BHARATIYA VIDYA BHAVAN'S

SARDAR PATEL INSTITUTE OF TECHNOLOGY

(Empowered Autonomous Institute Affiliated to University of Mumbai)

[Knowledge is Nectar]

Department of Computer Science Engineering

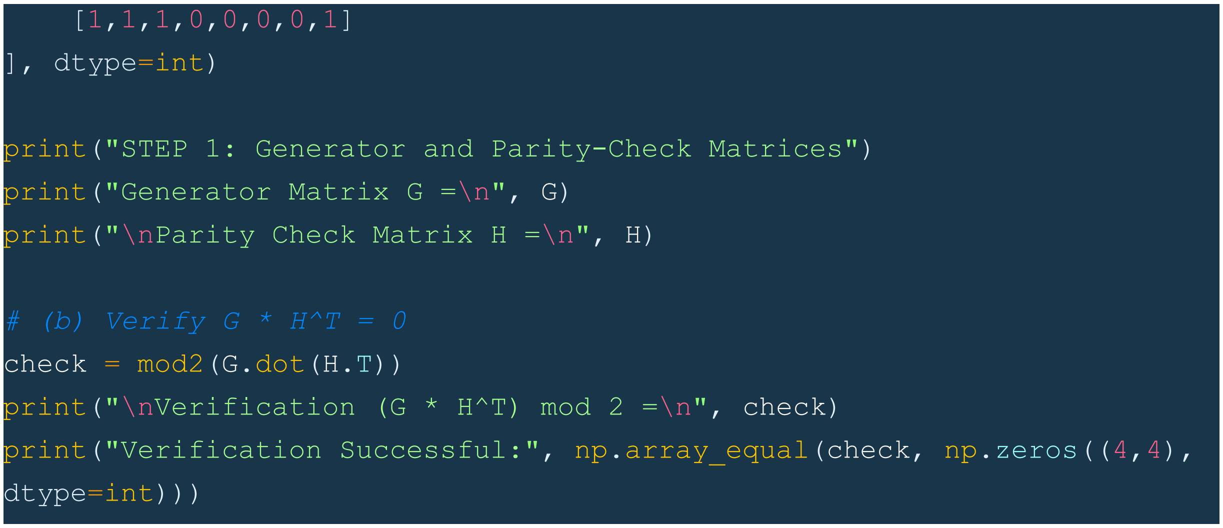
Name	Pratham Satyawan Masurkar
UID	2023800056
Department	CSE
Lab	5
Aim	**To analyze and implement a(8,4) Hamming code, which can detect and correct single-bit errors.**

Implementation

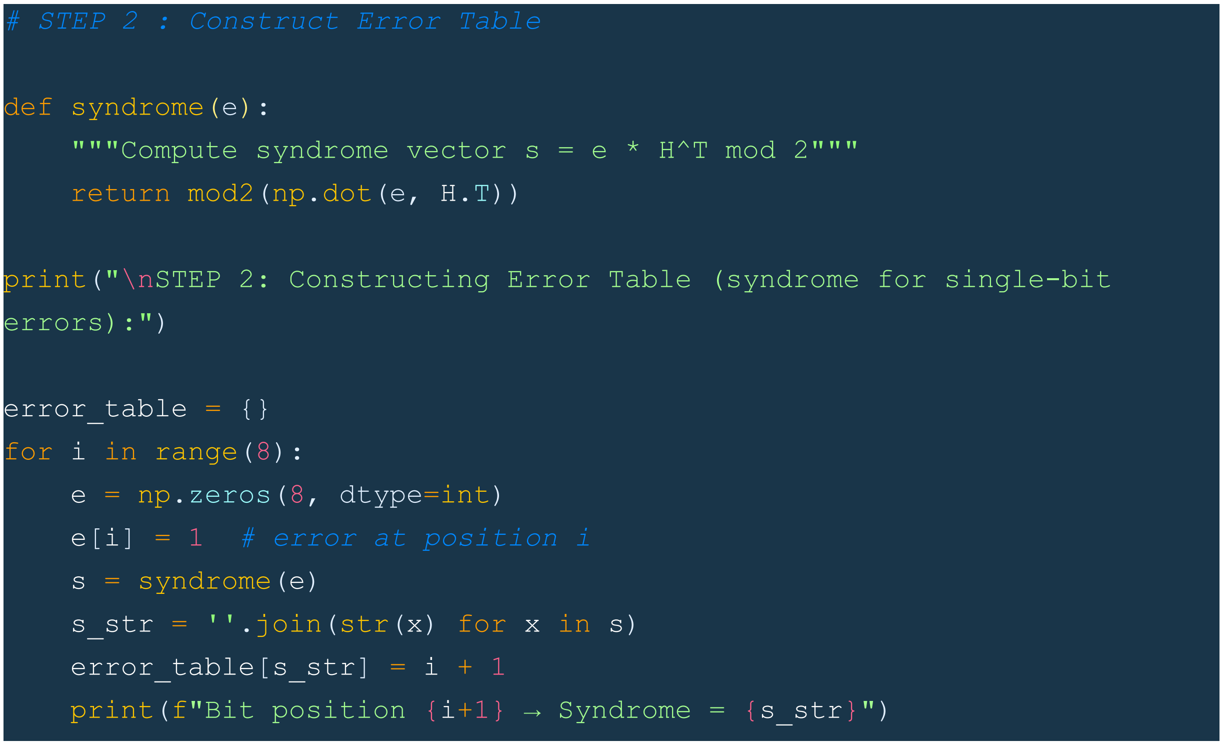
```
import numpy as np
```

```
# Helper function for modulo-2 arithmetic
def mod2(x):
    return np.mod(x, 2)
```

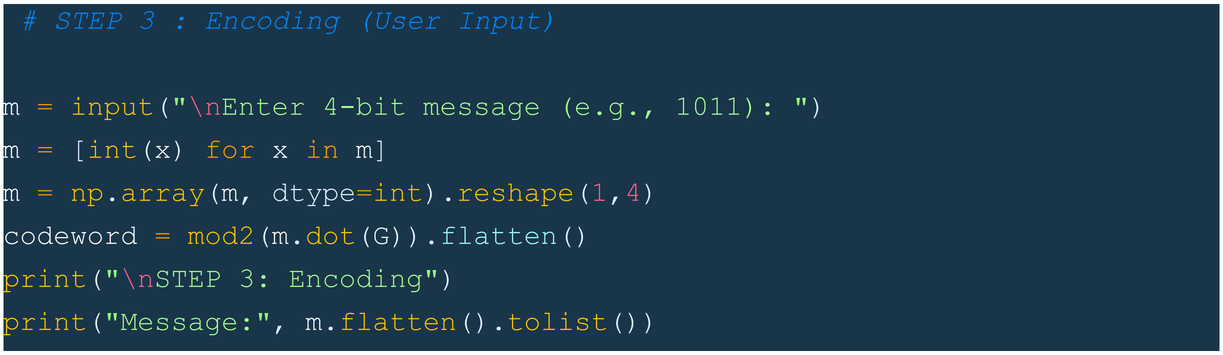
```
# STEP 1 : Generator and Parity-Check Matrices

# Generator matrix G
G = np.array([
    [1,0,0,0,0,1,1,1],
    [0,1,0,0,1,0,1,1],
    [0,0,1,0,1,1,0,1],
    [0,0,0,1,1,1,1,0]
], dtype=int)

# Parity-check matrix H
H = np.array([
    [0,1,1,1,1,0,0,0],
    [1,0,1,1,0,1,0,0],
    [1,1,0,1,0,0,1,0],
```



```
    [1,1,1,0,0,0,0,1]
], dtype=int)

print("STEP 1: Generator and Parity-Check Matrices")
print("Generator Matrix G =\n", G)
print("\nParity Check Matrix H =\n", H)

# (b) Verify G * H^T = 0
check = mod2(G.dot(H.T))
print("\nVerification (G * H^T) mod 2 =\n", check)
print("Verification Successful:", np.array_equal(check, np.zeros((4,4),
dtype=int)))
```



```
# STEP 2 : Construct Error Table

def syndrome(e):
    """Compute syndrome vector s = e * H^T mod 2"""
    return mod2(np.dot(e, H.T))

print("\nSTEP 2: Constructing Error Table (syndrome for single-bit
errors):")

error_table = {}
for i in range(8):
    e = np.zeros(8, dtype=int)
    e[i] = 1  # error at position i
    s = syndrome(e)
    s_str = ''.join(str(x) for x in s)
    error_table[s_str] = i + 1
    print(f"Bit position {i+1} → Syndrome = {s_str}")
```



```
 # STEP 3 : Encoding (User Input)

m = input("\nEnter 4-bit message (e.g., 1011): ")
m = [int(x) for x in m]
m = np.array(m, dtype=int).reshape(1,4)
codeword = mod2(m.dot(G)).flatten()
print("\nSTEP 3: Encoding")
print("Message:", m.flatten().tolist())
print("Encoded 8-bit Codeword:", codeword.tolist())
```

```
# STEP 4 : Transmission with Errors

# Introduce single-bit error
error_pos = int(input("\nEnter error position to introduce (1-8): "))
r = codeword.copy()
r[error_pos-1] ^= 1  # flip bit
print("\nSTEP 4: Transmission with Error")
print("Error introduced at position:", error_pos)
print("Received vector r =", r.tolist())

# (a) Compute syndrome
s = syndrome(r)
s_str = ''.join(str(x) for x in s)
print("Syndrome vector s =", s.tolist(), "→", s_str)

# (b) Identify error location
if s_str in error_table:
    detected_pos = error_table[s_str]
    print("Error detected at position:", detected_pos)
    # (c) Correct the error
    r[detected_pos-1] ^= 1
    print("Corrected codeword:", r.tolist())
else:
    print("No single-bit error detected (syndrome = 0000)")
```

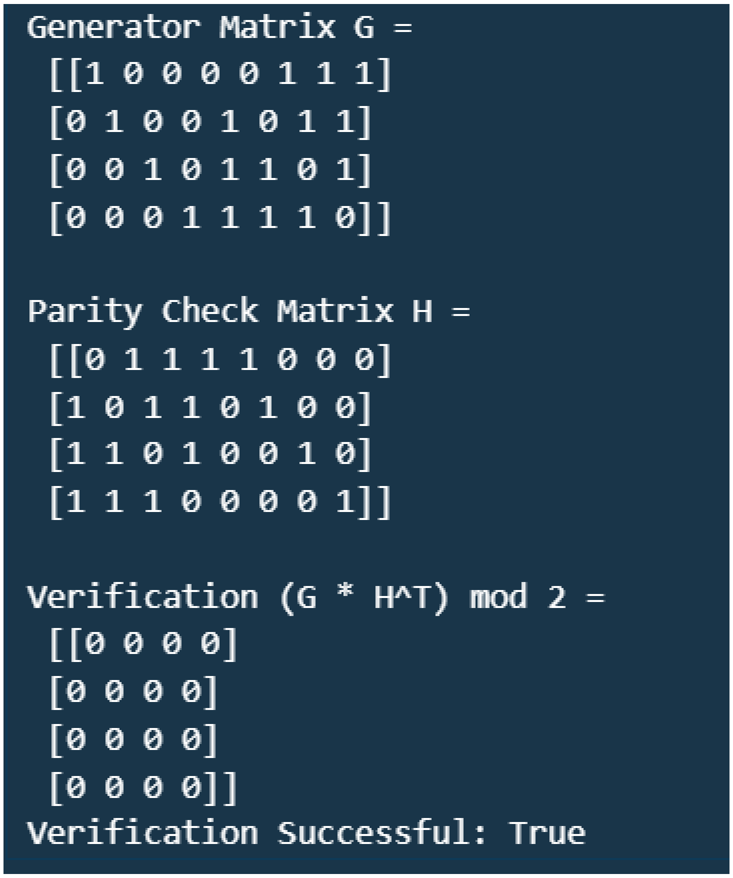
```
# STEP 5 : Decoding

decoded_message = r[:4]
print("\nSTEP 5: Decoding")
print("Decoded Message:", decoded_message.tolist())
```

```
# STEP 6 : Change Error Position

new_error_pos = int(input("\nSTEP 6: Enter a different error position (1-8): "))
r2 = codeword.copy()
r2[new_error_pos-1] ^= 1
s2 = syndrome(r2)
```

```
s2_str = ''.join(str(x) for x in s2)
print(f"Error at position {new_error_pos} → Syndrome = {s2.tolist()} →
{s2_str}")
```

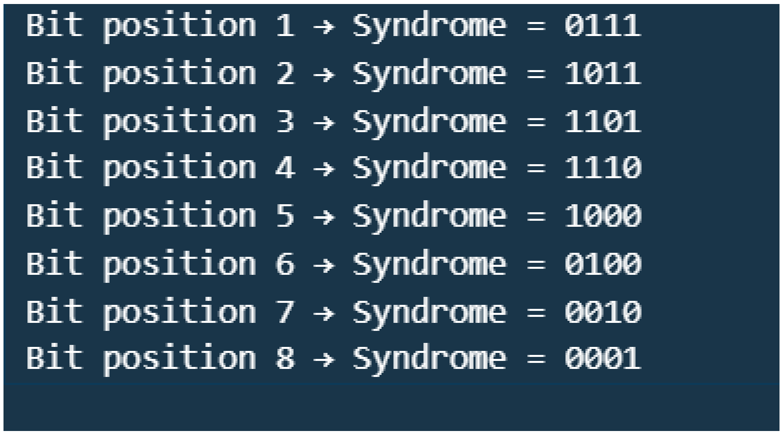
```
# STEP 7 : Different Data, Same Error Position

new_message = input("\nSTEP 7: Enter a different 4-bit message: ")
new_message = np.array([int(x) for x in new_message],
dtype=int).reshape(1,4)
new_codeword = mod2(new_message.dot(G)).flatten()
r3 = new_codeword.copy()
r3[error_pos-1] ^= 1
s3 = syndrome(r3)
s3_str = ''.join(str(x) for x in s3)

print("\nNew message:", new_message.flatten().tolist())
print("Error introduced at same position", error_pos)
print("New Syndrome =", s3.tolist(), "→", s3_str)
```

Outputs	Generator and Parity Check Matrix

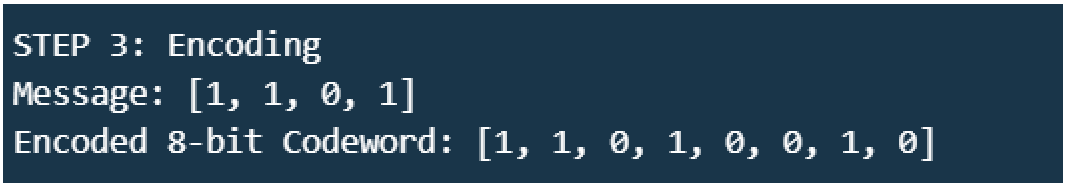


```
Generator Matrix G =
 [[1 0 0 0 0 1 1 1]
 [0 1 0 0 1 0 1 1]
 [0 0 1 0 1 1 0 1]
 [0 0 0 1 1 1 1 0]]

Parity Check Matrix H =
 [[0 1 1 1 1 0 0 0]
 [1 0 1 1 0 1 0 0]
 [1 1 0 1 0 0 1 0]
 [1 1 1 0 0 0 0 1]]

Verification (G * H^T) mod 2 =
 [[0 0 0 0]
 [0 0 0 0]
 [0 0 0 0]
 [0 0 0 0]]
Verification Successful: True
```

```
Constructing Error Table (syndrome for single-bit errors):
```

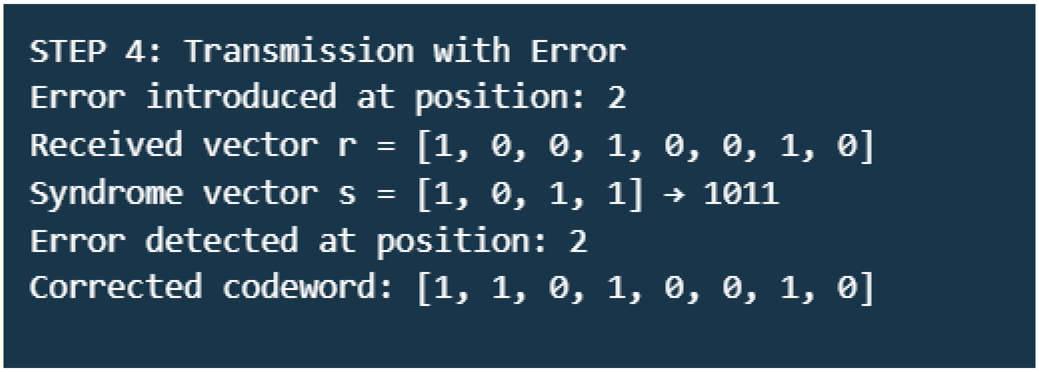


```
Bit position 1 → Syndrome = 0111
Bit position 2 → Syndrome = 1011
Bit position 3 → Syndrome = 1101
Bit position 4 → Syndrome = 1110
Bit position 5 → Syndrome = 1000
Bit position 6 → Syndrome = 0100
Bit position 7 → Syndrome = 0010
Bit position 8 → Syndrome = 0001
```

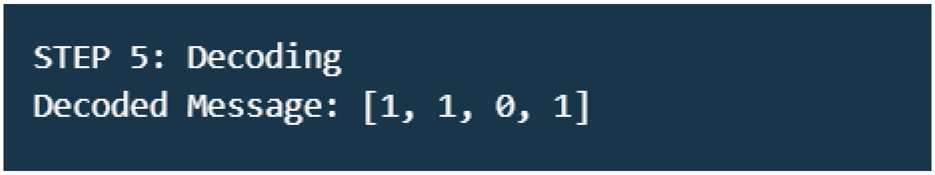


```
STEP 3: Encoding
Message: [1, 1, 0, 1]
Encoded 8-bit Codeword: [1, 1, 0, 1, 0, 0, 1, 0]
```

<table>
<tr>
<td></td>
<td>

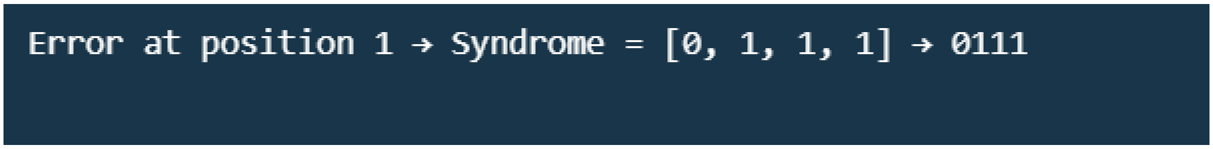


```
STEP 4: Transmission with Error
Error introduced at position: 2
Received vector r = [1, 0, 0, 1, 0, 0, 1, 0]
Syndrome vector s = [1, 0, 1, 1] → 1011
Error detected at position: 2
Corrected codeword: [1, 1, 0, 1, 0, 0, 1, 0]
```



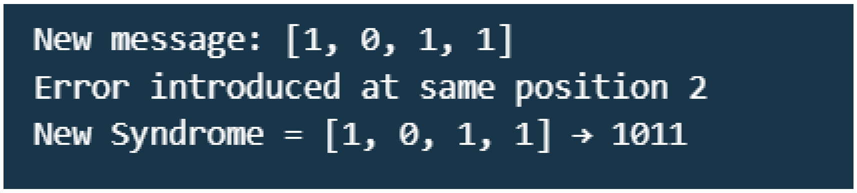
```
STEP 5: Decoding
Decoded Message: [1, 1, 0, 1]
```

Change Error Position



```
Error at position 1 → Syndrome = [0, 1, 1, 1] → 0111
```

Different Data, Same Error Position



```
New message: [1, 0, 1, 1]
Error introduced at same position 2
New Syndrome = [1, 0, 1, 1] → 1011
```

</td>
</tr>
<tr>
<td>Conclusion</td>
<td>

built a system using the (8,4) Hamming code that can automatically find and fix a single error in a piece of data. I created an "encoding" tool (Generator matrix) and a "checking" tool (Parity Check matrix) and proved they worked together correctly. I then made a "lookup table" (syndrome list) that shows the unique error message for every possible bit flip. Finally, I tested it by sending a message, flipping one bit, and watching the system use the error message to find and correct the mistake, proving it's a reliable way to protect data.

</td>
</tr>
</table>