MATH 2138: LINEAR ALGEBRA

Linear Algebra in Computer Science Proposal

Course: Math 2138 - Linear Algebra

Group Members: Benjamin Asjali, Kirby Calong, Luis Palparan, Mark Tan, Sam Tesoro

Date Submitted: 11/08/2025

Proposal #1 - Text To Image Cryptography

Introduction and CS Context:

In modern times, the security of our digital information remains a major concern due to the risks posed by evolving digital tools that are increasingly becoming accessible at a concerning pace. Data encryption (the process of converting readable data into unreadable data locked by a decryption key) was one way to solve this, with key-based ciphers and hashing being common examples used as cybersecurity measures, but these traditional methods have gotten predictable over the years and may not effectively conceal patterns within the encrypted output. **How can we adapt to this growing threat with innovative, harder-to-decrypt methods of encryption?**

A form of data encryption that is increasing in popularity due to its unique and secure encryption and decryption method is **Text to Image Cryptography**. This is the process of transforming a text into a square image where each pixel stores three encoded characters (using the R, G, B color channels). The numeric representation of each character (e.g., A=1, B=2, etc.) will serve as the base data for encryption. Then, the resulting image will undergo a series of linear algebraic manipulations and transformations to produce an encrypted image that conceals the original data. The transformations are stored as a decryption key, which allows only the owner of the key to reconstruct the image back to its original text by reversing the algebraic transformations applied to it. This approach uses the principles of linear algebra as a tool, where invertible matrix transformations provide a mathematical foundation for secure and reversible encryption (Stallings, 2017).

Its unique encryption method makes it difficult for unsuspecting individuals to realize that the image even holds an encrypted message. Additionally, this benefits its security immensely as it is also designed to be difficult to crack and decrypt without the necessary key that holds the unique steps to doing so. Such a system's security relies on the computational difficulty of matrix inversion without knowledge of the original key matrix operations, a foundation in cryptographic security (Katz &

Lindell, 2020). In conclusion, Image Cryptography is safe, secure, and emphasizes the use of linear algebra in the development of modern data security measures.

Mathematical Background:

- **Vector Operations**

  A vector is a mathematical object that has both magnitude and direction, and it is an element of a vector space (Axler, 2015). In computational applications, it is often represented as an ordered list of numbers called a column matrix.

  Vector operations will be applied in this project for the encoding and encryption process. Each character from the input text is first converted into a numerical value (e.g., A = 1, B = 2, etc.), which is then represented as a 3 x 1 matrix that contains 3 different characters corresponding to the R, G, B color channels:

$$v = \begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} f(C_1) \\ f(C_2) \\ f(C_3) \end{bmatrix}$$

  Figure 1.1. RGB Vector Representation
  ***Implementation note**: (Z256)^3 with arithmetic modulo 256.*

  where C_1, C_2, and C_3 represent the 3 characters. The vectors are then arranged in an n x n matrix to form the initial numerical matrix of dimensions n x n, where each element is itself a vector in R^3.

  ***Implementation Note:** While the exposition refers to vectors over the reals for intuition, the implementation operates component-wise modulo 256. Practically, each pixel is an element of (Z256)^3. This structure is a module over the ring Z256, not a Euclidean vector space; inner-product notions like angles and lengths are not used.*

- **Matrices**

  A matrix is a rectangular array of numbers, symbols, or expressions arranged in rows and columns. An m x n matrix defines a linear transformation from two vector spaces R^n (Lay et al., 2016).

  The core of the encryption process is the application of a linear transformation via matrix multiplication. A hidden, invertible key matrix *K* with dimension n x n is generated. The encryption is performed by the matrix multiplication of *K* with the image matrix ***I***. The resulting encrypted matrix *E* is calculated as:

  ***Implementation Note:** In code, K is a 3×3 key applied independently to each RGB pixel vector rather than an n×n global matrix over the whole image. Conceptually this is equivalent to a block-diagonal global matrix with identical 3×3 blocks.*

$$E = KI$$

Figure 1.2. Formula for encrypted matrix E
Implementation notes: *Implementation uses per-pixel 3×3 K: E_pixel = K v mod 256*

This operation is a linear combination. Each element *E*ij is the dot product of the i-th row of *K* and the j-th column of *I*, effectively mixing the values of all original character vectors. On the other hand, decryption requires the inverse of the key matrix K^{-1}, which exists if and only if *K* is nonsingular and invertible. The decryption is calculated by:

$$I = K^{-1}E$$

Figure 1.3. Formula for decryption (inverse operation)

This ensures that the original image matrix can be perfectly recovered, demonstrating that the encryption is a bijective map when K is invertible (Strang, 2016).

- **Determinants**

  The determinant of a square matrix A is a scalar value that determines a critical property for this cryptography process to work: invertibility.

  The fundamental rule for determinants is:

$$det(K) \neq 0 \leftrightarrow K^{-1} \ exists$$

Figure 1.4. Definition for Determinants

For the encryption process to be reversible, the key matrix K must have an inverse so that decryption can be calculated. Therefore, the algorithm must verify that det(K) is not equal to 0 during key generation. If det(K) = 0, the matrix is singular, the encryption is a one-way transformation, and the original data is lost.

Implementation Note: *Only the 3×3 pixel key's determinant is checked (det(K) ≠ 0 and coprime with 256); a full n×n determinant is not computed in this version.*

Implementation Note: *In code, we require gcd(det(K), 256) = 1 so that K has a modular inverse over Z256. This is stronger than det(K) ≠ 0 over the reals and matches the implementation.*

Mathematical Formulation:

- **Core Variables**
  - ***I***: n x n image matrix (each element is an RGB vector)
  - ***K***: n x n invertible key matrix
  - ***E***: n x n encrypted matrix

- **det(K)**: determinant (must be ≠ 0)
- K^{-1}: inverse key matrix

- **Step-by-Step Formulation**

1. **Encoding:**
   The text, composing a string and int, is divided into chunks of three characters. Each chunk is used to populate the R, G, and B channels of a single pixel vector in the image matrix *I.* The model for this is defined by:

$$I = \begin{bmatrix} v_{11} & \cdots & v_{1j} \\ \vdots & \ddots & \vdots \\ v_{i1} & \cdots & v_{ij} \end{bmatrix} \textit{ where } v_{ij} = \begin{bmatrix} f(C_1) \\ f(C_2) \\ f(C_3) \end{bmatrix}$$

Figure 1.5. Formula for the image matrix of pixel vectors

2. **Encryption:**
   The encryption is performed by multiplying the key matrix *K* by the image matrix *I*.

$$E = KI$$

Figure 1.2. Formula for encrypted matrix E

Implementation notes: *The current prototype applies the linear transform separately to each pixel's RGB vector. Replacing per-pixel 3×3 keys with a full n×n image-wide K would require assembling all channel values into a larger vector and is left as future work.*

Each pixel vector E_{ij} of the resulting encrypted matrix E is calculated by:

$$E_{ij} = \sum_{k=1}^{n} K_{ik} I_{kj}$$

Figure 1.6. Formula for the sum of scalar-vector multiplications for encryption

3. **Decryption:**
   This is the inverse operation of encryption in order to revert the encrypted image back to its original text form. The encrypted matrix E is multiplied by the inverse of the key matrix K^{-1}.

$$I = K^{-1}E$$

Figure 1.3. Formula for decryption (inverse operation)

Implementation Note: *Decryption in code applies K^{-1} to each pixel vector individually; this mirrors inverting each block on the block-diagonal representation.*

Implementation Plan:

Data Representation:

A. Vectors
   a. Represented as NumPy arrays with shape (3, 1) for RGB values
   b. Example: **vector = np.array([[R], [G], [B]])** where R, G, B are integers 0-255

B. Matrices
   a. Image matrix: NumPy array with shape (n, n, 3) where each element is an RGB vector
   b. Key matrices: NumPy arrays with shape (n, n) for transformation operations
   c. Example: **image_matrix = np.zeros((n, n, 3))**

C. Transformations
   a. Matrix multiplication: **np.dot() or @**
   b. Matrix inversion: **np.linalg.det()**
   c. Determinants: **np.linalg.det()**
   d. Transposition: **np.transpose()**

Algorithmic Steps:

1. **Import Libraries**
   a. Import **numpy (NumPy)** for matrix operations and **PIL.Image (Pillow)** for image generation and manipulation

2. **Convert Text to Numeric Vectors**
   a. Read the input text file
   b. Map each character to a numeric value
   c. Group every three numeric values into a 3x1 vector representing the RGB color channels

3. **Form Matrix**
   a. Arrange vectors into an n x n matrix using NumPy

4. **Encryption Process using Linear Algebra**
   a. Apply matrix operations such as matrix multiplication, matrix transposition, and determinant-based transformations to shuffle and alter the vectors
   b. Store each transformation step in a key that serves as the decryption guide

5. **Image Generation**
   a. Convert the encrypted matrix into an RGB image using Pillow and save it alongside the key

6. **Decryption**
   a. Take the encrypted image and key as inputs
   b. Reverse each matrix operation in the order stored in the key

c. Reconstruct the vectors back to characters and output

Expected Inputs/Outputs:

- **For Encryption**
    - Input: .txt file containing the text data, which will only contain a selected alphabet a-z capitalized, newlines, spaces, and punctuation.
    - Output: Encrypted .png image and decryption key

- **For Decryption**
    - Input: Encrypted .png image and decryption key
    - Output: Decrypted .txt file or console output of the decrypted message

Testing Plan:

- **Test Invertibility**
    - Generate random key matrix K
    - Verify if det(K) ≠ 0 using np.linalg.det(K) != 0
    - Confirm K @ inv(K) ≈ identity_matrix

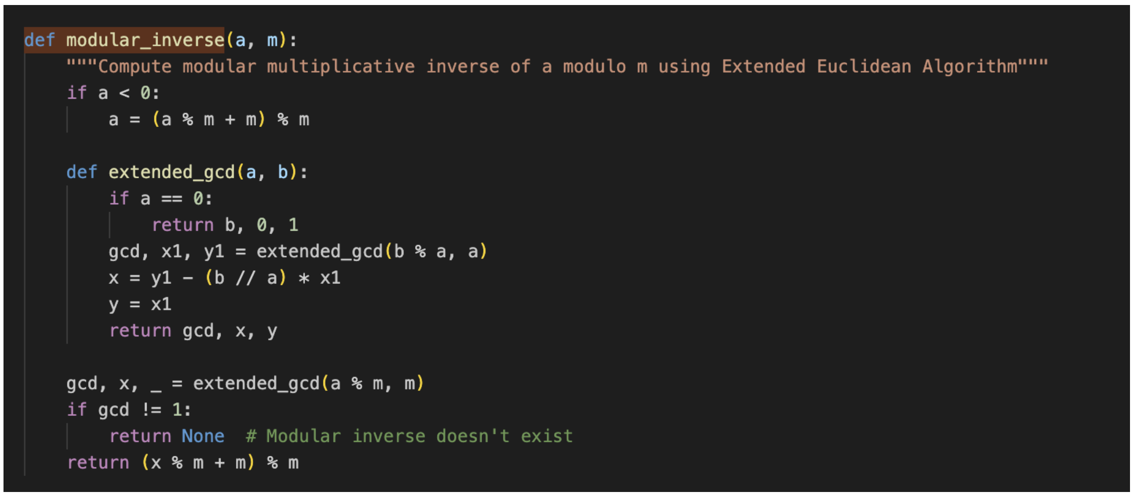
- **Test Determinant Properties**
    - Verify det(K @ T) = det(K) * det(T) for random matrices
    - Test that det = 0 singular matrices fail encryption

- **Test Vector Space Properties**
    - Test linearity: encrypt(I1 + I2, K) == encrypt(I1, K) + encrypt(I2, K)
    - Verify scalar multiplication: encrypt(c*I1, K) = c * encrypt(I1, K)

- **Test Equality**
    - Encrypt text and generate encrypted matrix *E*
    - Decrypt *E* to recover the original text
    - Verify equality

Code Implementation:

File: decryption.py



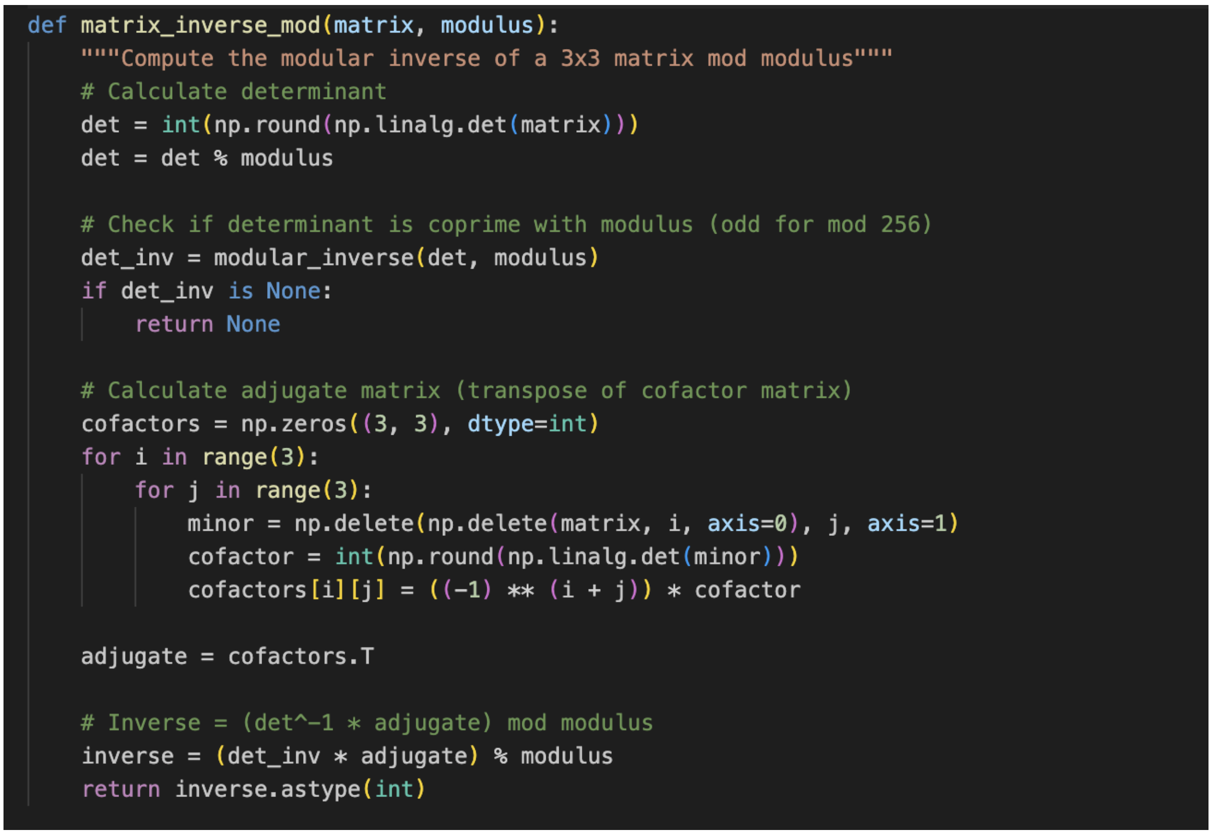
```python
def modular_inverse(a, m):
    """Compute modular multiplicative inverse of a modulo m using Extended Euclidean Algorithm"""
    if a < 0:
        a = (a % m + m) % m

    def extended_gcd(a, b):
        if a == 0:
            return b, 0, 1
        gcd, x1, y1 = extended_gcd(b % a, a)
        x = y1 - (b // a) * x1
        y = x1
        return gcd, x, y

    gcd, x, _ = extended_gcd(a % m, m)
    if gcd != 1:
        return None  # Modular inverse doesn't exist
    return (x % m + m) % m
```

Modular Inverse Calculation

Purpose: Computes the modular multiplicative inverse of a number of a and then under the modulus m. This is important since it helps track the inverse of the encryption key matrix under modular

Mathematical operation: Locates an integer a^-1 such that **(a x a^-1) mod m = 1**

Expected result and brief interpretation: if the modular inverse exists, then the function returns a^-1. If not, then the ciphertext cannot be decrypted.



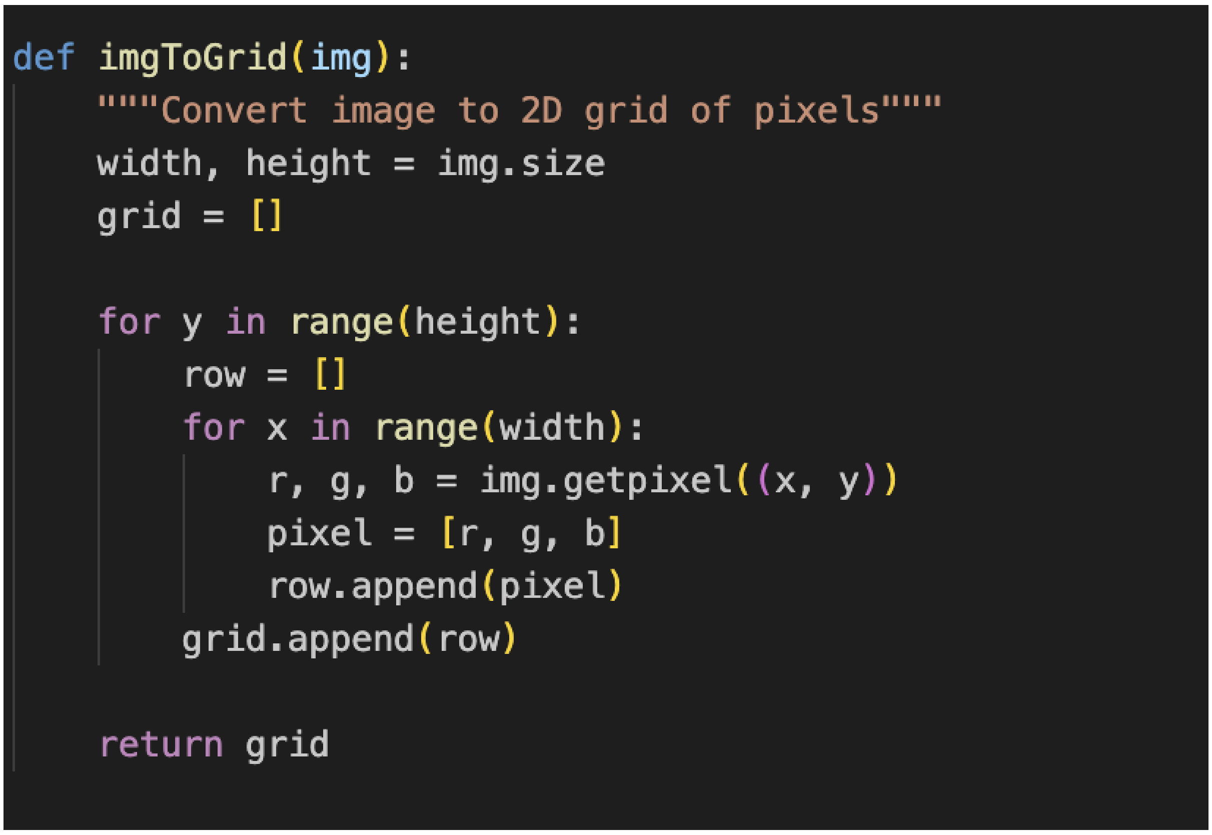
```python
def matrix_inverse_mod(matrix, modulus):
    """Compute the modular inverse of a 3x3 matrix mod modulus"""
    # Calculate determinant
    det = int(np.round(np.linalg.det(matrix)))
    det = det % modulus

    # Check if determinant is coprime with modulus (odd for mod 256)
    det_inv = modular_inverse(det, modulus)
    if det_inv is None:
        return None

    # Calculate adjugate matrix (transpose of cofactor matrix)
    cofactors = np.zeros((3, 3), dtype=int)
    for i in range(3):
        for j in range(3):
            minor = np.delete(np.delete(matrix, i, axis=0), j, axis=1)
            cofactor = int(np.round(np.linalg.det(minor)))
            cofactors[i][j] = ((-1) ** (i + j)) * cofactor

    adjugate = cofactors.T

    # Inverse = (det^-1 * adjugate) mod modulus
    inverse = (det_inv * adjugate) % modulus
    return inverse.astype(int)
```

Matrix Inversion Modulo 256

Purpose: Computes the **inverse of the key matrix K^-1** under modulo 256 arithmetic, which is used to reverse the linear transformation applied during encryption.

Mathematical Operation: Given the encrypted matrix **E = K x I mod 256**
The decryption retrieves **I** via: **I = K^-1 x E mod 256**

Expected result and brief explanation: Creates an integer-valued matrix that whenever it is multiplied with the encrypted data, it then restores the original pixel vectors.



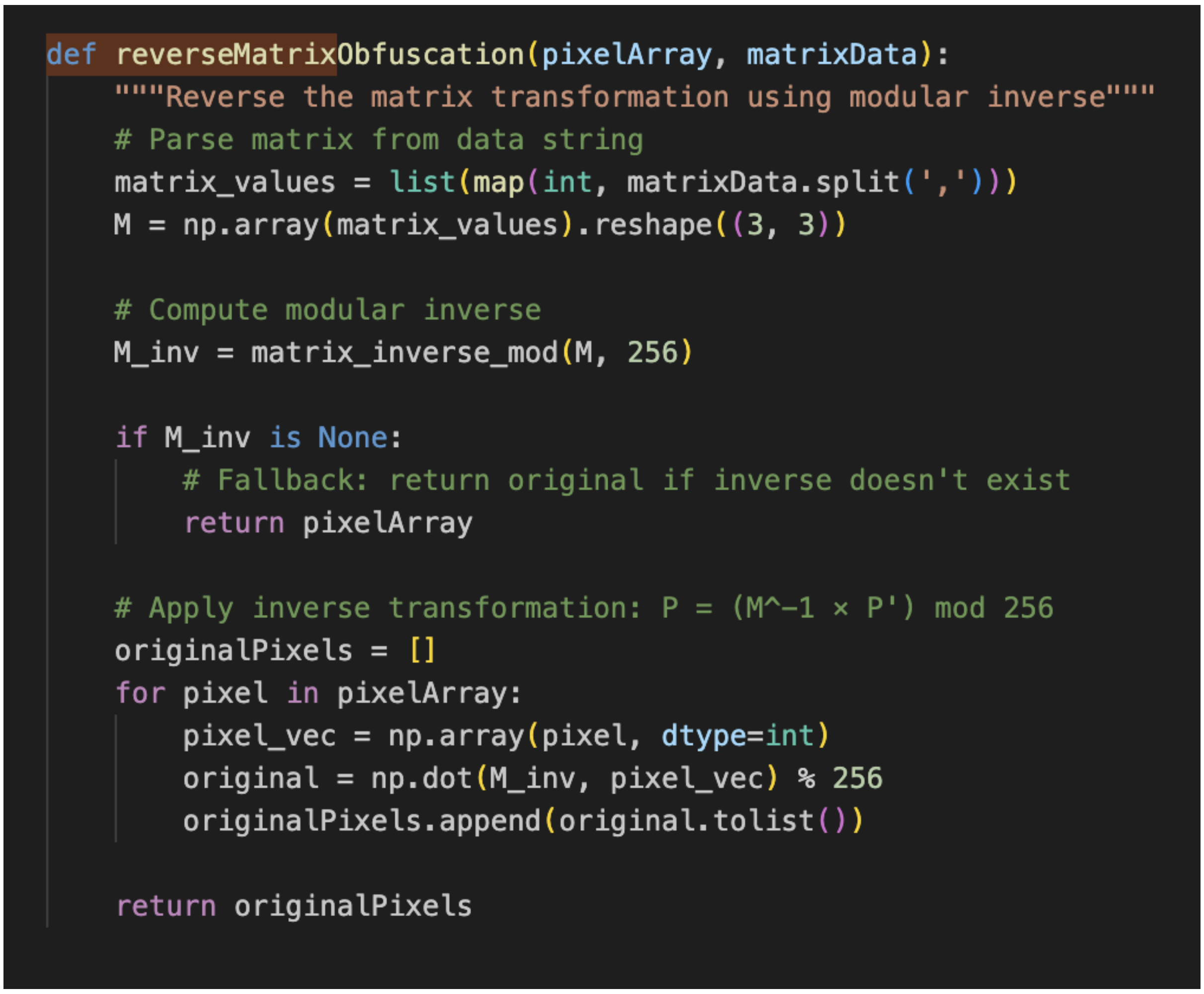
```python
def imgToGrid(img):
    """Convert image to 2D grid of pixels"""
    width, height = img.size
    grid = []

    for y in range(height):
        row = []
        for x in range(width):
            r, g, b = img.getpixel((x, y))
            pixel = [r, g, b]
            row.append(pixel)
        grid.append(row)

    return grid
```

Image-to-Grid Conversion

Purpose: Shapes the image into a 2D matrix of RGB pixel vectors

Mathematical Operation: Represents the encrypted image as matrix E, where each element **E sub ij = [R,G,B]** ∈ **R^3**

Expected result and brief explanation: The image becomes an n x n matrix where each pixel is treated as a 3D vector.



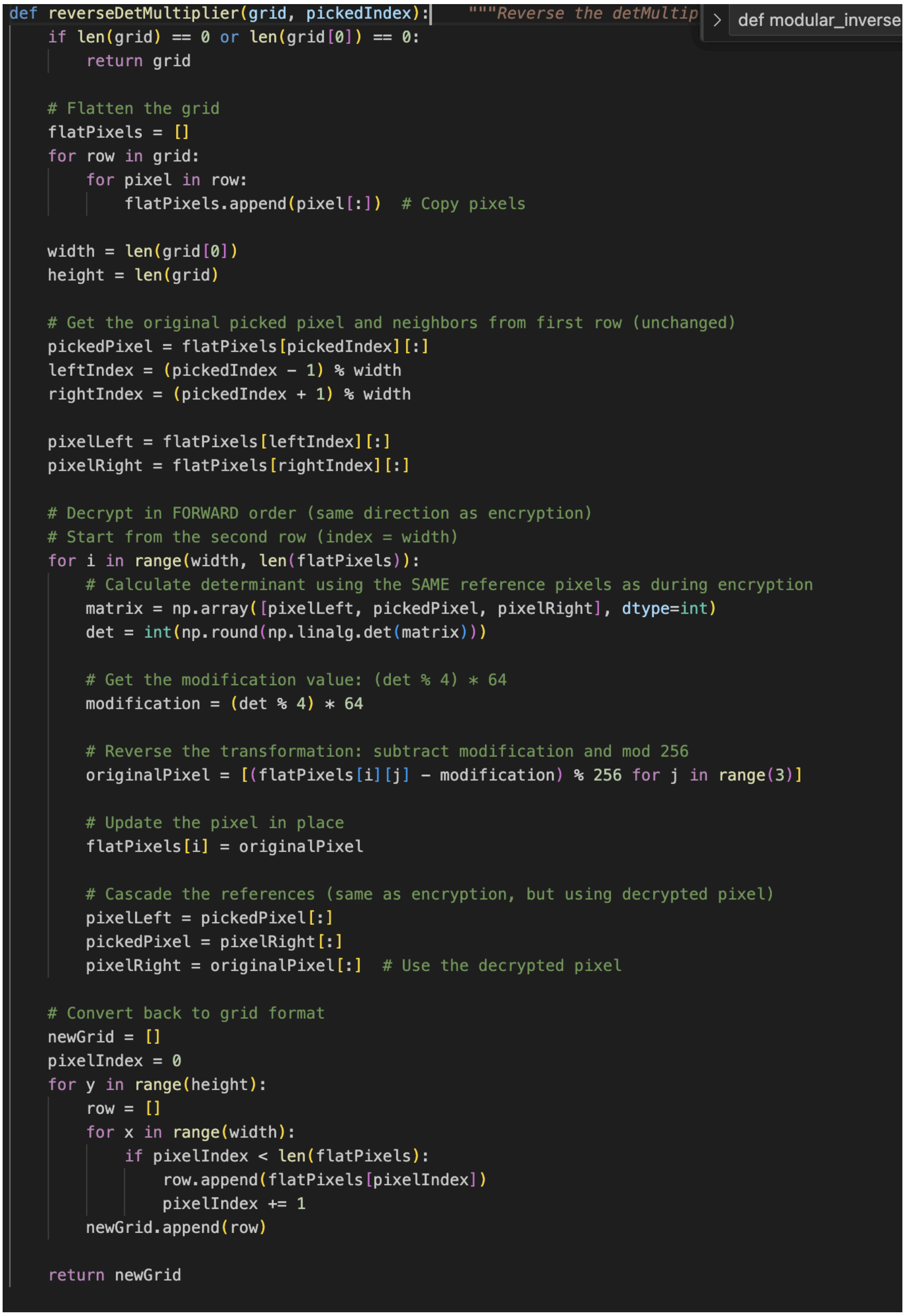
```python
def reverseMatrixObfuscation(pixelArray, matrixData):
    """Reverse the matrix transformation using modular inverse"""
    # Parse matrix from data string
    matrix_values = list(map(int, matrixData.split(',')))
    M = np.array(matrix_values).reshape((3, 3))

    # Compute modular inverse
    M_inv = matrix_inverse_mod(M, 256)

    if M_inv is None:
        # Fallback: return original if inverse doesn't exist
        return pixelArray

    # Apply inverse transformation: P = (M^-1 × P') mod 256
    originalPixels = []
    for pixel in pixelArray:
        pixel_vec = np.array(pixel, dtype=int)
        original = np.dot(M_inv, pixel_vec) % 256
        originalPixels.append(original.tolist())

    return originalPixels
```

Reverse Matrix Obfuscation

Purpose: Applies the inverse matrix to each encrypted pixel vector to recover its original RGB values

Mathematical Operation: If the encryption was **E = K x I,** the decryption computes:
I = K ^-1 x E mod 256

Expected result and brief explanation: Restores the original image matrix by applying the modular inverse of the key matrix to all pixel vectors.



```
def reverseDetMultiplier(grid, pickedIndex):    """Reverse the detMultip    > def modular_inverse
    if len(grid) == 0 or len(grid[0]) == 0:
        return grid

    # Flatten the grid
    flatPixels = []
    for row in grid:
        for pixel in row:
            flatPixels.append(pixel[:])  # Copy pixels

    width = len(grid[0])
    height = len(grid)

    # Get the original picked pixel and neighbors from first row (unchanged)
    pickedPixel = flatPixels[pickedIndex][:]
    leftIndex = (pickedIndex - 1) % width
    rightIndex = (pickedIndex + 1) % width

    pixelLeft = flatPixels[leftIndex][:]
    pixelRight = flatPixels[rightIndex][:]

    # Decrypt in FORWARD order (same direction as encryption)
    # Start from the second row (index = width)
    for i in range(width, len(flatPixels)):
        # Calculate determinant using the SAME reference pixels as during encryption
        matrix = np.array([pixelLeft, pickedPixel, pixelRight], dtype=int)
        det = int(np.round(np.linalg.det(matrix)))

        # Get the modification value: (det % 4) * 64
        modification = (det % 4) * 64

        # Reverse the transformation: subtract modification and mod 256
        originalPixel = [(flatPixels[i][j] - modification) % 256 for j in range(3)]

        # Update the pixel in place
        flatPixels[i] = originalPixel

        # Cascade the references (same as encryption, but using decrypted pixel)
        pixelLeft = pickedPixel[:]
        pickedPixel = pixelRight[:]
        pixelRight = originalPixel[:]  # Use the decrypted pixel

    # Convert back to grid format
    newGrid = []
    pixelIndex = 0
    for y in range(height):
        row = []
        for x in range(width):
            if pixelIndex < len(flatPixels):
                row.append(flatPixels[pixelIndex])
                pixelIndex += 1
        newGrid.append(row)

    return newGrid
```

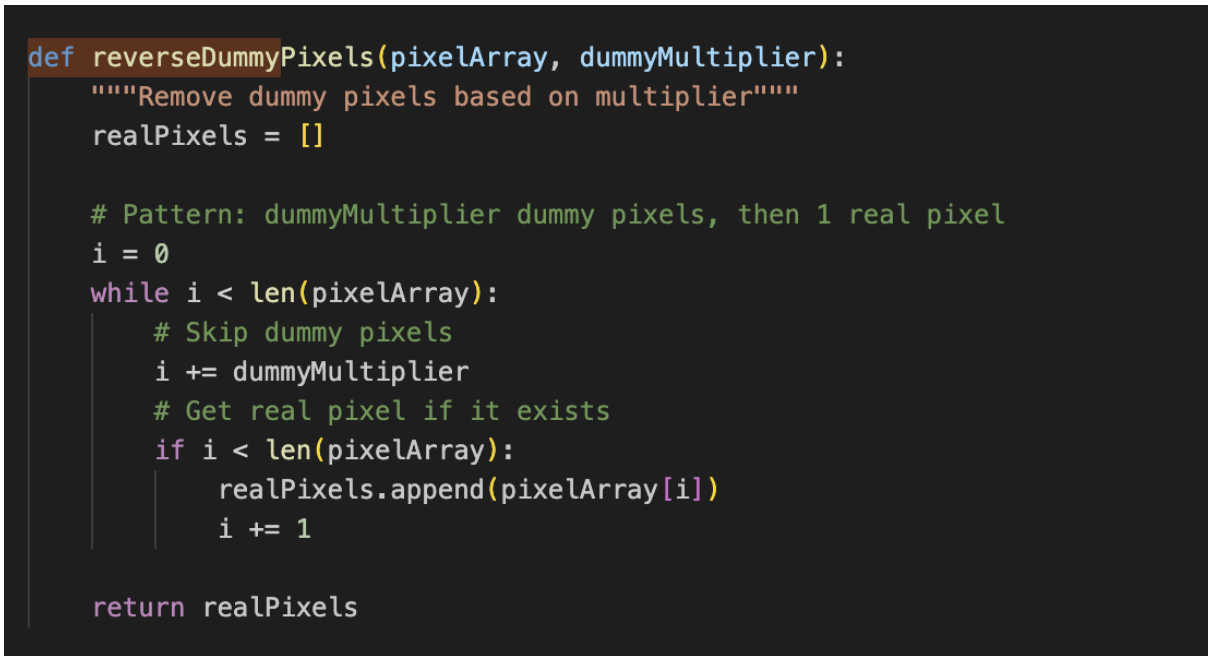
Reverse Determinant-Based Modification

Purpose: It reverses the determinant-based pixel value modification applied during encryption.

Mathematical Operation: During encryption, each pixel's value was altered based on: **modification = (det(M) mod 4) x 64**

Where **M** is a matrix formed by three neighboring pixels. The decryption subtracts this modification: **P sub original = (P sub encrypted - modification) mod 256**

Expected result and brief explanation: Reshapes the original pixel color values by removing determinant-based offsets.



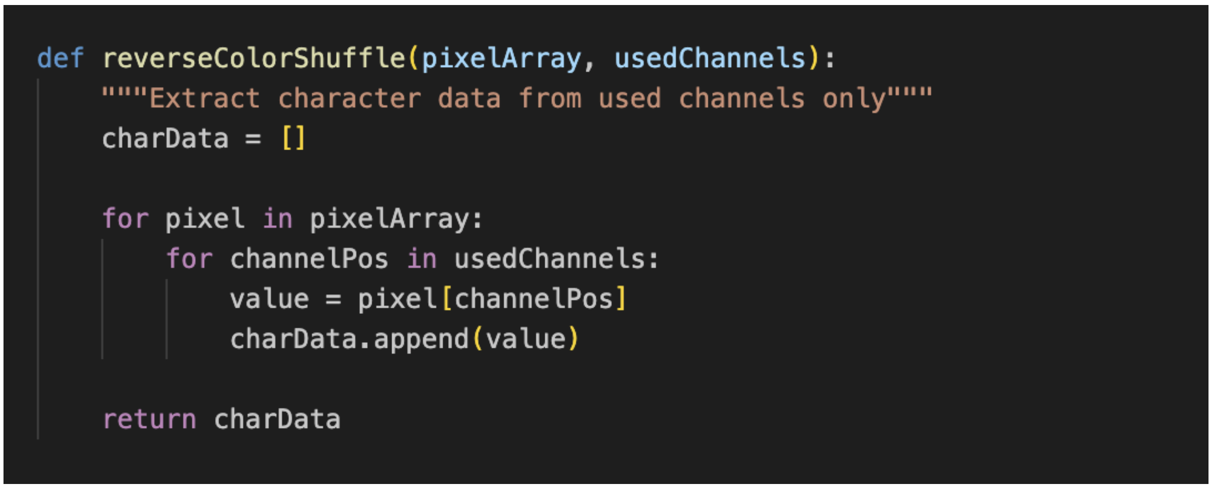
```python
def reverseDummyPixels(pixelArray, dummyMultiplier):
    """Remove dummy pixels based on multiplier"""
    realPixels = []

    # Pattern: dummyMultiplier dummy pixels, then 1 real pixel
    i = 0
    while i < len(pixelArray):
        # Skip dummy pixels
        i += dummyMultiplier
        # Get real pixel if it exists
        if i < len(pixelArray):
            realPixels.append(pixelArray[i])
            i += 1

    return realPixels
```

Reverse Dummy Pixel Removal

Purpose: Removes extra dummy pixels inserted to confuse attackers during encryption

Mathematical Operation: If D dummy pixels were added after every real pixel, this is reversed by keeping every **(D + 1)th** pixel

Expected result and brief explanation: Retrieves the actual data-carrying pixels only.



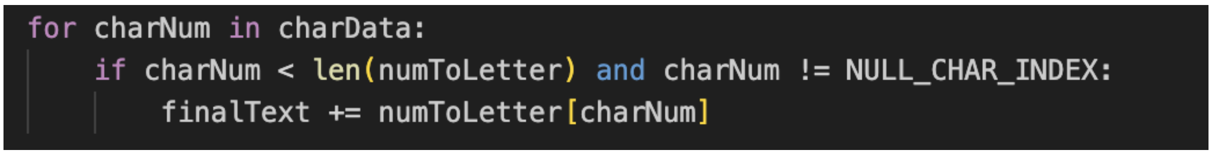
```python
def reverseColorShuffle(pixelArray, usedChannels):
    """Extract character data from used channels only"""
    charData = []

    for pixel in pixelArray:
        for channelPos in usedChannels:
            value = pixel[channelPos]
            charData.append(value)

    return charData
```

Reverse Color Channel Extraction

Purpose: Restores the correct character data by using the same channel order **(R, G, B)** chosen during encryption.

Mathematical Operation: Extracts the values from the pixel vector components based on selected indices.

Expected result and brief explanation: Produces a 1D list of numerical character representations **(ASCII indices)**



```python
for charNum in charData:
    if charNum < len(numToLetter) and charNum != NULL_CHAR_INDEX:
        finalText += numToLetter[charNum]
```

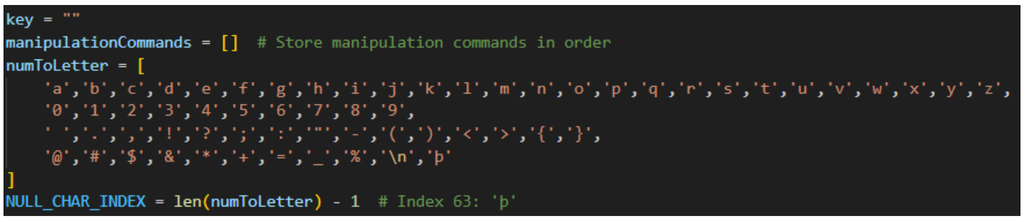
Reconstructing the Decrypted Text

Purpose: Maps numerical values back to their corresponding characters.

Mathematical Operation: Applies a bijective mapping **f^-1 : Z→Σ,** where **Σ** is the character alphabet

Expected result and brief explanation: the recovered plaintext message is printed as the decrypted output

File: encryption.py



```python
key = ""
manipulationCommands = []  # Store manipulation commands in order
numToLetter = [
    'a','b','c','d','e','f','g','h','i','j','k','l','m','n','o','p','q','r','s','t','u','v','w','x','y','z',
    '0','1','2','3','4','5','6','7','8','9',
    ' ','.',',','!','?',';',':','"','-','(',')','<','>','{','}',
    '@','#','$','&','*','+','=','_','%','\n','þ'
]
NULL_CHAR_INDEX = len(numToLetter) - 1  # Index 63: 'þ'
```

Global encoding and key state

Purpose:

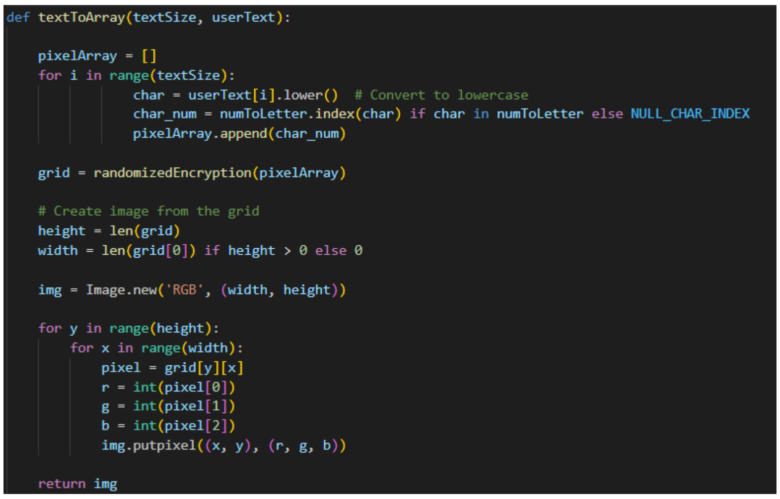
- Globals for reversible encoding and key logging.

Mathematical operations:

- numToLetter defines $f: \Sigma \to \{0,\dots,63\}$.
- key and manipulationCommands serialize the composition of transforms so we can build T^{-1}.

Expected result and brief interpretation:

- Deterministic symbol space and a reproducible key for perfect decryption.



```python
def textToArray(textSize, userText):

    pixelArray = []
    for i in range(textSize):
            char = userText[i].lower()  # Convert to lowercase
            char_num = numToLetter.index(char) if char in numToLetter else NULL_CHAR_INDEX
            pixelArray.append(char_num)

    grid = randomizedEncryption(pixelArray)

    # Create image from the grid
    height = len(grid)
    width = len(grid[0]) if height > 0 else 0

    img = Image.new('RGB', (width, height))

    for y in range(height):
        for x in range(width):
            pixel = grid[y][x]
            r = int(pixel[0])
            g = int(pixel[1])
            b = int(pixel[2])
            img.putpixel((x, y), (r, g, b))

    return img
```

Character encoding → RGB vectors (Vectors)

Purpose: Encode plaintext characters into integer symbols and pack them into RGB pixel vectors, then run the randomized encryption pipeline to produce an image.

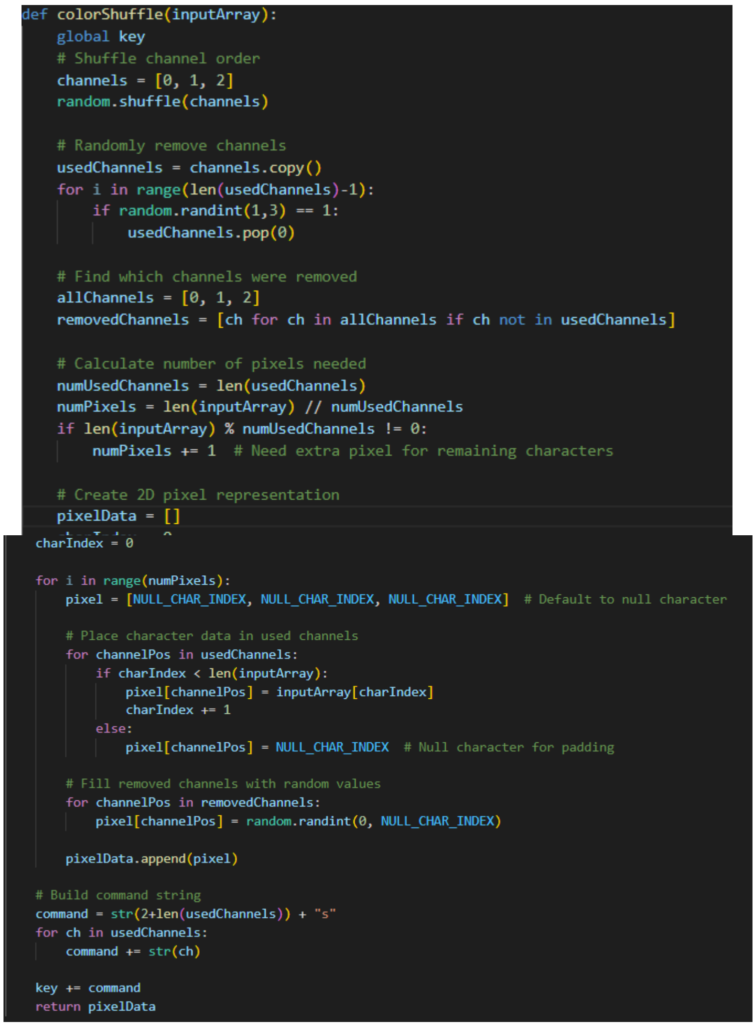
Mathematical operation:

- Character mapping $f: \Sigma \to \{0,\dots,63\}$
- Vectorization (Figure 1.1): for each pixel $v = [R, G, B]^T$ where entries are $f(C_i)$.
- The pipeline later treats the list of v as rows of an image matrix I.

Expected result and brief interpretation:

- Returns a PIL image whose pixels contain the encoded and encrypted version of the text.

- Each pixel is a 3×1 vector carrying up to three characters in its channels.



```python
def colorShuffle(inputArray):
    global key
    # Shuffle channel order
    channels = [0, 1, 2]
    random.shuffle(channels)

    # Randomly remove channels
    usedChannels = channels.copy()
    for i in range(len(usedChannels)-1):
        if random.randint(1,3) == 1:
            usedChannels.pop(0)

    # Find which channels were removed
    allChannels = [0, 1, 2]
    removedChannels = [ch for ch in allChannels if ch not in usedChannels]

    # Calculate number of pixels needed
    numUsedChannels = len(usedChannels)
    numPixels = len(inputArray) // numUsedChannels
    if len(inputArray) % numUsedChannels != 0:
        numPixels += 1  # Need extra pixel for remaining characters

    # Create 2D pixel representation
    pixelData = []
    charIndex = 0

    for i in range(numPixels):
        pixel = [NULL_CHAR_INDEX, NULL_CHAR_INDEX, NULL_CHAR_INDEX]  # Default to null character

        # Place character data in used channels
        for channelPos in usedChannels:
            if charIndex < len(inputArray):
                pixel[channelPos] = inputArray[charIndex]
                charIndex += 1
            else:
                pixel[channelPos] = NULL_CHAR_INDEX  # Null character for padding

        # Fill removed channels with random values
        for channelPos in removedChannels:
            pixel[channelPos] = random.randint(0, NULL_CHAR_INDEX)

        pixelData.append(pixel)

    # Build command string
    command = str(2+len(usedChannels)) + "s"
    for ch in usedChannels:
        command += str(ch)

    key += command
    return pixelData
```

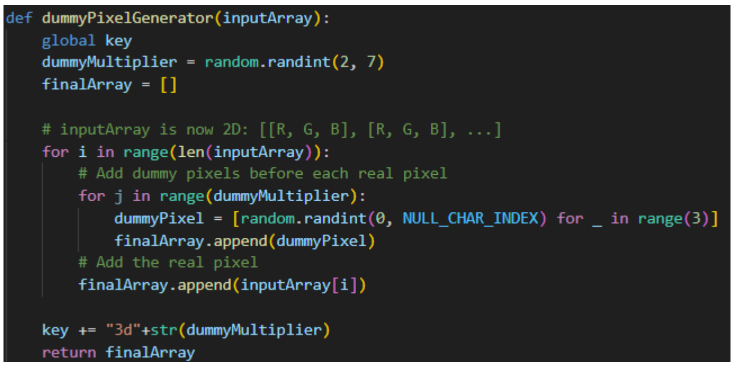
Packing symbols into RGB channels (Vectors)

Purpose:
Pack the stream of symbol values into RGB vectors using a random subset/order of channels, with noise in the unused channels.
Mathematical operation:

- Construct pixel vectors v = [R, G, B]^T (Figure 1.1).
- For data layout operation: It prepares columns v_i□ that later populate I (see Figure 1.5).

Expected result and brief interpretation:

- Produces a list of 3D vectors where chosen channels carry data and others carry noise.
- The command 's' records which channels are used so the mapping is reversible.



```python
def dummyPixelGenerator(inputArray):
    global key
    dummyMultiplier = random.randint(2, 7)
    finalArray = []

    # inputArray is now 2D: [[R, G, B], [R, G, B], ...]
    for i in range(len(inputArray)):
        # Add dummy pixels before each real pixel
        for j in range(dummyMultiplier):
            dummyPixel = [random.randint(0, NULL_CHAR_INDEX) for _ in range(3)]
            finalArray.append(dummyPixel)
        # Add the real pixel
        finalArray.append(inputArray[i])

    key += "3d"+str(dummyMultiplier)
    return finalArray
```

Dummy pixels (obfuscation layer)

Purpose:

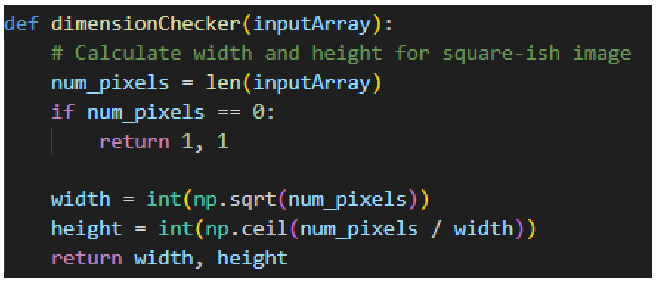
- Insert n dummy pixel vectors before each real pixel to hide structure and inflate size.

Mathematical operations:

- No new algebraic transformation is used; the program simply mixes random 3×1 vectors in between the actual data vectors.

Expected result and brief interpretation:

- Returns a longer list of 3D vectors; the key ('d') stores N so decryption can strip dummies.



```python
def dimensionChecker(inputArray):
    # Calculate width and height for square-ish image
    num_pixels = len(inputArray)
    if num_pixels == 0:
        return 1, 1

    width = int(np.sqrt(num_pixels))
    height = int(np.ceil(num_pixels / width))
    return width, height
```

Dimension Checker (layout helper)

Purpose:

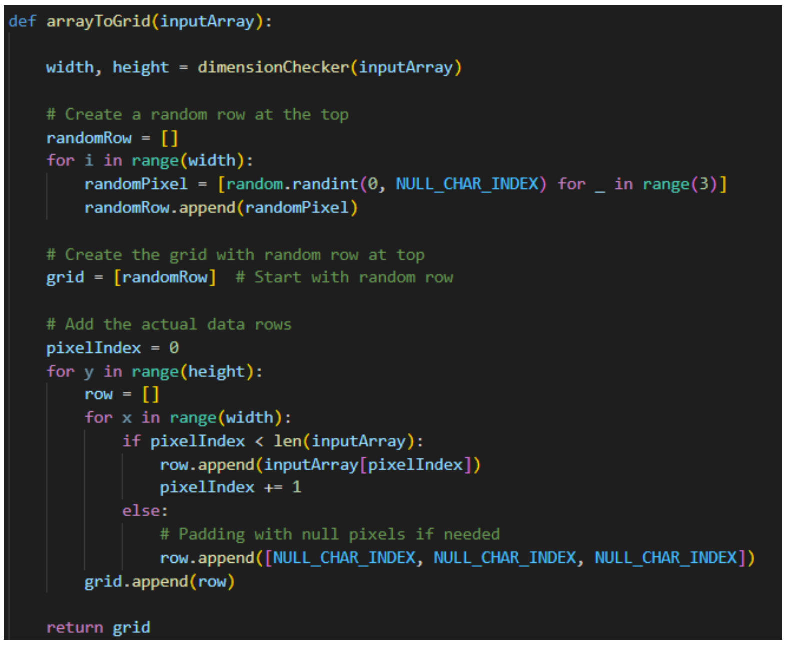
- Calculate width and height for square-ish image

Mathematical operations:

- Prepares the shape for the image matrix I (Figure 1.5).

Expected result and brief interpretation:

- (width, height) that packs all pixels with minimal padding.



```python
def arrayToGrid(inputArray):

    width, height = dimensionChecker(inputArray)

    # Create a random row at the top
    randomRow = []
    for i in range(width):
        randomPixel = [random.randint(0, NULL_CHAR_INDEX) for _ in range(3)]
        randomRow.append(randomPixel)

    # Create the grid with random row at top
    grid = [randomRow]  # Start with random row

    # Add the actual data rows
    pixelIndex = 0
    for y in range(height):
        row = []
        for x in range(width):
            if pixelIndex < len(inputArray):
                row.append(inputArray[pixelIndex])
                pixelIndex += 1
            else:
                # Padding with null pixels if needed
                row.append([NULL_CHAR_INDEX, NULL_CHAR_INDEX, NULL_CHAR_INDEX])
        grid.append(row)

    return grid
```

Arrange pixels into a 2D grid I (Matrices)

Purpose:

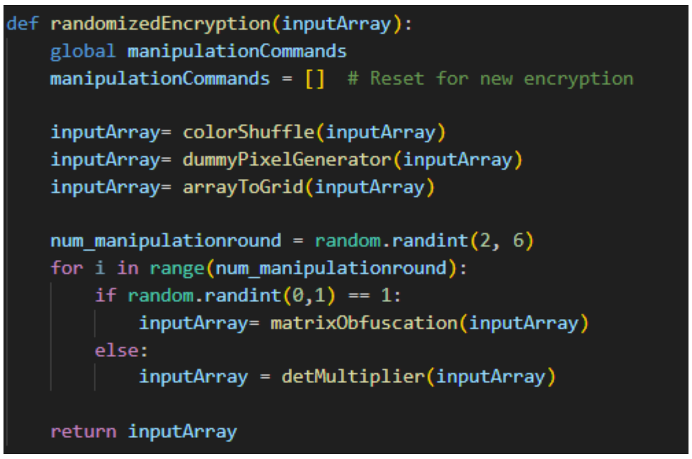
- Reshape the 1D stream of vectors into a near-rectangular grid and add a random top row.

Mathematical operations:

- Forms the image matrix I in Figure 1.5, whose entries are 3×1 vectors v_i□.
- The extra random first row is excluded from some transforms and supports the cascade step.

Expected result and brief interpretation:

- Returns a list-of-rows grid (height × width) that models I; the top row is random padding.



```python
def randomizedEncryption(inputArray):
    global manipulationCommands
    manipulationCommands = []  # Reset for new encryption

    inputArray= colorShuffle(inputArray)
    inputArray= dummyPixelGenerator(inputArray)
    inputArray= arrayToGrid(inputArray)

    num_manipulationround = random.randint(2, 6)
    for i in range(num_manipulationround):
        if random.randint(0,1) == 1:
            inputArray= matrixObfuscation(inputArray)
        else:
            inputArray = detMultiplier(inputArray)

    return inputArray
```

Randomized encryption pipeline (orchestration)

Purpose:

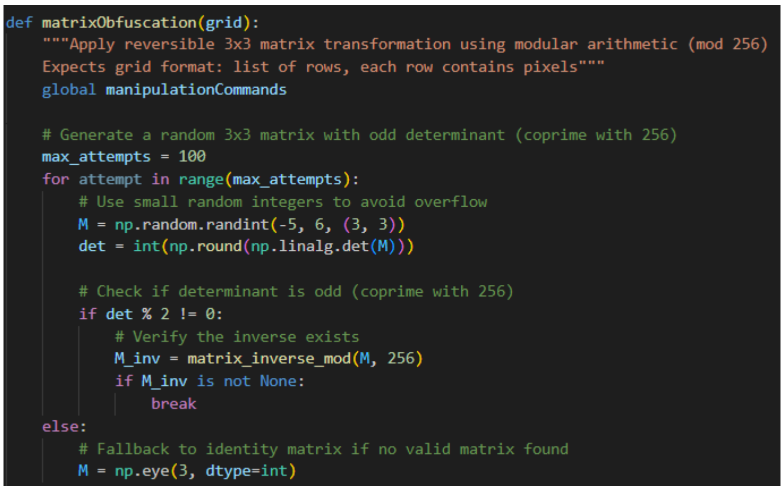
- Apply the sequence: channel packing → dummy insertion → grid formation → 2–6 rounds of reversible transforms (either linear transform by a 3×3 K or a determinant-based cascade).

Mathematical operations:

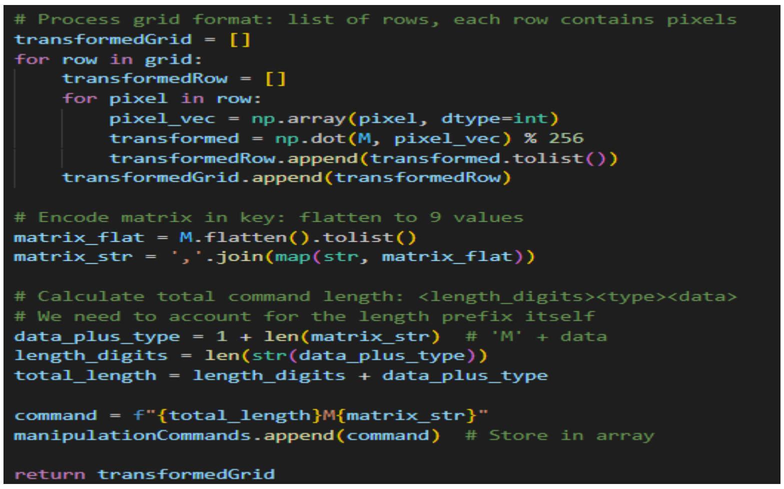
- Vector preparation (Figures 1.1, 1.5).
- Linear map on each pixel: E_pixel = K v mod 256 (Figure 1.2, at pixel level).
- The optional determinant cascade adds a scalar shift to each channel based on det of nearby pixels.

Expected result and brief interpretation:

- Returns a transformed grid ready to be written as an image, plus a key that records steps.



```python
def matrixObfuscation(grid):
    """Apply reversible 3x3 matrix transformation using modular arithmetic (mod 256)
    Expects grid format: list of rows, each row contains pixels"""
    global manipulationCommands

    # Generate a random 3x3 matrix with odd determinant (coprime with 256)
    max_attempts = 100
    for attempt in range(max_attempts):
        # Use small random integers to avoid overflow
        M = np.random.randint(-5, 6, (3, 3))
        det = int(np.round(np.linalg.det(M)))

        # Check if determinant is odd (coprime with 256)
        if det % 2 != 0:
            # Verify the inverse exists
            M_inv = matrix_inverse_mod(M, 256)
            if M_inv is not None:
                break
    else:
        # Fallback to identity matrix if no valid matrix found
        M = np.eye(3, dtype=int)
```



```python
# Process grid format: list of rows, each row contains pixels
transformedGrid = []
for row in grid:
    transformedRow = []
    for pixel in row:
        pixel_vec = np.array(pixel, dtype=int)
        transformed = np.dot(M, pixel_vec) % 256
        transformedRow.append(transformed.tolist())
    transformedGrid.append(transformedRow)

# Encode matrix in key: flatten to 9 values
matrix_flat = M.flatten().tolist()
matrix_str = ','.join(map(str, matrix_flat))

# Calculate total command length: <length_digits><type><data>
# We need to account for the length prefix itself
data_plus_type = 1 + len(matrix_str)  # 'M' + data
length_digits = len(str(data_plus_type))
total_length = length_digits + data_plus_type

command = f"{total_length}M{matrix_str}"
manipulationCommands.append(command)  # Store in array

return transformedGrid
```

Linear transform per pixel via 3×3 key K (Matrices, Determinants)

Purpose:

- Encrypt each pixel vector v by a 3×3 integer key matrix K modulo 256.

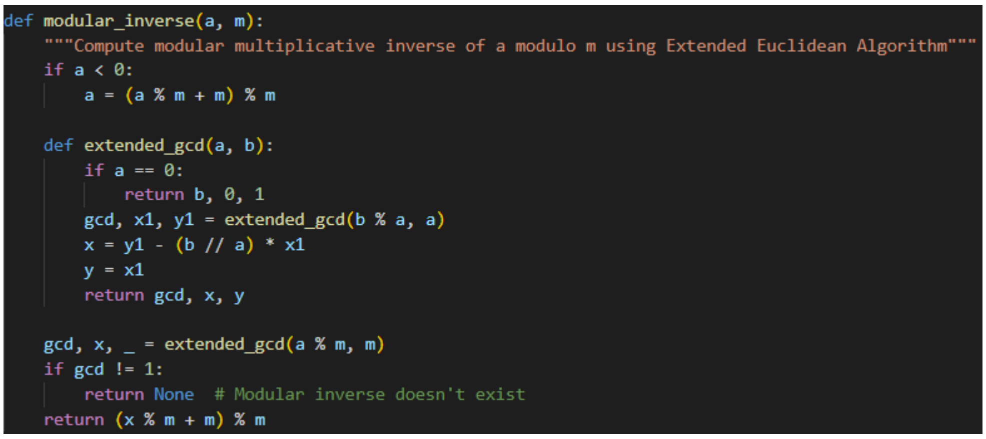
Mathematical operations:

- Choose K with det(K) odd ⇒ gcd(det(K), 256)=1 ⇒ K is invertible mod 256 (Figure 1.4).
- For each pixel v ∈ Z_256^3:

- E = K v (mod 256)   (Pixel-level instance of Figure 1.2)
- The inverse step in decryption uses K^{-1} mod 256 (Figure 1.3).

Expected result and brief interpretation:

- Produces visually scrambled pixel values while preserving perfect invertibility.
- The key records K so that decryption can apply K^{-1}.



```python
def modular_inverse(a, m):
    """Compute modular multiplicative inverse of a modulo m using Extended Euclidean Algorithm"""
    if a < 0:
        a = (a % m + m) % m

    def extended_gcd(a, b):
        if a == 0:
            return b, 0, 1
        gcd, x1, y1 = extended_gcd(b % a, a)
        x = y1 - (b // a) * x1
        y = x1
        return gcd, x, y

    gcd, x, _ = extended_gcd(a % m, m)
    if gcd != 1:
        return None  # Modular inverse doesn't exist
    return (x % m + m) % m
```

Matrix inverse and modular inverse (Determinants, Invertibility)

Purpose:

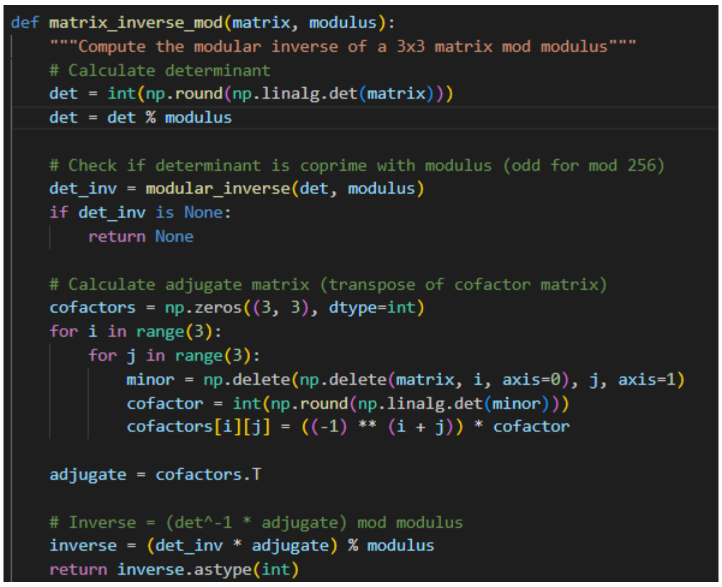
- Compute x such that a·x ≡ 1 (mod m) using the Extended Euclidean Algorithm.

Mathematical operations:

- Solve ax + my = gcd(a, m); when gcd(a, m)=1, x ≡ a^{-1} (mod m).

Expected result and brief interpretation:

- Returns a^{-1} mod m if it exists; None otherwise. Used to ensure det(K) is invertible mod 256.



```python
def matrix_inverse_mod(matrix, modulus):
    """Compute the modular inverse of a 3x3 matrix mod modulus"""
    # Calculate determinant
    det = int(np.round(np.linalg.det(matrix)))
    det = det % modulus

    # Check if determinant is coprime with modulus (odd for mod 256)
    det_inv = modular_inverse(det, modulus)
    if det_inv is None:
        return None

    # Calculate adjugate matrix (transpose of cofactor matrix)
    cofactors = np.zeros((3, 3), dtype=int)
    for i in range(3):
        for j in range(3):
            minor = np.delete(np.delete(matrix, i, axis=0), j, axis=1)
            cofactor = int(np.round(np.linalg.det(minor)))
            cofactors[i][j] = ((-1) ** (i + j)) * cofactor

    adjugate = cofactors.T

    # Inverse = (det^-1 * adjugate) mod modulus
    inverse = (det_inv * adjugate) % modulus
    return inverse.astype(int)
```

Matrix inverse and modular inverse (Determinants, Invertibility)

Purpose:

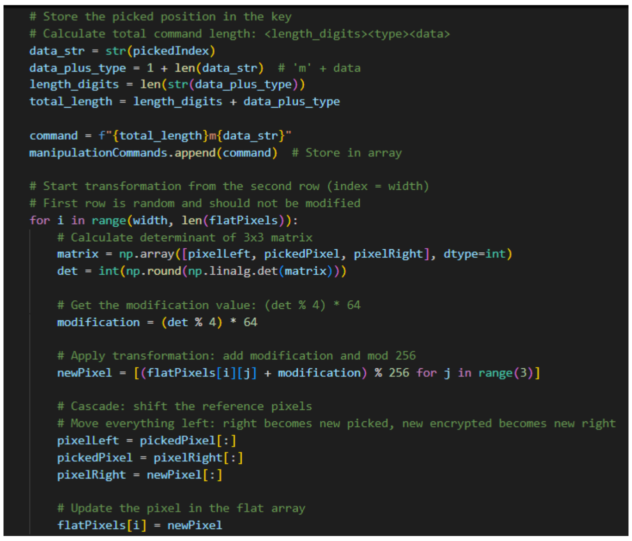
- Compute K^{-1} (mod 256) for a 3×3 integer matrix K.

Mathematical operations:

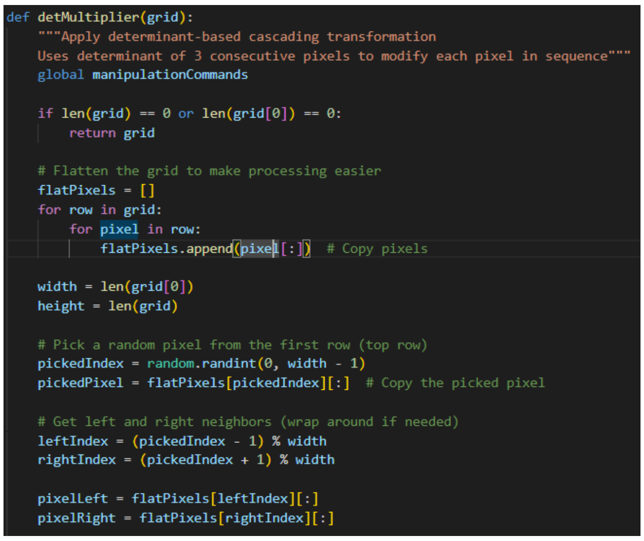
- detK = det(K); require gcd(detK, 256)=1 (Figure 1.4).
- K^{-1} ≡ det(K)^{-1} · adj(K) (mod 256).
- This is the algebraic inverse used in Figure 1.3 at the pixel level.

Expected result and brief interpretation:

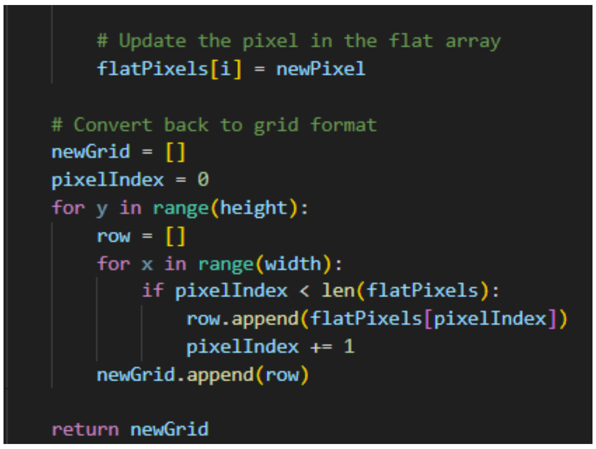
- Returns the 3×3 modular inverse if invertible; None otherwise (caller retries or uses I_3).



```python
    # Store the picked position in the key
    # Calculate total command length: <length_digits><type><data>
    data_str = str(pickedIndex)
    data_plus_type = 1 + len(data_str)  # 'm' + data
    length_digits = len(str(data_plus_type))
    total_length = length_digits + data_plus_type

    command = f"{total_length}m{data_str}"
    manipulationCommands.append(command)  # Store in array

    # Start transformation from the second row (index = width)
    # First row is random and should not be modified
    for i in range(width, len(flatPixels)):
        # Calculate determinant of 3x3 matrix
        matrix = np.array([pixelLeft, pickedPixel, pixelRight], dtype=int)
        det = int(np.round(np.linalg.det(matrix)))

        # Get the modification value: (det % 4) * 64
        modification = (det % 4) * 64

        # Apply transformation: add modification and mod 256
        newPixel = [(flatPixels[i][j] + modification) % 256 for j in range(3)]

        # Cascade: shift the reference pixels
        # Move everything left: right becomes new picked, new encrypted becomes new right
        pixelLeft = pickedPixel[:]
        pickedPixel = pixelRight[:]
        pixelRight = newPixel[:]

        # Update the pixel in the flat array
        flatPixels[i] = newPixel
```



```python
def detMultiplier(grid):
    """Apply determinant-based cascading transformation
    Uses determinant of 3 consecutive pixels to modify each pixel in sequence"""
    global manipulationCommands

    if len(grid) == 0 or len(grid[0]) == 0:
        return grid

    # Flatten the grid to make processing easier
    flatPixels = []
    for row in grid:
        for pixel in row:
            flatPixels.append(pixel[:])  # Copy pixels

    width = len(grid[0])
    height = len(grid)

    # Pick a random pixel from the first row (top row)
    pickedIndex = random.randint(0, width - 1)
    pickedPixel = flatPixels[pickedIndex][:]  # Copy the picked pixel

    # Get left and right neighbors (wrap around if needed)
    leftIndex = (pickedIndex - 1) % width
    rightIndex = (pickedIndex + 1) % width

    pixelLeft = flatPixels[leftIndex][:]
    pixelRight = flatPixels[rightIndex][:]
```



```python
        # Update the pixel in the flat array
        flatPixels[i] = newPixel

    # Convert back to grid format
    newGrid = []
    pixelIndex = 0
    for y in range(height):
        row = []
        for x in range(width):
            if pixelIndex < len(flatPixels):
                row.append(flatPixels[pixelIndex])
                pixelIndex += 1
        newGrid.append(row)

    return newGrid
```

Determinant-based cascading shift (Determinants)

Purpose:

- Add a cascading per-pixel shift determined by the determinant of a 3×3 matrix of neighboring pixel vectors from the first row/past steps.

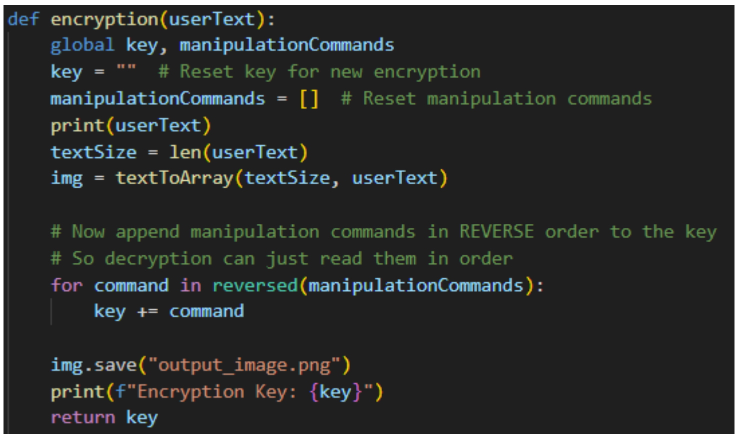
Mathematical operations:

- For each step, form M = [v_left; v_pick; v_right] (3×3, rows are pixel vectors).
- Compute s = (det(M) mod 4) * 64 ∈ {0,64,128,192}.
- Update pixel: E = (v + s·1) mod 256, where 1 is the all-ones vector in R^3.

This uses det(·) to derive a scalar from local structure; it is reversible by subtraction during decryption.

Expected result and brief interpretation:

- Diffuses information along the raster order using determinant-derived scalars.
- The key stores the starting index so the cascade can be replayed in reverse.



```python
def encryption(userText):
    global key, manipulationCommands
    key = ""  # Reset key for new encryption
    manipulationCommands = []  # Reset manipulation commands
    print(userText)
    textSize = len(userText)
    img = textToArray(textSize, userText)

    # Now append manipulation commands in REVERSE order to the key
    # So decryption can just read them in order
    for command in reversed(manipulationCommands):
        key += command

    img.save("output_image.png")
    print(f"Encryption Key: {key}")
    return key
```

Driver that assembles the final key and produces the image

Purpose:

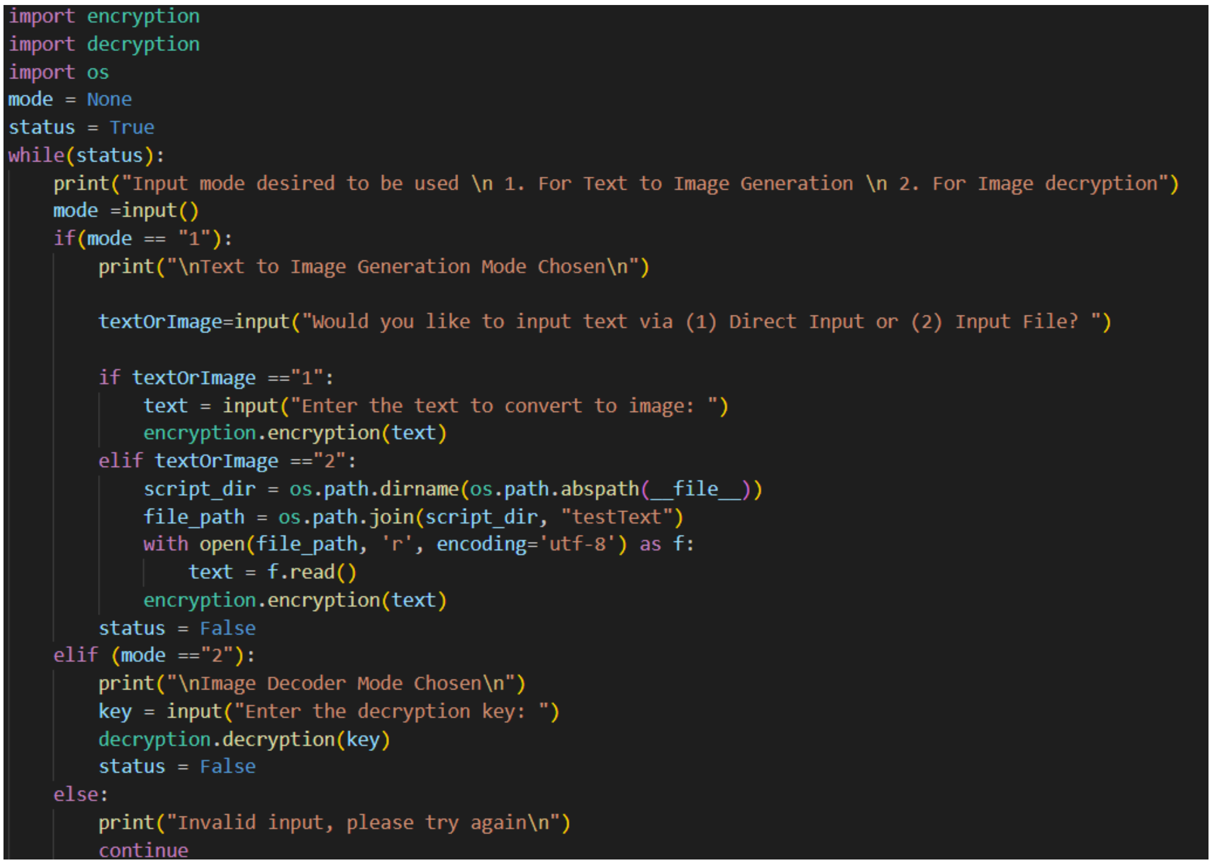
- End-to-end: encode text → build image grid → apply transforms → write PNG and emit key.

Mathematical operations:

- Aggregates the operations above:
- Build I (Figures 1.1, 1.5),
- Apply pixel-wise linear maps by K (Figure 1.2),
- Optionally apply the determinant cascade,
- K^{-1} will be used in decryption (Figure 1.3).

Expected result and brief interpretation:

- Produces 'output_image.png' and a key string that encodes the reversible steps.



```python
import encryption
import decryption
import os
mode = None
status = True
while(status):
    print("Input mode desired to be used \n 1. For Text to Image Generation \n 2. For Image decryption")
    mode =input()
    if(mode == "1"):
        print("\nText to Image Generation Mode Chosen\n")

        textOrImage=input("Would you like to input text via (1) Direct Input or (2) Input File? ")

        if textOrImage =="1":
            text = input("Enter the text to convert to image: ")
            encryption.encryption(text)
        elif textOrImage =="2":
            script_dir = os.path.dirname(os.path.abspath(__file__))
            file_path = os.path.join(script_dir, "testText")
            with open(file_path, 'r', encoding='utf-8') as f:
                text = f.read()
            encryption.encryption(text)
        status = False
    elif (mode =="2"):
        print("\nImage Decoder Mode Chosen\n")
        key = input("Enter the decryption key: ")
        decryption.decryption(key)
        status = False
    else:
        print("Invalid input, please try again\n")
        continue
```

File: main.py

Purpose:

- Command-line driver selecting encryption (text → image) or decryption (image → text).

Mathematical operations:

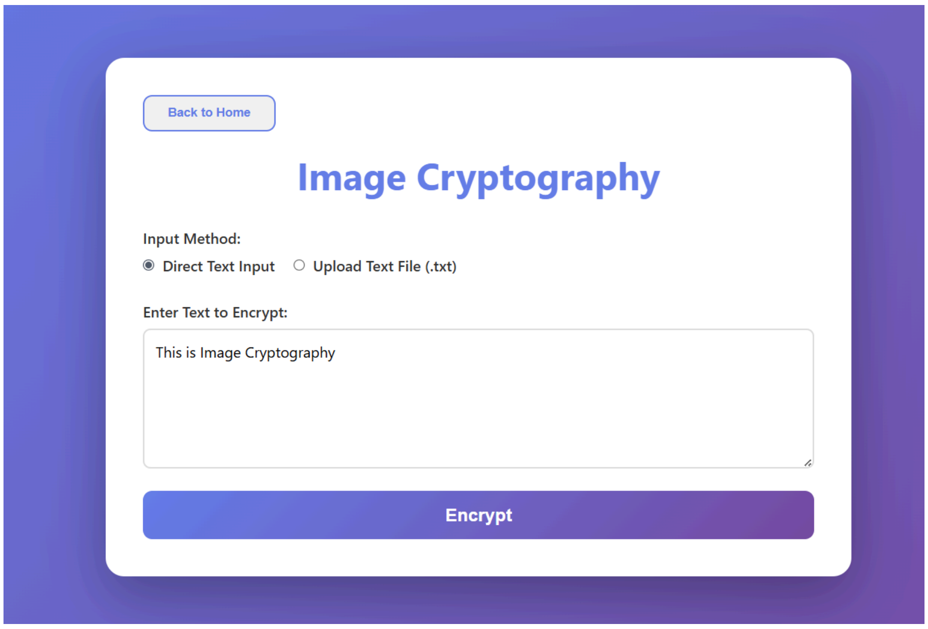
- If mode == 1(encryption): Builds plaintext symbol sequence, forms pixel vectors v = [R,G,B]^T, applies composed transforms T = (channel packing ◦ dummy insertion ◦ grid ◦ rounds of K·v (mod 256) or determinant cascade), producing the encrypted image.
- If mode == 2 (decryption): Applies inverse composition T^{-1} (reverse recorded steps) to recover original pixel vectors and decode characters.

Expected result and brief interpretation:

- Mode 1: Writes output_image.png and prints the key (serialization of T).
- Mode 2: Reads key, reconstructs plaintext exactly if key and image match.

Results and Discussion:

Mode 1: Encryption



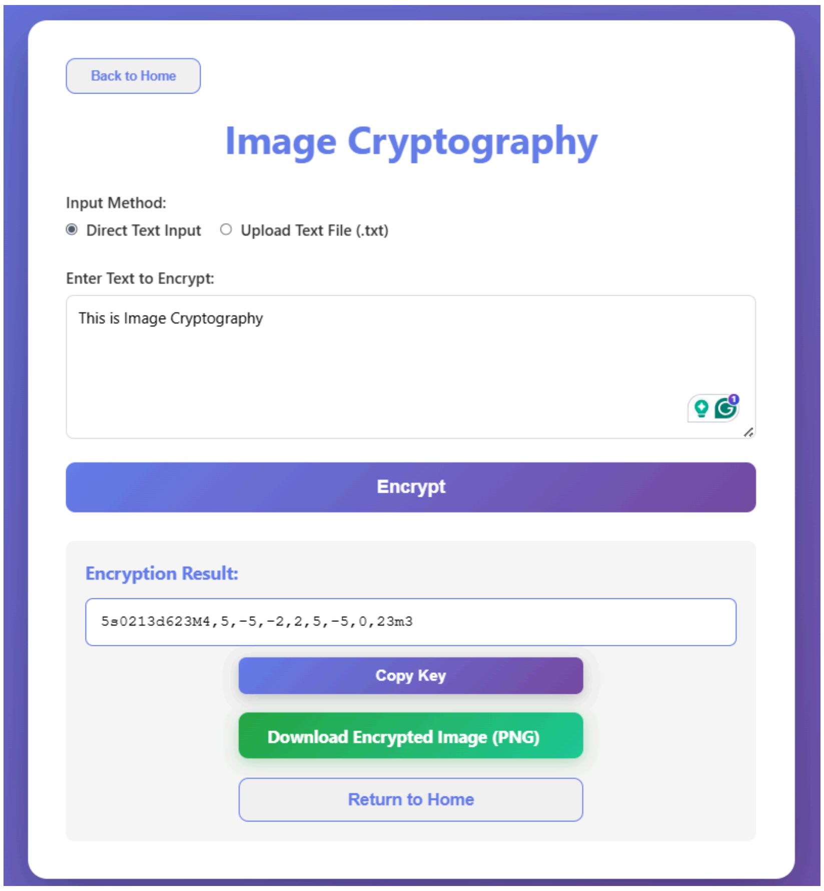
Input Method:
[✓ Direct Text Input] Upload Text File (.txt)

Enter Text to Encrypt:

This is Image Cryptography

Encryption input screen

Program output:



Input Method:
[✓ Direct Text Input] Upload Text File (.txt)

Enter Text to Encrypt:

This is Image Cryptography

Encryption Result:

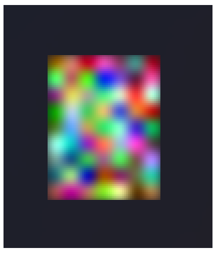
```
5s0213d623M4,5,-5,-2,2,5,-5,0,23m3
```

Encryption result screen

Encryption (mode 1):

After clicking the encrypt button, the program writes encrypted_image.png and prints a key string (records packing, dummy ratio, and manipulation commands).

encrypted_image.png

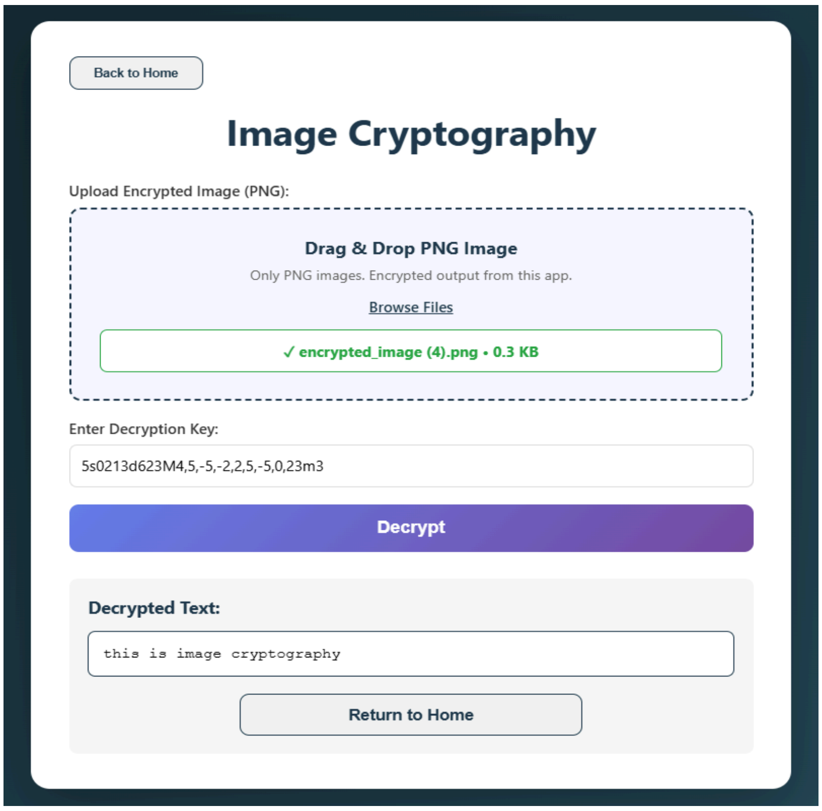


Encrypted image preview

A small rectangular RGB mosaic of high-contrast, randomly colored pixels with no discernible patterns, shapes, or readable glyphs.
The apparent visual noise results from:

- Channel shuffling (data only in selected channels; others are random).
- Dummy pixels that inflate the size and inject additional random colors.

Mode 2: Decryption



Upload Encrypted Image (PNG):

Only PNG images. Encrypted output from this app.

✓ encrypted_image (4).png • 0.3 KB

Enter Decryption Key:

```
5s0213d623M4,5,-5,-2,2,5,-5,0,23m3
```

Decrypted Text:

```
this is image cryptography
```

Decryption interface

Expected mathematical behavior vs actual computation

- **Vectorization**: Each pixel holds v = [R,G,B]^T (Figure 1.1). Observed as integer triples in the PNG (0–255).
- **Linear transform**: For matrix rounds, pixels are mapped by E = K v (mod 256); decryption applies K^{-1} to recover v (Figures 1.2–1.3). Round-trip should reproduce the original text exactly.
- **Determinant use**: det(K) is forced coprime with 256 (odd), ensuring K^{-1} exists (Figure 1.4). The cascade step adds a reversible shift derived from det of 3×3 pixel blocks; decryption subtracts it.
- **Image matrix:** Pixels are arranged to form I (Figure 1.5). The process is bijective given the key.

Efficiency and limitations of the code

The program executes and provides the output very quickly for texts of short to medium length. Although it can encrypt and decrypt very long bodies of text, it takes time to produce the output because the number of pixels that must be processed increases, resulting in a longer runtime. The program also reliably restores the original text as long as the key matches. The limitation here is that the key must be kept properly so that you can decrypt the image. The key that is generated is very random, and you can't memorize it very easily. In addition, the alphabet is limited to a predefined 64-character set, so unsupported characters are omitted in the decrypted output.

Discuss:

What does your result mean in the CS context?

In the CS context, any type of information that is being placed, such as the text itself, forms a representation of binary data. The project demonstrates how data can be formed to securely protect against any threats that can be accessed through it. That is why the principle that is being created here is represented as image pixels, where each character of the text → becomes a number → becomes an RGB vector → becomes a pixel in an image, and then finally the result of your encrypted data.

How does linear algebra make the implementation possible?

The encryption and decryption processes in cryptography rely heavily on linear algebra, as it provides a mathematical framework that secures data integrity. This means that the tools that create the algorithm make it easy to transform or untransform data systematically. The system also implements a diverse choice of encrypted pixel vectors and ensures that they can be recovered to the exact outcome through matrix inversion. With this idea, it can form the mathematical backbone of your image-based cryptographic system.

Conclusion and Future Work:

This project focuses on the text-to-image encoding system, which can also be reversed. It converts plain text into a PNG image where the pixels contain scrambled character data using channel packing, dummy padding, and invertible linear transformations, then uses the generated key to reverse the encryption and obtain the original text exactly in lowercase form. It demonstrates how vectors, matrices, and determinants enable controlled mixing and reversibility, serving as a tool for obfuscation.

What the program achieved:

- Built a reliable text-to-image converter using linear algebra and provided a key that will help recover the original message from the provided image.
- Built a working UI flow that outputs a PNG image and a decryption key, with the correctness of reversion already verified.

Potential improvements:

- The program will be able to handle more types of text like emojis and complex mathematical operations (e.g. Ω, √, π, Σ) besides the defined set of characters by the system.
- GUI enhancements:
    - Progress indicators
    - Clipboard auto-copy toggle for key generation
- Improve processing speed for larger bodies of text

Broader applications in CS:

- **Quantum development** is often open to this cryptographic method, which they use to relay information and secure reliable data in their projects. They are usually rendered obsolete because of the power of quantum computing. This marks the beginning of a new era in quantum computing, as it is essential to stay one step ahead of cyber attacks in the near future.

- **AI encryption** is a new trend for tools used for analyzing cryptographic systems which are designed to create small amounts of data and computation to formulate any vast computation. That is why artificial intelligence are also designed to make it from image representation to character recognition.

Reflection:

#1: Benjamin Asjali

For me, the most powerful mathematical concept in my journey in linear algebra is the idea of matrix operations in optical character recognition, it's because you're manipulating vast amounts of large multidimensional data (pixel arrays, for example). With this type of algorithm, it can handle and represent complex data information, helping to organize it so that people can understand its integrity more clearly. That is why, once you start manipulating the system, learn from it, or recreate it, you can turn a universal translation of the real world into numbers.

#2: Kirby Calong

Linear Algebra has taught me several concepts that I can see are very useful in our field in CS, but no topic for me is more powerful than Matrices. Matrices in CS lenses especially on how we implemented the message decryption is like how complex LLMs like ChatGpt, Gemini, etc generate complex images through decrypting the users prompts or requests by parsing sets of values like strings and integers into transcribable prompts through their complex sorting, parsing, and translation algorithms. In the sense of the correlation of Math and Code, I learned that coding mathematical equations is not really the same as writing it by hand, coding mathematical concepts require setting up methods, exchange of data between variables through various operations, and figuring out the logic through a different perspective, in this sense through the CS perspective. Though the development of the project was not really your so called "smooth sailing" considering we were facing some developmental issue in our frontend, we had to work as a team to resolve the issue with what little time we had, we took into considerations to put alerts and console warning in lines of code that may potentially generate errors throughout the flow of the program so that it won't be difficult for us developers to trace those errors.

#3: Luis Palparan

For me, the most powerful mathematical concept in my journey in linear algebra is matrix operations. I learned how addition, multiplication, and inversion of matrices can represent complex data transformations in real life. They really are useful in many ways that normal people can't see, and I think the math behind it is underappreciated. Like in image processing, each pixel can be represented as a matrix of pixel values. Also in cryptography, where the text can be converted into number matrices, and when multiplied by a key, it generates an encrypted text. For me, the difficulties that I have encountered were when I was introduced to topics that didn't really involve numerical operations. Just like in proofing, usually when you think about math, it is all about solving, finding x, or derivations. But in proofing, it is different, you have to really have a deeper understanding of the definitions, which I am not really into because I am more interested in problem-solving or anything as long as it does not involve many words. But I was able to overcome all of these, thankfully, because of the friends that I have, whom I can learn from whenever I have confusion on the topics.

#4: Mark Tan

One of the most powerful mathematical concepts in my journey is the usage of matrix multiplication. Matrix multiplication showed the ability of reversability, with its inverse basically being a key. The very core of our project was how matrix multiplication was able to serve as a scrambler and solver of our encryption.

#5: Sam Tesoro

The mathematical concept I found most powerful in my linear algebra journey is matrix multiplication. Not only are they present in most linear algebra concepts, from vectors to linear transformations, but they're critical for operations used in machine learning and scientific computing. I learned that math and code go hand in hand, wherein complex coding operations are composed of mathematical equations, and certain mathematical equations cannot be efficiently performed without a program to process and analyze them. My difficulties in this subject came from the overwhelming contents of the tests that could not be finished in the given timeframe, the limited timeframe given in submitting activities and projects with heavier workloads, and my personal difficulty in understanding less numeric computational concepts such as vectors and linear transformations, but I was able to solve these difficulties through proper time management and the commitment to learn and understand the inner workings of concepts I did not have an easy grasp with.

References:

Axler, S. (2015). Linear Algebra Done Right (3rd ed.). Springer. https://link.springer.com/book/10.1007/978-3-319-11080-6

Katz, J., & Lindell, Y. (2020). Introduction to Modern Cryptography (3rd ed.). CRC Press. https://www.crcpress.com/Introduction-to-Modern-Cryptography/Katz-Lindell/p/book/9780815354369

Lay, D. C., Lay, S. R., & McDonald, J. J. (2016). Linear Algebra and Its Applications (5th ed.). Pearson. https://www.pearson.com/en-us/subject-catalog/p/linear-algebra-and-its-applications/P200000005820/9780134022697

Stallings, W. (2017). Cryptography and Network Security: Principles and Practice (7th ed.). Pearson. https://www.pearson.com/en-us/subject-catalog/p/cryptography-and-network-security-principles-and-practice/P200000003415/9780137520177

Strang, G. (2016). Introduction to Linear Algebra (5th ed.). Wellesley-Cambridge Press. https://math.mit.edu/~gs/linearalgebra/