university of groningen / faculty of mathematics and natural sciences / mathematics

# Locally Recoverable Codes

## Examples Arising from Galois Covers of Curves

Master Project Mathematics

August 2017

Student: D.R.G. Hulzebos

First supervisor: Prof.dr. J. Top

Second supervisor: Dr.ir. R.W.C.P. Verstappen

**Abstract**

In this research Locally Recoverable Codes (LRC's) are studied. First a definition for LRC's is given. After that a method for calculating an upper bound for the locality is described. Then a method to construct LRC's is studied. A few examples of this are being described. Finally this method is being extended to a method where one constructs LRC's with multiple recovering sets. There will also be an example given for this construction. Besides that several `MAGMA` implementations are presented for the computations that are needed.

# Contents

**1 Introduction** **4**

**2 Calculating a bound for the locality** **5**

**3 Geometric construction of locally recoverable codes** **7**
3.1 An "example" over $\mathbb{F}_2$ . . . . . 11
3.2 An example over $\mathbb{F}_3$ . . . . . 13
3.3 An example over $\mathbb{F}_5$ . . . . . 14

**4 Multiple recovering sets** **15**
4.1 An example over $\mathbb{F}_3$ . . . . . 17

**5 Conclusion** **18**

**A Magma implementation for finding an upper bound for the locality of a code** **21**

**B Magma implementation for creating code with designated locality with Hermitian curves** **22**

**C Finding the extension with genus 50 and 40 rational points** **23**

**D Magma implementation for finding the extension with genus 50 and 40 rational points** **27**

**E Magma implementation for creating a code of length 40 over $\mathbb{F}_2$** **28**

**F Magma implementation for creating a code of length 39 over $\mathbb{F}_3$** **31**

**G Magma implementation for creating a code of length 50 over $\mathbb{F}_5$** **38**

**H Magma implementation for creating a code of length 36 over $\mathbb{F}_3$ with 2 recovering sets** **40**

# 1 Introduction

In today's society a lot of storage systems are used to store large amounts of data. As a result, many recoveries have to be performed to account for the failures that occur. There are many options known to perform such recoveries. However, most of them require a large storage overhead. For example, a method that is used in practice is the threefold replication. This has a 200% overhead, which is not very desirable.
One of the most common failures is the single node failure. Therefore the code used should be designed in such a way that repairing a node failure can be done efficiently. One method that does this without much overhead is the method that uses Locally Recoverable Codes. Besides the usual parameters $n$, $k$, and $d$ of any linear code, there is another parameter that will be of interest for LRC's. This parameter will be the locality $r$, which is defined in the following way.

**Definition 1.1.** *A code $\mathcal{C} \subset \mathbb{F}_q^n$ is locally recoverable with locality $r$ if for every $i \in \{1, 2, \ldots, n\}$ there exists a subset $I_i \subset \{1, 2, \ldots, n\} \setminus \{i\}$ of cardinality at most $r$ and a function $\phi_i$ such that for every codeword $c = (c_1, \ldots, c_n) \in \mathcal{C}$ we have*

$$c_i = \phi_i(\{c_j, j \in I_i\}). \tag{1.1}$$

*This definition can also be rephrased as follows. Given $a \in \mathbb{F}_q$, consider the sets of codewords*

$$\mathcal{C}(i, a) = \{c \in \mathcal{C} : c_i = a\}, \qquad i \in \{1, 2, \ldots, n\}.$$

*The code $C$ is said to have locality $r$ if for every $i$ there exists a subset $I_i \subset \{1, 2, \ldots, n\} \setminus \{i\}$ of cardinality at most $r$ such that the restrictions of the sets $\mathcal{C}(i, a)$ to the coordinates in $I_i$ for different $a$ are disjoint:*

$$\mathcal{C}_{I_i}(i, a) \cap \mathcal{C}_{I_i}(i, a') = \emptyset, \qquad a \neq a'. \tag{1.2}$$

Such codes are called *locally recoverable codes (LRC's)*. In this definition, the subset $I_i$ is called the *recovering set* of the coordinate $i$. In the case that all the recovering sets are of the same size, then we have a code with *all-symbol locality*.

We can see from this definition that if $d \geq 2$, then $r \leq k$. For practical usage, we are looking for codes with small locality, high rate $\frac{k}{n}$ and large minimum distance $d$. However, there are some constraints on these parameters, as we can see in Proposition 1.2, which was proven in [GHSY12] and [PD14].

**Proposition 1.2.** *Let $\mathcal{C}$ be an $[n, k, d]$ LRC code with locality $r$. The rate of $\mathcal{C}$ satisfies*

$$\frac{k}{n} \leq \frac{r}{r+1}, \tag{1.3}$$

*and the minimum distance of $\mathcal{C}$ satisfies*

$$d \leq n - k - \left\lceil \frac{k}{r} \right\rceil + 2. \tag{1.4}$$

As we can see, if we take $r = k$, then we obtain the Singleton bound. If the parameters of a code are such that the inequality from Equation (1.4) becomes an equality, then we call it an *optimal LRC*.

Let us look at an example to get a better feeling of the theory that was just explained.

**Example 1.3.** In this example we consider the linear code $C$ over $\mathbb{F}_3$ given by the generator matrix

$$G = \begin{pmatrix} 1 & 0 & 0 & 2 & 0 & 2 & 2 & 2 \\ 0 & 1 & 0 & 2 & 0 & 2 & 1 & 0 \\ 0 & 0 & 1 & 2 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \end{pmatrix}.$$

This code turns out to be an [8,4,3]-code. But besides that, it also has locality 3. The recovering sets are given by $\{1,2,3,4\}\setminus\{i\}$ if $1 \leq i \leq 4$ and $\{5,6,7,8\}\setminus\{i\}$ if $5 \leq i \leq 8$. In fact, the code is constructed in such a way that for a codeword $c = (c_1, \ldots, c_8)$ we have that $c_1 + c_2 + c_3 + c_4 = 0$ and $c_5 + c_6 + c_7 + c_8 = 0$. From this we can immediately see the recovering sets, and the function $\phi_i$ to reconstruct a missing element $c_i$. For example, $c_1 = \phi_1(c_2, c_3, c_4) = 2c_2 + 2c_3 + 2c_4$. Since the recovering sets have the same size, we have a code with all-symbol locality. Furthermore we see that Proposition 1.2 holds in the present case, since

$$\frac{k}{n} = \frac{1}{2} \leq \frac{3}{4} = \frac{r}{r+1}$$

and

$$d = 3 \leq 6 = n - k - \left\lceil \frac{k}{r} \right\rceil + 2.$$

The inequalities from Proposition 1.2 are not the only ones known. A lot more have been found, for example by [GHSY12], [PD14], [RPDV16], [TBF16] and [WZ14]. There have also been methods proposed to construct codes with certain locality. This was for example done in [BTV16], [BHH$^+$16] and [BM16]. A lot of examples have been produced that use these construction methods, as can be seen in the same papers. A variation on LRC's is codes with multiple recovery sets. This has been investigated by [BTV16], [BHH$^+$16] and [BT17].

In this report we will first look at a naive way to compute the locality of a code to get a better feeling about locality. This will be done in Section 2. After this we take a look at how to construct LRC's in Section 3. Then we use this method to construct an example of a code over $\mathbb{F}_2$ in Section 3.1. This will lead us to an important proposition that tells us if a certain locality can be reached. Using this proposition, we construct more examples in Section 3.2 and Section 3.3. Then we will extend our method to be able to have multiple recovering sets in Section 4. We will create an example for this in Section 4.1. After that we will summarize everything in our conclusion in Section 5.

## 2 Calculating a bound for the locality

In this section we will try to deepen our understanding of what locality means. We do this by constructing a naive method of calculating an upper bound for the locality of a code. We will see that this is indeed an upper bound, but for large codes the computation times will be extremely high, and thus this will not be an efficient method.

From Definition 1.1 we obtain an idea of what locality is. To put it simply, it tells us how many points of a certain codeword we need to retrieve an erased entry. The recovering set shows us which points we should take in order to retrieve this erased entry. Then the function $\phi_i$ tells us how we should combine these points in order to actually get this erased entry.

If we now consider a code $\mathcal{C}$, then we can write down its generator matrix $G$ of size $k \times n$. But there is also the dual code $D$ with parity check matrix $H = (h_{ij})$ of size $(n-k) \times n$. By definition, the rows generator matrix of the dual code give us equations of the form

$$h_i \cdot c = h_{i,1}c_1 + h_{i,2}c_2 + \cdots + h_{i,n}c_n = 0, \qquad i = 1, \ldots, n-k, \tag{2.1}$$

where $c = (c_1, \ldots, c_n) \in \mathcal{C}$ and $h_i$ is a row of the parity check matrix $H$. If $d \geq 2$, then the parity check matrix $H$ has no columns with all zeros. This means that there always is an $i \in \{1, \ldots, n-k\}$ such that

$$c_\ell = \frac{1}{h_{i,\ell}} \left( h_{i,1}c_1 + \cdots + h_{i,\ell-1}c_{\ell-1} + h_{i,\ell+1}c_{\ell+1} + \cdots\cdots + h_{i,n}c_n \right), \qquad l = 1, \ldots, n.$$

Thus for every $c_\ell$ we can write an equation to retrieve the value of $c_\ell$, which is exactly what the locality is about. But many of the $h_{i,j}$ could be zero, which means that the locality of a code is bounded by the maximum weight minus 1 of the words $h_i = (h_{i,1}, \ldots, h_{i,n})$, which are the rows of the parity check matrix $H$. However, $H$ is not fixed, and there are many different possibilities. Thus a better upper bound would be

$$r \leq \min_{H} \max_{h_i \in H} wt(h_i) - 1, \tag{2.2}$$

where $wt(h_i)$ denotes the weight of $h_i$. Using this method, we computed the locality of several cyclic codes and we looked at the time it took to do these computations. The algorithm implemented in `MAGMA` can be found in Appendix A. Some results are listed in Table 2.1. These results have been plotted in Figure 2.1.

Table 2.1: Computation time for calculating the locality using Equation (2.2)

| $q$ | $n$ | $k$ | $d$ | function | $r \leq$ | Cputime (s) |
|---|---|---|---|---|---|---|
| 2 | 10 | 1 | 10 | $x+1$ | 1 | 0.000 |
| 2 | 10 | 2 | 5 | $(x+1)^2$ | 2 | 0.000 |
| 2 | 10 | 4 | 4 | $x^4+x^3+x^2+x+1$ | 3 | 0.380 |
| 2 | 10 | 5 | 2 | $(x^4+x^3+x^2+x+1)(x+1)$ | 1 | 204.780 |
| 2 | 10 | 6 | 2 | $(x^4+x^3+x^2+x+1)(x+1)^2$ | ? | over 90 minutes |
| 2 | 20 | 1 | 20 | $x+1$ | 1 | 0.000 |
| 2 | 20 | 2 | 10 | $(x+1)^2$ | 2 | 0.000 |
| 2 | 20 | 3 | 10 | $(x+1)^3$ | 3 | 0.020 |
| 2 | 20 | 4 | 5 | $(x+1)^4$ | 4 | 1.120 |
| 2 | 20 | 4 | 8 | $x^4+x^3+x^2+x+1$ | 4 | 1.100 |
| 2 | 20 | 5 | 4 | $(x^4+x^3+x^2+x+1)(x+1)$ | 3 | 647.540 |
| 2 | 20 | 6 | 4 | $(x^4+x^3+x^2+x+1)(x+1)^2$ | ? | over 90 minutes |
| 2 | 50 | 4 | 20 | $x^4+x^3+x^2+x+1$ | 4 | 2.180 |
| 2 | 50 | 24 | 4 | $(x^{20}+x^{15}+x^{10}+x^5+1)(x^4+x^3+x^2+x+1)$ | ? | over 90 minutes |
| 3 | 10 | 1 | 10 | $x+1$ | 1 | 0.000 |
| 3 | 10 | 1 | 10 | $x+2$ | 1 | 0.000 |
| 3 | 10 | 2 | 5 | $(x+1)(x+2)$ | 2 | 0.000 |
| 3 | 10 | 4 | 4 | $x^4+x^3+x^2+x+1$ | 3 | 404.200 |
| 3 | 10 | 4 | 4 | $x^4+2x^3+x^2+2x+1$ | 3 | 470.240 |
| 3 | 10 | 5 | 4 | $(x^4+x^3+x^2+x+1)(x+1)$ | ? | over 90 minutes |

Computer used: Standard desktop computer.

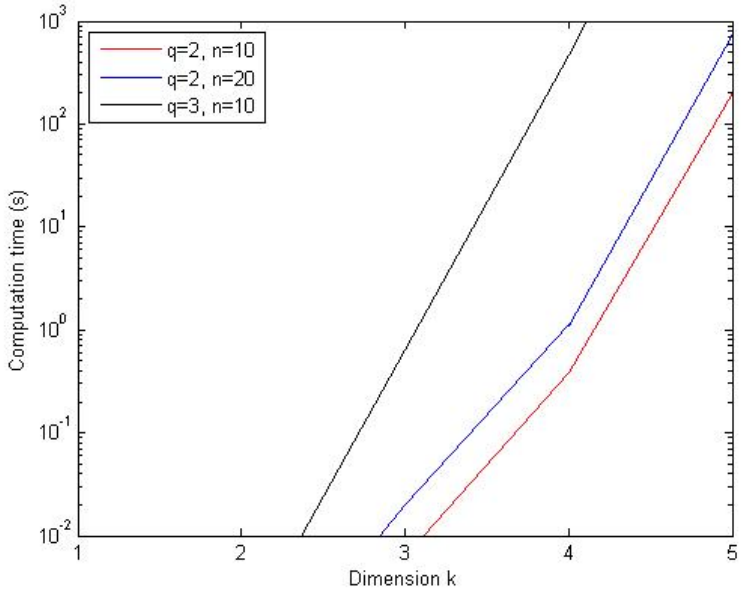



Figure 2.1: Plot of the values from Table 2.1 for $q = 2$ and $q = 3$.

As we see from Table 2.1 and Figure 2.1, the computation time increases rapidly as $k$ increases. This makes this method not very useful, since in practice we are considering codes with dimensions that can be much larger than 6. Besides that, this method only gives an upper bound for the locality, and not necessarily the exact locality. Although, in every case tested up until now this upper bound is actually equal to the exact locality of the code.
It might be possible to improve this method to compute the locality faster as explained in Appendix A, or to even find the exact locality. It could also be the case that this method already finds the exact locality, but then this has to be proven. Besides this there might be other methods to compute the locality. However, up until now we are not aware of any alternative general method. But since most codes with locality are constructed in such a way that the locality is known, this is not a main focus point within the field of LRC's. It might however be useful for the few codes that are not very large, but have unknown locality. For example, we could find an upper bound for the locality of our example from the introduction.

**Example 2.1.** If we look back at our example from Example 1.3, then we can check if our naive method finds the right locality for this code. If we do this, we see that we indeed get locality 3. The computation time that was needed was 478.580 seconds, which is similar to the results in Table 2.1.

As we can see, the method works for this small example, but it still takes a long time to compute the locality. Hence it might be more useful to construct codes in such a way that the locality is known beforehand. This is what we will be doing in Section 3.

# 3 Geometric construction of locally recoverable codes

Now that we have a better understanding of what locality means, we will look at how we can construct locally recoverable codes. First we will explain how this is done in the general case. After that we will look at explicit examples in Section 3.1, Section 3.2 and Section 3.3.

The way that we will construct codes is described in for example [BTV16], [BHH$^+$16] and [BM16]. For this we use theory from [TVN07]. The method that we use is to construct a Goppa code. For that we will need a function to evaluate in. So we have to choose a function such that we get a certain dependence between sets of $r+1$ points if the desired locality is equal to $r$. If a symbol of such a set gets deleted, then we can retrieve it by using the $r$ other symbols in that set.
To get such sets of $r+1$ points, we will take a map of degree $r+1$. Therefore we let $X$ and $Y$ be smooth projective absolutely irreducible curves over $\mathbb{F}_q$. We assume that between these curves we have a rational separable map $g : X \to Y$ of degree $r+1$. Then $g^* : \mathbb{F}_q(Y) \to \mathbb{F}_q(X)$ is the field homomorphism given by $g^*(f) = f \circ g$. Then $g^*(\mathbb{F}_q(Y)) \subset \mathbb{F}_q(X)$, and the degree of $g$ is defined as the degree of the extension $\mathbb{F}_q(X) \supset g^*\mathbb{F}_q(Y)$.
Now that we have such a map $g : X \to Y$ of degree $r+1$, and a $g^*$ defined as above, let $S = \{Q_1, \ldots, Q_s\}$ be the set of points of $Y(\mathbb{F}_q)$ that split completely in the cover $X \to Y$. That means that for each $Q_i$ there are $r+1$ points $P_{i,0}, \ldots, P_{i,r}$ in $X(\mathbb{F}_q)$ that map to $Q_i$.

What we now do is creating a vector space $V$ over $\mathbb{F}_q$ with $V \in \mathbb{F}_q(X)$, with generators $g^*(f_j)e_i$ for $i = 0, \ldots, r-1$ and $j = 1, \ldots, m$. Here the $e_i$ are elements of $\mathbb{F}_q(X)$ that are linearly independent over $\mathbb{F}_q(Y)$, and the $f_j$'s are some $\mathbb{F}_q$-linearly independent functions in $\mathbb{F}_q(Y)$. If we now evaluate in all the $P_{ij}$'s, then we get a Goppa code. That is, we take the map

$$\begin{aligned} ev : V &\to \mathbb{F}_q^{(r+1)s}, \\ F &\mapsto (F(P))_{P \in g^{-1}(S) \subset X(\mathbb{F}_q)}. \end{aligned}$$

If everything is defined correctly, then the result will be a locally recoverable code with locality at most $r$. The reason that we get locality at most $r$ comes from the fact that we let $i$ run from 0 up to $r-1$, and not to $r$, even though we have an extension of degree $r+1$. Therefore we can now recover a point if it gets erased. This is done by looking at a set $P_{i,0}, \ldots, P_{i,r}$ for fixed $i$, and assume that one of the evaluations in these points gets erased. If we evaluate $g^*(f_j)$, then we get the same result for every $P_{ij}$ with fixed $i$. Thus for these sets only the $e_i$ differ. But since these are $r$ linearly independent elements over $\mathbb{F}_q(Y)$, we can use the $r$ known values to reconstruct the deleted symbol by using interpolation. Hence the recovering sets are given by $\{g^{-1}(Q_i)\} \setminus \{P_{i,j}\}$, where $P_{i,j}$ is the erased symbol. If we would have let $i$ run from 0 to $r$, then this would not have been possible.

All that is left now is to make sure that everything is defined correctly. To that extend, we have to make sure that we do not encounter poles while evaluating. Therefore there are two things we need to check. First of all we should be sure that the $g^*(f_j)$'s do not have poles at the points $P_{ij}$. But that means that the polar sets of the $f_j$'s should be disjoint from the points $Q_i$. Thus this is something we have to keep in mind while choosing the $f_j$'s.
Secondly, we should have that the $e_i$ do not have poles when we evaluate in the points $P_{ij}$. Hence the polar sets of the $e_i$'s should be disjoint from the $P_{ij}$.

As an idea to make sure that the above requirements are satisfied, we could use the following construction. Take an effective divisor $D$ on $Y$ of degree $\ell \geq 1$. This divisor should be chosen such that its support is disjoint from the points $Q_i$. Then the $\mathbb{F}_q$-vector space $L(D)$ has a basis $\{f_1, \ldots, f_m\}$, which will be our $f_j$'s. By this construction we are certain that the polar sets of the $f_j$'s are disjoint from the points $Q_i$, and thus the first requirement is satisfied.
To satisfy the second requirement we can choose the $e_i$ in a specific way. Since $g$ is a rational separable map, the primitive elements theorem implies that there exists a function $x \in \mathbb{F}_q(X)$

such that $\mathbb{F}_q(X) = \mathbb{F}_q(Y)(x)$ and

$$x^{r+1} + b_r x^r + \cdots + b_1 x + b_0 = 0, \tag{3.1}$$

where $b_i \in \mathbb{F}_q(Y)$. This function $x$ can be seen as a function $x : X \to \mathbb{P}^1$, with a certain degree which we will denote by $h$. Then we can set $e_i = x^i$. In this case the polar sets of the $e_i$'s are indeed disjoint from the $P_{ij}$'s.

Based on the construction we can say several things about the parameters of the code. This is done in Theorem 3.1.

**Theorem 3.1.** *The code constructed as above will have parameters*

$$\begin{aligned} n &= (r+1)s, \\ k &= rm \geq r(\ell - g_Y + 1), \\ d &\geq n - \ell(r+1) - (r-1)h, \end{aligned}$$

*Provided that the right-hand side of the inequality for $d$ is a positive integer. Here $g_Y$ denotes the genus of the curve $Y$.*

*Proof.* The length of the code is determined by the number of points we evaluate in. These are all the points $P_{ij}$ for $i = 1, \ldots, s$, and thus we find that $n = (r+1)s$.
The minimum distance $d$ is determined by the number of zeros when we evaluate linear combinations of the functions $g^*(f_j)x^i$ in the points $P_{ij}$. The number of zeros is at most equal to the number of zeros in $g^*(f_j)$ plus the number of zeros in $x^{r-1}$. Based on the degrees of these functions we see that these are equal to respectively $\ell(r+1)$ and $(r-1)h$, where $h = \deg(x)$. These zeros could have a multiplicity greater than 1, thus we have at most $\ell(r+1) + (r-1)h$ zeros when we evaluate in the $P_{ij}$'s, and hence the minimum distance is greater or equal to $n - \ell(r+1) - (r-1)h$.
Since we assume that $d \geq 1$, it follows that $n > \ell(r+1) + (r-1)h$, and therefore the map $ev$ is injective. Hence the dimension of the code is equal to the dimension of the vector space $V$. Since this vector space is spanned by $g^*(f_j)e_i$ for $i = 0, \ldots, r-1$ and $j = 1, \ldots, m$, we find that $k = rm$. By applying the Riemann-Roch theorem on the divisor $D$ it follows that $m \geq \ell - g_Y + 1$, which proves the inequality for the dimension $k$. □

**Example 3.2.** In this example we will look at the influence of different values of $q$ and $l$ on the computation time. For this example we look at Hermitian curves, which are curves of the form $y^{q+1} = x^q + x$ over $\mathbb{F}_q$. We create codes with locality for different values of $q$ and $\ell$. While doing that we measure the computation time that was needed to construct that code. The results can be found in Table 3.1. The `MAGMA` algorithm that was used to create this table can be found in Appendix B.

From this table we can conclude several things. First of all we see that the $n$ remains the same if we change $\ell$. This makes sense, since $\ell$ does not occur in the formula for $n$ given in Theorem 3.1.
We also see that $k$ increases with a constant factor of $r = q-1$ when $\ell$ increases. This is because $\ell$ is the degree of the effective divisor $D$, and increasing this also increases the dimension $m$ of the Riemann-Roch space $\mathcal{L}(D)$. Then we can see from Theorem 3.1 that increasing the $m$ results into an increase of $k$.
For the minimum distance we notice something similar, since the upper bound for this decreases with a factor $r + 1 = q$ every time $\ell$ increases by 1. This can also be explained by looking at

Theorem 3.1. If we look at the upper bound given for $d$, then we see indeed that this behavior is described there.
Something else that we can see is that if $q$ increases, then $n$ and the computation time also both increase. This makes sense, since a higher $q$ means that we have more rational points on the curve. In particular, for these Hermitian curves the number of rational points is equal to $q^3$. And since we evaluate in all these points, we have that $n = q^3$. But evaluating in more points results in a larger computation time. Besides that, a larger $q$ means that we will evaluate in more functions which also increases the computation time.
Another thing that stands out is that if $l$ gets higher, then the $k$ gets higher and this also has a negative effect on the computation time. The reason for this is that if $\ell$ increases, then we have to evaluate more functions. This results in both a larger $k$ and a larger computation time.
Note that these things could already have been seen from Theorem 3.1. However, this table also shows the consequences for the computation time. And as we can see, this computation time increases when the values of $q$ or $\ell$ increase. This means that finding large codes can take a longer time, which is something to keep in mind when creating such codes. However, these computation times are a lot better than the ones needed to compute the locality of a code, as shown in Table 2.1.

Table 3.1: Computation times for different values of $q$ and $\ell$.

| $q$ | $\ell$ | Irreducible pol. | $n$ | $k$ | $d \geq$ | $r$ | Cputime (s) |
|---|---|---|---|---|---|---|---|
| 17 | 1 | $t^2 + 15t + 6$ | 4913 | 32 | 4626 | 16 | 23.000 |
| 17 | 2 | $t^2 + 12t + 8$ | 4913 | 48 | 4609 | 16 | 24.700 |
| 17 | 3 | $t^2 + 2t + 7$ | 4913 | 64 | 4592 | 16 | 26.220 |
| 17 | 4 | $t^2 + 13t + 1$ | 4913 | 80 | 4575 | 16 | 27.920 |
| 17 | 5 | $t^2 + 5t + 14$ | 4913 | 96 | 4558 | 16 | 29.280 |
| 17 | 6 | $t^2 + 6t + 14$ | 4913 | 112 | 4541 | 16 | 30.840 |
| 17 | 7 | $t^2 + 6t + 15$ | 4913 | 128 | 4524 | 16 | 32.340 |
| 17 | 8 | $t^2 + 16t + 16$ | 4913 | 144 | 4507 | 16 | 33.880 |
| 17 | 9 | $t^2 + 8t + 2$ | 4913 | 160 | 4490 | 16 | 35.540 |
| 17 | 10 | $t^2 + 2t + 13$ | 4913 | 176 | 4473 | 16 | 37.140 |
| 23 | 1 | $t^2 + 16t + 3$ | 12167 | 44 | 11640 | 22 | 117.960 |
| 23 | 2 | $t^2 + 7t + 8$ | 12167 | 66 | 11617 | 22 | 121.960 |
| 23 | 3 | $t^2 + 12t + 8$ | 12167 | 88 | 11594 | 22 | 127.660 |
| 23 | 4 | $t^2 + 3t + 21$ | 12167 | 110 | 11571 | 22 | 133.900 |
| 23 | 5 | $t^2 + 15t + 9$ | 12167 | 132 | 11548 | 22 | 140.440 |
| 23 | 6 | $t^2 + 21t + 19$ | 12167 | 154 | 11525 | 22 | 147.120 |
| 23 | 7 | $t^2 + t + 22$ | 12167 | 176 | 11502 | 22 | 152.040 |
| 23 | 8 | $t^2 + 9$ | 12167 | 198 | 11479 | 22 | 157.260 |
| 23 | 9 | $t^2 + 15t + 17$ | 12167 | 220 | 11456 | 22 | 164.820 |
| 23 | 10 | $t^2 + t + 18$ | 12167 | 242 | 11433 | 22 | 172.100 |
| 31 | 3 | $t^2 + 30t + 2$ | 29791 | 120 | 28770 | 30 | 368.080 |
| 31 | 13 | $t^2 + 11t + 21$ | 29791 | 420 | 28560 | 30 | 553.760 |

Now we will look at some examples of codes that we can construct using this method. In these examples we will restrict ourselves to a special case. This will be the case in which we construct one or more Artin-Schreier extensions to get our curves. Recall that an Artin-Schreier extension is an extension of the form

$$x^q - x - F = 0.$$

If we make sure that the function $F$ has zeros at the points in $S$, then these points will split completely. Thus by choosing a curve $X$ with many rational points, we can make sure that the extension $Y$ has the maximal number of rational points by taking a suitable Artin-Schreier extension. Besides that, the $x$ is precisely the $x$ such that $\mathbb{F}_q(X) = \mathbb{F}_q(Y)(x)$. Hence Artin-Schreier extensions provide exactly what we are looking for, and therefore we will restrict ourselves to these extensions.

In the following part we will look at some examples which make use of these Artin-Schreier extensions. First we look at an example over $\mathbb{F}_2$ in Section 3.1. After that we will look at examples over $\mathbb{F}_3$ and $\mathbb{F}_5$ in Section 3.2 and Section 3.3 respectively.

## 3.1 An "example" over $\mathbb{F}_2$

Now that we have an understanding of codes with locality, we are interested in finding codes with good locality. If we would like the code to be applicable in practice, if would be best if this code is over $\mathbb{F}_2$. We want this code to have a reasonable length, a high minimal distance and low locality. By construction, the length of the code is determined by the number of points we evaluate a function in. Hence we need a function that has many points over $\mathbb{F}_2$. Some research about the existence of curves with many rational points has been done, and the results can for example be found in the tables of Van Der Geer and Van Der Vlugt [VDGVDV00]. The highest number of points of a curve over $\mathbb{F}_2$ that we can find in the tables of Van Der Geer and Van Der Vlugt is 40, which corresponds to a curve of genus 50. However, this curve is not written down explicitly. Therefore we first have to find this curve explicitly before we could construct a code with it. But since this is not the goal of this research, it can be found in Appendix C. The result is the equation for the extension defined by Equation (C.3).

Now we can use this extension to create our code. This will become a code over $\mathbb{F}_2$, since our extension is defined over $\mathbb{F}_2$. And since our extension has 40 rational points, we know that our code will have length $n = 40$. The things that we can vary are the dimension $k$ and the minimum distance $d$. These are determined by the divisor $D$ that we use for constructing the functions that we will evaluate in.

Now, by construction, we should have that our code has locality $r = 7$, since the degree of our extension is 8. However, it turns out that this is not the case. To show that this is indeed not the case, we will look at an example. For this example, we take the divisor

$$D = \sum_{j=1}^{7} P_j - \sum_{i=1}^{5} R_i,$$

where the $P_j$'s are the points of degree 7, and the $R_i$'s are points of degree 5. This way we should by construction get a code with parameters $[40, 14, d]$ with $d \geq 18$, and $r = 7$. However, the generator matrix that we find by using the code from Appendix E is given by

$$\begin{pmatrix}
1&1&1&0&0&0&0&1&0&0&1&1&1&1&0&0&0&0&0&0&0&0&0&0&1&1&1&0&0&0&0&0&0&0&0&0&1&0&0\\
0&0&0&1&1&1&1&0&0&0&0&0&0&0&0&0&0&0&1&1&1&1&0&0&0&0&0&1&1&1&0&0&0&0&0&0&0&0&1\\
0&0&0&0&0&0&0&0&1&1&0&0&0&0&1&1&0&0&0&0&0&0&0&0&1&1&1&0&0&0&0&0&1&1&1&1&1&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&1&0&0&0&0&1&1&0&0&0&1&1&1&1&1&1&0&0&0&0&0&1&1
\end{pmatrix}$$

The code given by this generator matrix has parameters $[40, 4, 12]$. The reason why the parameters are not the same is as follows. The rows of the generator matrix are created by evaluating in functions of the form $g^*(f_j)x^i$. In this case the $i$ runs from 0 to 6. But since we are working

over $\mathbb{F}_2$, we have that $x^i = x$ for $i = 1, \ldots, 6$. This means that we could also just have evaluated in the same functions, but only for $i = 0, 1$. This has as a consequence that the locality is not equal to 7 anymore, and therefore also the dimension $k$ and the lower bound for the minimum distance $d$ are different.

So the parameters are completely different from what we hoped them to be, and the reason for this is that we are working over $\mathbb{F}_2$. But we would like to be certain about our parameters, without having to explicitly calculate them. Luckily there is a proposition that tells us immediately whether we will obtain our expected locality, and therefore also our expected $k$ and $d$. This proposition is as follows.

**Proposition 3.3.** *Let $i$ be an integer between 1 and $s$, and suppose every $r \times r$ submatrix of the matrix*

$$M := \begin{bmatrix} e_1(P_{i,0}) & e_2(P_{i,0}) & \ldots & e_r(P_{i,0}) \\ e_1(P_{i,1}) & e_2(P_{i,1}) & \ldots & e_r(P_{i,1}) \\ \vdots & \vdots & \ddots & \vdots \\ e_1(P_{i,r}) & e_2(P_{i,r}) & \ldots & e_r(P_{i,r}) \end{bmatrix}$$

*is invertible. Let $f$ be a function on $X$ such that $f = f_a$ for some $a \in \mathbb{F}_q^s$. Then the value of $f$ on any point in the recovering set $I_i$ can be calculated from the values of $f$ on the other points in the recovering set.*

*Proof.* This was proven in [BHH+16]. First we note that any $f$ that we evaluate in can be written as

$$f = \sum_{u=0}^{r-1} e_u \sum_{v=1}^{t} a_{u,v} g^* f_v,$$

where $a_{u,v}$ is some element of $\mathbb{F}_q^s$. Thus for every $j = 0, \ldots, r$ we have that

$$\begin{aligned} f(P_{i,j}) &= \sum_{u=0}^{r-1} e_u(P_{i,l}) \sum_{v=1}^{t} a_{u,v} g^* f_v(P_{i,l}) \\ &= \sum_{u=0}^{r-1} e_u(P_{i,l}) \sum_{v=1}^{t} a_{u,v} f_v(Q_i) \\ &= \sum_{u=0}^{r-1} c_u e_u(P_{i,l}), \end{aligned}$$

where

$$c_u = \sum_{v=1}^{t} a_{u,v} f_v(Q_i).$$

But every value for $e_u(P_{i,j})$ is known for every $u$ and $j$. And since every $r \times r$ submatrix of $M$ is invertible, we can calculate the values of $c_u$ from any subset of $r$ values from the $f(P_{i,j})$. Hence we can compute the value of any of the $f(P_{i,j})$ from the values of $f$ with $l \neq j$. □

From this proposition we can immediately see why we did not get locality 7 in our example over $\mathbb{F}_2$. In our example the $e_i$ are equal to $x^i$, and thus we get the matrix

$$M := \begin{bmatrix} x^0(P_{i,0}) & x(P_{i,0}) & \ldots & x^6(P_{i,0}) \\ x^0(P_{i,1}) & x(P_{i,1}) & \ldots & x^6(P_{i,1}) \\ \vdots & \vdots & \ddots & \vdots \\ x^0(P_{i,7}) & x(P_{i,7}) & \ldots & x^6(P_{i,7}) \end{bmatrix}$$

But now we note that the second till the last column will all be the same, since $x^i = x$ for $i \geq 1$ in $\mathbb{F}_2$. So we can even say in general that a code over $\mathbb{F}_2$ cannot have a locality greater than 1 if it is constructed in this way. And even more general, but with the same reasoning, we can say that a code over $\mathbb{F}_q$ that is constructed in the same way, can have a locality of at most $q-1$. However, this only holds if we take powers of $x$ as the basis elements. For example, if we look at Example 1.3, then we see that we have a code over $\mathbb{F}_3$ with locality 3. Thus other methods might result into higher locality, since Proposition 3.3 only holds for this specific method of constructing codes with locality.

## 3.2 An example over $\mathbb{F}_3$

As we saw in Section 3.1, it is impossible to construct an example over $\mathbb{F}_2$ that has a useful locality with our method. But for other fields $\mathbb{F}_q$ it is possible to construct a code. In this paragraph we will therefore look at an example where $q = 3$.
Just as with the example over $\mathbb{F}_2$, we start with a curve from which we know the number of points. In this case we take the curve $Y : (1-a)b^5 + a^3b^3 - b + a^4 - a^2 = 0$. This is a genus 5 curve with 13 rational points $Q_i$ over $\mathbb{F}_3$. Now we create our Artin-Schreier extension. Therefore we need to find a function that has the 13 points over $\mathbb{F}_3$ as zeros. This is done by looking at the Riemann-Roch space of a divisor of the form

$$D = \sum_j P_j - \sum_{i=1}^{13} Q_i. \tag{3.2}$$

Here we have to choose points $P_j$ disjoint from the $Q_i$ such that the space $\mathcal{L}(D) \neq 0$. For example, we can choose to put the points of degree 2 here. However, this will not yet create a Riemann-Roch space with nonzero dimension. If we take these degree 2 points 9 times, our Riemann-Roch space has dimension 2, and thus we can choose a function in this space to create our extension with. The function that we will choose is the function

$$\begin{aligned} F = {} & (a+1)^{-3}(a+2)^{-2}(a^3+a^2+2a+1)^{-5}((a^4+2a^3+a^2+2a)b^4 + (a^5+a^4+2a^2+2)b^3 \\ & + a^3b^2 + (a^5+2a^4+2a^3)b + a^6 + a^5 + 2a^3 + 2a^2 + 2a + 2)a^{-11}((a^2+2)b^4 + (2a^8+2a^7 \\ & + a^6 + a^5 + a^4 + 2a^3 + a + 2)b^3 + (2a^7 + a^5 + a^4 + 2a + 1)b^2 + (a^10 + 2a^9 + a^8 + 2a^5 + a^4 \\ & + 2a^3 + a + 1)b + a^9 + a^8 + 2a^7 + 2a^6 + 2a^5 + a^4 + a^3 + a^2)(a^6 + 2a^4 + 2a^3 + a^2 + 2a + 2)^{-1} \\ & ((a^5+2a^4+a^3+2a^2+2a+1)b^4 + (a^3+2a^2+2a+1)b^3 + (a^8+a^7+a^5+2a^4+2a^3+a)b^2 \\ & + (a^7+2a^6+a^4+a^3+a^2+a+2)b + 2a^8 + 2a^7 + a^5 + 2a^4 + a^3 + 2a^2 + a + 2)((a+2)b^4 \\ & + (a^4+a^3+2a^2+2a)b^3 + (a^4+2a^3+2a^2+2a+1)b^2 + (2a^6+a^5+2a^3+a^2+a+1)b \\ & + 2a^6 + 2a^4 + 2a^2 + 2a + 1)((a+2)b^4 + (a^3+a^2+1)b^3 + (a^5+2a+2)b^2 + (2a^6+a^3+2a^2 \\ & + 2a + 1)b + 2a^5 + 2a^4 + 2a^3 + 2a^2). \end{aligned} \tag{3.3}$$

Then we create the extension $Y(a,b) \leftarrow X(a,b,x)$, where $x$ satisfies the equation $x^3 - x - F = 0$. This way the rational points will split, and the points of degree 2 will not. Therefore $X$ will have 39 rational points. Using Remark C.1 with $q = 3$, $g(Y) = 5$, $m = 1$ and $n = 9$ we can predict that the genus of this curve will be equal to 31. Now we can use this extension to create a code of length 39. The `MAGMA` implementation for this can be found in Appendix F.
Based on the choice of the divisor one can change the $k$ and $d$ of the resulting code. For example, if we choose a divisor of dimension 9, then we obtain the $[39, 18, 6]$-code with generator matrix

$$\begin{pmatrix}
1&0&0&0&0&2&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&2&0&1&0&0&0&0&0&1&0&0&2&2&1&2&0&1\\
0&1&0&0&2&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&2&0&1&0&0&0&0&0&0&0&0&0&2&1&0&0&0\\
0&0&1&0&2&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&2&0&0&1&0&0&0&0&0&1&0&2&0&0&0&0\\
0&0&0&1&0&2&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&2&0&0&1&0&0&0&0&0&1&1&2&2&0&2&1&0\\
0&0&0&0&0&0&1&0&2&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&2&0&0&0&0&0&2&1\\
0&0&0&0&0&0&0&1&2&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&2&0&1&0&0&1&2&0\\
0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&2&0&0&0&0&0&2&0&1&0&0&0&0&0&2&0&0&1&2&1&2&0&1\\
0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&2&0&0&0&0&0&1&0&2&0&0&0&0&0&2&0&0&1&2&1&2&0&1\\
0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&2&0&0&0&0&0&0&2&0&1&0&0&0&0&0&1&0&0&2&0&0&1&0&2\\
0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&2&0&0&0&0&0&0&1&2&0&0&0&0&2&1&2&0&0&1&0&2&1\\
0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&2&0&0&0&0&0&0&2&0&0&1&0&0&0&0&0&1&0&2&0&0&1&2&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&2&0&0&0&0&0&1&0&0&2&0&0&0&0&0&2&1&1&2&0&2&1&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&2&0&0&0&1&2&0&0&0&0&1&2&1&0&0&2&0&2&1\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&2&0&1&0&0&2&0&0&0&0&0&2&1&1&2&0&1&2&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&2&0&0&0&0&2&1&1&2&0&1&2&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&2&0&0&0&1&2&1&0&0&2&0&2&1\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&2&1&2&1&0&0&2&0&1&2\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&2&0&2&1&1&2&0&2&1&0
\end{pmatrix}.$$

By construction, we obtained the recovering sets

$$\begin{array}{lll}
\mathcal{A}_1=\{1,4,6\}, & \mathcal{A}_6=\{12,14,16\}, & \mathcal{A}_{10}=\{28,29,30\},\\
\mathcal{A}_2=\{2,3,5\}, & \mathcal{A}_7=\{19,20,21\}, & \mathcal{A}_{11}=\{31,32,34\},\\
\mathcal{A}_3=\{7,8,9\}, & \mathcal{A}_8=\{22,24,27\}, & \mathcal{A}_{12}=\{33,35,36\},\\
\mathcal{A}_4=\{10,13,18\}, & \mathcal{A}_9=\{23,25,26\}, & \mathcal{A}_{13}=\{37,38,39\}.\\
\mathcal{A}_5=\{11,15,17\}, & &
\end{array}$$

For every recovery set $\mathcal{A}_i=\{a_{i,1},a_{i,2},a_{i,3}\}$ it holds that $a_{i,1}+a_{i,2}+a_{i,3}=0$. These relations can be used to perform the recovery process. One can use the generator matrix to see that these relations indeed hold for all recovering sets.

### 3.3 An example over $\mathbb{F}_5$

In Section 3.2 we looked at a code over $\mathbb{F}_3$. But since this is a case at the border, as we saw in Section 3.1 that $q=2$ cannot give a good locality, it might be interesting to look at a different case. Therefore we will look in this paragraph at an example where $q=5$.
Just as in the previous examples, we choose a curve with a known number of points. This time we take the elliptic curve $E: b^2=a^3+3a$, which has genus 1 and 10 points $Q_i$ over $\mathbb{F}_5$, which is the maximum number of points for an elliptic curve over $\mathbb{F}_5$. Now we will build our Artin-Schreier extension as usual. Therefore we need to find a function that has the 10 points over $\mathbb{F}_5$ as zeros. This is done by looking at the Riemann-Roch space of a divisor of the form

$$D=\sum_j P_j-\sum_{i=1}^{10}Q_i. \tag{3.4}$$

Here we have to choose points $P_j$ such that $\mathcal{L}(D)\neq 0$, and which are unequal to the rational points of the curve $E$. For example, we can choose to put the points of degree 2 here. There are in total 10 of these, which results into a Riemann-Roch space of dimension 1. We choose the function that is given as its basis as the function for our Artin-Schreier extension. This function is given by

$$F=\frac{(a+1)(a+2)(a+3)(a+4)b}{(a^2+3)(a^2+2a+3)(a^2+3a+3)}. \tag{3.5}$$

Then we create the covering of curves $E(a,b)\leftarrow E'(a,b,x)$ over $\mathbb{F}_5$, where $x$ satisfies the equation $x^5-x-f=0$. This way the rational points will split, and the points of degree 2 will not. In particular, $E'$ will have 50 rational points. Using Remark C.1 with $q=5$, $g(E)=1$, $m=1$ and $n=10$ we find that the genus of this curve will be equal to 41. Now we can use

this extension to create a code of length 50. The `MAGMA` implementation for this is found in Appendix G.
Based on the choice of the divisor one can change the $k$ and $d$ of the resulting code. For this code with locality 4 we find that the parameters are given by $[50, 4m, 20 - 2m]$ for $1 \leq m \leq 9$, where $m$ is the dimension of the Riemann-Roch space of the chosen divisor. As an example different choices of the divisor and the resulting parameters are given in Table 3.2.

Table 3.2: Different values of $k$ and $d$ by using different divisors for a code of length 50 over $\mathbb{F}_5$

| Divisor $D$ | $\text{Dim}(\mathcal{L}(D))$ | $k$ | $d$ |
|---|---|---|---|
| `5*Places(K1,2)[1]-Places(K1,4)[2]-Places(K1,5)[1]` | 1 | 4 | 18 |
| `3*Places(K1,2)[1]-Places(K1,4)[1]` | 2 | 8 | 16 |
| `6*Places(K1,2)[1]-Places(K1,4)[2]-Places(K1,5)[1]` | 3 | 12 | 14 |
| `4*Places(K1,2)[1]-Places(K1,4)[1]` | 4 | 16 | 12 |
| `7*Places(K1,2)[1]-Places(K1,4)[2]-Places(K1,5)[1]` | 5 | 20 | 10 |
| `5*Places(K1,2)[1]-Places(K1,4)[1]` | 6 | 24 | 8 |
| `8*Places(K1,2)[1]-Places(K1,4)[2]-Places(K1,5)[1]` | 7 | 28 | 6 |
| `6*Places(K1,2)[1]-Places(K1,4)[1]` | 8 | 32 | 4 |
| `9*Places(K1,2)[1]-Places(K1,4)[2]-Places(K1,5)[1]` | 9 | 36 | 2 |
| `7*Places(K1,2)[1]-Places(K1,4)[1]` | 10 | 36 | 4 |
| `10*Places(K1,2)[1]-Places(K1,4)[2]-Places(K1,5)[1]` | 11 | 40 | 2 |
| `8*Places(K1,2)[1]-Places(K1,4)[1]` | 12 | 40 | 2 |
| `11*Places(K1,2)[1]-Places(K1,4)[2]-Places(K1,5)[1]` | 13 | 40 | 2 |

We see that the values of $k$ and $d$ are equal to $4m$ and $20 - 2m$ respectively, up to a certain point. When the dimension of the Riemann-Roch space becomes larger than 9, then the values start to behave differently. The reason for this is that we have reached the boundary. The minimum distance has to be even, and when $m = 9$, then the minimum distance is equal to 2. Since this is the lowest possible value for $d$, the values for larger dimensions do not satisfy the formulas anymore.

# 4 Multiple recovering sets

Up until now we only looked at codes with one recovering set. However, it is possible to construct codes with multiple recovering sets. The problem of constructing such codes is called the *availability problem*. This is what we will look at in this section. We start with an explanation about how one constructs such codes, using the theory from [BTV16], [BHH$^+$16], [BT17] and [MMH16]. After that we will look at an example to show the theory in practice.

First of all let us think about why we would want multiple recovering sets. The most important reason for this is because one recovering set might not be enough. For example, we might have a code which uses the second and third position to recover the first position. But if both the first and the second entry are erased, then we cannot recover these erasures. But if we would have multiple (disjoint) recovering sets, then we could have used these different recovering sets to recover our original values. This is the motivation for wanting to have multiple recovering sets. Now let us first look at the definition of multiple recovering sets.

**Definition 4.1.** *(LRC codes with availability) A code $\mathcal{C} \subset \mathbb{F}_q^n$ of size $q^k$ is said to have $t$ recovering sets if for every coordinate $i \in \{1, \ldots, n\}$ and every $c \in \mathcal{C}$ Equation (1.1) holds for pairwise disjoint subsets $I_{i,j} \subset \{1, \ldots, n\} \setminus \{i\}, |I_{i,j}| = r_j, j = 1, \ldots, t$.*

For codes with availability we have several constraints. For example we have one given in Proposition 4.2, which was proven in [WZ14] and [RPDV16].

**Proposition 4.2.** *An $[n, k, d]$-code with locality $r$ and $t$ disjoint recovering sets satisfies*

$$d \leq n - k - \left\lceil \frac{t(k-1)+1}{t(r-1)+1} \right\rceil + 2.$$

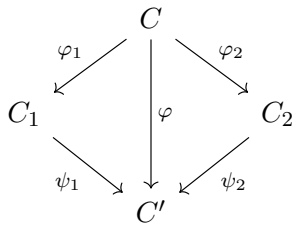
Another bound was proven by [TBF16]. This bound is shown in Proposition 4.3.

**Proposition 4.3.** *An $[n, k, d]$-code with locality $r$ and $t$ disjoint recovering sets satisfies*

$$d \leq n - \sum_{i=0}^{t} \left\lfloor \frac{k-1}{r^i} \right\rfloor.$$

Up until now, we are not aware of any general constructions known to attain the above bounds. However, it is possible to create a code with any number of recovering sets, given that the code is large enough. But for simplicity, we will focus on the case where we have two recovering sets. The cases with more recovering sets are analogous to the case with 2 recovering sets, and are for example described in [MMH16].

Let's say that we want to create a code over $\mathbb{F}_q$ with 2 recovering sets $I_{i,1}$ and $I_{i,2}$ of size $r_1$ and $r_2$ respectively for each index $i$. To create such a code we start with a curve $C$. What we want is a diagram that looks like




(4.1)

So instead of one extension, we now have two extensions such that $\psi_1 \circ \varphi_1 = \psi_2 \circ \varphi_2 = \varphi$. Here the extensions given by $\varphi_1$ and $\psi_2$ are of degree $r_1 + 1$ and the extensions given by $\varphi_2$ and $\psi_1$ are of degree $r_2 + 1$. The functions should be chosen in such a way that $s$ rational points $Q_1, \ldots, Q_s$ on $C'$ split totally. Using this diagram we can construct our code.

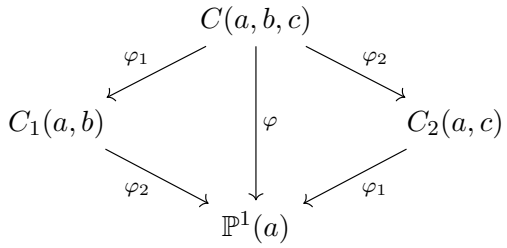
The way we construct our code is similar to the case with only one recovering set. But now, instead of taking only 1 set of linear independent functions $e_i$, we take two such sets, whose elements are denoted by $e_{1,j}$ and $e_{2,\ell}$. In this case the $e_{1,j}$'s are elements of $\mathbb{F}_q(C_1)$ that are linearly independent over $\mathbb{F}_q(C')$, and the $e_{2,\ell}$'s are elements of $\mathbb{F}_q(C_2)$ that are linearly independent over $\mathbb{F}_q(C')$. We then take a set of linearly independent functions $f_i$ in $\mathbb{F}_q(C')$. Now we create the vector space $V_2$ over $\mathbb{F}_q$ with basis given by $\varphi^*(f_i)\varphi_1^*(e_{1,j})\varphi_2^*(e_{2,\ell})$, for $i = 1, \ldots, t, j = 1, \ldots r_2 - 1$ and $\ell = 1, \ldots, r_1 - 1$. Then the image of the map

$$\begin{aligned} ev : V_2 &\to \mathbb{F}_q^n \\ F &\mapsto (F(P))_{\varphi^1(\{Q_1,\ldots,Q_s\})} \end{aligned} \tag{4.2}$$

will be our code. Just as in the case with one recovery set, we have to put some constrains on the choices of the $f_i$, $e_{1,j}$ and $e_{2,\ell}$, because we do not want to have poles in the points that

we evaluate in. To ensure that this does not happen, we have several requirements that have to be satisfied. First of all the polar sets of the $f_i$'s should be disjoint from all the rational points in $C'$. Furthermore we require that the polar sets of the $e_{1,j}$'s are disjoint from the rational points in $C_1$, and that the polar sets of the $e_{2,\ell}$'s are disjoint from the rational points in $C_2$.

Now we can make this more explicit by looking at the case where we use Artin-Schreier extensions. In that case we can for example choose our functions for the extensions such that $\psi_1 = \varphi_2$ and $\psi_2 = \varphi_1$. If we choose $C' = \mathbb{P}^1$, then we get a commutative diagram as seen below.




(4.3)

Here the $a$, $b$ and $c$ denote the variables generating the function fields of the curves.
For this construction, we can choose $e_{1,j} = b^j$ and $e_{2,\ell} = c^\ell$, where $b$ and $c$ are of degree $h_1$ and $h_2$ respectively. For the functions $f_i$ we take an effective divisor $D$ on $C'$ of degree $\ell > 1$. To satisfy the requirements on the $f_i$, we should choose $D$ such that its support is disjoint from the rational points on $C'$. Then we create the $\mathbb{F}_q$-vector space given by the Riemann-Roch space of $L(D)$, which has a basis $\{f_1, \ldots, f_m\}$. These will be our $f_i$'s. If we apply these choices to the method previously explained, then we can say something about the parameters, as done in Theorem 4.4.

**Theorem 4.4.** *The constructed code will have parameters*

$$\begin{aligned} n &= \#C(\mathbb{F}_q) = (r_1+1)(r_2+1)s, \\ k &= r_1 r_2 m, \\ d &\geq n - l(r_1+1)(r_2+1) - h_1 r_1 (r_2+1) - h_2 r_2 (r_1+1). \end{aligned}$$

The proof of theorem this is similar to the proof of Theorem 3.1. It is also given in [MMH16].

## 4.1 An example over $\mathbb{F}_3$

Now that we have an idea about how we can construct a code with multiple recovering sets, we can make an example. In this example we will create a code over $\mathbb{F}_3$ with 2 recovering sets.
We start by choosing $C'$, and build extensions from there. For this case, we set $C'$ to be equal to $\mathbb{P}^1$ over $\mathbb{F}_3$, such that it has 4 rational points. Then we will use two Artin-Schreier extensions to obtain a new curve $C$. The functions that we choose for the Artin-Schreier extensions that is of the form $\alpha^3 - \alpha - F_i = 0$ are given by

$$\begin{aligned} F_1 &= \frac{a^3 - a}{a^4 + a - 1}, \\ F_2 &= \frac{a^3 - a}{a^4 - a^3 - 1}. \end{aligned}$$

If we use this construction, then we get something as shown in Diagram (4.3).

Since the extensions are of degree 3, we will construct a code of length 36 with 2 recovering sets of locality 2. The `MAGMA` implementation that was used is shown in Appendix H. The

result is for example a $[36, 16, 4]$-code with two recovering sets of size 2 for each symbol. In that case the generator matrix is given by

$$\begin{pmatrix}
1&0&2&0&0&0&2&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&1&2&0&0&0&0&2&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&1&0&2&2&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&1&2&0&2&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&1&0&2&0&0&0&2&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&1&2&0&0&0&0&2&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&1&0&2&2&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&1&2&0&2&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&2&0&0&0&2&0&1&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&2&0&0&0&0&2&1&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&2&2&0&1&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&2&0&2&1&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&2&0&0&0&2&0&1\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&2&0&0&0&0&2&1\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&2&2&0&1&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&2&0&2&1&0
\end{pmatrix}.$$

The recovering sets are given by

$$\begin{aligned}
\mathcal{A}_{1,t} &= \{3t-2, 3t-1, 3t\}, && 1 \le t \le 12,\\
\mathcal{A}_{2,s,t} &= \{9s+t, 9s+t+3, 9s+t+6\}, && 1 \le t \le 3, \qquad 0 \le s \le 3.
\end{aligned}$$

These recovering sets can be checked by using the generator matrix above. For example, we can see that $\mathcal{A}_{1,1} = \{1, 2, 3\}$ is indeed a recovering set. The reason that $\mathcal{A}_{1,1}$ is a recovering set is because we can add the first, second and third column of the generator matrix together to obtain a column vector with only zeros. This shows that we will always have that $c_1+c_2+c_3 = 0$ for a codeword $c = (c_1, \ldots, c_n)$, and we can use this relation to recover a missing point if it is at the first, second or third position. But for these three positions we also have different recovering sets, given by $\mathcal{A}_{2,0,1}$, $\mathcal{A}_{2,0,2}$ and $\mathcal{A}_{2,0,3}$ respectively. Similarly one can check all the other recovering sets.

This is just a beautiful example that shows how the theory applies in practice. Within the generator matrix we can see blocks of length 9, which come from the fact that our total extension had degree 9. The 2 recovering sets come from the fact that we have a compositum of 2 Artin-Schreier extensions of degree 3, and they can be found within the generator matrix by just looking for them in the blocks. This really shows the power of this construction method.

# 5 Conclusion

We started by defining the locality and what a locally recoverable code is. We saw that there already exist some bounds for the locality, but we described a different one that uses the parity check matrix of a code. This upper bound often (and possibly always) gives the exact locality. However, its computation time increases extremely fast as the dimension of the parity check matrix increases.
Since the computation of the locality took a long time, we decided that it might be better to construct LRC's such that the locality is known beforehand. We described a method for this. After that we looked at some examples. The first one was an example over $\mathbb{F}_2$. For this example we first had to find an explicit realization of a curve that was described by Serre. We found an expression for this curve, which to our knowledge had never been done before. However, this curve turned out not to be useful for our method. The problem was that this curve was defined

over $\mathbb{F}_2$. This led to a proposition which tells us whether we will get the desired results or not. After that we looked at 2 more examples which satisfied the conditions of this proposition, and therefore they worked out well.
In the last part we extended our method to be able to construct LRC's with multiple recovering sets. We applied this method to an example over $\mathbb{F}_3$ to create a code with 2 recovering sets for each symbol. Within the generator matrix of the resulted code we could see the 2 recovering sets for each symbol.

## References


[BHH+16] Alexander Barg, Kathryn Haymaker, Everett W. Howe, Gretchen L. Matthews, and Anthony Várilly-Alvarado. *Locally Recoverable Codes from Algebraic Curves and Surfaces*, December 2016. `https://arxiv.org/pdf/1701.05212.pdf`.

[BM16] Edoardo Ballico and Chiara Marcolla. Higher hamming weights for locally recoverable codes on algebraic curves. *Finite Fields and Their Applications*, 40:61–72, 2016.

[BT17] Sourbh Bhadane and Andrew Thangaraj. *Irregular Recovery and Unequal Locality for Locally Recoverable Codes with Availability*, May 2017. `https://arxiv.org/pdf/1705.05005.pdf`.

[BTV16] Alexander Barg, Itzhak Tamo, and Serge Vlăduţ. *Locally Recoverable Codes on Algebraic Curves*, March 2016. `https://arxiv.org/pdf/1603.08876v1.pdf`.

[GHSY12] Parikshit Gopalan, Cheng Huang, Huseyin Simitci, and Sergey Yekhanin. On the locality of codeword symbols. *IEEE Transactions on Information Theory*, 58(1):6925–6934, November 2012. `https://pdfs.semanticscholar.org/ad2a/2a5e35a2b8036ba1fe7272689e886040dc18.pdf`.

[MMH16] Beth Malmskog, Gretchen Matthews, and Kathryn Haymaker. *Locally Recoverable Codes with Availability $t \geq 2$ from Fiber Products of Curves*, December 2016. `https://arxiv.org/pdf/1612.03841.pdf`.

[PD14] Dimitris S. Papailiopoulos and Alexandros G. Dimakis. Locally repairable codes. *IEEE Transactions on Information Theory*, 60(10):5843–5855, October 2014.

[RPDV16] Ankit Singh Rawat, Dimitris S. Papailiopoulos, Alexandros G. Dimakis, and Sriram Vishwanath. Locality and availability in distributed storage. *IEEE Transactions on Information Theory*, 62(8):4481–4493, Augustus 2016.

[Ser83] Jean-Pierre Serre. Sur le nombre des points rationnels d'une courbe algébrique sur un corps fini. *Comptes Rendus des Séances de l'Académie des Sciences. Série I. Mathématique*, 296(9):397–402, March 1983. `https://web.stanford.edu/class/cme324/classics/cooley-tukey.pdf`.

[Sti09] Henning Stichtenoth. *Algebraic Function Fields and Codes*. Springer, 2009.

[TBF16] Itzhak Tamo, Alexander Barg, and Alexey Frolov. Bounds on the parameters of locally recoverable codes. *IEEE Transactions on Information Theory*, 62(6):3070–3083, June 2016.

[TVN07] Michael Tsfasman, Serge Vlăduţ, and Dmitry Nogin. *Algebraic Geometric Codes: Basic Notions*, volume 139. Mathematical Surveys and Monographs, 2007.

[VDGVDV00] Gerard Van Der Geer and Marcel Van Der Vlugt. Tables of curves with many points. *Mathematics of Computations*, 69(230):797–810, April 2000. `http://www.jstor.org/stable/2584903`.

[WZ14] Anyu Wang and Zhifang Zhang. Repair locality with multiple erasure tolerance. *IEEE Transactions on Information Theory*, 60(11):6979–6987, November 2014. `https://pdfs.semanticscholar.org/ad2a/2a5e35a2b8036ba1fe7272689e886040dc18.pdf`.

# A Magma implementation for finding an upper bound for the locality of a code

```
P<x> := PolynomialRing(FiniteField(3));
n := 50;
F := Factorization(x^n - 1);
F;
C := CyclicCode(n, (F[2][1])* F[3][1]);
D := Dual(C);
D;
k := Dimension(D);
r := k;
DD := CartesianPower(D, k);
W := [];
A := [];

t := Cputime();
for x in DD do
    for j := 1 to k do
       A[j] := x[j];
    end for;

    if IsIndependent(A) then
        for i := 1 to k do
            W[i] := Weight(x[i]);
        end for;
        m := Maximum(W)-1;
        if m le r then
            r := m;
        end if;
    end if;
end for;
r;
print Cputime(t);
```

To find an upper bound for the locality of any code, one can change $C$ and replace it with the preferred code. The $P$, $n$ and $F$ are only used to construct this specific code $C$, thus these can be omitted if another code $C$ is used.

Note that this algorithm is far from optimal. The algorithm loops over all ordered sets of $k$ vectors in $D$, rather than only unordered sets. Besides that it does not take advantage of the fact that vectors $v$ and $\lambda v$ lead to the same answer for $\lambda \neq 0$. Hence several improvements are possible that will reduce the computation time.

# B Magma implementation for creating code with designated locality with Hermitian curves

```
q := 31;
l := 3;
Pq<t>:=PolynomialRing(GF(q));
rand := RandomIrreduciblePolynomial(GF(q), 2);
"The irreducible polynomial that we took is:", rand;
Fq2<a>:=ext<GF(q) | rand>;
Fy<y>:=FunctionField(Fq2);
PF<s>:=PolynomialRing(Fy);
K<x>:=FunctionField(s^q+s-y^(q+1));

T := Cputime();
finpts:=[];
for P in Places(K,1) do
    if Type(Evaluate(y,P)) ne Infty then
        Append(~finpts, P);
    end if;
end for;

r := q-1;
n := #finpts;
f := [];
for index1 in [0..r-1] do
    for index2 in [0..l] do
        Append(~f, x^index1*y^index2);
    end for;
end for;

lengthf := #f;
v := [];
for i in [1..lengthf] do
    vec:=[];
    for j in [1..n] do
        ev := Evaluate(f[i],finpts[j]);
        Append(~vec, ev);
    end for;
    Append(~v, vec);
end for;

C:=LinearCode<Fq2, n| v>;
"The created code has length", Length(C), "and dimension", Dimension(C);
"A lower boundary for d is", n - l*q-(q-2)*(q+1);
"The locality is equal to", q-1;
print Cputime(T);
```

To find different codes one can change the values of $l$ and $q$. The consequences of this are described in Example 3.2.

# C Finding the extension with genus 50 and 40 rational points

We will find an explicit equation for a covering of degree 8 of an elliptic curve $E$ over $\mathbb{F}_2$, such that this covering has 40 rational points over $\mathbb{F}_2$ and genus 40. This example has been proposed by Serre in [Ser83]. Serre proves that if we take an elliptic curve $E$ of genus 1 with 5 rational points together with a point $P_n$ of degree $n \geq 5$, then there is a Galois extension $C$ of $E$ such that its degree is equal to $2^{n-4}$ and it ramifies only over $P_n$. Over the rational points it will split completely. This curve will have genus $1 + n(2^{n-4} - 1)$ and the number of points will be equal to $5 \cdot 2^{n-4}$. Hence, for $n = 7$, we obtain a curve with genus 50 and 40 rational points. Furthermore, over $\mathbb{F}_2$ there exists at least one elliptic curve with 5 rational points, and thus this gives us the ingredients to create the function with the desired properties as found in the tables of Van Der Geer and Van Der Vlugt.

Now we will describe the desired equation explicitly, as we need it to create our code. First of all we need an elliptic cuve with 5 rational points over $\mathbb{F}_2$. As said before, there exists at least one elliptic curve with that property. For example, the elliptic curve $E : b^2 + b = a^3 + a$ satisfies this condition. Now we need to find a function of degree 8 such that we can lift this curve to a curve of genus 50 with 40 rational points. This will be done in 3 steps. In each step we make a quadratic extension of the previous curve. We will make these extensions such that the rational points will split, and the points $P_j$, which are defined as the $P_n$ by Serre, will not split. This way we will obtain $5 \cdot 2^3 = 40$ rational points. In order to do so we need the points $P_j$, which are of degree 7. Hence we need to find points that live in $E(\mathbb{F}_{2^7})$, but are not elements of $E(\mathbb{F}_2)$. Such points can be found by `MAGMA`. For example, we can take the $a$-coordinates to satisfy the equation $a^7 + a + 1 = 0$. The corresponding $b$-coordinates satisfy $b = a^6 + a^5 + a^2 + a$.
Now we have our 5 rational points $Q_i$ and our 7 points $P_j$. We want to use these to create an extension of $E$. More specifically, we are using these to create an Artin-Schreier extension of $E$. This extension will be defined by the points $\alpha$ that satisfy the quadratic equation $\alpha^2 + \alpha + F(P) = 0$ for $P \in E$. Since we are working in $\mathbb{F}_2$, we see that this equation has two solutions if $F(P) = 0$, and thus these points will split. But if $F$ has a pole at $P$, then no splitting will occur at these points.
Since we want the rational points to split, and that the points $P$ will not, we are searching a function $F$ that has zeros at the points $Q_i$, and poles at the points $P_j$. To find such a function we will look at the divisor

$$D = \sum_{j=1}^{7} P_j - \sum_{i=1}^{5} Q_i. \tag{C.1}$$

Then the functions in the Riemann-Roch space will have the required properties. Since the degree of our divisor is equal to 2, the Riemann-Roch theorem tells us that this Riemann-Roch space has dimension 2, and thus there do exist such functions. These can for example be found by `MAGMA`.

By choosing one of the functions in the Riemann-Roch space we can create an extension $E^1$ of $E$ such that it has 10 rational points. This trick has to be repeated twice to obtain a curve with 40 rational points. But while repeating this trick we have to take one thing into consideration. That is, if we define the divisor the same way, then in the second step we use 10 rational points $Q_i^1$, and 7 points $P_j^1$. But then the divisor will result in a Riemann-Roch space of dimension zero. This can be resolved by using the same divisor, except that we take the points $P_j^1$ for example twice. Thus if we have our 10 rational points $Q_i^1$ and our 7 points $P_j^1$, then we take

the divisor

$$D^1 = 2\sum_{j=1}^{7} P_j^1 - \sum_{i=1}^{10} Q_i^1.$$

The corresponding Riemann-Roch space has dimension 2 again, and thus we can take a function out of it to create a second extension. We do have to be careful here, because it might be possible that the polynomial $x^2 + x + F \in \mathbb{F}(E^1)[x]$ is not irreducible. If that is the case, then we cannot create our second extension. This restricts our choices, but it is possible to do so.

Now we have to repeat this step a third and final time. To ensure that the Riemann-Roch space exists, we have to take our points that result from the points $P_j$ at least 4 times. But it turns out that this is not even enough, because then none of the resulting polynomials over $\mathbb{F}_2(E^2)$ is irreducible. But with the right choices of the previous functions for the extensions, it turns out that if we take the points $P_j$ 6 times, then we find functions such that the third and final extension is possible. The functions $F_h$ used for the $h^{\text{th}}$ extension are given by

$$\begin{aligned}
F_1 &= \frac{a^5 + a^4 + a^3 + a}{a^7 + a + 1} b + \frac{a^6 + a^4}{a^7 + a + 1}, \\
F_2 &= \frac{(b+1)((a^4 + a^2 + a)b + a^5 + a^4)}{a(a^7 + a + 1)}, \\
F_3 &= \left( \frac{a^5 + a^4 + a^3 + a}{a^7 + a + 1} b + \frac{a^6 + a^4}{a^7 + a + 1} \right) d + \left( \frac{a^4 + a^3}{a^7 + a + 1} b + \frac{a^6 + a^4 + a^2 + a}{a^7 + a + 1} \right) c,
\end{aligned}$$

where the variables correspond to the extensions as shown below

$$\begin{array}{lcccc}
 & E & \longleftarrow & E^1 & \longleftarrow & E^2 & \longleftarrow & E^3 \\
\text{Variables:} & (a,b) & \xrightarrow{c^2+c=F_1} & (a,b,c) & \xrightarrow{d^2+d=F_2} & (a,b,c,d) & \xrightarrow{e^2+e=F_3} & (a,b,c,d,e) \\
\text{Genus:} & 1 & & 8 & & 22 & & 50 \\
\text{Number of rational points:} & 5 & & 10 & & 20 & & 40
\end{array}$$

**Remark C.1.** The genera that we found for the different extensions were calculated by using `MAGMA`. However, these could also be calculated by hand. To do this, one could for example use a formula given by Stichtenoth in [Sti09]. In our notation this formula is given by

$$2g(\widetilde{C}) - 2 = q(2g(C) - 2) + (q-1)(m+1)n. \tag{C.2}$$

Here $g(\widetilde{C})$ is the genus of the extension of the curve $C$ with genus $g(C)$. The number of poles of $F$ is denoted by $n$ and $m$ equals the absolute value of the valuation of $F$ in these poles. Since we construct our $F$ in a specific way, it follows that for our cases we will always have that $m = 1$. So, for example, if we look at the extension from $E$ to $E^1$, we have 7 poles, which come from the $P_j$'s. With $q = 2$ and $g(C) = 1$ we find that $2g(\widetilde{C}) - 2 = 14$. This tells us that $g(\widetilde{C})$ is indeed equal to 8, as was found by `MAGMA`. We note that for all the extensions the values of $q$, $m$ and $n$ remain the same, only the values of $g(C)$ and $g(\widetilde{C})$ change. Using this the genera of $E^2$ and $E^3$ can be computed.

Now that the equations are known, we can also define the extension from $E$ to $E^3$ by just one equation. This is

$$S^8 + ((a^{11}+a^{10}+a^6+a^3)/(a^{14}+a^2+1)b + (a^{14}+a^{13}+a^{11}+a^{10}+a^9+a^8+a^7+a^5 + a^4+a^3+1)/(a^{14}+a^2+1))S^4 + ((a^{18}+a^{17}+a^{16}+a^{15}+a^{14}+a^{13}+a^{10}+a^9+a^8 + a^6+a^5+a^3)/(a^{21}+a^{15}+a^{14}+a^9+a^7+a^3+a^2+a+1)b + (a^{20}+a^{19}+a^{18}+a^{16} + a^{14}+a^{10}+a^8+a^5+a^4+a^2)/(a^{21}+a^{15}+a^{14}+a^9+a^7+a^3+a^2+a+1))S^2 + ((a^{16} + a^{15}+a^{14}+a^{12}+a^{10}+a^9+a^8+a^7+a^5+a^4)/(a^{21}+a^{15}+a^{14}+a^9+a^7+a^3+a^2+a + 1)b + (a^{19}+a^{17}+a^{15}+a^{14}+a^{13}+a^{11}+a^9+a^8+a^7+a^6+a^5+a^4)/(a^{21}+a^{15}+a^{14} + a^9+a^7+a^3+a^2+a+1))S + (a^{32}+a^{30}+a^{28}+a^{24}+a^{20}+a^{18}+a^{16}+a^{14}+a^{10}+a^8)/ (a^{42}+a^{30}+a^{28}+a^{18}+a^{14}+a^6+a^4+a^2+1)b + (a^{38}+a^{35}+a^{34}+a^{30}+a^{29}+a^{28}+a^{27} + a^{26}+a^{25}+a^{23}+a^{22}+a^{18}+a^{16}+a^{15}+a^{14}+a^{13}+a^{12}+a^{10}+a^9+a^8)/(a^{42}+a^{30}+a^{28} + a^{18}+a^{14}+a^6+a^4+a^2+1) = 0.$$

Here we extend $E$ to $\widetilde{E} \cong E^3$ by adding the variable $S$ that satisfies the above equation.

The functions and extensions were all found by using `MAGMA`. The used implementation can be found in Appendix D.
To get some more information about our extension, we can look at its Galois group. Since we have a Galois extension of degree 8, we know that it has 8 elements. After looking at the order of these elements, it turns out that the Galois group is equal to $(\mathbb{Z}/2\mathbb{Z})^3$. This means that the functions $F_i$ could all have been found directly from the start, and thus we only needed to compute 1 Riemann-Roch space. But if we used our first Riemann-Roch space $D$ from Equation (C.1), then we did not find enough functions. Therefore we have to increase its dimension. For example, we can increase its dimension to 9 by taking the divisor

$$D = 2\sum_{j=1}^{7} P_j - \sum_{i=1}^{5} Q_i.$$

Now we have to look for a combination of 3 functions inside this Riemann-Roch space such that we can define our extensions as before. It turns out that the following three function do the trick.

$$\begin{aligned}
F_1 &= \frac{a^8+a^6+a^4+a^3+a^2+a}{a^{14}+a^2+1}b + \frac{a^7+a^6}{a^{14}+a^2+1},\\
F_2 &= \frac{a^6+a^5}{a^{14}+a^2+1}b + \frac{a^{10}+a^6+a^2+a}{a^{14}+a^2+1},\\
F_3 &= \frac{a^9+a^8}{a^{14}+a^2+1}b + \frac{a^{13}+a^9+a^5+a^4}{a^{14}+a^2+1}.
\end{aligned}$$

Then again, we can combine everything into one function $F$, that describes the extension immediately. This $F$ is given by

$$F = X^8 + ((a^{18} + a^{16} + a^{15} + a^{13} + a^{12} + a^{10})/(a^{28} + a^4 + 1)b + (a^{28} + a^{26} + a^{23} + a^{21} + a^{20} + a^{17} + a^{14} + a^{12} + a^{11} + a^{10} + a^8 + a^7 + a^5 + a^2 + 1)/(a^{28} + a^4 + 1))X^4 + ((a^{31} + a^{30} + a^{26} + a^{25} + a^{24} + a^{23} + a^{22} + a^{20} + a^{17} + a^{16} + a^{14} + a^{13} + a^{12} + a^{11})/(a^{42} + a^{30} + a^{28} + a^{18} + a^{14} + a^6 + a^4 + a^2 + 1)b + (a^{40} + a^{37} + a^{36} + a^{35} + a^{34} + a^{33} + a^{32} + a^{29} + a^{28} + a^{26} + a^{21} + a^{18} + a^{17} + a^{16} + a^{14} + a^{12} + a^{11} + a^{10} + a^9 + a^7 + a^5 + a^2)/(a^{42} + a^{30} + a^{28} + a^{18} + a^{14} + a^6 + a^4 + a^2 + 1))X^2 + ((a^{32} + a^{31} + a^{29} + a^{27} + a^{25} + a^{23} + a^{22} + a^{12} + a^{11} + a^{10})/(a^{42} + a^{30} + a^{28} + a^{18} + a^{14} + a^6 + a^4 + a^2 + 1)b + (a^{36} + a^{33} + a^{32} + a^{31} + a^{29} + a^{28} + a^{26} + a^{24} + a^{21} + a^{20} + a^{16} + a^{14} + a^{13} + a^{12} + a^{10} + a^8 + a^7 + a^6)/(a^{42} + a^{30} + a^{28} + a^{18} + a^{14} + a^6 + a^4 + a^2 + 1))X + (a^{100} + a^{94} + a^{90} + a^{88} + a^{85} + a^{83} + a^{82} + a^{81} + a^{79} + a^{77} + a^{76} + a^{75} + a^{74} + a^{73} + a^{68} + a^{65} + a^{64} + a^{61} + a^{59} + a^{57} + a^{56} + a^{54} + a^{53} + a^{48} + a^{47} + a^{43} + a^{42} + a^{38} + a^{36} + a^{33} + a^{32} + a^{30} + a^{26} + a^{25} + a^{24} + a^{22} + a^{21} + a^{19} + a^{16} + a^{15} + a^9 + a^8)/(a^{112} + a^{16} + 1)b + (a^{99} + a^{97} + a^{96} + a^{94} + a^{93} + a^{91} + a^{90} + a^{89} + a^{87} + a^{86} + a^{84} + a^{79} + a^{78} + a^{77} + a^{76} + a^{74} + a^{73} + a^{69} + a^{68} + a^{67} + a^{62} + a^{61} + a^{59} + a^{57} + a^{53} + a^{51} + a^{47} + a^{46} + a^{45} + a^{43} + a^{42} + a^{41} + a^{38} + a^{35} + a^{34} + a^{27} + a^{22} + a^{21} + a^{20} + a^{19} + a^{16} + a^{15} + a^{14} + a^{13} + a^{12} + a^{11} + a^{10} + a^9 + a^8 + a^6)/(a^{112} + a^{16} + 1) = 0. \tag{C.3}$$

This is the equation for the extension that we were looking for, and we will use this to construct our code over $\mathbb{F}_2$ of length 40, as described in Section 3.1.

# D Magma implementation for finding the extension with genus 50 and 40 rational points

```
F:=GF(2);
K0<a>:=FunctionField(F);
PK0<T>:=PolynomialRing(K0);
K<b>:=ext<K0 | T^2+T+(a^3+a)>;
DrK:=&+Places(K,1);
D:=Places(K,7)[1]-DrK;
fK,gK:=RiemannRochSpace(D);
hK:=gK(fK.1);


PK<S>:=PolynomialRing(K);
L<c>:=ext<K | S^2+S+hK>;
DrL:=&+Places(L,1);
DpL:=Poles(L!hK)[1];
fL,gL := RiemannRochSpace(2*DpL-DrL);
hL:=gL(fL.2);
PL<U>:=PolynomialRing(L);


"The function for the extension from E to E1 is given by";
hK;
" ";
"The function for the extension from E1 to E2 is given by";
hL;
" ";

M<d>:=ext<L | U^2+U+hL>;
DrM:=&+Places(M,1);
DpM:=Poles(M!hL)[1];
fM,gM := RiemannRochSpace(6*DpM-DrM);
di:=Dimension(fM);
PM<V>:=PolynomialRing(M);

for f in fM do
  fu:=gM(f);
  if IsIrreducible( V^2+V+fu) then
    N<e>:=ext<M | V^2+V+fu>;
     G := Genus(N);
     if (G eq 50)
      then fu;
    end if;
   end if;
end for;

N<e>:=ext<M | V^2+V+fu>;
```

# E Magma implementation for creating a code of length 40 over $\mathbb{F}_2$

```
F:=GF(2);
K0<a>:=FunctionField(F);
PK0<T>:=PolynomialRing(K0);
K<b>:=ext<K0 | T^2+T+(a^3+a)>;
PK<S>:=PolynomialRing(K);

pol:= T^16 + (a^18 + a^16 + a^15 + a^13 + a^12 + a^10)/(a^28 + a^4 + 1)*T^12 + (a^31
        + a^30 + a^26 + a^25 + a^24 + a^23 + a^22 + a^20 + a^17 + a^16 + a^14 + a^13
        + a^12 + a^11)/(a^42 + a^30 + a^28 + a^18 + a^14 + a^6 + a^4 + a^2 + 1)*T^10
        + (a^32 + a^31 + a^29 + a^27 + a^25 + a^23 + a^22 + a^12 + a^11 + a^10)/(a^42
        + a^30 + a^28 + a^18 + a^14 + a^6 + a^4 + a^2 + 1)*T^9 + (a^112 + a^108
        + a^100 + a^99 + a^97 + a^94 + a^90 + a^85 + a^84 + a^83 + a^82 + a^81 + a^80
        + a^79 + a^77 + a^76 + a^74 + a^71 + a^69 + a^68 + a^65 + a^61 + a^59 + a^57
        + a^56 + a^54 + a^53 + a^52 + a^51 + a^49 + a^48 + a^47 + a^42 + a^41 + a^40
        + a^38 + a^33 + a^32 + a^30 + a^28 + a^27 + a^26 + a^23 + a^22 + a^17 + a^16
        + a^13 + a^12 + a^9 + a^4 + 1)/(a^112 + a^16 + 1)*T^8 + (a^31 + a^29 + a^26
        + a^25 + a^24 + a^23 + a^17 + a^16 + a^14 + a^11)/(a^42 + a^30 + a^28 + a^18
        + a^14 + a^6 + a^4 + a^2 + 1)*T^6 + (a^32 + a^31 + a^30 + a^27 + a^25 + a^23
        + a^20 + a^13 + a^11 + a^10)/(a^42 + a^30 + a^28 + a^18 + a^14 + a^6 + a^4
        + a^2 + 1)*T^5 + (a^108 + a^102 + a^99 + a^98 + a^95 + a^94 + a^93 + a^90
        + a^89 + a^84 + a^82 + a^81 + a^78 + a^77 + a^74 + a^71 + a^70 + a^69 + a^68
        + a^65 + a^64 + a^59 + a^56 + a^52 + a^49 + a^48 + a^47 + a^43 + a^42 + a^41
        + a^40 + a^39 + a^38 + a^35 + a^34 + a^31 + a^29 + a^28 + a^27 + a^26 + a^24
        + a^23 + a^22 + a^21 + a^20 + a^16 + a^14 + a^13 + a^12 + a^11 + a^9 + a^4)/
        (a^112 + a^16 + 1)*T^4 + (a^30 + a^29 + a^22 + a^20 + a^13 + a^12)/(a^42
        + a^30 + a^28 + a^18 + a^14 + a^6 + a^4 + a^2 + 1)*T^3 + (a^100 + a^98 + a^96
        + a^95 + a^93 + a^92 + a^91 + a^89 + a^88 + a^87 + a^86 + a^82 + a^81 + a^79
        + a^77 + a^75 + a^73 + a^72 + a^71 + a^70 + a^66 + a^64 + a^63 + a^61 + a^60
        + a^58 + a^54 + a^53 + a^52 + a^49 + a^48 + a^47 + a^46 + a^45 + a^42 + a^40
        + a^39 + a^38 + a^37 + a^32 + a^29 + a^25 + a^22 + a^16 + a^11 + a^10)/(a^112
        + a^16 + 1)*T^2 + (a^94 + a^91 + a^90 + a^81 + a^77 + a^76 + a^74 + a^72 + a^71
        + a^63 + a^60 + a^58 + a^57 + a^54 + a^52 + a^51 + a^47 + a^46 + a^45 + a^44
        + a^43 + a^41 + a^40 + a^39 + a^38 + a^36 + a^34 + a^33 + a^30 + a^29 + a^21
        + a^20 + a^19 + a^14)/(a^112 + a^16 + 1)*T + (a^91 + a^89 + a^87 + a^86 + a^85
        + a^84 + a^80 + a^79 + a^77 + a^74 + a^72 + a^71 + a^68 + a^67 + a^64 + a^62
        + a^61 + a^60 + a^55 + a^53 + a^52 + a^49 + a^47 + a^45 + a^43 + a^42 + a^41
        + a^40 + a^38 + a^37 + a^32 + a^31 + a^27 + a^26 + a^24 + a^23 + a^20 + a^18
        + a^17 + a^15 + a^14 + a^12)/(a^112 + a^16 + 1);

N<x>:=FunctionField(pol);
b := (a^154*x^8+a^154*x^4+a^152*x^4+a^152*x^2+a^149*x^4+a^149*x^2+a^147*x^4+a^148*x^2
        +a^146*x^4+a^142*x^8+a^148*x+a^147*x^2+a^146*x^2+a^140*x^8+a^145*x^2+a^143*x^4
        +a^145*x+a^144*x^2+a^142*x^4+a^144*x+a^143*x+a^140*x^4+a^141*x^2+a^141*x
        +a^140*x^2+a^141+a^140*x+a^138*x^2+a^136*x^4+a^139+a^138*x+a^138+a^130*x^8
        +a^136*x+a^136+a^132*x^4+a^135+a^133*x^2+a^133*x+a^126*x^8+a^133+a^132*x
        +a^129*x^4+a^132+a^130*x^2+a^131+a^129*x^2+a^128*x^2+a^126*x^4+a^128*x
```

```
        +a^125*x^4+a^128+a^126*x^2+a^126*x+a^123*x^4+a^125*x+a^124*x^2+a^118*x^8+a^125
        +a^124*x+a^123*x^2+a^121*x^4+a^122*x^2+a^120*x^4+a^116*x^8+a^123+a^122*x
        +a^121*x^2+a^122+a^114*x^8+a^121+a^120*x+a^119*x^2+a^117*x^4+a^119*x+a^116*x^4
        +a^112*x^8+a^119+a^118*x+a^117*x^2+a^117+a^114*x^2+a^112*x^4+a^115+a^114+a^113
        +a^111+a^109+a^108+a^104+a^103+a^102+a^101+a^100+a^99+a^96+a^94+a^92+a^90+a^89
        +a^86+a^84+a^83+a^82+a^81+a^80+a^79+a^78+a^77+a^76+a^74+a^69+a^68+a^67+a^58*x^8
        +a^65+a^62+a^58*x^4+a^56*x^4+a^59+a^58+a^56*x^2+a^53*x^4+a^56+a^55+a^53*x^2
        +a^51*x^4+a^54+a^52*x^2+a^50*x^4+a^46*x^8+a^53+a^52*x+a^51*x^2+a^52+a^50*x^2
        +a^44*x^8+a^51+a^49*x^2+a^47*x^4+a^49*x+a^48*x^2+a^46*x^4+a^42*x^8+a^48*x+a^48
        +a^47*x+a^44*x^4+a^47+a^45*x^2+a^46+a^45*x+a^44*x^2+a^42*x^4+a^45+a^44*x
        +a^42*x^2+a^42*x+a^40*x^2+a^34*x^8+a^41+a^40*x+a^37*x^4+a^40+a^36*x^4+a^35*x^4
        +a^38+a^37*x+a^36*x^2+a^34*x^4+a^37+a^35*x^2+a^33*x^4+a^28*x^8+a^31*x^4+a^33*x
        +a^33+a^29*x^4+a^32+a^31*x+a^30*x^2+a^28*x^4+a^30*x+a^29*x^2+a^27*x^4+a^22*x^8
        +a^29+a^27*x^2+a^25*x^4+a^28+a^20*x^8+a^25*x^2+a^25+a^23*x^2+a^21*x^4+a^23*x
        +a^16*x^8+a^23+a^22*x+a^21*x+a^14*x^8+a^21+a^20*x+a^17*x^4+a^16*x^4+a^19
        +a^17*x^2+a^18+a^16*x^2+a^14*x^4+a^17+a^16*x+a^13*x^4+a^14*x^2+a^14*x+a^11*x^4
        +a^13*x+a^12*x^2+a^6*x^8+a^13+a^12*x+a^11*x^2+a^9*x^4+a^10*x^2+a^8*x^4+a^4*x^8
        +a^10*x+a^9*x^2+a^10+a^2*x^8+a^9+a^8*x+a^7*x^2+a^5*x^4+a^7*x+a^4*x^4+x^8+a^6*x
        +a^5*x^2+a^6+a^2*x^2+x^4)/((a^136*x^4+a^134*x^4+a^136*x+a^135*x^2+a^133*x^4
        +a^135*x+a^134*x^2+a^131*x^4+a^134+a^133*x+a^130*x^4+a^131*x+a^130*x^2
        +a^128*x^4+a^129*x^2+a^129*x+a^128*x^2+a^127*x^2+a^128+a^127*x+a^126*x^2
        +a^124*x^4+a^126*x+a^124*x^2+a^121*x^4+a^124+a^120*x^4+a^121*x^2+a^120*x^2
        +a^118*x^4+a^117*x^4+a^120+a^118*x^2+a^119+a^117*x^2+a^116*x^2+a^114*x^4+a^117
        +a^116*x+a^115*x^2+a^115*x+a^115+a^114*x+a^114+a^113+a^112+a^111+a^109+a^106
        +a^99+a^94+a^93+a^92+a^91+a^90+a^88+a^86+a^85+a^80+a^78+a^74+a^73+a^72+a^70
        +a^69+a^68+a^65+a^61+a^60+a^58+a^56+a^55+a^54+a^53+a^52+a^51+a^48+a^46+a^45
        +a^40*x^4+a^42+a^38*x^4+a^40*x+a^39*x^2+a^37*x^4+a^40+a^39*x+a^38*x^2+a^35*x^4
        +a^38+a^37*x+a^34*x^4+a^36+a^35*x+a^34*x^2+a^32*x^4+a^35+a^33*x^2+a^33*x
        +a^32*x^2+a^31*x^2+a^31*x+a^30*x^2+a^28*x^4+a^30*x+a^30+a^28*x^2+a^29+a^25*x^4
        +a^28+a^25*x^2+a^26+a^24*x^2+a^25+a^24*x+a^23*x^2+a^24+a^23*x+a^23+a^21*x^2
        +a^19*x^4+a^21*x+a^20*x^2+a^20*x+a^19*x^2+a^20+a^18*x^2+a^16*x^4+a^19+a^18*x
        +a^17*x^2+a^17*x+a^16*x^2+a^17+a^15*x^2+a^15*x+a^14*x^2+a^12*x^4+a^15+a^14*x
        +a^14+a^12*x^2+a^13+a^9*x^4+a^12+a^8*x^4+a^9*x^2+a^10+a^8*x^2+a^6*x^4+a^9+
        a^5*x^4+a^8+a^6*x^2+a^5*x^2+a^6+a^4*x^2+a^2*x^4+a^5+a^4*x+a^3*x^2+a^4+a^3*x
        +a^3+a^2*x+a^2+a+1)*a^8);

Pl := Places(N,1);
D:=Places(K,7)[1]-Places(K,5)[1];
fK,gK:=RiemannRochSpace(D);
gK(Basis(fK));

g := [(a^5 + a^4 + a^2 + 1)/(a^7 + a + 1)*b + (a^6 + a^4 + 1)/(a^7 + a + 1),
    (a^2 + a + 1)/(a^7 + a + 1)*b + (a^7 + a^5 + a^4)/(a^7 + a + 1)];

r := 7;
n := #Pl;
f := [];
for index1 in [0..r-1] do
    for index2 in [1..#g] do
```

```
        Append(~f, x^index1*g[index2]);
    end for;
end for;

lengthf := r*#g;
v := [];
for i in [1..lengthf] do
    vec:=[];
    for j in [1..n] do
        ev := Evaluate(f[i],Pl[j]);
        Append(~vec, ev);
    end for;
    Append(~v, vec);
end for;

C:=LinearCode<F, n| v>;
"The created code has length", Length(C), "and dimension", Dimension(C);
"The minimum distance is equal to", MinimumDistance(C);
"The locality is equal to", r;
C;
```

The functions for *pol* and $b$ were found by combining the equation from $E : b^2 + b = a^3 + a$ and the function $F$ given by Equation (C.3). Then *pol* is the equation used to create our extension with 40 rational points directly from $\mathbb{F}_2$. By doing this we can save a lot of computation time.

Note that the entries of $g$ are the same as given by the line `gK(Basis(fK));`. The reason we do not write `g := gK(Basis(fK));` is that now the $b$ is replaced by a polynomial in $a$ and $x$. Therefore it is seen as a polynomial in $N$. Otherwise it is seen as a polynomial in $K$, since the $b$ will not be replaced, and $g$ will be a polynomial in terms of $a$ and $b$. This would give errors when we want to create $f$, and therefore we have to write it down ourselves after it has been calculated by `MAGMA`.

With this `MAGMA` implementation one can construct different codes. This is done by changing the divisor $D$. This will give a different result for `gK(Basis(fK));`. Then the $g$ has to be set equal to this result. When this is done, one will get a code of length 40 over $\mathbb{F}_2$. The values of $k$ and $d$ depend on your choice of the divisor $D$.

# F Magma implementation for creating a code of length 39 over $\mathbb{F}_3$

```
F3:=GF(3);
K0<a>:=FunctionField(F3);
PK0<B>:=PolynomialRing(K0);
K1<b>:=ext<K0 | (1-a)*B^5+a^3*B^3-B+a^4-a^2>;
PK1<C>:=PolynomialRing(K1);

pol := a^44*(a^146+2*a^143+a^140+2*a^139+2*a^137+2*a^136+2*a^135+2*a^134+a^133+a^131+a
    ^130+a^128+a^126+a^125+a^123+2*a^122+2*a^121+2*a^120+a^118+2*a^117+2*a^114+2*a
    ^113+2*a^112+2*a^109+a^108+2*a^104+2*a^102+2*a^101+a^100+a^99+2*a^97+a^96+2*a^94+a
    ^93+2*a^92+2*a^91+2*a^90+a^89+a^88+a^87+a^85+a^83+2*a^82+a^81+2*a^79+a^76+2*a^75+a
    ^74+a^73+2*a^71+a^70+a^69+2*a^68+2*a^67+2*a^66+a^64+2*a^62+2*a^61+a^60+a^59+a
    ^58+2*a^57+a^56+a^55+a^54+a^51+2*a^50+a^49+a^48+2*a^47+2*a^46+2*a^43+a^42+2*a
    ^40+2*a^39+a^38+a^37+a^36+a^32+2*a^28+2*a^27+a^26+2*a^24+a^23+a^21+2*a^19+a^18+a
    ^16+2*a^15+2*a^14+a^13+2*a^12+2*a^11+a^10+a^9+a^8+2*a^6+a^5+a^3+2*a^2+2)*B^15+a
    ^44*(a^146+2*a^143+a^140+2*a^139+2*a^137+2*a^136+2*a^135+2*a^134+a^133+a^131+a
    ^130+a^128+a^126+a^125+a^123+2*a^122+2*a^121+2*a^120+a^118+2*a^117+2*a^114+2*a
    ^113+2*a^112+2*a^109+a^108+2*a^104+2*a^102+2*a^101+a^100+a^99+2*a^97+a^96+2*a^94+a
    ^93+2*a^92+2*a^91+2*a^90+a^89+a^88+a^87+a^85+a^83+2*a^82+a^81+2*a^79+a^76+2*a^75+a
    ^74+a^73+2*a^71+a^70+a^69+2*a^68+2*a^67+2*a^66+a^64+2*a^62+2*a^61+a^60+a^59+a
    ^58+2*a^57+a^56+a^55+a^54+a^51+2*a^50+a^49+a^48+2*a^47+2*a^46+2*a^43+a^42+2*a
    ^40+2*a^39+a^38+a^37+a^36+a^32+2*a^28+2*a^27+a^26+2*a^24+a^23+a^21+2*a^19+a^18+a
    ^16+2*a^15+2*a^14+a^13+2*a^12+2*a^11+a^10+a^9+a^8+2*a^6+a^5+a^3+2*a^2+2)*B^13+a
    ^35*(a^154+a^152+2*a^151+2*a^149+2*a^147+2*a^145+2*a^144+2*a^143+a^141+a^140+a
    ^139+2*a^138+a^136+2*a^133+a^131+2*a^128+a^127+2*a^126+2*a^124+2*a^123+a^122+a
    ^120+a^118+2*a^116+a^113+a^111+2*a^108+2*a^106+a^101+a^97+a^95+2*a^93+2*a^92+2*a
    ^91+2*a^90+a^88+a^87+2*a^84+a^80+a^79+a^77+a^76+2*a^73+2*a^72+2*a^69+2*a^68+a
    ^67+2*a^66+2*a^64+a^63+2*a^62+a^61+2*a^60+a^59+2*a^57+a^56+2*a^55+2*a^54+a^53+a
    ^50+2*a^49+a^46+a^45+a^44+2*a^43+a^42+a^41+a^40+2*a^38+2*a^37+a^36+2*a^35+2*a
    ^34+2*a^33+2*a^32+a^31+a^27+a^26+a^25+a^19+a^18+a^17+2*a^16+a^15+2*a^14+a^13+2*a
    ^12+2*a^11+a^8+2*a^7+a^5+a^4+2*a^3+2*a^2+1)*B^12+a^44*(a^146+2*a^143+a^140+2*a
    ^139+2*a^137+2*a^136+2*a^135+2*a^134+a^133+a^131+a^130+a^128+a^126+a^125+a^123+2*a
    ^122+2*a^121+2*a^120+a^118+2*a^117+2*a^114+2*a^113+2*a^112+2*a^109+a^108+2*a
    ^104+2*a^102+2*a^101+a^100+a^99+2*a^97+a^96+2*a^94+a^93+2*a^92+2*a^91+2*a^90+a^89+
    a^88+a^87+a^85+a^83+2*a^82+a^81+2*a^79+a^76+2*a^75+a^74+a^73+2*a^71+a^70+a^69+2*a
    ^68+2*a^67+2*a^66+a^64+2*a^62+2*a^61+a^60+a^59+a^58+2*a^57+a^56+a^55+a^54+a^51+2*a
    ^50+a^49+a^48+2*a^47+2*a^46+2*a^43+a^42+2*a^40+2*a^39+a^38+a^37+a^36+a^32+2*a
    ^28+2*a^27+a^26+2*a^24+a^23+a^21+2*a^19+a^18+a^16+2*a^15+2*a^14+a^13+2*a^12+2*a
    ^11+a^10+a^9+a^8+2*a^6+a^5+a^3+2*a^2+2)*B^11+a^35*(2*a^154+2*a^152+a^151+a^149+a
    ^147+a^145+a^144+a^143+2*a^141+2*a^140+2*a^139+a^138+2*a^136+a^133+2*a^131+a
    ^128+2*a^127+a^126+a^124+a^123+2*a^122+2*a^120+2*a^118+a^116+2*a^113+2*a^111+a
    ^108+a^106+2*a^101+2*a^97+2*a^95+a^93+a^92+a^91+a^90+2*a^88+2*a^87+a^84+2*a^80+2*a
    ^79+2*a^77+2*a^76+a^73+a^72+a^69+a^68+2*a^67+a^66+a^64+2*a^63+a^62+2*a^61+a^60+2*a
    ^59+a^57+2*a^56+a^55+a^54+2*a^53+2*a^50+a^49+2*a^46+2*a^45+2*a^44+a^43+2*a^42+2*a
    ^41+2*a^40+a^38+a^37+2*a^36+a^35+a^34+a^33+a^32+2*a^31+2*a^27+2*a^26+2*a^25+2*a
    ^19+2*a^18+2*a^17+a^16+2*a^15+a^14+2*a^13+a^12+a^11+2*a^8+a^7+2*a^5+2*a^4+a^3+a
    ^2+2)*B^10+a^26*(2*a^164+2*a^161+a^158+a^155+2*a^154+2*a^153+a^151+2*a^150+2*a
    ^148+a^145+2*a^144+2*a^141+2*a^140+a^139+a^138+2*a^137+2*a^136+2*a^135+2*a^133+2*a
    ^131+a^130+a^120+a^119+2*a^118+2*a^117+2*a^116+a^114+2*a^112+a^109+2*a^108+2*a
    ^107+2*a^106+2*a^102+a^101+2*a^100+2*a^98+2*a^97+2*a^95+a^94+a^93+a^90+2*a^88+2*a
    ^85+a^84+2*a^83+a^82+a^81+2*a^79+2*a^76+2*a^75+a^73+2*a^71+a^69+2*a^67+a^66+2*a
    ^65+a^64+a^63+2*a^62+a^60+2*a^59+a^57+a^56+2*a^54+a^53+a^52+a^51+2*a^50+a^48+a
    ^47+2*a^45+a^43+a^37+2*a^36+2*a^34+2*a^33+2*a^30+2*a^29+a^28+a^27+a^26+2*a^25+2*a
```

```
^24+a^23+a^21+a^19+2*a^18+2*a^17+2*a^16+a^14+a^13+a^10+a^9+2*a^8+2*a^7+2*a^5+a^4+a
^3+a+1)*B^9+a^44*(2*a^146+a^143+2*a^140+a^139+a^137+a^136+a^135+a^134+2*a^133+2*a
^131+2*a^130+2*a^128+2*a^126+2*a^125+2*a^123+a^122+a^121+a^120+2*a^118+a^117+a
^114+a^113+a^112+a^109+2*a^108+a^104+a^102+a^101+2*a^100+2*a^99+a^97+2*a^96+a
^94+2*a^93+a^92+a^91+a^90+2*a^89+2*a^88+2*a^87+2*a^85+2*a^83+a^82+2*a^81+a^79+2*a
^76+a^75+2*a^74+2*a^73+a^71+2*a^70+2*a^69+a^68+a^67+a^66+2*a^64+a^62+a^61+2*a
^60+2*a^59+2*a^58+a^57+2*a^56+2*a^55+2*a^54+2*a^51+a^50+2*a^49+2*a^48+a^47+a^46+a
^43+2*a^42+a^40+a^39+2*a^38+2*a^37+2*a^36+2*a^32+a^28+a^27+2*a^26+a^24+2*a^23+2*a
^21+a^19+2*a^18+2*a^16+a^15+a^14+2*a^13+a^12+a^11+2*a^10+2*a^9+2*a^8+a^6+2*a^5+2*a
^3+a^2+1)*B^7+a^28*(2*a^161+2*a^160+a^157+a^154+a^153+2*a^152+a^150+a^147+a^146+2*
a^145+a^144+2*a^140+2*a^138+a^136+a^135+a^133+2*a^132+a^130+2*a^129+2*a^128+a
^127+2*a^126+a^125+2*a^124+a^123+a^122+a^121+a^120+2*a^118+a^116+2*a^115+2*a^114+a
^112+2*a^111+2*a^107+a^105+a^103+a^102+2*a^99+2*a^98+2*a^94+2*a^93+a^92+a^91+a
^90+2*a^87+2*a^86+2*a^84+2*a^82+a^80+a^79+a^78+2*a^76+2*a^75+2*a^74+2*a^73+a^72+2*
a^71+a^69+a^68+2*a^66+2*a^65+a^63+a^62+a^60+a^59+a^58+a^57+a^54+a^53+2*a^52+2*a
^51+a^49+a^47+a^45+a^42+2*a^41+2*a^40+2*a^38+2*a^37+2*a^36+2*a^35+2*a^34+a^32+a
^31+a^30+a^28+2*a^27+a^21+a^20+2*a^19+2*a^18+2*a^17+a^16+2*a^15+2*a^14+a^13+2*a
^10+a^9+2*a^8+2*a^7+a^6+a^3+2*a+2)*B^6+a^44*(2*a^146+a^143+2*a^140+a^139+a^137+a
^136+a^135+a^134+2*a^133+2*a^131+2*a^130+2*a^128+2*a^126+2*a^125+2*a^123+a^122+a
^121+a^120+2*a^118+a^117+a^114+a^113+a^112+a^109+2*a^108+a^104+a^102+a^101+2*a
^100+2*a^99+a^97+2*a^96+a^94+2*a^93+a^92+a^91+a^90+2*a^89+2*a^88+2*a^87+2*a^85+2*a
^83+a^82+2*a^81+a^79+2*a^76+a^75+2*a^74+2*a^73+a^71+2*a^70+2*a^69+a^68+a^67+a
^66+2*a^64+a^62+a^61+2*a^60+2*a^59+2*a^58+a^57+2*a^56+2*a^55+2*a^54+2*a^51+a^50+2*
a^49+2*a^48+a^47+a^46+a^43+2*a^42+a^40+a^39+2*a^38+2*a^37+2*a^36+2*a^32+a^28+a
^27+2*a^26+a^24+2*a^23+2*a^21+a^19+2*a^18+2*a^16+a^15+a^14+2*a^13+a^12+a^11+2*a
^10+2*a^9+2*a^8+a^6+2*a^5+2*a^3+a^2+1)*B^5+a^28*(a^2-a+1)*(a^159+3*a^158+4*a
^157+2*a^156-a^155-2*a^154-a^153+3*a^152+5*a^151+2*a^150-2*a^149-2*a^148+4*a
^146+4*a^145-4*a^143-3*a^142+3*a^141+6*a^140+3*a^139-3*a^138-6*a^137-2*a^136+4*a
^135+7*a^134+5*a^133-3*a^131-a^130+3*a^129+6*a^128+4*a^127-4*a^125-2*a^124+2*a
^123+6*a^122+6*a^121+a^120-4*a^119-5*a^118-a^117+5*a^116+6*a^115+2*a^114-4*a
^113-4*a^112+a^111+6*a^110+7*a^109+a^108-6*a^107-5*a^106+3*a^105+8*a^104+6*a
^103-5*a^101-5*a^100+6*a^98+6*a^97-5*a^95-5*a^94+2*a^93+8*a^92+8*a^91+a^90-5*a
^89-5*a^88+5*a^86+6*a^85+2*a^84-4*a^83-5*a^82+a^81+8*a^80+7*a^79+a^78-4*a^77-4*a
^76+4*a^74+4*a^73+a^72-3*a^71-3*a^70+5*a^68+7*a^67+2*a^66-4*a^65-5*a^64+a^63+7*a
^62+6*a^61+a^60-4*a^59-5*a^58+6*a^56+6*a^55+a^54-5*a^53-5*a^52+6*a^50+7*a^49+2*a
^48-5*a^47-5*a^46+5*a^44+7*a^43+3*a^42-2*a^41-3*a^40-a^39+2*a^38+4*a^37+3*a^36+a
^35+a^33+2*a^32+3*a^31+a^30-a^29-a^28+2*a^26+4*a^25+4*a^24+2*a^23-a^21+a^20+4*a
^19+3*a^18-a^17-4*a^16-a^15+4*a^14+6*a^13+2*a^12-3*a^11-3*a^10+2*a^9+5*a^8+5*a
^7+2*a^6-2*a^5-3*a^4-a^3+2*a^2+4*a+2)*B^4+a^26*(a+1)*(a^160-a^159+3*a^158-a^157+2*
a^156-2*a^155+2*a^154-a^153+2*a^152-a^151+3*a^150-3*a^149+5*a^148-5*a^147+5*a
^146-3*a^145+4*a^144-3*a^143+3*a^142-2*a^141+3*a^140-a^139+a^138+a^137+a^136-a
^135+3*a^134-3*a^133+3*a^132-3*a^131+3*a^130-2*a^129+2*a^128-a^127+2*a^126-a
^125+2*a^124-2*a^123+4*a^122-2*a^121+4*a^120-3*a^119+5*a^118-4*a^117+4*a^116-2*a
^115+4*a^114-2*a^113+4*a^112-3*a^111+5*a^110-4*a^109+6*a^108-4*a^107+6*a^106-6*a
^105+6*a^104-4*a^103+4*a^102-4*a^101+5*a^100-4*a^99+5*a^98-3*a^97+4*a^96-2*a^95+4*
a^94-2*a^93+3*a^92-a^91+2*a^90+2*a^88-a^87+3*a^86-3*a^85+4*a^84-3*a^83+3*a^82-a
^81+a^80-a^79+3*a^78-2*a^77+2*a^76+2*a^74+a^72-a^71+a^70-a^69+2*a^68+2*a^66-2*a
^65+2*a^64-a^63+3*a^62-2*a^61+3*a^60-2*a^59+4*a^58-2*a^57+3*a^56-a^55+a^54+a^53-a
^52+a^51+2*a^49-2*a^48+4*a^47-2*a^46+2*a^45+a^43+2*a^41-a^40+2*a^39-2*a^38+2*a^37+
a^35+a^34+a^33+a^30+a^29+a^28-a^27+2*a^26-2*a^25+3*a^24-a^23+a^22+a^21+a^20-a
^19+3*a^18-2*a^17+3*a^16-a^15+3*a^14-3*a^13+5*a^12-4*a^11+4*a^10-4*a^9+4*a^8-2*a
^7+4*a^6-2*a^5+2*a^4-a^3+2*a^2+2)*B^3+a^28*(2*a^160+a^159+2*a^158+a^157+2*a^156+a
^153+a^152+2*a^151+a^148+2*a^147+2*a^146+a^145+a^144+a^143+a^140+a^136+a^134+2*a
^132+2*a^131+2*a^128+2*a^127+2*a^126+2*a^125+2*a^124+a^122+a^121+2*a^120+a^116+a
^115+2*a^114+2*a^113+a^112+2*a^111+a^108+2*a^107+a^105+a^104+a^103+2*a^102+2*a
```

```
    ^100+a^99+a^98+2*a^97+a^95+2*a^93+a^92+a^90+a^83+2*a^82+a^78+a^76+a^75+a^73+a^72+a
    ^71+a^70+2*a^68+2*a^67+2*a^65+2*a^64+2*a^63+2*a^61+2*a^60+a^59+a^58+2*a^57+2*a^56+
    a^54+2*a^53+2*a^50+2*a^49+a^48+2*a^47+2*a^44+a^43+a^41+a^40+2*a^39+2*a^37+2*a
    ^36+2*a^35+a^33+2*a^32+a^31+a^30+a^28+2*a^27+a^26+a^25+a^24+2*a^23+a^22+2*a^20+a
    ^19+a^18+2*a^17+a^16+a^14+a^13+a^12+a^11+a^10+2*a^8+a^6+a^3+2*a+2)*B^2+a^28*(a
    ^159+a^157+2*a^156+a^152+a^151+2*a^149+a^148+a^143+2*a^142+2*a^141+2*a^140+2*a
    ^139+a^136+2*a^135+a^131+a^130+2*a^129+2*a^128+2*a^126+a^124+2*a^123+a^121+2*a
    ^120+a^119+2*a^118+a^117+2*a^116+2*a^114+2*a^113+2*a^112+a^111+a^110+a^107+a
    ^105+2*a^103+a^102+2*a^101+a^100+a^98+a^97+2*a^96+a^95+a^94+2*a^93+2*a^92+2*a^91+a
    ^89+2*a^87+2*a^86+a^84+a^83+2*a^82+a^81+2*a^80+2*a^79+a^78+a^76+2*a^75+a^74+2*a
    ^70+a^69+2*a^66+a^65+a^64+2*a^63+2*a^62+2*a^57+2*a^56+2*a^55+2*a^54+2*a^53+a^52+2*
    a^51+2*a^50+a^49+a^48+a^44+a^42+a^41+2*a^40+a^39+2*a^38+2*a^37+a^35+2*a^34+a^33+a
    ^32+a^31+a^30+a^29+a^26+a^24+a^21+a^20+2*a^19+a^18+a^16+2*a^14+a^13+2*a^12+2*a
    ^10+2*a^8+2*a^7+2*a^6+2*a^5+a^4+a^3+a^2+a+1)*B+a^30*(2*a^155+2*a^152+2*a^149+a
    ^148+a^144+2*a^143+2*a^141+2*a^140+2*a^138+2*a^134+2*a^131+2*a^128+a^127+a^125+a
    ^123+2*a^122+2*a^120+2*a^119+2*a^118+2*a^117+2*a^116+a^115+2*a^114+2*a^113+2*a
    ^111+a^107+2*a^106+2*a^105+a^103+2*a^102+2*a^101+a^100+2*a^99+a^98+a^97+a^96+2*a
    ^93+a^92+a^91+a^90+a^86+2*a^85+2*a^84+a^83+a^81+2*a^80+2*a^79+a^78+a^77+2*a^76+a
    ^73+a^71+2*a^68+2*a^65+a^64+2*a^63+a^60+a^59+2*a^58+2*a^57+2*a^56+a^55+a^54+a^53+a
    ^52+2*a^51+a^49+a^48+a^46+2*a^45+a^44+a^43+2*a^42+2*a^40+2*a^39+2*a^34+2*a^32+2*a
    ^31+2*a^29+2*a^28+a^25+a^24+2*a^22+2*a^19+2*a^18+a^16+2*a^15+a^11+2*a^10+2*a^8+a
    ^6+a^5+a^3+a^2+1);
N<x> := FunctionField(pol);

b := 2*a^2*(a^28*(a^128+a^127+2*a^126+a^125+2*a^124+2*a^123+2*a^122+a^120+a^118+2*a
    ^116+2*a^114+2*a^113+2*a^111+2*a^107+2*a^105+2*a^104+a^103+2*a^101+2*a^100+2*a^98+
    a^96+a^95+a^94+a^93+2*a^91+2*a^90+a^88+2*a^87+2*a^86+a^85+a^84+2*a^82+2*a^81+2*a
    ^79+2*a^76+2*a^74+2*a^73+2*a^72+2*a^71+a^70+a^67+a^66+2*a^65+a^62+2*a^61+2*a^60+2*
    a^59+a^57+2*a^56+a^54+a^53+a^52+a^48+2*a^46+a^40+a^39+a^38+a^37+a^36+2*a^35+a
    ^34+2*a^29+a^27+a^24+a^23+2*a^21+a^20+2*a^19+2*a^18+a^17+2*a^12+2*a^10+a^8+2*a
    ^7+2*a^4+2*a^3+2*a^2+a+2)*x^9+a^21*(2*a^134+2*a^132+2*a^131+2*a^129+a^128+2*a
    ^127+2*a^126+a^125+a^123+2*a^122+2*a^119+2*a^118+a^117+a^115+a^114+a^113+2*a^112+a
    ^111+a^110+2*a^109+2*a^108+a^107+a^104+a^103+a^102+a^100+a^98+a^97+a^96+a^95+2*a
    ^94+a^93+a^92+2*a^90+2*a^89+a^88+2*a^87+2*a^86+a^84+a^78+a^76+a^75+2*a^74+a^73+a
    ^72+a^71+2*a^68+a^66+2*a^62+2*a^60+a^57+a^56+2*a^54+2*a^53+a^52+2*a^50+2*a^49+a
    ^48+2*a^47+a^45+2*a^44+2*a^43+a^41+2*a^40+a^39+a^38+2*a^36+a^35+a^34+2*a^33+a
    ^31+2*a^30+2*a^28+a^26+2*a^25+a^23+2*a^22+a^21+a^20+2*a^19+2*a^18+a^17+a^15+2*a
    ^13+2*a^11+a^10+a^9+a^8+2*a^6+a^3+a^2+2*a+2)*x^6+a^21*(2*a^134+2*a^132+2*a^131+2*a
    ^129+a^128+2*a^127+2*a^126+a^125+a^123+2*a^122+2*a^119+2*a^118+a^117+a^115+a^114+a
    ^113+2*a^112+a^111+a^110+2*a^109+2*a^108+a^107+a^104+a^103+a^102+a^100+a^98+a^97+a
    ^96+a^95+2*a^94+a^93+a^92+2*a^90+2*a^89+a^88+2*a^87+2*a^86+a^84+a^78+a^76+a^75+2*a
    ^74+a^73+a^72+a^71+2*a^68+a^66+2*a^62+2*a^60+a^57+a^56+2*a^54+2*a^53+a^52+2*a
    ^50+2*a^49+a^48+2*a^47+a^45+2*a^44+2*a^43+a^41+2*a^40+a^39+a^38+2*a^36+a^35+a
    ^34+2*a^33+a^31+2*a^30+2*a^28+a^26+2*a^25+a^23+2*a^22+a^21+a^20+2*a^19+2*a^18+a
    ^17+a^15+2*a^13+2*a^11+a^10+a^9+a^8+2*a^6+a^3+a^2+2*a+2)*x^4+a^20*(2*a^136+2*a
    ^135+2*a^134+2*a^130+a^128+2*a^127+a^125+a^124+a^123+a^122+2*a^119+2*a^118+a
    ^117+2*a^114+a^112+2*a^111+2*a^110+a^109+2*a^108+a^105+a^104+2*a^103+a^102+a
    ^101+2*a^99+2*a^98+2*a^97+a^96+2*a^95+a^94+2*a^93+a^92+a^89+2*a^88+2*a^87+2*a
    ^85+2*a^83+a^82+a^78+a^77+2*a^76+a^75+2*a^74+2*a^71+a^70+a^69+a^68+a^67+2*a^65+2*a
    ^63+2*a^62+a^61+a^60+a^59+a^58+a^56+a^54+2*a^53+2*a^52+a^51+2*a^50+a^49+2*a^48+2*a
    ^45+a^44+2*a^42+a^41+2*a^40+2*a^39+a^38+2*a^37+2*a^36+a^34+2*a^33+a^31+a^30+2*a
    ^29+2*a^28+2*a^27+2*a^26+a^25+2*a^23+2*a^22+2*a^21+2*a^20+a^19+2*a^18+2*a^16+a^15+
    a^14+2*a^13+2*a^12+2*a^10+a^9+2*a^7+2*a^6+2*a^5+a^4+2*a^2+2)*x^3+a^21*(2*a^134+2*a
    ^132+2*a^131+2*a^129+a^128+2*a^127+2*a^126+a^125+a^123+2*a^122+2*a^119+2*a^118+a
    ^117+a^115+a^114+a^113+2*a^112+a^111+a^110+2*a^109+2*a^108+a^107+a^104+a^103+a
```

```
^102+a^100+a^98+a^97+a^96+a^95+2*a^94+a^93+a^92+2*a^90+2*a^89+a^88+2*a^87+2*a^86+a
^84+a^78+a^76+a^75+2*a^74+a^73+a^72+a^71+2*a^68+a^66+2*a^62+2*a^60+a^57+a^56+2*a
^54+2*a^53+a^52+2*a^50+2*a^49+a^48+2*a^47+a^45+2*a^44+2*a^43+a^41+2*a^40+a^39+a
^38+2*a^36+a^35+a^34+2*a^33+a^31+2*a^30+2*a^28+a^26+2*a^25+a^23+2*a^22+a^21+a
^20+2*a^19+2*a^18+a^17+a^15+2*a^13+2*a^11+a^10+a^9+a^8+2*a^6+a^3+a^2+2*a+2)*x^2+a
^20*(2*a^134+2*a^133+a^132+a^131+2*a^130+a^128+a^127+2*a^126+2*a^125+2*a^123+a
^121+2*a^119+a^118+2*a^117+a^115+a^114+a^113+a^110+2*a^108+a^106+2*a^105+a^104+a
^102+a^101+2*a^99+2*a^98+a^97+a^96+2*a^95+a^92+a^90+a^88+2*a^87+a^85+a^84+a^83+a
^81+a^80+a^79+a^78+2*a^77+a^76+a^75+a^73+a^71+a^70+a^64+a^63+a^61+a^60+2*a^59+2*a
^58+a^56+a^53+a^52+2*a^51+a^50+2*a^49+2*a^47+2*a^46+a^44+a^43+2*a^41+a^40+a^39+2*a
^38+2*a^37+a^36+2*a^35+2*a^34+a^33+2*a^32+a^31+2*a^30+2*a^29+2*a^27+2*a^26+a^25+a
^23+a^22+a^21+2*a^20+2*a^19+2*a^18+2*a^14+a^13+2*a^12+a^11+2*a^10+a^9+a^8+a^7+a^6+
a^5+2*a^4+a^2+1)*x+a^21*(2*a^132+2*a^131+a^130+a^128+2*a^127+2*a^125+a^124+2*a
^121+2*a^120+a^119+2*a^118+a^116+2*a^115+a^113+2*a^111+2*a^109+2*a^106+a^105+a
^101+a^100+a^99+2*a^96+a^95+2*a^93+a^92+2*a^91+2*a^90+2*a^89+2*a^85+a^83+a^81+a
^80+a^79+2*a^76+2*a^75+a^68+a^66+2*a^65+2*a^64+2*a^63+2*a^62+a^61+a^60+a^59+a
^58+2*a^57+2*a^53+a^52+a^50+a^49+2*a^47+2*a^44+2*a^42+2*a^40+a^39+a^36+a^35+2*a
^31+a^28+a^27+2*a^25+2*a^22+a^21+2*a^20+2*a^18+2*a^17+2*a^16+2*a^15+a^14+2*a^13+2*
a^11+a^9+2*a^6+a^5+a^4+2*a^3+2*a^2+1))/(a^30*(a^127+2*a^126+a^125+a^124+a^123+a
^122+a^121+2*a^119+a^118+a^117+2*a^115+a^113+2*a^112+2*a^110+2*a^107+2*a^106+2*a
^103+2*a^102+2*a^100+a^99+2*a^98+a^97+2*a^94+a^93+a^90+a^89+2*a^87+2*a^84+2*a
^82+2*a^81+2*a^80+2*a^79+a^78+a^71+a^70+2*a^69+a^68+2*a^67+a^66+2*a^65+a^64+a^63+a
^62+a^61+a^60+a^59+a^58+2*a^56+2*a^54+2*a^53+a^52+2*a^51+a^49+2*a^46+2*a^45+2*a
^44+2*a^41+2*a^39+a^38+a^37+2*a^36+2*a^35+a^34+2*a^33+2*a^32+2*a^31+2*a^30+a^29+a
^28+a^27+a^25+a^21+a^19+a^18+2*a^14+a^13+2*a^12+a^11+a^9+2*a^8+2*a^7+a^6+2*a^5+a
^4+2*a^3+a^2+1)*x^9+a^21*(a^136+2*a^135+a^134+2*a^133+a^132+a^131+a^128+a^126+a
^125+2*a^124+2*a^123+2*a^122+2*a^119+a^117+2*a^115+2*a^113+a^111+2*a^110+2*a
^109+2*a^108+2*a^107+2*a^106+a^105+a^104+a^103+a^102+2*a^101+2*a^100+a^99+a^97+a
^96+2*a^94+2*a^93+a^91+2*a^90+2*a^89+a^88+2*a^86+a^85+2*a^84+a^82+a^78+a^77+2*a
^75+2*a^74+2*a^73+a^72+a^71+a^70+a^69+a^68+2*a^67+a^66+a^65+a^60+a^59+a^56+2*a^53+
a^51+a^48+a^46+2*a^45+a^44+a^43+a^41+a^40+2*a^38+a^37+a^35+2*a^34+2*a^33+a^32+a
^31+a^30+a^26+2*a^25+2*a^24+2*a^22+2*a^21+a^20+a^19+a^18+a^16+2*a^15+2*a^14+a^13+a
^11+a^10+a^7+a^4+2*a^3+2*a+2)*x^6+a^21*(a^136+2*a^135+a^134+2*a^133+a^132+a^131+a
^128+a^126+a^125+2*a^124+2*a^123+2*a^122+2*a^119+a^117+2*a^115+2*a^113+a^111+2*a
^110+2*a^109+2*a^108+2*a^107+2*a^106+a^105+a^104+a^103+a^102+2*a^101+2*a^100+a^99+
a^97+a^96+2*a^94+2*a^93+a^91+2*a^90+2*a^89+a^88+2*a^86+a^85+2*a^84+a^82+a^78+a
^77+2*a^75+2*a^74+2*a^73+a^72+a^71+a^70+a^69+a^68+2*a^67+a^66+a^65+a^60+a^59+a
^56+2*a^53+a^51+a^48+a^46+2*a^45+a^44+a^43+a^41+a^40+2*a^38+a^37+a^35+2*a^34+2*a
^33+a^32+a^31+a^30+a^26+2*a^25+2*a^24+2*a^22+2*a^21+a^20+a^19+a^18+a^16+2*a^15+2*a
^14+a^13+a^11+a^10+a^7+a^4+2*a^3+2*a+2)*x^4+a^22*(2*a^135+2*a^134+2*a^133+2*a^132+
a^130+a^128+a^126+a^125+2*a^124+2*a^123+2*a^122+a^121+a^120+2*a^119+2*a^116+a^115+
a^110+a^108+a^107+2*a^105+2*a^104+2*a^103+a^102+a^101+2*a^100+2*a^99+a^98+a^97+a
^95+2*a^92+a^91+a^89+a^88+2*a^87+2*a^85+a^83+2*a^82+a^81+a^80+2*a^79+a^78+2*a
^77+2*a^75+a^74+2*a^71+2*a^70+2*a^69+2*a^68+a^67+2*a^66+a^65+a^64+2*a^62+2*a^59+a
^57+a^56+2*a^54+a^53+2*a^52+a^51+a^50+2*a^49+2*a^48+2*a^47+2*a^45+2*a^41+a^40+a
^39+2*a^38+2*a^37+2*a^36+a^35+a^34+a^33+a^31+a^30+2*a^28+a^27+a^26+a^23+a^22+a^21+
a^20+a^19+a^18+2*a^17+a^16+2*a^15+2*a^14+2*a^12+2*a^11+2*a^10+a^9+a^5+a^3+2*a+2)*x
^3+a^21*(a^136+2*a^135+a^134+2*a^133+a^132+a^131+a^128+a^126+a^125+2*a^124+2*a
^123+2*a^122+2*a^119+a^117+2*a^115+2*a^113+a^111+2*a^110+2*a^109+2*a^108+2*a
^107+2*a^106+a^105+a^104+a^103+a^102+2*a^101+2*a^100+a^99+a^97+a^96+2*a^94+2*a^93+
a^91+2*a^90+2*a^89+a^88+2*a^86+a^85+2*a^84+a^82+a^78+a^77+2*a^75+2*a^74+2*a^73+a
^72+a^71+a^70+a^69+a^68+2*a^67+a^66+a^65+a^60+a^59+a^56+2*a^53+a^51+a^48+a^46+2*a
^45+a^44+a^43+a^41+a^40+2*a^38+a^37+a^35+2*a^34+2*a^33+a^32+a^31+a^30+a^26+2*a
^25+2*a^24+2*a^22+2*a^21+a^20+a^19+a^18+a^16+2*a^15+2*a^14+a^13+a^11+a^10+a^7+a
^4+2*a^3+2*a+2)*x^2+a^22*(2*a^134+2*a^131+a^130+2*a^129+2*a^128+a^127+a^126+a^125+
```

```
    a^124+2*a^123+a^122+a^121+a^119+a^118+a^116+a^114+a^111+a^107+a^106+a^104+a^103+a
    ^101+a^100+a^99+a^98+a^97+2*a^92+2*a^91+a^90+2*a^87+2*a^86+a^85+2*a^83+a^82+2*a
    ^81+2*a^80+a^78+2*a^77+2*a^76+2*a^75+a^74+a^73+2*a^72+a^67+2*a^65+2*a^62+a^61+2*a
    ^60+2*a^59+a^57+2*a^56+2*a^54+2*a^52+2*a^51+2*a^50+2*a^49+a^48+2*a^47+2*a^46+a^44+
    a^43+2*a^42+2*a^41+2*a^38+a^35+2*a^34+a^33+2*a^31+2*a^30+2*a^29+a^28+a^27+a^26+2*a
    ^23+a^21+a^19+2*a^18+2*a^15+a^13+2*a^11+2*a^9+2*a^8+2*a^5+2*a^3+a+1)*x+a^23*(a
    ^129+a^127+2*a^125+2*a^124+a^123+2*a^122+2*a^121+2*a^120+a^119+2*a^118+2*a^117+2*a
    ^115+2*a^114+a^113+a^112+2*a^111+2*a^110+a^109+2*a^107+2*a^103+a^102+a^101+a
    ^100+2*a^99+a^97+2*a^96+2*a^95+2*a^94+2*a^93+a^90+a^87+2*a^86+a^85+2*a^84+2*a^83+a
    ^82+a^80+a^79+a^78+2*a^76+2*a^73+2*a^72+a^71+2*a^69+2*a^68+2*a^67+2*a^64+2*a^63+2*
    a^62+2*a^61+a^60+2*a^59+2*a^58+a^57+a^56+a^55+2*a^54+a^53+a^51+2*a^50+2*a^49+a
    ^47+2*a^46+a^45+2*a^44+a^43+a^42+2*a^41+a^40+a^39+2*a^38+2*a^37+2*a^36+2*a^35+2*a
    ^34+a^33+a^31+a^29+a^28+2*a^26+2*a^25+a^24+a^23+a^21+a^20+a^18+a^17+2*a^16+a^14+2*
    a^13+a^12+2*a^9+2*a^8+a^6+a^5+a^4+a^3+2*a+1));
Pl := Places(N,1);

D:=11*Places(K1,2)[1]-Places(K1,4)[2]-Places(K1,5)[1];
fK,gK:=RiemannRochSpace(D);
gK(Basis(fK));

g := [
    (a^12 + 2*a^11 + 2*a^5 + a^4 + 2*a^2 + 1)/(a^15 + 2*a^14 + a^11)*b^4 +
        (2*a^9 + a^7 + 2*a^5 + a^4 + 2*a^3 + 1)/(a^13 + 2*a^12 + a^9)*b^3 +
        (2*a^14 + a^12 + a^11 + 2*a^9 + a^8 + 2*a^7 + 2*a^6 + a^5 + 2*a^4 +
        2*a^3 + a^2 + a + 2)/(a^15 + 2*a^14 + a^11)*b^2 + (a^10 + 2*a^8 + a^6 +
        2*a^2 + 2*a + 1)/(a^11 + 2*a^10 + a^7)*b + (a^10 + a^8 + a^6 + 2*a^3 + a
        + 1)/(a^10 + a^9 + 2*a^8 + a^7),
    (a^9 + a^8 + 2*a^7 + a^6 + 2*a^5 + a^4 + 2*a^2 + a + 1)/(a^13 + a^12 +
        2*a^11 + a^10)*b^4 + (2*a^11 + a^10 + 2*a^6 + a^5 + a^3 + 2*a^2 + 2*a +
        1)/(a^14 + a^13 + 2*a^12 + a^11)*b^3 + (a^9 + 2*a^8 + a^7 + a^5 + a^3 +
        2*a + 2)/(a^13 + a^12 + 2*a^11 + a^10)*b^2 + (2*a^13 + a^11 + 2*a^10 +
        2*a^8 + 2*a^6 + 2*a^5 + 2*a^3 + 2*a + 2)/(a^14 + a^13 + 2*a^12 + a^11)*b
        + (a^12 + 2*a^10 + 2*a^9 + 2*a^8 + a^7 + a^6 + 2*a^4 + a^3 + a^2 + 2*a +
        2)/(a^12 + a^11 + 2*a^10 + a^9),
    (a^9 + a^8 + 2*a^7 + a^6 + 2*a^5 + a^4 + 2*a^3 + a^2 + 2*a + 2)/(a^14 + a^13
        + 2*a^12 + a^11)*b^4 + (2*a^10 + 2*a^8 + 2*a^7 + a^6 + 2*a^5 + a^4 + a^3
        + 1)/(a^13 + a^12 + 2*a^11 + a^10)*b^3 + (2*a^11 + a^10 + a^8 + a^6 +
        a^3 + a^2 + 1)/(a^14 + a^13 + 2*a^12 + a^11)*b^2 + (a^12 + 2*a^11 + a^10
        + 2*a^9 + a^7 + 2*a^5 + a^4 + 2*a^3 + 2)/(a^13 + a^12 + 2*a^11 + a^10)*b
        + (2*a^11 + a^10 + a^7 + 2*a^6 + 2*a^5 + a^4 + a^2 + 2*a + 2)/(a^11 +
        a^10 + 2*a^9 + a^8),
    (a^8 + 2*a^7 + 2*a + 1)/(a^14 + 2*a^13 + a^10)*b^4 + (2*a^12 + a^11 + 2*a^10
        + 2*a^9 + a^8 + 2*a^7 + 2*a^6 + a^3 + 2*a^2 + 2*a + 1)/(a^15 + 2*a^14 +
        a^11)*b^3 + (2*a^12 + 2*a^11 + a^9 + 2*a^8 + 2*a^7 + a^4 + 2*a^3 + a +
        2)/(a^14 + 2*a^13 + a^10)*b^2 + (a^13 + a^12 + 2*a^11 + 2*a^10 + 2*a^8 +
        2*a^7 + 2*a^6 + a^4 + 2*a^3 + 2*a + 2)/(a^15 + 2*a^14 + a^11)*b + (a^11
        + a^10 + 2*a^9 + a^8 + a^7 + a^4 + a^2 + 2)/(a^12 + a^11 + 2*a^10 +
        a^9),
    (a^8 + 2*a^7 + 2*a^2 + 1)/(a^15 + 2*a^14 + a^11)*b^4 + (a^10 + 2*a^8 + 2*a^6
        + a^3 + a^2 + 2)/(a^14 + 2*a^13 + a^10)*b^3 + (2*a^11 + a^9 + a^8 +
        2*a^7 + 2*a^6 + a^4 + a^2 + a + 2)/(a^15 + 2*a^14 + a^11)*b^2 + (a^13 +
        a^12 + 2*a^11 + 2*a^10 + 2*a^5 + a^3 + a^2 + 1)/(a^14 + 2*a^13 + a^10)*b
        + (a^11 + 2*a^8 + 2*a^7 + 2*a^6 + a^5 + a^4 + 2*a^3 + 2*a^2 + 1)/(a^11 +
        a^10 + 2*a^9 + a^8),
    (a^7 + 2*a^6 + 2*a^5 + a^4 + a + 2)/(a^15 + 2*a^14 + a^11)*b^4 + (a^12 +
```

```
        a^11 + a^10 + a^9 + 2*a^7 + 2*a^6 + 2*a^3 + 2*a^2 + a + 2)/(a^15 +
        2*a^14 + a^11)*b^3 + (a^13 + a^10 + a^9 + 2*a^8 + a^6 + a^5 + a^4 +
        2*a^3 + 2*a^2 + a + 1)/(a^15 + 2*a^14 + a^11)*b^2 + (a^14 + 2*a^13 +
        a^12 + 2*a^11 + a^6 + 2*a^5 + 2*a^3 + a^2 + a + 1)/(a^15 + 2*a^14 +
        a^11)*b + (a^11 + a^9 + 2*a^8 + 2*a^6 + a^5 + a^3 + 2*a^2 + 1)/(a^12 +
        a^11 + 2*a^10 + a^9),
    (a^6 + 2*a^5 + 2*a^2 + 2*a + 2)/(a^15 + 2*a^14 + a^11)*b^4 + (a^12 + 2*a^11
        + 2*a^9 + 2*a^7 + 2*a^6 + a^5 + 2*a^4 + a^2 + 2)/(a^15 + 2*a^14 +
        a^11)*b^3 + (2*a^12 + a^11 + 2*a^10 + a^9 + a^8 + a^6 + 2*a^5 + 2*a^3 +
        a^2 + 1)/(a^15 + 2*a^14 + a^11)*b^2 + (2*a^14 + 2*a^13 + 2*a^12 + 2*a^11
        + a^6 + a^5 + a^3 + a^2 + 2*a + 1)/(a^15 + 2*a^14 + a^11)*b + (a^11 +
        2*a^10 + a^9 + 2*a^7 + a^6 + a^4 + a^3 + a^2 + a + 1)/(a^12 + a^11 +
        2*a^10 + a^9),
    (a + 2)/(a^12 + 2*a^11 + a^8)*b^4 + (a^9 + a^7 + 2*a^6 + 2*a^4 + 2*a^2 + 2*a
        + 2)/(a^14 + 2*a^13 + a^10)*b^3 + (2*a^10 + 2*a^8 + 2*a^7 + 2*a^5 + a^4
        + a^2 + 2*a + 2)/(a^12 + 2*a^11 + a^8)*b^2 + (a^10 + 2*a^8 + 2*a^6 + a^4
        + 2*a^3 + 1)/(a^14 + 2*a^13 + a^10)*b + (2*a^11 + a^10 + a^9 + a^7 + a^6
        + 2*a^5 + 2*a^4 + 2*a^3 + 2*a + 1)/(a^11 + a^10 + 2*a^9 + a^8),
    (a^3 + a + 1)/(a^15 + 2*a^14 + a^11)*b^4 + (a^11 + 2*a^10 + a^7 + 2*a^6 +
        2*a^4 + a^2 + 2*a + 1)/(a^14 + 2*a^13 + a^10)*b^3 + (a^13 + 2*a^12 +
        2*a^11 + a^8 + a^7 + a^3 + a^2 + 2)/(a^15 + 2*a^14 + a^11)*b^2 + (a^12 +
        a^10 + a^9 + 2*a^8 + 2*a^6 + a^5 + 2*a^3 + a^2 + 2)/(a^14 + 2*a^13 +
        a^10)*b + (a^11 + a^10 + 2*a^9 + a^8 + a^6 + 2*a^3 + 2*a + 2)/(a^11 +
        a^10 + 2*a^9 + a^8)
];

r := 2;
n := #Pl;

f := [];
for index1 in [0..r-1] do
    for index2 in [1..#g] do
        Append(~f, x^index1*g[index2]);
    end for;
end for;

lengthf := r*#g;
v := [];
for i in [1..lengthf] do
    vec:=[];
    for j in [1..n] do
        ev := Evaluate(f[i],Pl[j]);
        Append(~vec, ev);
    end for;
    Append(~v, vec);
end for;

C:=LinearCode<F3, n| v>;
"The created code has length", Length(C), "and dimension", Dimension(C);
"The minimum distance is equal to", MinimumDistance(C);
"The locality is equal to", r;
```

The functions for *pol* and $b$ were found by combining the equation from $Y : (1-a)b^5 + a^3b^3 - b + a^4 - a^2 = 0$ and the function $F$ given by Equation (3.3). Then *pol* is the equation used to create our extension with 39 rational points directly from $\mathbb{F}_3$. By doing this we can save a lot of computation time.

Note that the entries of $g$ are the same as given by the line `gK(Basis(fK));`. The reason we do not write `g := gK(Basis(fK));` is that now the $b$ is replaced by a polynomial in $a$ and $x$. Therefore it is seen as a polynomial in $N$. Otherwise it is seen as a polynomial in $K1$, since the $b$ will not be replaced, and $g$ will be a polynomial in terms of $a$ and $b$. This would give errors when we want to create $f$, and therefore we have to write it down ourselves after it has been calculated by `MAGMA`.

With this `MAGMA` implementation one can construct different codes. This is done by changing the divisor $D$. This will give a different result for `gK(Basis(fK));`. Then the $g$ has to be set equal to this result. When this is done, one will get a code of length 39 over $\mathbb{F}_3$. The values of $k$ and $d$ depend on your choice of the divisor $D$.

# G Magma implementation for creating a code of length 50 over $\mathbb{F}_5$

```
F5:=GF(5);
K0<a>:=FunctionField(F5);
PK0<B>:=PolynomialRing(K0);
K1<b>:=ext<K0 | a^3+3*a-B^2>;
PK1<C>:=PolynomialRing(K1);

pol := (4*a^10+3*a^8+a^6+3*a^4+a^2+2)*B^10+(2*a^10+4*a^8+3*a^6+4*a^4+3*a^2+1)*B^6+
        (4*a^10+3*a^8+a^6+3*a^4+a^2+2)*B^2+a*(a^8+3*a^4+1);

N<x> := FunctionField(pol);

b := (a^6+2)*x^5/((a^2-2*a+2)*(a^2+2*a+2))+(4*a^6+3)*x/((a^2-2*a+2)*(a^2+2*a+2));

Pl := Places(N,1);
D:=5*Places(K1,2)[1]-Places(K1,4)[1];
fK,gK:=RiemannRochSpace(D);
Dimension(fK);
gK(Basis(fK));

g := [(a^2 + 3)^-3 * (b) * (a)^2 * (a^2 + 4*a + 2),
    (a^2 + 3)^-3 * (b) * (a) * (a^2 + 4*a + 2),
    (a^2 + 3)^-3 * (b) * (a^2 + 4*a + 2),
    (a^2 + 2) * (a^2 + 3)^-2 * (a^2 + 4*a + 2),
    (a) * (a^2 + 3)^-2 * (a^2 + 4*a + 2),
    (a^2 + 3)^-2 * (a^2 + 4*a + 2)
];

r := 4;
n := #Pl;

f := [];
for index1 in [0..r-1] do
    for index2 in [1..#g] do
        Append(~f, x^index1*g[index2]);
    end for;
end for;

lengthf := r*#g;
v := [];
for i in [1..lengthf] do
    vec:=[];
    for j in [1..n] do
        ev := Evaluate(f[i],Pl[j]);
        Append(~vec, ev);
    end for;
    Append(~v, vec);
```

```
end for;

C:=LinearCode<F5, n| v>;
"The created code has length", Length(C), "and dimension", Dimension(C);
"The minimum distance is equal to", MinimumDistance(C);
"The locality is equal to", r;
```

The functions for *pol* and $b$ were found by combining the equation from $E : b^2 = a^3 + 3a$ and the function $F$ given by Equation (3.5). Then *pol* is the equation used to create our extension with 50 rational points directly from $\mathbb{F}_5$. By doing this we can save a lot of computation time.

Note that the entries of $g$ are the same as given by the line `gK(Basis(fK));`. The reason we do not write `g := gK(Basis(fK));` is that now the $b$ is replaced by a polynomial in $a$ and $x$. Therefore it is seen as a polynomial in $N$. Otherwise it is seen as a polynomial in $K1$, since the $b$ will not be replaced, and $g$ will be a polynomial in terms of $a$ and $b$. This would give errors when we want to create $f$, and therefore we have to write it down ourselves after it has been calculated by `MAGMA`.

With this `MAGMA` implementation one can construct different codes. This is done by changing the divisor $D$. This will give a different result for `gK(Basis(fK));`. Then the $g$ has to be set equal to this result. When this is done, one will get a code of length 50 over $\mathbb{F}_5$. The values of $k$ and $d$ depend on your choice of the divisor $D$.

# H Magma implementation for creating a code of length 36 over $\mathbb{F}_3$ with 2 recovering sets

```
F3 := GF(3);
K<a> := FunctionField(F3);
PK<B> := PolynomialRing(K);
K1<b> := ext<K | B^3-B-(a^3-a)/(a^4+a+2)>;
PK1<C> := PolynomialRing(K1);
N1<c>,G1 := ext<K1 | C^3-C-(a^3-a)/(a^4+2*a^3+2)>;
r2 := 2;

K2<c> := ext<K | B^3-B-(a^3-a)/(a^4+2*a^3+2)>;
PK2<D> := PolynomialRing(K2);
N2<b>,G2 := ext<K2 | D^3-D-(a^3-a)/(a^4+a+2)>;
r1 := 2;

D := Places(K,3)[1];
Dec := Decomposition(N2,D);

f,g := RiemannRochSpace((&+Dec));
"Dim(f)  = ", Dimension(f);
F := g(Basis(f));

Pl := Places(N2,1);
n := #Pl;

f := [];
for index1 in [0..r1-1] do
    for index2 in [0..r2-1] do
        for index3 in [1..#F] do
            Append(~f,b^index1*c^index2*F[index3]);
        end for;
    end for;
end for;

lengthf := #f;
v := [];
for i in [1..lengthf] do
    vec:=[];
    for j in [1..n] do
        ev := Evaluate(f[i],Pl[j]);
        Append(~vec, ev);
    end for;
    Append(~v, vec);
end for;

C := LinearCode<F3, n| v>;
"The created code has length", Length(C), "and dimension", Dimension(C);
"The minimum distance is equal to", MinimumDistance(C);
"The localities are equal to", r1, "and", r2;
```

This implementation is used for construction the code of length 36 over $\mathbb{F}_3$ with 2 recovering sets. One can construct different codes using this implementation by changing the divisor $D$. This will give a different result for `gK(Basis(fK));`. Then the $g$ has to be set equal to this result. When this is done, one will get a code of length 50 over $\mathbb{F}_5$. The values of $k$ and $d$ depend on your choice of the divisor $D$.
The lay-out can also be used to construct codes from different curves. In order to do so, one only has to change the defined fields, extensions and localities to the preferred ones. Here again, one can get different results by changing the divisor $D$.