Algebra and Coding Homework 3

-------Algebra and Coding Homework 3-------

#|########Student: Touer Mohamed Elamine.

#|##########Group: 4.

#|##Academic Year: 2024/2025.

#|######Professor: Rezki Chemlal.

-------Algebra and Coding Homework 3-------

0- Basic Info:

- **Hardware:** Intel Core I7-6600U CPU @2.60GHz 2.81Ghz, RAM: 8GB. (on a Windows 10 OS)
- **Software:** Visual Studio Code, the code is in Python 3.11.4.
- **Links:** The code used in this homework can be found in the following repo: click here
- (https://github.com/GoJita5/Algebra_Coding3 in case the hyperlink doesn't work).
- Important Info:
  1- I mostly use notations from R. A. Hill's book: A first course in coding theory.

  2- I don't usually write much details in the homework document, but it's been a fun (and painful) journey. So here are most of the details (with the rest being the tests and functions that didn't make it into the final file.)

1- Setup:

We import the necessary libraries

```
import numpy as np
import itertools as it
from time import time
```

Important notations:
$V(r,2) := \{0,1\}^r$
$n := 2^r - 1$

The hamming code $Ham(r,2)$, which we will call $\mathcal{C}$, is a code that has H as its parity-check matrix, where H is matrix defined by: each column of H is a nonzero element of $V(r,2)$.

This means that H has $|V(r,2)| - 1 = n$ rows.
so H^t is an $n \times r$ matrix, so the vectors of C must be of length n.

Also, keep in mind that C is a vector space of dimension $n-r$, so $|\mathcal{C}| = 2^{n-r}$.

First, we need a function to create H.

1.1- Creating H:

This is the code for H:

```
def ham(r):
    vr2 = it.product([0,1], repeat=r)
    next(vr2)  #remove (0,0,...,0)
    h = np.fromiter(it.chain(*vr2), dtype=int)
    h = np.transpose(h.reshape((2**r)-1,r))
    return h
```

`vr2 = it.product([0,1], repeat=r)` yields an itertools product object containing all the vectors of $V(r,2)$. The nice thing about it is that it does it in increasing binary order, for example, for r=2, it yields: `(0,0), (0,1), (1,0), (1,1)`.
Since a product object is like a generator object, this means that unlike lists and arrays, these vectors can't be accessed directly , but only through iteration, or through those weird functions that turn iterators into lists.

`next(vr2)` skips the first element, which is always the 0 vector.

`h = np.fromiter(it.chain(*vr2), dtype=int)` is one of the weird functions I was talking about, but it turns the iterator into an array. I should've used `np.fromiter` solely but it doesn't work with product objects, so I turned it into a chain object using `it.chain`.
Since the elements of vr2 are tuples and not lists, I used the unpacking operator `*` on them.

This unfortunately forces `h` to be an array made of one big row, so I have to reshape it using the `h = np.transpose(h.reshape((2**r)-1,r))` line. "Transposition" was necessary since the `reshape` function actually gave me the transpose of `h`, and yes I tried writing `(r,(2**r)-1)` instead but it didn't work.

1.2- Creating C:

We'll use the fact that

$$\mathcal{C} = \{y \in V(n,2) : y \cdot H^t = 0_{1\times r}\}$$

to construct $\mathcal{C}$.
{While it is possible to create $\mathcal{C}$ by using its generator matrix G and multiplying it over all the vectors of length $n-r$, I'm not going to do it, first because there's little to no gain in terms of time complexity, and second, its too complicated, because we need to construct G out of H.}

This means we'll test all the vectors of the space $V(n,2)$ and whenever one satisfies the formula, we add it to $\mathcal{C}$.

```
def HammingCode1(r, reshape=True):
    n = (2**r)-1
    v = it.product([0,1], repeat=n)
    c = np.array([], dtype=int)
    h = ham(r)
    for x in v:
        y = np.array(x)
        z = h.dot(y) #This is y*(H)^T.
        z = z%2
        if (np.any(z) == 0): #Check if y*(H)^T is zero on Z/2Z.
            c = np.append(c,y)
    if reshape == True:
        return c.reshape(2**(n-r) ,n) #reshape c so that each row is a codeword
    else: #Or just leave it as one big row.
        return c
```

`z = h.dot(y)` is kinda unintuitive, it looks like $H \cdot y^t$, but this is actually $y \cdot H^t$, I don't know why numpy behaves this way.
Of course, this product could yield a number greater than 1, so to fix this I wrote `z = z%2`.

I don't know how to deal with the `append` function in numpy, apparently, when you don't specify the optional parameter `axis` it flattens `c`.
So the end result is an array of one row containing all the codewords, which is returned if `reshape=False`.
If `reshape=True`, `c` is reshaped so that each row is a codeword, then returned.
I used `reshape(2**(n-r), n)` because I know $\mathcal{C}$. is a set of 2^{n-r} vectors, each of length n.
I thought at first that the `reshape=False` option will be useful in the following algorithms, but it isn't, so I should omit it.

1.3- Standard Table:

The good thing about the Hamming code is that we know exactly its coset leaders. Given that it is a perfect code for one error correction, the leaders are $0_{1\times n}$ (the zero of $V(n,2)$) and the standard basis vectors of $V(n,2)$, call them $e_j := (0,\dots,1,\dots,0)$.

Thus the cosets of $\mathcal{C}$ are $\mathcal{C}, e_1+\mathcal{C},\dots,e_n+\mathcal{C}$.

Since $V(n,2)$ is a disjoint union of the cosets, the standard table S is basically all of $V(n,2)$ arranged so that each row is a coset, starting from $\mathcal{C}$ and ending with $e_n+\mathcal{C}$.

Thus the code is as follows:

```
def StandardTable(r, reshape=1):
    #Creates the standard table of the Hamming code, assuming we want to correct only one
error.
    n = (2**r)-1
    c = HammingCode1(r)
    S = np.array([c])
```

```
    eyedentity = np.eye(n, dtype=int) #These are all the possible coset leaders, with A[j-1]
= (0,...,0,1,0,....,0), 1 is at the jth position.
    for i in eyedentity:
        S = np.append(S, c+i)%2
    if reshape == 1: #This reshapes S so that every row i of S is a_i + C, where a_i is a
coset leader
        return S.reshape(n+1, n*(2**(n-r)))
    elif reshape == 2: #This reshapes S into a one-column table, so every row i of S is a
vector of the space V(n,2)
        return S.reshape((n+1)*(2**(n-r)), n)
    else: #Leaves S as one big row of all the vectors in the space.
        return S
```

Note1: since the `append` function flattens the array, the result is one big row containing all of the vectors of $V(n,2)$, but ordered in this way: the vectors of $\mathcal{C}$ first (with them being also ordered in the original order they appeared in in $\mathcal{C}$), then the vectors of $e_1+\mathcal{C}$, then,..., then the vectors of $e_n+\mathcal{C}$. Of course it all appears as a sequence of 1's and 0's. So a reshape is necessary.

`if reshape == 1:`
We know that S is made of $n+1$ distinct cosets, each coset has the same number of elements of $\mathcal{C}$, which is 2^{n-r}. the elements of $\mathcal{C}$ are vectors of length n. If we want to reshape S so that each row of S is a coset, *Note1* ensures that taking the rows in `reshape` as `n+1` and the columns as `(2**(n-r))*n`, i.e, writing `S.reshape(n+1, n*(2**(n-r)))`, will yield the result.
The only problem is that we can't treat the elements of each cosets as separate arrays, but the whole coset is one flattened array (basically each coset takes the form of `reshape=False` that appears in the HammingCode1 function). Its not that bad for display.

`if reshape == 2:`
Since I don't know much about numpy, and I couldn't get non-flattened cosets in the previous case (to be able to find the row and column of each vector directly), I turned the table into one big column, whose rows are the vectors of the space, this is also possible to do by *Note1*, so it's enough to take the rows of `reshape` as the number of cosets times their cardinality, i.e., `(n+1)*(2**(n-r))`, and the columns as the vector lengths `n`.
This form is necessary for the `stdecode` algorithm

`else:`
Leaves the table as one big row, unnecessary case.

1.4-Finding the error position:

If we receive a vector x that has at most 1 error, we can calculate its syndrome $g=xH^t$, we know it will always be equal to the syndrome of only one of the coset leaders e_iH^t. given that e_i is a vector that has only 1 in the ith position, this means that g equals the ith row of H^t, i.e., the ith column of H. But since H was originally created in increasing binary order, we can see that the ith row of H is just the binary representation of i.
In other words, we don't need to find the row or column of the Standard table in which x lies, we also don't

need to create a table of coset leaders and their syndromes, all we need to do is to convert z from binary into decimal to find which coset leader e_i we are dealing with.

This is what the following function does.

```
def bin(x:np.ndarray): #Takes an array of binary numbers and gives the decimal representation
of it.
    a = 2**np.arange(np.size(x)-1, -1, -1)
    return np.dot(x, a)
```

`a` is an array containing the powers of 2, starting from 2^{r-1} and stopping at $2^0 = 1$. This is to give the greatest weight to the initial values of `x`, and the least weight to the last values.
Then we return the dot product of `a` and `x`.
for example if `x=np.array([0,0,1])` (r=3) then `bin(x)=1`, and if `x=np.array([1,0,1])` then `bin(x)=5`.

2-The Solution:

The professor asked for functions 1-3, but I added function 4 just to compare it to these algorithms, since func 4 is the original method of correcting errors.

Every function below receives an array `x` of length n, and they all have the parameter `r`, which is in fact unnecessary since we defined $n := 2^r - 1$ so $r = \log_2 n$. I just didn't want to import `log` nor `log2`, and it doesn't matter much as I'm not sending the code to the world.

2.1- Standard Table Decoding:

All it does is find the position of the received vector x in the standard table.

```
def Stdecode(x:np.ndarray, r): #a) Standard Table decoding.
    n = (2**r)-1 #Size of x
    A = np.eye(n, dtype=int) #All coset leaders of weight 1 (i.e. a_j, j>=1)
    s = StandardTable(r, reshape=2)
    w = np.where(np.all(s==x, axis=1)) #Finding the row where x exists in the one-column
table.

    v = w[0]

    j = v[0]//(2**(n-r)) #This yields the row in which x exist in the actual table.
    if j == 0:
        return x
    else:
        x = (x-A[j-1])%2
        return x
```

The table represents all the vectors of the space so x is always there, and it's enough to find in which row x lies, and the first element of that row would be the coset leader.

Let $a_0 = 0_{1\times n}$ and $a_i = e_i$, $\forall i \in \{1,\dots,n\}$. Let the position of x in S be denoted by v
Notice that we use `reshape=2` on the standard table to be able to deal with each vector separately.
By our construction of S, x is in a coset $a_j + \mathcal{C}$ if and only $j \cdot 2^{n-r} + 1 \leq v < (j+1)\cdot 2^{n-r} + 1$. This is a python table, i.e. it starts at position 0 and ends at at $(n+1)\cdot 2^{n-r} - 1$. so we can write $j \cdot 2^{n-r} \leq v < (j+1)\cdot 2^{n-r} \cdots (1)$.
This is because we can partition the table as $a_0 + \mathcal{C}, a_1 + \mathcal{C}, \dots, a_n + \mathcal{C}$, where each coset has exactly 2^{n-r} elements.
Naturally, a_j's position is exactly $j \cdot 2^{n-r}$.

From (1) we deduce that $j \leq \frac{v}{2^{n-r}} < j+1$, so $j = \lfloor \frac{v}{2^{n-r}} \rfloor$.

To get v, we use the line `w = np.where(np.all(s==x, axis=1))`.

`if j==0:` x is already on the first row of S, so it is a codeword.
`else:` x is in the coset $a_j + \mathcal{C}$, so return $x - a_j$.

2.2- Syndromes Decoding:

Refer to 1.4.
I could've created the code this way: Create a table containing all coset leaders a_i and their syndromes, then calculate the syndrome of x, it will be the same as that of a_j for some j, then fetch j from the table and return $x - a_j$.
But this is obviously inefficient, and the explanation in part 1.4 provides a better way to fetch a_j, so that I could compare the difference between using $x - a_j$ (i.e. this function), and flipping the jth bit of x (i.e. the function of part 2.3). This is what the teacher asked for.

The code is self-explanatory

```
def Syndrome(x:np.ndarray, r): #b) Syndromes decoding.
    n = (2**r)-1
    h = ham(r)
    g = (h.dot(x))%2 #This is the syndrome of x.
    b = bin(g) #This is the decimal representation of g.
    if b == 0: #If it's 0, then no error has occurred, so x is a codeword.
        return x
    e = np.zeros(n, dtype=int) #If not, then an error has occurred in position b of x, to
correct it, return x-e
    e[b-1] = 1 #where e is a vector equal to 1 at position b.
    return (x-e)%2
```

2.3- One Error Correction Algorithm:

It's the same as the previous algorithm, but instead of using $x - a_j$, flip the jth bit of x.

```
def Oneerrcorr(x:np.ndarray, r): #c) One error correction algorithm.
    n = (2**r)-1
```

```
    h = ham(r)
    g = (h.dot(x))%2
    b = bin(g)
    if b == 0:
        return x
    x[b-1] = 1-x[b-1] #Flip the bit at position b. Should be faster for big r.
    return x
```

2.4- Nearest Neighbor Decoding:

This is how we first learned to correct errors.

Since $\mathcal{C}$ is a perfect code for correcting one error, we can ensure there's always a "closest" codeword to x.

```
def Nearestneighbor(x:np.ndarray, r): #d) Correcting the error in x using the closest
codeword to x.
    C = HammingCode1(r)
    for c in C:
        y = np.sum((x-c)%2) #This is w(x-c) which equals d(x,c).
        if y == 0 or y == 1: #One of these two conditions must hold, because C is a perfect
error correcting code.
            return c
```

It's enough to calculate the weight of `x-c` where `c` is a codeword, until we find a `c` whose distance from x is 1, or one that is equal to x. This way, we don't have to compare x with every vector of $\mathcal{C}$, nor do we have to store codewords along with their distances in an array to get the one of minimum value.

3-Testing:

3.1- Test Function:

This is the testing function:

```
def bigtest(func1, func2, r, k=0, disp=False, appr=18):
    #Tests 2 of the decoding functions with some or all vectors of size n, and gives
execution times.
    #Also rounds the times to 'appr' decimal places (appr stands for "approximate to"). If
you wish to see each individual test, set disp to True.
    n = (2**r)-1
    f1time= []
    f2time= []
    if k==0: #If the number of vectors is not specified, tests all the vectors of the space.
        V = it.product([0,1], repeat=n)
    else: #If the number is specified, it tests k vectors of the space randomly.
        V = np.random.randint(2, size=(k, n))
    for x in V:
        if disp == True:
            print("----------------------")
```

```
            print("Test for: ", x)
        y = np.array(x)
        t0 = time()
        func1(y, r)
        t1 = time()
        func2(y, r)
        t2 = time()
        a = t1-t0
        b=t2-t1
        f1time.append(round(a, appr))
        f2time.append(round(b, appr))
        if disp == True:
            print(f"{func1.__name__}: {round(a, appr)} seconds")
            print(f"{func2.__name__}: {round(b, appr)} seconds")

    if disp == True:
        print("\n*************************\n-------------------------
\n*************************\n")
    print(f"total:\n\t{func1.__name__} times: {f1time}\n\t{func2.__name__} times: {f2time}")
    c = sum(f1time)/len(f1time)
    d = sum(f2time)/len(f2time)
    print(f"----------\nAverages:\n\t{func1.__name__} times: {round(c,
appr)}\n\t{func2.__name__} times: {round(d, appr)}")
```

I call it bigtest because it can test the whole space, also it's the biggest function in the file.

The test function gives the execution time for each 2 of the functions I wrote for a given `r` and a given number of vectors `k`.

If `k` is set to `0`, it tests them for the whole space $V(n, 2)$.

Otherwise, we choose `k` random vectors from the space using the line:
`V = np.random.randint(2, size=(threshold, n))`. `2` signifies we choose elements less than 2, i.e. we choose elements in {0, 1}, and `size=(threshold, n)` means we have `k` vectors of size `n`.

In any case, we start going through a loop of tests:
if `disp==True` we display each individual test vector, and the time it took for function1 and function2 to correct it. Otherwise we don't display it.
We get the execution time `a` for function1, and `b` for function2, we append `a` and `b`, rounded to `appr` decimal places set by the user, to the tables `f1time` and `f2time` respectively.

Then the function prints these two tables, along with the average time it took for each function to perform the correction. The averages are also rounded to `appr` decimal places.

Here's the input of an example comparison:

```
bigtest(Syndrome, Nearestneighbor, r=4, k=5, disp=True, appr=6)
```

Here's the output:

```
----------------------
Test for:  [1 0 0 1 1 0 0 0 1 1 0 1 0 0 0]
Syndrome: 0.000995 seconds
Nearestneighbor: 0.432842 seconds
----------------------
Test for:  [0 0 0 0 0 1 1 0 0 0 1 0 0 0 1]
Syndrome: 0.0 seconds
Nearestneighbor: 0.455784 seconds
----------------------
Test for:  [0 1 0 0 0 1 0 0 0 1 0 1 0 1 1]
Syndrome: 0.0 seconds
Nearestneighbor: 0.407908 seconds
----------------------
Test for:  [0 1 0 1 0 1 0 0 0 1 1 0 1 1 1]
Syndrome: 0.0 seconds
Nearestneighbor: 0.383973 seconds
----------------------
Test for:  [1 0 1 0 1 0 0 1 0 1 1 1 0 0 1]
Syndrome: 0.0 seconds
Nearestneighbor: 0.397939 seconds

************************
-------------------------
************************

total:
        Syndrome times: [0.000995, 0.0, 0.0, 0.0, 0.0]
        Nearestneighbor times: [0.432842, 0.455784, 0.407908, 0.383973, 0.397939]
----------
Averages:
        Syndrome times: 0.000199
        Nearestneighbor times: 0.415689
```

The function could get better in multiple ways like returning the best average, or returning the tables of times, or being able to compare less or more than two functions, but implementing all of these features is a headache. This is enough for now.

Now that I think about it, I should've written it such that it gives the times of only one function, and tests it multiple times, but it doesn't matter. Wait, I'll just make one now:

```
def actualtest(func1, r, k=0, testcount=1, appr=18):
    #Tests one of decoding functions t times with some or all vectors of size n, and gives
```

```
execution times, their averages, and the total average.
    #Also rounds the times to 'appr' decimal places.
    n = (2**r)-1
    averages = []
    print("_____________________________________________________________________")
    for i in range(testcount):
        test1 = []
        if k==0: #If the number of vectors is not specified, tests all the vectors of the
space.
            V = it.product([0,1], repeat=n)
        else: #If the number is specified, it tests k vectors of the space randomly.
            V = np.random.randint(2, size=(k, n))
        for x in V:
            y = np.array(x)
            t0 = time()
            func1(y, r)
            t1 = time()
            a = t1-t0
            test1.append(round(a, appr))
        print(f"\ntest{i+1} = {test1}\n----------------")
        avg= round(sum(test1)/len(test1), appr)
        averages.append(avg)
        print(f"avg{i+1} = {avg}")
        print("_____________________________________________________________________\n")

    AVERAGE = round(sum(averages)/len(averages), appr)

print("*********************************************************************************")
    print(f"------------------------ AVERAGE = {AVERAGE} --------------------------------")
print("*********************************************************************************")
```

3.2- The Tests:

Note: This section contains numerical data of the tests, if you're not interested in numbers but only in graphs, feel free to skip it.

I'll test the functions with as many possible values of r, and 20 vectors (if the space has less than 20, this means some vectors have been tested multiple times, anyways the random choice doesn't ensure uniqueness), or less if necessary. The results are approximated to 6 decimal places.
I'll repeat the test 3 times.

1) Stdecode:

`r=1:`

```
_____________________________________________________________

test1 = [0.0, 0.0, 0.0, 0.000998, 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, 0.000996, 0.0, 0.0]
----------------
avg1 = 0.0001
_____________________________________________________________

test2 = [0.0, 0.0, 0.0, 0.000989, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.000995, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.000998, 0.0]
----------------
avg2 = 0.000149
_____________________________________________________________

test3 = [0.001003, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.000987, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0]
----------------
avg3 = 0.0001
_____________________________________________________________

*************************************************************************
------------------------ AVERAGE = 0.000116 -------------------------------
*************************************************************************
```

r=2:

```
_____________________________________________________________

test1 = [0.0, 0.00099, 0.0, 0.0, 0.000998, 0.0, 0.000997, 0.0, 0.000997, 0.0, 0.0, 0.0,
0.000997, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.000996]
----------------
avg1 = 0.000299
_____________________________________________________________

test2 = [0.0, 0.000998, 0.000999, 0.0, 0.0, 0.0, 0.0, 0.0, 0.000996, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.000997, 0.0, 0.0, 0.0, 0.0]
----------------
avg2 = 0.0002
_____________________________________________________________

test3 = [0.0, 0.0, 0.0, 0.0, 0.000997, 0.0, 0.0, 0.0, 0.0, 0.0, 0.000997, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.000997, 0.0, 0.0]
----------------
avg3 = 0.00015
_____________________________________________________________
```

```
**************************************************************************
------------------------ AVERAGE = 0.000216 -------------------------------
**************************************************************************
```

r=3:

```
_____________________________________________________________

test1 = [0.001995, 0.001994, 0.000998, 0.001995, 0.001995, 0.000997, 0.001995, 0.001995,
0.001996, 0.003989, 0.002, 0.000992, 0.002005, 0.001999, 0.000981, 0.001996, 0.003015,
0.003966, 0.001994, 0.001995]
----------------
avg1 = 0.002045
_____________________________________________________________

test2 = [0.001993, 0.000997, 0.001994, 0.002996, 0.002008, 0.001978, 0.000997, 0.001994,
0.000998, 0.001996, 0.000997, 0.001995, 0.001994, 0.002994, 0.001993, 0.001997, 0.001992,
0.001995, 0.001995, 0.000996]
----------------
avg2 = 0.001845
_____________________________________________________________

test3 = [0.001993, 0.001997, 0.002991, 0.001995, 0.001993, 0.000998, 0.001996, 0.001995,
0.000996, 0.003011, 0.001978, 0.001994, 0.001996, 0.001992, 0.001995, 0.001994, 0.000998,
0.001993, 0.002995, 0.001993]
----------------
avg3 = 0.001995
_____________________________________________________________

**************************************************************************
------------------------ AVERAGE = 0.001962 -------------------------------
**************************************************************************
```

r=4:

```
test1 = [0.617348, 0.548535, 0.597401, 0.476726, 0.569477, 0.544546, 0.591417, 0.475727,
0.463759, 0.592415, 0.490689, 0.528586, 0.560501, 0.488693, 0.582443, 0.476724, 0.465755,
0.472735, 0.499475, 0.498243]
----------------
avg1 = 0.52706
_____________________________________________________________

test2 = [0.505647, 0.504652, 0.517616, 0.635301, 0.551529, 0.557509, 0.528585, 0.658244,
```

```
0.528023, 0.478719, 0.476725, 0.54654, 0.51707, 1.177849, 0.484704, 0.544543, 0.50466,
0.508638, 0.465757, 0.603802]
---------------
avg2 = 0.564806
_____________________________________________________________

test3 = [0.521603, 0.553854, 0.506933, 0.521606, 0.494189, 0.48914, 0.489245, 0.503659,
0.515086, 0.491241, 0.505198, 0.519116, 0.504259, 0.527473, 0.596827, 0.552438, 0.551525,
0.533159, 0.511646, 0.570396]
---------------
avg3 = 0.52293
_____________________________________________________________

*************************************************************************
------------------------ AVERAGE = 0.538265 -------------------------------
*************************************************************************
```

The execution times are getting much bigger, I'll reduce the number of vectors to 1 and the tests to 1 for the next case.
`r=5:` started at 12:02:25

```
I stopped the program after 16 mins, this is already too much.
```

Given how fast the execution average grows, I suppose it will take hours for even one vector in the case of `r=5`, so I'll stop here.

2) Syndrome:

From now on I'll only write the average.

`r=1,2,3:`

```
Average is always <3.3*10^-5, sometimes a bit more.
```

`r=4,5,6:`

```
Average around 5*10^-5, sometimes less.
```

`7<=r<=18:`

```
r = 7,  AVERAGE = 0.000141
r = 8,  AVERAGE = 0.000212
r = 9,  AVERAGE = 0.000497
r = 10, AVERAGE = 0.000762
r = 11, AVERAGE = 0.001688
```

```
r = 12, AVERAGE = 0.004396
r = 13, AVERAGE = 0.010376
r = 14, AVERAGE = 0.022112
r = 15, AVERAGE = 0.047225
r = 16, AVERAGE = 0.122978
r = 17, AVERAGE = 0.247249
r = 18, AVERAGE = 0.533834
```

The time it takes for each test to yield the result is getting bigger, so we reduce the number of vectors to 5.

`19<=r=22:`

```
r = 19, AVERAGE = 1.032130
r = 20, AVERAGE = 2.847949
r = 21, AVERAGE = 5.292109
r = 22, AVERAGE = 11.906927
```

Its getting too slow, I'm reducing each test to 1 vector.

`23<=r<=24:`

```
r = 23, AVERAGE = 30.307088
r = 24, AVERAGE = 86.058183
```

Last one, with only one test of one vector:

`r=25:`

```
AVERAGE = 921.187659
```

Roughly 15 mins. I'm not testing any more.

3) Oneerrcorr:

I'll test it in a similar way to Syndrome.

`r=1,2,3:`

```
Average is always <3.3*10^-5, sometimes a bit more.
```

`r=4,5,6:`

```
Average around 5*10^-5, more or less.
```

`7<=r<=18:`

```
r = 7,  AVERAGE = 0.000116
r = 8,  AVERAGE = 0.000249
r = 9,  AVERAGE = 0.000315
r = 10, AVERAGE = 0.000715
r = 11, AVERAGE = 0.001413
r = 12, AVERAGE = 0.003308
r = 13, AVERAGE = 0.007164
r = 14, AVERAGE = 0.025415
r = 15, AVERAGE = 0.035672
r = 16, AVERAGE = 0.091223
r = 17, AVERAGE = 0.208885
r = 18, AVERAGE = 0.407726
```

The following tests are reduced to 5 vectors too:

`19<=r=22:`

```
r = 19, AVERAGE = 1.112938
r = 20, AVERAGE = 1.916405
r = 21, AVERAGE = 4.204347
r = 22, AVERAGE = 9.311131
```

Now, each test has only one vector:

`23<=r<=24:`

```
r = 23, AVERAGE = 18.552687
r = 24, AVERAGE = 53.292143
```

And finally, a unique test of one vector:

`r=25:`

```
AVERAGE = 901.056936
```

4) Nearestneighbor:

`1<=r<=5:`

```
r = 1, AVERAGE = 6.6*10^-5
r = 2, AVERAGE = 0.000183
r = 3, AVERAGE = 0.002344
```

```
r = 4, AVERAGE = 0.504369
r = 5, AVERAGE = #Only one test of one vector. Also stopped after 16 mins.
```

3.3- Graphs and Comparisons:

In this section we compare the performance of the functions visually, this is the code I wrote to plot the results:

```
#This is the code for plotting some results I collected.

import matplotlib.pyplot as plt

# 1<=r<=5, the case of r=5 is considered as 960 seconds (16 mins), it's actually way greater
than that.
Stdecodet = [0.000116, 0.000216, 0.001962, 0.538265, 960]
Nearestneighbort = [6.6e-05, 0.000183, 0.002344, 0.504369, 960]
# 7<=r<=24, the cases r<7 is uninteresting, it's almost always 0. and the case r=25 will ruin
the plot.
Syndromet = [0.000141, 0.000212, 0.000497, 0.000762, 0.001688, 0.004396, 0.010376, 0.022112,
0.047225, 0.122978,
             0.247249, 0.533834, 1.032130, 2.847949, 5.292109, 11.906927, 30.307088,
86.058183]
Oneerrcorrt = [0.000116, 0.000249, 0.000315, 0.000715 ,0.001413, 0.003308, 0.007164,
0.025415, 0.035672, 0.091223,
               0.208885, 0.407726, 1.112938, 1.916405, 4.204347, 9.311131, 18.552687,
53.292143]

#Figure1: Stdecode vs Nearestneighbor.
plt.figure(figsize=(10, 6))
plt.plot(list(range(1, 5)), Stdecodet[0:4], label='Stdecode times', marker="o",
color="black")
plt.plot(list(range(1, 5)), Nearestneighbort[0:4], label="Nearestneighbor times", marker='x',
color="green")

plt.legend()
plt.grid(True)
plt.xlabel('r', loc='right')
plt.ylabel('Execution Time (Seconds)', loc='top')


#Figure2: Syndrome vs Oneerrcorr.
plt.figure(figsize=(10, 6))
plt.plot(list(range(7, 25)), Syndromet, label='Syndrome times', marker='d', color="red")
plt.plot(list(range(7, 25)), Oneerrcorrt, label="Oneerrcorr times", marker='*',
color="yellow")

plt.legend()
plt.grid(True)
```

```
plt.xlabel('r', loc='right')
plt.ylabel('Execution Time (Seconds)', loc='top')


for i in range(0,6):
    Syndromet.insert(0, 0)
    Oneerrcorrt.insert(0, 0)
for i in range(0,19):
    Stdecodet.append(960)
    Nearestneighbort.append(960)


#Figure3: Stdecode vs Oneerrcorr.
plt.figure(figsize=(10, 6))
plt.plot(list(range(1, 25)), Stdecodet, label='Stdecode times', marker='o', color="black")
plt.plot(list(range(1, 25)), Oneerrcorrt, label="Oneerrcorr times", marker='*',
color="yellow")

plt.legend()
plt.grid(True)
plt.xlabel('r', loc='right')
plt.ylabel('Execution Time (Seconds)', loc='top')


#Figure4: Everything.
plt.figure(figsize=(10, 6))
plt.plot(list(range(1, 25)), Stdecodet, label='Stdecode times', marker='o', color="black")
plt.plot(list(range(1, 25)), Oneerrcorrt, label="Oneerrcorr times", marker='*',
color="yellow")
plt.plot(list(range(1, 25)), Syndromet, label='Syndrome times', marker='d', color="red")
plt.plot(list(range(1, 25)), Nearestneighbort, label="Nearestneighbor times", marker='x',
color="green")

plt.legend()
plt.grid(True)
plt.xlabel('r', loc='right')
plt.ylabel('Execution Time (Seconds)', loc='top')


plt.show()
```

But I lack any willpower to explain how it works, not that it matters anyways.

We first compare the slowest algorithms, Stdecode (1) and Nearestneighbor (4).
Given how complicated (1) is, I supposed it would be much slower than (4). Here's the plot for $r \leq 4$:

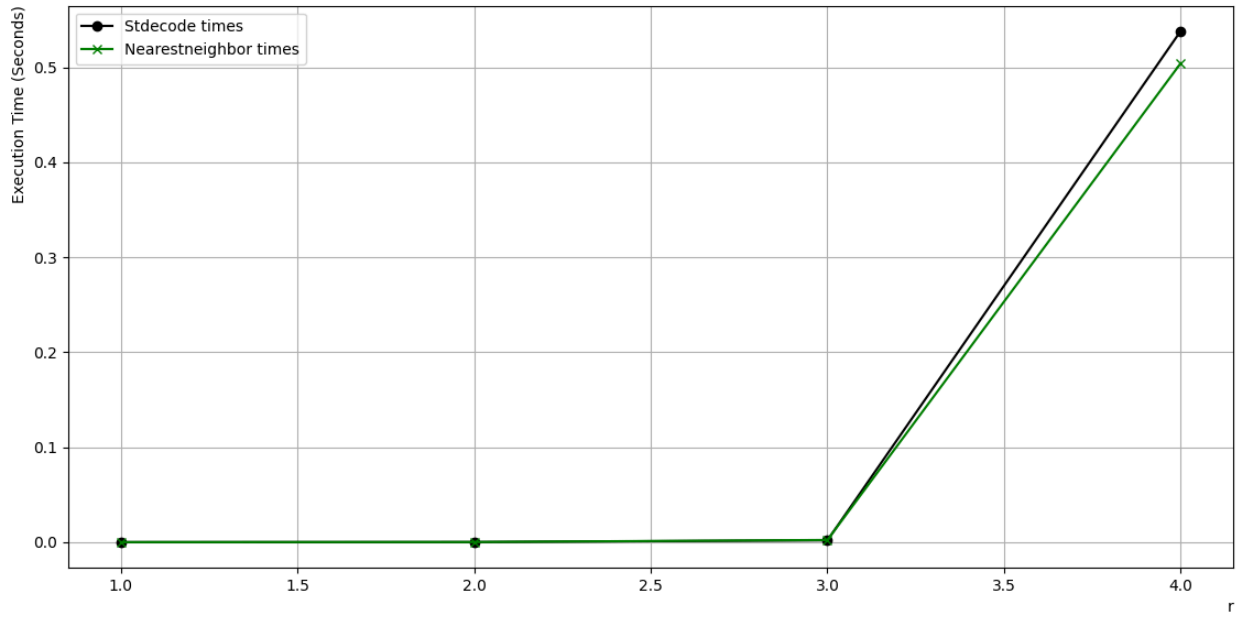

Fig1: Stdecode vs Nearestneighbor.

It's surprising that there isn't that much of a difference. I think the true difference will appear in the case $r = 5$, but since it took too much time I wouldn't bother.

Now we compare the fastest algorithms, Syndome (2), and Oneerrcorr (3):

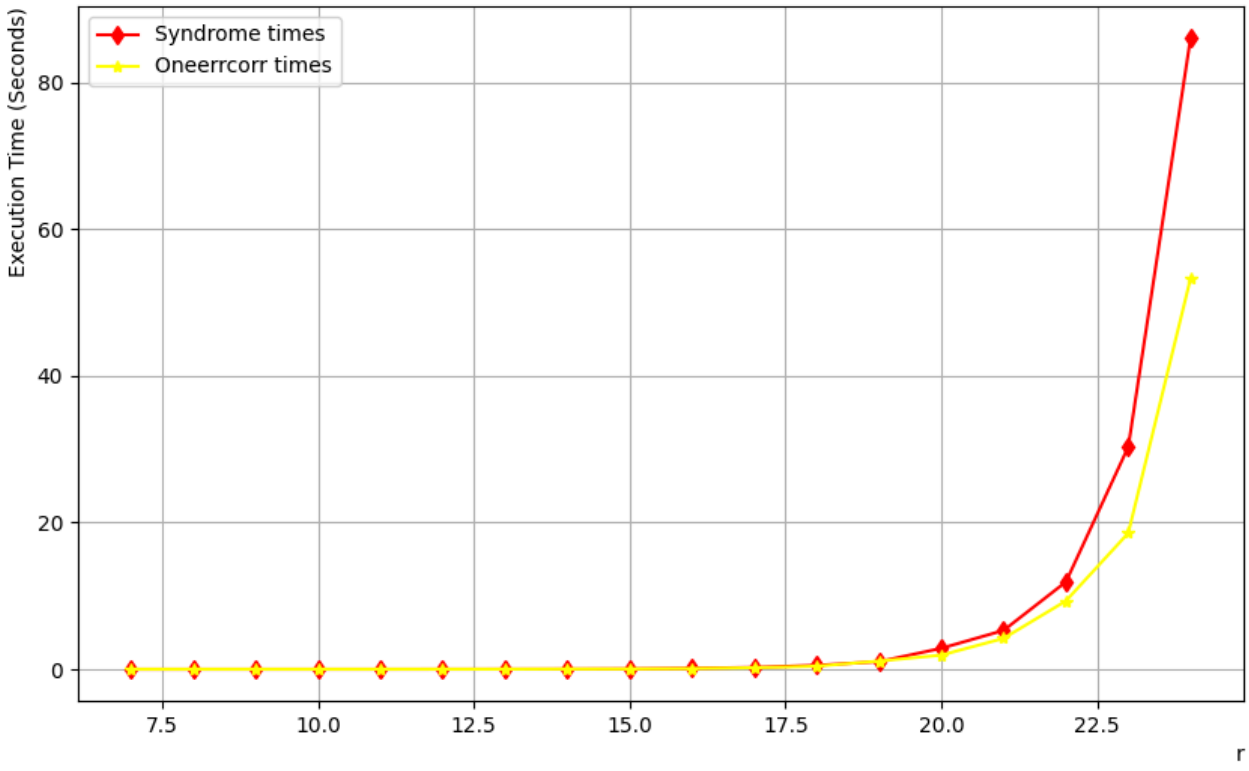

Fig2: Syndrome vs Oneerrcorr

As expected, (3) is somewhat faster than (2), the difference is negligible at first but as r grows it starts to appear.

So it's obvious that Oneerrcorr is the fastest and Stdecode is the slowest, let's compare them.

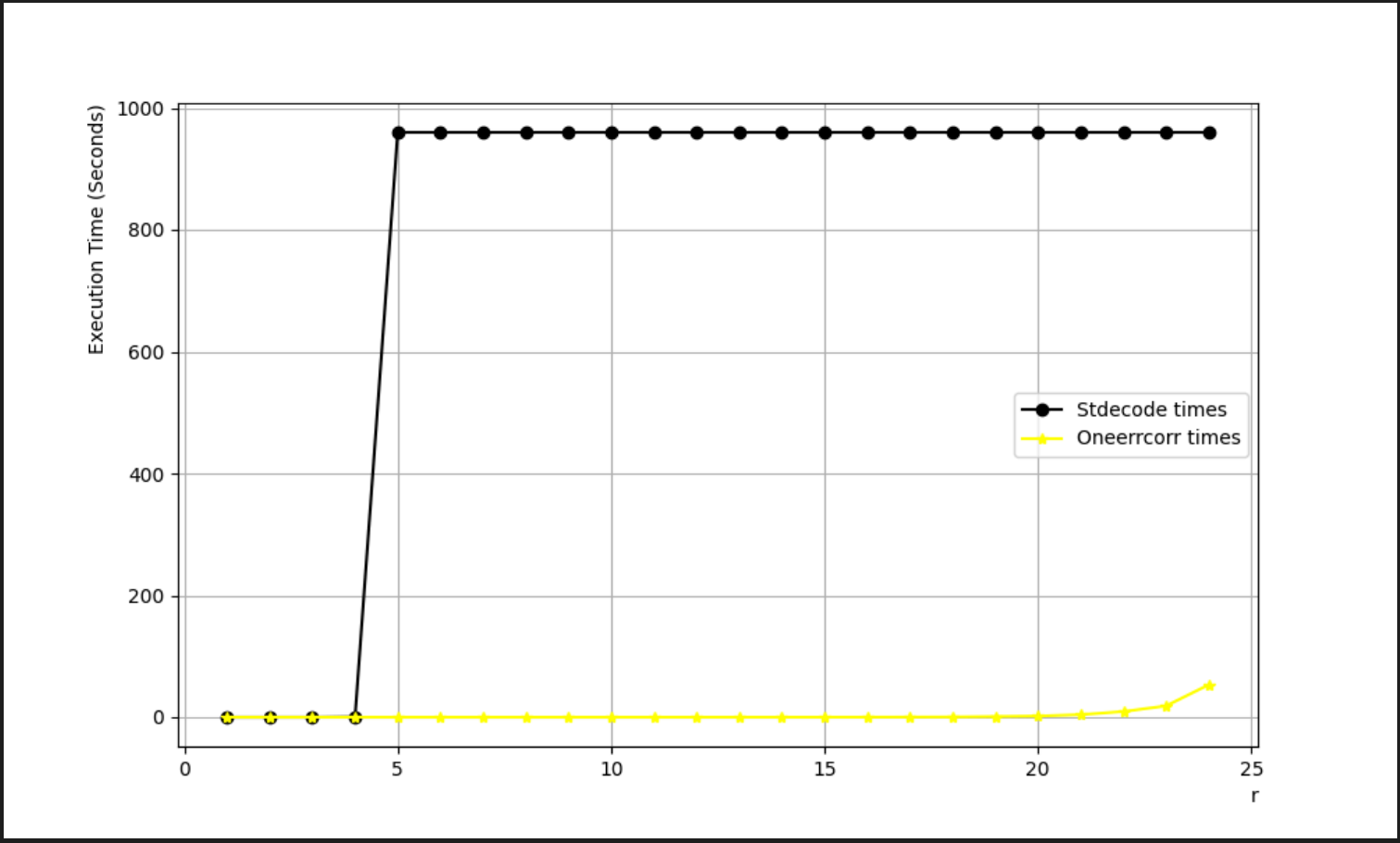

Fig3: Stdecode vs Oneerrcorr

Notice that I supposed in the graph that, for `r=5`, Stdecode takes 960 seconds, i.e. 16 minutes, while in reality takes much more than that, and I supposed that it stays constant at 960 seconds for `r>5` while it would probably take years.
For Oneerrcorr, I supposed that the time it takes for `r<7` is 0, since the time it took was negligible anyways.

It's interesting that Stdecode jumps so quickly at `r=5`, but looking at its table, this is natural:
`Stdecodet = [0.000116, 0.000216, 0.001962, 0.538265, 960]`.
the time it takes for `r=2` is approximately double as what it takes for `r=1`, then at `r=3`, the time grows around 9 fold, then at `r=4`, it grows 274 times. So since the growth itself is increasing so rapidly, the result for `r=5` shouldn't be surprising.

Meanwhile, Oneerrcorr is blazingly fast even for `r=20`, and it only starts to seriously slow down after `r=23`, then it blows up at `r=25` to 960 seconds.

Now, here's a rough comparison of all functions:

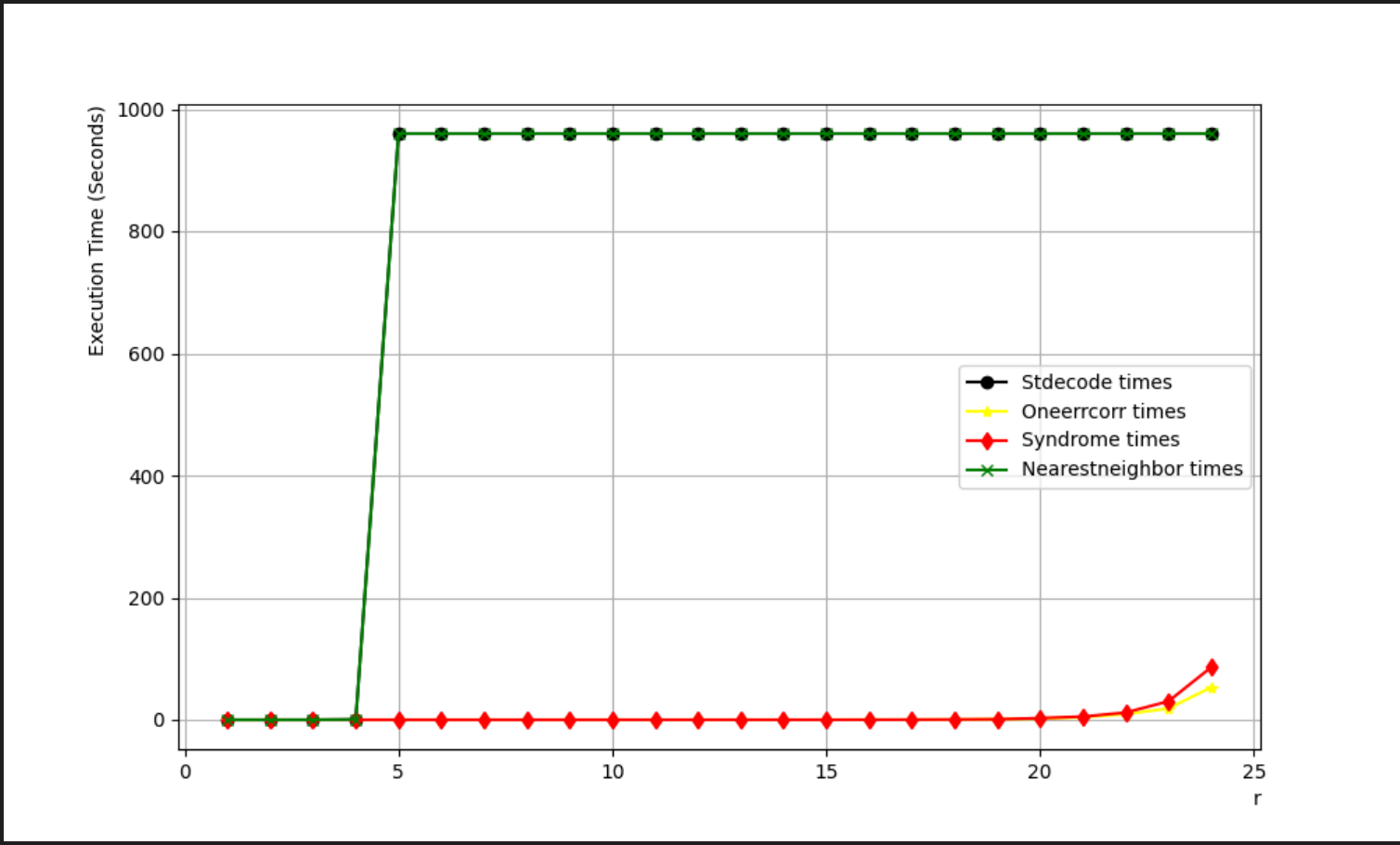

Fig4: All functions.

The big picture is: Nearestneighbor and Stdecode are much slower than the Oneerrcorr and Syndrome code, and they blow up very early on, and I suspect that only then the difference between Nearest and St will appear. Oneerrcorr and Syndrome take longer to blow up, and the difference between them starts to appear only at the beginning of the blow up.

4-Conclusion:

The one error correcting algorithm `Oneerrcorr` is the fastest one of the four algorithms, because it performs the least possible number of operations.

The syndromes decoding `Syndrome` is also good but it adds the unnecessary operation of substraction of zero $n-1$ times.

The Nearest Neighbor decoding algorithm `Nearestneighbor` is bad enough, it requires the creation of $\mathcal{C}$ and the calculation of the weight of the difference of x and many (if not all) of its codewords.

The Standard Table decoding algorithm `Stdecode` is by far the worst. It has too many unnecessary operations: It requires the creation of $\mathcal{C}$ and the creation of *all* of its cosets, and the search for the position of x in the table, and returning $x - a_i$.