1 Task 1: Design and Analysis Part I

For both channels in task 1, we choose to use a Reed-Muller code, but with varying parameters to push the information rate up, while respecting the bit-error restrictions. While the Reed-Muller codes have the disadvantage of always (except in trivial $\mathcal{RM}(m, m)$ cases) having even minimum distance (which is a disadvantidge because we do not really care about error-detection in this project, only correction), we still achieve decent information rates in both cases, while taking advantage of Reed-Muller's relatively fast decoding algorithm (Although, as we will see, still slow enough to cause us some trouble in estimating error-rate).

1.1 Command & Control:

For Command and Control, we use the $[32, 6, 16]$-code $\mathcal{RM}(1, 5)$. To create our Reed-Muller code, we first construct the set $M(r, m)$ of all monomials in m variables, of degree less than or equal to r, which in this case is just $M(1, 5) = \{1, x_1, x_2, x_3, x_4, x_5\}$. Next, we must order all elements in $\mathbb{F}_2^m$, which we simply do as all binary strings of length $m = 5$. Finally, for each $f_i \in M(r, m)$, we need to find a subset of the corresponding set $\mathcal{S}_{f_i}$ of size 2^{m-r} (this will be used in decoding). The set $\mathcal{S}_{f_i}$ consists of all polynomials of the form $\prod_{x_i \nmid f_i}(x_i + b)$, where $b \in \mathbb{F}_2$. Finally we need the map $\phi : \mathcal{V} \to \mathbb{F}_2^{2^m}$, where $\phi(f)$ is simply obtained by evaluating $f(\alpha)$ for all $\alpha \in \mathbb{F}_2^m$.

Encoding a message x is done by obtaining a polynomial f_x, where x is seen as the coefficients in from of some set order of elements in $M(r, m)$. Finally, the output of the encoder is $\phi(f_x)$. **Decoding** is then simply done by the majority-voting algorithm from lecture 6. Note that we will never obtain the polynomial z, but only work with $\phi(z)$. This is of course no problem by the linearity of ϕ (meaning instead of calculating $\phi(sz_j)$ in each loop, we instead calculate $\phi(s)\phi(z_j)$).

The information rate of $\mathcal{RM}(1, 5)$ is $\frac{6}{32} = 0.1875$. To compute the bit-error rate, we must first calculate the probability that a block decodes wrong, which corresponds to there being more than 8 errors in a 32 bit block (recall, $\mathcal{RM}(1, 5)$ is a $[32, 6, 16]$-code). This is given by the binomial distribution, where we wish to find $P(X \geq 8)$. We calculate it as

$$\sum_{i=8}^{32} \binom{32}{i} p^i (1-p)^{32-i} \approx 3.69 \cdot 10^{-12}$$

for $p = 0.005$. The number of bit errors pr. block that is decrypted is obviously upper-bounded by the length of the block, so we see that we are well within our requirement for command and control.

If we instead estimate the number of errors within a block to be about half of the block (note that we are using non-systematic encoding), we get the estimated number of bit errors to be $\frac{1}{2} \cdot 3.69 \cdot 10^{-12} = 1.845 \cdot 10^{-12}$.

Note that we are still assuming that 8 errors corresponds to a decoding error, which is not necessarily the case, since if we get 8 errors, we just choose arbitrarily between the tied codewords (which might result in correct decoding after all).

1.2 Data transfer

For data-transfer, we use the $[64, 42, 8]$-code $\mathcal{RM}(3, 6)$. Since we are still implementing it as Reed-Muller codes, algorithms for encoding and decoding are exactly the same as in section 1.1, except of course this time, $r, m = 3, 6$, so all the sets $M(r, m), \mathcal{S}_{f_i}, \mathbb{F}_2^m$ are different.

In this case, the information rate is $\frac{42}{64} = 0.65625$. The minimum distance is 8, so 4 errors is enough to (potentially) cause a decoding error, while the block size is 64. This time however, to also statistically show that we likely are within the requirements of less than 10^{-4} errors, we need to take into account that if we get 4 errors, we still estimate that we will decode to the correct block half the time. This time, we find the estimated number of blocks which decodes wrong as

$$\frac{1}{2}\binom{64}{4}p^4(1-p)^{64-4} + \sum_{i=5}^{64}\binom{64}{i}p^i(1-p)^{64-i} \approx 1.656 \cdot 10^{-4}$$

Again, we estimate that half of the bits in a wrongly decoded block turns into errors. Then we get that the estimated number of errors is $8.28 \cdot 10^{-5}$, which is (barely) within the requirement.
We summarize task 1 in table 1. Note that we have included bit-errors estimates for $p = 0.01$ as well. This is for reasons which will become apparent in the next section. The calculation is done exactly as for $p = 0.005$, only replacing p.

	Code	Information Rate	Est. Bit-Errors ($p = 0.005$)	Est. Bit-Errors ($p = 0.01$)
C&C	$\mathcal{RM}(1, 5)$	0.1875	$1.845 \cdot 10^{-12}$	$8.49 \cdot 10^{-10}$
Data	$\mathcal{RM}(3, 6)$	0.65625	$3.9125 \cdot 10^{-5}$	$1.10 \cdot 10^{-3}$

Table 1: Summary of task 1

2 Task 2: Implementation Part I

We provide an implementation attached at the end of this document. Note that instead of precomputing all sets $\mathcal{S}_{f_j}$ for $f_j \in M(r, m)$, we instead precompute all the sets $\mathcal{S}'_{f_j} =$

$\{\phi(f) \mid f \in \mathcal{S}_{f_j}\}$, since we only need the image under ϕ anyway. Other than that, the implementation follows what we described in section 1.1

2.1 Command and Control:

As can be seen in the notebook, the implementation encodes a block in $\approx$ 36 ms, and decodes a block in $\approx$ 200 ms. As the $\mathcal{RM}(1,5)$ information blocks are of length 6, we obtain the approximate run time of encode at $\frac{36}{6} = 6$ ms pr. information bit, and decode at $\frac{200}{6} = 33.3$ ms pr. information bit.

Further, the precomputation of all the sets $\mathcal{S}'_{f_j}$ (or at least 2^{m-r} elements of them) takes up some memory which we should calculate. In this case, $|M(r,m)| = 6$, $2^{m-r} = 16$, and each element is of size 32 bits, so in total it uses about $6 \cdot 16 \cdot 32 = 3072$ bits of memory as well (in addition to some other small stuff).

2.2 Data transfer

Again, we do the same as in the previous section. Encoding a block takes $\approx$ 80 ms, and decodes a block takes $\approx$ 3100 ms. This time however, the information blocks are of length 42, so we obtain the approximate run time of encode at $\frac{80}{42} \approx 1.9$ ms pr. information bit, and decode at $\frac{3100}{42} \approx 73.8$ ms pr. information bit.

In this case, the required precomputation is larger. Same calculation as in the previous section ($k = 42$, $2^{r-m} = 8$ and elements are 64 bits) gives $42 \cdot 8 \cdot 64 = 21504$ bits.

A summary of task 2 is provided in table 2

	Encoding pr. bit	Decoding pr. bit	Precomputed memory use
C&C	6 ms	33.33 ms	3 Kbit
Data	1.9 ms	73.8 ms	21 Kbit

Table 2: Summary of task 2

Now, we can note that as both the encoding and decoding times are way to big to actually measure if $\approx 10^{-12}$ is a decent estimate (just decoding 10^{12} bits with our $\mathcal{RM}(3,6)$ implementation would take over 2000 years), we felt that we needed to include the $p = 0.01$ case as well.

3 Task 3: Testing Part I

Implementation of the binary symmetric channel can also be found in the code.

In all cases we sample 30.000 random bits, encode them, send them through the binary-symmetric channel, decode them and finally compare with the original message. The results

are given in table 3.

	$p = (0.005)$		$p = (0.01)$	
	Est. error-rate	Measured error-rate	Est. error-rate	Measured error-rate
C&C	$1.845 \cdot 10^{-12}$	0	$8.49 \cdot 10^{-10}$	0
Data	$3.9125 \cdot 10^{-5}$	0	$1.10 \cdot 10^{-3}$	$1.69 \cdot 10^{-3}$

Table 3: Summary of task 3. Each message is 30.000 bits

For our command and control channel, we struggle to obtain decent estimates for the bit-error rate, but at least we get the expected results. In both cases ($p = 0.005$ and $p = 0.01$), the estimated error-rate was so low that we did not expect to see any errors for a message of 30.000 bits. So while the results were as expected, the slow run-time of our algorithm is unfortunately hindering us from measuring the actual error-rate accurately.

For data, we see that for $p = 0.005$, we could have maybe expected 1 or 2 errors. However, actually getting 1 or 2 errors is not really what we could expect, because once a decoding error happens, we estimate that half the block will be wrong, corresponding to around 21 errors. So not seeing any errors was the excepted result in this case as well. However, finally, we note that for $p = 0.01$, the error rate is fairly close to what we expected.

4 Task 4: Design and Analysis Part II

For the second part of this project, we spice things up a bit. For the data-transfer channel, we choose to use a $[127, 92, 11]$ primitive narrow-sense BCH code, while for the command and control channel, we attempt the common construction of concatenating Reed-Solomon and Convolutional Codes. The benefit of this construction is that Reed-Solomon codes are generally good at handling burst errors (because multiple errors might distribute over the same finite field element), while when convolutional codes get a decoding error, they will typically distribute the error in bursts. We hope that combining these two codes gives the desired super-reliable Command and Control channel.

4.1 Data Transfer

We design a primitive narrow-sense BCH code. The first part will be to find a fitting generating set[1]. We wrote a script which given q and n found all cyclotomic cosets of q in $\mathbb{Z}_n$. We ran our script with $q = 2$ and $n = q^8 - 1 = 127$ and find a fitting generating set

[1]In reality, We started by selecting a length/minimum distance ratio to get close to the 10^{-4} error correcting capabilities, and from there looked for my generating set...

which gives us a $\delta = 10$, or a minimum distance of 11 as

$$\mathcal{S} = \bigcup_{i=1}^{9} \mathcal{C}_i$$

From there we fix some primitive element $\alpha \in \mathbb{F}_{128}$, and calculate $g(X) = \prod_{i \in \mathcal{S}} (X - \alpha^i)$. From here, the design simply follows usual BCH codes. We note that $k = n - \deg g(X) = n - |\mathcal{S}| = 92$, so this is as mentioned a $[127, 92, 11]$ Code.

Encoding is given as usual by interperating the message block x of size $k = 92$ as a the coefficients of a polynomial, and multiplying this polynomial by g. For **decoding**, we use the Berlekamp-Massey algorithm to find the error locator polynomial Λ, which checks if position i is an error by evaluating $\Lambda(\alpha^{-i \bmod 127})$. From here, correcting errors is straight forward, as the coefficients are all in $\mathbb{F}_2$, so the only possible error is 1.

As a linear code, analysing the performance is straight forward. It has rate $\frac{k}{n} = \frac{92}{127} = 0.7422$, which is the best we have achieved so far. The block-error rate (with $p = 0.005$) is again given by the binomial distribution, as

$$\sum_{i=6}^{127} \binom{127}{i} p^i (1-p)^{127-i} \approx 4.82 \cdot 10^{-5}$$

which we (since we still use non-systematic encoding) estimate that turn into about half of the block being wrong. This gives an estimated bit error rate of $\approx 2.41 \cdot 10^{-5}$. Redoing the calculation with $p = 0.01$ gives us an estimated error rate of $9.23 \cdot 10^{-4}$.

4.2 Command and Control

For Command and Control, we combine a $[15, 7, 9]$-Reed-Solomon code over $\mathbb{F}_{16}$ with a convolutional encoder. Note that since the Reed-Solomon code is over $\mathbb{F}_{16}$, the code has bit-length $15 \cdot 4 = 60$. The Reed-Solomon code is designed to be similar to the one given in the notebook "l17-berlekamp-massey.jpynb", while the convolutional code is similar to the one discussed in lecture 20 and 21.

Encoding is done by first applying the usual Reed-Solomon/BCH encoding, but this time, the bits first have to be encoded as elements in $\mathbb{F}_{16}$, before they are regarded as coefficients of a polynomial. Next, output of the Reed-Solomon encoder is fed into the convolutional encoder which follow the shift-register logic (usually, these things are done in hard-ware, but that seemed a bit hardcore for this semester-project). We will also append two 0's at the end of the message, to make sure that it returns to the zero-state. Finally, the output of the convolutional encoder is the output. **Decoding** is of course done in the opposite order. The convolutional decoder implements the Viterbi-algorithm, by first creating the trelli-diagram with all nodes marked with their cost (this is done in one pass), and then

the algorithm backtracks to find the shortest path through the network. The output of the viterbi-algorithm is then input to the Reed-Solomon decoder, which works similarly to the BCH-decoder we described earlier, except this time we also have to recover the actual errors. This is simply done by solving a linear system of equations as described in the lectures.

The rate of this construction is the rate of both parts multiplied together, which is $\frac{7}{15} \cdot \frac{1}{2} = \frac{7}{30} = 0.2333$. The error-rate analysis however is much harder. We mentioned already that the convolutional code we use is the same as the one we went through in the lectures, namely the one generated by

$$G = \begin{pmatrix} 1 + X + X^2 & 1 + X^2 \end{pmatrix}$$

By looking at the corresponding trelli-diagram, we find that the free-distance is 5, corresponding to the path given by encoding $1, 0, 0$. From this, I do not know how to find the error probability. But, we can estimate that errors from the convolutional decoder typically happens than in bursts of length 3. From this, we estimate that on average, our Reed-Solomon decoder will be able to handle around less than 2 errors in the convolutional encoder, since errors typically distribute over 2 finite field elements, and the Reed-Solomon decoder handles up to 4 errors.

Note that the following calculation is entirely unscientific, and just meant to give a large ballpark estimate. We estimate the error rate of the convolutional encoder to be around the same as the $\mathcal{RM}(3, 6)$-code (which has slightly higher information rate than the convolutional encoder). Then for 15 *symbols* (consisting of 4 bits), we tolerate 2 errors. Using our ridiculus convolutional encoder estimate[2], we obtain the the total error-rate estimate in the usual way (replacing p with the r, the convolutional error-rate, and by regarding Finite Field symbols instead of bits)

$$\sum_{i=3}^{15} \binom{127}{i} r^i (1-r)^{15-i} \approx 2.72 \cdot 10^{-11}$$

when $r = 3.9125 \cdot 10^{-5}$. Redoing the calculation with $r = 1.1 \cdot 10^{-3}$ (which is the error rate of $\mathcal{RM}(3, 6)$ for $p = 0.01$), we get the estimate $\approx 6.00 \cdot 10^{-7}$. Again, we stress that these estimates should be taken with a grain of salt, as they are ridiculus. A summary is found in table 4

	Code	Information Rate	Est. Bit-E. ($p = 0.005$)	Est. Bit-E. ($p = 0.01$)
Data	$[127, 92, 11]$-BCH Code	0.7422	$2.41 \cdot 10^{-5}$	$9.23 \cdot 10^{-4}$
C&C	Concatenated Code	0.2333	$2.72 \cdot 10^{-11}$	$6.00 \cdot 10^{-7}$

Table 4: Summary of task 4

[2]Never mind the fact that even if we had the correct error-rate for the convolutional encoder, this would be inaccurate...

5 Task 5: Implementation Part II

Again, for the specifics of the implementation, see the end of the document.

5.1 Data transfer

We borrow the Berlekamp-Massey implementation given in "l17-berlekamp-massey.jpynb", and apart from some converting between bits and algebraic objects, the implementation is straight forward. As can be seen in the notebook, encoding a block of 92 bits takes about 2 ms, while decoding a block takes about 15 ms. This gives the approximate run time of encode at $\frac{2}{92} = 22$ μs pr. bit, while decode runs in $\frac{15}{92} = 163$ μs pr. information bit. Much better than what we saw in section 2!

5.2 Command and Control

The implementation is quite long, because translating between a stream of bits and elements of finite fields turned out to be quite messy. However, appart from that, the implementation of Reed-Solomon is very similar to our BCH code, but with the notable exception of solving a linear system of equations to obtain the actual errors. To do this, we use sage's built in Matrix.solve_right() function.

The Convolutional encode is implemented by using a logical shift-register. This is probably not practically a very bright idea, however, we did it to demonstrate the logic. The viterbi-algorithm is hard-coded for our convolutional code, which makes it suprisingly simple. Finally, we simply concatinate the codes.

The run time is found as usual. To our suprise, it is around the same/slightly faster than our BCH-implementation despite using two algorithms, and probably inefficient converting between bits and finite field elements. This is likely because we used a much larger block-size for BCH. A block of 7 finite field elements (or 28 bits) uses 659 μs to encode and 3.65 ms to decode. This gives the us approximate run time encode to be $\frac{659}{28} = 24$ μs pr. information bit, while decode uses $\frac{3.65}{28} = 130$ μs pr. information bit. We summarize in table 5

	Encoding pr. bit	Decoding pr. bit
BCH-Code	22 μs	163 μs
Reed-Solomon + Convolutional Code	24 μs	130 μs

Table 5: Summary of task 5

6 Task 6: Testing Part II

We run our code in our simulation environment, using our binary symmetric channel again.

In all cases, we sample 1.000.000 bits this time, since our implementation is much faster, and we get more interesting results than what we got in Task 3. The results are given in table

	$p = (0.005)$		$p = (0.01)$	
	Est. error-rate	Measured error-rate	Est. error-rate	Measured error-rate
Data	$2.41 \cdot 10^{-5}$	$4.7 \cdot 10^{-5}$	$9.23 \cdot 10^{-4}$	$9.97 \cdot 10^{-4}$
C&C	$2.72 \cdot 10^{-11}$	0	$6.00 \cdot 10^{-7}$	$9.5 \cdot 10^{-5}$

Table 6: Summary of task 3. Each message is 1.000.000 bits

We immidiately notice that our estimates for the BCH-codes (data channel) are very good. For $p = 0.005$, a $4.7 \cdot 10^{-5}$ part of 1.000.000 is 47 errors, which given that the block size is 92, is likely caused by a single block error, which is about what we whould expect given our estimates (0 or 1 maybe). For $p = 0.01$ however, the variance is lower, and the estimate is as we see very close to what we got in table 4.

Again, for our concatinated code, it is more difficult. Given that we are aiming for an error rate of less than 10^{-9}, given a 1.000.000-bit message, we should get 0 errors as we do. However, we see that our "guesstimate" for $p = 0.01$ is off by two orders of magnitude, and unfortunately in the wrong direction. We also note that our concatinated code only beats the BCH code by one order of magnitude for $p = 0.01$, even though it has way worse information rate. This suggests that there may be alot of improvement for our C&C implementation (and the fact that it might possibly not manage the required 10^{-9} error-rate limit).

7 Conclusion

Unfortunately, measuring very very small error-rates proved difficult, and in some cases, we were unable to do it. Further, analyzing the error-rate of our Reed-Solomon + Convolutional Code proved difficult aswell, and experimental results show that we might have parameterized it quite poorly.

Apart from this, we are glad that all of our implementations *seems* to work as they should. Further, we happily conclude that not only did we increase the information rate from task $1, 2, 3$ to task $4, 5, 6$, but the speed of our codes also increased with about two orders of magnitude, which was nice!

8 Bonus

We use our BCH code for a fun example right at the end here. Using the binary symmetric channel with $p = 0.01$, we transmit a 546KB photo from the surface of mars. The result is

shown in figure 1

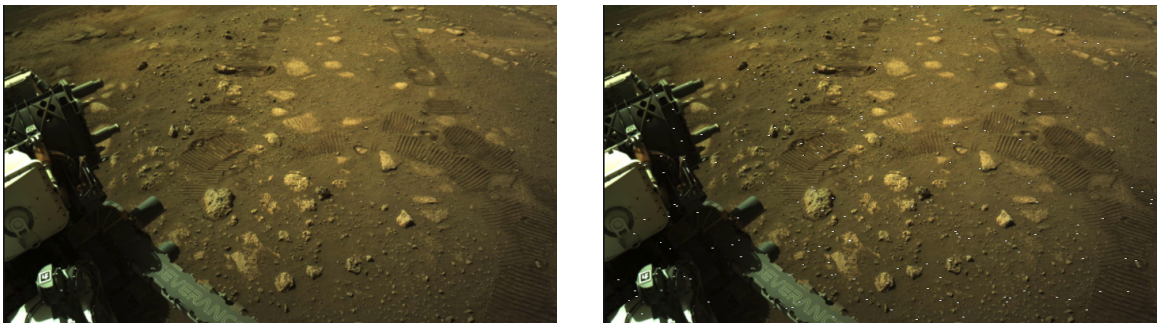

Figure 1: Picture before and after having been transmitted from "Mars"

As can be seen in the notebook, this corresponds to an error rate of about $1.04 \cdot 10^{-3}$ (Still very close to our estimate for $p = 0.01$!). Because we use such large block-sizes, and decoding errors complitely destroy the block, the result is clearly seen as $3 - 4$ pixels in a row just become obliterated. For this specific purpuse, maybe some kind of interleaving could have worked better, to distribute errors as less noticable errors over more pixels? Open ended question, as I have not had time to look into it for now...

9 Appendix (next page...)

Task 2 and 3

April 23, 2021

1 Task 2

1.1 Reed-Muller Code

Encoding and decoding

[2]:
```
import itertools # Needed for combinations

######################
#
#      Setup
#
######################

# Generates all elements in F_2^m (encoded as strings)
def genbinstrings(m, binstr, b=''):
    if len(b)==m:
        binstr.append(b)
    else:
        genbinstrings(m, binstr, b + '0')
        genbinstrings(m, binstr, b + '1')

# The map phi: V --> (F_2)^(2^m)
def phi(f, F2m):
    phif = []
    for B in F2m:
        Bnew = [F(b) for b in B]
        phif.append(f(Bnew))
    return phif

# Generates monomial basis for R(r, m)
def genM(r,m,indeterminates):
    M = [1]
    for i in range(1,r+1):
        for subset in itertools.combinations(indeterminates, i):
            M.append(prod(subset))
    return M
```

```
#Precomputes all sets S_{fj}
def genSfj(M, m, r, binstrings, F2m, indeterminates):
    getSfj = {}
    for f in M:
        binstr = [b + '0'*r for b in binstrings]
        divisors = []
        for x in indeterminates:
            if not x.divides(f):
                divisors.append(x)
        Sfj = []
        for i in range(2**(m-r)):
            si = []
            for j in range(len(divisors)):
                si.append(divisors[j]-F(binstr[i][j]))
            Sfj.append(prod(si))
        phiSfj = []
        for s in Sfj:
            phis = phi(s, F2m)
            phiSfj.append(phis)
        getSfj[f] = phiSfj
    return getSfj

def setupRM(r, m, F):
    #Order elements in F_2^m
    F2m = []
    genbinstrings(m, F2m)

    #Basis for RM(r,m)
    indeterminates = K.gens()
    M = genM(r, len(indeterminates), list(indeterminates))

    #Needed for Sfj construction
    binstrings = []
    genbinstrings(m-r, binstrings)

    #Precomputing S_{fj}'s
    getSfj = genSfj(M, m, r, binstrings, F2m, indeterminates)

    return F2m, M, getSfj


#######################
#
#      Encoding
#
#######################
```

```
def createPoly(x):
    f = 0
    for i in range(len(M)):
        f += x[i]*M[i]
    return f

def encodeblockRM(x):
    f = createPoly(x)
    y = phi(f, F2m)
    return y

def encodeRM(x, k):
    y = []
    for i in range(len(x)//k):
        xi = x[i*k:(i+1)*k]
        y += encodeblockRM(xi)
    return y


######################
#
#      Decoding
#
######################

def decodeblockRM(z, r, m):
    zj = vector(z)
    x = []
    for j in range(len(M)-1, -1, -1):
        vote = {0:0, 1:0}
        fj = M[j]
        Sfj = getSfj[fj]
        for phis in Sfj:
            vote[zj.dot_product(vector(phis))] += 1
        if vote[0] > vote[1]:
            xj = 0
        else:
            xj = 1
        x.append(xj)
        if j == 0:
            return x[::-1]
        phifj = phi(fj, F2m)
        zj = zj - xj*vector(phifj)

def decodeRM(z, r, m):
    w = []
    n = 2**m
```

```
    for i in range(len(z)//(n)):
        zi = z[i*n:(i+1)*n]
        w += decodeblockRM(zi, r, m)
    return w


F = GF(2)
K.<x1, x2, x3, x4, x5, x6> = PolynomialRing(F)
r, m = 3, len(K.gens())
F2m, M, getSfj = setupRM(r,m,F)
k = len(M)

x = [randint(0,1) for i in range(k*10)]
y = encodeRM(x, k)
w = decodeRM(y, r, m)
print(w == x)
```

```
True
```

1.1.1 Measure running-time

Command and Control:

[37]:
```
F = GF(2)
K.<x1, x2, x3, x4, x5> = PolynomialRing(F)
r, m = 1, len(K.gens())
F2m, M, getSfj = setupRM(r,m,F)
k = len(M)
```

[22]:
```
x = [randint(0,1) for i in range(k)]
%timeit encodeRM(x, k)
y = encodeRM(x, k)
%timeit decodeRM(y, r, m)
w = decodeRM(y, r, m)
print(w==x)
```

```
35.8 ms ± 2.99 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
205 ms ± 5.53 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
True
```

Data transfer:

[23]:
```
F = GF(2)
K.<x1, x2, x3, x4, x5, x6> = PolynomialRing(F)
r, m = 3, len(K.gens())
F2m, M, getSfj = setupRM(r,m,F)
k = len(M)
```

```
42
```

[27]:
```
x = [randint(0,1) for i in range(k)]
%timeit encodeRM(x, k)
y = encodeRM(x, k)
%timeit decodeRM(y, r, m)
w = decodeRM(y, r, m)
print(w==x)
```

```
76.1 ms ± 5.9 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
3.13 s ± 127 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
True
```

2 Task 3

We first provide the implementation of the binary symmetric channel, as well as methods for padding messages and calculating the error-rate

[6]:
```
from random import randint, random

#Calulate the error rate
def errorrate(x, w):
    if len(x) != len(w):
        error("Messages are not even of same length...")
    ers = 0
    for i in range(len(x)):
        if x[i] != w[i]:
            ers += 1
    print("number of errors = {}".format(ers))
    return N(ers/len(x), digits = 30)

#Applying noise to a single bit
def noise(b, p):
    if random() < p:
        if b == 1:
            return 0
        else:
            return 1
    else:
        return b

#The binary symmetric channel
def transmit(y, p):
    return [noise(b, p) for b in y]

#Pad message to have a multiple of k bits
def pad(x, k):
    while len(x) % k != 0:
        x.append(0)
```

```
    return x
```

2.0.1 Command and Control:

[12]:
```
F = GF(2)
K.<x1, x2, x3, x4, x5> = PolynomialRing(F)
r, m = 1, len(K.gens())
F2m, M, getSfj = setupRM(r,m,F)
k = len(M)
```

[13]:
```
p = 0.005
x = pad([randint(0,1) for i in range(30000)], k)
print("sending message of length {}".format(len(x)))
print("encoding...")
y = encodeRM(x, k)
print("transmitting...")
z = transmit(y, p)
print("decoding...")
w = decodeRM(z, r, m)
print("\n")
print("errorrate = {}".format(errorrate(x, w)))
```

```
sending message of length 30000
encoding…
transmitting…
decoding…


number of errors = 0
errorrate = 0.00000000000000000000000000000000
```

[14]:
```
p = 0.01
x = pad([randint(0,1) for i in range(30000)], k)
print("sending message of length {}".format(len(x)))
print("encoding...")
y = encodeRM(x, k)
print("transmitting...")
z = transmit(y, p)
print("decoding...")
w = decodeRM(z, r, m)
print("\n")
print("errorrate = {}".format(errorrate(x, w)))
```

```
sending message of length 30000
encoding…
transmitting…
```

```
decoding…


number of errors = 0
errorrate = 0.00000000000000000000000000000000
```

Data transfer

[15]:
```
F = GF(2)
K.<x1, x2, x3, x4, x5, x6> = PolynomialRing(F)
r, m = 3, len(K.gens())
F2m, M, getSfj = setupRM(r,m,F)
k = len(M)
```

[16]:
```
p = 0.005
x = pad([randint(0,1) for i in range(30000)], k)
print("sending message of length {}".format(len(x)))
print("encoding...")
y = encodeRM(x, k)
print("transmitting...")
z = transmit(y, p)
print("decoding...")
w = decodeRM(z, r, m)
print("\n")
print("errorrate = {}".format(errorrate(x, w)))
```

```
sending message of length 30030
encoding…
transmitting…
decoding…


number of errors = 0
errorrate = 0.00000000000000000000000000000000
```

[10]:
```
p = 0.01
x = pad([randint(0,1) for i in range(30000)], k)
print("sending message of length {}".format(len(x)))
print("encoding...")
y = encodeRM(x, k)
print("transmitting...")
z = transmit(y, p)
print("decoding...")
w = decodeRM(z, r, m)
print("\n")
print("errorrate = {}".format(errorrate(x, w)))
```

```
sending message of length 30030
encoding…
```

```
transmitting…
decoding…


number of errors = 51
errorrate = 0.00169830169830169830169830169830
```

Task 5 and 6

April 23, 2021

1 Task 5

1.1 Primitive, narrow-sense BCH code

[297]:
```
from random import randint, random

def allCyclotomicCosets(q, n):
    Z = Integers(n)
    allC = []
    if gcd(q,n) != 1:
        error("No no no no no")
    for i in range(1, n):
        Ci = []
        for j in range(0, n):
            Ci.append(Z(i)*Z(q)**j)
        #print("C{} = {}".format(i, sorted(set(Ci))))
        allC.append(sorted(set(Ci)))
    return allC


allC = allCyclotomicCosets(2, 127)
zeroset = allC[0:9]
zeroset = set([zero for Ci in zeroset for zero in Ci])
print(zeroset)
print("k = {}".format(127-len(zeroset)))
```

```
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 14, 16, 17, 18, 20, 24, 28, 32, 33, 34, 36,
40, 48, 56, 64, 65, 66, 67, 68, 72, 80, 96, 97, 112}
k = 92
```

[266]:
```
F = GF(2)
n = 127
E = GF(128,'w')
a = E.gen()
K.<X> = PolynomialRing(E)
d = 11
g = prod([(X-a^i) for i in zeroset])
k = 92
```

```
#Borrowed from provided notebook "l17-berlekamp-massey.jpynb"
def bm(F, R, s):
    X = R.0
    C = R.one()
    B = R.one()
    L = 0
    deltap = 1
    m = -1
    for j in range(len(s)):
        delta = sum([ s[j-i]*C[i] for i in range(C.degree()+1)])
        if delta != 0:
            T = C
            C = C - delta/deltap*X^(j-m)*B
            if 2*L <= j:
                B = T
                m = j
                deltap = delta
                L = j + 1 - L
    return (L,C)

# Polynomials with zeroes as coefficients at highest degree terms loose some␣
 ↪bits when converting...
def pad(x, k):
    while len(x) % k != 0:
        x.append(0)
    return x

def encodeblockBCH(x, g, X):
    xpoly = sum([x[i]*X**i for i in range(len(x))])
    return pad((xpoly*g).coefficients(sparse=False), 127) # Return list of␣
 ↪coefficients to get bits

def encodeBCH(x, g, X, k):
    y = []
    for i in range(len(x)//k):
        xi = x[i*k:(i+1)*k]
        y += encodeblockBCH(xi, g, X)
    return y

def decodeblockBCH(z, a, d, F, R, k):
    zpol = R(z) # Turn list of bits into polynomial
    syn = [zpol(a^i) for i in range(1,d)] # Syndromes
    Lambda = bm(F,R,syn)[1] # Berlekamp-Massey Algorithm to find the error␣
 ↪locator polynomial
    w = []
    for i in range(127):
```

```
        if Lambda(a**(-i)) == 0:
            zpol += X**(i) # Fix errors
    if g.divides(zpol):
        return pad((R(zpol/g)).coefficients(sparse=False), 92) # Convert back␣
↪to list of bits before returning
    else:
        return [0]*k

def decodeBCH(z, a, d, F, R, n, k):
    w = []
    for i in range(len(z)//(n)):
        zi = z[i*n:(i+1)*n]
        w += decodeblockBCH(zi, a, d, F, R, k)
    return w

x = [randint(0,1) for i in range(92)]
y = encodeBCH(x, g, X, k)
y[1] += 1
y[31] += 1
y[51] += 1
y[71] += 1
y[91] += 1
z = y
w = decodeBCH(z, a, d, F, K, n, k)
print(w == x)
```

```
True
```

1.1.1 Measuring runtime

[214]:
```
x = [randint(0,1) for i in range(92)]
%timeit y = encodeBCH(x, g, X, k)
y = encodeBCH(x, g, X, k)
%timeit decodeBCH(y, a, d, F, K, n, k)
w = decodeBCH(y, a, d, F, K, n, k)
print(w==x)
```

```
2 ms ± 96.4 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
15 ms ± 584 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
True
```

1.2 Reed Solomon + Convolutional Code

1.2.1 Reed Solomon part

[226]:
```
F = GF(16,'w')
n = 15
a = F.gen()
R = PolynomialRing(F,'X')
X = R.0
zeroset = range(1,9)
d = 9
g = prod([(X-a^i) for i in zeroset])
k = 7

# Polynomials with zeroes as coefficients at highest degree terms loose some␣
 ↪bits when converting...
def pad(x, k):
    while len(x) % k != 0:
        x.append(0)
    return x

def bitsToGF16(bits, a):
    li = []
    for i in range(len(bits)//4):
        block = bits[i*4:(i+1)*4]
        li.append(sum([block[j]*a^j for j in range(4)]))
    return li

def GF16ToBits(x):
    if x == 0:
        return [0, 0, 0, 0]
    else:
        l = (x).polynomial().coefficients(sparse=False)
        return pad(l, 4)

def GF16POLtoBits(x, padlength):
    x = pad(x.coefficients(sparse=False), padlength)
    x = [GF16ToBits(c) for c in x]
    return [b for c in x for b in c]

def encodeblockRS(x, a, g, X, n):
    x = bitsToGF16(x, a)
    xpoly = sum([x[i]*X**i for i in range(len(x))])
    return GF16POLtoBits(xpoly*g, n) # Return list of coefficients to get bits

def encodeRS(x, a, g, X, n, k):
    k = k*4
    y = []
```

```
    for i in range(len(x)//(k)):
        xi = x[i*k:(i+1)*k]
        y += encodeblockRS(xi, a, g, X, n)
    return y

def decodeblockRS(z, F, R, a, n, k, d):
    z = bitsToGF16(z, a)
    zpol = R(z) # Turn list of coefficients into polynomial
    syn = [zpol(a^i) for i in range(1,d)] # Syndromes
    (w, Lambda) = bm(F,R,syn)
    Xmatrix = []
    locations = []
    for i in range(n):
        if Lambda(a**(-i)) == 0:
            locations.append(i)
    for i in range(1,len(locations)+1):
        Xmatrix.append([(a**j)**i for j in locations])
    Xmatrix = Matrix(Xmatrix)
    Sv = vector(F, syn[:len(locations)])
    errors = Matrix(Xmatrix).solve_right(Sv)
    for i in range(len(locations)):
        zpol -= errors[i]*X**locations[i]
    if g.divides(zpol):
        return GF16POLtoBits(R(zpol/g), k)
    else:
        return [0]*4*k

def decodeRS(z, a, d, F, R, n, k):
    w = []
    n4 = n*4
    for i in range(len(z)//(n4)):
        zi = z[i*n4:(i+1)*n4]
        w += decodeblockRS(zi, F, R, a, n, k, d)
    return w

x = pad([randint(0,1) for i in range(100)], 4*7)
y = encodeRS(x, a, g, X, n, k)
y[2] += 1
y[10] += 1
y[20] += 1
y[30] += 1
z = y
w = decodeRS(z, a, d, F, R, n, k)
print(w)
print(x==w)
```

```
[1, 0, 1, 1, 0, 0, 1, 0, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 0, 1, 0, 0, 1, 1, 0,
```

```
0, 0, 0, 0, 1, 0, 0, 1, 1, 1, 1, 1, 0, 0, 1, 1, 0, 1, 1, 1, 0, 1, 1, 0, 1, 1, 0,
0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 0, 1, 1, 1, 1,
0, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 1, 0, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0]
True
```

1.3 Convolutional Code part

[227]:
```
def curcuit(b, sreg):
    y1 = b^^sreg[0]^^sreg[1]
    y2 = b^^sreg[1]
    sreg[1] = sreg[0]
    sreg[0] = b
    return y1, y2

def convencode(x):
    sreg = [0,0]
    y = []
    for i in range(len(x)):
        y1, y2 = curcuit(x[i], sreg)
        y.append(y1)
        y.append(y2)

    #Make sure to end up in zero state
    for i in range(2):
        y1, y2 = curcuit(0, sreg)
        y.append(y1)
        y.append(y2)
    return y


def err(bb, cc):
    c = 0
    for i in range(2):
        if bb[i] != cc[i]:
            c += 1
    return c

def nextnode0(in00, in01):
    if in00 < in01:
        return [in00, 0]
    else:
        return [in01, 2]

def nextnode1(in10, in11):
    if in10 < in11:
        return [in10, 1]
```

```
    else:
        return [in11, 3]


def decodeViterbi(obs):
    #[weight at node, previous node]
    trelli = [[[0, -1], [100000, -1], [100000, -1], [100000, -1]]]
    for i in range(len(obs)//2):
        bb = obs[i*2:(i+1)*2]
        nxt = [0,0,0,0]
        s00in0 = err(bb, [0,0])
        s00in1 = err(bb, [1,1])
        s10in0 = err(bb, [1,0])
        s10in1 = err(bb, [0,1])
        s01in0 = err(bb, [1,1])
        s01in1 = err(bb, [0,0])
        s11in0 = err(bb, [0,1])
        s11in1 = err(bb, [1,0])

        nxt[0] = nextnode0(s00in0+trelli[i][0][0], s01in0+trelli[i][2][0])
        nxt[1] = nextnode0(s00in1+trelli[i][0][0], s01in1+trelli[i][2][0])
        nxt[2] = nextnode1(s10in0+trelli[i][1][0], s11in0+trelli[i][3][0])
        nxt[3] = nextnode1(s10in1+trelli[i][1][0], s11in1+trelli[i][3][0])

        trelli.append(nxt)

    #Backtracking
    l = len(trelli)
    states = [0]
    w = [0]
    for i in range(1,l-1):
        prev = trelli[l-i][states[-1]][1]
        states.append(prev)
        if prev % 2 == 0:
            w.append(0)
        else:
            w.append(1)
    return w[::-1][:-2] # Remember to remove pad


x = [randint(0,1) for i in range(100000)]
#print(x)
y = convencode(x)
w = decodeViterbi(y)
#print(w)
print(w==x)
```

```
True
```

1.3.1 Combining Reed Solomon + Convolutional Code

[228]:
```
def encodeFull(x, a, g, X, n, k):
    x2 = encodeRS(x, a, g, X, n, k)
    x2 = [int(k) for k in x2]
    return convencode(x2)

def decodeFull(z, a, d, F, R, n, k):
    z2 = decodeViterbi(z)
    return decodeRS(z2, a, d, F, R, n, k)

x = [randint(0,1) for i in range(4*7)]
print(x)
y = encodeFull(x, a, g, X, n, k)

if y[10] == 1:
    y[10] = 0
else:
    y[10] = 1

z = y
w = decodeFull(z, a, d, F, R, n, k)
print(w)
print(x == w)
```

```
[0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 0, 1, 1, 0,
0, 0]
[0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 0, 1, 1, 0,
0, 0]
True
```

1.3.2 Measuring Run-time

[229]:
```
x = [randint(0,1) for i in range(4*7)]
%timeit y = encodeFull(x, a, g, X, n, k)
y = encodeFull(x, a, g, X, n, k)
z = y
%timeit w = decodeFull(z, a, d, F, R, n, k)
w = decodeFull(z, a, d, F, R, n, k)

print(x==w)
```

```
659 µs ± 52.4 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
3.65 ms ± 373 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
True
```

2 Task 6

2.1 Data Transfer:

[230]:
```
#Calulate the error rate
def errorrate(x, w):
    if len(x) != len(w):
        error("Messages are not even of same length...")
    ers = 0
    for i in range(len(x)):
        if x[i] != w[i]:
            ers += 1
    print("number of errors = {}".format(ers))
    return N(ers/len(x), digits = 30)

#Applying noise to a single bit
def noise(b, p):
    if random() < p:
        return b+1
    else:
        return b

#The binary symmetric channel
def transmit(y, p):
    return [noise(b, p) for b in y]
```

[231]:
```
F = GF(2)
n = 127
E = GF(128,'w')
a = E.gen()
K.<X> = PolynomialRing(E)
d = 11
g = prod([(X-a^i) for i in zeroset])
k = 92
```

[258]:
```
p = 0.005
x = pad([randint(0,1) for i in range(1000000)], 92)
print("sending message of length {}".format(len(x)))
print("encoding...")
y = encodeBCH(x, g, X, k)
print("transmitting...")
z = transmit(y, p)
print("decoding...")
w = decodeBCH(z, a, d, F, K, n, k)
print("\n")
print("errorrate = {}".format(errorrate(x, w)))
```

```
sending message of length 1000040
```

```
encoding…
transmitting…
decoding…


number of errors = 47
errorrate = 0.000046998120075196992120315187392 5
```

[261]:
```
p = 0.01
x = pad([randint(0,1) for i in range(1000000)], 92)
print("sending message of length {}".format(len(x)))
print("encoding...")
y = encodeBCH(x, g, X, k)
print("transmitting...")
z = transmit(y, p)
print("decoding...")
w = decodeBCH(z, a, d, F, K, n, k)
print("\n")
print("errorrate = {}".format(errorrate(x, w)))
```

```
sending message of length 1000040
encoding…
transmitting…
decoding…


number of errors = 997
errorrate = 0.000996960121595136194552217911284
```

2.2 Command and Control:

[262]:
```
F = GF(16,'w')
n = 15
a = F.gen()
R = PolynomialRing(F,'X')
X = R.0
zeroset = range(1,9)
d = 9
g = prod([(X-a^i) for i in zeroset])
k = 7
```

[246]:
```
p = 0.005


x = pad([randint(0,1) for i in range(1000000)], 4*7)
print("sending message of length {}".format(len(x)))
print("encoding...")
y = encodeRS(x, a, g, X, n, k)
```

```
print("transmitting...")
z = transmit(y, p)
print("decoding...")
w = decodeRS(z, a, d, F, R, n, k)
print("\n")
print("errorrate = {}".format(errorrate(x, w)))
```

```
sending message of length 1000020
encoding…
transmitting…
decoding…


number of errors = 0
errorrate = 0.00000000000000000000000000000000
```

[264]:
```
p = 0.01


x = pad([randint(0,1) for i in range(1000000)], 4*7)
print("sending message of length {}".format(len(x)))
print("encoding...")
y = encodeRS(x, a, g, X, n, k)
print("transmitting...")
z = transmit(y, p)
print("decoding...")
w = decodeRS(z, a, d, F, R, n, k)
print("\n")
print("errorrate = {}".format(errorrate(x, w)))
```

```
sending message of length 1000020
encoding…
transmitting…
decoding…


number of errors = 95
errorrate = 0.00009499810003799924001519969600 61
```

2.3 Bonus Transmitting a photo from mars:

[268]:
```
from PIL import Image
from numpy import asarray

# Probability of error
p = 0.01
```

```
# Open image, convert it to a list of 1 and 0
im = Image.open('MarsSurface.png')
picture = im.tobytes()
pictureBits = [int(c) for c in ''.join([format(byte, '08b') for byte in␣
 ↪picture])]
# Encode
x = pad(pictureBits, 92)
y = encodeBCH(x, g, X, k)

# Transmit
z = transmit(y, p)

# Decode
w = decodeBCH(z, a, d, F, K, n, k)
print("errorrate = {}".format(errorrate(x, w)))

w = w[:len(pictureBits)]
# Turn list of 1 and 0 into bytes (This is really slow...)
s = ''.join([str(b) for b in w])
picture = bytes(int(s[i : i + 8], 2) for i in range(0, len(s), 8))

# Recreate image from bytes
image = Image.frombytes(im.mode, im.size, picture, 'raw')

# Display image
display(image)
```

```
number of errors = 9384
errorrate = 0.0010447073283146412659394684283 3
```