## University of Toronto, Department of Electrical and Computer Engineering

## ECE 1501 — Error Control Codes

1. 10
2. 10
3. 10
4. 8

total: 38/40

# Exercise 2: Reed–Solomon Codes


### Last name: Fei

### First name: Yue

### Student number: 1003944146


The purpose of this Numerical Exercise is to implement Reed–Solomon codes and some of their subfield subcodes. There are five parts to this numerical exercise:

0. **Polynomials:** You will learn how to work with polynomials over finite fields in julia. This exercise will not be graded, but it will be important to solve as it will be needed for following exercises!
1. **Reed–Solomon Codes and Berlekamp–Welch Decoder:** You will implement basic encoding and decoding methods for Reed–Solomon Codes as polynomial codes.
2. **Extended Euclidean Algorithm:** You will implement the generalized Euclidean algorithm and, as an application, you will find the multiplicative inverse of nonzero elements in large finite fields.
3. **Generalized Reed–Solomon Codes:** You will learn about Generalized Reed–Solomon Codes and implement an efficient decoder for them.
4. **Bose–Chaudhuri–Hocquenghem codes:** You will learn how to decode BCH codes by considering them as subfield subcodes of Generalized Reed–Solomon codes.

Each exercise is prefaced by some introductory remarks to help you complete that exercise. As always, feel free to post your questions and comments on piazza.

Let's get started!

# 0. Polynomials

Let $R$ be any ring, and let $x$ be an **indeterminate** (a **variable**). A **polynomial** is a formal expression of the form $$a(x) = a_0 x^0 + a_1 x^1 + \cdots + a_n x^n = \sum_{i=0}^n a_i x^i,$$ where $n$ is any natural number. Here $a_0$, $a_1$, $\ldots$, $a_n$ are elements of $R$, called the **coefficients** of $a(x)$. Each $a_i x^i$ is called a **term** of $a(x)$. The set of all polynomials in $x$ with coefficients from $R$ is denoted as $R[x]$. A term of the form $1 \cdot x^i$ is usually written as $x^i$ and terms with zero coefficient are often omitted. Likewise the terms $a_0 x^0$ and $a_1 x^1$ are written as $a_0$ and $a_1 x$, respectively. Thus $1 + 2x + x^3$ corresponds to the formal expression $1x^0 + 2x^1 + 0 x^2 + 1 x^3$.

Polynomials can be added by adding the coefficients of like terms (where missing terms are treated as having zero coefficient). The **zero polynomial**, i.e., the polynomial all of whose coefficients are zero, serves as the additive identity. Polynomials can be multiplied by assuming that the distributive law holds, simplifying terms of the form $(a_i x^i)\cdot (b_j x^j)$ to $a_i b_j x^{i+j}$. The polynomial $1 = 1x^0$ serves as the multiplicative identity. Under these operations, $R[x]$ itself forms a ring.

Since $R[x]$ is itself a ring, it is possible to form a ring $(R[x])[y]$ of polynomials in indeterminate $y$ having elements of $R[x]$ as coefficients. One might also consider the ring $(R[y])[x]$ of polynomials in interminate $x$ have elements of $R[y]$ as coefficients. There is an obvious correspondence between these rings, either one of which is usually denoted as $R[x,y]$. The elements of $R[x,y]$ are referred to as **bivariate** polynomials (to distinguish them from the **univariate** polynomials of $R[x]$). One can extend this idea in the obvious way to form the **trivariate** ring $R[x,y,z]$, or, for any positive integer $m$, to form the **multivariate** ring $R[x_1,x_2,\ldots,x_m]$.

The **degree** of $a(x) = a_0 + a_1 x + \cdots + a_n x^n \in R[x]$, denoted as $\deg(a)$, is the largest index $i$ for which $a_i\neq 0$. This definition does not work for the zero polynomial (since no such index exists), thus by convention the degree of the zero polynomial is defined to be $-\infty$. Under this convention, provided that $R$ is an integral domain, we have $$\deg(pq) = \deg(p) + \deg(q)$$ for all polynomials $p(x), q(x) \in R[x]$. (Caution: addition of degrees doesn't generally hold in $R[x]$ when $R$ is a ring having zero divisors!) A degree $d$ polynomial $a(x) = a_0 + a_1 x + \cdots + a_d x^d \in R[x]$ is said to be **monic** if $a_d = 1$.

In $R[x,y]$ we can view a bivariate polynomial as an element of $(R[y])[x]$ and assign it a degree (called its $x$-degree). Alternatively we can view it as an element of $(R[x])[y]$ and assign it a $y$-degree. For example, consider $a(x,y) = a_0 + 2x + y + 3xy + xy^2$. This polynomial can be written as $(a_0 + y) + (2 + 3y + y^2 )x \in (R[y])[x]$, from which we see that it has $x$-degree equal to one. Alternatively, $a(x)$ can be written as $(a_0 + 2x) + (1 + 3x)y + (x)y^2 \in (R[x])[y]$, from which we see that it has $y$-degree equal to two.

In this numerical exercise, we will focus on $\mathbb{F}_q[x]$, i.e., on univariate polynomials with coefficients from the finite field $\mathbb{F}_q$. In the following, when we refer to a polynomial, we mean a univariate polynomial, unless clearly stated otherwise. (Bivariate polynomials will appear in the context of the Berlekamp–Welch decoding procedure.)

Associated with a polynomial $p(x)\in\mathbb{F}_q[x]$ is a function defined by evaluation: $$\hat{p}:\mathbb{F}_q\to \mathbb{F}_q$$ $$\alpha\mapsto p(\alpha).$$

For simplicity, we use $p$ instead of $\hat{p}$ when talking about the function obtained by evaluation of the polynomial $p(x)$. This is an abuse of notation as these are essentially different objects. For example, consider the polynomials $p_1(x), p_2(x)\in\mathbb{F}_2[x]$ defined by $p_1(x) = x$ and $p_2(x) = x^2$. These are two distinct polynomials while the functions $\hat{p}_1$ and $\hat{p}_2$ are the same! If the evaluation of a polynomial $p(x)$ at a point $\alpha$ is zero, then $\alpha$ is called a **root** or **zero** of $p(x)$.

In software, a polynomial of degree $d$ can be represented with a vector of length $d+1$. In particular, there is a `Polynomials` package in julia that uses this form of representation and allows us to perform basic polynomial operations. Run the following cell to load the `Galois2` module and install (if needed) and load the `Polynomials` package.

In [1]:
```
include("Galois2.jl"); using .Galois2
```

In [2]:
```
using Pkg; Pkg.add("Polynomials")     # uncomment if you need to install Polynomial
using Polynomials
```

```
   Updating registry at `C:\Users\yuefei\.julia\registries\General`
   Updating git-repo `https://github.com/JuliaRegistries/General.git`
  Resolving package versions...
 No Changes to `C:\Users\yuefei\.julia\environments\v1.8\Project.toml`
 No Changes to `C:\Users\yuefei\.julia\environments\v1.8\Manifest.toml`
```

A polynomial can be constructed by calling the function `Polynomial` from its vector of coefficients, lowest order first.

In [3]:
```
a = GF2[1, 0, 1, 1]
p = Polynomial(a)
```

Out[3]: $1 + 1\cdot x^2 + 1\cdot x^3$

You can also provide the variable for the polynomial. The default variable is `x`:

In [4]:
```
q = Polynomial(a, :x)
p == q
```

Out[4]: `true`

In [5]:
```
r = Polynomial(a, :y)
```

Out[5]: $1 + 1\cdot y^2 + 1\cdot y^3$

In [6]:
```
p == r
```

Out[6]: false

The evaluation of a polynomial can be performed by simply treating it as a function:

In [7]:
```
p(GF2(0))
```

Out[7]: 1

In [8]:
```
p(GF2(1))
```

Out[8]: 1

Let's create a primitive element in $\mathbb{F}_{2^8}$:

In [9]:
```
α = gfprimitive(8) # (to type this, use \alpha+TAB)
```

Out[9]: 2

In [10]:
```
typeof(α)
```

Out[10]: Gf2_8

In [11]:
```
gforder(α) # a primitive element in Fq should have multiplicative order q-1
```

Out[11]: 255

In [12]:
```
isprimitive(α)
```

Out[12]: true

In [13]:
```
isprimitive(α^3) # on the other hand, any power of alpha not relatively prime to q-
```

Out[13]: false

In [14]:
```
gforder(α^3)
```

Out[14]: 85

In the following cells, some basic functionality provided by `Polynomials` is illustrated.

In [15]:
```
# Create a polynomial from its roots
x = Polynomial(GF256[0,1])  # this gives the monomial x
roots = [α,α^2,α^3,α^4,α^5]
p = prod((x - r) for r in roots)
```

Out[15]: $38 + 197 \cdot x + 229 \cdot x^2 + 63 \cdot x^3 + 62 \cdot x^4 + 1 \cdot x^5$

In [16]:
```
degree(p) # compute the degree of p
```

Out[16]: 5

**Caution:** The package `Polynomials` has a different convention for defining the degree of the zero polynomial and sets $\deg(0) = -1$. This way, the identity $$ \deg(pq) = \deg(p) + \deg(q) $$ only holds when both $p(x)$ and $q(x)$ are nonzero. In case you need to use `degree` function, beware of this convention.

In [17]:
```
degree(Polynomial(zero(GF256)))
```

Out[17]:
```
-1
```

In [18]:
```
cp = coeffs(p)  # extract the coefficients of p into a vector
```

Out[18]:
```
6-element Vector{Gf2_8}:
  38
 197
 229
  63
  62
   1
```

In [19]:
```
x = cp[1]
cp[1] = zero(GF256) # Caution: modifying the coefficient vector..
p  # modifies the polynomial!
```

Out[19]: $197 \cdot x + 229 \cdot x^2 + 63 \cdot x^3 + 62 \cdot x^4 + 1 \cdot x^5$

In [20]:
```
cp[1] = x # restore the polynomial
p
```

Out[20]: $38 + 197 \cdot x + 229 \cdot x^2 + 63 \cdot x^3 + 62 \cdot x^4 + 1 \cdot x^5$

In [21]:
```
p(α)  # polynomial evaluation
```

Out[21]:
```
0
```

In [22]:
```
p(one(GF256))  # polynomial evaluation
```

Out[22]:
```
6
```

In [23]:
```
typeof(p(one(GF256)))  # be careful to note that the value is *not* an integer!
```

Out[23]:
```
Gf2_8
```

In [24]:
```
gflog(p(one(GF256))) # compute the discrete log of the previous result
```

Out[24]:
```
26
```

In [25]:
```
α^(gflog(p(one(GF256))))  # exp(log()) should be an identity function on nonzero el
```

Out[25]:
```
6
```

In [26]:
```
q = x - α^6
```

```
p * q    # polynomial multiplication
```

Out[26]: $21 + 54\cdot x + 106\cdot x^2 + 74\cdot x^3 + 44\cdot x^4 + 102\cdot x^5$

In [27]:
```
g = Polynomial(GF2[1,1,0,1])  # a generator polynomial for the (7,4) cyclic Hamming
u = Polynomial(GF2[1,0,0,1])  # a message polynomial
x = Polynomial(GF2[0,1])      # the monomial x
q = div(x^3 * u, g)           # find the quotient of x^3 * u and g
```

Out[27]: $1\cdot x + 1\cdot x^3$

In [28]:
```
r = rem(x^3 * u, g)         # find the remainder after dividing x^3* u
```

Out[28]: $1\cdot x + 1\cdot x^2$

In [29]:
```
q*g + r  # should be x^3 * u
```

Out[29]: $1\cdot x^3 + 1\cdot x^6$

In [30]:
```
v = x^3 * u - r  # form a Hamming codeword with message bits in higher order positi
```

Out[30]: $1\cdot x + 1\cdot x^2 + 1\cdot x^3 + 1\cdot x^6$

In [31]:
```
coeffs(v)  # extract the codeword symbols
```

Out[31]:
```
7-element Vector{Gf2_1}:
 0
 1
 1
 1
 0
 0
 1
```

In [32]:
```
rem(v,g) == Polynomial(zero(GF2))  # let's check if this is a valid codeword.  If s
```

Out[32]:
```
true
```

** CAUTION ** The implementors of the `Polynomials` package sometimes treat constant polynomials (polynomials of degree zero) as scalars, but they make an incorrect assumption about the underlying field. For example, try running the following cell.

In [33]:
```
rem(v,Polynomial(one(GF2)))  # divide v(x) by one
```

```
MethodError: no method matching real(::Gf2_1)
Closest candidates are:
  real(::LinearAlgebra.UnitUpperTriangular{var"#s885", S} where {var"#s885"<:Real, S
<:AbstractMatrix{var"#s885"}}) at D:\ProgramFiles\Julia-1.8.5\share\julia\stdlib\v1.
8\LinearAlgebra\src\triangular.jl:50
  real(::LinearAlgebra.UnitUpperTriangular{var"#s884", S} where {var"#s884"<:Comple
x, S<:AbstractMatrix{var"#s884"}}) at D:\ProgramFiles\Julia-1.8.5\share\julia\stdlib
\v1.8\LinearAlgebra\src\triangular.jl:51
  real(::LinearAlgebra.Diagonal) at D:\ProgramFiles\Julia-1.8.5\share\julia\stdlib\v
1.8\LinearAlgebra\src\diagonal.jl:151
  ...

Stacktrace:
 [1] real(T::Type)
   @ Base .\complex.jl:120
 [2] rtoldefault(x::Gf2_1, y::Int64, atol::Int64)
   @ Base .\floatfuncs.jl:330
 [3] isapprox(x::Gf2_1, y::Int64)
   @ Base .\floatfuncs.jl:300
 [4] divrem(num::Polynomial{Gf2_1, :x}, den::Polynomial{Gf2_1, :x})
   @ Polynomials C:\Users\yuefei\.julia\packages\Polynomials\Fh8md\src\polynomials\s
tandard-basis.jl:234
 [5] rem(n::Polynomial{Gf2_1, :x}, d::Polynomial{Gf2_1, :x})
   @ Polynomials C:\Users\yuefei\.julia\packages\Polynomials\Fh8md\src\common.jl:112
6
 [6] top-level scope
   @ In[33]:1
```

To avoid this issue, we suggest adding the following new `safediv` and `saferem` methods for polynomials over finite fields of characteristic two. These assume that the indeterminate is `x`.

In [34]:
```
function safediv(a::Polynomial{<:Gf2,:x},b::Polynomial{<:Gf2,:x})
    T = typeof(coeffs(a)[1])
    x = Polynomial([zero(T),one(T)])
    div(x*a,x*b)
end
```

Out[34]:
```
safediv (generic function with 1 method)
```

In [35]:
```
function saferem(a::Polynomial{<:Gf2,:x},b::Polynomial{<:Gf2,:x})
    T = typeof(coeffs(a)[1])
    x = Polynomial([zero(T),one(T)])
    div(rem(x*a,x*b),x)
end
```

Out[35]:
```
saferem (generic function with 1 method)
```

In [36]:
```
safediv(v,Polynomial([one(GF2)]))
```

Out[36]: $1{\cdot}x + 1{\cdot}x^2 + 1{\cdot}x^3 + 1{\cdot}x^6$

In [37]:
```
saferem(v,Polynomial([one(GF2)]))
```

Out[37]: 0

In the cell below, we form all polynomials in $\mathbb{F}_2^{<2}[x]$, i.e., all polynomials of degree at most 1 in $\mathbb{F}_2[x]$. This is a vector space of dimension 2 over $\mathbb{F}_2$.

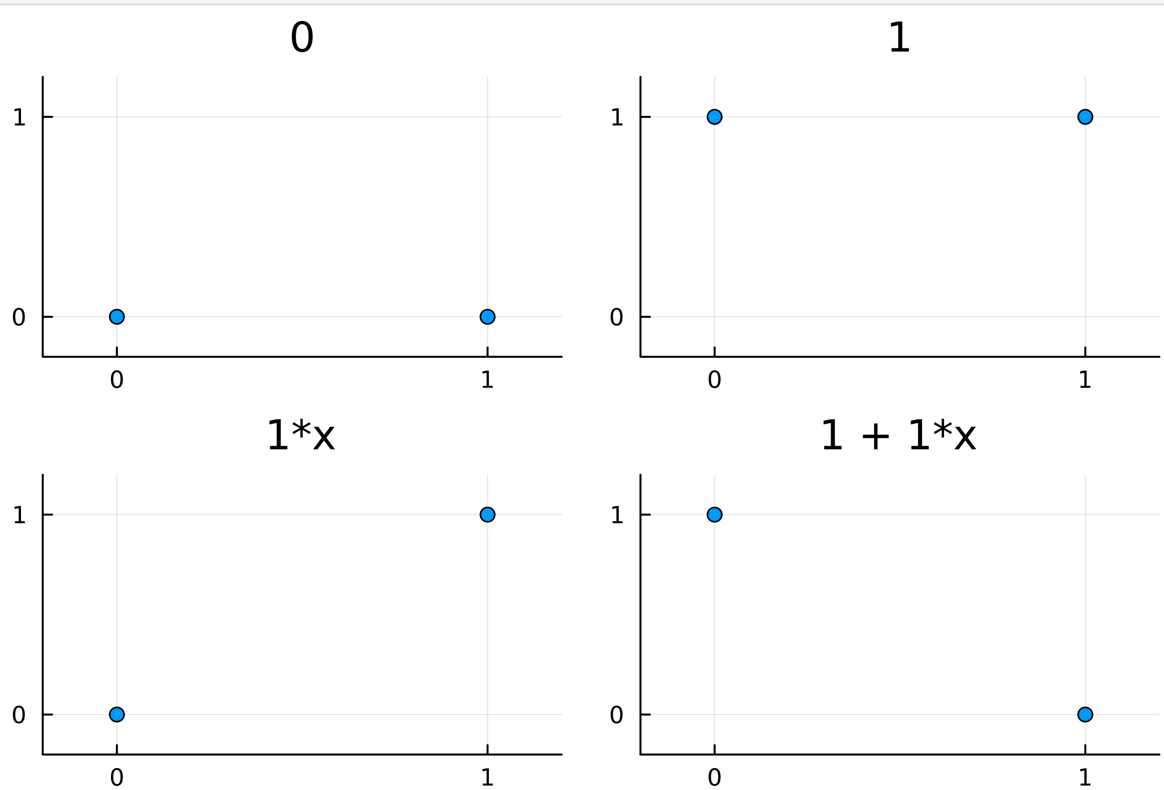
In [38]:
```
p0 = Polynomial(GF2[0])
p1 = Polynomial(GF2[1])
p2 = Polynomial(GF2[0, 1])
p3 = Polynomial(GF2[1, 1])
P_1 = [p0, p1, p2, p3]
println.(P_1);
```

```
0
1
1*x
1 + 1*x
```

Next, we plot the functions associated with each the polynomials defined in the above cell:

In [39]:
```
using Plots
x = [0, 1]
y = zeros(2, 4)
for k = 1:4
    y[:, k] = map(x -> x == GF2(0) ? 0 : 1, P_1[k].(GF2.(x)))
end
allplots = [plot(x, y[:, k], seriestype = :scatter, title = "$(P_1[k])", legend = f
        xlim = (-0.2, 1.2), ylim = (-0.2, 1.2), xticks = 0:1, yticks = 0:1) for k =
plot(allplots..., layout = (2, 2))
```

Out[39]:



Notice that we obtained all possible functions from $\mathbb{F}_2$ to $\mathbb{F}_2$ by considering the set of functions associated with $\mathbb{F}_2^{<2}[x]$.

**Remark:** In fact, there is a one-to-one correspondence between the set of functions from $\mathbb{F}_q$ to $\mathbb{F}_q$ and the set of polynomial functions $\mathbb{F}_q^{<q}[x]$.

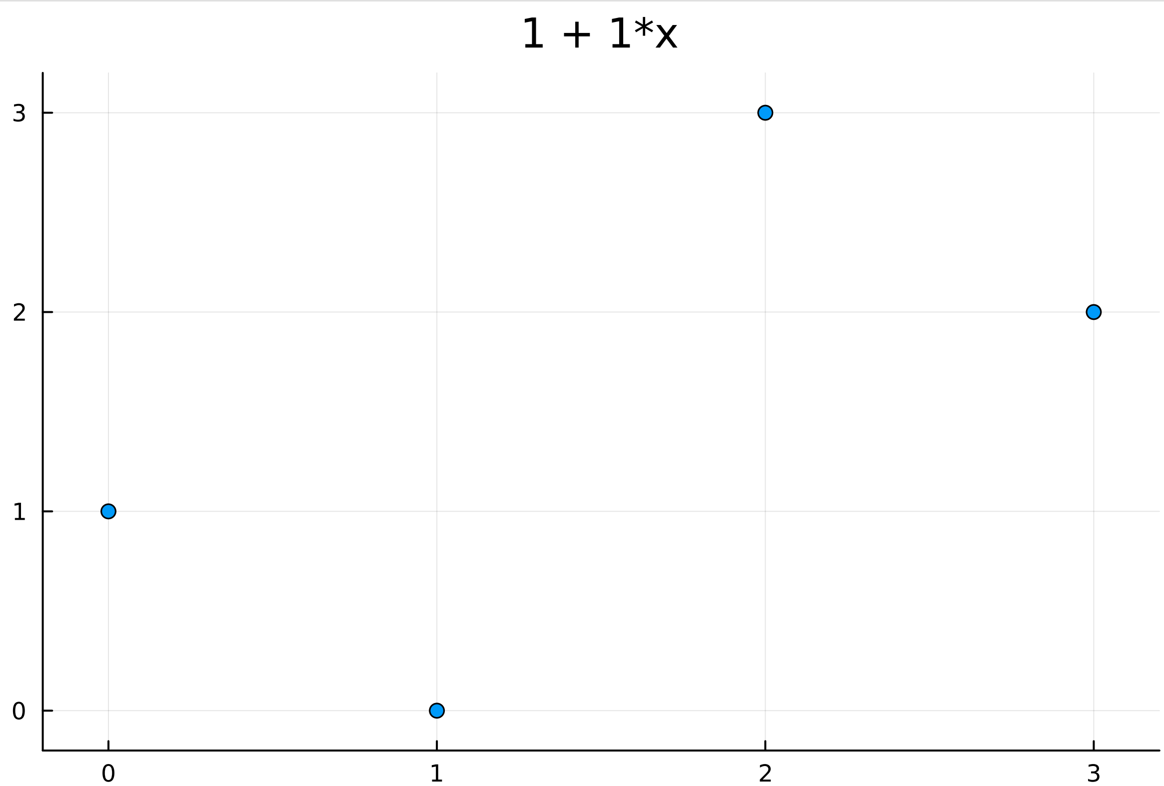
Let us now create the linear polynomial $p(x) = 1+x$ over $\mathbb{F}_4$ and plot its associated function:

In [40]:

```
a = GF4[1, 1]
p = Polynomial(a)
x = [0, 1, 2, 3]
y = map(x -> x == GF4(0) ? 0 : (x == GF4(1) ? 1 : x == GF4(2) ? 2 : 3), p.(GF4.(x)))

plot(x, y, seriestype = :scatter, title = "$(p)", legend = false,
        xlim = (-0.2, 3.2), ylim = (-0.2, 3.2), xticks = 0:3, yticks = 0:3)
```

Out[40]:



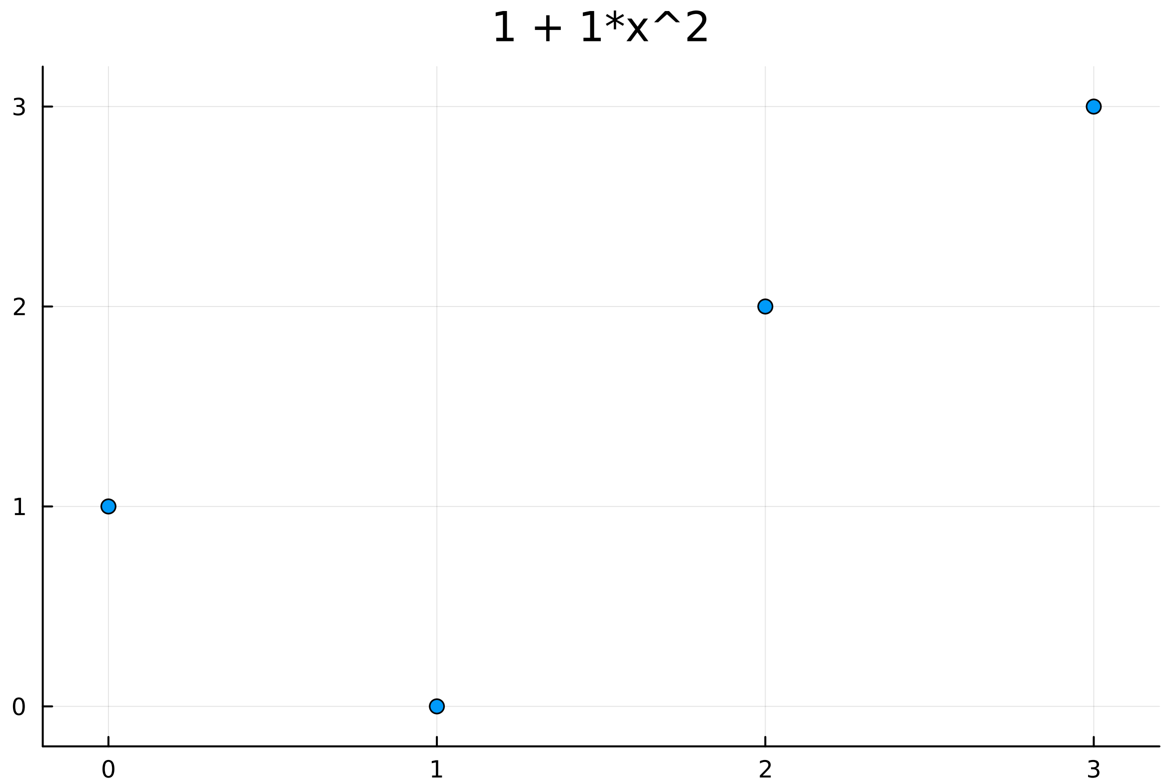

Let us now consider $q(x) = p^2(x) = (1+x)^2 = 1 + x^2$ over $\mathbb{F}_4$ and plot its associated function:

In [41]:

```
q = p^2
y = map(x -> x == GF4(0) ? 0 : (x == GF4(1) ? 1 : x == GF4(2) ? 2 : 3), q.(GF4.(x)))

plot(x, y, seriestype = :scatter, title = "$(q)", legend = false,
        xlim = (-0.2, 3.2), ylim = (-0.2, 3.2), xticks = 0:3, yticks = 0:3)
```

Out[41]:




# Exercise 0: This exercise will not be marked and merely serves the purpose of familiarizing you with `Polynomials`.

---

1. Evaluate the polynomial $p(x) = x^{256} - x$ at all elements of $\mathbb{F}_{256}$. For how many elements do you get 0? *Programming note:* if `v` is a vector of elements from `GF256`, then `count(x->x==zero(GF256),v)` counts the number of zero coordinates.

2. Implement a function `conjugates(a::Gf2)` that takes a field element `a` from $\mathbb{F}_{2^m}$ and returns a vector containing the conjugates of `a` with respect to $\mathbb{F}_{2}$. Test your algorithm by computing the conjugates of a primitive element in $\mathbb{F}_{2^m}$ for each $m \in \{ 1, 2, \ldots, 8 \}$.

   **Hint:** Recall that the set of conjugates of an element $a\in\mathbb{F}_{2^m}$ with respect to $\mathbb{F}_2$ is $$\left\{a, a^2, a^{2^2}, a^{2^3}, \dots\right\}.$$

3. Using your implementation in part 2, implement a function `minpoly(a::Gf2)` that takes, as input, a field element `a` from $\mathbb{F}_{2^m}$ and returns the minimal polynomial of `a` with respect to $\mathbb{F}_{2}$. For later use, ensure that your function returns a value of type `Polynomial{Gf2_1,:x}`. Test your algorithm by computing the minimal polynomial of a primitive element in $\mathbb{F}_{2^m}$ for each $m \in \{ 1, 2, \ldots, 8 \}$.

**Hint:** The minimal polynomial for an element $a\in\mathbb{F}_{2^m}$ with respect to $\mathbb{F}_2$ is the monic polynomial in $\mathbb{F}_2[x]$ of smallest degree having $a$ as a root, and can be found using $$M_a(x) = \prod_{\gamma\in C(a)}(x-\gamma)$$ where $C(a)$ is the set of conjugates of $a$ with respect to $\mathbb{F}_2$.

*Programming note:* `zero(typeof(a))` and `one(typeof(a))` return 0 and 1 elements in the same field as `a`.

### answer for 0.1 here

There are `256` elements that give 0.

In [42]:
```
a1 = GF256[0,1]
p1 = (Polynomial(a1))^256
a2 = GF256[0,1]
p2 = (Polynomial(a2))
px256 = p1 + p2
v = GF256.(0:255)
proots = px256.(v)
count(x->x==zero(GF256),proots)
```

Out[42]:
```
256
```

In [43]:
```
function conjugates(a::Gf2)
    # fill in the rest
    order = gforder(a)
    conj = typeof(a)[]
    push!(conj,a)
    i = 1
    tmp = a^(2^i)
    while tmp != a
        push!(conj,tmp)
        i = i + 1
        tmp = a^(2^i)
    end

    return conj
end
```

Out[43]:
```
conjugates (generic function with 1 method)
```

In [44]:
```
for m in 1:8
    println("$(m): $(conjugates(gfprimitive(m)))")
end
```

```
1: Gf2_1[1]
2: Gf2_2[2, 3]
3: Gf2_3[2, 4, 6]
4: Gf2_4[2, 4, 3, 5]
5: Gf2_5[2, 4, 16, 13, 27]
6: Gf2_6[2, 4, 16, 55, 19, 50]
7: Gf2_7[2, 4, 16, 6, 20, 22, 18]
8: Gf2_8[2, 4, 16, 29, 76, 157, 95, 133]
```

In [146…

```julia
function minpoly(a::Gf2)
    T = eltype(a)
    x = Polynomial(T[0,1])  # this gives the monomial x
    roots = conjugates(a)

    poly = prod((x - r) for r in roots)

    cp = coeffs(poly)

    # reduce the coeffs
    y = [c == T(0) ? GF2(0) : GF2(1) for c in cp]
    poly = Polynomial(y)
    return poly
end
```

Out[146]:

```
minpoly (generic function with 1 method)
```

In [147…

```julia
typeof(minpoly(gfprimitive(3)))  # you should get Polynomial{Gf2_1,:x}
```

Out[147]:

```
Polynomial{Gf2_1, :x}
```

In [148…

```julia
a = gfprimitive(6)
p = minpoly(a)
p(a)
```

Out[148]:

```
0
```

In [149…

```julia
for m in 1:8
    println("$(m): $(minpoly(gfprimitive(m)))")
end
```

```
1: 1 + 1*x
2: 1 + 1*x + 1*x^2
3: 1 + 1*x + 1*x^3
4: 1 + 1*x + 1*x^4
5: 1 + 1*x^2 + 1*x^5
6: 1 + 1*x + 1*x^3 + 1*x^4 + 1*x^6
7: 1 + 1*x + 1*x^7
8: 1 + 1*x^2 + 1*x^3 + 1*x^4 + 1*x^8
```

---

# 1. Reed–Solomon Codes and Berlekamp–Welch Decoder

Let $\mathbb{F}_q$ be a field of size $q$. For $k \geq 1$, let $\mathbb{F}_q^{<k}[x]$ denote the set of polynomials of degree at most $k - 1$ over $\mathbb{F}_q$. We note that $\mathbb{F}_q^{<k}[x]$ is a vector space of dimension $k$ over $\mathbb{F}_q$, with basis $\{1, x, x^2, \dots , x^{k-1}\}$. This vector space contains $q^k$ different polynomials, each of the form $$ u(x) = \sum_{i=0}^{k-1}u_ix^i,\quad u_i\in\mathbb{F}_q. $$

Let $\mathcal{E} = (\alpha_1, \alpha_2, \dots, \alpha_n)$ be some ordered subset of distinct elements of $\mathbb{F}_q$ called the **code locators**. The evaluation map $$ \mathtt{ev}_{\mathcal{E}}:\mathbb{F}_q^{<k}\to\mathbb{F}_q^n $$ with respect to $\mathcal{E}$ sends a polynomial $u(x)$ to the $n$-tuple $$ \left(u(\alpha_1), u(\alpha_2),\dots, u(\alpha_n)\right). $$

This evaluation map is a linear map. In fact, if $n\geq k$ the map $\mathtt{ev}_{\mathcal{E}}$ is injective. Therefore, the image of this linear map is a subspace of $\mathbb{F}_q^n$ of dimension $k$. Denote this image by $C$, i.e., $$ C = \mathtt{ev}_{\mathcal{E}}\left(\mathbb{F}_q^{<k}\right). $$ Hence, $C$ is an $(n,k)$ code called a Reed–Solomon (RS) code of dimension $k$ with code locators $\mathcal{E}$. By considering the fact that each nonzero polynomial $u(x)\in\mathbb{F}_q^{<k}$ has at most $k-1$ roots, it follows that the minimum weight of the nonzero codewords in this RS code is at least $n-(k-1)$. That is, RS codes are MDS!

To encode a message vector $u\in\mathbb{F}_q^k$ according to this RS code, we first associate the message vector with the message polynomial $$ u(x) = \sum_{i=0}^{k-1}u_ix^i $$

and then find the corresponding codeword by using the evaluation map $\mathtt{ev}_{\mathcal{E}}$.

The Berlekamp-Welch (BW) algorithm provides a way to decode RS codes. Let $u$ be a message vector that is mapped to the codeword $c$ via the evaluation map $\mathtt{ev}_\mathcal{E}$ where $\mathcal{E} = (\alpha_1,\ldots,\alpha_n)$ for distinct code locators $\alpha_,\ldots,\alpha_n \in \mathbb{F}_q$. Let the received word be $y = c + e$ where $e$ is the error vector which we assume has a Hamming weight of at most $t = \frac{n-k}{2}$.

The BW algorithm first finds a bivariate polynomial $Q(x,y)$ of the form $$ Q(x,y) = Q_0(x) + yQ_1(x), \quad \deg(Q_0)\leq n-t-1,\quad \deg(Q_1)\leq n-t-k $$ that **interpolates** the pairs $$ P = \{(\alpha_1, y_1), (\alpha_2, y_2), \dots, (\alpha_n, y_n)\} $$ in a way that $$ Q(\alpha_i,y_i) = 0,\quad \forall (\alpha_i,y_i)\in P. $$

This latter equation gives a system of linear equations in the coefficients of $Q_0$ and $Q_1$ that is guaranteed to have solutions and can be solved using standard techniques from linear algebra. More precisely, denote the (unknown) coefficients of $Q_0(x)$ as $a_0, a_1, \ldots, a_{n-t-1}$ and denote the (unknown) coefficients of $Q_1(x)$ as $b_0, b_1, \ldots, b_{n-t-k}$. The number of unknowns is $$(n-t) + (n - t -k +1) = n + (n-2t -k) +1 = \begin{cases} n+1 & \text{if }n-k=2t,\\ n+2 & \text{if }n-k=2t+1. \end{cases}$$ The interpolation condition gives $n$ homogeneous linear equations in these unknowns, which can be written in matrix form as follows: $$ \left[\begin{array}{cccccccccc} 1 & \alpha_1 & \alpha_1^2 & \cdots & \alpha_1^{n-t-1} & y_1 & y_1 \alpha_1 & y_1 \alpha_1^2 & \cdots & y_1 \alpha_1^{n-t-k} \\ 1 & \alpha_2 & \alpha_2^2 & \cdots & \alpha_2^{n-t-1} & y_2 & y_2

\alpha_2 & y_2 \alpha_2^2 & \cdots & y_2 \alpha_2^{n-t-k} \\ 1 & \alpha_3 & \alpha_3^2 & \cdots & \alpha_3^{n-t-1} & y_3 & y_3 \alpha_3 & y_3 \alpha_3^2 & \cdots & y_3 \alpha_3^{n-t-k} \\ 1 & \alpha_4 & \alpha_4^2 & \cdots & \alpha_4^{n-t-1} & y_4 & y_4 \alpha_4 & y_4 \alpha_4^2 & \cdots & y_4 \alpha_4^{n-t-k} \\ \vdots & \vdots & \vdots & & \vdots & \vdots & \vdots & \vdots & \vdots & & \vdots \\ 1 & \alpha_n & \alpha_n^2 & \cdots & \alpha_n^{n-t-1} & y_n & y_n \alpha_n & y_n \alpha_n^2 & \cdots & y_n \alpha_n^{n-t-k} \end{array}\right] \left[\begin{array}{c} a_0 \\ \vdots \\ a_{n-t-1} \\ b_0 \\ \vdots \\ b_{n-t-k} \end{array}\right] = \left[ \begin{array}{c} 0 \\ 0 \\ 0 \\ 0 \\ \vdots \\ 0 \end{array}\right] $$ **Hint:** to find a nonzero solution, consider the given matrix to be the parity-check matrix for a linear code of length $2n -2t -k +1$ over $\mathbb{F}_q$. You seek a nonzero codeword. Consider converting the parity-check matrix to a generator matrix, perhaps re-using a previously written function (from Numerical Exercise 1).

The BW algorithm is then described as follows:

> **Input:** received word $y = (y_1, y_2, \dots, y_n)$; code locators $\mathcal{E} = (\alpha_1, \dots, \alpha_n)$ and the dimension of the code $k$
>
> **Output:** message polynomial $u(x)$
>
> 1. Form a bivariate polynomial $Q(x,y) = Q_0(x) + yQ_1(x)$ as described.
> 2. Form $f(x) = -\frac{Q_0(x)}{Q_1(x)}$. If a nonzero remainder results, declare a decoding failure.
> 3. Check that the Hamming distance between $y$ and $\mathtt{ev}_{\mathcal{E}}(f(x))$ is at most $\frac{n-k}{2}$; otherwise declare a decoding failure.
> 4. Return $f(x)$

## Exercise 1:

---

1. Implement a function `rs_encoder(u::AbstractVector{<:Gf2}, E::AbstractVector{<:Gf2})` that takes a information vector `u` of length $k$ as well as a vector of code locators `E` of length $n$ and produces the corresponding codeword `c` in the $(n, k)$ RS code defined with code locators `E`.

2. Implement the Berlekamp–Welch decoder, as described above, by providing a function `bw_decoder(y::AbstractVector{<:Gf2},E::AbstractVector{<:Gf2},k::Int)` that takes a received word `y`, code locators `E` and the code dimension `k`. Your implemented function must return the correct message polynomial if `y` is at a Hamming distance of at most $t = \frac{n-k}{2}$ from a valid codeword. Your decoder must declare failure (by returning an empty vector) in case the division in the second

step 2 of BW algorithm is not possible or if the condition in step 3 of BW algorithm is not satisfied.

3. Test your implementation by running the following cell.

In [49]:
```
function rs_encoder(u::AbstractVector{<:Gf2},E::AbstractVector{<:Gf2})
    px = Polynomial(u,:x)
    c = px.(E)
    return c
end
```

Out[49]: `rs_encoder (generic function with 1 method)`

In [50]:
```
Pkg.add("LinearAlgebra");using LinearAlgebra
```

```
  Resolving package versions...
 No Changes to `C:\Users\yuefei\.julia\environments\v1.8\Project.toml`
 No Changes to `C:\Users\yuefei\.julia\environments\v1.8\Manifest.toml`
```

In [51]:
```
function dual(G::AbstractArray{<:Gf2})  # convert from G to H or vice-versa
    # you can copy/modify this from Numerical Exercise 1
    T = eltype(G)
    A = rref(G)
    (k,n) = size(A)
    if k != 0
        pivots = []
        for i=1:k
            tmp = findfirst(x->x!=zero(T),A[i,:])
            if tmp!=nothing
                push!(pivots,tmp)
            end
        end
        nonpivots = setdiff(1:1:n,pivots)
        H = zeros(T,n-length(pivots),n)

        P_G = A[1:length(pivots),nonpivots]

        P_tmp = -transpose(P_G)
        I_tmp = Array{T}(1I, n-length(pivots), n-length(pivots))

        p_H = view(H,:,pivots)
        i_H = view(H,:,nonpivots)
        copyto!(p_H,P_tmp)
        copyto!(i_H,I_tmp)
        return H
    else
        return zeros(T,0,n)
    end
end
```

Out[51]: `dual (generic function with 1 method)`

In [52]:
```
function bw_decoder(y::AbstractVector{<:Gf2},E::AbstractVector{<:Gf2},k::Int)
    n = length(y)
    if (n - k) % 2 == 0
```

```
        t = Int((n - k)/2)
    else
        t = Int((n - k - 1)/2)
    end
    max_ERR = t
    l0 = n-t-1
    l1 = n-t-k
    L0 = transpose(0:1:l0)
    L1 = transpose(0:1:l1)

    # Construct Parity-check matrix
    sz = 2*n-2*t-k+1
    P = Array{GF256,2}(undef,n,sz)
    for i=1:n
        alpha = E[i]
        cp1 = alpha.^L0
        cp2 = y[i]*(alpha.^L1)
        v=hcat(cp1,cp2)
        P[i,:] = v
    end
    G = dual(P) # Now we got the solution of Q
    msg = ones(GF256,1,size(G)[1])

    q_coeff = msg*G
    q0 = q_coeff[1:1+l0]
    q1 = q_coeff[l0+2:l0+2+l1]
    Q0 = Polynomial(q0,:x)
    Q1 = Polynomial(q1,:x)

    if (saferem(-Q0,Q1) == GF256(0))
        fx = safediv(-Q0,Q1)
        fx_ev = fx.(E)
        if count(fx_ev==y) <= max_ERR
            return fx
        else
            return Array{GF256}(undef, 0)
        end
    else
        return Array{GF256}(undef, 0)
    end
end
```

I see that your return type is a polynomial. However, the user gave you a vector to encode, so it would make sense to return a vector.

Later, you are missing the initial letter "I" because of this...

Out[52]:
```
bw_decoder (generic function with 1 method)
```

In [53]:
```
# Test your implementation.
message = GF256[73,116,32,119,111,114,107,101,100,33,32,32,67,111,110,103,114,97,11
    97,116,105,111,110,115,32,111,110,32,105,109,112,108,101,109,101,110,116,105,11
    32,121,111,117,114,32,102,105,114,115,116,32,82,101,101,100,45,83,111,108,111,1
    110,32,100,101,99,111,100,101,114,33]
k = length(message)
t = 30 # we will make a t-error-correcting code
n = k + 2*t
if n > 255
    println("Oops, excessive block-length!")
end
E = [gfprimitive(8)^i for i=1:n]
```

```
p = 0.2  # probability of symbol error
e = [rand() < p ? GF256(rand(1:255)) : zero(GF256) for i = 1:n]
w = count(x->x!=zero(GF256),e)
print("Error pattern has weight $(w), which ")
if w <= t
    println("should decode correctly.")
else
    println("will likely cause a decoding failure.")
end
c = rs_encoder(message,E)
u_hat = bw_decoder(c+e,E,k)
if length(u_hat) > 0
    println(String([Char(u_hat[j].value) for j = 0:length(u_hat)]))
else
    println("Decoding failure!")
end
```

```
Error pattern has weight 32, which will likely cause a decoding failure.
Decoding failure!
```

When we ran your code, it did not produce the correct message (the first symbol was missing!) See my earlier comment.

---

# 2. The Extended Euclidean Algorithm

The extended Euclidean algorithm is an extension to Euclid's Algorithm that computes, in addition to the greatest common divisor (gcd) of two elements $a$ and $b$ (which are not both zero) in a Euclidean domain, also the coefficients $s$ and $t$ of Bézout's identity so that $$ \gcd(a, b) = s\cdot a + t\cdot b. $$

Define the norm of a polynomial $p(x)\in\mathbb{F}_q[x]$ as its degree, $N(p) = \deg(p)$. Equipped with $N(\cdot)$, the ring of polynomials $\mathbb{F}_q[x]$ becomes a Euclidean domain. Naturally, the extended Euclidean algorithm for two polynomials $a(x)$ and $b(x)$ can be used to find the greatest common divisor of $a(x)$ and $b(x)$ as well as two polynomials $s(x)$ and $t(x)$ such that $$ \gcd(a, b) = s(x)a(x) + t(x)b(x). $$

A pseudocode description of the extended Euclidean algorithm is given below (see Algorithm 47 in Chapter 7 of the Lecture Notes):

> **Require:** $a,b\in E, a\neq 0, N(a)\geq N(b)$ or $b=0$, where $E$ is a Euclidean domain.
>
> $M \leftarrow \begin{bmatrix}a & 1 & 0\\b & 0 & 1\end{bmatrix}$ {Notation: $M = [m_{ij}]$}
>
> **while $m_{21}\neq 0$** do**
>
> Using the division algorithm in $E$, find $q$ such that $m_{11} = qm_{21} + r$ with $r = 0$ or $N(r) < N(m_{21})$;
>
> $M \leftarrow \begin{bmatrix}0 & 1\\1 & -q\end{bmatrix}M$

> **end while**
>
> **return** $(gcd(a,b), s, t) = (m_{11}, m_{12}, m_{13})$

# Exercise 2:

---

1. Implement a function `(g, s, t) = eea(a, b)` that takes two polynomials `a` and `b` both over the same finite field $\mathbb{F}$ and returns the greatest common divisor of `a` and `b` (called `g`) as well as two polynomials `s` and `t` such that `g = a * s + b * t`. In case both inputs are zero, your implementation must return `(0, 1, 0)`. **Reminder:** use the `safediv` and `saferem` functions (as needed) that were defined earlier.

2. Let $g(x) = 1 + x + x^{127} \in \mathbb{F}_2[x]$. Since $g(x)$ is irreducible, the quotient ring $\mathbb{F}_2[x]/\langle g(x) \rangle$ is the finite field $\mathbb{F}_{2^{127}}$. Let $a(x) = 1 + x + x^2 + x^3 + x^4$. Find the multiplicative inverse of the elements $[a(x)]$ and $[1 + x^{122}a(x)]$ in this field, and verify that the elements that you have found are indeed the multiplicative inverses. **Hint:** If $b(x)$ is any polynomial of degree < 127, then $\gcd(b, g) = 1$. Use your implementation of `eea` to find Bézout's coefficients.

In [54]:
```
function eea(a,b)
    # init
    T = typeof(a)
    r0 = a; r1 = b
    s0 = T(1); s1 = T(0)
    t0 = T(0); t1 = T(1)

    while r1 != T(0)
        q = safediv(r0,r1)
        tmp1= r0; tmp2 = s0; tmp3 = t0;
        r0 = r1; s0 = s1; t0 = t1;
        r1 = tmp1 - q*r1
        s1 = tmp2 - q*s1
        t1 = tmp3 - q*t1
    end
    g = r0
    s = s0
    t = t0
    return g, s, t
end
```

Out[54]:
```
eea (generic function with 1 method)
```

In [55]:
```
eea(Polynomial(zero(GF2)),Polynomial(zero(GF2)))
```

Out[55]:
```
(Polynomial(0), Polynomial(1), Polynomial(0))
```

```
In [56]: g1 = Polynomial(GF2[0,1],:x)^127
g2 = Polynomial(GF2[1,1],:x)
g = g1+g2
ax = Polynomial(GF2[1,1,1,1,1],:x)
cx = GF2(1) + Polynomial(GF2[0,1],:x)^122*ax
# Compute Multiplicative Inverse
g_ax, inv_ax, t_ax = eea(ax,g)
g_cx, inv_cx, t_cx = eea(cx,g)

println("Multiplicative Inverse of [a(x)]:")
display(inv_ax)
println("Multiplicative Inverse of [1 + x^{122}a(x)]:")
display(inv_cx)
```

```
Multiplicative Inverse of [a(x)]:
```

$$1{\cdot}x + 1{\cdot}x^{2} + 1{\cdot}x^{3} + 1{\cdot}x^{5} + 1{\cdot}x^{6} + 1{\cdot}x^{7} + 1{\cdot}x^{8} + 1{\cdot}x^{10} + 1{\cdot}x^{11} + 1{\cdot}x^{12} + 1{\cdot}x^{13} + 1{\cdot}x^{15} + 1{\cdot}x^{16} + 1{\cdot}x^{17} + 1{\cdot}x^{18} + 1{\cdot}x^{20} + 1{\cdot}x^{21} + 1{\cdot}x^{22} + 1{\cdot}x^{23} + 1{\cdot}x^{25} + 1{\cdot}x^{26} + 1{\cdot}x^{27} + 1{\cdot}x^{28} + 1{\cdot}x^{30} + 1{\cdot}x^{31} + 1{\cdot}x^{32} + 1{\cdot}x^{33} + 1{\cdot}x^{35} + 1{\cdot}x^{36} + 1{\cdot}x^{37} + 1{\cdot}x^{38} + 1{\cdot}x^{40} + 1{\cdot}x^{41} + 1{\cdot}x^{42} + 1{\cdot}x^{43} + 1{\cdot}x^{45} + 1{\cdot}x^{46} + 1{\cdot}x^{47} + 1{\cdot}x^{48} + 1{\cdot}x^{50} + 1{\cdot}x^{51} + 1{\cdot}x^{52} + 1{\cdot}x^{53} + 1{\cdot}x^{55} + 1{\cdot}x^{56} + 1{\cdot}x^{57} + 1{\cdot}x^{58} + 1{\cdot}x^{60} + 1{\cdot}x^{61} + 1{\cdot}x^{62} + 1{\cdot}x^{63} + 1{\cdot}x^{65} + 1{\cdot}x^{66} + 1{\cdot}x^{67} + 1{\cdot}x^{68} + 1{\cdot}x^{70} + 1{\cdot}x^{71} + 1{\cdot}x^{72} + 1{\cdot}x^{73} + 1{\cdot}x^{75} + 1{\cdot}x^{76} + 1{\cdot}x^{77} + 1{\cdot}x^{78} + 1{\cdot}x^{80} + 1{\cdot}x^{81} + 1{\cdot}x^{82} + 1{\cdot}x^{83} + 1{\cdot}x^{85} + 1{\cdot}x^{86} + 1{\cdot}x^{87} + 1{\cdot}x^{88} + 1{\cdot}x^{90} + 1{\cdot}x^{91} + 1{\cdot}x^{92} + 1{\cdot}x^{93} + 1{\cdot}x^{95} + 1{\cdot}x^{96} + 1{\cdot}x^{97} + 1{\cdot}x^{98} + 1{\cdot}x^{100} + 1{\cdot}x^{101} + 1{\cdot}x^{102} + 1{\cdot}x^{103} + 1{\cdot}x^{105} + 1{\cdot}x^{106} + 1{\cdot}x^{107} + 1{\cdot}x^{108} + 1{\cdot}x^{110} + 1{\cdot}x^{111} + 1{\cdot}x^{112} + 1{\cdot}x^{113} + 1{\cdot}x^{115} + 1{\cdot}x^{116} + 1{\cdot}x^{117} + 1{\cdot}x^{118} + 1{\cdot}x^{120} + 1{\cdot}x^{121} + 1{\cdot}x^{122} + 1{\cdot}x^{123} + 1{\cdot}x^{125} + 1{\cdot}x^{126}$$

```
Multiplicative Inverse of [1 + x^{122}a(x)]:
```

$$1{\cdot}x^{5}$$

```
In [57]: println("=== Verifying Result ===")
println("([ax]*[ax]^-1) mod (g(x)):\n $((ax * inv_ax) % g)")

println("([1 + x^{122}a(x)]*[1 + x^{122}a(x)]^-1) mod (g(x)):\n $((cx * inv_cx) % g
```

```
=== Verifying Result ===
([ax]*[ax]^-1) mod (g(x)):
 1
([1 + x^{122}a(x)]*[1 + x^{122}a(x)]^-1) mod (g(x)):
 1
```

---

# 3. Syndrome Decoding of Generalized Reed–Solomon Codes

Let $\mathbb{F}_q$ be a finite field and let $\mathcal{E}=(\alpha_1, \alpha_2, \dots, \alpha_n)$ be an ordered set of distinct elements from $\mathbb{F}_q$ called code locators. Let $\mathcal{M} = (\mu_1, \mu_2, \dots, \mu_n)$ be an ordered set of nonzero (not necessarily distinct) elements from $\mathbb{F}_q$ called column multipliers.

A generalized Reed–Solomon (GRS) code can be specified by giving either a generator matrix or a parity-check matrix in the form $$ \begin{bmatrix} 1 & 1 & 1 & \dots & 1\\ \alpha_1 & \alpha_2 & \alpha_3 & \dots & \alpha_n\\ \alpha_1^2 & \alpha_2^2 & \alpha_3^2 & \dots & \alpha_n^2\\ \vdots & \vdots & \vdots & \ddots & \vdots\\ \alpha_1^{r-1} & \alpha_2^{r-1} & \alpha_3^{r-1} & \dots & \alpha_n^{r-1}\\ \end{bmatrix} \mathit{\mathrm{diag}}\left(\mu_1, \mu_2, \mu_3, \dots, \mu_n\right) $$

where $\mathit{\mathrm{diag(\cdot)}}$ denotes a diagonal matrix with the given entries on the diagonal.

Taken as a generator matrix, the above equation defines an $(n,r)$ GRS code, and the column multipliers are then referred to as generator-matrix column multipliers. Taken as a parity-check matrix, it defines an $(n, n-r)$ GRS code, and the column multipliers are then referred to as parity-check-matrix column multipliers.

## 3.1 Encoding

Encoding of a GRS code with the generator matrix $$ G = \begin{bmatrix} 1 & 1 & 1 & \dots & 1\\ \alpha_1 & \alpha_2 & \alpha_3 & \dots & \alpha_n\\ \alpha_1^2 & \alpha_2^2 & \alpha_3^2 & \dots & \alpha_n^2\\ \vdots & \vdots & \vdots & \ddots & \vdots\\ \alpha_1^{k-1} & \alpha_2^{k-1} & \alpha_3^{k-1} & \dots & \alpha_n^{k-1}\\ \end{bmatrix} \mathit{\mathrm{diag}}\left(\mu_1, \mu_2, \mu_3, \dots, \mu_n\right) $$ can be accomplished by means of polynomial evaluation. For an input word $$ u = (u_0, u_1, \dots, u_{k-1})$$

define a corresponding polynomial $u(x)$ of degree at most $k-1$ by $$ u(x) = u_0 + u_1x + \dots + u_{k-1}x^{k-1}. $$ You can easily verify that $$ uG = \left(\mu_1u(\alpha_1), \mu_2u(\alpha_2), \dots, \mu_nu(\alpha_n)\right). $$

# Exercise 3 (Part I: Encoders)

1. Implement a function `grs_encoder(u::AbstractVector{<:Gf2},E::AbstractVector{<:Gf2},M::Abstract` that takes an information vector `u`, a vector of distinct code locators `E`, a vector of generator-matrix column multipliers `M` and returns the corresponding codeword. Run the following cells to verify your encoder.
2. Implement a function `M2M(E::AbstractVector{<:Gf2},M::AbstractVector{<:Gf2})` that takes a vector of distinct code locators `E` and a vector of generator-matrix column multipliers `M` and returns the corresponding parity-check-matrix column multipliers. **Hint:** See the proof of Theorem 15 in Chapter 6 of the lecture notes. In that proof you will observe that you seek a nonzero codeword in a linear code of length $n$ given by a parity-check matrix.

Consider converting the parity-check matrix to a generator matrix, perhaps re-using a previously written function (from Numerical Exercise 1).

In [58]:
```
function grs_encoder(u::AbstractVector{<:Gf2},E::AbstractVector{<:Gf2},M::AbstractV
    ux = Polynomial(u,:x)
    ua = M.*ux.(E)
    return ua
end
```

Out[58]: `grs_encoder (generic function with 1 method)`

In [59]:
```
grs_encoder(GF8[0,1,0],GF8[1,2,3,4,5,6,7],GF8[7,3,3,1,6,6,4])  # expect [7,6,5,4,3,
```

Out[59]:
```
7-element Vector{Gf2_3}:
 7
 6
 5
 4
 3
 2
 1
```

In [60]:
```
function M2M(E::AbstractVector{<:Gf2},M::AbstractVector{<:Gf2})
    T = eltype(E)
    dT = typeof(M)
    n = length(E)
    nr = n-1

    H_prime = Array{T}(undef,nr,n)
    lp = 0:1:nr-1
    for i=1:n
        tmp = M[i]*E[i].^lp
        H_prime[:,i] = tmp
    end
    G = dual(H_prime)
    msg = one(T)
    c = vec(msg*G)
    return c
end
```

Out[60]: `M2M (generic function with 1 method)`

In [61]:
```
M2M(GF8[1,2,3,4,5,6,7],GF8[7,3,3,1,6,6,4])  # expect (5,4,6,5,5,6,1)
```

Out[61]:
```
7-element Vector{Gf2_3}:
 5
 4
 6
 5
 5
 6
 1
```

In [62]:
```
v = grs_encoder(GF8[2,4,6],GF8[1,2,3,4,5,6,7],GF8[7,3,3,1,6,6,4])  # a "random" coo
w = grs_encoder(GF8[1,3,5,7],GF8[1,2,3,4,5,6,7],M2M(GF8[1,2,3,4,5,6,7],GF8[7,3,3,1,
sum(+,[v[i]*w[i] for i in 1:length(v)])  # compute their inner product; the result
```

Out[62]:
```
0
```

### 3.2 Syndrome Decoding

Let \$\mathcal{C}\$ be an \$(n,k, d = n-k+1)\$ GRS code over a finite field \$\mathbb{F}_q\$ with parity check matrix \$\$ H = \begin{bmatrix} 1 & 1 & 1 & \dots & 1\\ \alpha_1 & \alpha_2 & \alpha_3 & \dots & \alpha_n\\ \alpha_1^2 & \alpha_2^2 & \alpha_3^2 & \dots & \alpha_n^2\\ \vdots & \vdots & \vdots & \ddots & \vdots\\ \alpha_1^{d-2} & \alpha_2^{d-2} & \alpha_3^{d-2} & \dots & \alpha_n^{d-2}\\ \end{bmatrix} \mathit{\mathrm{diag}}\left(\mu_1, \mu_2, \mu_3, \dots, \mu_n\right) \$\$ where the code locators \$(\alpha_1, \alpha_2, \dots, \alpha_n)\$ are distinct and **nonzero**. For the syndrome-based decoder that we will now develop, which involves the reciprocals of the code locators, the zero code-locator is not permitted! Notice that the \$(\mu_1, \ldots, \mu_n)\$ now represent **parity-check column multipliers** rather than generator matrix column multipliers. Luckily we have a convenient function that takes us back and forth between them!

Suppose a codeword \$c\$ is being transmitted and the received word \$y\$ is given by \$y = c + e \$ where \$e = (e_0, e_1, \dots, e_{n-1})\$ is the error vector. We assume that the Hamming weight \$e\$ is at most \$(d-1)/2\$.

The **syndrome** corresponding to the received word \$y = (y_0,y_1, \dots, y_{n-1})\$ is \$\$ s = yH^T = (s_0, s_1, \ldots, s_{d-2}) \$\$ where \$\$ s_i = \sum_{j=1}^n y_j \mu_j \alpha_j^i. \$\$ Associate with the syndrome the **syndrome polynomial** \$\$s(x) = s_0 + s_1 x + \cdots + s_{d-2} x^{d-2}.\$\$

# Exercise 3 (Part II: Syndromes)

1. Implement a function `grs_syndrome(y::AbstractVector{<:Gf2},E::AbstractVector{<:Gf2},M::Abstrac` that takes a received vector `y`, a vector of distinct code locators `E`, a vector of parity-check column multipliers `M` and the code dimension `k`, and returns the corresponding syndrome polynomial. Run the following cells to verify your syndrome-former.

In [179…
```
function grs_syndrome(y::AbstractVector{<:Gf2},E::AbstractVector{<:Gf2},M::Abstract
    n = length(E)
    d = n-k+1
    T = eltype(E)
    nr = d-1
    lp = 0:1:nr-1

    H = Array{T}(undef,nr,n)
```

```
    for i=1:n
        tmp = M[i]*E[i].^lp
        H[:,i] = tmp
    end

    s = H*y
    sx = Polynomial(s,:x)
    return sx
end
```

Out[179]: grs_syndrome (generic function with 1 method)

In [180…
```
grs_syndrome(GF8[7,6,5,4,3,2,1],GF8[1,2,3,4,5,6,7],GF8[5,4,6,5,5,6,1],3)  # this is
```

Out[180]: 0

In [181…
```
grs_syndrome(GF8[7,6,5,4,3,2,0],GF8[1,2,3,4,5,6,7],GF8[5,4,6,5,5,6,1],3)  # error i
```

Out[181]: $1 + 7 \cdot x + 3 \cdot x^2 + 2 \cdot x^3$

In [182…
```
grs_syndrome(GF8[7,6,5,4,3,0,0],GF8[1,2,3,4,5,6,7],GF8[5,4,6,5,5,6,1],3)  # error i
```

Out[182]: $6 + 3 \cdot x + 6 \cdot x^2 + 1 \cdot x^3$

## 3.3 Solving the Key Equations for the Error-Locator and Error-Evaluator Polynomials

Now, since $y=c+e$, the syndrome $s$ is $$ (c+e)H^T = eH^T = (s_0, s_1, \dots, s_{d-2}) $$ with $$ s_i = \sum_{j = 1}^n e_j\mu_j\alpha_j^i = \sum_{j \in J} e_j\mu_j\alpha_j^i. $$ where $J$ is the set of positions in which $e$ is nonzero, i.e., the locations of errors. Note, $$ \lvert J\rvert \leq \frac{d-1}{2} $$ From this we see that $$ s(x) = \sum_{j\in J}e_j\mu_j\sum_{i=0}^{d-2}(\alpha_jx)^i. $$

If we consider polynomials modulo $x^{d-1}$, you easily verify that $$ (1-\alpha_jx)\sum_{i=0}^{d-2}(\alpha_jx)^i = 1 - (\alpha_jx)^{d-1} \equiv 1 \quad\mathrm{(mod~}x^{d-1}\mathrm{)}. $$

Hence, we have $$ s(x) \equiv \sum_{j\in J} \frac{e_j\mu_j}{1-\alpha_jx}\quad\mathrm{(mod~}x^{d-1}\mathrm{)}. $$ where the polynomial division corresponds to multiplication by the corresponding multiplicative inverse in the polynomial ring $\mathbb{F}_q[x]/\langle x^{d-1} \rangle$. (In this ring, an element $[a(x)]$ is invertible if and only if $a(0) \neq 0$.)

Recall the definition of the **error locator polynomial** $$ \Lambda(x) = \prod_{j\in J}(1 - \alpha_jx) $$ and the **error evaluator polynomial** $$ \Gamma(x) = \sum_{j\in J} e_j\mu_j\prod_{m\in J\setminus\{j\}} (1-\alpha_mx). $$

Note that the evaluation of the error locator polynomial at the reciprocal of a code locator corresponding to an error location is zero, hence the name. In other words, the roots of $\Lambda(x)$ are precisely the multiplicative inverses of code locators corresponding to the error locations. The roots of the error locator can be found by testing the reciprocal of the code locators one-by-one (a linear search called a **Chien search**).

This also leads us to the first key equation: $$ \gcd(\Lambda(x), \Gamma(x)) = 1 $$

Additionally, by inspection you can see that $$\deg(\Gamma)<\deg(\Lambda) = \lvert J\rvert \leq \frac{d-1}{2}$$

which will be the second key equation for us.

Define the formal derivative of a polynomial $p(x) = \sum_{k=0}^Lp_k x^k\in \mathbb{F}_q[x]$ by $$ p'(x) = \sum_{k=1}^L\bar{k}p_k x^{k-1} $$ with $$ \bar{k} = \underbrace{1 + 1 + \dots + 1}_{m~\mathrm{terms}},\quad 1 \mathrm{~is~the~identity~element~of~}\mathbb{F}_q. $$ Again, by inspection, you can see that $$ \Lambda(x)s(x)\equiv\Gamma(x)\quad\mathrm{(mod~}x^{d-1}\mathrm{)} $$ which is the third key equation.

The value of the error $e_j$, for $j\in J$, can be found using **Forney's formula** by $$ e_j = -\frac{\alpha_j}{\mu_j}\frac{\Gamma(a_j^{-1})}{\Lambda'(a_j^{-1})}. $$

The main step in decoding GRS codes, therefore, is to find the error locator polynomial and the error evaluator polynomial. This is accomplished by solving the key equations which are given here again for convenience: $$ \gcd(\Lambda(x), \Gamma(x)) = 1 $$ $$ \deg(\Gamma) <\deg(\Lambda)\leq \frac{d-1}{2} $$ $$ \Lambda(x)s(x)\equiv\Gamma(x)\quad\mathrm{(mod~}x^{d-1}\mathrm{)} $$

The key equations can be solved using the extended Euclidean algorithm, with a different stopping criterion, which we call the *modified EEA* (see Chapter 8 of the lecture notes, or Chapter 6 of R. M. Roth, *Introduction to Coding Theory*, Cambridge University Press, 2006). Suppose we run the EEA with $x^{d-1}$ and $s(x)$ as input. At each stage of the algorithm, we have a remainder $r(x)$ which is expressed as a linear combination $a(x) x^{d-1} + b(x) s(x)$. Reducing modulo $x^{d-1}$ we see that $r(x) = b(x)s(x) \bmod x^{d-1}$ and thus $r(x)$ is a candidate for $\Gamma(x)$ and $b(x)$ is a candidate for $\Lambda(x)$. It can be proved that indeed $r(x) = \Gamma(x)$ and $b(x) = \Lambda(x)$ on the iteration of the EEA when the condition $\deg(r(x)) < t$ is first satisfied. Thus the stopping criterion for the modified EEA is chosen so that as soon as $\deg(m_{21}) < t$, the algorithm exits the while loop (see the pseudocode given before Exercise 2).

## Exercise 3 (Part III: Error-Locator and Error-Evaluator Polynomials)

1. Copy your implementation of `eea` from Exercise 2 and modify it so that a new function `mod_eea` is defined that takes in the syndrome polynomial $s(x)$ and the integer $d$ and returns the error-locator and error-evaluator polynomials $\Lambda(x)$ and $\Gamma(x)$ (as a tuple, in that order). Verify your implementation by running the following cells.

In [183…

```
function mod_eea(s,d)
    # init
    T = eltype(s)
    xd = Polynomial(T[0,1],:x)^(d-1)
    r0 = xd; r1 = s
    t0 = T(0); t1 = T(1)

    maxErr = Int(floor((d-1)/2))
    while degree(r1) >= maxErr
        q = safediv(r0,r1)
        tmp1= r0; tmp3 = t0;
        r0 = r1; t0 = t1;
        r1 = tmp1 - q*r1
        t1 = tmp3 - q*t1
    end

    Lambdax = t1  #Error Locator
    Gammax = r1   #Error eva.
    return Lambdax,Gammax
end
```

Out[183]:

```
mod_eea (generic function with 1 method)
```

In [184…

```
E = GF8[1,2,3,4,5,6,7]
M = GF8[5,4,6,5,5,6,1]
n = length(E)
k = 3
d = n-k+1
s = grs_syndrome(GF8[7,6,5,4,3,2,1],E,M,k) # no errors
mod_eea(s,d)  # what degree do you expect for the error-locator? ==> 0
```

Out[184]:

```
(1, Polynomial(0))
```

In [185…

```
s = grs_syndrome(GF8[7,6,5,4,3,2,0],E,M,k) # one error
mod_eea(s,d)  # what degree do you expect for the error-locator? ==> 1
```

Out[185]:

```
(Polynomial(2 + 5*x), Polynomial(2))
```

In [186…

```
s = grs_syndrome(GF8[7,6,5,4,3,0,0],E,M,k) # two errors
mod_eea(s,d)  # what degree do you expect for the error-locator? ==> 2
```

Out[186]:

```
(Polynomial(7 + 7*x + 1*x^2), Polynomial(4 + 6*x))
```

In [187…

```
s = grs_syndrome(GF8[7,6,5,4,0,0,0],E,M,k) # three errors
mod_eea(s,d)  # what degree do you expect for the error-locator? ==> 2 (at most 2 e
```

```
Out[187]: (Polynomial(3 + 3*x + 5*x^2), Polynomial(6 + 5*x))
```

## 3.4 Putting the Pieces Together

All of this leads to the following syndrome-based decoding algorithm for GRS codes with nonzero code locators. Let $\mathcal{E} = (\alpha_1, \alpha_2, \ldots, \alpha_n)$ be the code locators and let $\mathcal{M} = (\mu_1, \mu_2, \ldots, \mu_n)$ be the parity-matrix column multipliers.

> **Input:** received word $y = (y_0, y_1, \dots, y_{n-1})$
>
> **Output:** error word $e = (e_0, e_1, \dots, e_{n-1})$
>
> 1. Compute the syndrome: compute the polynomial $s(x) = s_0 + s_1x+\dots+s_{d-2}x^{d-2}$ by using
>
> $$s_i = \sum_{j =0}^n y_j\mu_j\alpha_j^i.$$
>
> 2. Find the error-locator polynomial $\Lambda(x)$ and the error-evaluator polynomial $\Gamma(x)$: use the modified EEA with inputs $x^{d-1}$ and $s(x)$ to obtain $g(x)$ as well as two polynomials $a(x)$ and $b(x)$ such that
>
> $$ g(x) = a(x)x^{d-1} + b(x)s(x), $$ stopping on the first iteration where $\deg(g(x))<t$. Set $$ \Gamma(x) = g(x), \quad \Lambda(x) = b(x).$$
>
> 3. Finding the error locations and values: for each $j \in \{ 1, \ldots, n \}$ set
>
> $$ e_j = \begin{cases} - \frac{\alpha_j}{\mu_j} \frac{\Gamma\left(\alpha_j^{-1}\right)}{\Lambda'\left(\alpha_j^{-1}\right)} & \text{if } \Lambda(\alpha_j^{-1}) = 0 \\ 0 & \text{otherwise}. \end{cases} $$

# Exercise 3 (Part IV): Building the Decoder

1. Implement a function `D(p)` which takes in a polynomial `p` in $\mathbb{F}_{2^m}[x]$ and returns its formal derivative. Note that in any field of characteristic two, $1+1=0$, $1+1+1=1$, $1+1+1+1=0$, etc. Test your function by running the following cell.

2. Implement a function `grs_decoder(y::AbstractVector{<:Gf2},E::AbstractVector{<:Gf2},M::Abstract` that takes a received word `y`, which is a noisy version of some codeword `c`, the vector of distinct and nonzero code locators `E`, the vector of parity-check-matrix column multipliers `M` and the dimension of the code `k` and returns the error vector `e` obtained by syndrome decoding of GRS codes with solving the key equations. To declare a decoding failure, the decoder should return an empty vector. (A decoding

failure will arise if the number of nonzero positions found for `e` does not match the degree of the error-locator polynomial, or if the Forney formula would give a divide-by-zero error.) Test your implementation by running the following cells.

In [188…

```
function D(p)
    T = eltype(p)
    if degree(p) == -1 # zero polynomial
        return 0
    end

    deg = degree(p)
    pdiff = T(0)
    while deg > 0
        xn = Polynomial(T[0,1],:x)^(deg)
        xndiff = Polynomial(T[0,1],:x)^(deg-1)
        an = safediv(p,xn)
        d = T(deg%2)
        pn = an*d*xndiff
        pdiff = pdiff + pn

        p = saferem(p,xn)
        deg = degree(p)
    end
    return pdiff
end
```

Out[188]: D (generic function with 1 method)

In [189…

```
D(Polynomial(GF8[1,2,3,4,5,6,7]))  # expect D(1 + 2x + 3x^2 + 4x^3 + 5x^4 + 6x^5 +
```

Out[189]: $2 + 4{\cdot}x^2 + 6{\cdot}x^4$

In [190…

```
function grs_decoder(y::AbstractVector{<:Gf2},E::AbstractVector{<:Gf2},M::AbstractV
    sx = grs_syndrome(y,E,M,k)
    T = eltype(sx)
    v_empty = T[]
    if sx == T(0)
        return v_empty
    end
    n = length(y)
    d = n-k+1
    Lx,Gx = mod_eea(sx,d) #Error-locator and Error-evaluator polynomials

    err = zeros(T,n,1)
    for j=1:n
        alpha = E[j]
        La = Lx(alpha^(-1))
        if La == T(0)
            DL = D(Lx)
            DLa = DL(alpha^(-1))

            if M[j] == T(0) || DLa == T(0)
                return v_empty
```

```
            end

            Ga = Gx(alpha^(-1))
            err[j] = -(Ga/DLa)*(alpha/M[j])

        end
    end

    if degree(Lx) != length(err[err .!= T(0)])
        return v_empty
    end
    return err
end
```

Out[190]: grs_decoder (generic function with 1 method)

In [191…
```
E = GF8[1,2,3,4,5,6,7]
M = GF8[5,4,6,5,5,6,1]
n = length(E)
k = 3
d = n-k+1
grs_decoder(GF8[7,6,5,4,3,2,1],E,M,k) # no errors
```

Out[191]: Gf2_3[]

In [192…
```
grs_decoder(GF8[7,6,5,4,3,2,0],E,M,k) # one error, last position
```

Out[192]:
```
7×1 Matrix{Gf2_3}:
 0
 0
 0
 0
 0
 0
 1
```

In [193…
```
grs_decoder(GF8[7,6,5,4,3,0,0],E,M,k) # two errors, last two positions
```

Out[193]:
```
7×1 Matrix{Gf2_3}:
 0
 0
 0
 0
 0
 2
 1
```

In [194…
```
grs_decoder(GF8[7,6,5,4,0,0,0],E,M,k) # three errors, last three positions
```

Out[194]: Gf2_3[]

In [195…
```
α = gfprimitive(8)
E = [ α^i for i in 0:254]  # Let's make a cyclic GRS code this time
M = [ α^i for i in 0:254]
n = length(E)
```

```
k = 239  # the (255,239) RS code
t = div(n-k,2)
x = Polynomial(GF256[0,1])
g = prod( (x-α^i) for i in 1:n-k )  # the generator polynomial
println("Decoding the ($(n),$(k)) cyclic GRS code with generator polynomial $(g).")
TRIALS = 10000 # let's do this number of trials with errors in t random positions
FAILURES = 0
ERRORS = 0
for trial in 1:TRIALS
    u = GF256.(rand(0:255,k)) # generate a random k-symbol message
    v = coeffs(Polynomial(u)*g)  # encode by multiplying by the generator polynomia
    while length(v) < n  # it could happen that we get a low degree, so pad with ze
        push!(v,zero(GF256))
    end
    y = copy(v)  # uncorrupted received word
    for j in 1:t
        y[rand(1:n)] += GF256(rand(0:255))  # random error in a random position
    end
    e = grs_decoder(y,E,M,k)
    if length(e) == 0
        FAILURES += 1
    else
        y -= e
        if (vec(y) != v)
            ERRORS += 1
        end
    end
end
println("After $(TRIALS) trials, $(ERRORS) errors and $(FAILURES) failures were enc
```

```
Decoding the (255,239) cyclic GRS code with generator polynomial 79 + 44*x + 81*x^2
+ 100*x^3 + 49*x^4 + 183*x^5 + 56*x^6 + 17*x^7 + 232*x^8 + 187*x^9 + 126*x^10 + 104*
x^11 + 31*x^12 + 103*x^13 + 52*x^14 + 118*x^15 + 1*x^16.
After 10000 trials, 0 errors and 0 failures were encountered.
```

---

# 4. Binary BCH Codes

A cyclic GRS code $C_{RS}$ over a field $\mathbb{F}_{q^m}$ is a GRS code with code locators defined in terms of a primitive $n$th root of unity $\alpha$ in $\mathbb{F}_{q^m}$, i.e., an element $\alpha$ of multiplicative order $n$. The code has code locators $\mathcal{E} = (1, \alpha, \alpha^2, \dots, \alpha^{n-1})$ and parity-check-matrix column multipliers $\mathcal{M} = (1, \alpha^b, \alpha^{2b}, \dots, \alpha^{(n-1)b})$. Such a code is cyclic with generator polynomial $$ g(x) = (x-\alpha^b)(x-\alpha^{b+1})\dots (x-\alpha^{b+d-2}), $$ where $d$ is the minimum Hamming distance of the code. A vector $c\in \mathbb{F}_{q^m}^n$ is a codeword if and only if $g(x)\mid c(x)$ if and only if $$ c(\alpha^b) = c(\alpha^{b+1}) = \dots = c(\alpha^{b+d-2}) = 0 $$ where $c(x)$ is the associated polynomial of the codeword $c$.

In general, if $C$ is a cyclic code over $\mathbb{F}_q$ (which is a subfield of $\mathbb{F}_{q^m}$) of length $n$ with generator polynomial $g(x)$ and there exist integers $b\geq 0$ and $\delta\geq 2$ such that $$ g(\alpha^b) = g(\alpha^{b+1}) = \dots = g(\alpha^{b+\delta-2}) = 0, $$ then $d_{\mathrm{min}}(C)\geq \delta$.

The minimum distance of the smallest RS code containing a given cyclic code $C$ is called the **design distance** of $C$. The actual minimum distance of $C$ is at least as large as the design distance. Thus, if we wish to design a $t$-error correcting code, we choose $g(x)$ so that it has $2t$ consecutive zeros, i.e., design distance $2t + 1$. This is the main idea in development of Bose–Chaudhuri–Hocquenghem (BCH) codes. To design a BCH code over $\mathbb{F}_q$ of a minimum Hamming distance $\geq 2t+1$, we form the generator polynomial of the code by taking the product of as many minimal polynomials of powers of $\alpha$ in a way that $g(x)$ has $2t$ consecutive powers of $\alpha$ as its roots. Note, $\alpha$ is some primitive $n$th roots of unity (i.e., an element of multiplicative order $n$) in some extension field $\mathbb{F}_{q^m}$. An example is given below (see Chapter 8 of the Lecture Notes for more examples):

**Example:** Consider the class of binary cyclic codes of length 15. We can find a primitive 15th root of unity $\alpha$ as the primitive element in $\mathbb{F}_{16}$. The conjugacy classes in the integer powers of $\alpha$ with respect to $\mathbb{F}_2$ are $$ \{\alpha^0\},\,\{\alpha^5, \alpha^{10}\},\,\{\alpha, \alpha^2, \alpha^4, \alpha^8\},\,\{\alpha^3, \alpha^6, \alpha^{12}, \alpha^{9}\},\,\{\alpha^7, \alpha^{14}, \alpha^{13}, \alpha^{11}\}. $$

By taking $g(x) = M_{\alpha}(x)M_{\alpha^3}(x)$, we see that $g(\alpha^i) = 0$ for all $i\in\{1,2,3,4,8,9,12\}$ which contains 4 consecutive powers of $\alpha$. Hence, the binary BCH code obtained this way has a minimum distance of at least 5.

# Exercise 4:

---

1. Implement a function `bch_generator(a::Gf2,t::Int,b::Int)` that takes an element `a` of multiplicative order $n$ from some binary field, and integer `t` and an integer `b` and outputs a generator polynomial of least possible degree having $a^b, a^{b+1}, \ldots, a^{b+2t-1}$ as roots. **Hint:** You can use your implementation in Exercise 0 to find the minimal polynomial of powers of `a` with respect to $\mathbb{F}_2$. The generator polynomial is a product of such minimal polynomials, but each such minimal polynomial needs to be included only once. To see whether the minimal polynomial for $a^i$ has already been included in a candidate $g(x)$, simply evaluate $g(a^i)$ to see if $a^i$ is a zero. Test your implementation by running the following cell.

2. Copy your function `grs_decoder` and modify it to create a function `bch_decoder(y::AbstractVector{GF2},a::Gf2,t::Int,b::Int)` that returns a binary error pattern vector corresponding to the received vector `y` for the code with

generator polynomial given by `bch_generator(a,t,b)`. **Hint:** the binary BCH code is a subcode of the RS code with a generator polynomial having $2t$ consecutive powers of `a` as zeros, starting at $a^b$. There is no need to invoke the Forney formula, since there is only one possible error value. The code locators and parity-check column multipliers can be computed from `a` and `b`.

In [196…

```
function bch_minpoly(a::Gf2)
    T = eltype(a)
    x = Polynomial(T[0,1])  # this gives the monomial x
    roots = conjugates(a)
    poly = prod((x - r) for r in roots)
    cp = coeffs(poly)

    # reduce the coeffs
    y = [c == T(0) ? GF2(0) : GF2(1) for c in cp]

    poly = Polynomial(y,:x)
    return poly
end
```

Out[196]: `bch_minpoly (generic function with 1 method)`

In [197…

```
function bch_generator(a::Gf2,t::Int,b::Int)
    T = GF2
    nroots = 2*t
    gx = Polynomial(T[1],:x)
    for i=1:nroots
        j = b+(i-1)
        r = a^j
        if gx(r) != T(0)
            mx = bch_minpoly(r)
            gx = gx * mx
        end
    end
    return gx
end
```

Out[197]: `bch_generator (generic function with 1 method)`

In [205…

```
α = gfprimitive(4)
bch1 = bch_generator(α,2,1)  # we expect minpoly(alpha)*minpoly(alpha^3)
```

Out[205]: $1 + 1 \cdot x^4 + 1 \cdot x^6 + 1 \cdot x^7 + 1 \cdot x^8$

In [199…

```
bch2 = minpoly(α) * minpoly(α^3)
```

Out[199]: $1 + 1 \cdot x^4 + 1 \cdot x^6 + 1 \cdot x^7 + 1 \cdot x^8$

In [200…

```
bch1 == bch2
```

Out[200]: `true`

In [289…

```
function bch_syndrome(y::AbstractVector{<:Gf2},E::AbstractVector{<:Gf2},M::Abstract
    n = length(E)
    d = n-k+1
    T = eltype(E)
    nr = d-1
    lp = 0:1:nr-1

    H = Array{T}(undef,nr,n)
    for i=1:n
        tmp = M[i]*E[i].^lp
        H[:,i] = tmp
    end

    synd = T.(H*y)
    sx = Polynomial(synd)
    return sx
end
```

Out[289]: bch_syndrome (generic function with 1 method)

In [310…

```
function bch_decoder(y::AbstractVector{GF2},a::Gf2,t::Int,b::Int)

    dmin = 2*t + 1
    n = length(y)
    # error locator
    E = a.^(0:1:(n-1))
    # column multiplier
    M = a.^(b*(0:1:(n-1)))

    sx = bch_syndrome(y,E,M,k)
    T = eltype(E)
    v_empty = T[]
    if sx == T(0)
        return v_empty
    end

    Lx, Gx = mod_eea(sx,dmin) #Error-locator and Error-evaluator polynomials

    err = zeros(T,n,1)
    for j=1:n
       alpha = E[j]
       mu = M[j]
       if (Lx(alpha^(-1)) == T(0))
            DL = D(Lx)
            DLa = DL(alpha^(-1))
            err[j] = -(alpha / mu)*(Gx(alpha^(-1))/DLa)
        end
    end

    if degree(Lx) != length(err[err .!= T(0)])
        return v_empty
    end
    return err

end
```

Out[310]: bch_decoder (generic function with 1 method)

In [314…

```
α = gfprimitive(4)
t = 2
b = 1
g = bch_generator(α,t,b)
n = gforder(α)
k = n-degree(g)
println("Decoding the ($(n),$(k)) binary cyclic BCH code with generator polynomial
TRIALS = 10000  # let's do this number of trials with errors in t random positions
FAILURES = 0
ERRORS = 0
for trial in 1:TRIALS
    u = GF2.(rand(0:1,k)) # generate a random k-bit message
    v = coeffs(Polynomial(u)*g)  # encode by multiplying by the generator polynomia
    while length(v) < n  # it could happen that we get a low degree, so pad with ze
        push!(v,zero(GF2))
    end
    y = copy(v)  # uncorrupted received word
    for j in 1:t
        y[rand(1:n)] += GF2(1)  # bit error in a random position
    end
    e = bch_decoder(y,α,t,b)
    if length(e) == 0
        FAILURES += 1
    else
        y -= e
        if (vec(y) != v)
            ERRORS += 1
        end
    end
end
println("After $(TRIALS) trials, $(ERRORS) errors and $(FAILURES) failures were enc
```

I would not expect to see any decoding failures, since at most t errors are added. Something is wrong with your implementation.

8/10

```
Decoding the (15,7) binary cyclic BCH code with generator polynomial 1 + 1*x^4 + 1*x
^6 + 1*x^7 + 1*x^8.
After 10000 trials, 0 errors and 9 failures were encountered.
```

---

This completes Numerical Exercise 2. Following the same directions as in Exercise 0, convert to html, and then print to PDF to create a file that can be uploaded on Quercus on or before the due date.

In [ ]: