Galwa Field

Version 1.0.0

By Ido Nahum and Assaf Yossef

Table of Contents

Welcome to Galwa Fields's documentation!

Getting started and examples

Introduction

In this tutorial we will go step by step through the process of extending a prime field.

This tutorial will give a mathematical overview and examples of how to extend a prime field using our library which we implement - “Galwa”.

We assume that the reader has a basic understanding in abstract algebra - groups, rings, fields, order, etc..

Definitions

What is a prime field?

A prime field is a field that contains a prime number of elements.

We denote a prime field as F_p where p is a prime number.

For example F_5 is a prime field with 5 elements.

$F_5 = \{ 0, 1, 2, 3, 4 \}$

What is an extension of a field?

Given a field F and a fixed polynomial $f(x)$ with coefficients in F of degree n .

We define the ideal generated by $f(x)$:

$(f(x)) = \{f(x)g(x) | g(x) \in F[x]\}$

Then the extension of F by $f(x)$ is the set:

$F[x]/(f(x)) = \{a(x) + (f(x)) | a(x) \in F[x]\}$

In other words, the extension of F by $f(x)$ is the set of all polynomials of degree less than n with coefficients in F .

This is an extension of F with dimension n . The extension will have $|F|^n$ elements.

In this tutorial we will extend a prime field denoted as $k = F_p$ by a fixed polynomial $f(x) \in k[x]$ of degree n using python.

Lets denote the extension as:

$$l = F_{p^n} = F_p[x]/(f(x)) = \{a_0 + a_1x + a_2x^2 + ... + a_{n-1}x^{n-1} | f(x) = 0, a_i \in F_p\}$$

And lets denote the multiplicative group of l as:

$$l^x = l - \{0\}$$

The cardinality in this case will be p^n . for example F_3 extended by $f(x) = x^2 + 1$ will have $3^2 = 9$ elements.

Primitive element

We define an extension k/F as simple if there exists an element α in k such that $k = F(\alpha)$.

In other words, every element in k can be written as a polynomial in α with coefficients in F .This element is called a primitive element of the extension.

For our extension field $l = F_{p^n}$ is simple since it is finite we can assure that a primitive element exists.

So we can define all elements in multiplicative group l^x with the following basis $\{1, \alpha, \alpha^2, ..., \alpha^{n-1}\}$

Embedding l^x in $GL_n(F_p)$

We want to calculate each element easily or doing some arithmetic between elements without using Euclidean division for polynomials.

To do so we will embedd each element $a \in l^x$ in a matrix $A \in GL_n(F_p)$.

Then any operation between elements will be done by matrix multiplication, inversion.

To embedd the element we will use our basis and define a linear transformation ϕ_a from l^x to l^x , by $\phi_a(k) = ak$ for all $k \in l^x$.

The final matrix columns will be define as the transformation on each basis element.

$$A = \begin{pmatrix} | & | & | & \dots & | \\ \phi_{\mathbf{a}}(\mathbf{1}) & \phi_{\mathbf{a}}(\mathbf{x}) & \phi_{\mathbf{a}}(\mathbf{x^2}) & \dots & \phi_{\mathbf{a}}(\mathbf{x^{n-1}}) \\ | & | & | & \dots & | \end{pmatrix}$$

Where $\phi_a(i) = ai$ for all $i \in B$.

$$B = \{1, x, ..., x^{n-1}\} = \left\{ \begin{bmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \\ 0 \\ \vdots \\ 0 \\ 0 \end{bmatrix}, \ldots, \begin{bmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ 0 \\ 1 \end{bmatrix} \right\}$$

Code examples

Lets move to the code.

First, we need to install the library, from the base directory of the project run the following command:

```
>>> pip install .
```

We define three classes:

1. PrimeFieldElement - a class that represents an element in a prime field F_p

2. FiniteField - a class that represents an extended field F_{p^n}

3. FiniteFieldElement - a class that represents an element in an extension field F_{p^n}

And some algorithms:

1. Extended Euclidean algorithm - to find the inverse of an element in a prime field.

2. Baby Step Giant Step algorithm - to find the discrete logarithm in a finite field.

PrimeFieldElement

We implemented the class PrimeFieldElement that represents an element in a prime field F_p.

The class gets as an input an integer a and the prime number p.

Example:

```
>>> from galwa import PrimeFieldElement
>>> a = PrimeFieldElement(3, 5)
>>> a
PrimeFieldElement(value= 3,prime= 5
```

```
>>> print(a)
3 mod 5
```

Lets define another element:

```
>>> b = PrimeFieldElement(4, 5)
>>> b
4
```

We can now perform some arithmetic operations:

```
>>> a + b
PrimeFieldElement(value= 2,prime= 5)
>>> a - b
PrimeFieldElement(value= 4,prime= 5)
>>> a * b
PrimeFieldElement(value= 2,prime= 5)
>>> a / b
PrimeFieldElement(value= 2,prime= 5)
>>> a ** 2
PrimeFieldElement(value= 4,prime= 5)
>>> a**-1
PrimeFieldElement(value= 2,prime= 5)
>>> a.inverse
PrimeFieldElement(value= 2,prime= 5)
>>> a == b
False
```

As seen above, we also implement the inverse property which uses the extended euclidean algorithm to find the inverse of the element.

The inverse can also be calculated by a^{-1} .

FiniteField

We implemented the class FiniteField that represents an extension of a prime field $l = F_{p^n}$.

The class gets as an input the prime number p and the irreducible polynomial $f(x)$.

Example:

```
>>> from galwa import FiniteField
>>> import numpy as np
>>> p = 2
>>> f = np.array([1, 1, 1])  # x^2 + x + 1
```

```
>>> F = FiniteField(p, f)
>>> F
FiniteField(p=2, f(x)= 1 + x + x², p=2)
```

The elements created automaticly upon initialization:

```
>>> F.elements
[FiniteFieldElement(0, f(x)= 1 + x + x², p=2), FiniteFieldElement(1,
f(x)= 1 + x + x², p=2), FiniteFieldElement(x, f(x)= 1 + x + x², p=2),
FiniteFieldElement(1 + x, f(x)= 1 + x + x², p=2)]
```

We can change the representation method for printing, representation can be “polynomial”, “vector”, or “matrix”

The default value is “polynomial”:

```
>>> F.representation
'polynomial'
>>> print(F)
FiniteField(p= 2, f(x)= 1 + x + x²)
```

We can change the representation, all elements will be printed in the new representation:

Changing to “vector”:

```
>>> F.elements_as_vectors()
>>> F.elements
[FiniteFieldElement([0 0], f(x) = [1 1 1] p=2), FiniteFieldElement([1
0], f(x) = [1 1 1] p=2), FiniteFieldElement([0 1], f(x) = [1 1 1]
p=2), FiniteFieldElement([1 1], f(x) = [1 1 1] p=2)]
>>> print(F)
FiniteField(p= 2, f(x)= [1 1 1])
```

Changing to “matrix”:

```
>>> F.elements_as_matrix()
>>> F.elements
[FiniteFieldElement(None, f(x) = [1 1 1] p=2),
 FiniteFieldElement([[1 0][0 1]], f(x) = [1 1 1] p=2),
 FiniteFieldElement([[0 1][1 1]], f(x) = [1 1 1] p=2),
 FiniteFieldElement([[1 1][1 0]], f(x) = [1 1 1] p=2)]
>>> print(F) # f(x) stays vector in matrix as well.
FiniteField(p= 2, f(x)= [1 1 1])
```

Back to polynomial:

```
>>> F.elements_as_polynomials()
>>> F.elements
[FiniteFieldElement(0, f(x)= 1 + x + x², p=2), FiniteFieldElement(1,
f(x)= 1 + x + x², p=2), FiniteFieldElement(x, f(x)= 1 + x + x², p=2),
FiniteFieldElement(1 + x, f(x)= 1 + x + x², p=2)]
>>> print(F)
FiniteField(p= 2, f(x)= 1 + x + x²)
```

Some other properties and methods.

1. To get the order of the field:

```
>>> F.order # p=2 f(x) is degree 2, so 2^2 = 4
4
```

The order includes the zero element.

2. Getting the generators of the multiplicative group of the field:

We know that generator element is an element whose powers generate all the elements in the field.

$l^x = \{a^i | 0 <= i < |l^x|\}$

So to get all generators in the field

```
>>> F.generators
[FiniteFieldElement(x, f(x)= 1 + x + x², p=2),
FiniteFieldElement(1 + x, f(x)= 1 + x + x², p=2)]
```

What happens in the background is that we check the order of each element in the field, if the order is equal to the order of the multiplicative group then it is a generator.

3. Getting a specific element:

You can give a vector represents an element in the field, and the function will return the element.

```
>>> F.get_element(np.array[1, 0])
FiniteFieldElement(1, f(x)= 1 + x + x², p=2)
```

FiniteFieldElement

We implemented the class FiniteFieldElement that represents an element in the extension field $l = F_{p^n}$.

The class gets as an input a numpy array that represents the element and the field object that the element belongs to.

```
>>> from galwa import FiniteField, FiniteFieldElement
>>> import numpy as np
>>> f = np.array([1, 1, 0, 1])
>>> p = 2
>>> field = FiniteField(p, f)
>>> a = FiniteFieldElement(np.array([1, 0, 1]), field)
>>> a
FiniteFieldElement(1 + x², f(x)= 1 + x + x³, p=2)
```

The default representation is polynomial, but we can change it to vector or matrix:

```
>>> a.representation
'polynomial'
>>> a
FiniteFieldElement(1 + x², f(x)= 1 + x + x³, p=2)
>>> print(a)
1 + x²
```

Changing to vector:

```
>>> a.as_vector()
>>> a
FiniteFieldElement([1 0 1], f(x) = [1 1 0 1] p=2)
>>> print(a)
[1 0 1]
```

Changing to matrix:

```
>>> a.as_matrix()
>>> a
FiniteFieldElement([[1 0 1]
                    [1 0 0]
                    [0 1 0]], f(x) = [1 1 0 1] p=2)
>>> print(a)
[[1 0 1]
```

```
[1 0 0]
[0 1 0]]
```

Back to polynomial:

```
>>> a.as_polynomial()
>>> a
FiniteFieldElement(1 + x², f(x)= 1 + x + x³, p=2)
>>> print(a)
1 + x²
```

Arithmetic operations:

We can make some arithmetic operations between elements:

```
>>> from galwa import FiniteField, FiniteFieldElement
>>> import numpy as np
>>> f = np.array([1, 1, 0, 1])
>>> p = 2
>>> field = FiniteField(p, f)
>>> a = FiniteFieldElement(np.array([1, 0, 1]), field)
>>> b = FiniteFieldElement(np.array([1, 1, 0]), field)
>>> a + b
FiniteFieldElement(x + x², f(x)= 1 + x + x³, p=2)
>>> a - b
FiniteFieldElement(x + x², f(x)= 1 + x + x³, p=2)
>>> a * b
FiniteFieldElement(x², f(x)= 1 + x + x³, p=2)
>>> a / b
FiniteFieldElement(1 + x, f(x)= 1 + x + x³, p=2)
>>> a ** 2
FiniteFieldElement(1 + x + x², f(x)= 1 + x + x³, p=2)
>>> a**-1
FiniteFieldElement(x, f(x)= 1 + x + x³, p=2)
```

Other methods and properties:

1. Getting the multiplicative order of the element:

```
>>> a.multiplicative_order()
7
```

From lagrange theorem we know that for H as subgroup of G then the order of any element in G divides the order of G .

That is true for all $g \in G, | < g > | | |G|$

In our case the multiplicative group of $F_p^x = F_p - \{0\}$ is a subgroup of the multiplicative group l^x .

So for all $a \in l^x, O(a)|O(l^x)$.

So first, we calculate all divisors of the order of the multiplicative group of the field, the complexity of this operation is $O(\sqrt{n})$ where n is the order of the multiplicative group.

The divisors array will be sorted, we will start from the smallest and calculate a^d for each divisor d and check if the result is the identity element.

Calculating a^d can be done using exponentiation by squaring algorithm, the complexity of this operation is $O(\log(d))$.

So the complexity in the best case will be $O(\log(d))$ and in the worst case $O(k\log(d))$ where k is the number of divisors.

2. Checking if the element is a generator:

```
>>> a.is_generator()
True
```

3. Checking if the element is the identity element of l^x :

```
>>> a.is_identity_of_multiplication()
False
```

4. Getting the order of the element:

```
>>> a.order
7
```

5. Calculate the embedding matrix of the element in $GL_n(F_p)$:

```
>>> a.embed_in_gln()
array([[1, 1, 0],
       [0, 0, 1],
       [1, 0, 0]])
```

Extended Euclidean algorithm

We implemented the extended Euclidean algorithm to find the inverse of an element in a prime field.

We know that if the greatest common divisor of two numbers is 1, then they are coprime and the inverse exists.

We can get the inverse of an element using the extended Euclidean algorithm, and Bozout's identity.

$ax + by = gcd(a, b)$.

Example:

```
>>> from galwa.utils import xgcd
>>> d, x, y = xgcd(3, 5)
>>> d
1
>>> x
2
>>> y
```

Baby-step giant-step algorithm

We implemented the baby-step giant-step algorithm to find the discrete logarithm in a finite field.

Given a generator g and an element h in the field, we want to find the exponent x such that

$g^x = h$.

The above expression can be expressed as $g^{im+j} = h$.

Where m is the giant step and j is the baby step.

Taking $-im$ from both sides we get:

$g^j = h(g^{-m})^i$.

Steps:

1. We start from the group order, the group must be cyclic, lets denote the order as n .

2. We set the giant step to be the ceiling of the square root of the group order, $m = ceil(\sqrt{n})$.

3. We calculate the baby step table $\{g^j : j, 0 <= j < m\}$.

4. Now for the giant step, we start by defining g^{-m} and calculate $h(g^{-m})^i$ for all $0 <= i < m$.

5. If the result is in the baby step table, then we found the exponent creates it $x = im + j$.

Code Example:

```
>>> from galwa import FiniteFieldElement, FiniteField
>>> from galwa.utils import bsgs
>>> import numpy as np
>>> f = np.array([2, 0, 0, 2, 1])
>>> p = 3
>>> F = FiniteField(p, f)
>>> g = FiniteFieldElement(np.array([1, 1, 0, 0]), F)
>>> h = g ** 10
>>> h
FiniteFieldElement(2, f(x)= 2 + 2·x³ + x⁴, p=3)
>>> order = F.order - 1
>>> bsgs(g, h, order)
10
```

Tests for the project

We created several tests for our project.

We split the tests into two:

The first tests are for the 'PrimeFieldElement' class, and it should test all the functionality of this class according to what we had to implement in this project.

The second tests are for the rest of the project and mostly for the 'FiniteField' and 'FiniteFieldElement' classes.

In order to implement this tests and to check the correctness of our results we used 'galois' library, which is a library in python that can do all the arithmetic in prime fields and extension fields.

In order to run our tests make sure first that you installed our project as package

```
>>> pip install .
```

Now, you should have all the dependencies in order to run our tests.

To run our tests open terminal inside 'tests' folder and type the following command:

```
>>> python RunTests.py
```

Prime Field Element tests

class tests.PrimeFieldTests. PrimeFieldTests (*p*)

static test_divide_members_different_fields ( )

Test the division of members in different fields, which should raise a ValueError

Raises :

AssertionError – If the division of the members doesnt throw a ValueError

test_divide_members_in_field ( )

Test the division of members in the field

Raises :

AssertionError – If the division of the members is not the expected one.

test_find_legal_inverse ( )

Test the inverse of members in the field

Raises :

AssertionError – If the inverse of the members is not the expected one.

test_illegal_inverse ( )

Test the inverse of zero, which should be None

Raises :

AssertionError – If the inverse of zero is not None

static test_mul_members_different_fields ( )

Test the multiplication of members in different fields, which should raise a ValueError

Raises :

AssertionError – If the multiplication of the members doesnt throw a ValueError

test_mul_members_in_field ( )

Test the multiplication of members in the field

Raises :

AssertionError – If the multiplication of the members is not the expected one.

static test_sub_members_different_fields ( )

Test the subtraction of members in different fields, which should raise a ValueError

Raises :

AssertionError – If the subtraction of the members doesnt throw a ValueError

test_sub_members_in_field ( )

Test the subtraction of members in the field

Raises :

AssertionError – If the subtraction of the members is not the expected one.

static test_sum_members_different_fields ( )

Test the sum of members in different fields, which should raise a ValueError

Raises :

AssertionError – If the sum of the members doesnt throw a ValueError

test_sum_members_in_field ( )

Test the sum of members in the field

Raises :

AssertionError – If the sum of the members is not the expected one.

Finite Field and Finite Field Element tests

class tests.FiniteFieldTests. FiniteFieldTests

Class to test the FiniteField class.

static test_bsgs ( )

Test the bsgs algorithm.

Raises :

AssertionError – if the bsgs algorithm is not working as expected. which mean we received the power x but g^x ! = element.

static test_divide_elements_in_field ( )

Test the division of elements in the field.

Raises :

- AssertionError – if the division of the elements is not the expected division result.
- AssertionError – division in 0 doesnt throw an error.

static test_elements_in_field_easy ( )

Test the elements in the field with easy polynomial. The test checks that all elements generated by FiniteField are also present in galois results library.

Raises :

- AssertionError – if the number of elements in the field is not the expected number of elements.
- AssertionError – if the elements in the field are not the expected elements.

static test_elements_in_field_hard ( )

Test the elements in the field with hard polynomial. The test checks that all elements generated by FiniteField are also present in galois results library.

Raises :

- AssertionError – if the number of elements in the field is not the expected number of elements.

- **AssertionError** – if the elements in the field are not the expected elements.

static test_elements_in_field_hell ( )

Test the elements in the field with hell polynomial. The test checks that all elements generated by FiniteField are also present in galois results library.

Raises :

- **AssertionError** – if the number of elements in the field is not the expected number of elements.
- **AssertionError** – if the elements in the field are not the expected elements.

static test_elements_in_field_medium ( )

Test the elements in the field with medium polynomial. The test checks that all elements generated by FiniteField are also present in galois results library.

Raises :

- **AssertionError** – if the number of elements in the field is not the expected number of elements.
- **AssertionError** – if the elements in the field are not the expected elements.

static test_generators ( )

Test the generators of the field.

Raises :

AssertionError – if the generators of the field is not the expected generators.

static test_inverse ( )

Test the inverse of elements in the field.

Raises :

- **AssertionError** – if the inverse of the elements is not the expected inverse.
- **AssertionError** – inverse of 0 doesnt throw an error.

static test_mul_elements_in_field ( )

Test the multiplication of elements in the field.

Raises :

AssertionError – if the multiplication of the elements is not the expected multiplication result.

static test_operations_on_elements_different_fields ( )

Test the operations on elements in different fields - sum, sub, mul, truediv, for different fields we should get an error.

Raises :

AssertionError – if the operations on elements in different fields are not raising an error.

static test_order_of_elements ( )

Test the order of elements in the field.

Raises :

AssertionError – if the order of the elements is not the expected order.

static test_pow ( )

Test the power of elements in the field.

Raises :

AssertionError – if the power of the elements is not the expected power.

static test_sub_elements_in_field ( )

Test the subtraction of elements in the field.

Raises :

AssertionError – if the subtraction of the elements is not the expected subtraction result.

static test_sum_elements_in_field ( )

Test the sum of elements in the field.

Raises :

AssertionError – if the sum of the elements is not the expected sum.

tests.FiniteFieldTests. str_rep (*poly : list*) → str

Helper function to convert the polynomial to string representation for galois library.

Parameters :

poly (*list*) – polynomial coefficients

Returns :

string representation of the polynomial

Return type :

str

Example

```
>>> str_rep([1, 2, 3])
"3*X^2 + 2*X + 1"
```

Galwa Fields - API Documentation

galwa.fields

class galwa.fields. FiniteField (*p* , *f : ndarray* , *representation : str | None = 'polynomial'*)

FiniteField class represents an extension field of the form F_{p^n} where p is a prime number and n is the degree of polynomial $f(x)$ used for the extension.

$l = F_{p^n} = F_p[x]/f(x)$

Example:

```
>>> from galwa import FiniteField
>>> import numpy as np
>>> p = 2
>>> f = np.array([1, 1, 1])  # x^2 + x + 1
>>> F = FiniteField(p, f)
>>> F.elements
[FiniteFieldElement(0, f(x)= 1 + x + x², p=2),
FiniteFieldElement(1, f(x)= 1 + x + x², p=2),
FiniteFieldElement(x, f(x)= 1 + x + x², p=2),
FiniteFieldElement(1 + x, f(x)= 1 + x + x², p=2)]
```

__init__ (*p* , *f : ndarray* , *representation : str | None = 'polynomial'*)

Initialize the FiniteField class.

Parameters :

- p (*int*) – the prime number for the field.
- f (*np.ndarray*) – the irreducible polynomial f(x) used for the extension.
- Optional [str] (*representation*) – the representation of the elements in the field - polynomial, vector, or matrix, default is polynomial.

Raises :

AssertionError – if the polynomial f(x) is reducible over F_p or the representation is invalid

_calculate_illegal_powers ( ) → List [ndarray]

Calculate the representation of x^i for $n <= i <= 2n-2$ as $x^0, x^1, x^2, ..., x^{n-1}$ terms.

Methodology: since an element in the extension can have a degree of at most $n-1$, and the highest degree basis vector is $n-1$, all "illegal" x powers can be in the range of n to $2(n-1) = 2n-2$.

So we calculate each x^i using an induction:

$$x^i = x^{i-1} * x$$

Where x^{i-1} is the previous x^i and x is the basis vector.

So we start from x^n and calculate x^{n+1} using x^n and x , and so on until we reach x^{2n-2} .

The multiplication of x^i by x can be seen as a shift of x^i to the right, so we shift x^i to the right. Then we split the vector to a "valid" x^i and "illegal" x^i , the valid x^i are the x^i which have a degree smaller then n , and the illegal x^i are the x^i which have a degree bigger or equal to n .

The illegal x^i will be represented as a sum of the valid x^i . So we will be left with:

some_valid_vector + illgeal_degree_coef * converted_illegal_vector

Returns :

a list of the representation of x^i for i in range n to 2n-2

Return type :

List[np.ndarray]

_check_that_f_is_irreducible ( ) → bool

Checks if the polynomial $f(x)$ is irreducible over the field F_p for degree 2 or 3 For bigger degrees we assume that the polynomial is irreducible. A polynomial $f(x)$ of degree 2 or 3 is irreducible iff it has no roots in the field F_p .

Returns :

True if the polynomial is irreducible, False otherwise.

Return type :

bool

_create_elements ( ) → Dict [Tuple [ndarray] , FiniteFieldElement]

Create all possible elements in the field.

The elements are of the form $a_0 + a_1 x + a_2 x^2 + ... + a_{n-1} x^{n-1}$ where n is the degree of $f(x)$.

So we create a permutations of all possible coefficients for the elements in the field and use them to create the elements.

Returns :

a dictionary of all elements in the field where the key is
the vector representation of the element.

Return type :

Dict[Tuple[np.ndarray], FiniteFieldElement]

_find_all_dividers_of_field_size ( )

Find all the dividers of the size of the extension field.

Returns :

sorted array with all the dividers of the extension field

Return type :

List[int]

_find_generators ( )

Finds all the generators in the field. A generator element is an element whose order is equal to the order of the multiplicative group In other words, a generator element is an element whose powers generate all the elements in the field.

$$< g >= \{g^0, g^1, g^2, ..., g^{p^n-2}\} = l^x = l - \{0\}$$

Returns :

a list of all the generators in the field

Return type :

List[FiniteFieldElement]

property **elements** *: List [FiniteFieldElement]*

Returns all elements in the field as a list.

Returns :

a list of all the elements in the field.

Return type :

List[FiniteFieldElement]

Example:

```
>>> from galwa import FiniteField
>>> import numpy as np
>>> p = 2
>>> f = np.array([1, 1, 1])
>>> F = FiniteField(p, f)
>>> F.elements
[FiniteFieldElement(0, f(x)= 1 + x + x², p=2),
FiniteFieldElement(1, f(x)= 1 + x + x², p=2),
FiniteFieldElement(x, f(x)= 1 + x + x², p=2),
FiniteFieldElement(1 + x, f(x)= 1 + x + x², p=2)]
```

elements_as_matrices ( ) → None

Changes the representation of the elements to matrices.

Returns :

None

Example:

```
>>> from galwa import FiniteField
>>> import numpy as np
>>> p = 2
>>> f = np.array([1, 1, 1])
>>> F = FiniteField(p, f)
>>> F.elements_as_matrices()
>>> F.elements
[FiniteFieldElement(None, f(x) = [1 1 1] p=2),
 FiniteFieldElement([[1 0][0 1]], f(x) = [1 1 1] p=2),
```

```
 FiniteFieldElement([[0 1][1 1]], f(x) = [1 1 1] p=2),
 FiniteFieldElement([[1 1][1 0]], f(x) = [1 1 1] p=2)]
```

elements_as_polynomials ( ) → None

Changes the representation of the elements to polynomials.

Returns :

None

Example:

```
>>> from galwa import FiniteField
>>> import numpy as np
>>> p = 2
>>> f = np.array([1, 1, 1])
>>> F = FiniteField(p, f)
>>> F.elements_as_polynomials()
>>> F.elements
[FiniteFieldElement(0, f(x)= 1 + x + x², p=2),
FiniteFieldElement(1, f(x)= 1 + x + x², p=2),
FiniteFieldElement(x, f(x)= 1 + x + x², p=2),
FiniteFieldElement(1 + x, f(x)= 1 + x + x², p=2)]
```

elements_as_vectors ( ) → None

Changes the representation of the elements to vectors.

Returns :

None

Example:

```
>>> from galwa import FiniteField
>>> import numpy as np
>>> p = 2
>>> f = np.array([1, 1, 1])
>>> F = FiniteField(p, f)
>>> F.elements_as_vectors()
>>> F.elements
[FiniteFieldElement([0 0], f(x) = [1 1 1] p=2),
FiniteFieldElement([1 0], f(x) = [1 1 1] p=2),
FiniteFieldElement([0 1], f(x) = [1 1 1] p=2),
FiniteFieldElement([1 1], f(x) = [1 1 1] p=2)]
```

property **generators** *: List [FiniteFieldElement]*

Property to get all the generators in the field

> **Note**
>
> At the first call of this property the generators are calculated and stored in the generators attribute The reason for this is that the calculation of the generators is an expensive operation. First call might be slow depend on the p value, but once the generators are calculated they are stored and can be accessed quickly.

Returns :

a list of all the generators in the field

Return type :

List[FiniteFieldElement]

Example:

```
>>> from galwa import FiniteField
>>> import numpy as np
>>> p = 2
>>> f = np.array([1, 1, 1])
>>> F = FiniteField(p, f)
>>> F.generators
[FiniteFieldElement(x, f(x)= 1 + x + x², p=2),
FiniteFieldElement(1 + x, f(x)= 1 + x + x², p=2)]
```

get_element (*a : ndarray*) → FiniteFieldElement

Given a vector representation of an element in the field, returns the element.

Parameters :

a (*np.ndarray*) – the vector representation of the element.

Returns :

the element in the field, None if the element is not in the field

Return type :

FiniteFieldElement

Raises :

AssertionError – if the element is not in the field.

Example:

```
>>> from galwa import FiniteField
>>> import numpy as np
>>> p = 2
>>> f = np.array([1, 1, 1])
>>> F = FiniteField(p, f)
>>> F.get_element(np.array([1, 1]))
FiniteFieldElement(1 + x, f(x)= 1 + x + x², p=2)
```

property **order** *: int*

Property to get the order of the field, which is p^n where n is the degree of the polynomial $f(x)$.

Returns :

the order of the field

Return type :

int

Example:

```
>>> from galwa import FiniteField
>>> import numpy as np
>>> p = 2
>>> f = np.array([1, 1, 1])
>>> F = FiniteField(p, f)
>>> F.order
4
```

property **representation** *: str*

Get the representation of the elements in the field - polynomial, vector, or matrix

Returns :

the representation of the elements in the field

Return type :

str

Example:

```
>>> from galwa import FiniteField
>>> import numpy as np
>>> p = 2
>>> f = np.array([1, 1, 1])
>>> F = FiniteField(p, f)
>>> F.representation
'polynomial'
```

galwa.elements

class galwa.elements. FiniteFieldElement (*a : ndarray* , *field : FiniteField* , *representation : str | None = 'polynomial'*)

FiniteFieldElement class represents an element in an extension F_{p^n} where p is a prime number and n is the degree of polynomial $f(x)$ used for the extension. In other words, a class to represent an element $a \in l = F_{p^n}[x]/ < f(x) >$

Example:

```
>>> from galwa import FiniteField, FiniteFieldElement
>>> import numpy as np
>>> f = np.array([1, 1, 0, 1])
>>> p = 2
>>> field = FiniteField(p, f)
>>> a = FiniteFieldElement(np.array([1, 0, 1]), field)
>>> a
FiniteFieldElement(1 + x², f(x)= 1 + x + x³, p=2)
```

__init__ (*a : ndarray* , *field : FiniteField* , *representation : str | None = 'polynomial'*)

Initialize the FiniteFieldElement class.

Parameters :

- **a** (*np.ndarray*) – the element in the field.

- field (*FiniteField*) – the field the element belongs to.
- Optional [str] (*representation*) – the representation of the element (polynomial, vector, matrix), default is polynomial.

Raises :

AssertionError – if the representation is invalid or the element is not in the field or wrong type.

static _exponentiation_by_squaring (*a* , *n*) → FiniteFieldElement

This function calculate the exponentiation of the element a^n using the exponentiation by squaring algorithm.

Parameters :

- a (*FiniteFieldElement*) – the element to exponentiate.
- n (*int*) – the power to exponentiate the element by.

Returns :

the result of the exponentiation.

Return type :

FiniteFieldElement

_get_inverse ( ) → FiniteFieldElement

Get the inverse of the element (for multiplicative group) The inverse of the element is the element that when multiplied by the element gives the identity element 1

Note

This is the only operation that we use PrimeFieldElement, since regular matrix inverse can return float numbers. We must use PrimeFieldElement to perform the division between two elements.

Returns :

the inverse of the element

Return type :

FiniteFieldElement

as_matrix ( ) → None

Changes the representation of the element to matrix. So once printing the element, it will be printed as a matrix.

Returns :

None

Example

```
>>> from galwa import FiniteField, FiniteFieldElement
>>> import numpy as np
>>> f = np.array([1, 1, 0, 1])
>>> p = 2
>>> field = FiniteField(p, f)
>>> a = FiniteFieldElement(np.array([1, 0, 1]), field)
>>> a.as_matrix()
>>> a
FiniteFieldElement([[1 0 1]
                    [1 0 0]
                    [0 1 0]], f(x) = [1 1 0 1] p=2)
>>> print(a)
[[1 0 1]
[1 0 0]
[0 1 0]]
```

as_polynomial ( ) → None

Changes the representation of the element to polynomial. So once printing the element, it will be printed as a polynomial.

Returns :

None

Example

```
>>> from galwa import FiniteField, FiniteFieldElement
>>> import numpy as np
```

```
>>> f = np.array([1, 1, 0, 1])
>>> p = 2
>>> field = FiniteField(p, f)
>>> a = FiniteFieldElement(np.array([1, 0, 1]), field)
>>> a.as_polynomial()
>>> a
FiniteFieldElement(1 + x², f(x)= 1 + x + x³, p=2)
>>> print(a)
1 + x²
```

as_vector ( ) → None

Change sthe representation of the element to vector. So once printing the element, it will be printed as a vector.

Returns :

None

Example

```
>>> from galwa import FiniteField, FiniteFieldElement
>>> import numpy as np
>>> f = np.array([1, 1, 0, 1])
>>> p = 2
>>> field = FiniteField(p, f)
>>> a = FiniteFieldElement(np.array([1, 0, 1]), field)
>>> a.as_vector()
>>> a
FiniteFieldElement([1 0 1], f(x) = [1 1 0 1] p=2)
>>> print(a)
[1 0 1]
```

static check_that_element_is_in_field (*element : ndarray* , *field : FiniteField*) → bool

Check if the element is in the field with the given irreducible polynomial $f(x)$

A valid element must be of the form $a = a_0 + a_1x + a_2x^2 + ... + a_{n-1}x^{n-1}$ where n is the degree of $f(x)$. In other words, the degree of the element must be less than the degree of $f(x)$

Parameters :

- **element** (*np.ndarray*) – the element to check.
- **field** (*FiniteField*) – the field to check the element in.

Returns :

True if the element is in the field, False otherwise.

Return type :

bool

Example

```
>>> from galwa import FiniteField, FiniteFieldElement
>>> import numpy as np
>>> f = np.array([1, 1, 0, 1])
>>> p = 2
>>> field = FiniteField(p, f)
>>> a = np.array([1, 0, 1])
>>> FiniteFieldElement.check_that_element_is_in_field(a,
field)
```

property **dimension** *: int*

Get the dimension of the element. (vector dimension)

Returns :

the dimension of the element

Return type :

int

embed_in_gln ( ) → ndarray | None

Embed the element in $GL_n(p)$ where n is the degree of the irreducible polynom $f(x)$.

Methodology: for an element a in the field, we calculate the multiplication of $a * x^i$ for $0 <= i <= (n-1)$ and store the results in the columns of a matrix.

If the multiplication has a degree higher than $n-1$, we use previously calculated vectors which represent x^i for $n <= i <= 2n-2$.

The maximum degree of the multiplication is at most 2n-2 since the highest degree in an element is n-1 and the highest degree basis vector is also n-1. So the highest degree in the multiplication is $2(n-1) = 2n-2$

Note

For the zero element, there is no representation in $GL_n(p)$ since the zero element is not part of the multiplicative group. In this case we return None

Returns :

the matrix representation of the element in $GL_n(p)$

Return type :

np.ndarray

Example

```
>>> from galwa import FiniteField, FiniteFieldElement
>>> import numpy as np
>>> f = np.array([1, 1, 0, 1])
>>> p = 2
>>> field = FiniteField(p, f)
>>> a = FiniteFieldElement(np.array([1, 0, 1]), field)
>>> a.embed_in_gln()
array([[1, 1, 0],
       [0, 0, 1],
       [1, 0, 0]])
```

is_generator ( )

Check if the element is a generator in the field.

A generator element is an element whose order is equal to the order of the multiplicative group. In other words, a generator element is an element whose powers generate all the elements in the field.

Returns :

True if the element is a generator, False otherwise

Return type :

bool

Example

```
>>> from galwa import FiniteField, FiniteFieldElement
>>> import numpy as np
```

```
>>> f = np.array([1, 1, 0, 1])
>>> p = 2
>>> field = FiniteField(p, f)
>>> a = FiniteFieldElement(np.array([1, 0, 1]), field)
>>> a.is_generator()
True
```

is_identity_of_multiplication ( )

Checks if the element is the identity element of the multiplication operation. The identity element of the multiplication operation is the element 1.

Returns :

True if the element is the identity element, False otherwise

Return type :

bool

Example

```
>>> from galwa import FiniteField, FiniteFieldElement
>>> import numpy as np
>>> f = np.array([1, 1, 0, 1])
>>> p = 2
>>> field = FiniteField(p, f)
>>> a = FiniteFieldElement(np.array([1, 0, 1]), field)
>>> a.is_identity_of_multiplication()
False
```

multiplicative_order ( ) → int | None

Calculates the multiplicative order of the element in the field.

The multiplicative order of an element a in a finite field is the smallest positive integer n such that $a^n = 1$ For the 0 element, the order is not defined since 0 is not part of the multiplicative group.

Methodology:

From lagrange theorem we know that for H as subgroup of G then the order of any element in G divides the order of G . That is true for all $g \in G, | < g > |||G|$

In our case the multiplicative group of $F_p^x = F_p - \{0\}$ is a subgroup of the multiplicative group l^x .

So for all $a \in l^x, O(a)|O(l^x)$.

So first, we calculate all divisors of the order of the multiplicative group of the field, the complexity of this operation is $O(\sqrt{n})$ where n is the order of the multiplicative group.

The divisors array will be sorted, we will start from the smallest and calculate a^d for each divisor d and check if the result is the identity element.

Calculating a^d can be done using exponentiation by squaring algorithm, the complexity of this operation is $O(\log(d))$.

So the complexity in the best case will be $O(\log(d)$ and in the worst case $O(k \log(d))$ where k is the number of divisors.

Returns :

the multiplicative order of the element, None if the element is 0.

Return type :

int

Example

```
>>> from galwa import FiniteField, FiniteFieldElement
>>> import numpy as np
>>> f = np.array([1, 1, 0, 1])
>>> p = 2
>>> field = FiniteField(p, f)
>>> a = FiniteFieldElement(np.array([1, 0, 1]), field)
>>> a.multiplicative_order()
7
```

property **order** *: int | None*

Returns the multiplicative order of the element. The order is define as

$O(a) = min(n > 0 : a^n = 1 mod p, n \in N)$

Note

At first call, the order is calculated and stored in the ord attribute. The reason for that is that the calculation of the order can be an expensive operation. First call

might take some time depending on the value of p, but once calculated the order can be accessed quickly.

Returns :

the multiplicative order of the element. None if the element is 0

Return type :

int

property **representation**

Get the representation of the element (polynomial, vector, matrix)

Returns :

the representation of the element.

Return type :

str

Example

```
>>> from galwa import FiniteField, FiniteFieldElement
>>> import numpy as np
>>> f = np.array([1, 1, 0, 1])
>>> p = 2
>>> field = FiniteField(p, f)
>>> a = FiniteFieldElement(np.array([1, 0, 1]), field)
>>> a.representation
'polynomial'
```

class galwa.elements. PrimeFieldElement (*a : int* , *p : int*)

PrimeFieldElement class represents an element in a prime field F_p where p is a prime number. The element represent the value a mod p

Example:

```
>>> from galwa import PrimeFieldElement
>>> a = PrimeFieldElement(3, 5)
>>> a
```

```
PrimeFieldElement(value= 3,prime= 5)
>>> print(a)
3 mod 5
>>> b = PrimeFieldElement(4, 5)
>>> a + b
PrimeFieldElement(value= 2,prime= 5)
>>> a - b
PrimeFieldElement(value= 4,prime= 5)
>>> a * b
PrimeFieldElement(value= 2,prime= 5)
>>> a / b
PrimeFieldElement(value= 2,prime= 5)
>>> a ** 2
PrimeFieldElement(value= 4,prime= 5)
>>> a**-1
PrimeFieldElement(value= 2,prime= 5)
>>> a.inverse
PrimeFieldElement(value= 2,prime= 5)
>>> a == b
False
```

__init__ (*a : int* , *p : int*)

Initialize the PrimeFieldElement class.

Parameters :

- a (*int*) – the element in the prime field.
- p (*int*) – the prime number for the prime field.

_find_inverse ( )

Finds the inverse of the element. (for multiplicative group)

The inverse of an element a is the element b such that $a * b = 1 mod p$, to find b we are using the extended Euclidean algorithm.

Returns :

the inverse of the element.

Return type :

int

Raises :

ValueError – if the element does not have an inverse.

property **inverse**

Returns the inverse of the element. (for multiplicative group)

The inverse is being defined as the element b such that $a * b = 1 mod p$.

Returns :

the inverse of the element.

Return type :

PrimeFieldElement

Example

```
>>> from galwa import PrimeFieldElement
>>> a = PrimeFieldElement(3, 5)
>>> a.inverse
PrimeFieldElement(value= 2,prime= 5)
```

galwa.utils

galwa.utils. _adjugate_matrix (*matrix : list*) → list

Calculates the adjugate matrix of a square matrix.

Parameters :

matrix (*list*) – a square matrix represented as a list of lists.

Returns :

the adjugate matrix.

Return type :

list

galwa.utils. _cofactor_matrix (*matrix : list*) → list

Calculates the cofactor matrix of a square matrix.

Parameters :

matrix (*list*) – a square matrix represented as a list of lists.

Returns :

the cofactor matrix.

Return type :

list

galwa.utils. _transpose (*matrix : list*) → list

Transposes a matrix.

Parameters :

matrix (*list*) – a matrix represented as a list of lists.

Returns :

the transposed matrix.

Return type :

list

galwa.utils. bsgs (*generator* , *element* , *group_order*)

Baby-step Giant-step algorithm to solve the discrete logarithm problem.

$g^x = h$

Parameters :

- **generator** (*FiniteFieldElement*) – g in $g^x = h$
- **element** (*FiniteFieldElement*) – h in $g^x = h$
- **group_order** (*int*) – group order , to initialize the table.

Returns :

x such that g^x = h mod p

Return type :

int

Raises :

ValueError – if the discrete logarithm is not found

Example

```
>>> from galwa import FiniteFieldElement, FiniteField
>>> from galwa.utils import bsgs
>>> import numpy as np
>>> f = np.array([2, 0, 0, 2, 1])
>>> p = 3
>>> F = FiniteField(p, f)
>>> g = FiniteFieldElement(np.array([1, 1, 0, 0]), F)
>>> h = g ** 10
>>> h
FiniteFieldElement(2, f(x)= 2 + 2·x³ + x⁴, p=3)
>>> order = F.order - 1
>>> bsgs(g, h, order)
10
```

galwa.utils. find_all_dividers_of_number (*num : int*) → list

Find all the dividers of the size of the given number.

Parameters :

num (*int*) – number to find the dividers of.

Returns :

sorted array with all the dividers of the given number

Return type :

List[int]

Example

```
>>> find_all_dividers_of_number(12)
[1, 2, 3, 4, 6, 12]
```

galwa.utils. invert_matrix (*matrix : list*) → list

Inverts a square matrix using the determinant and adjugate.

Parameters :

matrix (*list*) – a square matrix represented as a list of lists

Returns :

the inverse of the matrix

Return type :

list

galwa.utils. xgcd (*a* , *b*)

Extended Euclidean algorithm to find the greatest common divisor and the coefficients of Bezout's identity.

Parameters :

- **a** (*int*) – first number
- **b** (*int*) – second number

Returns :

gcd, s, t such that $gcd(a, b) = s * a + t * b$

Return type :

tuple

Example

```
>>> from galwa.utils import xgcd
>>> xgcd(240, 46)
(2, -9, 47)
```

Created using Sphinx 7.1.2.