Analysis of the Viterbi Algorithm and IRIDIUM satellite network

Johnathan McDougal

Spring 2022

1 Background

The Viterbi algorithm is a method of convolutional signal encoding and decoding. Proposed by Andrew Viterbi in 1967 and first implemented in 1987 It is widely (and mostly) used for digital communications systems from WiFi to deep space. However, the Viterbi algorithm is also used for language processing, chiefly speech tagging.

The IRIDIUM satellite network is a global telecommunication service for civilian and federal use. IRIDIUM was conceptualized by a team within Motorola and received component-wise collaboration from Lockheed-Martin, Raytheon, and Scientific Atlanta. The network was launched in 1996 and fully completed (and operational) in 1998. IRIDIUM is also the inspiration for this project as it utilizes the Viterbi algorithm, which is why it is being described here.

1.1 Viterbi Algorithm

The Viterbi Algorithm is a popular signal coding strategy that utilizes shift registering, hamming, and Hidden Markov methods for pattern detection and isolation.

1.1.1 Encoding Path

The Viterbi Algorithm is capable of encoding and decoding, as such, the coding process must begin with the encoding path. During this half of the process, the binary elements will undergo BPSK, shift registering, codeword branching, and finally parity coding to construct what will be considered the generated or ”received” codeword.

1.1.2 Shift register and codeword Branching

m: input data sequence

U: Transmitted code word

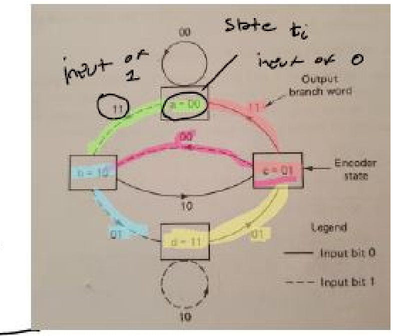
z: Received sequence

Convolutional encoding For K = 3

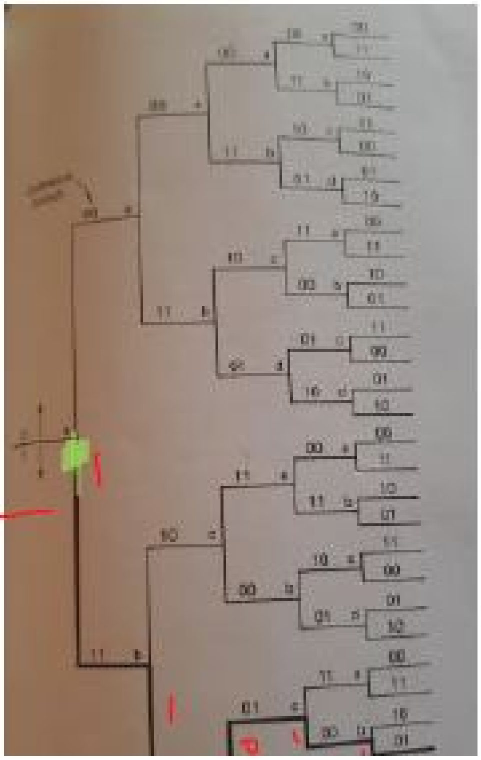
Given m = 1 1 0 1 1

↓

input bit	Register (Flushed by K-zeros)	State at time t_i	State at time t_{i+1}	Branch word at t_i	
				U_1	U_2
null	0 0 0	0 0	0 0	null	null
1	1 0 0	0 0	1 0	1	1
1	1 1 0	1 0	1 1	0	1
0	0 1 1	1 1	0 1	0	1
1	1 0 1	0 1	1 0	0	0
1	1 1 0	1 0	1 1	0	1
0	0 1 1	1 1	0 1	0	1
0	0 0 1	0 1	0 0	1	1



Produced U = 11 01 01 00 01 01 11



Path defined by m

These first two processes are rather simple, so they will be grouped together. Shift registering is a form of convolution in which the original binary string is convolved by a null array of length K. Codeword branching refers to the encoding path a bit of information takes, defined by input and derived register. The above figures demonstrate both shift registering and the codeword branch. The viterbi process is mostly defined by a constraint length of K. In Figure 1, one can observe that the constraint length used is 3 (nominal range is 2 to 6). Because the constraint length is 3, the binary sequence must be flushed with 3 null values (zeroes). That register then processes the original data, i.e., starts taking in the non null values, e.g: the next register is 001, with that 1 coming from the first part of the original string. Each register and input then determines the state machine and branch word "u" at each time interval. After obtaining all of these attributes, a "full" codeword is generated. This is not truly the full codeword, one more value must be added before the real "transmitted" codeword is produced (and then transmitted).

1.1.3 Parity Code

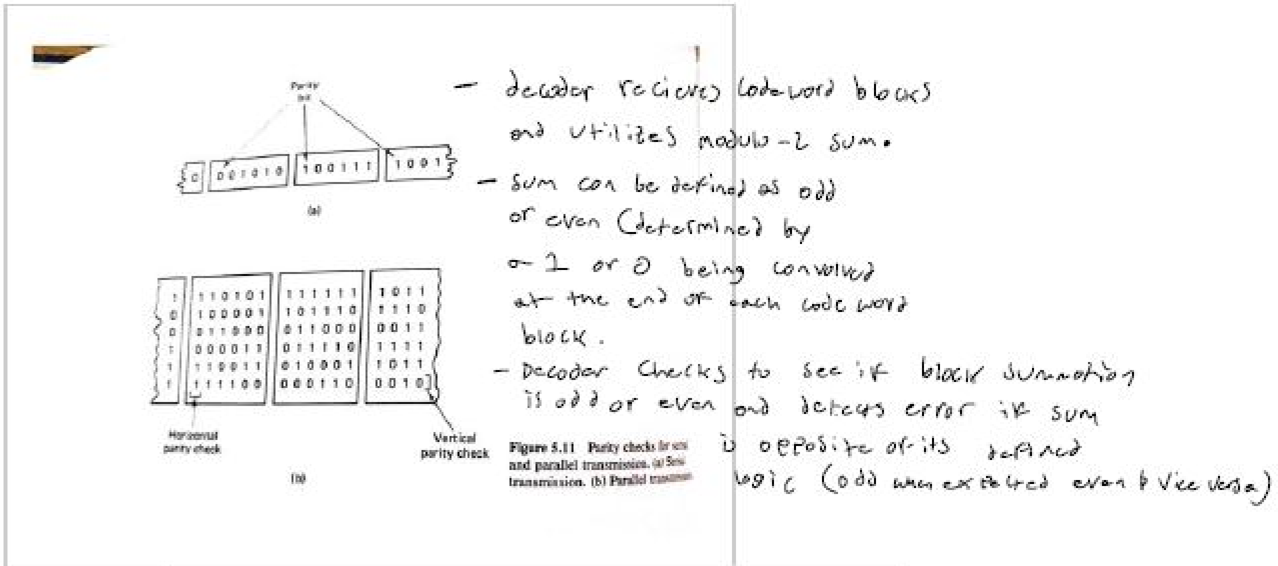
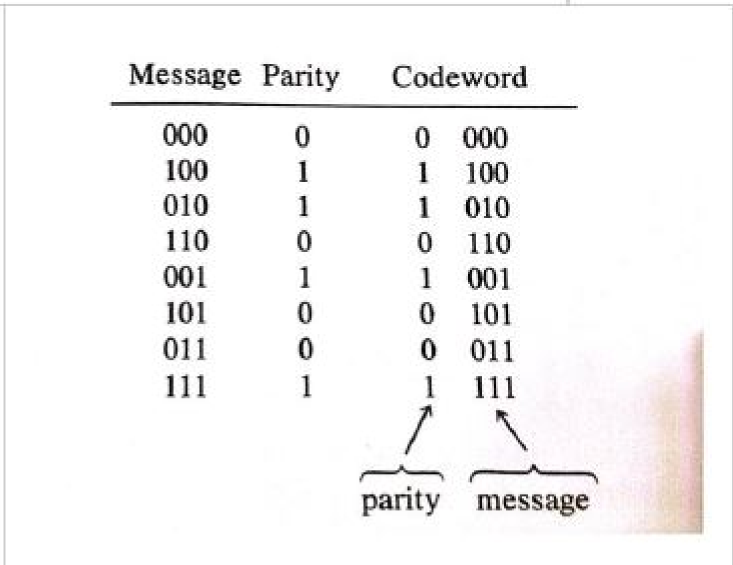


Figure 5.11 Parity checks for seri and parallel transmission. (a) Seri transmission. (b) Parallel transmi

- decoder recieves codeword blocks and utilizes modulo-2 sum.
- sum can be defined as odd or even (determined by a 1 or 0 being convolved at the end of each code word block.
- Decoder checks to see if block summation is odd or even and detects error if sum is opposite of its defined logic (odd when expected even & vice versa)



Message	Parity	Codeword
000	0	0 000
100	1	1 100
010	1	1 010
110	0	0 110
001	1	1 001
101	0	0 101
011	0	0 011
111	1	1 111

parity
message

After the branch codeword is generated, a method of error detection is implemented in the form of parity coding. The system evaluates the sum of the entire codeword and concatenates either a lagging (left hand side) 1 or 0 to the string. This all depends on a predetermined choice of odd or even parity on the decoder's end. depending on what parity the system desires, a parity bit of 1 or 0 is added to change the sum of a string to an odd or even value.

1.2 Decoding Path

The decoding path is far more complex. This is due to the necessity of including error detection and correction. As previously mentioned, the detection process is somewhat simplified through parity coding. Although it is possible for the parity of a received codeword to match that of the transmitted one (and the desired parity of the system), temporarily eliminating the need to detect and correct each sequence reduces the computational cost of the entire communications system. After evaluating parity, the decoder will then perform syndrome testing on any discovered error (wrong parity) to determine the pattern and potential repetition of error. After finding the pattern, the decoder will then perform some form of linear regression to reduce or correct the error. Lastly, the decoder will then test the branch metrics of each potential sequence until the one with the least difference between the expected codeword is selected. Until a difference of 0 (or perhaps a specified threshold) is achieved, this entire process will repeat with disregard for parity. Performance and hardware complexity of the Viterbi algorithm is mainly dependent on constraint length. Larger constraint length requires more memory but provides a lower Bit Error rate (BER) without increasing the signal power. If z is the received sequence, xm is one of the possible transmitted sequences and all input sequences are equally likely, comparison of conditional probabilities to achieve the minimum probability is called the likelihood function. Convolutional coding is represented by mainly three parameters: k, n and K. The k indicates input data bits entering into encoder, n indicates coded bits from encoder and K is called constraint length and (K-1) are shift register stages or encoder memory size, while the code rate is k/n

1.2.1 Error Detection

Types of Error Control:

(pre error detection/correction)

Automatic Repeat Request (ARQ)/Automatic retransmission Query (ARQ)

- For error detection only
- system detects an error and requests a retransmission

Procedures:

1.) Stop and wait ARQ

- Transmitter waits for Acknowlegement (ACK)
- Reciever respons with Negative Acknowledgement (NAK) if error is detected; other wise continues with ACK.

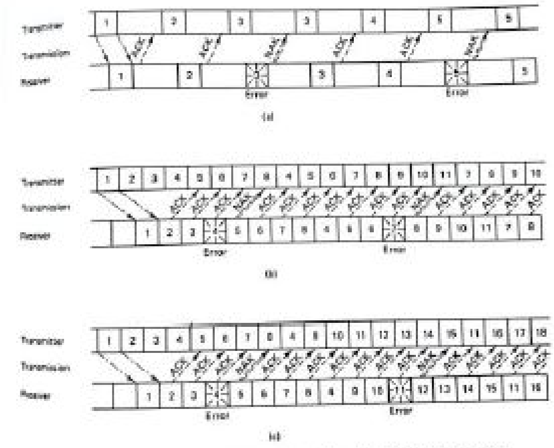
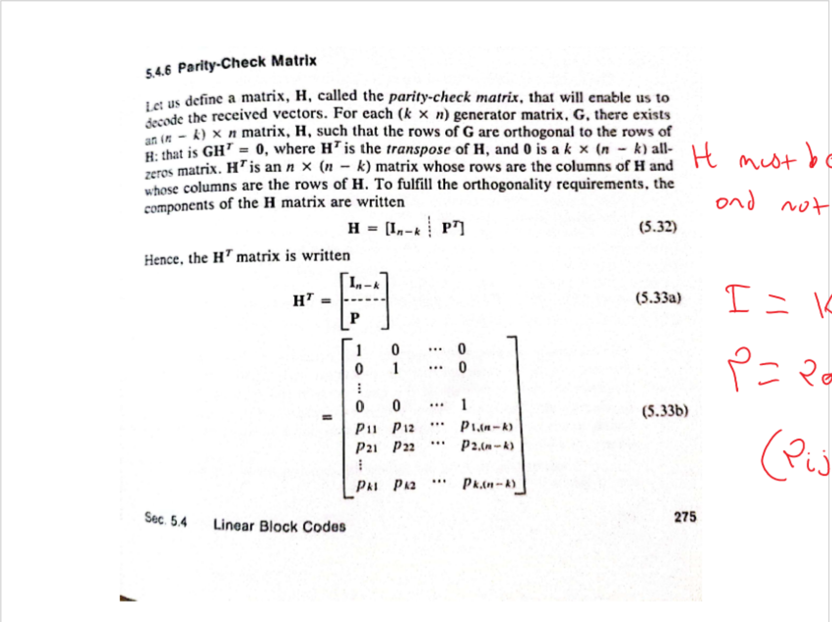


Figure 5.10 Automatic repeat request (ARQ). (a) Stop-and-wait ARQ (half-duplex). (b) Continuous ARQ with pullback (full-duplex). (c) Continuous ARQ with selective repeat (full-duplex).



5.4.6 Parity-Check Matrix

Let us define a matrix, **H**, called the *parity-check matrix*, that will enable us to decode the received vectors. For each $(k \times n)$ generator matrix, **G**, there exists an $(n-k) \times n$ matrix, **H**, such that the rows of **G** are orthogonal to the rows of **H**; that is $\mathbf{GH}^T = \mathbf{0}$, where $\mathbf{H}^T$ is the *transpose* of **H**, and **0** is a $k \times (n-k)$ all-zeros matrix. $\mathbf{H}^T$ is an $n \times (n-k)$ matrix whose rows are the columns of **H** and whose columns are the rows of **H**. To fulfill the orthogonality requirements, the components of the **H** matrix are written

$$\mathbf{H} = [\mathbf{I}_{n-k} \,\vdots\, \mathbf{P}^T] \tag{5.32}$$

Hence, the $\mathbf{H}^T$ matrix is written

$$\mathbf{H}^T = \begin{bmatrix} \mathbf{I}_{n-k} \\ \hdashline \mathbf{P} \end{bmatrix} \tag{5.33a}$$

$$= \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & & & \\ 0 & 0 & \cdots & 1 \\ p_{11} & p_{12} & \cdots & p_{1,(n-k)} \\ p_{21} & p_{22} & \cdots & p_{2,(n-k)} \\ \vdots & & & \\ p_{k1} & p_{k2} & \cdots & p_{k,(n-k)} \end{bmatrix} \tag{5.33b}$$

Sec. 5.4 Linear Block Codes 275

H must be linearly independent and not have sparse columns

$I = k \times k$ identity matrix

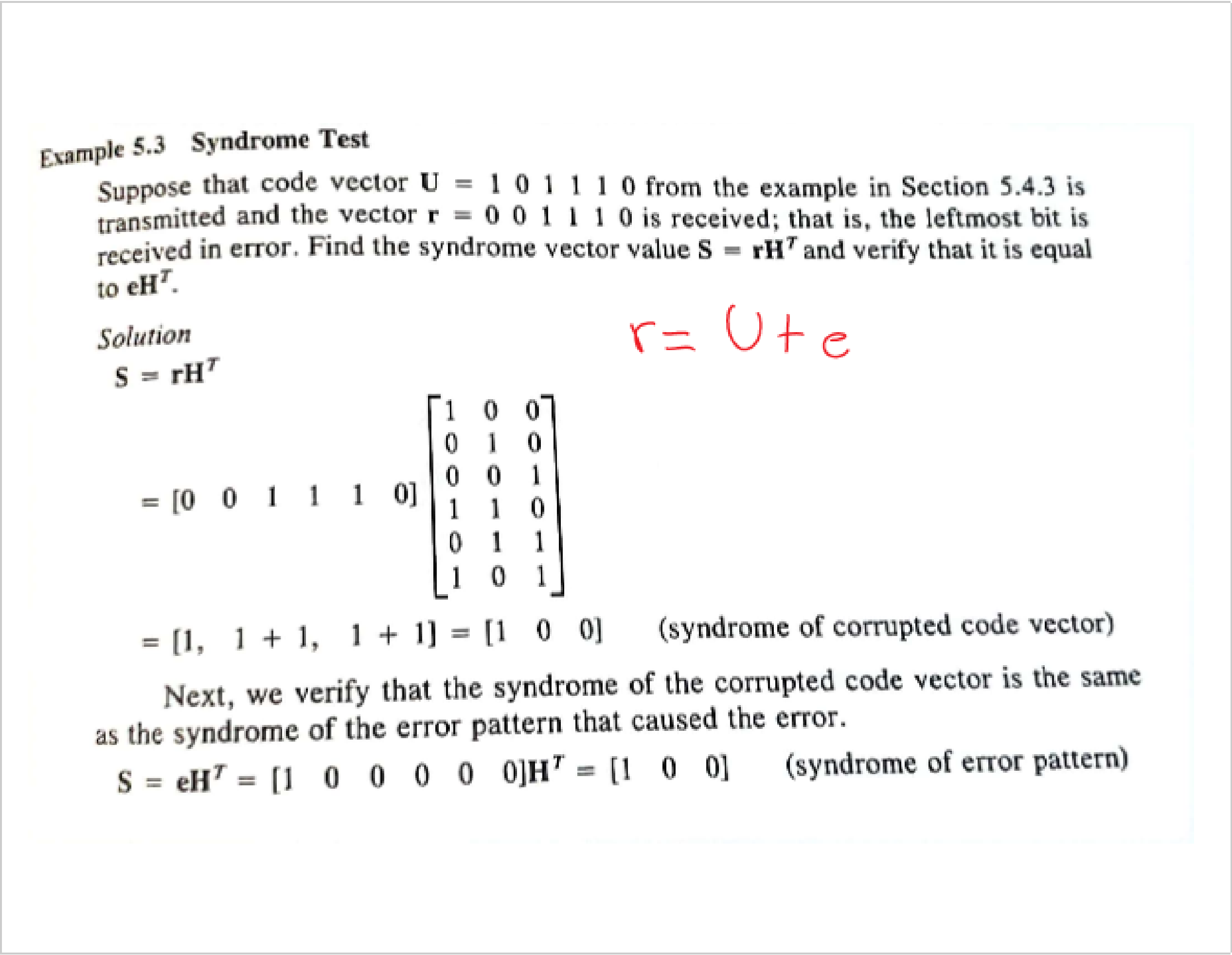
$P =$ parity array

$(P_{ij} = 0 \text{ or } 1)$

On the first attempt at decoding, the system will utilize the aforementioned parity method. The purpose of this is to reduce computing time and cost. After that, the decoder then searches for a likely pattern in which error is being convolved into the system. If the decoder detects multiple syndromes, or irregular points of error (there is error, but no real pattern), the system must now re-evaluate the received information without relying on the parity code. Although this still accomplishes the goal of decoding a transmitted data stream, it is no longer reducing the computational budget or computation time.

1.3 Error Correction

1.3.1 Syndrome Testing



Example 5.3 **Syndrome Test**

Suppose that code vector $\mathbf{U}$ = 1 0 1 1 1 0 from the example in Section 5.4.3 is transmitted and the vector $\mathbf{r}$ = 0 0 1 1 1 0 is received; that is, the leftmost bit is received in error. Find the syndrome vector value $\mathbf{S} = \mathbf{r}\mathbf{H}^T$ and verify that it is equal to $\mathbf{e}\mathbf{H}^T$.

Solution

$r = U + e$

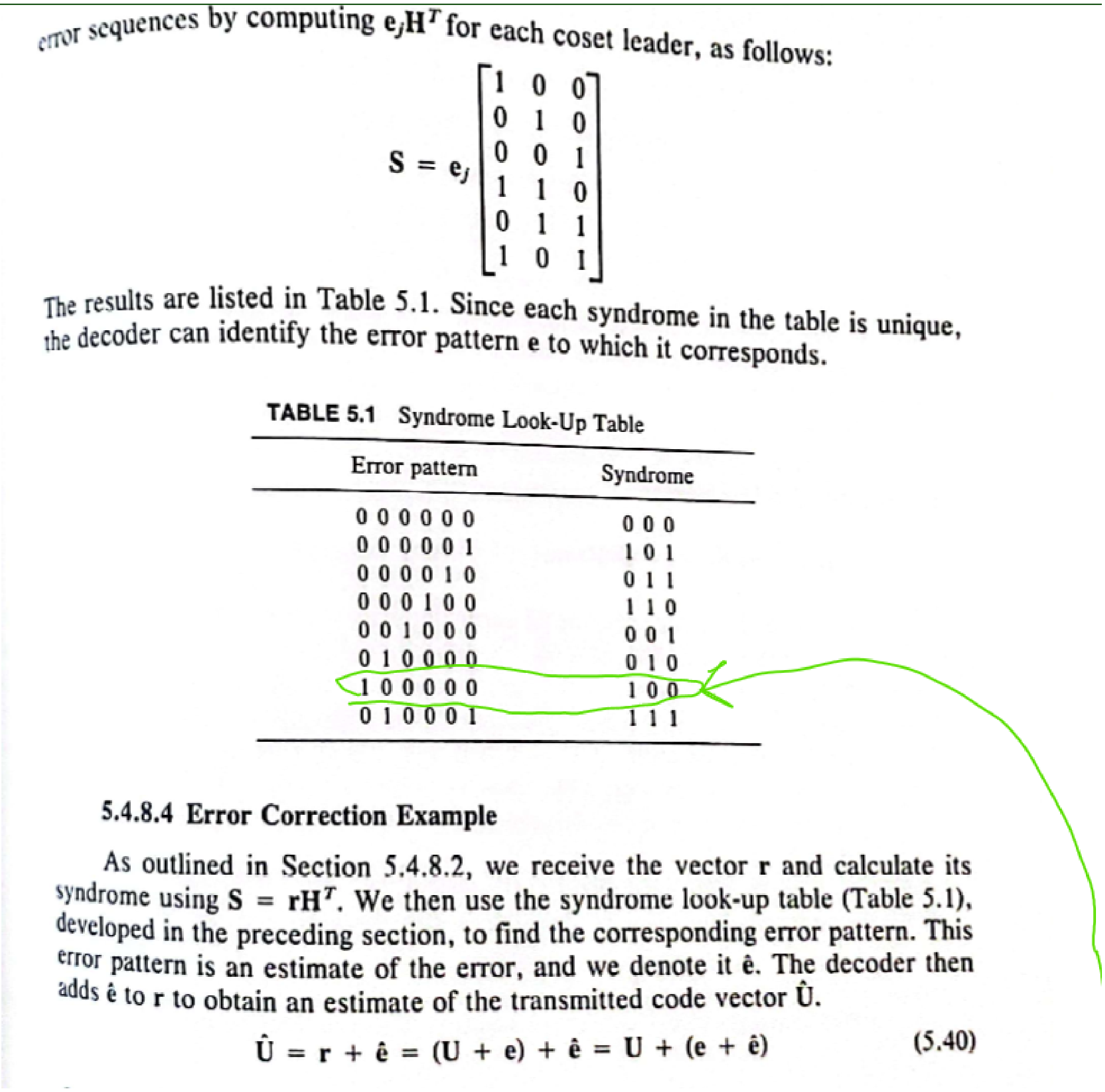
$$\mathbf{S} = \mathbf{r}\mathbf{H}^T$$

$$= [0 \quad 0 \quad 1 \quad 1 \quad 1 \quad 0]\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \end{bmatrix}$$

$$= [1, \quad 1+1, \quad 1+1] = [1 \quad 0 \quad 0] \qquad \text{(syndrome of corrupted code vector)}$$

Next, we verify that the syndrome of the corrupted code vector is the same as the syndrome of the error pattern that caused the error.

$$\mathbf{S} = \mathbf{e}\mathbf{H}^T = [1 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0]\mathbf{H}^T = [1 \quad 0 \quad 0] \qquad \text{(syndrome of error pattern)}$$



error sequences by computing $\mathbf{e}_j\mathbf{H}^T$ for each coset leader, as follows:

$$\mathbf{S} = \mathbf{e}_j \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \end{bmatrix}$$

The results are listed in Table 5.1. Since each syndrome in the table is unique, the decoder can identify the error pattern **e** to which it corresponds.

TABLE 5.1 Syndrome Look-Up Table

Error pattern	Syndrome
0 0 0 0 0 0	0 0 0
0 0 0 0 0 1	1 0 1
0 0 0 0 1 0	0 1 1
0 0 0 1 0 0	1 1 0
0 0 1 0 0 0	0 0 1
0 1 0 0 0 0	0 1 0
1 0 0 0 0 0	1 0 0
0 1 0 0 0 1	1 1 1

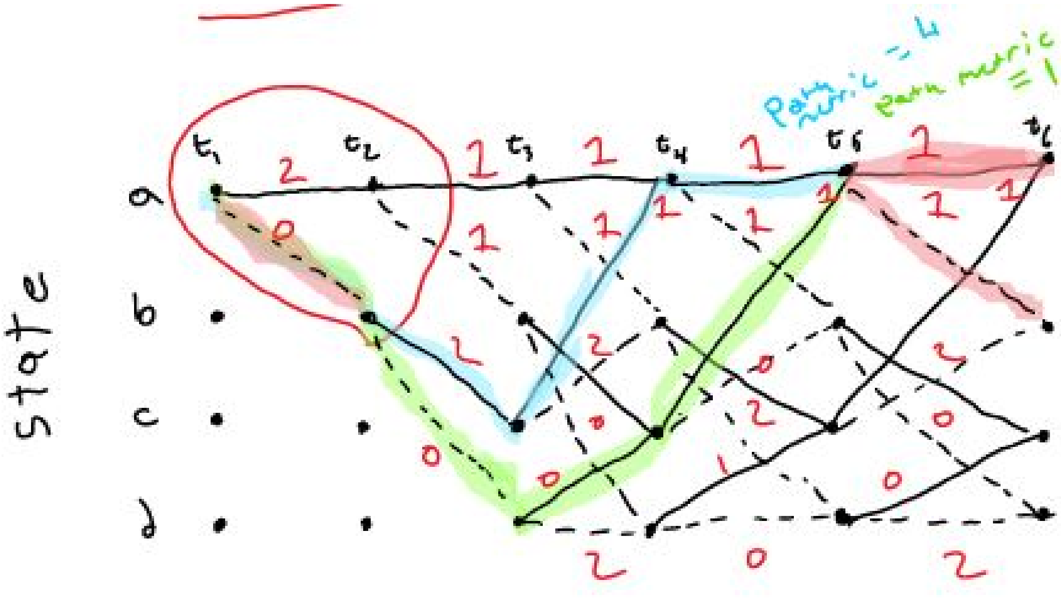
5.4.8.4 Error Correction Example

As outlined in Section 5.4.8.2, we receive the vector **r** and calculate its syndrome using $\mathbf{S} = \mathbf{r}\mathbf{H}^T$. We then use the syndrome look-up table (Table 5.1), developed in the preceding section, to find the corresponding error pattern. This error pattern is an estimate of the error, and we denote it $\hat{\mathbf{e}}$. The decoder then adds $\hat{\mathbf{e}}$ to **r** to obtain an estimate of the transmitted code vector $\hat{\mathbf{U}}$.

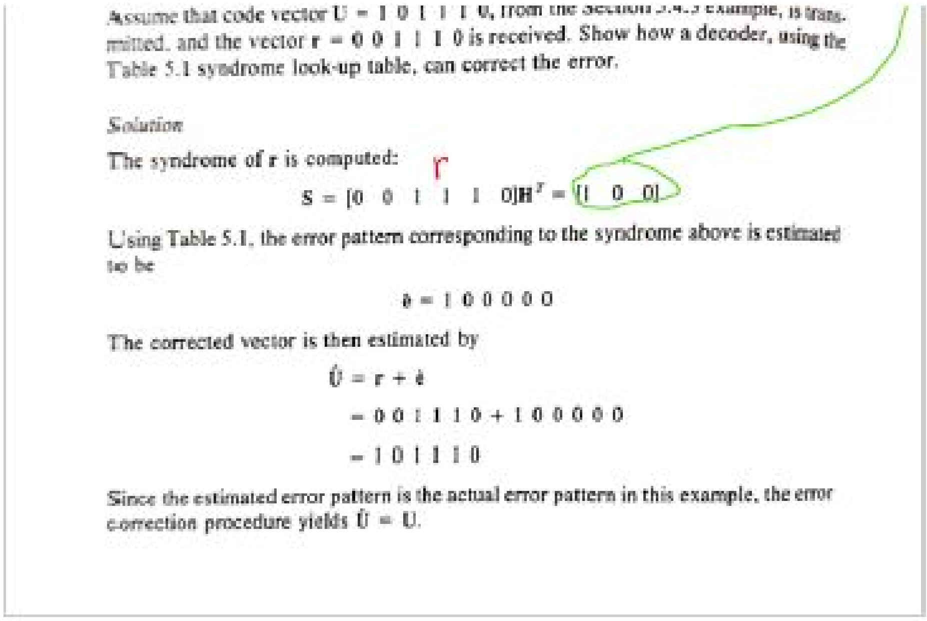
$$\hat{\mathbf{U}} = \mathbf{r} + \hat{\mathbf{e}} = (\mathbf{U} + \mathbf{e}) + \hat{\mathbf{e}} = \mathbf{U} + (\mathbf{e} + \hat{\mathbf{e}}) \tag{5.40}$$

This is where the coding process becomes very complicated. As shown in the above diagram, extensive linear algebra and regression starts being implemented. In order to even begin syndrome tests, the decoder requires the codeword matrix (U, composed of the codeword vector, which is every codeword transmitted), the error matrix (e, composed of every error vector associated with every codeword vector), and the parity matrix (H, which is every generated by every parity check). Of course these aren't difficult to understand conceptually, but for nominal communications processes, these end up being extremely large arrays manipulating each other, thus very computationally expensive, and the chance for error is effectively greater. The advantage of Syndrome Testing is that it just utilizes linear algebra, rather than more taxing calculus differential equation methods. It's also dependent on the parity check matrix, thus it is dependent mostly on the constraint length K. So, the user can decide the computation complexity (reducing matrix size) at the cost of the number of computations. Although this doesn't necessarily reduce the data budget nor improve error rate, it does allow for faster processing and potentially sooner error detection (which would reduce data cost).

1.3.2 Branch Metrics and Hamming Distance



After the first, less expensive, syndrome tests, the decoder then evaluates the accepted codewords. These are then mapped to a trellis that determines the overall hamming difference between the received codeword and the expected codeword. The hamming distance is calculated simply by summing the bit difference between received and expected codewords.



Assume that code vector U = 1 0 1 1 1 0, from the Section 5.4.3 example, is transmitted, and the vector **r** = 0 0 1 1 1 0 is received. Show how a decoder, using the Table 5.1 syndrome look-up table, can correct the error.

Solution

The syndrome of **r** is computed:

r

$$\mathbf{S} = [0 \quad 0 \quad 1 \quad 1 \quad 1 \quad 0]\mathbf{H}^T = [1 \quad 0 \quad 0]$$

Using Table 5.1, the error pattern corresponding to the syndrome above is estimated to be

$$\hat{e} = 1\ 0\ 0\ 0\ 0\ 0$$

The corrected vector is then estimated by

$$\begin{aligned}\hat{U} &= \mathbf{r} + \hat{e}\\ &= 0\ 0\ 1\ 1\ 1\ 0 + 1\ 0\ 0\ 0\ 0\ 0\\ &= 1\ 0\ 1\ 1\ 1\ 0\end{aligned}$$

Since the estimated error pattern is the actual error pattern in this example, the error correction procedure yields $\hat{U} = U$.

U= code vector (codeword)

V= vector space

weight of vectors = # of non zero value)

Ex:

U= 1 0 0 1 0 1 1 0 1 } w(U) = 5

V= 0 1 1 1 1 0 1 0 0 } w(V) = 5

w(U,V)= 1 1 1 0 1 1 0 1 1 = 6

Hamming distance = d(U,V) = 6

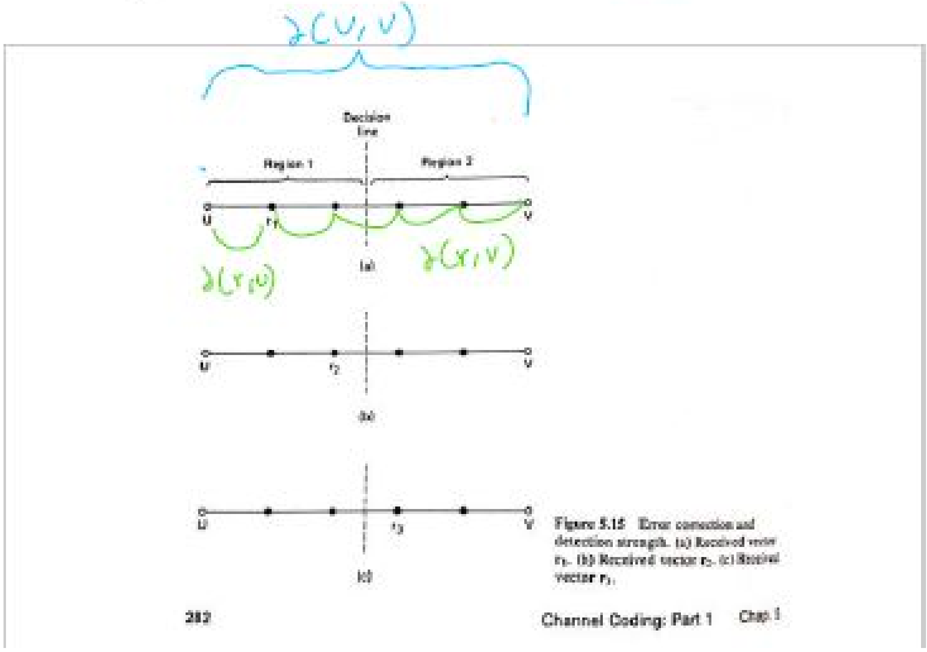
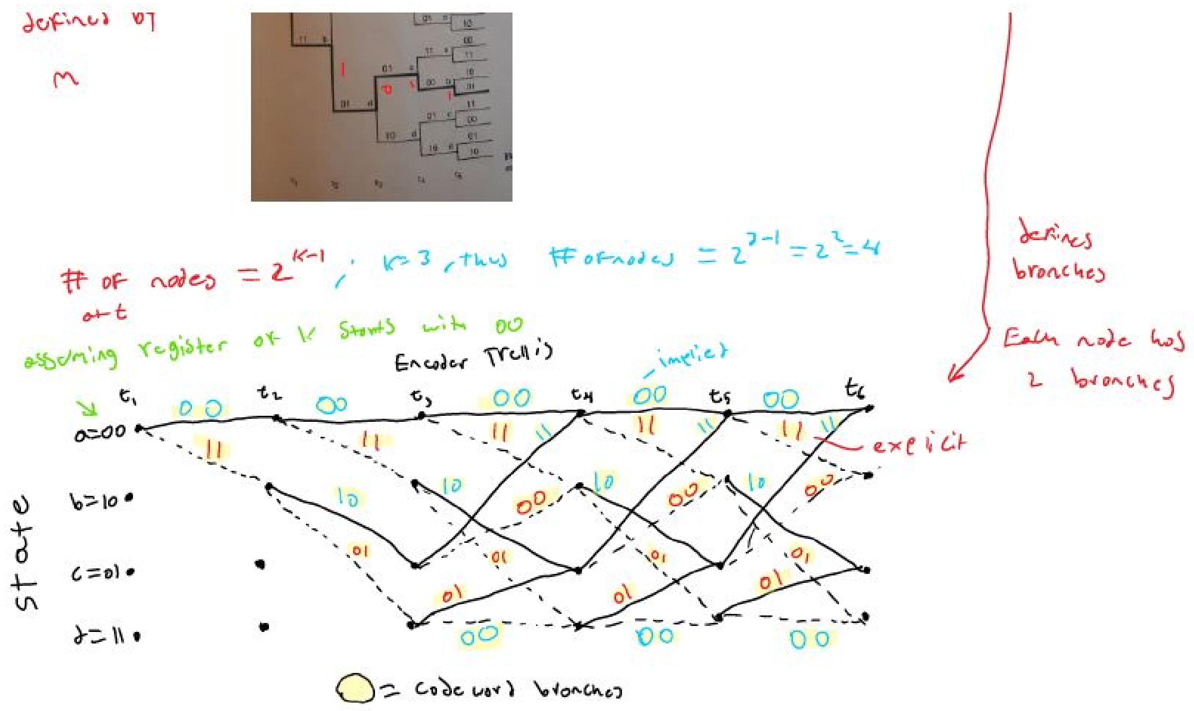


Figure 5.15 Error correction and detection strength. (a) Received vector r_1. (b) Received vector r_2. (c) Received vector r_3.

282 Channel Coding: Part 1 Chap. 5

The system then compares the weight (sum of non zero values) of the vector U (codeword) and the vector space V (a random anticipated error vector). Using these extracted parameters, the decoder then produces a decision branch.



Decoder Trellis

input m: 1 1 0 1 1

Transmit U: 11 01 01 00 01

received Z: 11 01 01 10 01

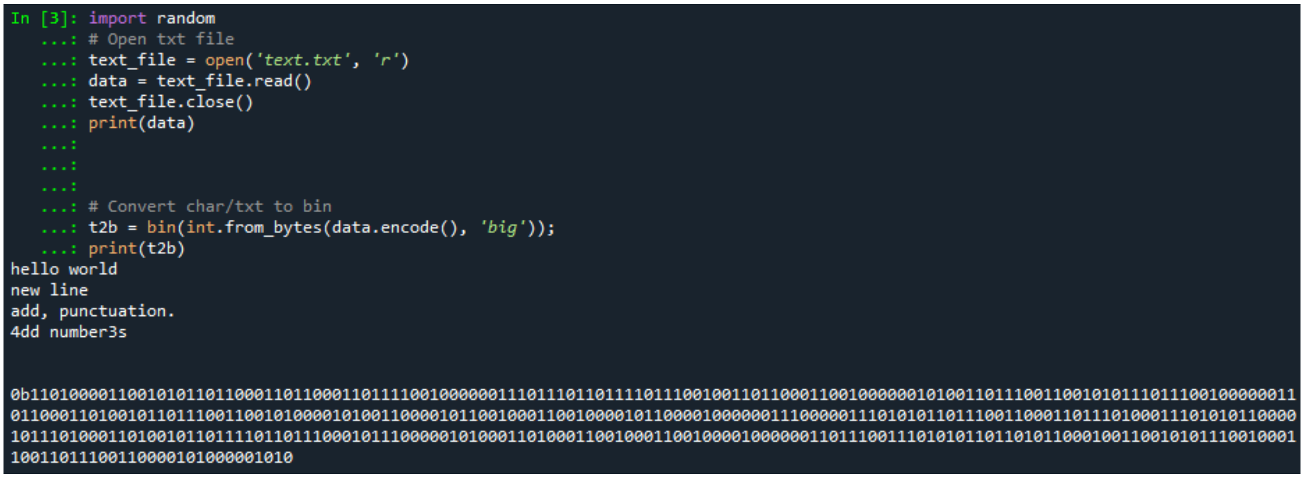
based on Encoder trellis

The length, or number of options, is defined by the hamming distance and the decision line is shifted by the vector weights. Above, there is a state machine with U and V vectors of equal weight (5), so the decision line is placed exactly between them. If U had a higher weight than V, the decision line would shift to the right, towards V. Vector weight essentially determines the probability of the r vector being categorized as a U or V vector. Ideally, the U vector would have a greater weight as this means the accepted codeword is more likely to not contain error.

2 Methods: Python Simulation

2.1 ASCII to Binary

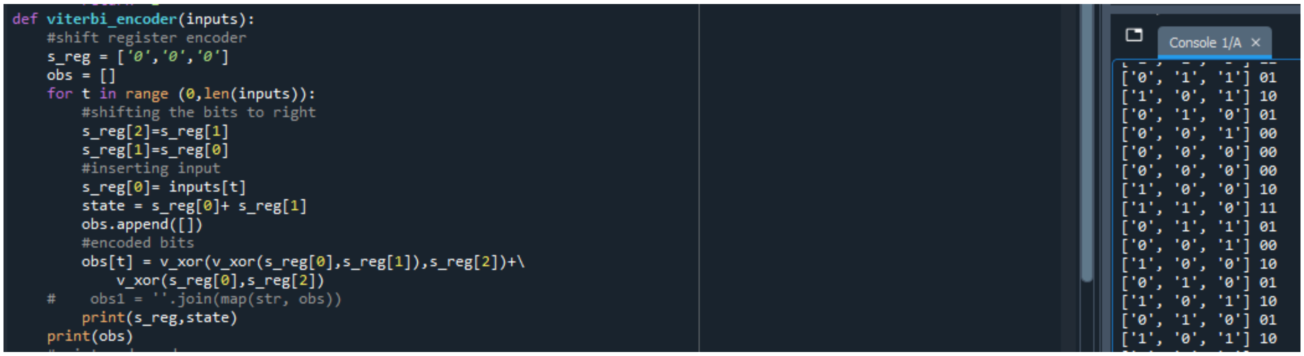
This process is very simple and really has no Viterbi relevance. The purpose is to use Viterbi in a realistic scenario. In order to use viterbi, the data being processed must be binary, not ASCII, so a simple ASCII to binary conversion is necessary



```
In [3]: import random
   ...: # Open txt file
   ...: text_file = open('text.txt', 'r')
   ...: data = text_file.read()
   ...: text_file.close()
   ...: print(data)
   ...:
   ...:
   ...:
   ...: # Convert char/txt to bin
   ...: t2b = bin(int.from_bytes(data.encode(), 'big'));
   ...: print(t2b)
hello world
new line
add, punctuation.
4dd number3s


0b1101000011001010110110001101100011011110010000001110111011011110111001001101100011001000000101001101110011001010111011100100000011
0110001101001011011100110010100001010011000010110010001100100001011000010000001110000011101010110111001100011011101000111010101100000
1011101000110100101101111011011100010111000001010001101000110010001100100001000000110111001110101011011010110001001100101011100100001
10011011100110000101000001010
```

2.2 Shift Register

Here, the Viterbi process truly begins. The raw, original data is produced from the ASCII to binary operation as the value ”inputs” of the function ”viterbi encoder.” As explained earlier, the first step in the Viterbi process is to establish a shift register by flushing the data with a null vector. In terms of this python scenario, the original string is simply concatenated with a null string of length K.

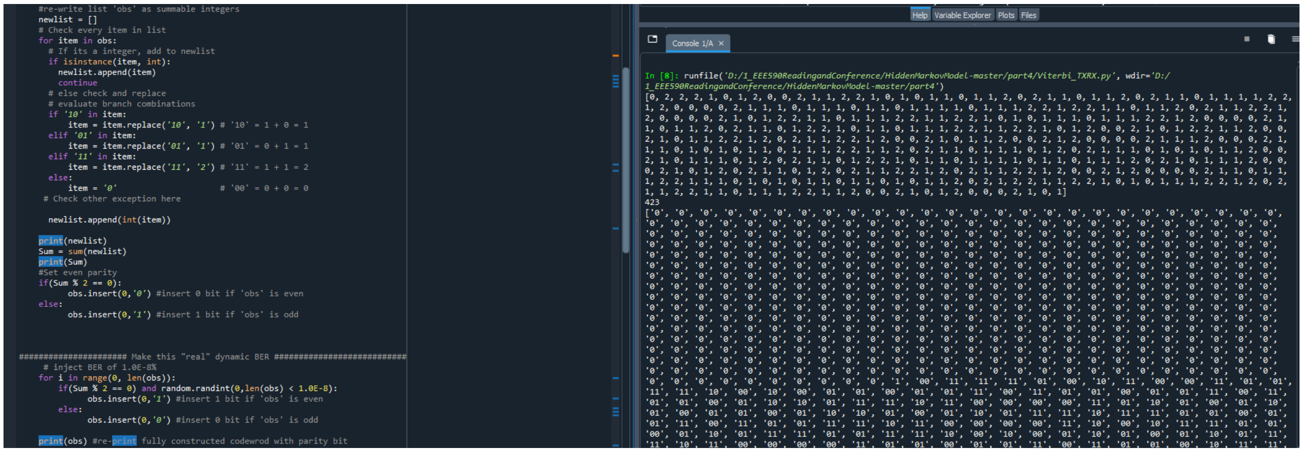


```
def viterbi_encoder(inputs):
    #shift register encoder
    s_reg = ['0','0','0']
    obs = []
    for t in range (0,len(inputs)):
        #shifting the bits to right
        s_reg[2]=s_reg[1]
        s_reg[1]=s_reg[0]
        #inserting input
        s_reg[0]= inputs[t]
        state = s_reg[0]+ s_reg[1]
        obs.append([])
        #encoded bits
        obs[t] = v_xor(v_xor(s_reg[0],s_reg[1]),s_reg[2])+\
            v_xor(s_reg[0],s_reg[2])
    #    obs1 = ''.join(map(str, obs))
        print(s_reg,state)
    print(obs)
```

```
['0', '1', '1'] 01
['1', '0', '1'] 10
['0', '1', '0'] 01
['0', '0', '1'] 00
['0', '0', '0'] 00
['0', '0', '0'] 00
['1', '0', '0'] 10
['1', '1', '0'] 11
['0', '1', '1'] 01
['0', '0', '1'] 00
['1', '0', '0'] 10
['0', '1', '0'] 01
['1', '0', '1'] 10
['0', '1', '0'] 01
['1', '0', '1'] 10
```

2.3 Parity

After shifting the register and generating the partial codeword, the parity bit is finally added on. In order to do this, the string is again recast to a list so that more information may be added. The proper parity bit is added exactly as explained in the background section; the encoder simply sums each codeword vector and places the correct bit to have even parity. The caveat here is that error must be simulated, so the efficacy of the encoder/decoder may be evaluated. In this case, the BER has been specified to 1E-8, a nominal BER. The system can experience higher BER (up to 1E-2), but that’s much higher than a usual case and the impact is negligible.

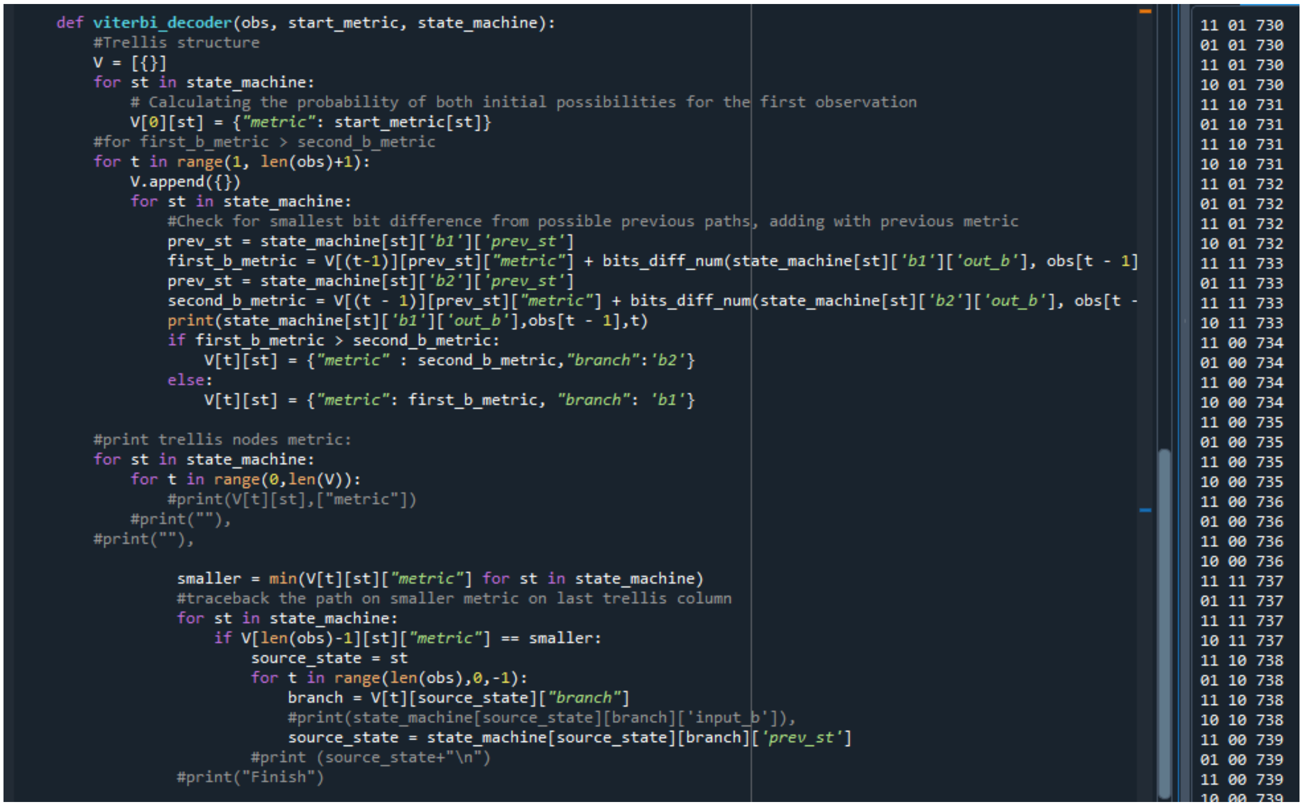


```
    #re-write list 'obs' as summable integers
    newlist = []
    # Check every item in list
    for item in obs:
      # If its a integer, add to newlist
      if isinstance(item, int):
        newlist.append(item)
        continue
      # else check and replace
      # evaluate branch combinations
      if '10' in item:
          item = item.replace('10', '1') # '10' = 1 + 0 = 1
      elif '01' in item:
          item = item.replace('01', '1') # '01' = 0 + 1 = 1
      elif '11' in item:
          item = item.replace('11', '2') # '11' = 1 + 1 = 2
      else:
          item = '0'                     # '00' = 0 + 0 = 0
     # Check other exception here

      newlist.append(int(item))

    print(newlist)
    Sum = sum(newlist)
    print(Sum)
    #Set even parity
    if(Sum % 2 == 0):
          obs.insert(0,'0') #insert 0 bit if 'obs' is even
    else:
          obs.insert(0,'1') #insert 1 bit if 'obs' is odd



###################### Make this "real" dynamic BER ##########################
     # inject BER of 1.0E-8%
    for i in range(0, len(obs)):
        if(Sum % 2 == 0) and random.randint(0,len(obs) < 1.0E-8):
              obs.insert(0,'1') #insert 1 bit if 'obs' is even
        else:
              obs.insert(0,'0') #insert 0 bit if 'obs' is odd

    print(obs) #re-print fully constructed codewrod with parity bit
```

```
In [8]: runfile('D:/1_EEE590ReadingandConference/HiddenMarkovModel-master/part4/Viterbi_TXRX.py', wdir='D:/
1_EEE590ReadingandConference/HiddenMarkovModel-master/part4')
[0, 2, 2, 2, 1, 0, 1, 2, 0, 0, 2, 1, 1, 2, 2, 1, 0, 1, 0, 1, 1, 0, 1, 1, 2, 0, 2, 1, 1, 0, 1, 1, 2, 0, 2, 1, 1, 0, 1, 1, 1, 1, 2, 2,
1, 2, 0, 0, 0, 0, 2, 1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 2, 2, 1, 2, 2, 1, 1, 0, 1, 1, 2, 0, 2, 1, 1, 2, 2, 1,
2, 0, 0, 0, 0, 2, 1, 0, 1, 2, 2, 1, 1, 0, 1, 1, 1, 2, 2, 1, 1, 2, 2, 1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 2, 2, 1, 2, 0, 0, 0, 0, 2, 1,
1, 0, 1, 1, 2, 0, 2, 1, 1, 0, 1, 2, 2, 1, 0, 1, 1, 0, 1, 1, 1, 2, 2, 1, 1, 2, 2, 1, 0, 1, 2, 0, 0, 2, 1, 0, 1, 2, 2, 1, 1, 2, 0, 0,
2, 1, 0, 1, 1, 2, 2, 1, 2, 0, 2, 1, 1, 2, 2, 1, 2, 0, 0, 2, 1, 0, 1, 1, 2, 0, 0, 2, 1, 2, 0, 0, 0, 0, 2, 1, 1, 1, 2, 0, 0, 0, 2, 1,
1, 1, 0, 1, 0, 1, 0, 1, 1, 0, 1, 1, 1, 2, 2, 1, 1, 2, 0, 2, 1, 1, 0, 1, 1, 1, 0, 1, 2, 0, 2, 1, 1, 1, 0, 1, 0, 1, 0, 1, 1, 2, 0, 0,
2, 1, 0, 1, 1, 1, 0, 1, 2, 0, 2, 1, 1, 0, 1, 2, 2, 1, 0, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 2, 0, 2, 1, 0, 1, 1, 1, 2, 0, 0,
0, 2, 1, 0, 1, 2, 0, 2, 1, 1, 0, 1, 2, 0, 2, 1, 1, 2, 2, 1, 2, 0, 2, 1, 1, 2, 2, 1, 2, 0, 0, 2, 1, 2, 0, 0, 0, 0, 2, 1, 1, 0, 1, 1,
1, 2, 2, 1, 1, 1, 0, 1, 0, 1, 0, 1, 1, 0, 1, 1, 0, 1, 0, 1, 1, 2, 0, 2, 1, 2, 2, 1, 1, 2, 2, 1, 0, 1, 0, 1, 1, 1, 2, 2, 1, 2, 0, 2,
1, 1, 2, 2, 1, 1, 0, 1, 1, 1, 2, 2, 1, 1, 2, 0, 0, 2, 1, 0, 1, 2, 0, 0, 0, 2, 1, 0, 1]
423
['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', '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', '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', '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', '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', '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', '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', '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', '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', '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', '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', '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', '0', '0', '0', '0', '0', '0', '0', '0',
'0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '1', '00', '11', '11', '11', '01', '00', '10', '11', '00', '00', '11', '01', '01',
'11', '11', '10', '00', '10', '00', '01', '01', '00', '01', '01', '11', '00', '11', '01', '01', '00', '01', '01', '11', '00', '11',
'01', '01', '00', '01', '10', '10', '01', '11', '11', '10', '11', '00', '00', '00', '00', '11', '01', '10', '01', '00', '01', '10',
'01', '00', '01', '01', '00', '01', '10', '10', '01', '00', '01', '10', '01', '11', '11', '10', '11', '11', '01', '01', '00', '01',
'01', '11', '00', '11', '01', '01', '11', '11', '10', '11', '00', '00', '00', '00', '11', '10', '00', '10', '11', '11', '01', '01',
'00', '01', '10', '01', '11', '11', '01', '01', '11', '11', '10', '00', '10', '00', '01', '10', '01', '00', '01', '10', '01', '11',
'11', '10', '11', '00', '00', '00', '00', '11', '01', '01', '00', '01', '01', '11', '00', '11', '01', '01', '00', '10', '11', '11',
```

2.4 Decoding: Parity Detection, Syndrome Testing, Error Correction, and Branch Metric Evaluation

This section looks the most different from the theoretical model; however, it encompasses all decoding methods in a more efficient manner. Rather than evaluating and printing each step (which isn't even possible for even the parity or shift register, as the printout cuts off after about 300 lines) we just simplify the decoder. This model is still able to show the state machines and branch metrics though. This is all contained within the "viterbi decoder" function. This is where the actual work ends

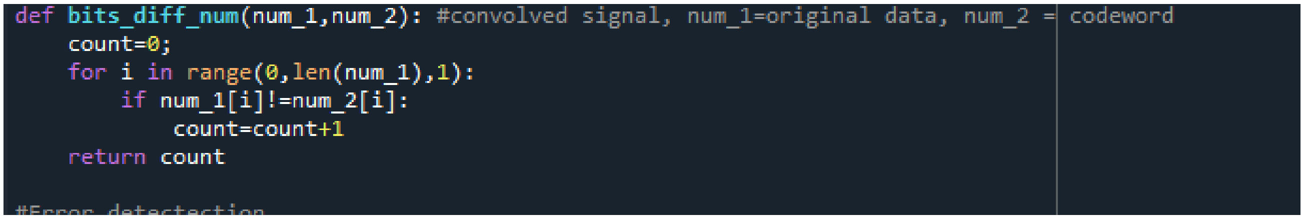


```
def viterbi_decoder(obs, start_metric, state_machine):
    #Trellis structure
    V = [{}]
    for st in state_machine:
        # Calculating the probability of both initial possibilities for the first observation
        V[0][st] = {"metric": start_metric[st]}
    #for first_b_metric > second_b_metric
    for t in range(1, len(obs)+1):
        V.append({})
        for st in state_machine:
            #Check for smallest bit difference from possible previous paths, adding with previous metric
            prev_st = state_machine[st]['b1']['prev_st']
            first_b_metric = V[(t-1)][prev_st]["metric"] + bits_diff_num(state_machine[st]['b1']['out_b'], obs[t - 1]
            prev_st = state_machine[st]['b2']['prev_st']
            second_b_metric = V[(t - 1)][prev_st]["metric"] + bits_diff_num(state_machine[st]['b2']['out_b'], obs[t -
            print(state_machine[st]['b1']['out_b'],obs[t - 1],t)
            if first_b_metric > second_b_metric:
                V[t][st] = {"metric" : second_b_metric,"branch":'b2'}
            else:
                V[t][st] = {"metric": first_b_metric, "branch": 'b1'}

    #print trellis nodes metric:
    for st in state_machine:
        for t in range(0,len(V)):
            #print(V[t][st],["metric"])
        #print(""),
    #print(""),

             smaller = min(V[t][st]["metric"] for st in state_machine)
             #traceback the path on smaller metric on last trellis column
             for st in state_machine:
                 if V[len(obs)-1][st]["metric"] == smaller:
                     source_state = st
                     for t in range(len(obs),0,-1):
                         branch = V[t][source_state]["branch"]
                         #print(state_machine[source_state][branch]['input_b']),
                         source_state = state_machine[source_state][branch]['prev_st']
                     #print (source_state+"\n")
             #print("Finish")
```

```
11 01 730
01 01 730
11 01 730
10 01 730
11 10 731
01 10 731
11 10 731
10 10 731
11 01 732
01 01 732
11 01 732
10 01 732
11 11 733
01 11 733
11 11 733
10 11 733
11 00 734
01 00 734
11 00 734
10 00 734
11 00 735
01 00 735
11 00 735
10 00 735
11 00 736
01 00 736
11 00 736
10 00 736
11 11 737
01 11 737
11 11 737
10 11 737
11 10 738
01 10 738
11 10 738
10 10 738
11 00 739
01 00 739
11 00 739
10 00 739
```

2.4.1 Parity Detection

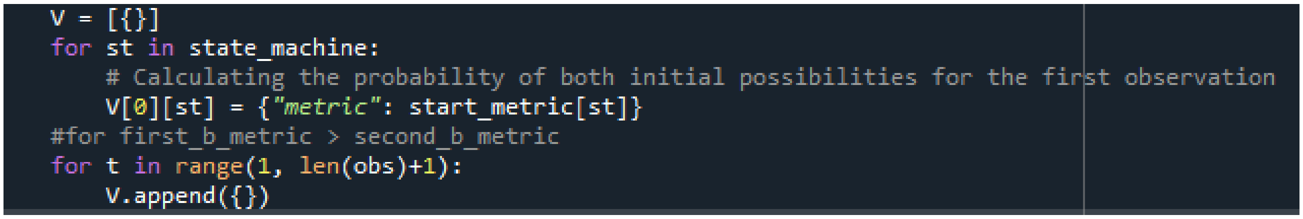
Logically, the decoder handles the coding process inverted to the encoder. Since the last step of the decoder is to produce a parity bit, the system now checks for the parity type (even or odd) and flags parts of the stream if there is a parity mismatch.



```
def bits_diff_num(num_1,num_2): #convolved signal, num_1=original data, num_2 = codeword
    count=0;
    for i in range(0,len(num_1),1):
        if num_1[i]!=num_2[i]:
            count=count+1
    return count

#Error detectection
```

2.4.2 Syndrome Testing

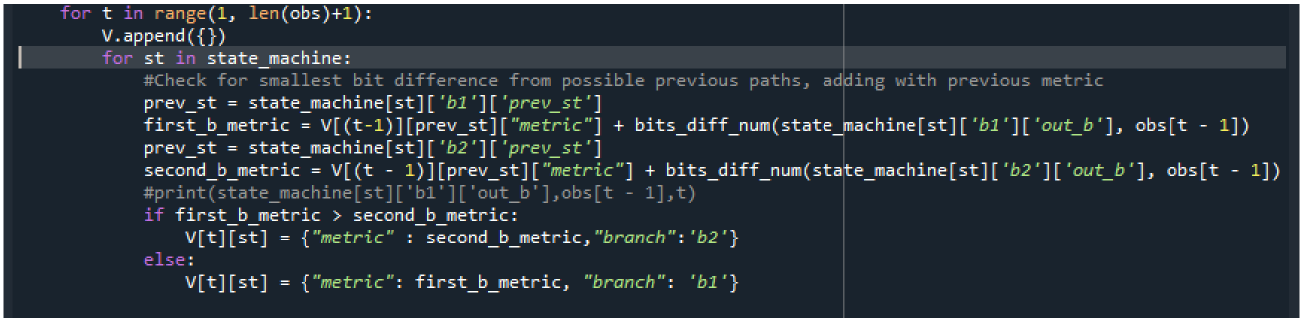
After the decoder checks for parity, the system then begins syndrome testing. As explained in the background section, the bits and strings flagged for error are evaluated through syndrome testing to find a pattern of error. The syndrome, or pattern, is described as the relationship S = rH(transpose) = eH(transpose). The python implementation here is evaluating each error vector and calculating the difference between r and e.



```
V = [{}]
for st in state_machine:
    # Calculating the probability of both initial possibilities for the first observation
    V[0][st] = {"metric": start_metric[st]}
#for first_b_metric > second_b_metric
for t in range(1, len(obs)+1):
    V.append({})
```

2.4.3 Error Correction

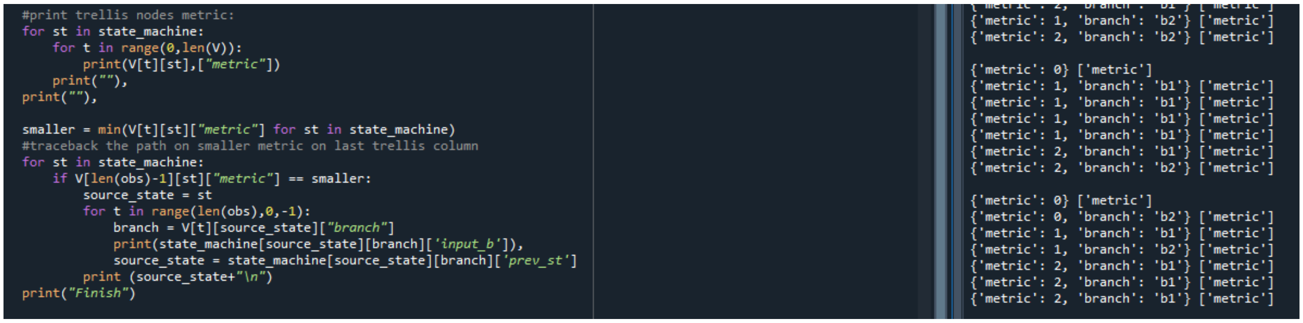
This aspect of the code is extracting the information from the syndrome tester, evaluating the bit difference, and replacing (when it starts appending) parts of the received information with the corrected stream (those with reduced bit difference). Be wary of the use of metric here, this is not branch metrics, this is bit metric, specifically bit difference for each tagged string.



```
    for t in range(1, len(obs)+1):
        V.append({})
        for st in state_machine:
            #Check for smallest bit difference from possible previous paths, adding with previous metric
            prev_st = state_machine[st]['b1']['prev_st']
            first_b_metric = V[(t-1)][prev_st]["metric"] + bits_diff_num(state_machine[st]['b1']['out_b'], obs[t - 1])
            prev_st = state_machine[st]['b2']['prev_st']
            second_b_metric = V[(t - 1)][prev_st]["metric"] + bits_diff_num(state_machine[st]['b2']['out_b'], obs[t - 1])
            #print(state_machine[st]['b1']['out_b'],obs[t - 1],t)
            if first_b_metric > second_b_metric:
                V[t][st] = {"metric" : second_b_metric,"branch":'b2'}
            else:
                V[t][st] = {"metric": first_b_metric, "branch": 'b1'}
```

2.4.4 Branch Metric Evaluation

This is where more comprehensive evaluation begins. No complicated math or actual decoding occurs here, but it is the most important step. Merely verification is happening, the received code is compared to a possible correct value. This changes with every K+3 length string so that each string is examined with their unique correct counterpart. As shown in the figure below a state machine and set of metrics is printed. This shows the received codeword and the corresponding bit difference. This process will repeat for two cases. The first case is to simply evaluate the metrics of the first attempt at decoding and correcting the received signal. The second case encompasses every nth attempt afterwards that does not achieve a complete bit difference of 0 (or whatever desired threshold).

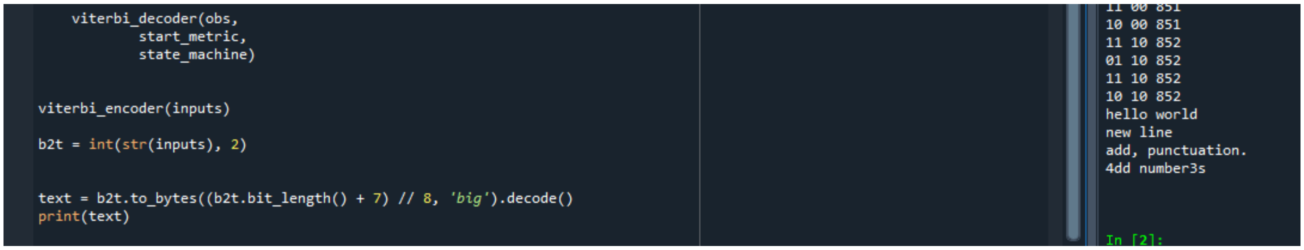


```
#print trellis nodes metric:
for st in state_machine:
    for t in range(0,len(V)):
        print(V[t][st],["metric"])
    print(""),
print(""),

smaller = min(V[t][st]["metric"] for st in state_machine)
#traceback the path on smaller metric on last trellis column
for st in state_machine:
    if V[len(obs)-1][st]["metric"] == smaller:
        source_state = st
        for t in range(len(obs),0,-1):
            branch = V[t][source_state]["branch"]
            print(state_machine[source_state][branch]['input_b']),
            source_state = state_machine[source_state][branch]['prev_st']
        print (source_state+"\n")
print("Finish")
```

```
{'metric': 2, 'branch': 'b1'} ['metric']
{'metric': 1, 'branch': 'b2'} ['metric']
{'metric': 2, 'branch': 'b2'} ['metric']

{'metric': 0} ['metric']
{'metric': 1, 'branch': 'b1'} ['metric']
{'metric': 1, 'branch': 'b1'} ['metric']
{'metric': 1, 'branch': 'b1'} ['metric']
{'metric': 1, 'branch': 'b1'} ['metric']
{'metric': 2, 'branch': 'b1'} ['metric']
{'metric': 2, 'branch': 'b2'} ['metric']

{'metric': 0} ['metric']
{'metric': 0, 'branch': 'b2'} ['metric']
{'metric': 1, 'branch': 'b1'} ['metric']
{'metric': 1, 'branch': 'b2'} ['metric']
{'metric': 2, 'branch': 'b1'} ['metric']
{'metric': 2, 'branch': 'b1'} ['metric']
{'metric': 2, 'branch': 'b1'} ['metric']
```

2.5 Binary Back to ASCII

Finally, the system takes the outputs produced by the decoder and converts the binary array back to ASCII. Once this process is complete, the user can compare this output to the input and determine if the encoder and decoder performed successfully. In the figure below, we can see that the system indeed worked, as the output text is the same as the input text.



```
    viterbi_decoder(obs,
            start_metric,
            state_machine)


viterbi_encoder(inputs)

b2t = int(str(inputs), 2)


text = b2t.to_bytes((b2t.bit_length() + 7) // 8, 'big').decode()
print(text)
```

```
11 00 851
10 00 851
11 10 852
01 10 852
11 10 852
10 10 852
hello world
new line
add, punctuation.
4dd number3s



In [2]:
```

3 Acknowledgements

First, I would like to thank Dr. Christopher Groppi for giving me the opportunity to even do this research. He has also been a great mentor and pushed me to be as successful as possible. Without him, this paper would have been a mess and so would my code. I would then like to extend my thanks to the greater School of Earth and Space Exploration for having such an insightful, helpful, and inclusive community. I would never have gotten involved in this type of work had it not been for diverse and exciting research going on at the school and the opportunities to become involved. Lastly, I would like to

thank Bernard Sklar for publishing such a clear and comprehensive text book on all things related to digital communications. Without that book I would have been totally lost.

For anyone interested in viewing the code used for this project, please follow the link below. This contains not only this specific code, but all other coding and collaborative work I've done thus far. https://github.com/jbmcdoug?tab=repositories

You may also browse more of my publications, my areas of expertise, and reach out to me directly through my Linkedin page. https://www.linkedin.com/in/johnathan-mcdougal-a71749191/