in reverse order.

**Parameters**

- **steps: int = 1**
  
  The direction and number of output symbols to produce. The default is 1. If negative, the Fibonacci LFSR will step backward.

**Returns**

- An array of output symbols of type field with size abs(steps).
Examples

Scalar output

Step the Fibonacci LFSR one output at a time. Notice the first $n$ outputs of a Fibonacci LFSR are its state reversed.

```
In [1]: c = galois.primitive_poly(7, 4)
In [2]: lfsr = galois.FLFSR(c.reverse(), state=[1, 2, 3, 4]); lfsr
Out[2]: <Fibonacci LFSR: f(x) = 5x^4 + 3x^3 + x^2 + 1 over GF(7)>
In [3]: lfsr.state, lfsr.step()
Out[3]: (GF([1, 2, 3, 4], order=7), GF(4, order=7))
In [4]: lfsr.state, lfsr.step()
Out[4]: (GF([4, 1, 2, 3], order=7), GF(3, order=7))
In [5]: lfsr.state, lfsr.step()
Out[5]: (GF([6, 4, 1, 2], order=7), GF(2, order=7))
In [6]: lfsr.state, lfsr.step()
Out[6]: (GF([4, 6, 4, 1], order=7), GF(1, order=7))
In [7]: lfsr.state, lfsr.step()
Out[7]: (GF([5, 4, 6, 4], order=7), GF(4, order=7))
# Ending state
In [8]: lfsr.state
Out[8]: GF([0, 5, 4, 6], order=7)
```

Vector output

Step the Fibonacci LFSR 5 steps in one call. This is more efficient than iterating one output at a time. Notice the output and ending state are the same from the Scalar output tab.

```
In [9]: c = galois.primitive_poly(7, 4)
In [10]: lfsr = galois.FLFSR(c.reverse(), state=[1, 2, 3, 4]); lfsr
Out[10]: <Fibonacci LFSR: f(x) = 5x^4 + 3x^3 + x^2 + 1 over GF(7)>
In [11]: lfsr.state
Out[11]: GF([1, 2, 3, 4], order=7)
In [12]: lfsr.step(5)
Out[12]: GF([4, 3, 2, 1, 4], order=7)
# Ending state
In [13]: lfsr.state
Out[13]: GF([0, 5, 4, 6], order=7)
```
Step backwards

Step the Fibonacci LFSR 10 steps forward.

```
In [14]: c = galois.primitive_poly(7, 4)
In [15]: lfsr = galois.FLFSR(c.reverse(), state=[1, 2, 3, 4]); lfsr
Out[15]: <Fibonacci LFSR: f(x) = 5x^4 + 3x^3 + x^2 + 1 over GF(7)>
In [16]: lfsr.state
Out[16]: GF([1, 2, 3, 4], order=7)
In [17]: lfsr.step(10)
Out[17]: GF([4, 3, 2, 1, 4, 6, 4, 5, 0, 2], order=7)
In [18]: lfsr.state
Out[18]: GF([3, 1, 1, 0], order=7)
```

Step the Fibonacci LFSR 10 steps backward. Notice the output sequence is the reverse of the original sequence. Also notice the ending state is the same as the initial state.

```
In [19]: lfsr.step(-10)
Out[19]: GF([2, 0, 5, 4, 6, 4, 1, 2, 3, 4], order=7)
In [20]: lfsr.state
Out[20]: GF([1, 2, 3, 4], order=7)
```

galois.FLFSR.to_galois_lfsr() \rightarrow GLFSR

Converts the Fibonacci LFSR to a Galois LFSR that produces the same output.

**Returns**

An equivalent Galois LFSR.

**Examples**

Create a Fibonacci LFSR with a given initial state.

```
In [1]: c = galois.primitive_poly(7, 4); c
Out[1]: Poly(x^4 + x^2 + 3x + 5, GF(7))
In [2]: fibonacci_lfsr = galois.FLFSR(c.reverse(), state=[1, 2, 3, 4])
In [3]: print(fibonacci_lfsr)
Fibonacci LFSR:
  field: GF(7)
  feedback_poly: 5x^4 + 3x^3 + x^2 + 1
  characteristic_poly: x^4 + x^3 + 2x + 5
  taps: [0, 6, 4, 2]
  order: 4
  state: [1, 2, 3, 4]
  initial_state: [1, 2, 3, 4]
```

Convert the Fibonacci LFSR to an equivalent Galois LFSR. Notice the initial state is different.
The characteristic polynomial $c(x) = x^n - c_{n-1}x^{n-1} - c_{n-2}x^{n-2} - \cdots - c_1 x - c_0$ that defines the linear recurrent sequence. The characteristic polynomial is the reciprocal of the feedback polynomial $c(x) = x^n f(x^{-1})$.

**property** feedback_poly : Poly

The feedback polynomial $f(x) = -c_0 x^n - c_1 x^{n-1} - \cdots - c_{n-2} x^2 - c_{n-1} x + 1$ that defines the feedback arithmetic. The feedback polynomial is the reciprocal of the characteristic polynomial $f(x) = x^n c(x^{-1})$.

**property** field : Type[FieldArray]

The FieldArray subclass for the finite field that defines the linear arithmetic.

**property** initial_state : FieldArray

The initial state vector $S = [S_0, S_1, \ldots, S_{n-2}, S_{n-1}]$.

**property** order : int

The order of the linear recurrence/linear recurrent sequence. The order of a sequence is defined by the degree of the minimal polynomial that produces it.

**property** state : FieldArray

The current state vector $S = [S_0, S_1, \ldots, S_{n-2}, S_{n-1}]$.

**property** taps : FieldArray

The shift register taps $T = [c_{n-1}, c_{n-2}, \ldots, c_1, c_0]$. The taps of the shift register define the linear recurrence relation.

**property** galois.LFSR.characteristic_poly : Poly

The characteristic polynomial $c(x) = x^n - c_{n-1}x^{n-1} - c_{n-2}x^{n-2} - \cdots - c_1 x - c_0$ that defines the linear recurrent sequence. The characteristic polynomial is the reciprocal of the feedback polynomial $c(x) = x^n f(x^{-1})$.
Examples

```python
In [1]: c = galois.primitive_poly(7, 4); c
Out[1]: Poly(x^4 + x^2 + 3x + 5, GF(7))

In [2]: lfsr = galois.FLFSR(c.reverse()); lfsr
Out[2]: <Fibonacci LFSR: f(x) = 5x^4 + 3x^3 + x^2 + 1 over GF(7)>

In [3]: lfsr.characteristic_poly
Out[3]: Poly(x^4 + x^2 + 3x + 5, GF(7))

In [4]: lfsr.characteristic_poly == lfsr.feedback_poly.reverse()
Out[4]: True
```

**property** `galois.FLFSR.feedback_poly` : `Poly`

The feedback polynomial \( f(x) = -c_0x^n - c_1x^{n-1} - \cdots - c_{n-2}x^2 - c_{n-1}x + 1 \) that defines the feedback arithmetic. The feedback polynomial is the reciprocal of the characteristic polynomial \( f(x) = x^n c(x^{-1}) \).

Examples

```python
In [1]: c = galois.primitive_poly(7, 4); c
Out[1]: Poly(x^4 + x^2 + 3x + 5, GF(7))

In [2]: lfsr = galois.FLFSR(c.reverse()); lfsr
Out[2]: <Fibonacci LFSR: f(x) = 5x^4 + 3x^3 + x^2 + 1 over GF(7)>

In [3]: lfsr.feedback_poly
Out[3]: Poly(5x^4 + 3x^3 + x^2 + 1, GF(7))

In [4]: lfsr.feedback_poly == lfsr.characteristic_poly.reverse()
Out[4]: True
```

**property** `galois.FLFSR.field` : `Type[FieldArray]`

The `FieldArray` subclass for the finite field that defines the linear arithmetic.

Examples

```python
In [1]: c = galois.primitive_poly(7, 4); c
Out[1]: Poly(x^4 + x^2 + 3x + 5, GF(7))

In [2]: lfsr = galois.FLFSR(c.reverse()); lfsr
Out[2]: <Fibonacci LFSR: f(x) = 5x^4 + 3x^3 + x^2 + 1 over GF(7)>

In [3]: lfsr.field
Out[3]: galois.GF(7)
```

**property** `galois.FLFSR.initial_state` : `FieldArray`

The initial state vector \( S = [S_0, S_1, \ldots, S_{n-2}, S_{n-1}] \).
Examples

```python
In [1]: c = galois.primitive_poly(7, 4)
In [2]: lfsr = galois.FLFSR(c.reverse(), state=[1, 2, 3, 4]); lfsr
Out[2]: <Fibonacci LFSR: f(x) = 5x^4 + 3x^3 + x^2 + 1 over GF(7)>
In [3]: lfsr.initial_state
Out[3]: GF([1, 2, 3, 4], order=7)
```

The initial state is unaffected as the Fibonacci LFSR is stepped.

```python
In [4]: lfsr.step(10)
Out[4]: GF([4, 3, 2, 1, 4, 6, 4, 5, 0, 2], order=7)
In [5]: lfsr.initial_state
Out[5]: GF([1, 2, 3, 4], order=7)
```

**property** `galois.FLFSR.order`: int

The order of the linear recurrence/linear recurrent sequence. The order of a sequence is defined by the degree of the minimal polynomial that produces it.

**property** `galois.FLFSR.state`: `FieldArray`

The current state vector $S = [S_0, S_1, \ldots, S_{n-2}, S_{n-1}]$.

Examples

```python
In [1]: c = galois.primitive_poly(7, 4)
In [2]: lfsr = galois.FLFSR(c.reverse(), state=[1, 2, 3, 4]); lfsr
Out[2]: <Fibonacci LFSR: f(x) = 5x^4 + 3x^3 + x^2 + 1 over GF(7)>
In [3]: lfsr.state
Out[3]: GF([1, 2, 3, 4], order=7)
```

The current state is modified as the Fibonacci LFSR is stepped.

```python
In [4]: lfsr.step(10)
Out[4]: GF([4, 3, 2, 1, 4, 6, 4, 5, 0, 2], order=7)
In [5]: lfsr.state
Out[5]: GF([3, 1, 1, 0], order=7)
```

**property** `galois.FLFSR.taps`: `FieldArray`

The shift register taps $T = [c_{n-1}, c_{n-2}, \ldots, c_1, c_0]$. The taps of the shift register define the linear recurrence relation.
**Examples**

| In [1]: c = galois.primitive_poly(7, 4); c  |
| Out[1]: Poly(x^4 + x^2 + 3x + 5, GF(7)) |

| In [2]: taps = -c.coeffs[1:]; taps  |
| Out[2]: GF([0, 6, 4, 2], order=7) |

| In [3]: lfsr = galois.FLFSR.Taps(taps); lfsr  |
| Out[3]: Fibonacci LFSR: f(x) = 5x^4 + 3x^3 + x^2 + 1 over GF(7) |

| In [4]: lfsr.taps  |
| Out[4]: GF([0, 6, 4, 2], order=7) |

### class galois.GLFSR

A Galois linear-feedback shift register (LFSR).

### Notes

A Galois LFSR is defined by its feedback polynomial \( f(x) \).

\[
f(x) = -c_0x^n - c_1x^{n-1} - \cdots - c_{n-2}x^2 - c_{n-1}x + 1 = x^n c(x^{-1})
\]

The feedback polynomial is the reciprocal of the characteristic polynomial \( c(x) \) of the linear recurrent sequence \( y \) produced by the Galois LFSR.

\[
c(x) = x^n - c_{n-1}x^{n-1} - c_{n-2}x^{n-2} - \cdots - c_1x - c_0
\]

\[
y_t = c_{n-1}y_{t-1} + c_{n-2}y_{t-2} + \cdots + c_1y_{t-n+2} + c_0y_{t-n+1}
\]

### Listing 2: Galois LFSR Configuration

```
References


See also:

berlekamp_massey

Examples

GF(2)

Create a Galois LFSR from a degree-4 primitive characteristic polynomial over GF(2).

```python
In [1]: c = galois.primitive_poly(2, 4); c
Out[1]: Poly(x^4 + x + 1, GF(2))
```

```python
In [2]: lfsr = galois.GLFSR(c.reverse())
```

```python
In [3]: print(lfsr)
Galois LFSR:
    field: GF(2)
    feedback_poly: x^4 + x^3 + 1
    characteristic_poly: x^4 + x + 1
    taps: [1, 1, 0, 0]
    order: 4
    state: [1, 1, 1, 1]
    initial_state: [1, 1, 1, 1]
```

Step the Galois LFSR and produce 10 output symbols.

```python
In [4]: lfsr.state
Out[4]: GF([1, 1, 1, 1], order=2)
```

```python
In [5]: lfsr.step(10)
Out[5]: GF([1, 1, 1, 0, 0, 0, 1, 0, 0, 1], order=2)
```

```python
In [6]: lfsr.state
Out[6]: GF([1, 1, 0, 1], order=2)
```

GF(p)

Create a Galois LFSR from a degree-4 primitive characteristic polynomial over GF(7).

```python
In [7]: c = galois.primitive_poly(7, 4); c
Out[7]: Poly(x^4 + x^2 + 3x + 5, GF(7))
```

```python
In [8]: lfsr = galois.GLFSR(c.reverse())
```

```python
In [9]: print(lfsr)
Galois LFSR:
    field: GF(7)
    feedback_poly: x^4 + x^2 + 3x + 5
    characteristic_poly: x^4 + x + 1
    taps: [1, 2, 3, 0]
    order: 4
    state: [1, 1, 1, 1]
    initial_state: [1, 1, 1, 1]
```

(continues on next page)
Step the Galois LFSR and produce 10 output symbols.

```
In [10]: lfsr.state
Out[10]: GF([1, 1, 1, 1], order=7)

In [11]: lfsr.step(10)
Out[11]: GF([1, 1, 0, 4, 6, 5, 3, 6, 1, 2], order=7)

In [12]: lfsr.state
Out[12]: GF([4, 3, 0, 1], order=7)
```

**GF(2^m)**

Create a Galois LFSR from a degree-4 primitive characteristic polynomial over GF(2^3).

```
In [13]: c = galois.primitive_poly(2**3, 4); c
Out[13]: Poly(x^4 + x + 3, GF(2^3))

In [14]: lfsr = galois.GLFSR(c.reverse())

In [15]: print(lfsr)
Galois LFSR:
   field: GF(2^3)
   feedback_poly: 3x^4 + x^3 + 1
   characteristic_poly: x^4 + x + 3
   taps: [3, 1, 0, 0]
   order: 4
   state: [1, 1, 1, 1]
   initial_state: [1, 1, 1, 1]
```

Step the Galois LFSR and produce 10 output symbols.

```
In [16]: lfsr.state
Out[16]: GF([1, 1, 1, 1], order=2^3)

In [17]: lfsr.step(10)
Out[17]: GF([1, 1, 1, 0, 2, 2, 3, 2, 4, 5], order=2^3)

In [18]: lfsr.state
Out[18]: GF([4, 2, 2, 7], order=2^3)
```
Create a Galois LFSR from a degree-4 primitive characteristic polynomial over \( \text{GF}(3^3) \).

```
In [19]: c = galois.primitive_poly(3**3, 4); c
Out[19]: Poly(x^4 + x + 10, GF(3^3))

In [20]: lfsr = galois/GLFSR(c.reverse())

In [21]: print(lfsr)
Galois LFSR:
   field: GF(3^3)
   feedback_poly: 10x^4 + x^3 + 1
   characteristic_poly: x^4 + x + 10
   taps: [20, 2, 0, 0]
   order: 4
   state: [1, 1, 1, 1]
   initial_state: [1, 1, 1, 1]
```

Step the Galois LFSR and produce 10 output symbols.

```
In [22]: lfsr.state
Out[22]: GF([1, 1, 1, 1], order=3^3)

In [23]: lfsr.step(10)
Out[23]: GF([ 1, 1, 1, 0, 19, 19, 20, 11, 25, 24], order=3^3)

In [24]: lfsr.state
Out[24]: GF([23, 25, 6, 26], order=3^3)
```

### Constructors

**GLFSR** *(feedback_poly: Poly, state: ArrayLike | None = None)*

Constructs a Galois LFSR from its feedback polynomial \( f(x) \).

**classmethod Taps(taps: FieldArray, ...) → GLFSR**

Constructs a Galois LFSR from its taps \( T = [c_0, c_1, \ldots, c_{n-2}, c_{n-1}] \).

galois/GLFSR(feedback_poly: Poly, state: ArrayLike | None = None)

Constructs a Galois LFSR from its feedback polynomial \( f(x) \).

**Parameters**

- **feedback_poly: Poly**
  
  The feedback polynomial \( f(x) = -c_0x^n - c_1x^{n-1} - \cdots - c_{n-2}x^2 - c_{n-1}x + 1 \).

- **state: ArrayLike | None = None**
  
  The initial state vector \( S = [S_0, S_1, \ldots, S_{n-2}, S_{n-1}] \). The default is \( \text{None} \) which corresponds to all ones.

**See also:**

- **irreducible_poly, primitive_poly**
Notes

A Galois LFSR may be constructed from its characteristic polynomial $c(x)$ by passing in its reciprocal as the feedback polynomial. This is because $f(x) = x^n c(x^{-1})$.

classmethod galois.GLFSR.Taps(taps: FieldArray, state: ArrayLike | None = None) → GLFSR

Constructs a Galois LFSR from its taps $T = [c_0, c_1, \ldots, c_{n-2}, c_{n-1}]$.

Parameters

- **taps**: FieldArray
  The shift register taps $T = [c_0, c_1, \ldots, c_{n-2}, c_{n-1}]$.

- **state**: ArrayLike | None = None
  The initial state vector $S = [S_0, S_1, \ldots, S_{n-2}, S_{n-1}]$. The default is None which corresponds to all ones.

Returns

A Galois LFSR with taps $T = [c_0, c_1, \ldots, c_{n-2}, c_{n-1}]$.

Examples

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

In [2]: taps = -c.coeffs[1:][::-1]; taps
Out[2]: GF([2, 4, 6, 0], order=7)

In [3]: lfsr = galois.GLFSR.Taps(taps)

In [4]: print(lfsr)
Galois LFSR:
  field: GF(7)
  feedback_poly: 5x^4 + 3x^3 + x^2 + 1
  characteristic_poly: x^4 + x^2 + 3x + 5
  taps: [2, 4, 6, 0]
  order: 4
  state: [1, 1, 1, 1]
  initial_state: [1, 1, 1, 1]
```

String representation

- **__repr__() → str**
  A terse representation of the Galois LFSR.

- **__str__() → str**
  A formatted string of relevant properties of the Galois LFSR.

```python
galois.GLFSR.__repr__() → str
A terse representation of the Galois LFSR.
```
Examples

```python
In [1]: c = galois.primitive_poly(7, 4); c
Out[1]: Poly(x^4 + x^2 + 3x + 5, GF(7))

In [2]: lfsr = galois.GLFSR(c.reverse())

In [3]: lfsr
Out[3]: <Galois LFSR: f(x) = 5x^4 + 3x^3 + x^2 + 1 over GF(7)>
```

galois.GLFSR.__str__() → str

A formatted string of relevant properties of the Galois LFSR.

Examples

```python
In [1]: c = galois.primitive_poly(7, 4); c
Out[1]: Poly(x^4 + x^2 + 3x + 5, GF(7))

In [2]: lfsr = galois.GLFSR(c.reverse())

In [3]: print(lfsr)
Galois LFSR:
   field: GF(7)
   feedback_poly: 5x^4 + 3x^3 + x^2 + 1
   characteristic_poly: x^4 + x^2 + 3x + 5
   taps: [2, 4, 6, 0]
   order: 4
   state: [1, 1, 1, 1]
   initial_state: [1, 1, 1, 1]
```

Methods

**reset**(state: ArrayLike | *None* = *None*)

Resets the Galois LFSR state to the specified state.

**step**(steps: *int* = 1) → FieldArray

Produces the next *steps* output symbols.

**to_fibonacci_lfsr**() → FLFSR

Converts the Galois LFSR to a Fibonacci LFSR that produces the same output.

galois.GLFSR.reset(state: ArrayLike | *None* = *None*)

Resets the Galois LFSR state to the specified state.

**Parameters**

**state**: ArrayLike | *None* = *None*

The state vector $S = [S_0, S_1, \ldots, S_{n-2}, S_{n-1}]$. The default is *None* which corresponds to the initial state.
Examples

Initial state

Step the Galois LFSR 10 steps to modify its state.

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

In [2]: lfsr = galois.GLFSR(c.reverse()); lfsr
Out[2]: <Galois LFSR: f(x) = 5x^4 + 3x^3 + x^2 + 1 over GF(7)>

In [3]: lfsr.state
Out[3]: GF([1, 1, 1, 1], order=7)

In [4]: lfsr.step(10)
Out[4]: GF([1, 1, 0, 4, 6, 5, 3, 6, 1, 2], order=7)

In [5]: lfsr.state
Out[5]: GF([4, 3, 0, 1], order=7)

Reset the Galois LFSR state.

```

```
In [6]: lfsr.reset()

In [7]: lfsr.state
Out[7]: GF([1, 1, 1, 1], order=7)
```

Specific state

Create an Galois LFSR and view its initial state.

```
In [8]: c = galois.primitive_poly(7, 4); c
Out[8]: Poly(x^4 + x^2 + 3x + 5, GF(7))

In [9]: lfsr = galois.GLFSR(c.reverse()); lfsr
Out[9]: <Galois LFSR: f(x) = 5x^4 + 3x^3 + x^2 + 1 over GF(7)>

In [10]: lfsr.state
Out[10]: GF([1, 1, 1, 1], order=7)

Reset the Galois LFSR state to a new state.

```

```
In [11]: lfsr.reset([1, 2, 3, 4])

In [12]: lfsr.state
Out[12]: GF([1, 2, 3, 4], order=7)
```

galois.GLFSR.step(steps: int = 1) → FieldArray

Produces the next steps output symbols.

Negative values may be passed which reverses the direction of the shift registers and produces outputs in reverse order.
Parameters

steps: int = 1

The direction and number of output symbols to produce. The default is 1. If negative, the Galois LFSR will step backward.

Returns

An array of output symbols of type field with size abs(steps).

Examples

Scalar output

Step the Galois LFSR one output at a time.

```
In [1]: c = galois.primitive_poly(7, 4)
```

```
In [2]: lfsr = galois.GLFSR(c.reverse(), state=[1, 2, 3, 4]); lfsr
Out[2]: <Galois LFSR: f(x) = 5x^4 + 3x^3 + x^2 + 1 over GF(7)>
```

```
In [3]: lfsr.state, lfsr.step()
Out[3]: (GF([1, 2, 3, 4], order=7), GF(4, order=7))
```

```
In [4]: lfsr.state, lfsr.step()
Out[4]: (GF([1, 3, 5, 3], order=7), GF(3, order=7))
```

```
In [5]: lfsr.state, lfsr.step()
Out[5]: (GF([6, 6, 0, 5], order=7), GF(5, order=7))
```

```
In [6]: lfsr.state, lfsr.step()
Out[6]: (GF([3, 5, 1, 0], order=7), GF(0, order=7))
```

```
In [7]: lfsr.state, lfsr.step()
Out[7]: (GF([0, 3, 5, 1], order=7), GF(1, order=7))
```

# Ending state

```
In [8]: lfsr.state
Out[8]: GF([2, 4, 2, 5], order=7)
```

Vector output

Step the Galois LFSR 5 steps in one call. This is more efficient than iterating one output at a time. Notice the output and ending state are the same from the Scalar output tab.

```
In [9]: c = galois.primitive_poly(7, 4)
```

```
In [10]: lfsr = galois.GLFSR(c.reverse(), state=[1, 2, 3, 4]); lfsr
Out[10]: <Galois LFSR: f(x) = 5x^4 + 3x^3 + x^2 + 1 over GF(7)>
```

```
In [11]: lfsr.state
Out[11]: GF([1, 2, 3, 4], order=7)
```

```
In [12]: lfsr.step(5)
```
(continued from previous page)

```
Out[12]: GF([4, 3, 5, 0, 1], order=7)

# Ending state
In [13]: lfsr.state
Out[13]: GF([2, 4, 2, 5], order=7)
```

**Step backwards**

Step the Galois LFSR 10 steps forward.

```
In [14]: c = galois.primitive_poly(7, 4)
In [15]: lfsr = galois.GLFSR(c.reverse(), state=[1, 2, 3, 4]); lfsr
Out[15]: <Galois LFSR: f(x) = 5x^4 + 3x^3 + x^2 + 1 over GF(7)>
In [16]: lfsr.state
Out[16]: GF([1, 2, 3, 4], order=7)
In [17]: lfsr.step(10)
Out[17]: GF([4, 3, 5, 0, 1, 5, 2, 6, 6, 5], order=7)
In [18]: lfsr.state
Out[18]: GF([3, 4, 3, 1], order=7)
```

Step the Galois LFSR 10 steps backward. Notice the output sequence is the reverse of the original sequence. Also notice the ending state is the same as the initial state.

```
In [19]: lfsr.step(-10)
Out[19]: GF([5, 6, 6, 2, 5, 1, 0, 5, 3, 4], order=7)
In [20]: lfsr.state
Out[20]: GF([1, 2, 3, 4], order=7)
```

galois.GLFSR.to_fibonacci_lfsr() → FLFSR

Converts the Galois LFSR to a Fibonacci LFSR that produces the same output.

**Returns**

An equivalent Fibonacci LFSR.

**Examples**

Create a Galois LFSR with a given initial state.

```
In [1]: c = galois.primitive_poly(7, 4); c
Out[1]: Poly(x^4 + x^2 + 3x + 5, GF(7))
In [2]: galois_lfsr = galois.GLFSR(c.reverse(), state=[1, 2, 3, 4])
In [3]: print(galois_lfsr)
Galois LFSR:
  field: GF(7)
```

(continues on next page)
Convert the Galois LFSR to an equivalent Fibonacci LFSR. Notice the initial state is different.

In [4]: fibonacci_lfsr = galois_lfsr.to_fibonacci_lfsr()  
In [5]: print(fibonacci_lfsr)  
Fibonacci LFSR:  
field: GF(7)  
feedback_poly: 5x^4 + 3x^3 + x^2 + 1  
characteristic_poly: x^4 + x^2 + 3x + 5  
taps: [0, 6, 4, 2]  
order: 4  
state: [0, 5, 3, 4]  
initial_state: [0, 5, 3, 4]

Step both LFSRs and see that their output sequences are identical.

In [6]: galois_lfsr.step(10)  
Out[6]: GF([4, 3, 5, 0, 1, 5, 2, 6, 6, 5], order=7)  
In [7]: fibonacci_lfsr.step(10)  
Out[7]: GF([4, 3, 5, 0, 1, 5, 2, 6, 6, 5], order=7)

Properties

property characteristic_poly : Poly  
The characteristic polynomial $c(x) = x^n - c_{n-1}x^{n-1} - c_{n-2}x^{n-2} - \cdots - c_1x - c_0$ that defines the linear recurrent sequence. The characteristic polynomial is the reciprocal of the feedback polynomial $c(x) = x^n f(x^{-1})$.

property feedback_poly : Poly  
The feedback polynomial $f(x) = -c_0x^n - c_1x^{n-1} - \cdots - c_{n-2}x^2 - c_{n-1}x + 1$ that defines the feedback arithmetic. The feedback polynomial is the reciprocal of the characteristic polynomial $f(x) = x^n c(x^{-1})$.

property field : Type[FieldArray]  
The FieldArray subclass for the finite field that defines the linear arithmetic.

property initial_state : FieldArray  
The initial state vector $S = [S_0, S_1, \ldots, S_{n-2}, S_{n-1}]$.

property order : int  
The order of the linear recurrence/linear recurrent sequence. The order of a sequence is defined by the degree of the minimal polynomial that produces it.

property state : FieldArray  
The current state vector $S = [S_0, S_1, \ldots, S_{n-2}, S_{n-1}]$.  

3.18. galois


**property taps**: `FieldArray`

The shift register taps $T = [c_0, c_1, \ldots, c_{n-2}, c_{n-1}]$. The taps of the shift register define the linear recurrence relation.

**property galois.GLFSR.characteristic_poly**: `Poly`

The characteristic polynomial $c(x) = x^n - c_{n-1}x^{n-1} - c_{n-2}x^{n-2} - \cdots - c_1x - c_0$ that defines the linear recurrent sequence. The characteristic polynomial is the reciprocal of the feedback polynomial $c(x) = x^nf(x^{-1})$.

**Examples**

```python
In [1]: c = galois.primitive_poly(7, 4); c
Out[1]: Poly(x^4 + x^2 + 3x + 5, GF(7))

In [2]: lfsr = galois.GLFSR(c.reverse()); lfsr
Out[2]: <Galois LFSR: f(x) = 5x^4 + 3x^3 + x^2 + 1 over GF(7)>

In [3]: lfsr.characteristic_poly
Out[3]: Poly(x^4 + x^2 + 3x + 5, GF(7))

In [4]: lfsr.characteristic_poly == lfsr.feedback_poly.reverse()
Out[4]: True
```

**property galois.GLFSR.feedback_poly**: `Poly`

The feedback polynomial $f(x) = -c_0x^n - c_1x^{n-1} - \cdots - c_{n-2}x^2 - c_{n-1}x + 1$ that defines the feedback arithmetic. The feedback polynomial is the reciprocal of the characteristic polynomial $f(x) = x^nc(x^{-1})$.

**Examples**

```python
In [1]: c = galois.primitive_poly(7, 4); c
Out[1]: Poly(x^4 + x^2 + 3x + 5, GF(7))

In [2]: lfsr = galois.GLFSR(c.reverse()); lfsr
Out[2]: <Galois LFSR: f(x) = 5x^4 + 3x^3 + x^2 + 1 over GF(7)>

In [3]: lfsr.feedback_poly
Out[3]: Poly(5x^4 + 3x^3 + x^2 + 1, GF(7))

In [4]: lfsr.feedback_poly == lfsr.characteristic_poly.reverse()
Out[4]: True
```

**property galois.GLFSR.field**: `Type[FieldArray]`

The `FieldArray` subclass for the finite field that defines the linear arithmetic.
Examples

In [1]: c = galois.primitive_poly(7, 4); c
Out[1]: Poly(x^4 + x^2 + 3x + 5, GF(7))

In [2]: lfsr = galois.GLFSR(c.reverse()); lfsr
Out[2]: <Galois LFSR: f(x) = 5x^4 + 3x^3 + x^2 + 1 over GF(7)>

In [3]: lfsr.field
Out[3]: galois.GF(7)

property galois.GLFSR.initial_state: FieldArray
The initial state vector \(S = [S_0, S_1, ..., S_{n-2}, S_{n-1}]\).

Examples

In [1]: c = galois.primitive_poly(7, 4)
In [2]: lfsr = galois.GLFSR(c.reverse(), state=[1, 2, 3, 4]); lfsr
Out[2]: <Galois LFSR: f(x) = 5x^4 + 3x^3 + x^2 + 1 over GF(7)>

In [3]: lfsr.initial_state
Out[3]: GF([1, 2, 3, 4], order=7)

The initial state is unaffected as the Galois LFSR is stepped.

In [4]: lfsr.step(10)
Out[4]: GF([4, 3, 5, 0, 1, 5, 2, 6, 6, 5], order=7)

In [5]: lfsr.initial_state
Out[5]: GF([1, 2, 3, 4], order=7)

property galois.GLFSR.order: int
The order of the linear recurrence/linear recurrent sequence. The order of a sequence is defined by the degree of the minimal polynomial that produces it.

property galois.GLFSR.state: FieldArray
The current state vector \(S = [S_0, S_1, ..., S_{n-2}, S_{n-1}]\).

Examples

In [1]: c = galois.primitive_poly(7, 4)
In [2]: lfsr = galois.GLFSR(c.reverse(), state=[1, 2, 3, 4]); lfsr
Out[2]: <Galois LFSR: f(x) = 5x^4 + 3x^3 + x^2 + 1 over GF(7)>

In [3]: lfsr.state
Out[3]: GF([1, 2, 3, 4], order=7)

The current state is modified as the Galois LFSR is stepped.
property galois.GLFSR.taps: FieldArray

The shift register taps $T = [c_0, c_1, \ldots, c_{n-2}, c_{n-1}]$. The taps of the shift register define the linear recurrence relation.

Examples

```python
In [1]: c = galois.primitive_poly(7, 4); c
Out[1]: Poly(x^4 + x^2 + 3x + 5, GF(7))

In [2]: taps = -c.coeffs[1:][::-1]; taps
Out[2]: GF([2, 4, 6, 0], order=7)

In [3]: lfsr = galois.GLFSR.Taps(taps); lfsr
Out[3]: <Galois LFSR: f(x) = 5x^4 + 3x^3 + x^2 + 1 over GF(7)>

In [4]: lfsr.taps
Out[4]: GF([2, 4, 6, 0], order=7)
```

`galois.berlekamp_massey(sequence: FieldArray, output: 'minimal' = 'minimal') -> Poly`

`galois.berlekamp_massey(sequence: FieldArray, output: 'fibonacci') -> FLFSR`

`galois.berlekamp_massey(sequence: FieldArray, output: 'galois') -> GLFSR`

Finds the minimal polynomial $c(x)$ that produces the linear recurrent sequence $y$.

This function implements the Berlekamp-Massey algorithm.

Parameters

- **sequence**: FieldArray
  A linear recurrent sequence $y$ in $\mathbb{F}(p^m)$.

- **output**: 'minimal' = 'minimal'
  - 'minimal' (default): Returns the minimal polynomial that generates the linear recurrent sequence. The minimal polynomial is a characteristic polynomial $c(x)$ of minimal degree.
  - 'fibonacci': Returns a Fibonacci LFSR that produces $y$.
  - 'galois': Returns a Galois LFSR that produces $y$.

Returns

The minimal polynomial $c(x)$, a Fibonacci LFSR, or a Galois LFSR, depending on the value of `output`.
Notes

The minimal polynomial is the characteristic polynomial $c(x)$ of minimal degree that produces the linear recurrent sequence $y$.

$$c(x) = x^n - c_{n-1}x^{n-1} - c_{n-2}x^{n-2} - \cdots - c_1x - c_0$$

$$y_t = c_{n-1}y_{t-1} + c_{n-2}y_{t-2} + \cdots + c_1y_{t-n+2} + c_0y_{t-n+1}$$

For a linear sequence with order $n$, at least $2n$ output symbols are required to determine the minimal polynomial.

References


Examples

The sequence below is a degree-4 linear recurrent sequence over $\text{GF}(7)$.

```
In [1]: GF = galois.GF(7)
In [2]: y = GF([5, 5, 1, 3, 1, 4, 6, 6, 5, 5])
```

The characteristic polynomial is $c(x) = x^4 + x^2 + 3x + 5$ over $\text{GF}(7)$.

```
In [3]: galois.berlekamp_massey(y)
Out[3]: Poly(x^4 + x^2 + 3x + 5, GF(7))
```

Use the Berlekamp-Massey algorithm to return equivalent Fibonacci LFSR that reproduces the sequence.

```
In [4]: lfsr = galois.berlekamp_massey(y, output="fibonacci")
In [5]: print(lfsr)
Fibonacci LFSR:
  field: GF(7)
  feedback_poly: 5x^4 + 3x^3 + x^2 + 1
  characteristic_poly: x^4 + x^2 + 3x + 5
  taps: [0, 6, 4, 2]
  order: 4
  state: [3, 1, 5, 5]
  initial_state: [3, 1, 5, 5]
```

```
In [6]: z = lfsr.step(y.size); z
Out[6]: GF([5, 5, 1, 3, 1, 4, 6, 6, 5, 5], order=7)
```

```
In [7]: np.array_equal(y, z)
Out[7]: True
```

Use the Berlekamp-Massey algorithm to return equivalent Galois LFSR that reproduces the sequence.
In [8]: lfsr = galois.berlekamp_massey(y, output="galois")

In [9]: print(lfsr)
Galois LFSR:
   field: GF(7)
   feedback_poly: 5x^4 + 3x^3 + x^2 + 1
   characteristic_poly: x^4 + x^2 + 3x + 5
   taps: [2, 4, 6, 0]
   order: 4
   state: [2, 6, 5, 5]
   initial_state: [2, 6, 5, 5]

In [10]: z = lfsr.step(y.size); z
Out[10]: GF([5, 5, 1, 3, 1, 4, 6, 6, 5, 5], order=7)

In [11]: np.array_equal(y, z)
Out[11]: True

3.18.6 Transforms

\texttt{galois.intt}(x: \text{ArrayLike}, ...) \rightarrow \textit{FieldArray}

Computes the Inverse Number-Theoretic Transform (INTT) of \( X \).

\texttt{galois.ntt}(x: \text{ArrayLike}, \text{size: int | None = None}, ...) \rightarrow \textit{FieldArray}

Computes the Number-Theoretic Transform (NTT) of \( x \).

\texttt{galois.intt}(X: \text{ArrayLike}, \text{size: int | None = None, modulus: int | None = None, scaled: bool = True}) \rightarrow \textit{FieldArray}

Computes the Inverse Number-Theoretic Transform (INTT) of \( X \).

\textbf{Parameters}

\( X: \text{ArrayLike} \)

The input sequence of integers \( X \).

\text{size: int | None = None}

The size \( N \) of the INTT transform, must be at least the length of \( X \). The default is \texttt{None} which corresponds to \texttt{len(X)}. If \text{size} is larger than the length of \( X \), \( X \) is zero-padded.

\text{modulus: int | None = None}

The prime modulus \( p \) that defines the field \( GF(p) \). The prime modulus must satisfy \( p > \max(X) \) and \( p = mN + 1 \) (i.e., the size of the transform \( N \) must divide \( p - 1 \)). The default is \texttt{None} which corresponds to the smallest \( p \) that satisfies the criteria. However, if \( x \) is a \( GF(p) \) array, then \texttt{None} corresponds to \( p \) from the specified field.

\text{scaled: bool = True}

Indicates to scale the INTT output by \( N \). The default is \texttt{True}. If true, \( x = \text{INTT}(\text{NTT}(x)) \). If false, \( Nx = \text{INTT}(\text{NTT}(x)) \).

\textbf{Returns}

The INTT \( x \) of the input \( X \), with length \( N \). The output is a \( GF(p) \) array. It can be viewed as a normal NumPy array with \texttt{.view(np.ndarray)} or converted to a Python list with \texttt{.tolist()}.
Notes

The Number-Theoretic Transform (NTT) is a specialized Discrete Fourier Transform (DFT) over a finite field \( \mathbb{F}_p \) instead of over \( \mathbb{C} \). The DFT uses the primitive \( N \)-th root of unity \( \omega_N = e^{-i2\pi/N} \), but the NTT uses a primitive \( N \)-th root of unity in \( \mathbb{F}_p \). These roots are such that \( \omega_N^N = 1 \) and \( \omega_N^k \neq 1 \) for \( 0 < k < N \).

In \( \mathbb{F}_p \), where \( p \) is prime, a primitive \( N \)-th root of unity exists if \( N \) divides \( p - 1 \). If that is true, then \( p = mN + 1 \) for some integer \( m \). This function finds \( \omega_N \) by first finding a primitive \( p - 1 \)-th root of unity \( \omega_{p-1} \) in \( \mathbb{F}_p \) using `primitive_root()`. From there \( \omega_N \) is found from \( \omega_N = \omega_{p-1}^m \).

The \( j \)-th value of the scaled \( N \)-point INTT \( x = \text{INTT}(X) \) is

\[
x_j = \frac{1}{N} \sum_{k=0}^{N-1} X_k \omega_N^{-kj},
\]

with all arithmetic performed in \( \mathbb{F}_p \). The scaled INTT has the property that \( x = \text{INTT}(\text{NTT}(x)) \).

A radix-2 Cooley-Tukey FFT algorithm is implemented, which achieves \( O(N\log(N)) \).

References

- https://cgyurgyik.github.io/posts/2021/04/brief-introduction-to-ntt/
- https://www.nayuki.io/page/number-theoretic-transform-integer-dft
- https://www.geeksforgeeks.org/python-number-theoretic-transformation/

Examples

The default modulus is the smallest \( p \) such that \( p > \max(X) \) and \( p = mN + 1 \). With the input \( X = [0, 4, 3, 2] \) and \( N = 4 \), the default modulus is \( p = 5 \).

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

However, other moduli satisfy \( p > \max(X) \) and \( p = mN + 1 \). For instance, \( p = 13 \) and \( p = 17 \). Notice the INTT outputs are different with different moduli. So it is important to perform forward and reverse NTTs with the same modulus.

\begin{verbatim}
In [2]: galois.intt([0, 4, 3, 2], modulus=13)
Out[2]: GF([12, 5, 9, 0], order=13)

In [3]: galois.intt([0, 4, 3, 2], modulus=17)
Out[3]: GF([15, 14, 12, 10], order=17)
\end{verbatim}

Instead of explicitly specifying the prime modulus, a \( \mathbb{F}_p \) array may be explicitly passed in and the modulus is taken as \( p \).

\begin{verbatim}
In [4]: GF = galois.GF(13)

In [5]: X = GF([10, 8, 11, 1]); X
Out[5]: GF([10, 8, 11, 1], order=13)

In [6]: x = galois.intt(X); x
Out[6]: GF([1, 2, 3, 4], order=13)
\end{verbatim}
The forward NTT and scaled INTT are the identity transform, i.e. $x = \text{INTT}(\text{NTT}(x))$.

In [8]: $\text{GF} = \text{galois.GF}(13)$
In [9]: $x = \text{GF}([1, 2, 3, 4]); x$
Out[9]: $\text{GF}([1, 2, 3, 4], \text{order}=13)$
In [10]: $\text{galois.intt}(\text{galois.ntt}(x))$
Out[10]: $\text{GF}([1, 2, 3, 4], \text{order}=13)$

This is also true in the reverse order, i.e. $x = \text{NTT}(\text{INTT}(x))$.

In [11]: $\text{galois.ntt}(\text{galois.intt}(x))$
Out[11]: $\text{GF}([1, 2, 3, 4], \text{order}=13)$

The `numpy.fft.ifft()` function may also be used to compute the inverse NTT over $\text{GF}(p)$.

In [12]: $X = \text{np.fft.fft}(x); X$
Out[12]: $\text{GF}([10, 8, 11, 1], \text{order}=13)$
In [13]: $\text{np.fft.ifft}(X)$
Out[13]: $\text{GF}([1, 2, 3, 4], \text{order}=13)$

galois.n tt(x: ArrayLike, size: int | None = None, modulus: int | None = None) $\rightarrow$ FieldArray

Computes the Number-Theoretic Transform (NTT) of $x$.

Parameters

x: ArrayLike
  The input sequence of integers $x$.

size: int | None = None
  The size $N$ of the NTT transform, must be at least the length of $x$. The default is None which corresponds to $\text{len}(x)$. If size is larger than the length of $x$, $x$ is zero-padded.

modulus: int | None = None
  The prime modulus $p$ that defines the field $\text{GF}(p)$. The prime modulus must satisfy $p > \text{max}(x)$ and $p = mN + 1$ (i.e., the size of the transform $N$ must divide $p - 1$). The default is None which corresponds to the smallest $p$ that satisfies the criteria. However, if $x$ is a $\text{GF}(p)$ array, then None corresponds to $p$ from the specified field.

Returns

The NTT $X$ of the input $x$, with length $N$. The output is a $\text{GF}(p)$ array. It can be viewed as a normal NumPy array with .view(np.ndarray) or converted to a Python list with .tolist().

See also:

intt
Notes

The Number-Theoretic Transform (NTT) is a specialized Discrete Fourier Transform (DFT) over a finite field \( \text{GF}(p) \) instead of over \( \mathbb{C} \). The DFT uses the primitive \( N \)-th root of unity \( \omega_N = e^{-i2\pi/N} \), but the NTT uses a primitive \( N \)-th root of unity in \( \text{GF}(p) \). These roots are such that \( \omega_N^N = 1 \) and \( \omega_N^k \neq 1 \) for \( 0 < k < N \).

In \( \text{GF}(p) \), where \( p \) is prime, a primitive \( N \)-th root of unity exists if \( N \) divides \( p - 1 \). If that is true, then \( p = mN + 1 \) for some integer \( m \). This function finds \( \omega_N \) by first finding a primitive \( p - 1 \)-th root of unity \( \omega_{p-1} \) in \( \text{GF}(p) \) using \texttt{primitive_root()} \. From there \( \omega_N \) is found from \( \omega_N = \omega_{p-1}^m \).

The \( k \)-th value of the \( N \)-point NTT \( X = \text{NTT}(x) \) is

\[
X_k = \sum_{j=0}^{N-1} x_j \omega_N^j,
\]

with all arithmetic performed in \( \text{GF}(p) \).

A radix-2 Cooley-Tukey FFT algorithm is implemented, which achieves \( O(N\log(N)) \).

References

- https://cgyurgyik.github.io/posts/2021/04/brief-introduction-to-ntt/
- https://www.nayuki.io/page/number-theoretic-transform-integer-dft
- https://www.geeksforgeeks.org/python-number-theoretic-transformation/

Examples

The default modulus is the smallest \( p \) such that \( p > \max(x) \) and \( p = mN + 1 \). With the input \( x = [1, 2, 3, 4] \) and \( N = 4 \), the default modulus is \( p = 5 \).

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

However, other moduli satisfy \( p > \max(x) \) and \( p = mN + 1 \). For instance, \( p = 13 \) and \( p = 17 \). Notice the NTT outputs are different with different moduli. So it is important to perform forward and reverse NTTs with the same modulus.

```
In [2]: galois.ntt([1, 2, 3, 4], modulus=13)
Out[2]: GF([10, 8, 11, 1], order=13)

In [3]: galois.ntt([1, 2, 3, 4], modulus=17)
Out[3]: GF([10, 6, 15, 7], order=17)
```

Instead of explicitly specifying the prime modulus, a \( \text{GF}(p) \) array may be explicitly passed in and the modulus is taken as \( p \).

```
In [4]: GF = galois.GF(13)

In [5]: galois.ntt(GF([1, 2, 3, 4]))
Out[5]: GF([10, 8, 11, 1], order=13)
```

The \( \text{size} \) keyword argument allows convenient zero-padding of the input (to a power of two, for example).
The `numpy.fft.fft()` function may also be used to compute the NTT over \( GF(p) \).

```python
In [8]: GF = galois.GF(17)
In [9]: x = GF([1, 2, 3, 4, 5, 6])
In [10]: np.fft.fft(x, n=8)
```

```
Out[10]: GF([ 4, 8, 4, 1, 14, 11, 2, 15], order=17)
```

### 3.18.7 Number theory

#### Divisibility

- **galois.\texttt{are_coprime}(\texttt{*values}: int) → bool**
- **galois.\texttt{are_coprime}(\texttt{*values}: Poly) → bool**
  - Determines if the arguments are pairwise coprime.
- **galois.\texttt{egcd}(a: int, b: int) → Tuple[int, int, int]**
- **galois.\texttt{egcd}(a: Poly, b: Poly) → Tuple[Poly, Poly]**
  - Finds the multiplicands of \( a \) and \( b \) such that \( as + bt = \gcd(a, b) \).
- **galois.\texttt{euler_phi}(n: int) → int**
  - Counts the positive integers (totatives) in \([1, n)\) that are coprime to \( n \).
- **galois.\texttt{gcd}(a: int, b: int) → int**
- **galois.\texttt{gcd}(a: Poly, b: Poly) → Poly**
  - Finds the greatest common divisor of \( a \) and \( b \).
- **galois.\texttt{lcm}(\texttt{*values}: int) → int**
- **galois.\texttt{lcm}(\texttt{*values}: Poly) → Poly**
  - Computes the least common multiple of the arguments.
- **galois.\texttt{prod}(\texttt{*values}: int) → int**
- **galois.\texttt{prod}(\texttt{*values}: Poly) → Poly**
  - Computes the product of the arguments.
- **galois.\texttt{totatives}(n: int) → List[int]**
  - Returns the positive integers (totatives) in \([1, n)\) that are coprime to \( n \).
- **galois.\texttt{are_coprime}(\texttt{*values}: int) → bool**
- **galois.\texttt{are_coprime}(\texttt{*values}: Poly) → bool**
  - Determines if the arguments are pairwise coprime.

**Parameters**

- **\texttt{*values}**
  - Each argument must be an integer or polynomial.
Returns

**True** if the arguments are pairwise coprime.

See also:

* lcm, prod

Notes

A set of integers or polynomials are pairwise coprime if their LCM is equal to their product.

Examples

Integers

Determine if a set of integers are pairwise coprime.

```python
In [1]: galois.are_coprime(3, 4, 5)
Out[1]: True

In [2]: galois.are_coprime(3, 7, 9, 11)
Out[2]: False
```

Polynomials

Generate irreducible polynomials over GF(7).

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

In [4]: f1 = galois.irreducible_poly(7, 1); f1
Out[4]: Poly(x, GF(7))

In [5]: f2 = galois.irreducible_poly(7, 2); f2
Out[5]: Poly(x^2 + 1, GF(7))

In [6]: f3 = galois.irreducible_poly(7, 3); f3
Out[6]: Poly(x^3 + 2, GF(7))
```

Determine if combinations of the irreducible polynomials are pairwise coprime.

```python
In [7]: galois.are_coprime(f1, f2, f3)
Out[7]: True

In [8]: galois.are_coprime(f1 * f2, f2, f3)
Out[8]: False
```

__galois.egcd__

\[
\text{galois.egcd}(a: \text{ int}, b: \text{ int}) \rightarrow \text{Tuple}[\text{int}, \text{int}, \text{int}]
\]

\[
\text{galois.egcd}(a: \text{ Poly}, b: \text{ Poly}) \rightarrow \text{Tuple}[\text{Poly}, \text{Poly}, \text{Poly}]
\]

Finds the multiplicands of \(a\) and \(b\) such that \(as + bt = \gcd(a, b)\).

Parameters
a: int
The first integer or polynomial argument.

b: int
Poly
The second integer or polynomial argument.

Returns
• Greatest common divisor of a and b.
• The multiplicand s of a.
• The multiplicand t of b.

See also:
gcd, lcm, prod

Notes
This function implements the Extended Euclidean Algorithm.

References
• Algorithm 2.107 from https://cacr.uwaterloo.ca/hac/about/chap2.pdf
• Algorithm 2.221 from https://cacr.uwaterloo.ca/hac/about/chap2.pdf
• Moon, T. “Error Correction Coding”, Section 5.2.2: The Euclidean Algorithm and Euclidean Domains, p. 181

Examples

Integers
Compute the extended GCD of two integers.

<table>
<thead>
<tr>
<th>In [1]:</th>
<th>a, b = 12, 16</th>
</tr>
</thead>
<tbody>
<tr>
<td>In [2]:</td>
<td>gcd, s, t = galois.egcd(a, b)</td>
</tr>
<tr>
<td>In [3]:</td>
<td>gcd, s, t</td>
</tr>
<tr>
<td>Out[3]:</td>
<td>(4, -1, 1)</td>
</tr>
<tr>
<td>In [4]:</td>
<td>a<em>s + b</em>t == gcd</td>
</tr>
<tr>
<td>Out[4]:</td>
<td>True</td>
</tr>
</tbody>
</table>
Polynomials

Generate irreducible polynomials over GF(7).

```
In [5]: GF = galois.GF(7)
In [6]: f1 = galois.irreducible_poly(7, 1); f1
Out[6]: Poly(x, GF(7))
In [7]: f2 = galois.irreducible_poly(7, 2); f2
Out[7]: Poly(x^2 + 1, GF(7))
In [8]: f3 = galois.irreducible_poly(7, 3); f3
Out[8]: Poly(x^3 + 2, GF(7))
```

Compute the extended GCD of $f_1(x)^2 f_2(x)$ and $f_1(x)f_3(x)$.

```
In [9]: a = f1**2 * f2
In [10]: b = f1 * f3
In [11]: gcd, s, t = galois.egcd(a, b)
In [12]: gcd, s, t
Out[12]: (Poly(x, GF(7)), Poly(2x^2 + 4x + 1, GF(7)), Poly(5x^2 + 3x + 4, GF(7)))
In [13]: a*s + b*t == gcd
Out[13]: True
```

galois.euler_phi(n: int) → int

Counts the positive integers (totatives) in $[1, n)$ that are coprime to $n$.

**Parameters**

- **n**: int
  - A positive integer.

**Returns**

The number of totatives that are coprime to $n$.

**See also:**

carmichael_lambda, totatives, is_cyclic

**Notes**

This function implements the Euler totient function

$$
\phi(n) = n \prod_{p \mid n} \left(1 - \frac{1}{p}\right) = \prod_{i=1}^{k} p_i^{e_i-1}(p_i - 1)
$$

for prime $p$ and the prime factorization $n = p_1^{e_1} \ldots p_k^{e_k}$. 
References

- Section 2.4.1 from https://cacr.uwaterloo.ca/hac/about/chap2.pdf
- https://oeis.org/A000010

Examples

Compute $\phi(20)$.

```python
In [1]: n = 20
In [2]: phi = galois.euler_phi(n); phi
Out[2]: 8
```

Find the totatives that are coprime with $n = 20$. The number of totatives of $n$ is $\phi(n)$.

```python
In [3]: x = galois.totatives(n); x
Out[3]: [1, 3, 7, 9, 11, 13, 17, 19]
In [4]: len(x) == phi
Out[4]: True
```

For prime $n$, $\phi(n) = n - 1$.

```python
In [5]: n = 13
In [6]: galois.euler_phi(n)
Out[6]: 12
```

galois.gcd($a: int, b: int$) $\rightarrow$ int

galois.gcd($a: Poly, b: Poly$) $\rightarrow$ Poly

Finds the greatest common divisor of $a$ and $b$.

**Parameters**

- $a$: int
- $a$: Poly
  
The first integer or polynomial argument.

- $b$: int
- $b$: Poly
  
The second integer or polynomial argument.

**Returns**

Greatest common divisor of $a$ and $b$.

See also:

egcd, lcm, prod
Notes

This function implements the Euclidean Algorithm.

References

- Algorithm 2.104 from https://cacr.uwaterloo.ca/hac/about/chap2.pdf
- Algorithm 2.218 from https://cacr.uwaterloo.ca/hac/about/chap2.pdf

Examples

Integers

Compute the GCD of two integers.

```python
In [1]: galois.gcd(12, 16)
Out[1]: 4
```

Polynomials

Generate irreducible polynomials over GF(7).

```python
In [2]: GF = galois.GF(7)
In [3]: f1 = galois.irreducible_poly(7, 1); f1
Out[3]: Poly(x, GF(7))
In [4]: f2 = galois.irreducible_poly(7, 2); f2
Out[4]: Poly(x^2 + 1, GF(7))
In [5]: f3 = galois.irreducible_poly(7, 3); f3
Out[5]: Poly(x^3 + 2, GF(7))
```

Compute the GCD of \( f_1(x)^2 f_2(x) \) and \( f_1(x)f_3(x) \), which is \( f_1(x) \).

```python
In [6]: galois.gcd(f1**2 * f2, f1 * f3)
Out[6]: Poly(x, GF(7))
```

\texttt{galois.lcm}(\texttt{*values: int}) \rightarrow \texttt{int}
\texttt{galois.lcm}(\texttt{*values: Poly}) \rightarrow \texttt{Poly}

Computes the least common multiple of the arguments.

Parameters

*values

Each argument must be an integer or polynomial.

Returns

The least common multiple of the arguments.

See also:

\texttt{gcd, egcd, prod}
Examples

Integers

Compute the LCM of three integers.

\begin{verbatim}
In [1]: galois.lcm(2, 4, 14)
Out[1]: 28
\end{verbatim}

Polynomials

Generate irreducible polynomials over \( GF(7) \).

\begin{verbatim}
In [2]: GF = galois.GF(7)
In [3]: f1 = galois.irreducible_poly(7, 1); f1
Out[3]: Poly(x, GF(7))
In [4]: f2 = galois.irreducible_poly(7, 2); f2
Out[4]: Poly(x^2 + 1, GF(7))
In [5]: f3 = galois.irreducible_poly(7, 3); f3
Out[5]: Poly(x^3 + 2, GF(7))
\end{verbatim}

Compute the LCM of three polynomials \( f_1(x)^2 f_2(x), f_1(x)f_3(x), \) and \( f_2(x)f_3(x) \), which is \( f_1(x)^2 f_2(x)f_3(x) \).

\begin{verbatim}
In [6]: galois.lcm(f1**2 * f2, f1 * f3, f2 * f3)
Out[6]: Poly(x^7 + x^5 + 2x^4 + 2x^2, GF(7))
In [7]: f1**2 * f2 * f3
Out[7]: Poly(x^7 + x^5 + 2x^4 + 2x^2, GF(7))
\end{verbatim}

\texttt{galois.prod}(*values: int) \rightarrow int
\texttt{galois.prod}(*values: Poly) \rightarrow Poly

Computes the product of the arguments.

Parameters

*values

Each argument must be an integer or polynomial.

Returns

The product of the arguments.

See also:

\texttt{gcd, egcd, lcm}
Examples

Integers

Compute the product of three integers.

```
In [1]: galois.prod(2, 4, 14)
Out[1]: 112
```

Polynomials

Generate random polynomials over GF(7).

```
In [2]: GF = galois.GF(7)
In [3]: f1 = galois.Poly.Random(2, field=GF); f1
Out[3]: Poly(2x^2 + 6x + 1, GF(7))
In [4]: f2 = galois.Poly.Random(3, field=GF); f2
Out[4]: Poly(2x^3 + x^2 + 6, GF(7))
In [5]: f3 = galois.Poly.Random(4, field=GF); f3
Out[5]: Poly(x^4 + 3x^3 + 2x^2 + 6x, GF(7))
```

Compute the product of three polynomials.

```
In [6]: galois.prod(f1, f2, f3)
Out[6]: Poly(4x^9 + 5x^8 + 2x^7 + 5x^6 + 6x^4 + 4x^2 + x, GF(7))
In [7]: f1 * f2 * f3
Out[7]: Poly(4x^9 + 5x^8 + 2x^7 + 5x^6 + 6x^4 + 4x^2 + x, GF(7))
```

galois.totatives(n: int) → List[int]

Returns the positive integers (totatives) in [1, n) that are coprime to n.

The totatives of n form the multiplicative group ($\mathbb{Z}/n\mathbb{Z}$)$^\times$.

**Parameters**

- **n**: int
  - A positive integer.

**Returns**

The totatives of n.

See also:

*euler_phi*, *carmichael_lambda*, *is_cyclic*
References

- Section 2.4.3 from https://cacr.uwaterloo.ca/hac/about/chap2.pdf
- https://oeis.org/A000010

Examples

Find the totatives that are coprime with $n = 20$.

```
In [1]: n = 20
In [2]: x = galois.totatives(n); x
Out[2]: [1, 3, 7, 9, 11, 13, 17, 19]
```

The number of totatives of $n$ is $\phi(n)$.

```
In [3]: phi = galois.euler_phi(n); phi
Out[3]: 8
In [4]: len(x) == phi
Out[4]: True
```

Congruences

galois\texttt{.carmichael_lambda}(n: int) \to int

Finds the smallest positive integer $m$ such that $a^m \equiv 1 \pmod{n}$ for every integer $a$ in $[1, n)$ that is coprime to $n$.

galois\texttt{.crt}(\texttt{remainders: Sequence[int]}, \texttt{moduli: Sequence[int]}) \to int

galois\texttt{.crt}(\texttt{remainders: Sequence[Poly]}, \texttt{moduli}) \to Poly

Solves the simultaneous system of congruences for $x$.

galois\texttt{.is_cyclic}(n: int) \to bool

Determines whether the multiplicative group $(\mathbb{Z}/n\mathbb{Z})^\times$ is cyclic.

galois\texttt{.jacobi_symbol}(a: int, n: int) \to int

Computes the Jacobi symbol $\left(\frac{a}{n}\right)$.

galois\texttt{.kronecker_symbol}(a: int, n: int) \to int

Computes the Kronecker symbol $\left(\frac{a}{n}\right)$.

galois\texttt{.legendre_symbol}(a: int, p: int) \to int

Computes the Legendre symbol $\left(\frac{a}{p}\right)$.

galois\texttt{.carmichael_lambda}(n: int) \to int

Finds the smallest positive integer $m$ such that $a^m \equiv 1 \pmod{n}$ for every integer $a$ in $[1, n)$ that is coprime to $n$. This function implements the Carmichael function $\lambda(n)$.

Parameters

- n: int
  A positive integer.
Returns
The smallest positive integer \( m \) such that \( a^m \equiv 1 \pmod{n} \) for every \( a \) in \([1, n)\) that is coprime to \( n \).

See also:
euler_phi, totatives, is_cyclic

References
- https://oeis.org/A002322

Examples
The Carmichael \( \lambda(n) \) function and Euler \( \phi(n) \) function are often equal. However, there are notable exceptions.

```
In [1]: [galois.euler_phi(n) for n in range(1, 20)]
Out[1]: [1, 1, 2, 2, 4, 2, 6, 4, 6, 4, 10, 4, 12, 6, 8, 8, 16, 6, 18]

In [2]: [galois.carmichael_lambda(n) for n in range(1, 20)]
Out[2]: [1, 1, 2, 2, 4, 2, 6, 2, 6, 4, 10, 2, 12, 6, 4, 4, 16, 6, 18]
```

For prime \( n \), \( \phi(n) = \lambda(n) = n - 1 \). And for most composite \( n \), \( \phi(n) = \lambda(n) < n - 1 \).

```
In [3]: n = 9

In [4]: phi = galois.euler_phi(n); phi
Out[4]: 6

In [5]: lambda_ = galois.carmichael_lambda(n); lambda_
Out[5]: 6

In [6]: totatives = galois.totatives(n); totatives
Out[6]: [1, 2, 4, 5, 7, 8]

In [7]: for power in range(1, phi + 1):
   ...:     y = [pow(a, power, n) for a in totatives]
   ...:     print("Power {}: {} (mod {})".format(power, y, n))
   ...:
Power 1: [1, 2, 4, 5, 7, 8] (mod 9)
Power 2: [1, 4, 7, 7, 4, 1] (mod 9)
Power 3: [1, 8, 1, 8, 1, 8] (mod 9)
Power 4: [1, 7, 4, 4, 7, 1] (mod 9)
Power 5: [1, 5, 7, 2, 4, 8] (mod 9)
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 \texttt{is_cyclic()}.

```
In [9]: n = 8
```
In [10]: phi = galois.euler_phi(n); phi
Out[10]: 4

In [11]: lambda_ = galois.carmichael_lambda(n); lambda_
Out[11]: 2

In [12]: totatives = galois.totatives(n); totatives
Out[12]: [1, 3, 5, 7]

In [13]: for power in range(1, phi + 1):
   ....:   y = [pow(a, power, n) for a in totatives]
   ....:   print("Power {}": {} (mod {})").format(power, y, n))
   ....: Power 1: [1, 3, 5, 7] (mod 8)
   Power 2: [1, 1, 1, 1] (mod 8)
   Power 3: [1, 3, 5, 7] (mod 8)
   Power 4: [1, 1, 1, 1] (mod 8)

In [14]: galois.is_cyclic(n)
Out[14]: False

galois.crt(remainders: Sequence[int], moduli: Sequence[int]) → int
galois.crt(remainders: Sequence[Poly], moduli: Sequence[Poly]) → Poly

Solves the simultaneous system of congruences for x.

Parameters

- remainders: Sequence[int]
- remainders: Sequence[Poly]
  The integer or polynomial remainders $a_i$.

- moduli: Sequence[int]
- moduli: Sequence[Poly]
  The integer or polynomial moduli $m_i$.

Returns

The simultaneous solution $x$ to the system of congruences.

Notes

This function implements the Chinese Remainder Theorem.

\[ x \equiv a_1 \pmod{m_1} \]
\[ x \equiv a_2 \pmod{m_2} \]
\[ x \equiv \ldots \]
\[ x \equiv a_n \pmod{m_n} \]
References

- Section 14.5 from https://cacr.uwaterloo.ca/hac/about/chap14.pdf

Examples

Integers

Define a system of integer congruences.

```python
In [1]: a = [0, 3, 4]
In [2]: m = [3, 4, 5]
```

Solve the system of congruences.

```python
In [3]: x = galois.crt(a, m); x
Out[3]: 39
```

Show that the solution satisfies each congruence.

```python
In [4]: for i in range(len(a)):
   ...:     ai = x % m[i]
   ...:     print(ai, ai == a[i])
   ...:
0  True
3  True
4  True
```

Polynomials

Define a system of polynomial congruences over GF(7).

```python
In [5]: GF = galois.GF(7)
In [6]: x_truth = galois.Poly.Random(6, field=GF); x_truth
Out[6]: Poly(x^6 + 4x^3 + x + 5, GF(7))
In [7]: m = [galois.Poly.Random(3, field=GF), galois.Poly.Random(4, field=GF),
   ...:     galois.Poly.Random(5, field=GF)]; m
Out[7]:
[Poly(4x^3 + 3x^2 + 1, GF(7)),
 Poly(6x^4 + 6x^3 + x^2 + 4, GF(7)),
 Poly(6x^5 + x^4 + 5x^3 + 6x + 2, GF(7))]
In [8]: a = [x_truth % mi for mi in m]; a
Out[8]:
[Poly(x^2 + 6x + 6, GF(7)),
 Poly(x^3 + 6x^2 + 4x + 6, GF(7)),
 Poly(6x^4 + 2x^3 + 6x^2 + 2x, GF(7))]
```

Solve the system of congruences.
show that the solution satisfies each congruence.

```
In [10]: for i in range(len(a)):
....:   ai = x % m[i]
....:   print(ai, ai == a[i])
.....
x^2 + 6x + 6 True
x^3 + 6x^2 + 4x + 6 True
6x^4 + 2x^3 + 6x^2 + 2x True
```

galois.is_cyclic(n: int) → bool

Determines whether the multiplicative group \((\mathbb{Z}/n\mathbb{Z})^\times\) is cyclic.

**Parameters**

- **n**: int
  A positive integer.

**Returns**

- **True** if the multiplicative group \((\mathbb{Z}/n\mathbb{Z})^\times\) is cyclic.

**See also:**

euler_phi, carmichael_lambda, totatives

**Notes**

The multiplicative group \((\mathbb{Z}/n\mathbb{Z})^\times\) is the set of positive integers \(1 \leq a < n\) 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 \(\{1, g, g^2, \ldots, g^{\phi(n)-1}\}\),
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**

\(n = 14\)

The elements of \((\mathbb{Z}/14\mathbb{Z})^\times\) = \{1, 3, 5, 9, 11, 13\} are the totatives of 14.

```
In [1]: n = 14
In [2]: Znx = galois.totatives(n); Znx
Out[2]: [1, 3, 5, 9, 11, 13]
```

The Euler totient \(\phi(n)\) function counts the totatives of \(n\), which is equivalent to the order of \((\mathbb{Z}/n\mathbb{Z})^\times\).

```
In [3]: phi = galois.euler_phi(n); phi
Out[3]: 6
In [4]: len(Znx) == phi
Out[4]: True
```
Since 14 is of the form $2^p$, the multiplicative group $(\mathbb{Z}/14\mathbb{Z})^\times$ is cyclic, meaning there exists at least one element that generates the group by its powers.

```python
In [5]: galois.is_cyclic(n)
Out[5]: True
```

Find the smallest primitive root modulo 14. Observe that the powers of $g$ uniquely represent each element in $(\mathbb{Z}/14\mathbb{Z})^\times$.

```python
In [6]: g = galois.primitive_root(n); g
Out[6]: 3

In [7]: [pow(g, i, n) for i in range(0, phi)]
Out[7]: [1, 3, 9, 13, 11, 5]
```

Find the largest primitive root modulo 14. Observe that the powers of $g$ also uniquely represent each element in $(\mathbb{Z}/14\mathbb{Z})^\times$, although in a different order.

```python
In [8]: g = galois.primitive_root(n, method="max"); g
Out[8]: 5

In [9]: [pow(g, i, n) for i in range(0, phi)]
Out[9]: [1, 5, 11, 13, 9, 3]
```


n = 15

A non-cyclic group is $(\mathbb{Z}/15\mathbb{Z})^\times = \{1, 2, 4, 7, 8, 11, 13, 14\}$.

```python
In [10]: n = 15

In [11]: Znx = galois.totatives(n); Znx
Out[11]: [1, 2, 4, 7, 8, 11, 13, 14]

In [12]: phi = galois.euler_phi(n); phi
Out[12]: 8
```

Since 15 is not of the form $2, 4, p^k$, or $2p^k$, the multiplicative group $(\mathbb{Z}/15\mathbb{Z})^\times$ is not cyclic, meaning no elements exist whose powers generate the group.

```python
In [13]: galois.is_cyclic(n)
Out[13]: False
```

Below, every element is tested to see if it spans the group.

```python
In [14]: for a in Znx:
   ....:     span = set([pow(a, i, n) for i in range(0, phi)])
   ....:     primitive_root = span == set(Znx)
   ....:     print("Element: \{a\}, Span: \{:<13\}, Primitive root: {}\).format(a, str(span), primitive_root))
   ....:
Element: 1, Span: {1} , Primitive root: False
Element: 2, Span: {8, 1, 2, 4} , Primitive root: False
Element: 4, Span: {1, 4} , Primitive root: False
```
Element: 7, Span: \{1, 4, 13, 7\}, Primitive root: False
Element: 8, Span: \{8, 1, 2, 4\}, Primitive root: False
Element: 11, Span: \{1, 11\}, Primitive root: False
Element: 13, Span: \{1, 4, 13, 7\}, Primitive root: False
Element: 14, Span: \{1, 14\}, Primitive root: False

The Carmichael $\lambda(n)$ function finds the maximum multiplicative order of any element, which is 4 and not 8.

In [15]: galois.carmichael_lambda(n)
Out[15]: 4

Observe that no primitive roots modulo 15 exist and a RuntimeError is raised.

In [16]: galois.primitive_root(n)
---------------------------------------------------------------------------
StopIteration Traceback (most recent call last)
File ~/checkouts/readthedocs.org/user_builds/galois/checkouts/latest/galois/__modular.py:579, in primitive_root(n, start, stop, method)
 578 if method == "min":
---> 579 return next(primitive_roots(n, start, stop=stop))
 580 elif method == "max":
StopIteration:
The above exception was the direct cause of the following exception:

RuntimeError Traceback (most recent call last)
Input In [16], in <cell line: 1>()
----> 1 galois.primitive_root(n)

File ~/checkouts/readthedocs.org/user_builds/galois/checkouts/latest/galois/__modular.py:585, in primitive_root(n, start, stop, method)
 583 return _primitive_root_random_search(n, start, stop)
 584 except StopIteration as e:
---> 585 raise RuntimeError("No primitive roots modulo \{n\} exist in the range [\{start\}, \{stop\}].") from e

RuntimeError: No primitive roots modulo 15 exist in the range [1, 15].

Prime fields

For prime $n$, a primitive root modulo $n$ is also a primitive element of the Galois field $GF(n)$.

In [17]: n = 31

In [18]: galois.is_cyclic(n)
Out[18]: True

A primitive element is a generator of the multiplicative group $GF(p)^\times = \{1, 2, \ldots, p - 1\} = \{1, g, g^2, \ldots, g^{\phi(n)-1}\}$. 

284 Chapter 3. Citation
In [19]: GF = galois.GF(n)

In [20]: galois.primitive_root(n)
Out[20]: 3

In [21]: GF.primitive_element
Out[21]: GF(3, order=31)

The number of primitive roots/elements is \( \phi(\phi(n)) \).

In [22]: list(galois.primitive_roots(n))
Out[22]: [3, 11, 12, 13, 17, 21, 22, 24]

In [23]: GF.primitive_elements
Out[23]: GF([ 3, 11, 12, 13, 17, 21, 22, 24], order=31)

In [24]: galois.euler_phi(galois.euler_phi(n))
Out[24]: 8

galois.jacobi_symbol(a: int, n: int) → int

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\}.

See also:
legendre_symbol, kronecker_symbol

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 $Q_9 = \{1, 4, 7\}$ and these all satisfy $\left( \frac{a}{9} \right) = 1$. The quadratic non-residues modulo 9 are $\overline{Q}_9 = \{2, 3, 5, 6, 8\}$, but notice $\{2, 5, 8\}$ also satisfy $\left( \frac{a}{9} \right) = 1$. The set of integers $\{3, 6\}$ not coprime to 9 satisfies $\left( \frac{a}{9} \right) = 0$.

```python
In [1]: [pow(x, 2, 9) for x in range(9)]
Out[1]: [0, 1, 4, 0, 7, 7, 0, 4, 1]
In [2]: for a in range(9):
   ...:     print(f"({a} / 9) = \{galois.jacobi_symbol(a, 9)\}"")
   ...
(0 / 9) = 0
(1 / 9) = 1
(2 / 9) = 1
(3 / 9) = 0
(4 / 9) = 1
(5 / 9) = 1
(6 / 9) = 0
(7 / 9) = 1
(8 / 9) = 1
```

galois.kronecker_symbol(a: int, n: int) → int
Computes the Kronecker symbol $\left( \frac{a}{n} \right)$.

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 $\{0, -1, 1\}$.

See also:

`legendre_symbol`, `jacobi_symbol`

References

- Algorithm 2.149 from https://cacr.uwaterloo.ca/hac/about/chap2.pdf

galois.legendre_symbol(a: int, p: int) → int
Computes the Legendre symbol $\left( \frac{a}{p} \right)$.

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 $\{0, 1, -1\}$.
See also:

jacobi_symbol, kronecker_symbol

Notes

The Legendre symbol is useful for determining if $a$ is a quadratic residue modulo $p$, namely $a \in \mathbb{Q}_p$. A quadratic residue $a$ modulo $p$ satisfies $x^2 \equiv a \pmod{p}$ for some $x$.

$$\left( \frac{a}{p} \right) = \begin{cases} 0, & p \mid a \\ 1, & a \in \mathbb{Q}_p \\ -1, & a \in \mathbb{Q}_p \\ \end{cases}$$

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 $\mathbb{Q}_7 = \{3, 5, 6\}$.

```python
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
```

Primitive roots

galois.is_primitive_root(g: int, n: int) → bool

Determines if $g$ is a primitive root modulo $n$.

galois.primitive_root(n: int, start: int = 1, ...) → int

Finds a primitive root modulo $n$ in the range [start, stop).

galois.primitive_roots(n: int, start: int = 1, ...) → Iterator[int]

Iterates through all primitive roots modulo $n$ in the range [start, stop).

galois.is_primitive_root(g: int, n: int) → bool

Determines if $g$ is a primitive root modulo $n$.

Parameters

g: int
    A positive integer.
n: int
A positive integer.

Returns
True if \( g \) is a primitive root modulo \( n \).

See also:
primitive_root, primitive_roots, is_cyclic, euler_phi

Notes
The integer \( g \) is a primitive root modulo \( n \) if the totatives of \( n \), the positive integers \( 1 \leq a < n \) 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 \( n \),

\[
(\mathbb{Z}/n\mathbb{Z})^\times = \{1, g, g^2, \ldots, g^{\phi(n)-1}\}
\]

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]: list(galois.primitive_roots(7))
Out[1]: [3, 5]

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

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

galois.primitive_root(n: int, start: int = 1, stop: int | None = None, method: 'min' | 'max' | 'random' = 'min') → int

Finds a primitive root modulo \( n \) in the range \([\text{start}, \text{stop})\).

Parameters

- **n**: int
  A positive integer.

- **start**: int = 1
  Starting value (inclusive) in the search for a primitive root.

- **stop**: int | None = None
  Stopping value (exclusive) in the search for a primitive root. The default is None which corresponds to \( n \).

- **method**: 'min' | 'max' | 'random' = 'min'
  The search method for finding the primitive root.

Returns

A primitive root modulo \( n \) in the specified range.
Raises

RuntimeError – If no primitive roots exist in the specified range.

See also:

primitive_roots, is_primitive_root, is_cyclic, totatives, euler_phi, carmichael_lambda

Notes

The integer $g$ is a primitive root modulo $n$ if the totatives of $n$ can be generated by the powers of $g$. The totatives of $n$ are the positive integers in $[1, n)$ that are coprime with $n$.

Alternatively said, $g$ is a primitive root modulo $n$ if and only if $g$ is a generator of the multiplicative group of integers modulo $n$ $(\mathbb{Z}/n\mathbb{Z})^\times = \{1, g, g^2, \ldots, g^{\phi(n)-1}\}$, 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

• http://www.numbertheory.org/courses/MP313/lectures/lecture7/page1.html

Examples

n = 14

The elements of $(\mathbb{Z}/14\mathbb{Z})^\times = \{1, 3, 5, 9, 11, 13\}$ are the totatives of 14.

```
In  [1]: n = 14
In  [2]: Znx = galois.totatives(n); Znx
Out[2]: [1, 3, 5, 9, 11, 13]
```

The Euler totient $\phi(n)$ function counts the totatives of $n$, which is equivalent to the order of $(\mathbb{Z}/n\mathbb{Z})^\times$.

```
In  [3]: phi = galois.euler_phi(n); phi
Out[3]: 6
In  [4]: len(Znx) == phi
Out[4]: True
```

Since 14 is of the form $2p^k$, the multiplicative group $(\mathbb{Z}/14\mathbb{Z})^\times$ is cyclic, meaning there exists at least one element that generates the group by its powers.

```
In  [5]: galois.is_cyclic(n)
Out[5]: True
```

Find the smallest primitive root modulo 14. Observe that the powers of $g$ uniquely represent each element in $(\mathbb{Z}/14\mathbb{Z})^\times$.
Find the largest primitive root modulo 14. Observe that the powers of \( g \) also uniquely represent each element in \((\mathbb{Z}/14\mathbb{Z})^\times\), although in a different order.

\[
\text{In [8]: } g = \text{galois.primitive_root}(n, \text{method}="\text{max}"); g \\
\text{Out[8]: } 5
\]

\[
\text{In [9]: } [\text{pow}(g, i, n) \text{ for } i \text{ in range}(0, \phi)] \\
\text{Out[9]: } [1, 5, 11, 13, 9, 3]
\]

\( n = 15 \)

A non-cyclic group is \((\mathbb{Z}/15\mathbb{Z})^\times = \{1, 2, 4, 7, 8, 11, 13, 14\}\).

\[
\text{In [10]: } n = 15 \\
\text{In [11]: } \text{Znx} = \text{galois.totatives}(n); \text{Znx} \\
\text{Out[11]: } [1, 2, 4, 7, 8, 11, 13, 14] \\
\text{In [12]: } \phi = \text{galois.euler_phi}(n); \phi \\
\text{Out[12]: } 8
\]

Since 15 is not of the form \(2, 4, p^k\), or \(2p^k\), the multiplicative group \((\mathbb{Z}/15\mathbb{Z})^\times\) is not cyclic, meaning no elements exist whose powers generate the group.

\[
\text{In [13]: } \text{galois.is_cyclic}(n) \\
\text{Out[13]: } \text{False}
\]

Below, every element is tested to see if it spans the group.

\[
\text{In [14]: } \text{for } a \text{ in Znx:} \\
\text{......: } \text{span} = \text{set}([\text{pow}(a, i, n) \text{ for } i \text{ in range}(0, \phi)]) \\
\text{......: } \text{primitive_root} = \text{span} == \text{set}(\text{Znx}) \\
\text{......: } \text{print("Element: {a}, Span: {:<13}, Primitive root: {b}".format(a,} \\
\text{......: } \text{-str(span), primitive_root))} \\
\]

Element:  1, Span: {1} , Primitive root: False
Element:  2, Span: {8, 1, 2, 4} , Primitive root: False
Element:  4, Span: {1, 4} , Primitive root: False
Element:  7, Span: {1, 4, 13, 7}, Primitive root: False
Element:  8, Span: {8, 1, 2, 4} , Primitive root: False
Element: 11, Span: {1, 11} , Primitive root: False
Element: 13, Span: {1, 4, 13, 7}, Primitive root: False
Element: 14, Span: {1, 14} , Primitive root: False

The Carmichael \(\lambda(n)\) function finds the maximum multiplicative order of any element, which is 4 and not 8.
Observe that no primitive roots modulo 15 exist and a `RuntimeError` is raised.

```
In [16]: galois.primitive_root(n)
```

```
StopIteration
```

The above exception was the direct cause of the following exception:

```
RuntimeError: No primitive roots modulo 15 exist in the range [1, 15).
```

### Very large n

The algorithm is also efficient for very large $n$.

```
In [17]: n = 1000000000000000035000061
In [18]: phi = galois.euler_phi(n); phi
```

```
1000000000000000035000060
```

Find the smallest, the largest, and a random primitive root modulo $n$.

```
In [19]: galois.primitive_root(n)
Out[19]: 7

In [20]: galois.primitive_root(n, method="max")
Out[20]: 1000000000000000035000054

In [21]: galois.primitive_root(n, method="random")
Out[21]: 707699438972461127252700
```

galois.primitive_roots(n: int, start: int = 1, stop: int | None = None, reverse: bool = False) → Iterator[int]

Iterates through all primitive roots modulo $n$ in the range [start, stop).
Parameters

n: int
A positive integer.

start: int = 1
Starting value (inclusive) in the search for a primitive root. The default is 1.

stop: int | None = None
Stopping value (exclusive) in the search for a primitive root. The default is None which corresponds to

reverse: bool = False
Indicates to return the primitive roots from largest to smallest. The default is False.

Returns
An iterator over the primitive roots modulo \( n \) in the specified range.

See also:

primitive_root, is_primitive_root, is_cyclic, totatives, euler_phi, carmichael_lambda

Notes

The integer \( g \) is a primitive root modulo \( n \) if the totatives of \( n \) can be generated by the powers of \( g \). The totatives of \( n \) are the positive integers in \([1, n]\) that are coprime with \( n \).

Alternatively said, \( g \) is a primitive root modulo \( n \) if and only if \( g \) is a generator of the multiplicative group of integers modulo \( n \) \( \left( \mathbb{Z}/n\mathbb{Z} \right)^* = \{1, g, g^2, \ldots, g^{\phi(n)-1}\} \), where \( \phi(n) \) is order of the group.

If \( \left( \mathbb{Z}/n\mathbb{Z} \right)^* \) is cyclic, the number of primitive roots modulo \( n \) is given by \( \phi(\phi(n)) \).

References


Examples

All primitive roots modulo 31. You may also use tuple() on the returned generator.

\[
\text{In [1]: } \text{list(galois.primitive_roots(31))} \\
\text{Out[1]: } [3, 11, 12, 13, 17, 21, 22, 24]
\]

There are no primitive roots modulo 30.

\[
\text{In [2]: } \text{list(galois.primitive_roots(30))} \\
\text{Out[2]: } []
\]

Show the each primitive root modulo 22 generates the multiplicative group \( \left( \mathbb{Z}/22\mathbb{Z} \right)^* \).
In [3]: \( n = 22 \)

In [4]: \( Z_{nx} = \text{galois.totatives}(n); Z_{nx} \)
\[
\begin{bmatrix}
1, 3, 5, 7, 9, 13, 15, 17, 19, 21
\end{bmatrix}
\]

In [5]: \( \phi = \text{galois.euler_phi}(n); \phi \)
\[
10
\]

In [6]: for root in galois.primitive_roots(22):
   ...
   span = set(pow(root, i, n) for i in range(0, phi))
   ...
   print(f"Element: {root}, Span: {span}")

Element: 7, Span: \{1, 3, 5, 7, 9, 13, 15, 17, 19, 21\}
Element: 13, Span: \{1, 3, 5, 7, 9, 13, 15, 17, 19, 21\}
Element: 17, Span: \{1, 3, 5, 7, 9, 13, 15, 17, 19, 21\}
Element: 19, Span: \{1, 3, 5, 7, 9, 13, 15, 17, 19, 21\}

Find the three largest primitive roots modulo 31 in reversed order.

In [7]: generator = galois.primitive_roots(31, reverse=True)
Out[7]: \langle\text{generator object primitive_roots at 0x7f6c18423350}\rangle

In [8]: [next(generator) for _ in range(3)]
Out[8]: \{24, 22, 21\}

Loop over all the primitive roots in reversed order, only finding them as needed. The search cost for the roots that would have been found after the \texttt{break} condition is never incurred.

In [9]: for root in galois.primitive_roots(31, reverse=True):
   ...
   print(root)
   ...
   if root % 7 == 0:  # Arbitrary early exit condition
   ...
   break

24
22
21

\textbf{Integer arithmetic}

\texttt{galois.ilog(n: int, b: int)} \to int

Computes \( x = \lfloor \log_b(n) \rfloor \) such that \( b^x \leq n < b^{x+1} \).

\texttt{galois.iroot(n: int, k: int)} \to int

Computes \( x = \lfloor n^{\frac{1}{k}} \rfloor \) such that \( x^k \leq n < (x + 1)^k \).

\texttt{galois.isqrt(n: int)} \to int

Computes \( x = \lfloor \sqrt{n} \rfloor \) such that \( x^2 \leq n < (x + 1)^2 \).

\texttt{galois.ilog(n: int, b: int)} \to int

Computes \( x = \lfloor \log_b(n) \rfloor \) such that \( b^x \leq n < b^{x+1} \).
galois

\[
\begin{align*}
\text{n: int} & \quad \text{A positive integer.} \\
\text{b: int} & \quad \text{The logarithm base } b, \text{ must be at least 2.}
\end{align*}
\]

**Returns**

The integer logarithm base \( b \) of \( n \).

**See also:**

`iroot`, `isqrt`

**Examples**

```python
In [1]: n = 1000
In [2]: x = galois.ilog(n, 5); x
Out[2]: 4
In [3]: print(f"{5**x} \leq {n} < {5**(x + 1)}")
625 \leq 1000 < 3125
```

galois.\textit{iroot}(n: \textit{int}, k: \textit{int}) \to \textit{int}

Computes \( x = \lfloor n^{\frac{1}{k}} \rfloor \) such that \( x^k \leq n < (x + 1)^k \).

**Parameters**

\[
\begin{align*}
\text{n: int} & \quad \text{A non-negative integer.} \\
\text{k: int} & \quad \text{The positive root } k.
\end{align*}
\]

**Returns**

The integer \( k \)-th root of \( n \).

**See also:**

`isqrt`, `ilog`

**Examples**

```python
In [1]: n = 1000
In [2]: x = galois.iroot(n, 5); x
Out[2]: 3
In [3]: print(f"{x**5} \leq {n} < {(x + 1)**5}")
243 \leq 1000 < 1024
```

galois.\textit{isqrt}(n: \textit{int}) \to \textit{int}

Computes \( x = \lfloor \sqrt{n} \rfloor \) such that \( x^2 \leq n < (x + 1)^2 \).
This function is included for Python versions before 3.8. For Python 3.8 and later, this function calls `math.isqrt()` from the standard library.

**Parameters**

- **n**: int
  A non-negative integer.

**Returns**

The integer square root of \( n \).

**See also:**

`iroot`, `ilog`

**Examples**

```python
In [1]: n = 1000
In [2]: x = galois.isqrt(n); x
Out[2]: 31
In [3]: print(f"{x**2} <= {n} < {(x + 1)**2}\)
961 <= 1000 < 1024
```

### 3.18.8 Factorization

**Prime factorization**

`galois.factors(value: int) → Tuple[List[int], List[int]]`

`galois.factors(value: Poly) → Tuple[List[Poly], List[int]]`

Computes the prime factors of a positive integer or the irreducible factors of a non-constant, monic polynomial.

**Parameters**

- **value**: int or Poly
  A positive integer \( n \) or a non-constant, monic polynomial \( f(x) \).

**Returns**

- 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 of corresponding multiplicities \( \{e_1, e_2, \ldots, e_k\} \).
Notes

Integers

This function factors a positive integer $n$ into its $k$ prime factors such that $n = p_1^{e_1} p_2^{e_2} \ldots p_k^{e_k}$.

Steps:
1. Test if $n$ is prime. If so, return $[n], [1]$. See is_prime().
2. Test if $n$ is a perfect power, such that $n = x^k$. If so, prime factor $x$ and multiply the exponents by $k$. See perfect_power().
3. Use trial division with a list of primes up to $10^6$. If no residual factors, return the discovered prime factors. See trial_division().
4. Use Pollard’s Rho algorithm to find a non-trivial factor of the residual. Continue until all are found. See pollard_rho().

Polynomials

This function factors a monic polynomial $f(x)$ into its $k$ irreducible factors such that $f(x) = g_1(x)^{e_1} g_2(x)^{e_2} \ldots g_k(x)^{e_k}$.

Steps:
1. Apply the Square-Free Factorization algorithm to factor the monic polynomial into square-free polynomials. See Poly.square_free_factors().
2. Apply the Distinct-Degree Factorization algorithm to factor each square-free polynomial into a product of factors with the same degree. See Poly.distinct_degree_factors().
3. Apply the Equal-Degree Factorization algorithm to factor the product of factors of equal degree into their irreducible factors. See Poly.equal_degree_factors().

This factorization is also available in Poly.factors().

References

- Section 2.1 from https://people.csail.mit.edu/dmoshkov/courses/codes/poly-factorization.pdf

Examples

Integers

Construct a composite integer from prime factors.

```
In [1]: n = 2**3 * 3 * 5; n
Out[1]: 120
```

Factor the integer into its prime factors.

```
In [2]: galois.factors(n)
Out[2]: ([2, 3, 5], [3, 1, 1])
```
Polynomials

Generate irreducible polynomials over GF(3).

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

In [4]: g1 = galois.irreducible_poly(3, 3); g1
Out[4]: Poly(x^3 + 2x + 1, GF(3))

In [5]: g2 = galois.irreducible_poly(3, 4); g2
Out[5]: Poly(x^4 + x + 2, GF(3))

In [6]: g3 = galois.irreducible_poly(3, 5); g3
Out[6]: Poly(x^5 + 2x + 1, GF(3))
```

Construct a composite polynomial.

```
In [7]: e1, e2, e3 = 5, 4, 3

In [8]: f = g1**e1 * g2**e2 * g3**e3; f
Out[8]: Poly(x^46 + x^44 + 2x^41 + x^40 + 2x^39 + 2x^38 + 2x^37 + 2x^36 + 2x^34 + x^33 + 2x^32 + x^31 + 2x^30 + 2x^29 + 2x^28 + 2x^25 + 2x^24 + 2x^23 + x^20 + x^19 + x^18 + x^15 + 2x^10 + 2x^8 + x^5 + x^4 + x^3 + 1, GF(3))
```

Factor the polynomial into its irreducible factors over GF(3).

```
In [9]: galois.factors(f)
Out[9]: ([Poly(x^3 + 2x + 1, GF(3)), Poly(x^4 + x + 2, GF(3)), Poly(x^5 + 2x + 1, GF(3))], [5, 4, 3])
```

Composite factorization

```python
galois.divisor_sigma(n: int, k: int = 1) -> int
    Returns the sum of k-th powers of the positive divisors of n.

galois.divisors(n: int) -> List[int]
    Computes all positive integer divisors d of the integer n such that d | n.

galois.divisor_sigma(n: int, k: int = 1) -> int
    Returns the sum of k-th powers of the positive divisors of n.

    Parameters:
    n: int
        An integer.
    k: int = 1
        The degree of the positive divisors. The default is 1 which corresponds to \(\sigma_1(n)\) which is the sum of positive divisors.

    Returns:
    The sum of divisors function \(\sigma_k(n)\).
```
This function implements the $\sigma_k(n)$ function. It is defined as:

$$\sigma_k(n) = \sum_{d \mid n} d^k$$

**Examples**

```python
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
```

**Notes**

The `divisors()` function finds all positive integer divisors or factors of $n$, where the `factors()` function only finds the prime factors of $n$. 

**See also:**

`factors`, `divisor_sigma`
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])
```

Specific factorization algorithms

galois.**perfect_power**(n: int) → Tuple[int, int]

Returns the integer base \(c\) and exponent \(e\) of \(n = c^e\).

galois.**pollard_p1**(n: int, B: int, B2: int | None = None) → int

Attempts to find a non-trivial factor of \(n\) if it has a prime factor \(p\) such that \(p - 1\) is \(B\)-smooth.

galois.**pollard_rho**(n: int, c: int = 1) → int

Attempts to find a non-trivial factor of \(n\) using cycle detection.

galois.**trial_division**(n, ...) → Tuple[List[int], List[int], int]

Finds all the prime factors \(p_i^{e_i}\) of \(n\) for \(p_i \leq B\).

galois.**perfect_power**(n: int) → Tuple[int, int]

Returns the integer base \(c\) and exponent \(e\) of \(n = c^e\).

If \(n\) is a not perfect power, then \(c = n\) and \(e = 1\).

**Parameters**

- **n**: int
  An integer.

**Returns**

- The potentially composite base \(c\).
- The exponent \(e\).

**See also:**

`factors, is_perfect_power, is_prime_power`
Examples

Primes are not perfect powers because their exponent is 1.

```python
In [1]: n = 13
In [2]: galois.perfect_power(n)
Out[2]: (13, 1)
In [3]: galois.is_perfect_power(n)
Out[3]: False
```

Products of primes are not perfect powers.

```python
In [4]: n = 5 * 7
In [5]: galois.perfect_power(n)
Out[5]: (35, 1)
In [6]: galois.is_perfect_power(n)
Out[6]: False
```

Products of prime powers where the GCD of the exponents is 1 are not perfect powers.

```python
In [7]: n = 2 * 3 * 5**3
In [8]: galois.perfect_power(n)
Out[8]: (750, 1)
In [9]: galois.is_perfect_power(n)
Out[9]: False
```

Products of prime powers where the GCD of the exponents is greater than 1 are perfect powers.

```python
In [10]: n = 2**2 * 3**2 * 5**4
In [11]: galois.perfect_power(n)
Out[11]: (150, 2)
In [12]: galois.is_perfect_power(n)
Out[12]: True
```

Negative integers can be perfect powers if they can be factored with an odd exponent.

```python
In [13]: n = -64
In [14]: galois.perfect_power(n)
Out[14]: (-4, 3)
In [15]: galois.is_perfect_power(n)
Out[15]: True
```

Negative integers that are only factored with an even exponent are not perfect powers.
In [16]: n = -100
In [17]: galois.perfect_power(n)
Out[17]: (-100, 1)
In [18]: galois.is_perfect_power(n)
Out[18]: False

galois.pollard_p1(n: int, B: int, B2: int | None = None) → int
Attempts to find a non-trivial factor of \( n \) if it has a prime factor \( p \) such that \( p - 1 \) is \( B \)-smooth.

Parameters

- **n**: int
  An odd composite integer \( n > 2 \) that is not a prime power.
- **B**: int
  The smoothness bound \( B > 2 \).
- **B2**: int | None = None
  The smoothness bound \( B_2 \) for the optional second step of the algorithm. The default is None which will not perform the second step.

Returns

A non-trivial factor of \( n \), if found. None if not found.

Raises

- **RuntimeError** – If a non-trivial factor cannot be found.

See also:

- factors, pollard_rho

Notes

For a given odd composite \( n \) with a prime factor \( p \), Pollard’s \( p - 1 \) algorithm can discover a non-trivial factor of \( n \) if \( p - 1 \) is \( B \)-smooth. Specifically, the prime factorization must satisfy \( p - 1 = p_1^{e_1} \cdots p_k^{e_k} \) with each \( p_i \leq B \).

A extension of Pollard’s \( p - 1 \) algorithm allows a prime factor \( p \) to be \( B \)-smooth with the exception of one prime factor \( B < p_{k+1} \leq B_2 \). In this case, the prime factorization is \( p - 1 = p_1^{e_1} \cdots p_{k+1}^{e_{k+1}} \). Often \( B_2 \) is chosen such that \( B_2 \gg B \).

References

- Section 3.2.3 from https://cacr.uwaterloo.ca/hac/about/chap3.pdf
Examples

Here, \( n = pq \) where \( p - 1 \) is 1039-smooth and \( q - 1 \) is 17-smooth.

In [1]: p, q = 1458757, 1326001

In [2]: galois.factors(p - 1)
Out[2]: ([2, 3, 13, 1039], [2, 3, 1, 1])

In [3]: galois.factors(q - 1)
Out[3]: ([2, 3, 5, 13, 17], [4, 1, 3, 1, 1])

Searching with \( B = 15 \) will not recover a prime factor.

In [4]: galois.pollard_p1(p*q, 15)
---------------------------------------------------------------------------
RuntimeError Traceback (most recent call last)
Input In [4], in <cell line: 1>()
----> 1 galois.pollard_p1(p*q, 15)
File ~/checkouts/readthedocs.org/user_builds/galois/checkouts/latest/galois/_prime.py:1208, in pollard_p1(n, B, B2)
1205     if d not in [1, n]:
1206         return d
--> 1208     raise RuntimeError(f"A non-trivial factor of {n} could not be found using the Pollard p-1 algorithm with smoothness bound {B} and secondary bound {B2}.")

RuntimeError: A non-trivial factor of 1934313240757 could not be found using the Pollard p-1 algorithm with smoothness bound 15 and secondary bound None.

Searching with \( B = 17 \) will recover the prime factor \( q \).

In [5]: galois.pollard_p1(p*q, 17)
Out[5]: 1326001

Searching \( B = 15 \) will not recover a prime factor in the first step, but will find \( q \) in the second step because \( p_{k+1} = 17 \) satisfies \( 15 < 17 \leq 100 \).

In [6]: galois.pollard_p1(p*q, 15, B2=100)
Out[6]: 1326001

Pollard’s \( p - 1 \) algorithm may return a composite factor.

In [7]: n = 2133861346249

In [8]: galois.factors(n)
Out[8]: ([37, 41, 5471, 257107], [1, 1, 1, 1])

In [9]: galois.pollard_p1(n, 10)
Out[9]: 1517

In [10]: 37*41
Out[10]: 1517
galois.pollard_rho(n: int, c: int = 1) → int

Attempts to find a non-trivial factor of \( n \) using cycle detection.

**Parameters**

- **n**: int
  - An odd composite integer \( n > 2 \) that is not a prime power.

- **c**: int = 1
  - The constant offset in the function \( f(x) = x^2 + c \mod n \). 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. **None** if not found.

**Raises**

- **RuntimeError** – If a non-trivial factor cannot be found.

**See also:**

* factors, pollard_p1

**Notes**

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 \( x_i = f(x_{i-1}) = x_{i-1}^2 + 1 \mod p \) where \( p \) is an unknown small prime factor of \( n \). This happens when \( x_m \equiv x_{2m} \pmod{p} \). Because \( p \) is unknown, this is accomplished by computing the sequence modulo \( n \) and looking for \( \gcd(x_m - x_{2m}, n) > 1 \).

**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.

```
In [1]: n = 503**7 * 10007 * 1000003

In [2]: galois.pollard_rho(n)
Out[2]: 503
```

It is also efficient for finding relatively small factors.

```
In [3]: n = 1182640843 * 1716279751

In [4]: galois.pollard_rho(n)
Out[4]: 1716279751
```

galois.trial_division(n: int, B: int | None = None) → Tuple[List[int], List[int], int]

Finds all the prime factors \( p_i^{e_i} \) of \( n \) for \( p_i \leq B \).

The trial division factorization will find all prime factors \( p_i \leq B \) such that \( n \) factors as \( n = p_1^{e_1} \cdots p_k^{e_k} n_r \) where \( n_r \) is a residual factor (which may be composite).

**Parameters**
galois

n: int
A positive integer.

B: int | None = None
The max divisor in the trial division. The default is None which corresponds to \( B = \sqrt{n} \). If \( B > \sqrt{n} \), the algorithm will only search up to \( \sqrt{n} \), since a prime factor of \( n \) cannot be larger than \( \sqrt{n} \).

Returns

• The discovered prime factors \( \{p_1, \ldots, p_k\} \).
• The corresponding prime exponents \( \{e_1, \ldots, e_k\} \).
• The residual factor \( n_r \).

See also:
factors

Examples

```
In [1]: n = 2**4 * 17**3 * 113 * 15013
In [2]: galois.trial_division(n)
Out[2]: ([2, 17, 113, 15013], [4, 3, 1, 1], 1)
In [3]: galois.trial_division(n, B=500)
Out[3]: ([2, 17, 113], [4, 3, 1], 15013)
In [4]: galois.trial_division(n, B=100)
Out[4]: ([2, 17], [4, 3], 1696469)
```

3.18.9 Primes

Prime number generation

galois.kth_prime(k: int) \to int
Returns the \( k \)-th prime, where \( k = \{1, 2, 3, 4, \ldots\} \) for primes \( p = \{2, 3, 5, 7, \ldots\} \).

galois.mersenne_exponents(n: int | None = None) \to List[int]
Returns all known Mersenne exponents \( e \) for \( e \leq n \).

galois.mersenne_primes(n: int | None = None) \to List[int]
Returns all known Mersenne primes \( p \) for \( p \leq 2^n - 1 \).

galois.next_prime(n: int) \to int
Returns the nearest prime \( p \), such that \( p > n \).

galois.prev_prime(n: int) \to int
Returns the nearest prime \( p \), such that \( p \leq n \).

galois.primes(n: int) \to List[int]
Returns all primes \( p \) for \( p \leq n \).
galois.random_prime(bits: int) → int

Returns a random prime \( p \) with \( b \) bits, such that \( 2^b \leq p < 2^{b+1} \).

galois.kth_prime(k: int) → int

Returns the \( k \)-th prime, where \( k = \{1, 2, 3, 4, \ldots \} \) for primes \( p = \{2, 3, 5, 7, \ldots \} \).

Parameters

   k: int

   The prime index (1-indexed).

Returns

The \( k \)-th prime.

See also:

primes, prev_prime, next_prime

Examples

```python
In [1]: galois.kth_prime(1)
Out[1]: 2

In [2]: galois.kth_prime(2)
Out[2]: 3

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

In [4]: galois.kth_prime(1000)
Out[4]: 7919
```

galois.mersenne_exponents(n: int | None = None) → List[int]

Returns all known Mersenne exponents \( e \) for \( e \leq n \).

A Mersenne exponent \( e \) is an exponent of 2 such that \( 2^e - 1 \) is prime.

Parameters

   n: int | None = None

   The max exponent of 2. The default is \mathbf{None} which returns all known Mersenne exponents.

Returns

The list of Mersenne exponents \( e \) for \( e \leq n \).

See also:

mersenne_primes
galois

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]:
→ 10407932194664399081925249327364085538615262247266704805319112350403608059673360298512239441733239

In [3]: galois.is_prime(p)
Out[3]: True

galois.mersenne_primes(n: int | None = None) → List[int]

Returns all known Mersenne primes p for p ≤ 2^n − 1.

Mersenne primes are primes that are one less than a power of 2.

Parameters

n: int | None = None

The max power of 2. The default is None which returns all known Mersenne exponents.

Returns

The list of known Mersenne primes p for p ≤ 2^n − 1.

See also:

mersenne_exponents

References

• https://oeis.org/A000668

Examples

# List all Mersenne primes up to 2000 bits
In [1]: p = galois.mersenne_primes(2000); p
Out[1]: [3, 7, 31, 127, 8191, 131071, 524287, 2147483647, 2305843009213693951, (continues on next page)
In [2]: galois.is_prime(p[-1])
Out[2]: True

---

**galois.next_prime**

```
(n: int) \rightarrow int
```

Returns the nearest prime \( p \), such that \( p > n \).

**Parameters**

- **n**: int
  - An integer.

**Returns**

The nearest prime \( p > n \).

**See also:**

`primes, kth_prime, prev_prime`

**Examples**

```
In [1]: galois.next_prime(13)
Out[1]: 17

In [2]: galois.next_prime(15)
Out[2]: 17

In [3]: galois.next_prime(6852976918500265458318414454675831645298)
Out[3]: 6852976918500265458318414454675831645343
```

---

**galois.prev_prime**

```
(n: int) \rightarrow int
```

Returns the nearest prime \( p \), such that \( p \leq n \).

**Parameters**

- **n**: int
  - An integer \( n \geq 2 \).

**Returns**

The nearest prime \( p \leq n \).

**See also:**

`primes, kth_prime, next_prime`
Examples

```python
In [1]: galois.prev_prime(13)
Out[1]: 13

In [2]: galois.prev_prime(15)
Out[2]: 13

In [3]: galois.prev_prime(629889120124192954877199440981228280038)
Out[3]: 629889120124192954877199440981228279991
```

**galois.primes** *(n: int) → List[int]*

Returns all primes \( p \) for \( p \leq n \).

**Parameters**

- **n: int**
  - An integer.

**Returns**

All primes up to and including \( n \). If \( n < 2 \), the function returns an empty list.

**See also:**

kth_prime, prev_prime, next_prime

**Notes**

This function implements the Sieve of Eratosthenes to efficiently find the primes.

**References**

- [https://oeis.org/A000040](https://oeis.org/A000040)

Examples

```python
In [1]: galois.primes(19)
Out[1]: [2, 3, 5, 7, 11, 13, 17, 19]

In [2]: galois.primes(20)
Out[2]: [2, 3, 5, 7, 11, 13, 17, 19]
```

**galois.random_prime** *(bits: int) → int*

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 `is_prime()` to determine if \( p \) is prime.

**Parameters**

- **bits: int**
  - The number of bits in the prime \( p \).

**Returns**

A random prime in \( 2^b \leq p < 2^{b+1} \).
See also:
prev_prime, next_prime

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]:
→ 31329716741311021427450712358016288548417253537851966814450883301018013402700630078108105992667239545539301382183496328039...

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

$ openssl prime
→ 23686178792695738220699688608721459202975252407802639235893684447966742357083311612650692787787711514D68EDB7C650F1FF7135311A43255A4BE6D6EE1FDBD96F4EB32757C1B1BAF16A5933E24D45FAD6C6A814F3C8C14F30
→ (23686178792695738220699688608721459202975252407802639235893684447966742357083311612650692787787711514D68EDB7C650F1FF7135311A43255A4BE6D6EE1FDBD96F4EB32757C1B1BAF16A5933E24D45FAD6C6A814F3C8C14F30
→ is prime

Primality tests

galois.is_composite(n: int) → bool
    Determines if n is composite.

galois.is_perfect_power(n: int) → bool
    Determines if n is a perfect power \( n = c^e \) with \( e > 1 \).

galois.is_powersmooth(n: int, B: int) → bool
    Determines if the integer n is B-powersmooth.

galois.is_prime(n: int) → bool
    Determines if n is prime.

galois.is_prime_power(n: int) → bool
    Determines if n is a prime power \( n = p^k \) for prime p and \( k \geq 1 \).

galois.is_smooth(n: int, B: int) → bool
    Determines if the integer n is B-smooth.

galois.is_square_free(value: int) → bool

galois.is_square_free(value: Poly) → bool
    Determines if an integer or polynomial is square-free.

galois.is_composite(n: int) → bool
    Determines if n is composite.
**is_composite**

**n**: int

An integer.

**Returns**

*True* if the integer *n* is composite.

**See also:**

*is_prime, is_square_free, is_perfect_power*

**Examples**

In [1]: galois.is_composite(13)
Out[1]: False

In [2]: galois.is_composite(15)
Out[2]: True

**is_perfect_power**

**n**: int → bool

Determines if *n* is a perfect power *n* = *c*^*e* with *e* > 1.

**Parameters**

**n**: int

An integer.

**Returns**

*True* if the integer *n* is a perfect power.

**See also:**

*is_prime_power, is_square_free*

**Examples**

Primes are not perfect powers because their exponent is 1.

In [1]: galois.perfect_power(13)
Out[1]: (13, 1)

In [2]: galois.is_perfect_power(13)
Out[2]: False

Products of primes are not perfect powers.

In [3]: galois.perfect_power(5*7)
Out[3]: (35, 1)

In [4]: galois.is_perfect_power(5*7)
Out[4]: False

Products of prime powers where the GCD of the exponents is 1 are not perfect powers.

In [5]: galois.perfect_power(2 * 3 * 5**3)
Out[5]: (750, 1)
In [6]: galois.is_perfect_power(2 * 3 * 5**3)
Out[6]: False

Products of prime powers where the GCD of the exponents is greater than 1 are perfect powers.

In [7]: galois.perfect_power(2**2 * 3**2 * 5**4)
Out[7]: (150, 2)
In [8]: galois.is_perfect_power(2**2 * 3**2 * 5**4)
Out[8]: True

Negative integers can be perfect powers if they can be factored with an odd exponent.

In [9]: galois.perfect_power(-64)
Out[9]: (-4, 3)
In [10]: galois.is_perfect_power(-64)
Out[10]: True

Negative integers that are only factored with an even exponent are not perfect powers.

In [11]: galois.perfect_power(-100)
Out[11]: (-100, 1)
In [12]: galois.is_perfect_power(-100)
Out[12]: False

galois.is_powersmooth(n: int, B: int) → bool
Determines if the integer \(n\) is \(B\)-powersmooth.

Parameters
- \(n\): int
  An integer.
- \(B\): int
  The smoothness bound \(B \geq 2\).

Returns
- True if \(n\) is \(B\)-powersmooth.

See also:
- factors, is_smooth

Notes
An integer \(n\) with prime factorization \(n = p_1^{e_1} \ldots p_k^{e_k}\) 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.

```
In [1]: galois.is_smooth(2**4 * 3**2 * 5, 5)
Out[1]: True

In [2]: galois.is_powersmooth(2**4 * 3**2 * 5, 5)
Out[2]: False
```

galois.is_prime(n: int) → bool
Determines if $n$ is prime.

Parameters

- n: int
  An integer.

Returns

- True if the integer $n$ is prime.

See also:

is_composite, is_prime_power, is_perfect_power

Notes

This algorithm will first run Fermat’s primality test to check $n$ for compositeness, see `fermat_primality_test()`. If it determines $n$ is composite, the function will quickly return.

If Fermat’s primality test returns True, then $n$ could be prime or pseudoprime. If so, then the algorithm will run 10 rounds of Miller-Rabin’s primality test, see `miller_rabin_primality_test()`. With this many rounds, a result of True should have high probability of $n$ being a true prime, not a pseudoprime.

Examples

```
In [1]: galois.is_prime(13)
Out[1]: True

In [2]: galois.is_prime(15)
Out[2]: False
```

The algorithm is also efficient on very large $n$.

```
In [3]: galois.is_prime(1000000000000000035000061)
Out[3]: True
```

galois.is_prime_power(n: int) → bool
Determines if $n$ is a prime power $n = p^k$ for prime $p$ and $k \geq 1$.

Parameters

- n: int
  An integer.

Returns

- True if the integer $n$ is a prime power.
See also:

is_perfect_power, is_prime

Notes

There is some controversy over whether 1 is a prime power $p^0$. Since 1 is the 0-th power of all primes, it is often regarded not as a prime power. This function returns False for 1.

Examples

```python
In [1]: galois.is_prime_power(8)
Out[1]: True

In [2]: galois.is_prime_power(6)
Out[2]: False

In [3]: galois.is_prime_power(1)
Out[3]: False
```

galois.is_smooth(n: int, B: int) → bool

Determines if the integer $n$ is $B$-smooth.

Parameters

- **n**: int
  - An integer.

- **B**: int
  - The smoothness bound $B \geq 2$.

Returns

- **True** if $n$ is $B$-smooth.

See also:

factors, is_powersmooth

Notes

An integer $n$ with prime factorization $n = p_1^{e_1} \ldots p_k^{e_k}$ 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.

Examples

```python
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)
```

(continues on next page)
\texttt{Out[3]}: True

\texttt{In [4]}: \texttt{galois.is_smooth(60**2, 5)}
\texttt{Out[4]}: True

\texttt{galois.is_square_free}(value: \textit{int}) \rightarrow \textit{bool}
\texttt{galois.is_square_free}(value: \textit{Poly}) \rightarrow \textit{bool}

Determine if an integer or polynomial is square-free.

\textbf{Parameters}

- value: \textit{int}
- value: \textit{Poly}
  
  An integer \(n\) or polynomial \(f(x)\).

\textbf{Returns}

\texttt{True} if the integer or polynomial is square-free.

\textbf{See also:}

\texttt{is_prime_power, is_perfect_power}

\textbf{Notes}

\textbf{Integers}

A square-free integer \(n\) is divisible by no perfect squares. As a consequence, the prime factorization of a square-free integer \(n\) is

\[ n = \prod_{i=1}^{k} p_i^{e_i} = \prod_{i=1}^{k} p_i. \]

\textbf{Polynomials}

A square-free polynomial \(f(x)\) has no irreducible factors with multiplicity greater than one. Therefore, its canonical factorization is

\[ f(x) = \prod_{i=1}^{k} g_i(x)^{e_i} = \prod_{i=1}^{k} g_i(x). \]

This test is also available in \texttt{Poly.is_square_free()}.  

\textbf{Examples}

\textbf{Integers}

Determine if integers are square-free.
```python
In [1]: galois.is_square_free(10)
Out[1]: True

In [2]: galois.is_square_free(18)
Out[2]: False
```

**Polynomials**

Generate irreducible polynomials over GF(3).

```python
In [3]: GF = galois.GF(3)
In [4]: f1 = galois.irreducible_poly(3, 3); f1
Out[4]: Poly(x^3 + 2x + 1, GF(3))
In [5]: f2 = galois.irreducible_poly(3, 4); f2
Out[5]: Poly(x^4 + x + 2, GF(3))
```

Determine if composite polynomials are square-free over GF(3).

```python
In [6]: galois.is_square_free(f1 * f2)
Out[6]: True
In [7]: galois.is_square_free(f1**2 * f2)
Out[7]: False
```

**Specific primality tests**

galois.fermat_primality_test(n: int,...) → bool
   Determines if n is composite using Fermat’s primality test.

galois.miller_rabin_primality_test(n: int, a: int = 2,...) → bool
   Determines if n is composite using the Miller-Rabin primality test.

galois.fermat_primality_test(n: int, a: int | None = None, rounds: int = 1) → bool
   Determines if n is composite using Fermat’s primality test.

**Parameters**

- **n**: int
  - An odd integer \( n \geq 3 \).

- **a**: int | None = None
  - An integer in \( 2 \leq a \leq n - 2 \). The default is None which selects a random \( a \).

- **rounds**: int = 1
  - The number of iterations attempting to detect \( n \) as composite. Additional rounds will choose a new \( a \). The default is 1.

**Returns**

- False if \( n \) is shown to be composite. True if \( n \) is a probable prime.

See also:

- is_prime, miller_rabin_primality_test
Notes

Fermat’s theorem says that for prime \( p \) and \( 1 \leq a \leq p - 1 \), the congruence \( a^{p-1} \equiv 1 \pmod{p} \) holds. Fermat’s primality test of \( n \) computes \( a^{n-1} \mod n \) for some \( 1 \leq a \leq n - 1 \). If \( a \) is such that \( a^{p-1} \neq 1 \pmod{p} \), 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]: [False, True, True]
```

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]
```
galois

Out[10]: [True, True, True]

# But they may not satisfy \( a^{(p-1)} = 1 \pmod{p} \) for other \( a \)

In [11]: [galois.fermat_primality_test(p) for p in pseudoprimes]

Out[11]: [False, False, False]

**galois.miller_rabin_primality_test** (*n: int, a: int = 2, rounds: int = 1*) → bool

Determines if \( n \) is composite using the Miller-Rabin primality test.

**Parameters**

- **n**: int
  - An odd integer \( n \geq 3 \).
- **a**: int = 2
  - An integer in \( 2 \leq a \leq n - 2 \). The default is 2.
- **rounds**: int = 1
  - The number of iterations attempting to detect \( n \) as composite. Additional rounds will choose consecutive primes for \( a \). The default is 1.

**Returns**

- **False** if \( n \) is shown to be composite. **True** if \( n \) is probable prime.

**See also:**

*is_prime, fermat_primality_test*

**Notes**

The Miller-Rabin primality test is based on the fact that for odd \( n \) with factorization \( n = 2^s r \) for odd \( r \) and integer \( a \) such that \( \gcd(a, n) = 1 \), then either \( a^r \equiv 1 \pmod{n} \) or \( a^{2^j r} \equiv -1 \pmod{n} \) for some \( j \) in \( 0 \leq j \leq s - 1 \).

In the Miller-Rabin primality test, if \( a^r \not\equiv 1 \pmod{n} \) and \( a^{2^j r} \not\equiv -1 \pmod{n} \) for all \( j \) in \( 0 \leq j \leq s - 1 \), then \( a \) is called a **strong witness** to the compositeness of \( n \). If not, namely \( a^r \equiv 1 \pmod{n} \) or \( a^{2^j r} \equiv -1 \pmod{n} \) for any \( j \) in \( 0 \leq j \leq s - 1 \), then \( a \) is called a **strong liar** to the primality of \( n \) and \( n \) is called a **strong pseudoprime** to the base \( a \).

Since \( a = \{1, n - 1\} \) are strong liars for all composite \( n \), it is common to reduce the range of possible \( a \) to \( 2 \leq a \leq n - 2 \).

For composite odd \( n \), the probability that the Miller-Rabin test declares it a probable prime is less than \( (\frac{1}{4})^t \), where \( t \) is the number of rounds, and is often much lower.

**References**

- Section 4.2.3 from https://cacr.uwaterloo.ca/hac/about/chap4.pdf

3.18. galois
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.

```python
In [4]: strong_liars = [9, 10, 12, 16, 17, 22, 29, 38, 53, 62, 69, 74, 75, 79, 81, 82]
In [5]: witnesses = [a for a in range(2, 90) if a not in strong_liars]
# All strong liars falsely assert that 91 is prime
In [6]: [galois.miller_rabin_primality_test(91, a=a) for a in strong_liars] ==
   \[True,]*len(strong_liars)
Out[6]: True
# All other a are witnesses to the compositeness of 91
In [7]: [galois.miller_rabin_primality_test(91, a=a) for a in witnesses] == [False, 
   \[False,]*len(witnesses)
Out[7]: True
```

### 3.18.10 Configuration

**galois.get_printoptions()** → Dict[str, Any]

Returns the current print options for the package. This function is the *galois* equivalent of *numpy*. *get_printoptions()*.

**galois.printoptions(**kwargs**)** → Generator[None, None, None]

A context manager to temporarily modify the print options for the package. This function is the *galois* equivalent of *numpy*. *printoptions()*.

**galois.set_printoptions**(coeffs: 'desc' | 'asc' = 'desc')

Modifies the print options for the package. This function is the *galois* equivalent of *numpy*. *set_printoptions()*.

**galois.get_printoptions()** → Dict[str, Any]

Returns the current print options for the package. This function is the *galois* equivalent of *numpy*. *get_printoptions()*.

Returns

A dictionary of current print options.

See also:

*set_printoptions*, *printoptions*
Examples

```python
In [1]: galois.get_printoptions()
Out[1]: {'coeffs': 'desc'}
In [2]: galois.set_printoptions(coeffs="asc")
In [3]: galois.get_printoptions()
Out[3]: {'coeffs': 'asc'}
```

galois.printoptions(**kwargs) → Generator[None, None, None]
A context manager to temporarily modify the print options for the package. This function is the `galois` equivalent of `numpy.printoptions()`.

See `set_printoptions()` for the full list of available options.

Returns
A context manager for use in a `with` statement. The print options are only modified inside the `with` block.

See also:
`set_printoptions`, `get_printoptions`

Examples

By default, polynomials are displayed with descending degrees.

```python
In [1]: GF = galois.GF(3**5, display="poly")
In [2]: a = GF([109, 83])
In [3]: f = galois.Poly([3, 0, 5, 2], field=galois.GF(7))
```

Modify the print options only inside the context manager.

```python
In [4]: print(a); print(f)
[^4 + ^3 + 1, ^4 + 2]
3x^3 + 5x + 2
In [5]: with galois.printoptions(coeffs="asc"):
    ...:     print(a); print(f)
    ...:
[1 + ^3 + ^4, 2 + ^4]
2 + 5x + 3x^3
In [6]: print(a); print(f)
[^4 + ^3 + 1, ^4 + 2]
3x^3 + 5x + 2
```

galois.set_printoptions(coeffs: 'desc' | 'asc' = 'desc')
Modifies the print options for the package. This function is the `galois` equivalent of `numpy.set_printoptions()`.

Parameters
coeffs: 'desc' | 'asc' = 'desc'

The order in which to print the coefficients, either in descending degrees (default) or ascending degrees.

See also:

get_printoptions, printoptions

Examples

By default, polynomials are displayed with descending degrees.

```
In [1]: GF = galois.GF(3**5, display="poly")
In [2]: a = GF([109, 83]); a
Out[2]: GF([^4 + ^3 + 1, ^4 + 2], order=3^5)
In [3]: f = galois.Poly([3, 0, 5, 2], field=galois.GF(7)); f
Out[3]: Poly(3x^3 + 5x + 2, GF(7))
```

Modify the print options to display polynomials with ascending degrees.

```
In [4]: galois.set_printoptions(coeffs="asc")
In [5]: a
Out[5]: GF([1 + ^3 + ^4, 2 + ^4], order=3^5)
In [6]: f
Out[6]: Poly(2 + 5x + 3x^3, GF(7))
```

3.19 Versioning

The galois library uses semantic versioning. Releases are versioned major.minor.patch.

Major versions introduce API-changing features. Minor versions add features that are backwards-compatible with other releases. Patch versions make backwards-compatible bug fixes.

3.19.1 Alpha releases

Versions before 0.1.0 are alpha releases. Alpha releases are versioned 0.0.alpha. There is no API compatibility guarantee for them. They can be thought of as 0.0.alpha-major.
3.19.2 Beta releases

Versions before 1.0.0 are beta releases. Beta releases are versioned 0.beta.x and are API-compatible. They can be thought of as 0.beta-major.beta-minor.

3.20 v0.0.31

Released July 24, 2022

3.20.1 Breaking changes

- Renamed FieldArray.Elements() classmethod to FieldArray.elements class property. This naming convention is more consistent with primitive_elements, units, quadratic_residues, and quadratic_non_residues. (#373)

```python
>>> GF = galois.GF(3**2, display="poly")
>>> GF.elements
GF([ 0, 1, 2, 1, 1, 2, 2 + 1, 2 + 2], order=3^2)
```

- Renamed BCH.systematic to BCH.is_systematic. (#376)
- Renamed ReedSolomon.systematic to ReedSolomon.is_systematic. (#376)

3.20.2 Changes

- Added support for polynomial composition in Poly.__call__(). (#377)

```python
>>> GF = galois.GF(3**5)
>>> f = galois.Poly([37, 123, 0, 201], field=GF); f
Poly(37x^3 + 123x^2 + 201, GF(3^5))
>>> g = galois.Poly([55, 0, 1], field=GF); g
Poly(55x^2 + 1, GF(3^5))
>>> f(g)
Poly(77x^6 + 5x^4 + 104x^2 + 1, GF(3^5))
```

- Added FieldArray.units class property. (#373)

```python
>>> GF = galois.GF(3**2, display="poly")
>>> GF.units
GF([ 1, 2, 1, 1, 2, 2 + 1, 2 + 2], order=3^2)
```
3.20.3 Documentation

- Reworked API reference using Sphinx Immaterial’s python-apigen. (#370)
- Shortened website URLs to use directories. https://galois.readthedocs.io/en/v0.0.30/getting-started.html is converted to https://galois.readthedocs.io/en/v0.0.31/getting-started/. (#370)

3.20.4 Contributors

- Matt Hostetter (@mhostetter)

3.21 v0.0.30

Released July 12, 2022

3.21.1 Changes

- Added support for NumPy 1.22 with Numba 0.55.2. This allows users to upgrade NumPy and avoid recently-discovered vulnerabilities CVE-2021-34141, CVE-2021-41496, and CVE-2021-41495. (#366)
- Made FieldArray.repr_table() more compact. (#367)

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

<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]</td>
<td>0</td>
</tr>
<tr>
<td>x^0</td>
<td>1</td>
<td>[0, 0, 1]</td>
<td>1</td>
</tr>
<tr>
<td>x^1</td>
<td>x</td>
<td>[0, 1, 0]</td>
<td>3</td>
</tr>
<tr>
<td>x^2</td>
<td>x^2</td>
<td>[1, 0, 0]</td>
<td>9</td>
</tr>
<tr>
<td>x^3</td>
<td>x + 2</td>
<td>[0, 1, 2]</td>
<td>5</td>
</tr>
<tr>
<td>x^4</td>
<td>x^2 + 2x</td>
<td>[1, 2, 0]</td>
<td>15</td>
</tr>
<tr>
<td>x^5</td>
<td>2x^2 + x + 2</td>
<td>[2, 1, 2]</td>
<td>23</td>
</tr>
<tr>
<td>x^6</td>
<td>x^2 + x + 1</td>
<td>[1, 1, 1]</td>
<td>13</td>
</tr>
<tr>
<td>x^7</td>
<td>x^2 + 2x + 2</td>
<td>[1, 2, 2]</td>
<td>17</td>
</tr>
<tr>
<td>x^8</td>
<td>2x^2 + 2</td>
<td>[2, 0, 2]</td>
<td>20</td>
</tr>
<tr>
<td>x^9</td>
<td>x + 1</td>
<td>[0, 1, 1]</td>
<td>4</td>
</tr>
<tr>
<td>x^10</td>
<td>x^2 + x</td>
<td>[1, 1, 0]</td>
<td>12</td>
</tr>
<tr>
<td>x^11</td>
<td>x^2 + x + 2</td>
<td>[1, 1, 2]</td>
<td>14</td>
</tr>
<tr>
<td>x^12</td>
<td>x^2 + 2</td>
<td>[1, 0, 2]</td>
<td>11</td>
</tr>
<tr>
<td>x^13</td>
<td>2</td>
<td>[0, 0, 2]</td>
<td>2</td>
</tr>
<tr>
<td>x^14</td>
<td>2x</td>
<td>[0, 2, 0]</td>
<td>6</td>
</tr>
<tr>
<td>x^15</td>
<td>2x^2</td>
<td>[2, 0, 0]</td>
<td>18</td>
</tr>
<tr>
<td>x^16</td>
<td>2x + 1</td>
<td>[0, 2, 1]</td>
<td>7</td>
</tr>
<tr>
<td>x^17</td>
<td>2x^2 + x</td>
<td>[2, 1, 0]</td>
<td>21</td>
</tr>
<tr>
<td>x^18</td>
<td>x^2 + 2x + 1</td>
<td>[1, 2, 1]</td>
<td>16</td>
</tr>
<tr>
<td>x^19</td>
<td>2x^2 + 2x + 2</td>
<td>[2, 2, 2]</td>
<td>26</td>
</tr>
<tr>
<td>x^20</td>
<td>2x^2 + x + 1</td>
<td>[2, 1, 1]</td>
<td>22</td>
</tr>
<tr>
<td>x^21</td>
<td>x^2 + 1</td>
<td>[1, 0, 1]</td>
<td>10</td>
</tr>
<tr>
<td>x^22</td>
<td>2x + 2</td>
<td>[0, 2, 2]</td>
<td>8</td>
</tr>
</tbody>
</table>

(continues on next page)
x^23 2x^2 + 2x [2, 2, 0] 24
x^24 2x^2 + 2x + 1 [2, 2, 1] 25
x^25 2x^2 + 1 [2, 0, 1] 19

• Made FieldArray.arithmetic_table() more compact. (#367)

In [2]: GF = galois.GF(13)

In [3]: print(GF.arithmetic_table("*"))

x * y | 0 1 2 3 4 5 6 7 8 9 10 11 12
------|----------------------------------------------------
  | 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 | 1 1 2 3 4 5 6 7 8 9 10 11 12
 1 | 0 2 4 6 8 10 12 1 3 5 7 9 11
 2 | 0 3 6 9 12 2 5 8 11 1 4 7 10
 3 | 0 4 8 12 3 7 11 2 6 10 1 5 9
 4 | 0 5 10 2 7 12 4 9 1 6 11 3 8
 5 | 0 6 12 5 11 4 10 3 9 2 8 1 7
 6 | 0 7 1 8 2 9 3 10 4 11 5 12 6
 7 | 0 8 3 11 6 1 9 4 12 7 2 10 5
 8 | 0 9 5 1 10 6 2 11 7 3 12 8 4
 9 | 0 10 7 4 1 11 8 5 2 12 9 6 3
10 | 0 11 9 7 5 3 1 12 10 8 6 4 2
11 | 0 12 11 10 9 8 7 6 5 4 3 2 1

3.21.2 Contributors

• Iyán Méndez Veiga (@iyanmv)
• Matt Hostetter (@mhostetter)

3.22 v0.0.29

Released May 18, 2022

3.22.1 Breaking changes

• Moved galois.square_free_factorization() function into Poly.square_free_factors() method. (#362)
• Moved galois.distinct_degree_factorization() function into Poly.distinct_degree_factors() method. (#362)
• Moved galois.equal_degree_factorization() function into Poly.equal_degree_factors() method. (#362)
• Moved galois.is_irreducible() function into Poly.is_irreducible() method. This is a method, not property, to indicate it is a computationally-expensive operation. (#362)
• Moved galois.is_primitive() function into Poly.is_primitive() method. This is a method, not property, to indicate it is a computationally-expensive operation. (#362)
• Moved galois.is_monic() function into Poly.is_monic property. (#362)

### 3.22.2 Changes

• Added galois.set_printoptions() function to modify package-wide printing options. This is the equivalent of np.set_printoptions(). (#363)

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

In [2]: a = GF([109, 83]); a
Out[2]: GF([^4 + ^3 + 1, ^4 + 2], order=3^5)

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

In [4]: galois.set_printoptions(coeffs="asc")

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

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

• Added galois.get_printoptions() function to return the current package-wide printing options. This is the equivalent of np.get_printoptions(). (#363)

• Added galois.printoptions() context manager to modify printing options inside of a with statement. This is the equivalent of np.printoptions(). (#363)

• Added a separate Poly.factors() method, in addition to the polymorphic galois.factors(). (#362)

• Added a separate Poly.is_square_free() method, in addition to the polymorphic galois.is_square_free(). This is a method, not property, to indicate it is a computationally-expensive operation. (#362)

• Fixed a bug (believed to be introduced in v0.0.26) where Poly.degree occasionally returned np.int64 instead of int. This could cause overflow in certain large integer operations (e.g., computing q**m when determining if a degree-m polynomial over GF(q) is irreducible). When the integer overflowed, this created erroneous results. (#360, #361)

• Increased code coverage.

### 3.22.3 Contributors

• Matt Hostetter (@mhostetter)
3.23 v0.0.28

Released May 11, 2022

3.23.1 Changes

- Modified JIT-compiled functions to use explicit calculation or lookup tables. Previously, JIT functions only used explicit calculation routines. Now all ufuncs and functions are JIT-compiled once on first invocation, but use the current ufunc_mode to determine the arithmetic used. This provides a significant performance boost for fields which use lookup tables by default. The greatest performance improvement can be seen in GF(p^m) fields. (#354)
  
  - Polynomial multiplication is 210% faster.

```
In [2]: GF = galois.GF(7**5)
In [3]: f = galois.Poly.Random(10, seed=1, field=GF)
In [4]: g = galois.Poly.Random(5, seed=2, field=GF)

# v0.0.27
In [6]: %timeit f * g
   168 µs ± 722 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)

# v0.0.28
In [6]: %timeit f * g
   54 µs ± 574 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

- Polynomial modular exponentiation is 5,310% faster.

```
# v0.0.27
In [8]: %timeit pow(f, 123456789, g)
   5.9 ms ± 9.4 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)

# v0.0.28
In [8]: %timeit pow(f, 123456789, g)
   109 µs ± 527 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

- Matrix multiplication is 6,690% faster.

```
In [9]: A = GF.Random((100, 100), seed=1)
In [10]: B = GF.Random((100, 100), seed=2)

# v0.0.27
In [12]: %timeit A @ B
   1.1 s ± 4.76 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

# v0.0.28
In [12]: %timeit A @ B
   16.2 ms ± 50.1 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

- Simplified FieldArray subclasses’ repr() and str(). Since these classes may be displayed in error logs, a concise representation is necessary. (#350)
```python
>>> GF = galois.GF(3**5)
>>> GF
<class 'galois.GF(3^5)'>
```

- Added back `FieldArray.properties` for a detailed description of the finite field’s relevant properties. (#350)
  ```python
  >>> GF = galois.GF(3**5)
  >>> print(GF.properties)
  Galois Field:
  name: GF(3^5)
  characteristic: 3
  degree: 5
  order: 243
  irreducible_poly: x^5 + 2x + 1
  is_primitive_poly: True
  primitive_element: x
  ```

- Increased code coverage.
- Various documentation fixes.

### 3.23.2 Contributors

- Matt Hostetter (@mhostetter)

### 3.24 v0.0.27

Released April 22, 2022

### 3.24.1 Breaking changes

- Sunsetted support for Python 3.6. This was necessary to support forward references with `from __future__ import annotations` (available in Python 3.7+). That import is required to support the type aliases in the new `galois.typing` subpackage. (#339)
  ```python
  – Use `issubclass(GF, galois.FieldArray)` anywhere `isinstance(GF, galois.FieldClass)` was previously used.
  – Annotate with `Type[galois.FieldArray]` anywhere `galois.FieldClass` was previously used.
  ```

- Removed the `FieldClass` metaclass from the public API. It was previously included due to an inability of Sphinx to document class properties. In this release, we monkey patched Sphinx to document all classmethods, class properties, and instance methods in `FieldArray` itself. (#343)
3.24.2 Changes

- Added the `galois.typing` subpackage, similar to `np.typing`. It contains type hints for common coercible data types used throughout the library, including `ElementLike`, `ArrayLike`, and `PolyLike`. With these type hints, the annotations are simpler and more clear. (#339)

- Modified functions to accept coercible data types wherever possible. For example, functions now accept `PolyLike` objects instead of strictly `Poly` instances. (#339)

- Added `Array` which is an abstract base class of `FieldArray` (and `RingArray` in a future release). (#336)

- Added support for the DFT over any finite field using `np.fft.fft()` and `np.fft.ifft()`. (#335)

```python
>>> x
GF([127, 191, 69, 35, 221, 242, 193, 108, 72, 102, 80, 163, 13, 74, 218, 159, 207, 12, 159, 129, 92, 71], order=3^5)
```

```python
>>> X = np.fft.fft(x); X
GF([ 16, 17, 20, 137, 58, 166, 178, 52, 19, 109, 115, 93, 99, 214, 187, 235, 195, 96, 232, 45, 241, 24], order=3^5)
```

```python
>>> np.fft.ifft(X)
GF([127, 191, 69, 35, 221, 242, 193, 108, 72, 102, 80, 163, 13, 74, 218, 159, 207, 12, 159, 129, 92, 71], order=3^5)
```

- Implemented the Cooley-Tukey radix-2 $O(N\log(N))$ algorithm for the NTT and JIT compiled it. (#333)

```python
In [2]: x = list(range(1, 1024 + 1))
# v0.0.26
In [4]: %timeit X = galois.ntt(x)
5.2 ms ± 121 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
# v0.0.27
In [4]: %timeit X = galois.ntt(x)
695 µs ± 4.56 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

- Added the `FieldArray.primitive_root_of_unity()` classmethod. (#333)

```python
>>> GF = galois.GF(3**5)
>>> GF.primitive_root_of_unity(22)
GF(39, order=3^5)
```

- Added the `FieldArray.primitive_roots_of_unity()` classmethod. (#333)

```python
>>> GF = galois.GF(3**5)
>>> GF.primitive_roots_of_unity(22)
GF([ 14, 39, 44, 59, 109, 114, 136, 200, 206, 226], order=3^5)
```

- Made 0-th degree coefficients more differentiated when using the polynomial element representation. (#328)

```python
# v0.0.26
>>> print(f)
(^2 + + 1)x^4 + (^3)x + ^3 + 2^2 + 2 + 2
# v0.0.27
>>> print(f)
(^2 + + 1)x^4 + (^3)x + (^3 + 2^2 + 2 + 2)
```

- Restructured code base for clarity. (#336)
• Fixed display of overloaded functions in API reference. (#337)
• Fixed broken “References” sections in API reference. (#281)
• Fixed other small bugs.

3.24.3 Contributors

• Matt Hostetter (@mhostetter)

3.25 v0.0.26

Released March 30, 2022

3.25.1 Breaking changes

• Removed the Poly.copy() method as it was unnecessary. Polynomial objects are immutable. Use g = f wherever g = f.copy() was previously used. (#320)
• Disabled true division f / g on polynomials since true division was not actually being performed. Use floor division f // g moving forward. (#312)
• Refactored irreducible_polys() to return an iterator rather than list. Use list(irreducible_polys(...)) to obtain the previous output. (#325)
• Refactored primitive_polys() to return an iterator rather than list. Use list(primitive_polys(...)) to obtain the previous output. (#325)
• Refactored primitive_root() and primitive_roots(). (#325)
  – Added method keyword argument and removed reverse from primitive_root(). Use primitive_root(..., method="max") where primitive_root(..., reverse=True) was previously used.
  – Refactored primitive_roots() to now return an iterator rather than list. Use list(primitive_roots(...)) to obtain the previous output.
• Refactored primitive_element() and primitive_elements(). (#325)
  – Added method keyword argument to primitive_element().
  – Removed start, stop, and reverse arguments from both functions.
• Search functions now raise RuntimeError instead of returning None when failing to find an answer. This applies to primitive_root(), pollard_p1(), and pollard_rho(). (#312)

3.25.2 Changes

• The galois.Poly class no longer returns subclasses BinaryPoly, DensePoly, and SparsePoly. Instead, only Poly classes are returned. The classes otherwise operate the same. (#320)
• Made Galois field array creation much more efficient by avoiding redundant element verification. (#317)
  – Scalar creation is 625% faster.
In [2]: GF = galois.GF(3**5)
# v0.0.25
In [3]: %timeit GF(10)
21.2 µs ± 181 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
# v0.0.26
In [3]: %timeit GF(10)
2.88 µs ± 8.03 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)

- Nested iterable array creation is 150% faster.

# v0.0.25
In [4]: %timeit GF([[10, 20], [30, 40]])
53.6 µs ± 436 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
# v0.0.26
In [4]: %timeit GF([[10, 20], [30, 40]])
20.9 µs ± 11.2 µs per loop (mean ± std. dev. of 7 runs, 10,000 loops each)

- Nested iterable (with strings) array creation is 25% faster.

# v0.0.25
In [5]: %timeit GF([[10, "2x^2 + 2"], ["x^3 + x", 40]])
67.9 µs ± 910 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
# v0.0.26
In [5]: %timeit GF([[10, "2x^2 + 2"], ["x^3 + x", 40]])
54.7 µs ± 16.3 µs per loop (mean ± std. dev. of 7 runs, 10,000 loops each)

- Made array arithmetic 35% faster by avoiding unnecessary element verification of outputs. (#309)

In [2]: GF = galois.GF(3**5)
In [3]: x = GF.Random((10_000), seed=1)
In [4]: y = GF.Random((10_000), seed=2)
# v0.0.25
In [6]: %timeit x * y
39.4 µs ± 237 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
# v0.0.26
In [6]: %timeit x * y
28.8 µs ± 172 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)

- Made polynomial arithmetic over binary fields 10,900% faster by making polynomial creation from integers much more efficient. (#320)

In [5]: f
Out[5]: Poly(x^10 + x^9 + x^6 + x^5 + x^3 + x, GF(2))
In [6]: g

(continues on next page)
• JIT-compiled polynomial modular exponentiation. (#306)
  – Binary fields are 14,225% faster.
    
    ```python
    In [5]: f
    Out[5]: Poly(x^10 + x^9 + x^6 + x^5 + x^3 + x, GF(2))
    
    In [6]: g
    Out[6]: Poly(x^5 + x^2, GF(2))
    
    # v0.0.25
    In [7]: %timeit pow(f, 123456789, g)
    19.2 ms ± 206 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
    
    # v0.0.26
    In [7]: %timeit pow(f, 123456789, g)
    134 µs ± 2.21 µs per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
    
    – Other fields are 325% faster.
    
    In [6]: f
    Out[6]: Poly(242x^10 + 216x^9 + 32x^8 + 114x^7 + 230x^6 + 179x^5 + 5x^4 + 124x^3 + 3 + 96x^2 + 159x + 77, GF(3^5))
    
    In [7]: g
    Out[7]: Poly(183x^5 + 83x^4 + 101x^3 + 203x^2 + 68x + 2, GF(3^5))
    
    # v0.0.25
    In [8]: %timeit pow(f, 123456789, g)
    17.6 ms ± 61.7 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
    
    # v0.0.26
    In [8]: %timeit pow(f, 123456789, g)
    4.19 ms ± 11.9 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
    
    • Made irreducible and primitive polynomial search routines faster. (#306, #309, #317, #320)
  – Binary fields are 1,950% faster.
    
    # v0.0.25
    In [2]: %time f = galois.primitive_poly(2, 14)
    CPU times: user 296 ms, sys: 70.9 ms, total: 367 ms
    Wall time: 313 ms
    
    (continues on next page)
# v0.0.26

In [2]: %time f = galois.primitive_poly(2, 14)
CPU times: user 14.7 ms, sys: 5.53 ms, total: 20.2 ms
Wall time: 15.3 ms

- Other fields are 400% faster.

# v0.0.25

In [5]: %time f = galois.primitive_poly(7, 10)
CPU times: user 2.22 s, sys: 0 ns, total: 2.22 s
Wall time: 2.21 s

In [4]: %time f = galois.primitive_poly(7, 10)
CPU times: user 442 ms, sys: 0 ns, total: 442 ms
Wall time: 439 ms

- Made FieldArray.Vector() 100% faster and FieldArray.vector() 25% faster by making better use of divmod() when converting between integer and vector representations. (#322)

3.25.3 Contributors

- Matt Hostetter (@mhostetter)

3.26 v0.0.25

Released March 21, 2022

3.26.1 Breaking changes

- Separated LFSR into FLFSR/GLFSR and fixed/redefined terms (feedback poly, characteristic poly, state). (#285)
- Removed galois.pow() and replaced it with the built-in pow(). (#300)

```python
>>> f = galois.Poly([6, 3, 0, 1], field=galois.GF(7))
>>> g = galois.Poly([5, 0, 3], field=galois.GF(7))
>>> pow(f, 123456789, g)
Poly(6x + 2, GF(7))
```

- Removed FieldClass.properties and replaced with FieldClass.__str__. (#289)

```python
>>> GF = galois.GF(3**5)
>>> print(GF)
Galois Field:
  name: GF(3^5)
  characteristic: 3
  degree: 5
  order: 243
  irreducible_poly: x^5 + 2x + 1
```

(continues on next page)
Differentiated `repr()` and `str()` for *Galois field arrays*, like NumPy. `repr()` displays the finite field’s order, but `str()` does not.

```python
>>> GF = galois.GF(31, display="power")
>>> x = GF([1, 23, 0, 15])
>>> x
GF([1, ^27, 0, ^21], order=31)
>>> print(x)
[ 1, ^27, 0, ^21]
```

Renamed `Poly.String()` to `Poly.Str()`. Removed `Poly.string` and replaced it with `Poly.__str__`. (#300)

```python
>>> f = galois.Poly.Str("x^3 + x + 1"); f
Poly(x^3 + x + 1, GF(2))
>>> str(f)
'x^3 + x + 1'
```

Renamed `Poly.Integer()` to `Poly.Int()`. Removed `Poly.integer` and replaced it with `Poly.__int__`. (#300)

```python
>>> f = galois.Poly.Int(11); f
Poly(x^3 + x + 1, GF(2))
>>> int(f)
11
```

### 3.26.2 Changes

- Fixed bug in Fibonacci/Galois LFSRs where feedback polynomial wasn’t interpreted correctly for fields with characteristic greater than 2. (#299)
- Utilized memoization for expensive search routines (`irreducible_poly()` and `primitive_poly()`) to speed-up subsequent calls. (#295)

```python
In [2]: %time galois.primitive_poly(7, 4)
CPU times: user 675 ms, sys: 6.24 ms, total: 682 ms
Wall time: 741 ms
Out[2]: Poly(x^4 + x^2 + 3x + 5, GF(7))

In [3]: %time galois.primitive_poly(7, 4)
CPU times: user 30 μs, sys: 0 ns, total: 30 μs
Wall time: 31.7 μs
Out[3]: Poly(x^4 + x^2 + 3x + 5, GF(7))
```

- Added support for `bin()`, `oct()`, and `hex()` on Poly objects. (#300)

```python
>>> f = galois.Poly.Int(11); f
Poly(x^3 + x + 1, GF(2))
>>> bin(f)
```
'0b1011'
>>> oct(f)
'0o13'
>>> hex(f)
'0xb'

• Made *Galois field arrays* display with fixed-width elements, like NumPy. (#270)
• Achieved speed-up of *repr()* and *str()* on *Galois field arrays* of at least 25x. Achieved a much greater speed-up for large arrays, since now elements converted to ... are no longer needlessly converted to their string representation. (#270)
• Overhauled documentation and website. Type hints are now displayed in the API reference. (#263)
• Various bug fixes.

### 3.26.3 Contributors

• Matt Hostetter (@mhostetter)

### 3.27 v0.0.24

*Released February 12, 2022*

#### 3.27.1 Breaking changes

• Move `galois.minimal_poly()` functionality into `FieldArray.minimal_poly()`.
• Refactor `FieldArray.lup_decompose()` into `FieldArray.plu_decompose()`.
• Raise `ValueError` instead of returning `None` for `prev_prime(2)`.
• Return `(n, 1)` from `perfect_power(n)` if `n` is not a perfect power rather than returning `None`.

#### 3.27.2 Changes

• Compute a finite field element’s square root (if it exists) with `np.sqrt()`.
• Test if finite field elements have a square root with `FieldArray.is_quadratic_residue()`.
• List which finite field elements are/aren’t quadratic residues (have a square root) with `FieldClass.quadratic_residues` and `FieldClass.quadratic_non_residues`.
• Compute standard vector spaces with `FieldArray.row_space()`, `FieldArray.column_space()`, `FieldArray.left_null_space()`, and `FieldArray.null_space()`.
• Compute a finite field element’s additive and multiplicative orders with `FieldArray.additive_order()` and `FieldArray.multiplicative_order()`.
• Evaluate polynomials at square matrix inputs using `f(X, elementwise=False)`.
• Compute the characteristic polynomial of a single element or square matrix with `FieldArray.characteristic_poly()`.
• Compute the minimal polynomial of a single element with `FieldArray.minimal_poly()`.
• Compute a Lagrange interpolating polynomial with lagrange_poly(x, y).
• Support non-square matrix inputs to FieldArray.lu_decompose() and FieldArray.plu_decompose().
• Support polynomial scalar multiplication. Now p * 3 is valid syntax and represents p + p + p.
• Support polynomial comparison with integers and field scalars. Now galois.Poly([0]) == 0 and galois.Poly([0]) == GF(0) return True rather than raising TypeError.
• Support testing 0-degree polynomials for irreducibility and primitivity.
• Extend crt() to work over non co-prime moduli.
• Extend prev_prime() and next_prime() to work over arbitrarily-large inputs.
• Allow negative integer inputs to primes(), is_prime(), is_composite(), is_prime_power(), is_perfect_power(), is_square_free(), is_smooth(), and is_powersmooth().
• Fix various type hinting errors.
• Various other bug fixes.

3.27.3 Contributors

• Iyán Méndez Veiga (@iyanmv)
• Matt Hostetter (@mhostetter)

3.28 v0.0.23

Released January 14, 2022

3.28.1 Changes

• Add support for Python 3.10.
• Add support for NumPy 1.21.
• Add support for Numba 0.55.
• Add type hints to library API.
• Add FieldArray.characteristic_poly() method to return the characteristic polynomial of a square matrix.
• Add Poly.coefficients() method to return the coefficient array with fixed size and order.
• Fix bug in Poly.Degrees() when duplicate degrees were present.
• Fix bug in Reed-Solomon decode when c != 1.
• Various other bug fixes.
3.28.2 Contributors

• Matt Hostetter (@mhostetter)

3.29 v0.0.22

Released December 3, 2021

3.29.1 Breaking changes

• Random integer generation is handled using new style random generators. Now each .Random() call will generate a new seed rather than using the NumPy “global” seed used with np.random.randint().

• Add a seed=None keyword argument to FieldArray.Random() and Poly.Random(). A reproducible script can be constructed like this:

```python
rng = np.random.default_rng(123456789)
x = GF.Random(10, seed=rng)
y = GF.Random(10, seed=rng)
poly = galois.Poly.Random(5, seed=rng, field=GF)
```

3.29.2 Changes

• Official support for Python 3.9.

• Major performance improvements to “large” finite fields (those with dtype=np.object_).

• Minor performance improvements to all finite fields.

• Add the Number Theoretic Transform (NTT) in ntt() and intt().

• Add the trace of finite field elements in FieldArray.field_trace().

• Add the norm of finite field elements in FieldArray.field_norm().

• Support len() on Poly objects, which returns the length of the coefficient array (polynomial order + 1).

• Support x.dot(y) syntax for the expression np.dot(x, y).

• Minimum NumPy version bumped to 1.18.4 for new style random usage.

• Various bug fixes.

3.29.3 Contributors

• Iyán Méndez Veiga (@iyanmv)

• Matt Hostetter (@mhostetter)
3.30 v0.0.21

Released September 2, 2021

3.30.1 Changes

- Fix docstrings and code completion for Python classes that weren’t rendering correctly in an IDE.
- Various documentation improvements.

3.30.2 Contributors

- Matt Hostetter (@mhostetter)

3.31 v0.0.20

Released August 24, 2021

3.31.1 Breaking changes

- Move poly_gcd() functionality into gcd().
- Move poly_egcd() functionality into egcd().
- Move poly_factors() functionality into factors().

3.31.2 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

3.31.3 Contributors

- Matt Hostetter (@mhostetter)
3.32 v0.0.19

Released August 9, 2021

3.32.1 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.

3.32.2 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.

3.32.3 Contributors

- Matt Hostetter (@mhostetter)

3.33 v0.0.18

Released July 6, 2021
3.33.1 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"].

3.33.2 Changes

- Clean up `galois` API and `dir()` so only public classes and functions are displayed.
- Speed-up `galois.is_primitive()` test and search for primitive polynomials in `galois.primitive_poly()`.
- Speed-up `galois.is_smooth()`.
- Add Reed-Solomon codes in `galois.ReedSolomon`.
- Add shortened BCH and Reed-Solomon codes.
- Add error detection for BCH and Reed-Solomon with the `detect()` method.
- Add general cyclic linear block code functions.
- Add Matlab default primitive polynomial with `galois.matlab_primitive_poly()`.
- Add number theoretic functions:
  - Add `galois.legendre_symbol()`, `galois.jacobi_symbol()`, `galois.kronecker_symbol()`.
  - Add `galois.divisors()`, `galois.divisor_sigma()`.
  - Add `galois.is_composite()`, `galois.is_prime_power()`, `galois.is_perfect_power()`, `galois.is_square_free()`, `galois.is_powersmooth()`.
  - Add `galois.are_coprime()`.
- Clean up integer factorization algorithms and add some to public API:
  - Add `galois.perfect_power()`, `galois.trial_division()`, `galois.pollard_p1()`, `galois.pollard_rho()`.
- Clean up API reference structure and hierarchy.
- Fix minor bugs in BCH codes.
3.33.3 Contributors

- Matt Hostetter (@mhostetter)

3.34 v0.0.17

Released June 15, 2021

3.34.1 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.

3.34.2 Changes

- Speed-up Galois field class creation time.
- Speed-up JIT compilation time by caching functions.
- Speed-up Poly.roots() by JIT compiling it.
- Add BCH codes with galois.BCH.
- Add ability to generate irreducible polynomials with irreducible_poly() and irreducible_polys().
- Add ability to generate primitive polynomials with primitive_poly() and primitive_polys().
- Add computation of the minimal polynomial of an element of an extension field with minimal_poly().
- Add display of arithmetic tables with FieldClass.arithmetic_table().
- Add display of field element representation table with FieldClass.repr_table().
- Add Berlekamp-Massey algorithm in berlekamp_massey().
- Enable ipython tab-completion of Galois field classes.
- Cleanup API reference page.
- Add introduction to Galois fields tutorials.
- Fix bug in is_primitive() where some reducible polynomials were marked irreducible.
- Fix bug in integer<->polynomial conversions for large binary polynomials.
- Fix bug in "power" display mode of 0.
- Other minor bug fixes.
3.34.3 Contributors

- Dominik Wernberger (@Werni2A)
- Matt Hostetter (@mhostetter)

3.35 v0.0.16

Released May 19, 2021

3.35.1 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.

3.35.2 Contributors

- Matt Hostetter (@mhostetter)

3.36 v0.0.15

Released May 12, 2021

3.36.1 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().
3.36.2 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.

```python
>>> 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.

```python
>>> p = galois.Poly(1)
Poly(1, GF(2))
```
- Add the four Oakley fields from RFC 2409.
- Speed-up unit tests.
- Restructure API reference.

3.36.3 Contributors

- Matt Hostetter (@mhostetter)

3.37 v0.0.14

Released May 7, 2021

3.37.1 Breaking changes

- Rename `GFArray.Eye()` to `GFArray.Identity()`.
- Rename `chinese_remainder_theorem()` to `crt()`.
3.37.2 Changes

- Lots of performance improvements.
- Additional linear algebra support.
- Various bug fixes.

3.37.3 Contributors

- Baalateja Kataru (@BK-Modding)
- Matt Hostetter (@mhostetter)

3.38 Index
g

galois, 95
INDEX

G
galois
    module, 95

M
module
galois, 95