Classic McEliece with Binary Goppa Codes

This is a notebook by Felix Paul inspired by the work of Michiel Marcus under the supervision of Prof. Dr. Tanja Lange.

The notebook presents the McEliece cryptosystem in its (insecure) textbook version. It is based on the Whitepaper .

- To just use the code set verbose=0.
- For a description of all the necessary steps set verbose = 1.
- For educational purposes and debugging set verbose=2.
- The verbosity is set in each called function so you can see each step during the calculation.

For technical reasons, additional restrictions are imposed on the polynomials here, which are theoretically not necessary.

In case of unforeseen errors, it is recommended to rerun all previous code cells.

Bits from the original document will be used to clarify which parts of the code belong to which parts of the whitepaper. Please note that for larger parameters it might take a longer time to display the output. Additionally, please be warned that the code used in the notebook is **not** safe to use in real-world applications. This code is purely meant for educational purposes and is prone to side-channel attacks. Additionally, due to technical reasons there are some restrictions on the polynomials, which reduces the security even more. Lastly, if the notebook produces strange errors, run all previous cells from top to bottom first. You can do this by holding shift and clicking on the first and last cell of your desired selection and clicking the *run* button.

The first part of the notebook sets up a binary Goppa code step-by-step, so that the user can play with values and see how it works. After creating the binary Goppa code, the notebook shows how the code is used in the McEliece cryptographic system. The user is encouraged to edit and re-run the code as they like. A selected cell can be re-run by clicking the *run* button in the top menu or using *shift* + *enter*.

Imports and helper functions for nice outputs and easy handling

In [ ]:
```
from IPython.display import display, Math, Latex
import random
import numpy as np
import warnings
import math
warnings.simplefilter('always', Warning)
```

If you uncomment the second lateprint function all the latex outputs will not be rendered and can directly be copied to a latex document.

In [ ]:
```
def int_to_binary_list(y):
    # function to encode messages
    return [int(x) for x in bin(y)[2:]]

def latexprint(l):
    if type(l) is list:
        for element in l:
            display(Math(r'{}'.format(element)))
    else:
        display(Math(r'{}'.format(l)))

"""
use this function to copy the output in latex documents
- the output is not rendered but raw latex code
comment out this function to see rendered output
"""
"""
def latexprint(l):
    if type(l) is list:
        for element in l:
            display(((r'{}'.format(element))))
    else:
        display(((r'{}'.format(l))))
"""
print('') # so the cell produces no bad rendered output
```

The first order of business is creating the binary Goppa code. Recall that we have 3 parameters to specify:

- m: the degree of the modulus of $\mathbb{F}_{2^m}$
- t: the maximum number of correctable errors
- n: the number of elements in our support, taken from $\mathbb{F}_{2^m}$

Together, they implicitly define our dimension k.

- k: $n - m \cdot t$

Therefore, n should be greater than $m * t$.

Additionally, we run into problems if m is divisble by t, because then we cannot guarantee later on that $g(\sigma_i(z))$ is not 0 for all support elements. As we divide by $g(\sigma_i(z))$ during the computation of the parity check matrix, we would then divide by 0. This requirement is a result of the limited choices that are reasonable to make in Sage. Normally this is not a requirement as in general, t is much bigger than m.

Lastly, $n = k + t * m$, so n can in theory have any value between $t \cdot m + 1$ and 2^m, because in the smallest case $k = 1$ and in the largest case we can use all the elements of $\mathbb{F}_{2^m}$. Note that n and k are security parameters, so choosing them with care important. For simplicity, in the given example we take n to be the highest possible value. Feel free to change the value of n.

In [ ]:
```
def check_parameters(m, t, n, k, verbose):
    assert k > 0, '''k should be bigger than 0. Try increasing n
                 or decreasing t or m'''
    assert m % t != 0, '''if m is divisible by t, we will get
                        problems when we divide by g(sigma_i)
                        in the parity check matrix.
                        make sure this is not the case'''
    assert n <= 2^m, '''n exceeds the number of elements
                        in the support'''
    if verbose:
        print(f'\nParameter setting: \n'
              f'---------------------------------\n'
              f'm = {m}\nt = {t}\nn = {n}\nk = {k} \n')
```

Now that the parameters have been checked, we have to decide which modulus we want to use. The following cell enumerates all possibilities for $\mathbb{F}_{2^m}$.

Now we can pick one modulus from the list. In the code, f denotes the modulus for $\mathbb{F}_{2^m}$. We can now initialise the field.

In [ ]:
```
def create_F_2m(m, verbose):
    #initialisation for the field indeterminates
    _.<z> = GF(2)[]
    _.<x> = GF(2)[]

    #list all irreducible polynomials of degree m.
    # These are the viable candidates for the moduli.
    irreducibles_z = [p for p in (GF(2)['z']).polynomials(m)
                      if p.is_irreducible()]

    #This is the modulus for f_{2^m}
    #We can either take the first element of the list of
    # irreducible polynomials
    #or pick our own form the list above
    f = irreducibles_z[0]

    #Now we can initialise the field
    K.<z> =  GF(2^m, modulus = f)

    if verbose:
        print(f'\nCreation of F_2^m:\n'
              f'----------------------')
        if verbose > 1:
            print(type(irreducibles_z))
            print(f'These are the {len(irreducibles_z)}'
                  f'viable moduli of degree {m}')
            latexprint(latex(irreducibles_z))
        print("This is the chosen modulus f(z)")
        latexprint(f)
        print(f'This is an enumeration of the elements'
              f'of F_2^{m}')
        latexprint(["{}: {}".format(i,element)
                    for i, element in enumerate(K)])
    return K
```

Now that the field $\mathbb{F}_{2^m}$ has been created, we can select a modulus for the big field $\mathbb{F}_{2^m}[x]/g(x,z)$. There are a lot of possibilities for $g(x,z)$, but in this notebook we will consider only the $g(x,z)$ with binary coefficients for technical reasons. Do not make this choice when building a real cryptosystem. For educational purposes, 5 viable options for $g(x,z)$ are shown that have coefficients in F_{2^m}.

In the code, g denotes $g(x,z)$. We can now pick a modulus for $\mathbb{F}_{2^m}[x]/g(x,z)$ by picking an element from the list above.

In [ ]:
```
def show_possible_g_without_binary_coefficients(K, t, verbose):
    # function to show possible non binary polynomials
    # - only for description - not use!
    G.<x> = K[]
    counter = 0
    ps = []
    number_examples = 5
    while counter < number_examples:
        p = G.random_element(t)
        if p.is_irreducible():
            ps.append(p)
            counter = counter + 1

    print(f'These are {number_examples} examples of possible'
          f'goppa polynomials (moduli of degree {t})'
          f' with coefficients in z, which we will not use')
    latexprint(ps)

def create_F_2m_mod_g(K, t, verbose,  method='first_binary'):
    # method: 'random_binary', 'first_binary',
    #           Integer i (choose i-th binary irreducible polynomial)
    #This extends F_{2^m} to F_{2^m}[x]
    G.<x> = K[]
    # choose first irreducible binary g and apply g
    # as the modulus to the field F_{2^m}[x]
    if method == 'random_binary':
        binary_irreducibles_x = [p for p in GF(2)['x'].polynomials(t)
                                 if p.is_irreducible()]
        g = random.choice(binary_irreducibles_x)
    # !'random' does not work!
    elif method == 'random':
        raise_exception = True
        for i in range(10*t):
            print(f't: {t}, type: {type(t)}')
            p = G.random_element(t)
            if p.is_irreducible():
                print(p, p.is_irreducible())
                #G.<x> = K.extension(p)
                #print(f'is field mod g: {G.is_field()}')
                g = p
                raise_exception = False
                break
        if raise_exception:
            raise Exception('Finding random g took to long!')
    elif type(method) == Integer:
        print('choose g type Integer')
        irreducibles_x = [p for p in GF(2)['x'].polynomials(t)
                          if p.is_irreducible()]
        g = irreducibles_x[method]
    else: # first_binary
        for p in GF(2)['x'].polynomials(t):
            if p.is_irreducible():
                g = p
    # G.<x> = K.extension(g) use g as modulus
    if verbose:
        if verbose > 1:
            show_possible_g_without_binary_coefficients(K,
                                                        t,
                                                        verbose)
            print(f'These are the viable moduli of degree {t}'
                  f' with binary coefficients')
            binary_irreducibles_x = [p for p in GF(2)['x'].polynomials(t)
                                     if p.is_irreducible()]
            latexprint(binary_irreducibles_x)
        print('This is the chosen modulus g(x,z)')
        latexprint(g)
    return g
```

Now that the fields have been created, we can define our support σ. By default we take the entirety of $\mathbb{F}_{2^m}$, but it is possible to take a subset if n is smaller than 2^m. We will see later on that it is best to take a random permutation for σ for security reasons, but from a general point of view we can take the first n elements of $\mathbb{F}_{2^m}$.

In [ ]:
```
def choose_support(K, n, m, verbose, method='first_n'):
    #method: 'first_n', 'random_n',
    #          list of length n with indizes
    if method == 'random_n':
        index_perm = list(np.random.permutation(range(2^m)))
        K_permutation = [list(K)[index_perm[i]]
                         for i in range(2^m)]
        permuted_sigma = K_permutation[:n]
        if verbose:
            latexprint(r"\text{This is the randomly permuted support }\sigma")
            latexprint(["{}: {}".format(i,element)
                                for i, element in enumerate(permuted_sigma)])
        return permuted_sigma
    # just give a list of length n to choose support
    elif type(method) == list:
        print('choose support type: given list')
        assert len(method) == n, '''given support does not
                                    have length n'''
        sigma = [list(K)[i] for i in method]
    else: # first_n
        sigma = list(K)[:n]
    if verbose:
        print(f'{method} Support: \n'
              f'-------------\n'
              f'These are the elements in the support')
        latexprint([f'{i}: {j}'
                    for i,j in enumerate(sigma)])
    return sigma
```

The following piece of code creates the first of 3 matrices that are used to create the parity check matrix. This matrix looks at the modulus $g(x,z)$ and has the function of taking elements from the second matrix that together form the numerator of the inverse for each support element. For a more detailed explanation, consult the respective document. This matrix is always of the form

$$\begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ g_{t-1} & 1 & 0 & \dots & 0 \\ g_{t-2} & g_{t-1} & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ g_1 & g_2 & g_3 & \dots & 1 \end{bmatrix}$$

In general, this means that all g_i are in $\mathbb{F}_{2^m}$, but this notebook only considers binary polynomials for $g(x,z)$, as mentioned before.

In [ ]:
```
# This function contains which elements
# from the second matrices form the numerators of the inverses
def generate_g_matrix(g, t, K, verbose):
    # sparse = False means also include non zero coefficients
    coefs = g.coefficients(sparse=False)[::-1]
    result = []
    for i in range(t):
        subarray = []
        for j in range(i+1):
            subarray.append(coefs[i-j])
        for k in range(t-i-1):
            subarray.append(0)
        result.append(subarray)
    g_matrix = matrix(K, result)
    if verbose > 1:
        print(f'First Matrix of Check Matrix:\n'
              f'--------------------------------')
        latexprint(latex(g_matrix))
    return g_matrix
```

The next cell generates the second of 3 afore-mentioned matrices. This matrix contains all the support elements and their increasing powers. This matrix is of the form

$$\begin{bmatrix} 1 & 1 & \dots & 1 \\ \sigma_0(z) & \sigma_1(z) & \dots & \sigma_{n-1}(z) \\ \sigma_0(z)^2 & \sigma_1(z)^2 & \dots & \sigma_{n-1}(z)^2 \\ \vdots & \vdots & \ddots & \vdots \\ \sigma_0(z)^{t-1} & \sigma_1(z)^{t-1} & \dots & \sigma_{n-1}(z)^{t-1} \end{bmatrix}$$

In [ ]:
```
#This function applies \sigma_i(z)^j to each entry of the matrix
def create_zs_matrix(K, t, n, sigma, verbose):
    zs = matrix(K, t, n, lambda i, j: sigma[j]^i)
    if verbose > 1:
        print(f'Second Matrix of Check Matrix:\n'
              f'----------------------------------')
        latexprint(latex(zs))
    return zs
```

The following matrix is the last of the 3 matrices involved in the parity check matrix. This matrix handles the denominators of the inverses of the support elements. This matrix is of the form

$$\begin{bmatrix} \frac{1}{g(\sigma_0(z))} & 0 & 0 & 0 \\ 0 & \frac{1}{g(\sigma_1(z))} & 0 & 0 \\ 0 & 0 & \ddots & 0 \\ 0 & 0 & 0 & \frac{1}{g(\sigma_{n-1}(z))} \end{bmatrix}$$

In [ ]:
```
def create_diagonal_matrix(K, g, sigma, verbose):
    #this is the diagonal matrix with elements 1/g(sigma_i(z))
    diagonal = diagonal_matrix(K, [g(element)^-1
                                   for element in sigma])
    if verbose > 1:
        print(f'Third Matrix of Check Matrix:\n'
              f'----------------------------------')
        latexprint(latex(diagonal))
    return diagonal
```

Now we can calculate the parity check matrix by multiplying these matrices.

Note: You can omit the g_matrix because it is invertible!

Here the g_matrix is not used!

Feel free to change that.

In [ ]:
```
def create_parity_check_matrix_over_f2m(g, t, K, n,
                                        sigma, verbose):
    # This is the multiplication of the g matrix,
    # matrix with z powers
    # and the diagonal matrix with 1/g(\sigma_i(z)),
    # which results in the t*n parity check matrix in F_{2^m}
    g_matrix = generate_g_matrix(g,t,K, verbose)
    zs = create_zs_matrix(K, t, n, sigma, verbose)
    diagonal = create_diagonal_matrix(K, g, sigma, verbose)
    # alternative: g_matrix * zs * diagonal
    parity_check_matrix_f2m = zs * diagonal
    if verbose > 1:
        print(f'Check Matrix over F_2^m with dimensions'
              f'{parity_check_matrix_f2m.nrows()}x'
              f'{parity_check_matrix_f2m.ncols()}:\n'
              f'--------------------------------')
        latexprint(latex(parity_check_matrix_f2m))
    return parity_check_matrix_f2m
```

As we prefer working with binary matrices, we will convert the $t \times n$ parity check matrix in $\mathbb{F}_{2^m}$ to a $t * m \times n$ binary parity check matrix by extending all elements vertically. The first binary value is the coefficient belonging to z^0. The second is the coefficient belonging to z^1, etc. In the paper, this is denoted as

$$H = \begin{bmatrix} I_{0,0,0} & I_{1,0,0} & \cdots & I_{n-1,0,0} \\ I_{0,0,1} & I_{1,0,1} & \cdots & I_{n-1,0,1} \\ \vdots & \vdots & \ddots & \vdots \\ I_{0,0,m-1} & I_{1,0,m-1} & \cdots & I_{n-1,0,m-1} \\ I_{0,1,0} & I_{1,1,0} & \cdots & I_{n-1,1,0} \\ I_{0,1,1} & I_{1,1,1} & \cdots & I_{n-1,1,1} \\ \vdots & \vdots & \ddots & \vdots \\ I_{0,1,m-1} & I_{1,1,m-1} & \cdots & I_{n-1,1,m-1} \\ \vdots & \vdots & \ddots & \vdots \\ I_{0,t-1,0} & I_{1,t-1,0} & \cdots & I_{n-1,t-1,0} \\ I_{0,t-1,1} & I_{1,t-1,1} & \cdots & I_{n-1,t-1,1} \\ \vdots & \vdots & \ddots & \vdots \\ I_{0,t-1,m-1} & I_{1,t-1,m-1} & \cdots & I_{n-1,t-1,m-1} \end{bmatrix} \tag{1}$$

In [ ]:
```
#This function vertically extends the elements
def f2m_to_binary_matrix(M_f2m, m, verbose):
    resultlist = []
    rows = [r.list() for r in list(M_f2m)]
    for row in rows:
        expandedrow = []
        for element in row:
            coefs = element.polynomial().coefficients(sparse=
                                                       False)
            coefs.extend((m - len(coefs))*[0])
            expandedrow.append(coefs)
        expandedrow_T = list(map(list, zip(*expandedrow)))
        resultlist.extend(expandedrow_T)
    parity_check_matrix_f2 = matrix(GF(2), resultlist)
    if verbose > 1:
        print(f'The check matrix over F_2 has dimensions '
              f'{parity_check_matrix_f2.nrows()}x'
              f'{parity_check_matrix_f2.ncols()} and is  :\n'
              f'--------------------------------')
        latexprint(latex(parity_check_matrix_f2))
    return parity_check_matrix_f2

def create_binary_check_matrix(g, t, K, n, sigma, m, verbose):
    # just a function calling the sub functions
    # to have a nicer interface
    parity_check_matrix_f2m = create_parity_check_matrix_over_f2m(g,
                                                                  t,
                                                                  K,
                                                                  n,
                                                                  sigma,
                                                                  verbose)
    parity_check_matrix_f2 =  f2m_to_binary_matrix(parity_check_matrix_f2m,
                                                   m,
                                                   verbose)
    return parity_check_matrix_f2
```

Now that we have the parity check matrix, we want to construct the generator matrix. Recall that we need to convert our parity check matrix to the systematic form $\left(P^T_{(n-k)\times k} | I_{n-k} \right)$ to get the generator matrix $\left(I_k | P_{k\times(n-k)} \right)$. However, since we start with the parity check matrix, it is easier to convert it to the form $\left(I_{n-k} | P^T_{(n-k)\times k} \right)$ using the row echelon function, which basically applies Gaussian elimination. Then our generator matrix becomes $\left(P_{k\times(n-k)} | I_k \right)$

In [ ]:
```
def parity_check_matrix_standard_form(g,
                                      t,
                                      K,
                                      n,
                                      sigma,
                                      m,
                                      k,
                                      verbose):
    parity_check_matrix_f2 =  create_binary_check_matrix(g,
                                                         t,
                                                         K,
                                                         n,
                                                         sigma,
                                                         m,
                                                         verbose)
    #This is the t*m by n parity check matrix in F_2 in systematic form
    parity_check_matrix_f2_standard_form = parity_check_matrix_f2.echelon_form()
    left_side = parity_check_matrix_f2_standard_form[0:, :n-k]
    assert left_side == identity_matrix(n-k), '''the parity check matrix cannot be
                                                 converted to systematic form.
                                                 Please choose other parameters'''
    if verbose:
        print(f'Parity check matrix in echolon form:\n'
              f'-------------------------------------')
        latexprint(r"\text{This is the parity check matrix}")
        latexprint(latex(parity_check_matrix_f2_standard_form))
    return parity_check_matrix_f2_standard_form

# Here we calculate the parity check part
# of the parity check matrix
# and use it to construct the generator matrix
def parity_to_generator(parity, k, verbose):
    P_T =parity[0:, -k:]
    P = P_T.transpose()
    generator_matrix = P.augment(identity_matrix(k))
    if verbose:
        latexprint(r"\text{This is the generator matrix}")
        latexprint(latex(generator_matrix))
    return generator_matrix
```

Now we have successfully constructed the binary Goppa code including the generator matrix and the parity check matrix. This means that we construct a message and the respective code word.

In [ ]:
```
# This is a helper function to create a message
# of length k
# for niederreiter set the function parameter k to n,
# so the message is extended to length n
def to_lengthk_message(b, k, verbose):
    cur_len = len(b)
    assert cur_len <= k, f'''the list {b} of bits you defined
                             is longer than {k}.
                             This should be {k} or less'''
    b.extend([0] * (k - cur_len))
    message = matrix(GF(2), b)
    if verbose:
        print(f'This is the message of length {k}'
              f' that will be encoded')
        latexprint(latex(message))
    return message

def encode_message(message, matrix, verbose,
                   method='mceliece'):
    if method == 'niederreiter':
        codeword = matrix * message.transpose()
        if verbose:
            print("This is the encrypted message.")
            latexprint(latex(codeword))
        return codeword
    else: #mceliece
        codeword = message * matrix
        if verbose:
            print('''This is the respective code word
            of length {len(codeword}.
            Note that the last {k} bits of the code word
            are the message''')
            latexprint(latex(codeword))
        return codeword

def check_syndrome(vector, parity_check_matrix_f2_standard_form,
                   verbose, note=''):
    # helper function for debugging
    syndrome =  parity_check_matrix_f2_standard_form * vector.transpose()
    if verbose > 1:
        print(f'Syndrome of {note}')
        latexprint(latex(syndrome.transpose()))
    return syndrome
```

Now we will introduce an error vector and add it to our code word. This results in an erroneous word w.

$$w = m \cdot G + e$$

In [ ]:
```
def hamming_weight(v):
    return len(v.coefficients())

# This is a helper function to
# easily create an error vector
def create_errorvector(n, t, verbose, method='first'):
    if method == 'dummy':
        # this method is onnly kept because
        # it was in original code
        # probability of error depends on position
        # early positions more likely to contain error
        # first few position 50/50, later positions
        # probability zero after t errors
        result = []
        counter = 0
        for i in range(n):
            rand = randint(0,1)
            if rand == 1:
                if counter < t:
                    counter += 1
                else:
                    rand = 0
            result.append(rand)
    elif method == 'uniformally':
        # each position has equal probability
        # to contain error
        selected_positions = random.sample(range(n), t)
        result = [1 if i in selected_positions else 0
                  for i in range(n)]
    elif type(method) == list:
        print('error type list')
        assert len(method)== n, '''given error does
                                   not have length n'''
        result = method
    else: # first
        # first t positions contain error
        result = [1]*t + [0]*(n-t)
    errorvector = matrix(GF(2), result)
    assert hamming_weight(errorvector) <= t, '''You introduced
                                                too many errors'''
    if verbose:
        print(f'This is the error vector of length {n}'
              f' and weight {hamming_weight(errorvector)}'
              f' we will introduce')
        latexprint(latex(errorvector))
    return errorvector
```

In [ ]:
```
def add_error(codeword, errorvector, verbose):
    w = codeword + errorvector
    if verbose == 1:
        print(f'codeword')
        latexprint(latex(w))
    if verbose > 1:
        print(f'codeword      {latexprint(latex(codeword))}\n'
              f'errorvector   {latexprint(latex(codeword))}\n'
              f'send vector {latexprint(latex(codeword))}\n')
    return w
```

Now comes the trickiest part. We want to decode the erroneous word and retrieve the message. As the paper showed, we know that

$$S_w(x,z) = \sum_{i=0}^{n-1} \frac{c_i + e_i}{x - \sigma_i(z)} = \sum_{i=0}^{n-1} \frac{c_i}{x - \sigma_i(z)} + \sum_{i=0}^{n-1} \frac{e_i}{x - \sigma_i(z)} \equiv 0 + \sum_{i=0}^{n-1} \frac{e_i}{x - \sigma_i(z)} \mod g(x,z) \tag{2}$$

so let's reconstruct that first by looking at the bits in w. Note that we need knowledge of our support σ and the modulus $g(x,z)$ to do this.

In [ ]:
```
def construct_Sw(word,sigma, g, K):
    G.<x> = K.extension(g)
    w_list = word.list()
    #here we reconstruct (c_i + e_i)/(x - sigma_i)
    return sum([entry * (x - sigma[index])^-1
                for index, entry in enumerate(w_list)])

def compare_syndrom_polynomial(Sw, errorvector, sigma, g, K):
    # only helper function for debugging and additinal explanation
    Se = construct_Sw(errorvector, sigma, g, K)
    latexprint(r"\text{This is } S_w(x,z) \text{. For comparison,  }")
    latexprint(r"S_e(x, z) \text{ of the error vector is ")
    latexprint(r"printed below it to show they are equal}")
    latexprint(latex(Sw))
    latexprint(latex(Se))
```

Now that we have $S_w(x,z)$, we can calculate $V(x,z)$ such that

$$V(x,z) = \sqrt{x + \frac{1}{S_w(x,z)}}$$

Note that for any element with coefficients c_i in our field $\mathbb{F}_{2^m}[x]$,

$$(c_{t-1}x^{t-1} + \cdots + c_1 x + c_0)^2 = c_{t-1}^2 x^{2(t-1)} + \cdots + c_1^2 x^2 + c_0^2$$

Therefore, the square root of an element is

$$\sqrt{c_{t-1}x^{t-1} + \cdots + c_1 x + c_0} = \sqrt{c_t}\sqrt{x^{t-1}} + \cdots + \sqrt{c_1}\sqrt{x} + \sqrt{c_0}$$

We use this observation to circumvent a limitation of Sage, namely that Sage does not support square roots of elements in our field $\mathbb{F}_{2^m}[x]/g(x,z)$.

```
In [ ]: def calculate_v(g, K, t, Sw, verbose):
            #This helper field with indeterminate b is used
            # to check for all x's if they are a square,
            #as this functionality is not available for
            # F_{2^m}[x]/g(x,z), but is available for F_{2^t}
            G.<x> = K.extension(g)
            _.<b> = GF(2)[]
            SqrtField.<b> = GF(2^t, modulus = g(b))

            #This helper function checks for all pairs of
            # C_i*x^i if c_i is a square and x^i is a square,
            #because only then c_i*x^i is a square
            def is_square_f2mx(el):
                coefs = list(el)
                result = True
                for index,coef in enumerate(coefs):
                    if coef != 0:
                        result = result and (b^index).is_square() and coef.is_square()
                return result

            #This helper function converts an element from helper
            # field SqrtField back to F_{2^m}[x]/g(x)
            def b_to_x(element):
                xcoefs = element.polynomial().coefficients(sparse = False)
                xvalue = 0
                for exponent,xcoef in enumerate(xcoefs):
                    if xcoef != 0:
                        xvalue += x^exponent
                return xvalue

            #Using the observation from the previous cell,
            # this function calculates the square root
            # of an element in F_{2^m}[x]
            #by taking the square root of each coefficient c_i
            # and each x^i separately
            # and then multiplying them afterwards
            def sqrt_f2mx(element, Sw):
                assert is_square_f2mx(element), "not a square"
                result = 0
                for index,coef in enumerate(list(element)):
                    if coef != 0:
                        bsquare_for_x = sqrt(b^index)
                        xsquare = b_to_x(bsquare_for_x)
                        coefsqrt = sqrt(coef)
                        result += coefsqrt*xsquare
                return result
            v = sqrt_f2mx(x + Sw^-1, Sw)
            if verbose:
```

```
        latexprint(r"\text{This is our polynomial }V(x,z)")
        latexprint(latex(v))
    return v
```

Now that we have $V(x,z)$, we can apply the extended Euclidean algorithm to get an equation of the form

$$A(x,z) = B(x,z)V(x,z) + H(x,z)g(x) \tag{3}$$

Here $A(x,z)$ and $B(x,z)$ have the following requirements.

$$\text{the degree of } A(x,z) \leq \left\lfloor \frac{t}{2} \right\rfloor \text{ and the degree of } B(x,z) \leq \left\lfloor \frac{t-1}{2} \right\rfloor \tag{4}$$

Then the error locator polynomial is

$$\epsilon(x,z) = \prod_{i,e_i\neq 0} (x - \sigma_i(z)) = A(x,z)^2 + x * B(x,z)^2 \tag{5}$$

For an elaborate explanation, please consult the respective white paper that was mentioned in the introduction.

In [ ]:
```
#This helper function returns the degree of an element
def degree_poly(el):
    as_list = list(el)
    return len(as_list) - 1

def calculate_error_locator_polynomial(g,
                                       t,
                                       K,
                                       v,
                                       verbose):
    #The indeterminate h is used to denote a field
    # that has modulus f(z)
    # but not modulus g(x), also know as F_{2^m}[x],
    #or in this case, F_{2^m}[h],
    # because we shouldn't reduce mod g(x,z)
    # during the extended Euclidean algorithm
    G.<x> = K.extension(g)
    _.<h> = K[]

    #This helper function converts an element from
    # F_{2^m}[x]/g(x) to the helper field
    def hpoly(el):
        as_list = list(x - x + el)
        return sum([h^exponent * entry
                    for exponent,entry in enumerate(as_list)])

    def error_locator_polynomial(v_xz):
        a_degree = floor(t/2)
        b_degree = floor((t-1)/2)
        v_poly =  hpoly(v_xz)
        g_poly = g(h)
        remainder = [g_poly, v_poly]
        v_mult = [0, 1]
        g_mult = [1, 0]
        a_hz = v_poly
        #adding and subtracting h to force class type
        b_hz = h - h + 1
        index = 2
        while not (degree_poly(a_hz) <= a_degree and degree_poly(b_hz) <= b_degree):
            a_hz = remainder[index - 2] % remainder[index - 1]
            q = (remainder[index - 2] - a_hz) / remainder[index - 1]
```

```
                #sometimes q has a fraction class,
                # so this makes sure we can extract coefficients
                q = q.numerator()
                remainder.append(a_hz)
                v_mult.append(v_mult[index - 2] - q * v_mult[index - 1])
                g_mult.append(g_mult[index - 2] - q * g_mult[index - 1])
                b_hz = v_mult[index]
                index = index + 1
            index = index - 1
            result = a_hz^2 + h*(b_hz^2)
            return result
        e_hz = error_locator_polynomial(v)
        if verbose:
            print(f'The error locator polynomial is \n'
                  f'-----------------------------------')
            latexprint(latex(e_hz))
            print(f'note that the degree of the error locator polynomial is,'
                  f'equal to the number of errors {t} we introduced')
        return e_hz
```

Now that we have constructed the error locator polynomial, we can reconstruct the error vector that was added to the code word. We do this by iterating over all the elements of our support σ and checking whether $\epsilon(\sigma_i(z), z) = 0$. For all $\sigma_i(z)$ that this is true, we know that index i of the error vector was 1. Otherwise it was 0. Once we have reconstructed the error vector, we can subtract it from the erroneous word to retrieve the code word and extract the message.

In [ ]:
```
def reconstruct_error_vector(error_polynomial,
                             sigma,
                             verbose):
    result_list = []
    for element in sigma:
        if error_polynomial(element) == 0:
            result_list.append(1)
        else:
            result_list.append(0)
    reconstructed_errorvector = matrix(GF(2), result_list)
    if verbose:
        # error vector comparison
        print(f'\nerror comparison \n'
              f'-----------------------------')
        print("reconstructed error vector:")
        latexprint(latex(reconstructed_errorvector))
    return reconstructed_errorvector

def reconstruct_send_message(w, reconstructed_errorvector,
                             k, verbose):
    reconstructed_codeword = w - reconstructed_errorvector
    retrieved_message = (matrix(GF(2),
                        reconstructed_codeword.list()[-k:]))
    if verbose:
    # send word comparison
        print(f'\nsend vs reconstructed code word:\n'
              f'----------------------------')
        print('reconstructed code word:')
        latexprint(latex(reconstructed_codeword))

        print(f'\nreconstruction of message\n'
              f'----------------------------')
        print(f'the message is therefore the last {k}'
              f' bits of the reconstructed code word')
        latexprint(latex(retrieved_message))
    return retrieved_message
```

sometimes the parity check matrix is not able to be converted to systematic form then a different random support is chosen to make the matrix convertable

this procedure is repeated max 'num_entries' times

a warning is given if the initial method for the support was not random

```
In [ ]: def repeat_choosing_support( num_retries,
                                     g,
                                     t,
                                     K,
                                     n,
                                     m,
                                     k,
                                     verbose,
                                     method):
            # method: 'first_n', 'random_n', list of length n with indizes
            sigma = choose_support(K, n, m, verbose, method)
            for attempt_no in range(num_retries):
                try:
                    parity_check_matrix_f2_standard_form = parity_check_matrix_standard_form(
                        g, t, K, n, sigma, m, k, verbose
                    )
                    return sigma, parity_check_matrix_f2_standard_form
                except AssertionError as error:
                    if method != 'random':
                        warnings.warn(f'Parity check matrix not'
                                      f'convertable to standard form'
                                      f'- choose different support randomly')
                    if attempt_no < (num_retries - 1):
                        if verbose:
                            print(f'Error: Could not convert parity check'
                                  f'matrix to standard form\n'
                                  f'choose different random support')
                        sigma = choose_support(K, n, m, verbose, 'random_n')
                    else:
                        raise error
            return sigma, parity_check_matrix_f2_standard_form
```

Now that we have successfully created and used the binary Goppa code, we can turn it into the McEliece cryptographic system. Recall from the paper that to hide any code, we multiply the generator matrix by a random invertible matrix and permute it. The cipher text T is then

$$T = mSGP + e$$

where m is the message, S is a random invertible matrix, G is the generator matrix. P is the permutation matrix and e is the error vector. However, this was the generalised approach for *all* codes. We can use the properties of the binary Goppa code to achieve this without much effort. First of all, note that we can permute rows by permuting our support σ, as the order of the elements in the support defines the order of the columns in the parity check matrix. This means that we can simply take a random permutation of the support elements to get this part done. However, if $n < 2^m$ we will give away information on the original support if we simply permute the elements we chose. Therefore, it is best to randomly permute the entire field $\mathbb{F}_{2^m}$ and then take the first n elements again.

Secondly, we need to incorporate a random invertible matrix. This is implicitly done when we convert the parity check matrix to *systematic form*.

For the McEliece system in the optimized version the public key is

- The new generator matrix

- the error correction capability t

And the private key is

- the permuted support σ
- the modulus $g(x, z)$

Note that the generator matrix always contains an identity matrix (either on the left or the right). Our public key is therefore everything but the identity matrix.

McEliece crypto system optimized

In [ ]:
```
def mc_eliece_key_gen(m, t, verbose,
                      choose_g='random_binary',
                      choose_support='random_n'):
    # define Goppa Code, calculate generator
    # and parity check matrix
    # choose g: 'random_binary', 'first_binary',
    #           Integer i (choose i-th. binary irreducible polynomial)
    # choose support: # 'first_n',
    #                  'random_n', list of length n with indizes
    # #---------------------
    n = 2^m
    k = n - t*m
    check_parameters(m, t, n, k, verbose)
    K = create_F_2m(m, verbose)
    # 'random_binary', 'first_binary', ('random' does not work)
    g = create_F_2m_mod_g(K, t, verbose, choose_g)
    sigma, parity_check_matrix_f2_standard_form = repeat_choosing_support(
        8, g, t, K, n, m, k, verbose, choose_support )
    generator_matrix = parity_to_generator(parity_check_matrix_f2_standard_form,
                                           k,
                                           verbose)
    # store only part without identity matrix
    k_pub = (generator_matrix[:, :(n-k)], t, m, k, n)
    k_priv = (sigma, g, K)
    if verbose:
        if verbose > 1:
            print(f'generator matrix print')
            latexprint(latex(generator_matrix))
            print(f'stored subset of generator matrix')
            latexprint(latex(generator_matrix[:, :n-k]))
        print(f'k_pub:\n-----------')
        latexprint(latex(k_pub))
        print(f'k_priv:\n-----------')
        latexprint(latex(k_priv))
    return k_pub, k_priv
```

In [ ]:
```
def mc_eliece_encrypt(short_message,
                      k_pub,
                      verbose2,
                      choose_error='uniformally' ):
    # choose_error='uniformally', 'dummy',
    #              'first', list with error of length n
    verbose = False
    generator_matrix_without_identity_part, t, m, k, n = [elem for elem in k_pub]
    generator_matrix = generator_matrix_without_identity_part.augment(
        identity_matrix(k)
    )
    message = to_lengthk_message(short_message, k, verbose)
    codeword = encode_message(message,
                              generator_matrix,
                              verbose,
                              'mceliece')
    # 'dummy', 'uniformally', 'first'
    errorvector = create_errorvector(n,
                                     t,
                                     verbose,
                                     choose_error)
    w = add_error(codeword, errorvector, verbose)
    if verbose2:
        if verbose2 > 1:
            print(f't = {t}, m = {m}, k = {k}, n = {n}')
            print('generator matrix without identity part')
            latexprint(latex(generator_matrix_without_identity_part))
        print('generator matrix with identity part')
        latexprint(latex(generator_matrix))
        print('message')
        latexprint(latex(message))
        print('codeword')
        latexprint(latex(codeword))
        print('errorvector')
        latexprint(latex(errorvector))
        print('encrypted message')
        latexprint(latex(w))
    return w
```

In [ ]:
```
def mc_eliece_decrypt(vector, k_priv, verbose):
    sigma, g, K = [elem for elem in k_priv]
    t = g.degree()
    m = Integer(math.log(len(K),2))
    n = len(sigma)
    k = n - t*m # n - tm
    G.<x> = K.extension(g)
    Sw = construct_Sw(vector, sigma, g, K)
    v = calculate_v(g, K, t, Sw, verbose)
    e_hz = calculate_error_locator_polynomial(g,
                                              t,
                                              K,
                                              v,
                                              verbose)
    reconstructed_errorvector = reconstruct_error_vector(e_hz,
                                                         sigma,
                                                         verbose)
    decoded_message = reconstruct_send_message(vector,
                                               reconstructed_errorvector,
                                               k,
                                               verbose)
    return decoded_message
```

Encrypt and Decrypt with McEliece

here you can actually test the encryption and decryption of messages

In [ ]:
```
verbose = 1

# choose g: 'random_binary', 'first_binary',
#           Integer i (choose i-th. binary irreducible polynomial)
# choose support: # 'first_n', 'random_n',
#                   list of length n with indizes
k_pub, k_priv = mc_eliece_key_gen(4,
                                  3,
                                  verbose,
                                  'random_binary',
                                  'random_n')
```

```
message = [1,1]
# choose_error='uniformally', 'dummy', 'first',
#                list with error of length n
encrypted_message = mc_eliece_encrypt(message,
                                      k_pub,
                                      verbose,
                                      'uniformally')
decrypted_message = mc_eliece_decrypt(encrypted_message,
                                      k_priv,
                                      verbose)
```

Niederreiter crypto system optimized

analog to the McEliece system the niederreiter version is implemented

In [ ]:
```
def niederreiter_key_gen(m, t, verbose,
                         choose_g='random_binary',
                         choose_support='random_n'):
    # define Goppa Code, calculate generator
    # and parity check matrix
    # choose g: 'random_binary', 'first_binary',
    #           Integer i (choose i-th. binary irreducible polynomial)
    # choose support: # 'first_n', 'random_n',
    #                   list of length n with indizes
    # #-------------------------------------------------------------------
    n = 2^m
    k = n - t*m
    check_parameters(m, t, n, k, verbose)
    K = create_F_2m(m, verbose)
    # 'random_binary', 'first_binary', ('random' does not work)
    g = create_F_2m_mod_g(K, t, verbose, choose_g)
    sigma, parity_check_matrix_f2_standard_form = repeat_choosing_support(
        12, g, t, K, n, m, k, verbose, choose_support
    )
    # store only part without identity matrix
    k_pub = (parity_check_matrix_f2_standard_form[:, -k:], t, m, k, n)
    k_priv = (sigma, g, K)
    if verbose:
        if verbose > 1:
            print(f'parity check matrix print')
            latexprint(latex(parity_check_matrix_f2_standard_form))
            print(f'stored subset of parity check matrix')
            latexprint(latex(parity_check_matrix_f2_standard_form[:, -k:]))
        print(f'k_pub:\n-----------')
        latexprint(latex(k_pub))
        print(f'k_priv:\n-----------')
        latexprint(latex(k_priv))
    return k_pub, k_priv
```

In [ ]:
```
def niederreiter_encrypt(short_message,
                         k_pub,
                         verbose2):
    verbose = False
    parity_check_matrix_without_identity_part, t, m, k, n = [elem for elem in k_pub]
    parity_check_matrix = identity_matrix(n-k).augment(
        parity_check_matrix_without_identity_part
    )
    message = to_lengthk_message(short_message,
                                 n,
                                 verbose)
    assert hamming_weight(message) <=t, 'Message does not have weight <= t'
    codeword = encode_message(message,
                              parity_check_matrix,
                                verbose,
                                'niederreiter')
    w = codeword
    if verbose2:
        if verbose2 > 1:
            print(f't = {t}, m = {m}, k = {k}, n = {n}')
            print('parity check matrix without identity part')
            latexprint(latex(parity_check_matrix_without_identity_part))
        print('parity check matrix with identity part')
        latexprint(latex(parity_check_matrix))
        print('message')
        latexprint(latex(message))
        print('encrypted message')
        latexprint(latex(w.transpose()))
    return w
```

In [ ]:
```
def niederreiter_decrypt(vector, k_priv, verbose):
    sigma, g, K = [elem for elem in k_priv]
    t = g.degree()
    m = Integer(math.log(len(K),2))
    n = len(sigma)
    k = n - t*m # n - tm
    G.<x> = K.extension(g)
    Sw = construct_Sw(vector, sigma, g, K)
    v = calculate_v(g, K, t, Sw, verbose)
    e_hz = calculate_error_locator_polynomial(g,
                                              t,
                                              K,
                                              v,
                                              verbose)
    reconstructed_errorvector = reconstruct_error_vector(e_hz,
                                                         sigma,
                                                         verbose)

    if verbose:
        print(f'\nsend vs reconstructed code word:\n'
              f'----------------------------')
        print('reconstructed message:')
        latexprint(latex(reconstructed_errorvector))
    return reconstructed_errorvector
```

Encrypt and Decrypt with Niederreiter

In [ ]:
```
verbose = 1
# choose g: 'random_binary', 'first_binary',
#           Integer i (choose i-th. binary irreducible polynomial)
# choose support: # 'first_n', 'random_n', list of length n with indizes
k_pub, k_priv = niederreiter_key_gen(4,
                                     3,
                                     verbose,
                                     'random_binary',
                                     'random_n')
message = [0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,0,0]
encrypted_message = niederreiter_encrypt(message,
                                         k_pub,
                                         verbose)
decrypted_message = niederreiter_decrypt(encrypted_message,
                                         k_priv,
                                         verbose)
```

create reproducable deterministic example

for reproducability a deterministic example is created to be published a deterministic example also allows to compare the systems for different support, goppa polynomials or messages

McEliece

In [ ]:
```
verbose = 2

# choose g: 'random_binary', 'first_binary',
#            Integer i (choose i-th. binary irreducible polynomial)
# choose support: # 'first_n',
#                   'random_n',
#                    list of length n with indizes
k_pub, k_priv = mc_eliece_key_gen(4,
                                  3,
                                  verbose,
                                  0,
                                  [i for i in range(2^4)])
message = [1,1,1]
# choose_error='uniformally', 'dummy',
#               'first', list with error of length n
encrypted_message = mc_eliece_encrypt(message,
                                      k_pub,
                                      verbose,
                                      [1,1,0,0,0,0,1,0,0,0,0,0,0,0,0,0])
decrypted_message = mc_eliece_decrypt(encrypted_message,
                                      k_priv,
                                      verbose)
```

McEliece change support order and review changes

In [ ]:
```
verbose = 2

# choose g: 'random_binary', 'first_binary',
#            Integer i (choose i-th. binary irreducible polynomial)
# choose support: # 'first_n', 'random_n',
#                     list of length n with indizes

# change order of support
k_pub, k_priv = mc_eliece_key_gen(4,
                                  3,
                                  verbose,
                                  0,
                                  [1,0,4,2,5,3] + [i for i in range(6,2^4)])
message = [1,1,1]
# choose_error='uniformally', 'dummy', 'first',
#                list with error of length n
encrypted_message = mc_eliece_encrypt(message,
                                      k_pub,
                                      verbose,
                                      [1,1,0,0,0,0,1,0,0,0,0,0,0,0,0,0])
decrypted_message = mc_eliece_decrypt(encrypted_message,
                                      k_priv,
                                      verbose)
```

Niederreiter

In [ ]:
```
verbose = 2
# choose g: 'random_binary', 'first_binary',
#            Integer i (choose i-th. binary irreducible polynomial)
# choose support: # 'first_n', 'random_n',
#                   list of length n with indizes
k_pub, k_priv = niederreiter_key_gen(4, 3,
                                                       verbose,
                                                       0,
                                                       [i for i in range(2^4)])
message = [0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,0]
encrypted_message = niederreiter_encrypt(message,
                                          k_pub,
                                          verbose)
decrypted_message = niederreiter_decrypt(encrypted_message,
                                          k_priv,
                                          verbose)
```

Demonstration that the McEliece schoolbook version is not CCA-2 secure

not CCA-2 secure

attacker receives r and knows G' $r = mG' + e$ create code word $c = m_2G'$ with arbitrary m_2 ask oracle for decryption of $r + c$ $Dec(r + c) = Dec(mG' + e + m_2G') = m + m_2 := m_{dec}$
$m = m_{dec} - m_2$

In [ ]:
```
verbose = 0
# alice sends her message
k_pub, k_priv = mc_eliece_key_gen(4,
                                  3,
                                  verbose,
                                  'random_binary',
                                  'random_n')
m = [1,1,1,0]
r = mc_eliece_encrypt(m,
                      k_pub,
                      verbose,
                      'uniformally')

# carol creates m'
m_2 = [1,0,0,0]
c = mc_eliece_encrypt(m_2,
                      k_pub,
                      verbose,
                      [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0])


#oracle decrypts r+c
m_dec = mc_eliece_decrypt(r+c,
                          k_priv,
                          verbose)
attackers_m = m_dec - matrix(GF(2), m_2)
print(f'message was successfully reconstructed\nsend:'
      f'          {list(m)}\nreconstructed: {attackers_m}')
```