Deep Dive - Vector Spaces

Error Correcting Codes (#Transformations, #Algorithms)

Repetition

Interpretation:

- (0,0,0) suggests that the original bit was 0.
- (1,0,1) suggests that the original bit was 1, assuming a single-bit error.
- (0,0,1) suggests that the original bit was 0, assuming a single-bit error.

Certainty of Correction: The repetition code can correct a single-bit error by majority voting. However, if two bits are flipped, the code will incorrectly correct to the opposite bit.

Example: If the original bit was 1, and the transmitted vector is (1,0,0), majority vote will incorrectly correct to 0.

Parity Codes

Interpretation:

- Vector (1): (1, 1, 1, 1, 0)
  The vector is correct as is. Original Message: (1, 1)
- Vector (2): (0, 1, 0, 0, 0)
  Likely Corrected to: (0, 0, 0, 0, 0). Original Message: (0, 0)
- Vector (3): (1, 0, 0, 0, 1)
  Likely Corrected to: (1, 1, 0, 0, 1). Original Message: (1, 0)
- Vector (4): (0, 1, 1, 1, 1)
  Likely Corrected to: (0, 0, 1, 1, 1). Original Message: (0,1)
- Vector (5): (0, 1, 0, 0, 1)
  Likely Corrected to: (1, 1, 0, 0, 1). Original Message: (1, 0)
- Vector (6): (0, 1, 0, 1, 0)
  Likely Corrected to: (0, 0, 1, 1, 1). Original Message: (0, 1)

Certainty of Correction: These interpretations are based on the assumption of single or double-bit errors. The parity code scheme helps in identifying inconsistencies in the transmitted vectors, but it may not always provide a clear path to correction, especially when multiple bits have flipped. Hence, while it's possible to make educated guesses regarding the original message, certainty of correction isn't always achievable.

Single Bit Errors

With single-bit errors, both the repetition code and the parity code can correctly identify and potentially correct the error.

Example: In the repetition code, if the original bit was 1 and the transmitted vector is (1,0,1), a majority vote will correctly correct to 1.

Similarly, in the parity code, if the original message was (0,1) and the transmitted vector is (0,1,1,0,1), it can be identified that the fourth bit has been flipped and can be corrected to (0,1,1,1,1).

Double Bit Errors

When two bits are flipped, these schemes may fail.

Example: In the repetition code, if the original bit was 1 and the transmitted vector is (1,0,0), a majority vote will incorrectly correct to 0.

In the parity code, if the original message was (0,1) and the transmitted vector is (1,0,1,0,1), it becomes ambiguous as to which bits have been flipped, leading to incorrect or uncertain correction.

Multiple Bit Errors

When multiple bits are flipped, these schemes become increasingly unreliable. The parity code has an additional challenge in that while it can detect an error, it may not always be able to correct it, especially when multiple bits are flipped.

Example: In the repetition code, if the original bit was 1 and the transmitted vector is (0,0,0), all bits have been flipped and the scheme will incorrectly correct to 0.

In the parity code, if the original message was (1,1) and the transmitted vector is (0,0,1,1,0), it's unclear which two bits have been flipped, making the correction uncertain.

Linear Codes

It is an application of linear transformation. A linear code uses a generator matrix G to transform an n-bit message block (vector x) into an m-bit codeword (vector y).

Repetition Method

For the repetition method, where each bit is simply repeated, the generator matrix is:

$$G_{\text{repetition}} = \begin{bmatrix} 1 & 1 & 1 \end{bmatrix}$$

In [ ]:
```python
def encode_repetition(x):
    # Define variables for the generator (transformation) matrix

    a = b = c = 1  # For repetition code, these are all 1

    # Apply transformation to get the codeword
    codeword = [
        a * x,
        b * x,
        c * x
    ]
```

```
    return codeword

# Test the encoding
print(encode_repetition(0))  # Expected output: [0, 0, 0]
print(encode_repetition(1))  # Expected output: [1, 1, 1]
```

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

Parity Method

For the parity method, two bits (x1 and x2) are encoded (x1, x1, x2, x2, x1 + x2). The generator matrix is:

$$G_{\text{parity}}: \begin{bmatrix} 1 & 0 \\ 1 & 0 \\ 0 & 1 \\ 0 & 1 \\ 1 & 1 \end{bmatrix}$$

In [ ]:
```
def encode_parity(x1, x2):
    # Apply the generator (transformation )matrix to get the codeword
    codeword = [
        1 * x1 + 0 * x2,
        1 * x1 + 0 * x2,
        0 * x1 + 1 * x2,
        0 * x1 + 1 * x2,
       (1 * x1 + 1 * x2) % 2  # Parity bit, sum of x1 and x2 modulo 2
    ]
    return codeword

# Test the encoding
print(encode_parity(0, 0))  # Expected output: [0, 0, 0, 0, 0]
print(encode_parity(1, 0))  # Expected output: [1, 1, 0, 0, 1]
print(encode_parity(0, 1))  # Expected output: [0, 0, 1, 1, 1]
print(encode_parity(1, 1))  # Expected output: [1, 1, 1, 1, 0]
```

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

Lossy Compression (#Vectors, #ComputationalTools, #Modeling, #DataViz)

1. Understanding the Basis Vectors and Linear Combinations

a. **What do the eight standard basis vectors represent in terms of this image?** The standard basis vectors represent individual pixels in the row. In this context, a basis vector refers to a situation where one pixel is "on" (1) and all the other pixels are "off" (0).

The standard basis vectors for this 8-dimensional space are:

$$e_1 = [1, 0, 0, 0, 0, 0, 0, 0]$$

$$e_2 = [0, 1, 0, 0, 0, 0, 0, 0]$$

$$e_3 = [0, 0, 1, 0, 0, 0, 0, 0]$$

$$e_4 = [0, 0, 0, 1, 0, 0, 0, 0]$$

$$e_5 = [0, 0, 0, 0, 1, 0, 0, 0]$$

$$e_6 = [0, 0, 0, 0, 0, 1, 0, 0]$$

$$e_7 = [0, 0, 0, 0, 0, 0, 1, 0]$$

$$e_8 = [0, 0, 0, 0, 0, 0, 0, 1]$$

In [ ]:
```
# Each vector has 8 components corresponding to 8 pixels.

# v1 has all pixels with the same intensity.
v1 = vector([1, 1, 1, 1, 1, 1, 1, 1])

# v2 has the first four pixels with positive intensity and the last four with negative
v2 = vector([1, 1, 1, 1, -1, -1, -1, -1])

# v3 has the first two pixels with positive intensity, the next two with negative, and
v3 = vector([1, 1, -1, -1, 0, 0, 0, 0])

# v4 has the first four pixels as neutral, the next two with positive intensity, and t
v4 = vector([0, 0, 0, 0, 1, 1, -1, -1])

# v5 has the first pixel with positive intensity, the second with negative, and the re
v5 = vector([1, -1, 0, 0, 0, 0, 0, 0])

# v6 has the third pixel with positive intensity, the fourth with negative, and the re
v6 = vector([0, 0, 1, -1, 0, 0, 0, 0])

# v7 has the fifth pixel with positive intensity, the sixth with negative, and the res
v7 = vector([0, 0, 0, 0, 1, -1, 0, 0])

# v8 has the seventh pixel with positive intensity and the last pixel with negative.
v8 = vector([0, 0, 0, 0, 0, 0, 1, -1])
```

b. **What does a linear combination of the standard vectors represent?** When we combine these basis vectors in different ways (a linear combination), we are describing the shading or intensity of the entire row of pixels. It is like mixing different amounts of light from each pixel to get a specific overall look.

$$\mathbf{v} = a_1\mathbf{e}_1 + a_2\mathbf{e}_2 + \ldots + a_8\mathbf{e}_8$$

The resulting vector $\mathbf{v}$ will have its pixels set to the values $a_1, a_2, \ldots, a_8$. This is how we can represent any row of 8 grayscale pixels.

c. **What do the coordinates of a vector represent?** The coordinates of a vector are like the brightness settings for each pixel. If a coordinate has a high value, that pixel is bright. If it has a low value, the pixel is dark.

For the given vector $\mathbf{v} = [31, 151, 9, 162, 223, 54, 217, 2]$, each number is the brightness of a pixel. So, the first pixel has a brightness of 31 out of 256, the second has a brightness of 151 out of 256, and so on.

2. Orthogonal Basis in $\mathbb{R}^8$

a. **Demonstrate that these vectors form an orthogonal basis $\mathbf{S}$ of $\mathbb{R}^8$** Recall that orthogonality, in the context of vectors, means that two vectors are perpendicular (90°) to each other. More formally, two vectors are orthogonal if their dot product is zero. The dot product of two vectors gives a measure of how much one vector goes in the direction of the other. If the dot product is zero, it means the vectors are at right angles to each other; they are orthogonal.

Two vectors $\mathbf{u}$ and $\mathbf{v}$ are orthogonal if: $\mathbf{u} \cdot \mathbf{v} = 0$

With Sage Math, I show that all pairs of vectors are orthogonal.

(Small win → I feel more comfortable working with for loops!)

In [ ]:
```
# List of vectors
vectors = [v1, v2, v3, v4, v5, v6, v7, v8]
# 8 - just wanted to practice
numOfVectors = len(vectors)

# Check for orthogonality
    # Dot product of each pair of vectors (v1 & v2 for example)
    # should be zero because that will mean they are orthogonal

# Iterate over each pair of vectors.
for i in range(8):
    for j in range(i+1, 8):
    # Start from i+1 to avoid checking a vector with itself and to avoid duplicate che

        # Compute the dot product of the pair.
        if vectors[i].dot_product(vectors[j]) == 0:
            print(f"v{i+1} and v{j+1} = 0 → orthogonal.")
            # If dot product is zero, they are orthogonal.
        else:
            print(f"v{i+1} and v{j+1} → not orthogonal.")
            # Otherwise, they are not orthogonal.
```

```
v1 and v2 = 0 → orthogonal.
v1 and v3 = 0 → orthogonal.
v1 and v4 = 0 → orthogonal.
v1 and v5 = 0 → orthogonal.
v1 and v6 = 0 → orthogonal.
v1 and v7 = 0 → orthogonal.
v1 and v8 = 0 → orthogonal.
v2 and v3 = 0 → orthogonal.
v2 and v4 = 0 → orthogonal.
v2 and v5 = 0 → orthogonal.
v2 and v6 = 0 → orthogonal.
v2 and v7 = 0 → orthogonal.
v2 and v8 = 0 → orthogonal.
v3 and v4 = 0 → orthogonal.
v3 and v5 = 0 → orthogonal.
v3 and v6 = 0 → orthogonal.
v3 and v7 = 0 → orthogonal.
v3 and v8 = 0 → orthogonal.
v4 and v5 = 0 → orthogonal.
v4 and v6 = 0 → orthogonal.
v4 and v7 = 0 → orthogonal.
v4 and v8 = 0 → orthogonal.
v5 and v6 = 0 → orthogonal.
v5 and v7 = 0 → orthogonal.
v5 and v8 = 0 → orthogonal.
v6 and v7 = 0 → orthogonal.
v6 and v8 = 0 → orthogonal.
v7 and v8 = 0 → orthogonal.
```

(b) Linear Combinations of Matrix Plots

- **Vector (v_1)**: This vector represents a row of pixels where all pixels have the same intensity. In a grayscale image, this would appear as a uniform row.

$$100v_1 + 50v_2$$

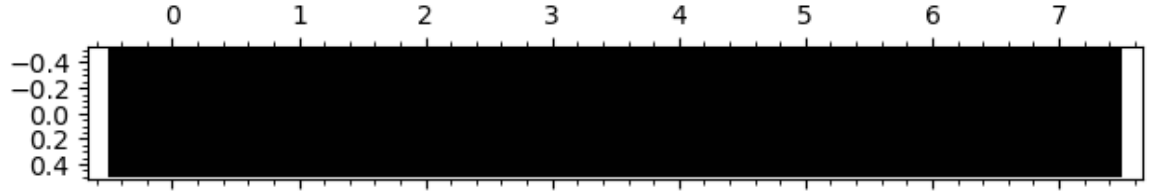
: The first four pixels have a higher intensity (150) compared to the last four pixels (50). This would appear as a row where the left half is brighter than the right half.

$$128v_1 - 64v_3 + 32v_5 - 16v_7$$

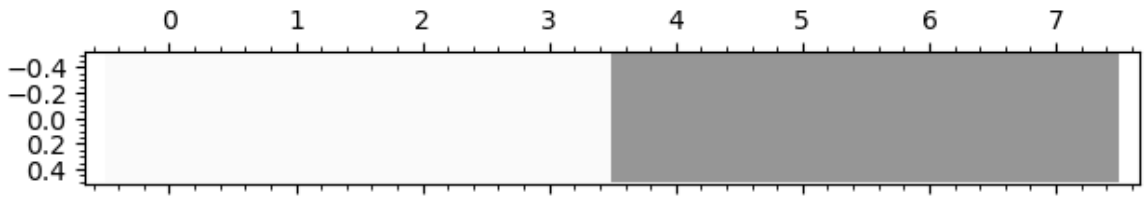
: This vector has varying intensities for the pixels. The third and seventh pixels are the brightest (192), while the second pixel is the darkest (32). This would appear as a row with a mix of bright and dark pixels.

In [ ]:
```
M1 = matrix([v1])
M2 = matrix([200*v1 + 50*v2])
M3 = matrix([128*v1 - 64*v3 + 32*v5 - 16*v7])

# Plotting the grayscale images
matrix_plot(M1, cmap='gray', vmin=0, vmax=256).show()
matrix_plot(M2, cmap='gray', vmin=0, vmax=256).show()
matrix_plot(M3, cmap='gray', vmin=0, vmax=256).show()
```

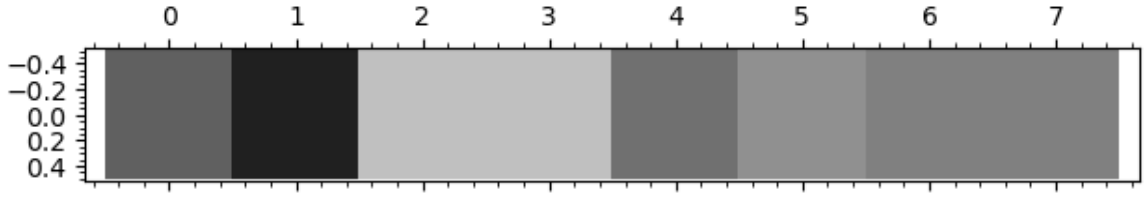
Out[ ]:



Out[ ]:



Out[ ]:



General Description

Linear combinations of the vectors (v_i) represent different grayscale images. By adjusting the coefficients of these vectors, we can control the intensity of each pixel in the row. The vectors (v_i) serve as "basis" vectors, and their linear combinations allow us to represent a wide range of grayscale images. The intensity of each pixel in the resulting image is determined by the coefficients used in the linear combination.

3 Image Compression

(a) Find the Coordinates of $\mathbf{v}$ in Terms of the New Basis $\mathbf{S} = \{\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_8\}$

To find the coordinates of $\mathbf{v}$ in the new basis $\mathbf{S}$, we express $\mathbf{v}$ as a linear combination of the basis vectors $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_8$. This involves solving the matrix equation $\mathbf{S}\mathbf{x} = \mathbf{v}$, where $\mathbf{x}$ represents the coordinates of $\mathbf{v}$ in the basis $\mathbf{S}$.

Step 1: Define the Vector and Basis

We start by defining the original vector $\mathbf{v}$ and the basis vectors that form the matrix $\mathbf{S}$.

The vector $\mathbf{v}$ is given by:

$$\mathbf{v} = \begin{bmatrix} 31 \\ 151 \\ 9 \\ 162 \\ 223 \\ 54 \\ 217 \\ 2 \end{bmatrix}$$

The basis matrix $\mathbf{S}$ is composed of the basis vectors as its columns:

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

In [ ]:
```
# Define the original vector and the basis matrix S
v = vector([31, 151, 9, 162, 223, 54, 217, 2])
S = matrix([
    [1, 1, 1, 1, 1, 1, 1, 1],
    [1, 1, 1, 1, -1, -1, -1, -1],
    [1, 1, -1, -1, 0, 0, 0, 0],
    [0, 0, 0, 0, 1, 1, -1, -1],
    [1, -1, 0, 0, 0, 0, 0, 0],
    [0, 0, 1, -1, 0, 0, 0, 0],
    [0, 0, 0, 0, 1, -1, 0, 0],
    [0, 0, 0, 0, 0, 0, 1, -1]
])
```

Step 2: Find the Coordinates of $\mathbf{v}$ in the New Basis $\mathbf{S}$

To find the coordinates of $\mathbf{v}$ in the new basis $\mathbf{S}$, we solve the equation $\mathbf{S} \cdot \mathbf{c} = \mathbf{v}$, where $\mathbf{c}$ is the vector of coordinates we want to find. Since $\mathbf{S}$ is orthogonal, we can use the inverse of $\mathbf{S}$ to find $\mathbf{c}$.

In [ ]:
```
# Ensure S is invertible
assert S.det() != 0, "Matrix S is not invertible."

# Calculate the inverse of S
S_inv = S.inverse()

# Transform v to the new basis by multiplying with the inverse of S
new_basis_coords = S_inv * v
show(f"Coordinates of v in the new basis: {new_basis_coords}")

# Get the rref() to verify
Sv = S.augment(v, subdivide=True)
show(Sv.rref())
```

Out[ ]: `Coordinates of v in the new basis: (273/2, -173/2, 95/2, -13/2, 134, -83, -109`

Out[ ]:
$$\left(\begin{array}{cccccccc|c} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{273}{2} \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & -\frac{173}{2} \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & \frac{95}{2} \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & -\frac{13}{2} \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 134 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & -83 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & -\frac{109}{2} \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -\frac{113}{2} \end{array}\right)$$

(b) Compression Using Thresholds

Choose three thresholds $\theta_1, \theta_2, \theta_3$. Let $\mathbf{c}_1, \mathbf{c}_2, \mathbf{c}_3$ be the vectors you get by compressing $\mathbf{v}$ using thresholds $\theta_1, \theta_2, \theta_3$ respectively. Express $\mathbf{c}_1, \mathbf{c}_2, \mathbf{c}_3$ in the standard basis. What do these new vectors represent? Create visual representations of $\mathbf{v}, \mathbf{c}_1, \mathbf{c}_2, \mathbf{c}_3$ in the standard basis vectors.

Given the coordinates of $\mathbf{v}$ in the new basis $\mathbf{S}$, we can apply compression by setting a threshold. Any coordinate with an absolute value less than the threshold will be set to zero.

The coordinates of $\mathbf{v}$ in the new basis are:

$$\mathbf{c} = \left(\frac{273}{2}, -\frac{173}{2}, \frac{95}{2}, -\frac{13}{2}, 134, -83, -\frac{109}{2}, -\frac{113}{2}\right)$$

We will now apply different thresholds to compress these coordinates

In [ ]:
```
# Define the thresholds for compression
thresholds = [25, 75, 125]  # Example thresholds

# Apply the thresholds to create compressed versions of the coordinates
compressed_coords_list = [new_basis_coords.apply_map(lambda x: 0 if abs(x) < t else x)

# Convert the compressed coordinates back to the standard basis to get the compressed
compressed_vectors = [S * c for c in compressed_coords_list]

# Output the compressed vectors
for threshold, c_vec in zip(thresholds, compressed_vectors):
    show(f"Compressed vector with threshold {threshold}: {list(c_vec)}")
```

Out[ ]:
```
Compressed vector with threshold 25: [75/2, 315/2, 5/2, 162, 223, 95/2, 217, 1
```

Out[ ]:
```
Compressed vector with threshold 75: [101, -1, 50, 51, 223, 0, 217, 0]
```

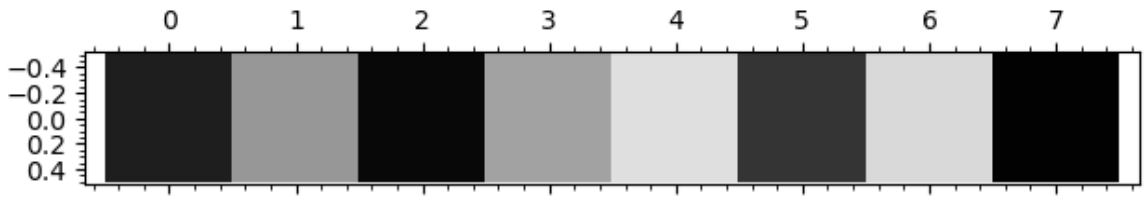
Out[ ]:
```
Compressed vector with threshold 125: [541/2, 5/2, 273/2, 134, 273/2, 0, 134,
```

Step 4: Visualize the Original Vector

Before compression, it's useful to visualize the original vector $\mathbf{v}$ as a grayscale image. This will provide a baseline for comparison with the compressed versions.

In [ ]:
```
# Visualize the original vector 'v' as a grayscale image
matrix_plot([v], cmap='gray', vmin=0, vmax=256).show()
```

Out[ ]:



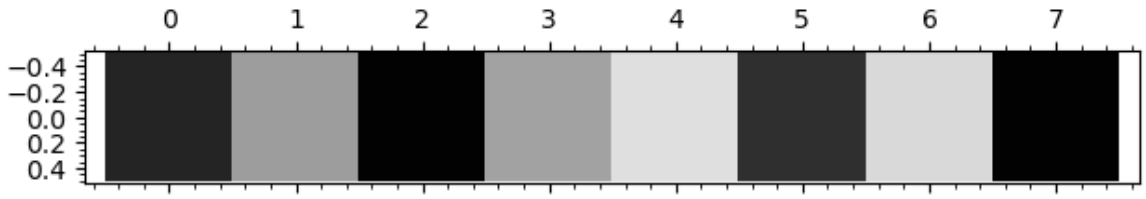
Step 5: Visualize Compressed Vectors

We will now visualize the compressed vectors obtained by applying the thresholds 25, 75, and 125. These visualizations will help us understand the effect of compression on the image represented by the vector $\mathbf{v}$.

In [ ]:
```
# Define the thresholds for compression
thresholds = [25, 75, 125]

# Apply the thresholds to create compressed versions of the coordinates
compressed_coords_list = [new_basis_coords.apply_map(lambda x: 0 if abs(x) < t else x)

# Convert the compressed coordinates back to the standard basis to get the compressed
compressed_vectors = [S * c for c in compressed_coords_list]

# Visualize each compressed vector as a grayscale image
for threshold, c_vec in zip(thresholds, compressed_vectors):
    show(f"Compressed vector with threshold {threshold}:")
    show(c_vec)
    matrix_plot([list(c_vec)], cmap='gray', vmin=0, vmax=256).show()
```

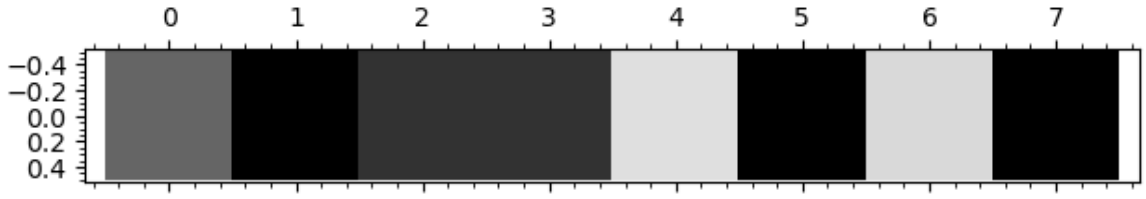
Out[ ]: Compressed vector with threshold 25:

Out[ ]: $\left(\frac{75}{2}, \frac{315}{2}, \frac{5}{2}, 162, 223, \frac{95}{2}, 217, 2\right)$

Out[ ]:


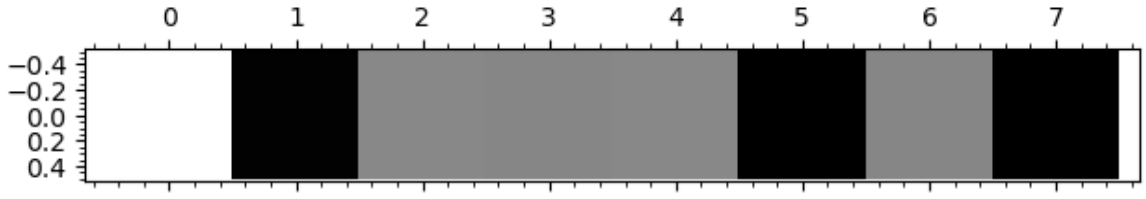
Out[ ]: Compressed vector with threshold 75:

Out[ ]: $(101, -1, 50, 51, 223, 0, 217, 0)$

Out[ ]:


Out[ ]: Compressed vector with threshold 125:

Out[ ]: $\left(\frac{541}{2}, \frac{5}{2}, \frac{273}{2}, 134, \frac{273}{2}, 0, 134, 0\right)$

Out[ ]:


(c) Experimenting with Compression Thresholds

Experiment with different choices of threshold ϵ. How large can you make ϵ before the compression noticeably degrades the quality of the image? This exploration helps us understand the trade-off between compression level and image quality. The threshold ϵ determines the minimum value of pixel intensity that will be preserved during the compression process. Values below this threshold are set to zero, effectively removing them from the image data and thus compressing the image.

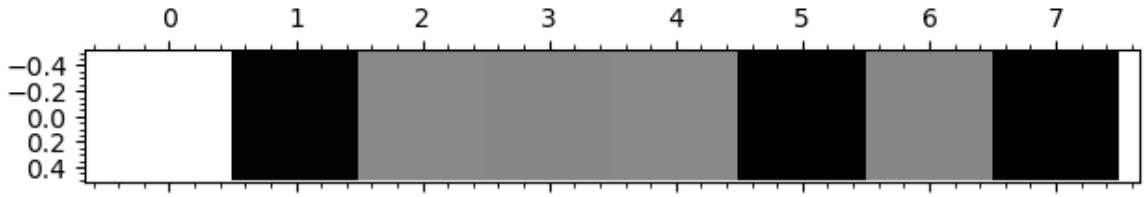
In [ ]:
```
# Define the thresholds for compression
thresholds = [130, 135, 140, 150, 200]
```

```
# Apply the thresholds to create compressed versions of the coordinates
compressed_coords_list = [new_basis_coords.apply_map(lambda x: 0 if abs(x) < t else x)

# Convert the compressed coordinates back to the standard basis to get the compressed
compressed_vectors = [S * c for c in compressed_coords_list]

# Visualize each compressed vector as a grayscale image
for threshold, c_vec in zip(thresholds, compressed_vectors):
    show(f"Compressed vector with threshold {threshold}:")
    show(c_vec)
    matrix_plot([list(c_vec)], cmap='gray', vmin=0, vmax=256).show()
```

Out[ ]: Compressed vector with threshold 130:

Out[ ]: $\left(\frac{541}{2}, \frac{5}{2}, \frac{273}{2}, 134, \frac{273}{2}, 0, 134, 0\right)$

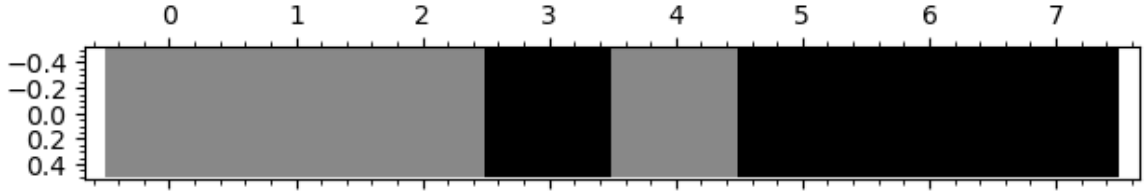
Out[ ]:



Out[ ]: Compressed vector with threshold 135:

Out[ ]: $\left(\frac{273}{2}, \frac{273}{2}, \frac{273}{2}, 0, \frac{273}{2}, 0, 0, 0\right)$

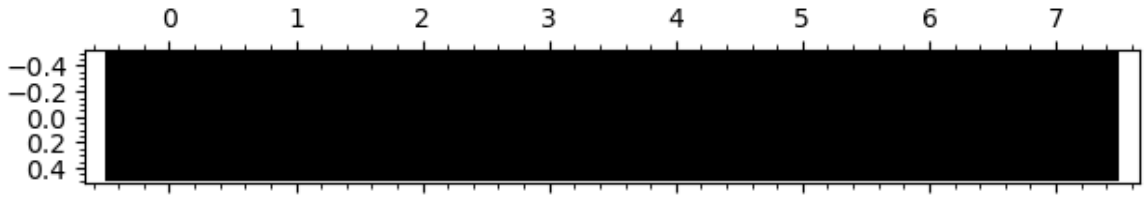
Out[ ]:



Out[ ]: Compressed vector with threshold 140:

Out[ ]: $(0, 0, 0, 0, 0, 0, 0, 0)$

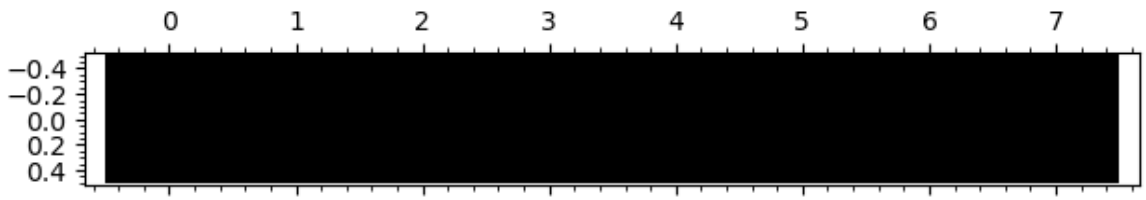
Out[ ]:



Out[ ]: Compressed vector with threshold 150:

Out[ ]: $(0, 0, 0, 0, 0, 0, 0, 0)$

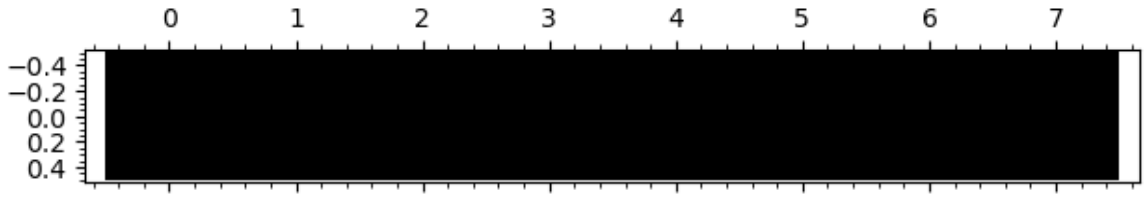
Out[ ]:



Out[ ]: Compressed vector with threshold 200:

Out[ ]: $(0, 0, 0, 0, 0, 0, 0, 0)$

Out[ ]:



Observing the Effects of Compression

When examining the matrix plots from cells 2 to 4, take note of how the visual distinction between the pixels begins to diminish as the compression threshold ϵ increases. At lower thresholds, the individual characteristics of each pixel are more pronounced. However, as ϵ is increased, the pixels start to "merge," losing their distinct features and becoming less distinguishable to the human eye. This merging effect is a direct consequence of the lossy compression, where finer details are sacrificed for the sake of reducing the image's data size.

Checking with other tools

Notably, when the compression threshold ϵ is set to 140 or above, a color picker indicates that the pixels display a hexadecimal color value of #000000, which corresponds to pure black. This suggests that the compression has consolidated the pixel values to a point where distinct grayscale variations are no longer preserved, resulting in a uniform black color across the affected pixels.

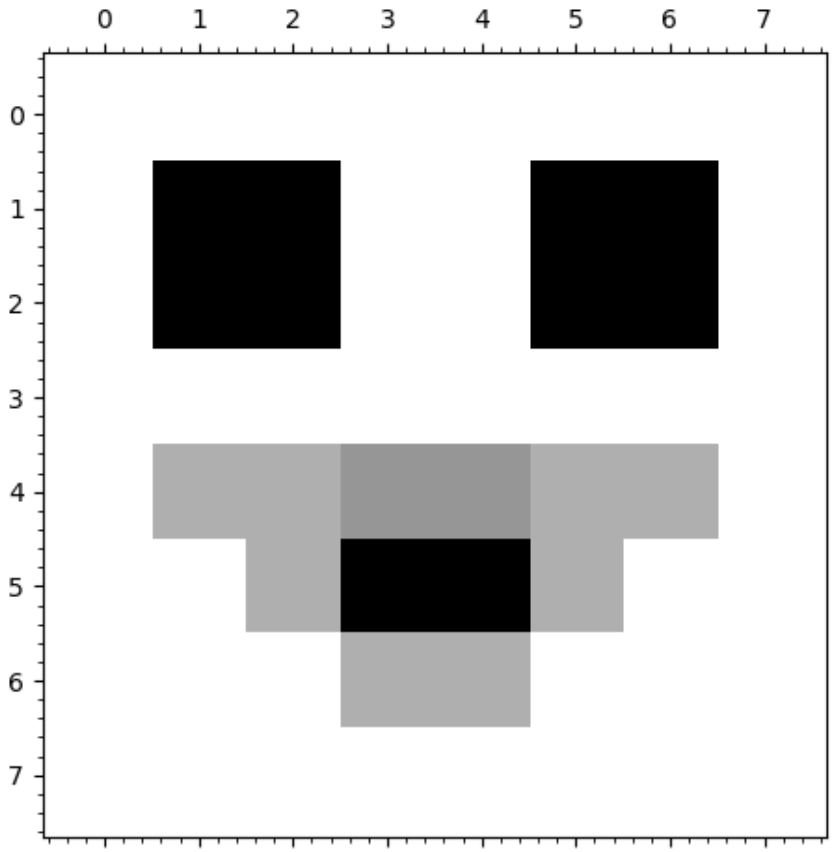
Optional

In [ ]:

```
# I defined the original image matrix with gray values around the smiley face
carl_startOfSemester = matrix([
    [256, 256, 256, 256, 256, 256, 256, 256],
    [256,   0,   0, 256, 256,   0,   0, 256],
    [256,   0,   0, 256, 256,   0,   0, 256],
    [256, 256, 256, 256, 256, 256, 256, 256],
    [256, 175, 175, 150, 150, 175, 175, 256],
    [256, 256, 175,   0,   0, 175, 256, 256],
    [256, 256, 256, 175, 175, 256, 256, 256],
    [256, 256, 256, 256, 256, 256, 256, 256]
])

# Visualize the original image with gray values
print("Carl at the start of the semester")
matrix_plot(carl_startOfSemester, cmap='gray', vmin=0, vmax=256).show()
```

```
Carl at the start of the semester
```

Out[ ]:



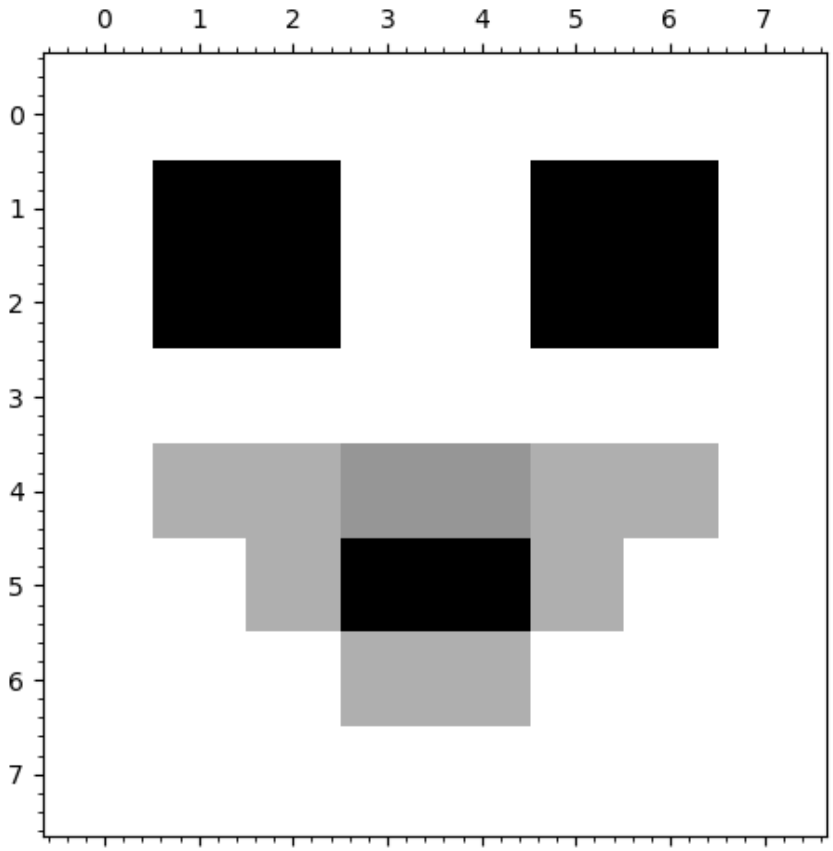
In [ ]:

```
# Define the original image matrix with gray values around the smiley face
carl_startOfSemester = matrix([
    [256, 256, 256, 256, 256, 256, 256, 256],
    [256,   0,   0, 256, 256,   0,   0, 256],
    [256,   0,   0, 256, 256,   0,   0, 256],
    [256, 256, 256, 256, 256, 256, 256, 256],
    [256, 175, 175, 150, 150, 175, 175, 256],
    [256, 256, 175,   0,   0, 175, 256, 256],
    [256, 256, 256, 175, 175, 256, 256, 256],
    [256, 256, 256, 256, 256, 256, 256, 256]
])

# Define the basis matrix S - just like earlier
S = matrix([
    [1, 1, 1, 1, 1, 1, 1, 1],
    [1, 1, 1, 1, -1, -1, -1, -1],
    [1, 1, -1, -1, 0, 0, 0, 0],
    [0, 0, 0, 0, 1, 1, -1, -1],
    [1, -1, 0, 0, 0, 0, 0, 0],
    [0, 0, 1, -1, 0, 0, 0, 0],
    [0, 0, 0, 0, 1, -1, 0, 0],
    [0, 0, 0, 0, 0, 0, 1, -1]
]).transpose()

# Ensure S is invertible
assert S.det() != 0, "Matrix S is not invertible."

# Calculate the inverse of S
S_inv = S.inverse()

# Transform the image matrix to the new basis by multiplying with the inverse of S
new_basis_coords = S_inv * carl_startOfSemester * S_inv.transpose()

# Define the threshold for compression
```

```
threshold = 15

# Apply the threshold to create a compressed version of the coordinates
compressed_coords = new_basis_coords.apply_map(lambda x: 0 if abs(x) < threshold else

# Transform the compressed coordinates back to the standard basis
compressed_image = S * compressed_coords * S.transpose()

# Visualize the original image
print("Original Image:")
print("Carl at the start of the semester")
matrix_plot(carl_startOfSemester, cmap='gray', vmin=0, vmax=256).show()

# Visualize the compressed image
print(f"Compressed Image with threshold {threshold}:")
print("Carl at the end of the semester")
matrix_plot(compressed_image, cmap='gray', vmin=0, vmax=256).show()
```

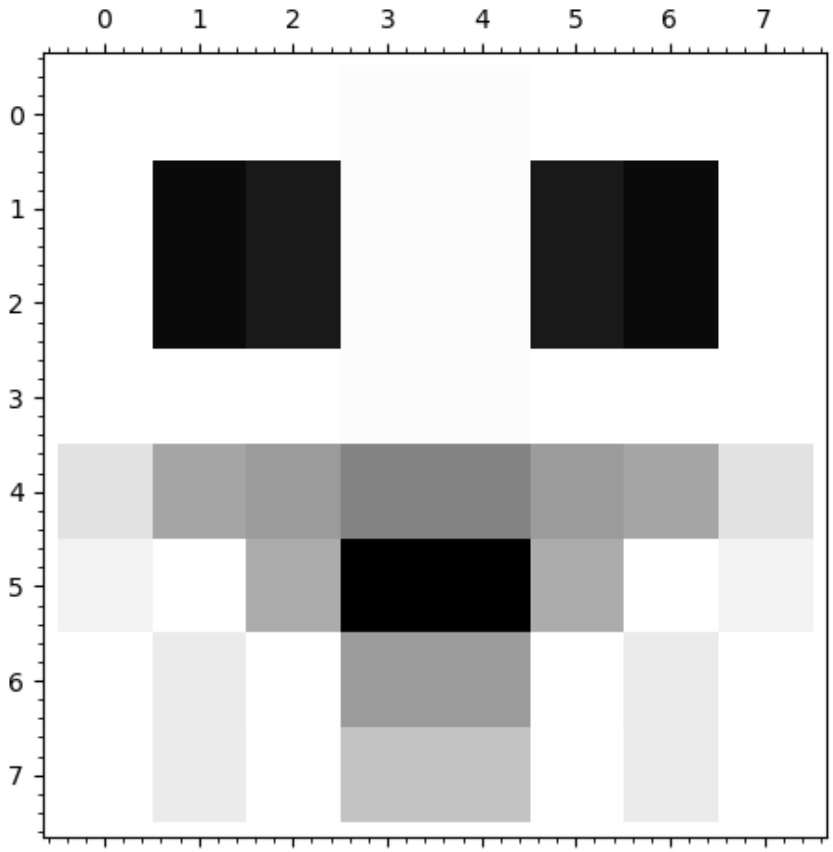
```
Original Image:
Carl at the start of the semester
```

Out[ ]:




```
Compressed Image with threshold 15:
Carl at the end of the semester
```

Out[ ]:



In the provided code, I started by defining the original image matrix, which contains a smiley face and various shades of gray. Then, I created the basis matrix, S, ensuring it's invertible, a crucial step for the transformation. After that, I calculated the inverse of S and used it to transform the image matrix into the new basis. To show the effects of compression, I introduced a threshold of 15. Applying this threshold allowed me to reduce the image's data, setting values below 15 to zero, essentially compressing the image. I opted for a threshold of 15 because I recognized that the high-intensity background of white pixels (255) wouldn't be impacted by the tolerance. This choice allowed me to focus the compression on the areas with gray shades near the smiley face, where the effects of compression would be more noticeable.

Unlisted HC Use

#Analogy The image compression section brought back memories of my time playing Minecraft, a voxel-based pixel game where I enjoyed recreating famous images like memes and paintings in their blocky, lower-resolution versions. One notable example was transforming "The Starry Night" (https://imgur.com/a/05y5X5y) into a 256 by 256 pixel rendition. This experience offered valuable insights into the image compression process and clarified my expectations regarding the final output.

Rather than using the entire spectrum of colors available in the original painting, I focused on the most prominent ones, like variations of blue, gold, orange, and white. Like how compression saves space, my choice of using limiting my colors also saved me time and effort. Knowing this assured me that I was on the right coding path.

#Organization Recognizing the complexity of this assignment, I leveraged my recently-acquired knowledge of working with LaTeX (converted by GPT) to tackle the final problem. I adopted a

structured approach by alternating between Markdown and code cells, ensuring that the content would be more comprehensible for the audience (i.e. you, professor). I placed a greater emphasis on using LaTeX for new basis vectors and mathematical objects than for plain text, as I understood their significance in the assignment's output.

AI Use

Like before, I used CoCalc's built-in AI debugger to help me fix minor syntax errors. Given that this assignment had more qualitative elements than before, I used Whiper API (AI-powered voice transcript) to capture my thoughts in words, helping me create holistic explanation. After this, I sought help from GPT-3 and iterated with the output from WhisperAPI to ensure that my wordings were clear. The biggest contribution by AI was the re-formatting of my text to LaTeX. I used ChatGPT to write LaTeX in my notebook!