Linear Block Code

Shahin Majazi

September 2, 2023

Implementation of Encoder and Decoder for Linear Block Code C(5, 2)

[1]:
```
import numpy as np
from time import time
import itertools as it
from termcolor import colored
```

Encoder :

Lookup Table that Includes Standard Array and P Array:

[2]:
```
LOOK_UP_TABLE_P = {(5, 2): {'P': np.array([[1, 1, 1], [1, 0, 1]], dtype=np.
↪int64), 'Standard_Array': \
                            {'e0': [[0, 0, 0, 0, 0], [1, 0, 1, 0, 1], [1, 1, 1,␣
↪1, 0], [0, 1, 0, 1, 1]], \
                             'e1': [[0, 0, 0, 0, 1], [1, 0, 1, 0, 0], [1, 1, 1,␣
↪1, 1], [0, 1, 0, 1, 0]], \
                             'e2': [[0, 0, 0, 1, 0], [1, 0, 1, 1, 1], [1, 1, 1,␣
↪0, 0], [0, 1, 0, 0, 1]], \
                             'e3': [[0, 0, 1, 0, 0], [1, 0, 0, 0, 1], [1, 1, 0,␣
↪1, 0], [0, 1, 1, 1, 1]], \
                             'e4': [[0, 1, 0, 0, 0], [1, 1, 1, 0, 1], [1, 0, 1,␣
↪1, 0], [0, 0, 0, 1, 1]], \
                             'e5': [[1, 0, 0, 0, 0], [0, 0, 1, 0, 1], [0, 1, 1,␣
↪1, 0], [1, 1, 0, 1, 1]], \
```

```
                              'e6': [[0, 0, 1, 1, 0], [1, 0, 0, 1, 1], [1, 1, 0,␣
↪0, 0], [0, 1, 1, 0, 1]], \
                              'e7': [[0, 1, 1, 0, 0], [1, 1, 0, 0, 1], [1, 0, 0,␣
↪1, 0], [0, 0, 1, 1, 1]]}}}
```

[3]:
```
St_Arr = LOOK_UP_TABLE_P[(5, 2)]['Standard_Array']
print(f'\n{colored("For Systematic Linear Block Code of C(5, 2) Standard Array␣
↪is:", "blue", attrs=["bold"])}\n')

for key in St_Arr.keys():
    print(f'\n{colored(key, "red", attrs=["bold"])} = \n{St_Arr[key]}\n')
```

```
For Systematic Linear Block Code of C(5, 2) Standard Array is:


e0 =
[[0, 0, 0, 0, 0], [1, 0, 1, 0, 1], [1, 1, 1, 1, 0], [0, 1, 0, 1, 1]]


e1 =
[[0, 0, 0, 0, 1], [1, 0, 1, 0, 0], [1, 1, 1, 1, 1], [0, 1, 0, 1, 0]]


e2 =
[[0, 0, 0, 1, 0], [1, 0, 1, 1, 1], [1, 1, 1, 0, 0], [0, 1, 0, 0, 1]]


e3 =
[[0, 0, 1, 0, 0], [1, 0, 0, 0, 1], [1, 1, 0, 1, 0], [0, 1, 1, 1, 1]]


e4 =
[[0, 1, 0, 0, 0], [1, 1, 1, 0, 1], [1, 0, 1, 1, 0], [0, 0, 0, 1, 1]]


e5 =
[[1, 0, 0, 0, 0], [0, 0, 1, 0, 1], [0, 1, 1, 1, 0], [1, 1, 0, 1, 1]]


e6 =
[[0, 0, 1, 1, 0], [1, 0, 0, 1, 1], [1, 1, 0, 0, 0], [0, 1, 1, 0, 1]]


e7 =
[[0, 1, 1, 0, 0], [1, 1, 0, 0, 1], [1, 0, 0, 1, 0], [0, 0, 1, 1, 1]]
```

Note:
The error pattern e_6 in above has been corrected.

G Generation:

[4]:
```
def G_generator(Linear_Block_Code: tuple) -> np.ndarray:
    n, k = Linear_Block_Code
    I_k = np.identity(k, dtype=np.int64)
    P = LOOK_UP_TABLE_P[Linear_Block_Code]['P']
    G = np.concatenate((P, I_k), axis=1, dtype=np.int64)
    return G
```

Test:

[5]:
```
G = G_generator(Linear_Block_Code=(5, 2))
print(f'\n{colored("For Systematic Linear Block Code of C(5, 2) G Matrix will be:
↪", "blue", attrs=["bold"])}\n\n{colored("G =", "black", attrs=["bold"])}␣
↪\n{G}\n')
```

```
For Systematic Linear Block Code C(5, 2) G Matrix is:

G =
[[1 1 1 1 0]
 [1 0 1 0 1]]
```

U Generation:

[6]:
```
def U_generator(k: int) -> np.ndarray:
    U = np.array(list(it.product([0, 1], repeat=k)), dtype=np.int64)
    return U
```

Test:

[7]:
```
U = U_generator(k=2)
print(f'\n{colored("For k=2 U Matrix that includes our Messages will be: ",␣
↪"blue", attrs=["bold"])}\n\n{colored("U =", "black", attrs=["bold"])}\n{U}\n')
```

```
For k=2 U Matrix that includes our Messages will be:

U =
[[0 0]
```

 [0 1]
 [1 0]
 [1 1]]

V Generation Using 2 Methods:

[8]:
```
def Coder_1(Linear_Block_Code: tuple) -> np.ndarray:
    n, k = Linear_Block_Code
    G = G_generator(Linear_Block_Code=Linear_Block_Code)
    U = U_generator(k=k)
    V = (U @ G) % 2
    return V
```

Test:

[9]:
```
tic = time()
V = Coder_1(Linear_Block_Code=(5, 2))
toc = time()
run_time_coder_1 = toc - tic
print(f'\n{colored("For Systematic Linear Block Code of C(5, 2) V Matrix that␣
 ↪includes our Codewords will be:", "blue", attrs=["bold"])}\
\n\n{colored("V =", "black", attrs=["bold"])} \n{V}\n')
print(f'\n{colored("Run-time: ", "red", attrs=["bold"])}{run_time_coder_1: 0.5f}␣
 ↪(s)\n')
```

```
For Systematic Linear Block Code C(5, 2) V Matrix that includes our

Codewords will be:

V =
[[0 0 0 0 0]
 [1 0 1 0 1]
 [1 1 1 1 0]
 [0 1 0 1 1]]


Run-time:  0.00045 (s)
```

[10]:
```
def Coder_2(Linear_Block_Code: tuple) -> np.ndarray:
    k = Linear_Block_Code[1]
    P = LOOK_UP_TABLE_P[Linear_Block_Code]['P']
    U = U_generator(k=k)
    Parity_mat = (U @ P) % 2
    V = np.concatenate((Parity_mat, U), axis=1)
```

```
    return V
```

Test:

[11]:
```
tic = time()
V = Coder_2(Linear_Block_Code=(5, 2))
toc = time()
run_time_coder_2 = toc - tic
print(f'\n{colored("For Systematic Linear Block Code of C(5, 2) V Matrix that␣
 ↪includes our Codewords will be:", "blue", attrs=["bold"])}\
\n\n{colored("V =", "black", attrs=["bold"])} \n{V}\n')
print(f'\n{colored("Run-time: ", "red", attrs=["bold"])}{run_time_coder_2: 0.5f}␣
 ↪(s)\n')
```

```
For Systematic Linear Block Code of C(5, 2) V Matrix that includes our

Codewords will be:

V =
[[0 0 0 0 0]
 [1 0 1 0 1]
 [1 1 1 1 0]
 [0 1 0 1 1]]


Run-time:  0.00016 (s)
```

Test by Using 99.73 % Sigma-Rule :

[12]:
```
def Run_time(Coder_func, Linear_Block_Code: tuple=(5, 2), times: int=10) -> np.
 ↪ndarray:
    run_time_list = []
    for i in range(times):
        tic = time()
        V = Coder_func(Linear_Block_Code)
        toc = time()
        run_time = toc - tic
        run_time_list.append(run_time)
    return np.array(run_time_list)
```

[13]:
```
Run_times_Coder_1 = Run_time(Coder_func=Coder_1, Linear_Block_Code=(5, 2),␣
 ↪times=1000000)
print(f'\n{colored("Run-time(s) for Coder 1 after 1,000,000 times test (mean +-␣
 ↪3std):", "blue", attrs=["bold"])} \
```

```
{Run_times_Coder_1.mean() * 1e5: 0.2f} e-5 -+ \
{3 * Run_times_Coder_1.std() * 1e5: 0.2f} e-5\n')
```

```
Run-time for Encoder 1 after 1,000,000 times test (μ ± 3σ):
1.15 e-5 ± 1.38 e-5
```

[14]:
```
Run_times_Coder_2 = Run_time(Coder_func=Coder_2, Linear_Block_Code=(5, 2),␣
↪times=1000000)
print(f'\n{colored("Run-time(s) for Coder 2 after 1,000,000 times test (mean +-␣
↪3std):", "blue", attrs=["bold"])} \
{Run_times_Coder_2.mean() * 1e5: 0.2f} e-5 -+ \
{3 * Run_times_Coder_2.std() * 1e5: 0.2f} e-5\n')
```

```
Run-time for Encoder 2 after 1,000,000 times test (μ ± 3σ):
0.77 e-5 ± 1.04 e-5
```

Conclusion:

As we saw Coder-2 is better for the Run-time parameter when n = 5 but for n > 5 we can't say anything before doing the test.

Decoder :

U Generation:

[15]:
```
U = U_generator(k=2)
print(f'\n{colored("U =", "black", attrs=["bold"])}\n\n{U}\n')
```

```
U =
```

```
[[0 0]
 [0 1]
 [1 0]
 [1 1]]
```

V Generation:

[16]:
```
Selected_Coder = Coder_2
V = Selected_Coder(Linear_Block_Code=(5, 2))
print(f'\n{colored("V =", "black", attrs=["bold"])}\n\n{V}\n')
```

```
V =


[[0 0 0 0 0]
 [1 0 1 0 1]
 [1 1 1 1 0]
 [0 1 0 1 1]]
```

Channel Effect (Error Patterns Generation Using Discrete Uniform Distribution):

[17]:
```
np.random.seed(4)
Error_patterns = np.random.randint(low=0, high=2, size=V.shape, dtype=np.int64)
print(f'\n{colored("Error_patterns =", "black",␣
↪attrs=["bold"])}\n\n{Error_patterns}\n')
```

```
Error_patterns =


[[0 0 1 1 1]
 [0 1 0 0 1]
 [0 0 1 1 0]
 [1 1 1 0 0]]
```

Received Vectors (R) Generation:

[18]:
```
def Channel_Out(Codewords: np.ndarray, E_patt: np.ndarray) -> np.ndarray:
    r = (Codewords + E_patt) % 2
    return r
```

[19]:
```
R = Channel_Out(Codewords=V, E_patt=Error_patterns)
print(f'\n{colored("R =", "black", attrs=["bold"])}\n\n{R}\n')
```

```
R =


[[0 0 1 1 1]
 [1 1 1 0 0]
 [1 1 0 0 0]
 [1 0 1 1 1]]
```

Optional:

[20]:
```
def H_generator(Linear_Block_Code: tuple) -> np.ndarray:
    n, k = Linear_Block_Code
    P = LOOK_UP_TABLE_P[Linear_Block_Code]['P']
    I_n_k = np.identity(n - k, dtype=np.int64)
    H = np.concatenate((I_n_k, P.T), axis=1)
    return H
```

Test:

[21]:
```
H = H_generator(Linear_Block_Code=(5, 2))
print(f'\n{colored("For Systematic Linear Block Code of C(5, 2) Parity-Check␣
 ↪Matrix will be:", "blue", attrs=["bold"])}\n\n\
{colored("H =", "black", attrs=["bold"])} \n{H}\n')
```

```
For Systematic Linear Block Code C(5, 2) Parity-Check Matrix is:

H =
```

```
[[1 0 0 1 1]
 [0 1 0 1 0]
 [0 0 1 1 1]]
```

[22]:
```
def S_generator(R: np.ndarray, H: np.ndarray) -> np.ndarray:
    S = (R @ H.T) % 2
    return S
```

[23]:
```
def S_generator(R: np.ndarray, H: np.ndarray) -> np.ndarray:
    S = (R @ H.T) % 2
    return S
```

Test:

[24]:
```
S = S_generator(R=R, H=H)
print(f'\n{colored("S =", "black", attrs=["bold"])}\n\n{S}\n')
```

```
S =

[[0 1 1]
 [1 1 1]
 [1 1 0]
 [1 1 1]]
```

Decoder:

[25]:
```
def Decoder(R: np.ndarray) -> np.ndarray:
    check_dict = {}
    _, n = R.shape
    Error_patt_hat_list_list = []
    for i, r in enumerate(R):
        for key in St_Arr.keys():
            if list(r) not in St_Arr[key]:
                check_dict['r' + str(i)] = n * [0]
            else:
```

```
                check_dict['r' + str(i)] = St_Arr[key][0]
                break
    for ke in check_dict.keys():
        Error_patt_hat_list_list.append(check_dict[ke])

    Error_patt_hat_ndarray = np.array(Error_patt_hat_list_list, dtype=np.int64)
    V_hat = (R + Error_patt_hat_ndarray) % 2
    return V_hat, Error_patt_hat_ndarray
```

For Random Error Pattern Matrix:

[26]:
```
V_hat, e = Decoder(R=R)
print(f'\nWhen {colored("Error Patterns", "blue", attrs=["bold"])} (Channel␣
 ↪Effect) is: \n{Error_patterns}\n\nand \
{colored("V", "blue", attrs=["bold"])} is: \n{V}\n\nthen \n\nthen \
{colored("R", "blue", attrs=["bold"])} will be: \n{R}\n\n\
and {colored("V_hat", "blue", attrs=["bold"])} will be: \n{V_hat}\n\n')
```

```
When Error Patterns (Channel Effect) is:
[[0 0 1 1 1]
 [0 1 0 0 1]
 [0 0 1 1 0]
 [1 1 1 0 0]]

and V is:
[[0 0 0 0 0]
 [1 0 1 0 1]
 [1 1 1 1 0]
 [0 1 0 1 1]]

then

then R will be:
[[0 0 1 1 1]
 [1 1 1 0 0]
 [1 1 0 0 0]
 [1 0 1 1 1]]
```

and V_hat will be:

```
[[0 1 0 1 1]
 [1 1 1 1 0]
 [1 1 1 1 0]
 [1 0 1 0 1]]
```

For Desired Error Pattern Matrix:

[27]:
```
E = np.array([[0, 0, 0, 0, 1], [0, 1, 0, 0, 0], [0, 0, 1, 1, 0], [0, 1, 1, 0,␣
↪0]], dtype=np.int64)
print(f'\n{colored("Desired Error Patterns:", "blue", attrs=["bold"])}␣
↪\n\n{colored("E = ", "black", attrs=["bold"])}\n{E}\n')
```

```
Desired Error Patterns:

E =
[[0 0 0 0 1]
 [0 1 0 0 0]
 [0 0 1 1 0]
 [0 1 1 0 0]]
```

[28]:
```
R2 = Channel_Out(Codewords=V, E_patt=E)
print(f'\n{colored("For Desired Error Patterns:", "blue", attrs=["bold"])}␣
↪\n\n{colored("R = ", "black", attrs=["bold"])}\n{R2}\n')
```

```
For Desired Error Patterns:

R =
[[0 0 0 0 1]
 [1 1 1 0 1]
 [1 1 0 0 0]
 [0 0 1 1 1]]
```

[29]:
```
print(f'\n{colored("V = ", "black", attrs=["bold"])}\n{V}\n')
```

```
V =

[[0 0 0 0 0]
 [1 0 1 0 1]
 [1 1 1 1 0]
 [0 1 0 1 1]]
```

[30]:
```
V_hat_2, E_hat = Decoder(R=R2)
print(f'\n{colored("For Desired Error Patterns:", "blue", attrs=["bold"])}
 ↪\n\n{colored("V-hat = ", "black", attrs=["bold"])}\n{V_hat_2}\n')
```

```
For Desired Error Patterns:

V-hat =

[[0 0 0 0 0]
 [1 0 1 0 1]
 [1 1 1 1 0]
 [0 1 0 1 1]]
```

Conclusion:

As we saw for error patterns that there are in the Standard Array, the Decoder does decoding as correctly.

References:

- **Book** : Shu Lin, Daniel J. Costello - Error Control Coding. 2nd Edition-Prentice Hall, 2004.