Hill Cipher

Problem:

Implement the Hill Cipher, a polygraphic substitution cipher based on linear algebra. The task involves creating a program to encrypt and decrypt messages using a matrix key. The Hill Cipher is an example of a block cipher that encrypts blocks of text using matrix multiplication.

Description:

The Hill Cipher is a type of substitution cipher that uses linear algebra to encrypt and decrypt messages. It operates on blocks of text rather than individual characters. The encryption and decryption processes involve matrix multiplication and modular arithmetic. In this implementation:

- Characters are converted to numeric values (A=0, B=1, ..., Z=25).
- A key matrix is used for encryption and decryption.
- The modular inverse of the matrix is computed for decryption.

Code:

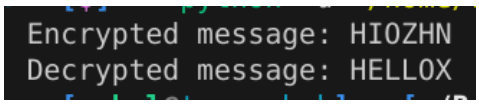
1. hill_cipher_basic.py

```
import numpy as np

def mod_matrix_inverse(matrix, modulus):
    """
    Calculate the modular inverse of a given matrix.
    """
    # Compute the determinant of the matrix and round it to ensure it's an integer
    det = int(np.round(np.linalg.det(matrix)))

    # Find the modular inverse of the determinant under the given modulus (usually 26)
    det_inv = pow(det, -1, modulus)

    # Calculate the matrix inverse, multiply by the determinant's inverse, and apply modulus
    # This gives us the modular inverse of the matrix
    matrix_mod_inv = det_inv * np.round(det * np.linalg.inv(matrix)).astype(int) % modulus

    return matrix_mod_inv

def hill_encrypt(message, key_matrix):
    """
    Encrypt a message using the Hill cipher method.
```

```
    """
    # Convert each character in the message to its corresponding numeric value (A=0, B=1, ..., Z=25)
    message_numbers = [ord(char) - ord('A') for char in message.upper()]

    # If the message length is odd, pad it with 'X' (value 23) to ensure even length
    if len(message_numbers) % 2 != 0:
        message_numbers.append(ord('X') - ord('A'))

    # Reshape the message numbers into a 2xN matrix where each column is a pair of letters
    message_matrix = np.array(message_numbers).reshape(-1, 2).T

    # Multiply the key matrix by the message matrix to encrypt it, then apply modulus 26
    encrypted_matrix = np.dot(key_matrix, message_matrix) % 26

    # Convert the encrypted numeric values back to letters and concatenate them into a string
    encrypted_message = ''.join(chr(num + ord('A')) for num in encrypted_matrix.T.flatten())

    return encrypted_message

def hill_decrypt(encrypted_message, key_matrix):
    """
    Decrypt a message encrypted using the Hill cipher method.
    """
    # Convert the encrypted message back to numeric values (A=0, B=1, ..., Z=25)
    encrypted_numbers = [ord(char) - ord('A') for char in encrypted_message.upper()]

    # Reshape the encrypted numbers into a 2xN matrix, similar to how the message was processed
    encrypted_matrix = np.array(encrypted_numbers).reshape(-1, 2).T

    # Calculate the inverse of the key matrix under modulus 26 for decryption
    key_matrix_inv = mod_matrix_inverse(key_matrix, 26)

    # Multiply the inverse key matrix by the encrypted matrix to decrypt it, then apply modulus 26
    decrypted_matrix = np.dot(key_matrix_inv, encrypted_matrix) % 26
```

```
    # Convert the decrypted numeric values back to letters and concatenate them into a string
    decrypted_message = ''.join(chr(int(num) + ord('A')) for num in decrypted_matrix.T.flatten())

    return decrypted_message

# Example usage of the Hill cipher

# Define the encryption key matrix (2x2 matrix)
key_matrix = np.array([[3, 3], [2, 5]])  # This matrix should be invertible mod 26

# Define the message to encrypt
message = "HELLO"

# Encrypt the message using the Hill cipher
encrypted_message = hill_encrypt(message, key_matrix)
print(f"Encrypted message: {encrypted_message}")

# Decrypt the message back to the original text
decrypted_message = hill_decrypt(encrypted_message, key_matrix)
print(f"Decrypted message: {decrypted_message}")
```

Output:



Code Explanation:

1. **Importing Libraries:**
    - numpy is imported for matrix operations and modular arithmetic.
2. **mod_matrix_inverse Function:**
    - Calculates the modular inverse of a matrix under a specified modulus (usually 26).
    - Computes the determinant, finds its modular inverse, and uses it to find the modular inverse of the matrix.
3. **hill_encrypt Function:**
    - Converts the plaintext message into numeric values and reshapes it into a matrix.
    - Multiplies the key matrix by the message matrix to encrypt it, applying modulus 26 to ensure results stay within the alphabetic range.
    - Converts the encrypted numeric values back to characters.
4. **hill_decrypt Function:**

- Converts the encrypted message into numeric values and reshapes it into a matrix.
- Computes the inverse of the key matrix under modulus 26.
- Multiplies the inverse key matrix by the encrypted matrix to decrypt it and converts the result back to characters.

5. **Main Block:**
    - Demonstrates encryption and decryption with an example message and key matrix.
    - Prints the encrypted and decrypted messages.

Performance Analysis:

- **Space Complexity:**
    - The space complexity is O(n), where n is the number of characters in the message. This is due to storing the numeric representations and matrices.
- **Time Complexity:**
    - The time complexity is O(n), where n is the number of characters in the message. The encryption and decryption operations involve matrix multiplications and inversions, which are linear with respect to the number of characters processed.

Modified Version:

Code:

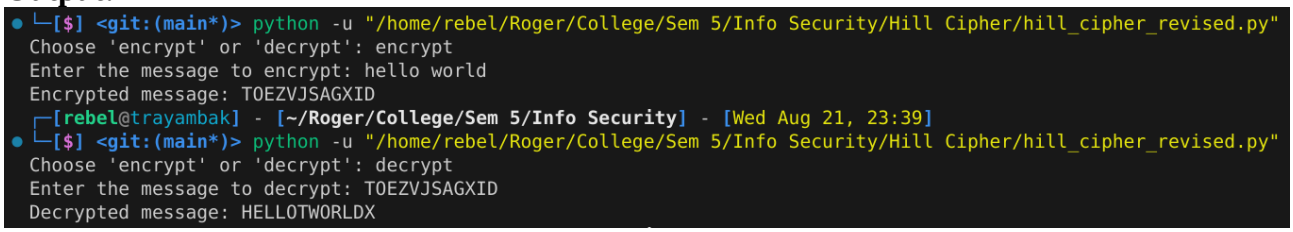
1. hill_cipher_revised.py

```
import numpy as np
import random

# Function to save a matrix to a text file
def save_matrix_to_file(matrix, filename):
    # Save the matrix to a file with integers format
    np.savetxt(filename, matrix, fmt='%d')

# Function to load a matrix from a text file
def load_matrix_from_file(filename):
    # Load the matrix from the file with integer type
    return np.loadtxt(filename, dtype=int)

# Function to generate a random matrix of a given size that is invertible mod 26
def generate_random_matrix(size, modulus):
    while True:
        # Generate a random matrix with values between 0 and the modulus (26)
        matrix = np.random.randint(0, modulus, size=(size, size))
        # Calculate the determinant of the matrix
        det = int(np.round(np.linalg.det(matrix)))
        # Ensure the determinant is coprime with the modulus (26) for invertibility
        if np.gcd(det, modulus) == 1:
            return matrix

# Function to calculate the modular inverse of a matrix under a given modulus (26)
def mod_matrix_inverse(matrix, modulus):
    # Calculate the determinant of the matrix and round it to avoid precision issues
    det = int(np.round(np.linalg.det(matrix)))
    # Calculate the modular inverse of the determinant mod 26
    det_inv = pow(det, -1, modulus)
    # Calculate the matrix inverse, scale by determinant inverse, and apply mod 26
    matrix_mod_inv = det_inv * np.round(det * np.linalg.inv(matrix)).astype(int) % modulus
    return matrix_mod_inv

# Function to encrypt a message using the Hill cipher with an additional random matrix
def hill_encrypt(message, key_matrix, random_matrix):
    # Convert the message characters to corresponding numbers (A=0, B=1, ..., Z=25)
    message_numbers = [ord(char) - ord('A') for char in message.upper()]

    # If the message length is odd, pad it with 'X' to ensure even length
    if len(message_numbers) % 2 != 0:
        message_numbers.append(ord('X') - ord('A'))
```

```
    # Reshape the message numbers into a 2xN matrix for encryption
    message_matrix = np.array(message_numbers).reshape(-1, 2).T

    # Multiply the message matrix with the random matrix, then apply modulus 26
    intermediate_matrix = np.dot(random_matrix, message_matrix) % 26
    # Multiply the resulting matrix with the key matrix and apply modulus 26
    encrypted_matrix = np.dot(key_matrix, intermediate_matrix) % 26

    # Convert the resulting numbers back to letters to form the encrypted message
    encrypted_message = ''.join(chr(num + ord('A')) for num in encrypted_matrix.T.flatten())

    return encrypted_message

# Function to decrypt a message using the Hill cipher with an additional random matrix
def hill_decrypt(encrypted_message, key_matrix, random_matrix):
    # Convert the encrypted message characters back to numbers (A=0, B=1, ..., Z=25)
    encrypted_numbers = [ord(char) - ord('A') for char in encrypted_message.upper()]

    # Reshape the encrypted numbers into a 2xN matrix for decryption
    encrypted_matrix = np.array(encrypted_numbers).reshape(-1, 2).T

    # Calculate the modular inverse of the key matrix for decryption
    key_matrix_inv = mod_matrix_inverse(key_matrix, 26)
    # Calculate the modular inverse of the random matrix for decryption
    random_matrix_inv = mod_matrix_inverse(random_matrix, 26)

    # Multiply the encrypted matrix with the inverse key matrix and apply modulus 26
    intermediate_matrix = np.dot(key_matrix_inv, encrypted_matrix) % 26
    # Multiply the resulting matrix with the inverse random matrix and apply modulus 26
    decrypted_matrix = np.dot(random_matrix_inv, intermediate_matrix) % 26

    # Convert the resulting numbers back to letters to form the decrypted message
    decrypted_message = ''.join(chr(int(num) + ord('A')) for num in decrypted_matrix.T.flatten())

    return decrypted_message

# Main program to handle user input and perform encryption or decryption
def main():
    # Define the filename to save and load the random matrix
    filename = "random_matrix.txt"
    # Define a fixed key matrix for encryption (2x2 matrix)
    key_matrix = np.array([[3, 3], [2, 5]])

    # Prompt the user to choose between encryption and decryption
    choice = input("Choose 'encrypt' or 'decrypt': ").strip().lower()

    if choice == 'encrypt':
        # Get the plaintext message to encrypt from the user
        message = input("Enter the message to encrypt: ").strip().upper()

        # Generate a random 2x2 matrix for encryption and save it to a file
        random_matrix = generate_random_matrix(2, 26)
```

```
        save_matrix_to_file(random_matrix, filename)

        # Encrypt the message using the Hill cipher and the random matrix
        encrypted_message = hill_encrypt(message, key_matrix, random_matrix)
        print(f"Encrypted message: {encrypted_message}")

    elif choice == 'decrypt':
        # Get the encrypted message to decrypt from the user
        encrypted_message = input("Enter the message to decrypt: ").strip().upper()

        # Load the random matrix from the file for decryption
        random_matrix = load_matrix_from_file(filename)

        # Decrypt the message using the Hill cipher and the random matrix
        decrypted_message = hill_decrypt(encrypted_message, key_matrix, random_matrix)
        print(f"Decrypted message: {decrypted_message}")

    else:
        # Print an error message if the user makes an invalid choice
        print("Invalid choice. Please choose 'encrypt' or 'decrypt'.")

# Run the main program when the script is executed
if __name__ == "__main__":
    main()
```

Output:



Code Explanation:

1. **Importing Libraries:**
    - numpy is imported for matrix operations and modular arithmetic.
    - random is imported for generating random matrices.
2. **save_matrix_to_file Function:**
    - Saves a matrix to a text file with integer format.
3. **load_matrix_from_file Function:**
    - Loads a matrix from a text file with integer type.
4. **generate_random_matrix Function:**
    - Generates a random matrix of a specified size with values between 0 and the modulus.
    - Ensures the matrix is invertible by checking if the determinant is coprime with the modulus.
5. **mod_matrix_inverse Function:**
    - Calculates the modular inverse of a matrix under a given modulus.

- Computes the determinant, finds its modular inverse, and scales the matrix inverse.

6. **hill_encrypt Function:**
    - Converts the message into numeric values and reshapes it into a matrix.
    - Multiplies the message matrix with a random matrix and then with the key matrix, applying modulus 26.
    - Converts the resulting numbers back to characters for the encrypted message.
7. **hill_decrypt Function:**
    - Converts the encrypted message into numeric values and reshapes it into a matrix.
    - Calculates the modular inverses of both the key and random matrices.
    - Multiplies the encrypted matrix with the inverse key matrix and then with the inverse random matrix, applying modulus 26.
    - Converts the resulting numbers back to characters for the decrypted message.
8. **main Function:**
    - Handles user input for encryption or decryption.
    - Generates and saves a random matrix for encryption or loads it for decryption.
    - Encrypts or decrypts the message based on user choice and displays the result.

Performance Analysis:

- **Space Complexity:**
    - The space complexity is O(n), where n is the number of characters in the message. This is due to storing the numeric representations and matrices.

- **Time Complexity:**
    - The time complexity is O(n), where n is the number of characters in the message. The encryption and decryption involve matrix multiplications and inversions, which are linear with respect to the number of characters processed.