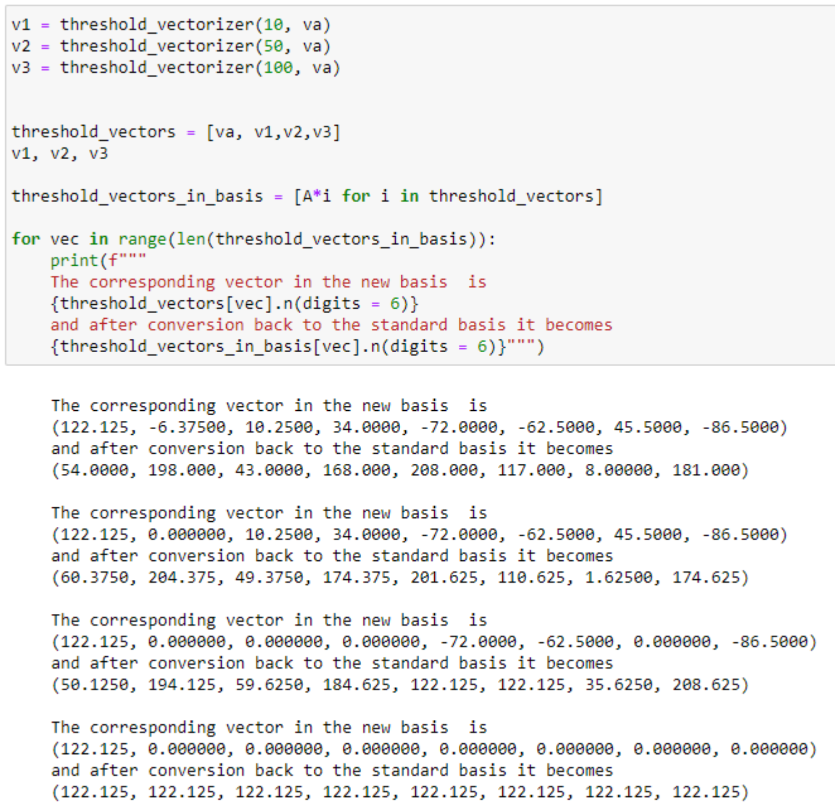
(c) i. In order to change the appropriate threshold change on to the vector, we know that it must lie within the new basis, the coordinates for which we found within the last part of the previous question.
Choosing the three thresholds $\epsilon_1 = 10$, $\epsilon_2 = 50$ and $\epsilon_3 = 100$, we get the vectors

$$v_1 = \begin{bmatrix} 122.125 \\ 0 \\ 10.25 \\ 34 \\ -72 \\ -62.5 \\ 45.5 \\ -86.5 \end{bmatrix}, v_2 = \begin{bmatrix} 122.125 \\ 0 \\ 0 \\ 0 \\ -72 \\ -62.5 \\ 0 \\ -86.5 \end{bmatrix} \text{ and } v_3 = \begin{bmatrix} 122.125 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

ii. In order to express the transformed vectors in the standard basis, we have to change the basis back to the standard basis. To do that we can just multiply the change of basis matrix from (in this case, just the collection of the basis vectors since that's the representation of the new basis in the standard basis). Doing this through sage we can see that they equate to the vectors shown in Fig 3. These new vectors represent the row of pixels we saw previously getting compressed within the new basis. We see from this that all the vectors generally tend towards a smaller difference in their magnitude within the standard basis which tells us that by decreasing the constrast within the image, we're storing less information about the image but still able to give almost the same visual representation.

iii. We can easily create visual representations by using matrix_plot within sage as was done in Fig 4

iv. By making code that iterates through threshold values from 5 to 100 with jumps of 5, I was able to directly see through small increments in the threshold at which point does the compression become obvious. This point is shown within Fig 5. Specifically, when the threshold is 35, the fifth pixel



```
v1 = threshold_vectorizer(10, va)
v2 = threshold_vectorizer(50, va)
v3 = threshold_vectorizer(100, va)


threshold_vectors = [va, v1,v2,v3]
v1, v2, v3

threshold_vectors_in_basis = [A*i for i in threshold_vectors]

for vec in range(len(threshold_vectors_in_basis)):
    print(f"""
    The corresponding vector in the new basis  is
    {threshold_vectors[vec].n(digits = 6)}
    and after conversion back to the standard basis it becomes
    {threshold_vectors_in_basis[vec].n(digits = 6)}""")
```

```
    The corresponding vector in the new basis  is
    (122.125, -6.37500, 10.2500, 34.0000, -72.0000, -62.5000, 45.5000, -86.5000)
    and after conversion back to the standard basis it becomes
    (54.0000, 198.000, 43.0000, 168.000, 208.000, 117.000, 8.00000, 181.000)

    The corresponding vector in the new basis  is
    (122.125, 0.000000, 10.2500, 34.0000, -72.0000, -62.5000, 45.5000, -86.5000)
    and after conversion back to the standard basis it becomes
    (60.3750, 204.375, 49.3750, 174.375, 201.625, 110.625, 1.62500, 174.625)

    The corresponding vector in the new basis  is
    (122.125, 0.000000, 0.000000, 0.000000, -72.0000, -62.5000, 0.000000, -86.5000)
    and after conversion back to the standard basis it becomes
    (50.1250, 194.125, 59.6250, 184.625, 122.125, 122.125, 35.6250, 208.625)

    The corresponding vector in the new basis  is
    (122.125, 0.000000, 0.000000, 0.000000, 0.000000, 0.000000, 0.000000, 0.000000)
    and after conversion back to the standard basis it becomes
    (122.125, 122.125, 122.125, 122.125, 122.125, 122.125, 122.125, 122.125)
```

Figure 3: Calculating the values of v with the implementation of the threshold

gets noticeably lighter whereas this 8th pixel gets noticeably darker. Looking to better optimize where this threshold loses the original visualization, this method does not seem the most efficient since it relies on visual interpretation of a change whereas a more analytical method could instead be looking to consistently compare the compressed image back with the image to see whether it goes beyond a certain threshold for loss.

(d) In order to extend this method to use within images, it was first important to understand the concept behind the compression methodology being used. Looking up resources online with Jessey, we were able to find that this method of compression utilizes the haar wavelets within the haar bases. As we see within *(MIT, Lecture 31: Change of basis; image compression 2011)*, this compression method is a form of lossy compression where by converting to a form with many zeroes, you attempt to filter out the parts of the image for which you would need as much information to convey the same visual feeling.

To be able to do this within sage, it was first important to make a function that creates a haar basis for the image being used which we can see being done within Fig 6.

After creating the function, the actual compression process is trivial because we follow the same method where first after finding the image within the haar basis, we employ the compression and then convert it back to the original process. This process is algorithmically equivalent to first multiplying the vector for the image by the inverse of the haar basis and then doing the compression within the new basis by setting all vectors with magnitudes less than the compression amount to zero and then converting that back to the standard basis by just multiplying with the haar basis.

To do this process then, I used 32 bit images which correspond to images within 32 x 32 pixels. Since every pixel would need to have a separate separate basis vectors that corresponds to it, this results in 4096 basis vectors. In sage, this translated to using the Pillow and numpy packages to convert the images to greyscale and express them as an array.

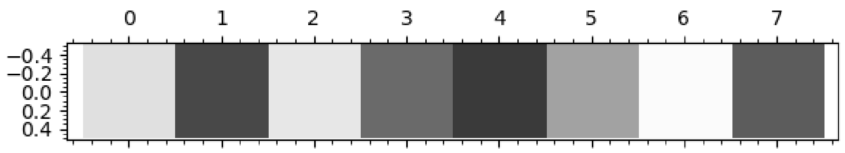
This code used to the conversion can be seen in Fig 7. In this code, one aspect that was important to convert within real images is that the scale used is opposite to the one we discuss in matrix plots where 0 is completely black and 255 is completely black and thus, in order to simulate that within the current method, I inverted the matrix to change it to a form that's usable with the method being utilized. The results can be seen in Fig 8

One thing that became clear from this was the lossy nature of the compression which was especially visible in all images when the threshold became 35 because it clearly looked as if most chunks of the original image were being lost in the process of compressing. One interesting thing to see was that instead file size decreasing due to the compression, it was instead increasing with each iteration. I wasn't too sure about why this was happening but I thought that due to the inverted greyscale, the low values for storing the image might not have translated but that didn't seem to be the case since even converting the image back before saving produced the same effect. I have to imagine that this must be because of some internal factor involved with converting an array to a png since the filetype itself might have its own inherent compression which this lossy compression interferes with, thus creating a need for more storage space.

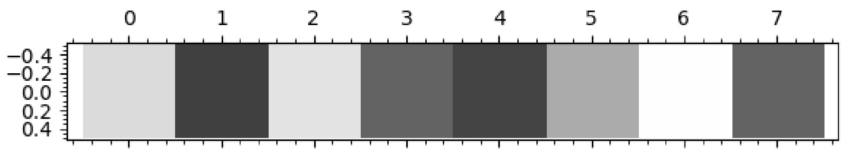
```
def vector_map(vector):
    return matrix_plot(matrix([vector]), cmap = 'Greys', vmin = 0, vmax = 256)
```

```
thresholds = [0,10,50,100]
for a in range(len(threshold_vectors)):
    print(f'''
    With the threshold {thresholds[a]}, you see the following compression''')
    show(vector_map(threshold_vectors_in_basis[a]))
```

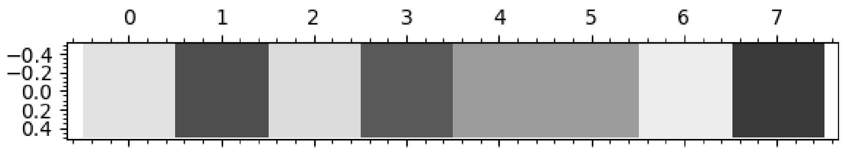
```
    With the threshold 0, you see the following compression
```



```
    With the threshold 10, you see the following compression
```



```
    With the threshold 50, you see the following compression
```



```
    With the threshold 100, you see the following compression
```

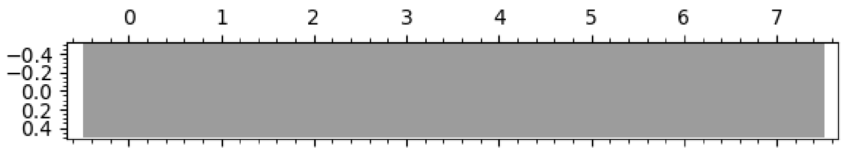
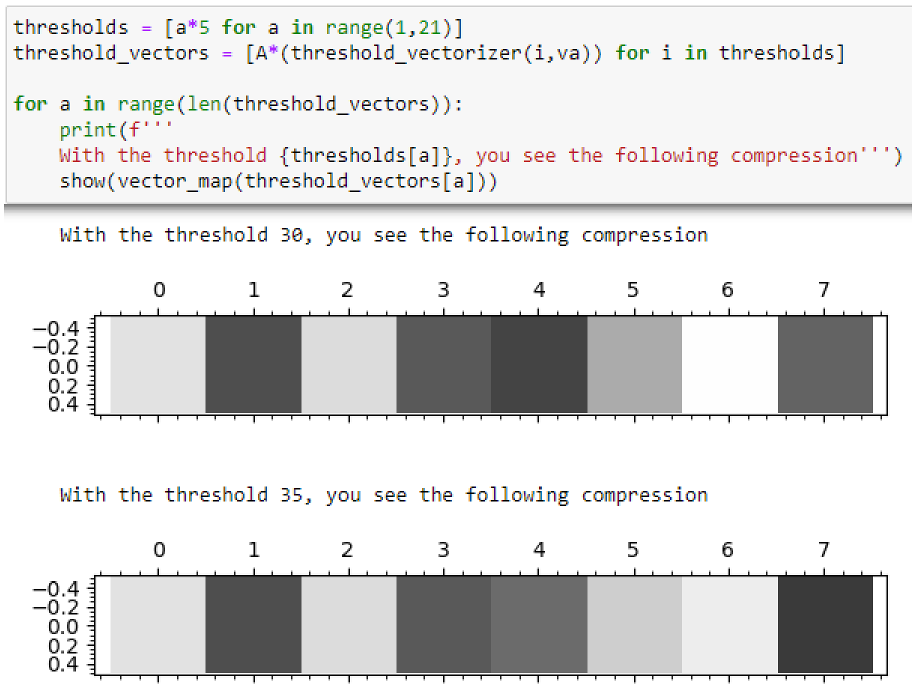


Figure 4: Visual representations of using thresholds to compress pixels

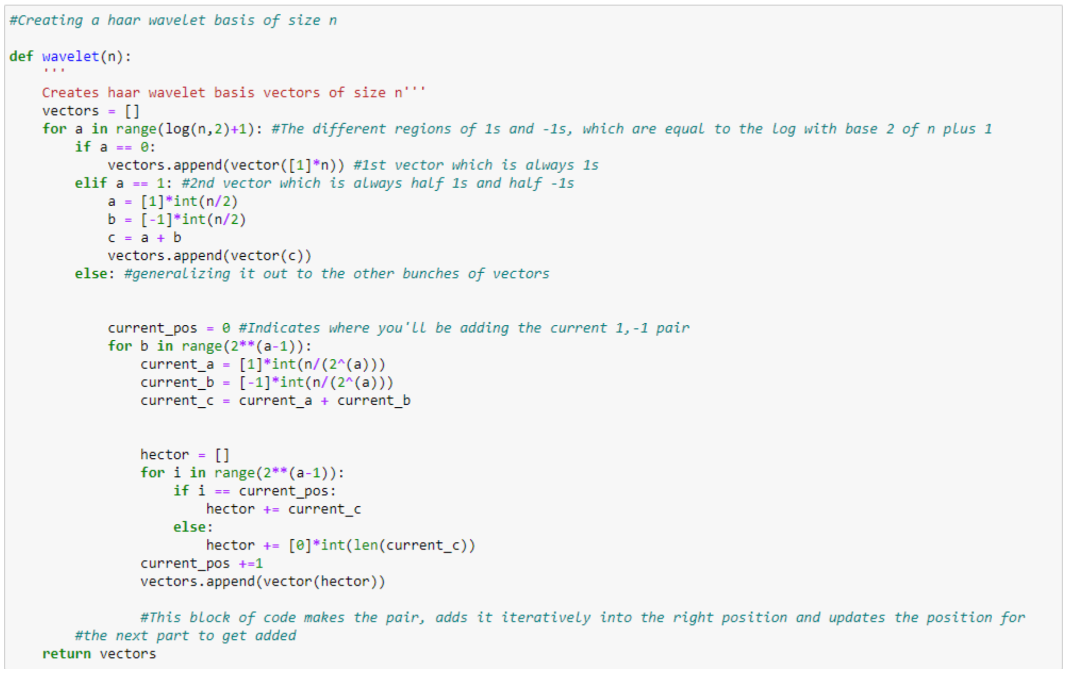


```
thresholds = [a*5 for a in range(1,21)]
threshold_vectors = [A*(threshold_vectorizer(i,va)) for i in thresholds]

for a in range(len(threshold_vectors)):
    print(f'''
    With the threshold {thresholds[a]}, you see the following compression''')
    show(vector_map(threshold_vectors[a]))
```

```
    With the threshold 30, you see the following compression
```

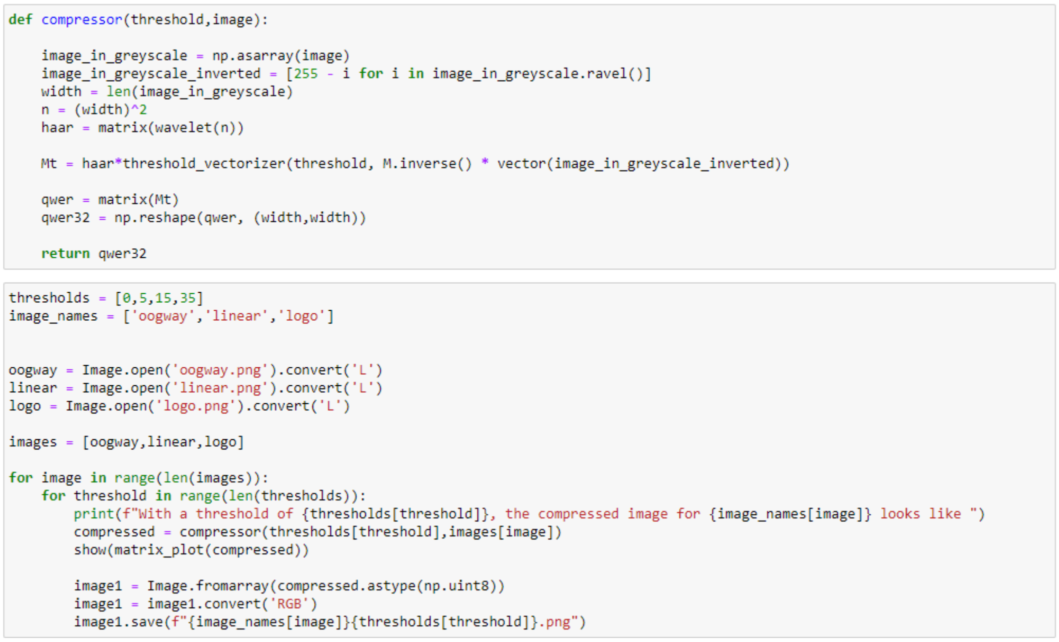
```
    With the threshold 35, you see the following compression
```

Figure 5: Finding the point at which the compression becomes noticeable through sage



```python
#Creating a haar wavelet basis of size n

def wavelet(n):
    '''
    Creates haar wavelet basis vectors of size n'''
    vectors = []
    for a in range(log(n,2)+1): #The different regions of 1s and -1s, which are equal to the log with base 2 of n plus 1
        if a == 0:
            vectors.append(vector([1]*n)) #1st vector which is always 1s
        elif a == 1: #2nd vector which is always half 1s and half -1s
            a = [1]*int(n/2)
            b = [-1]*int(n/2)
            c = a + b
            vectors.append(vector(c))
        else: #generalizing it out to the other bunches of vectors


            current_pos = 0 #Indicates where you'll be adding the current 1,-1 pair
            for b in range(2**(a-1)):
                current_a = [1]*int(n/(2^(a)))
                current_b = [-1]*int(n/(2^(a)))
                current_c = current_a + current_b


                hector = []
                for i in range(2**(a-1)):
                    if i == current_pos:
                        hector += current_c
                    else:
                        hector += [0]*int(len(current_c))
                current_pos +=1
                vectors.append(vector(hector))

                #This block of code makes the pair, adds it iteratively into the right position and updates the position for
        #the next part to get added
    return vectors
```

Figure 6: Function which creates a haar basis of size n



```python
def compressor(threshold,image):

    image_in_greyscale = np.asarray(image)
    image_in_greyscale_inverted = [255 - i for i in image_in_greyscale.ravel()]
    width = len(image_in_greyscale)
    n = (width)^2
    haar = matrix(wavelet(n))

    Mt = haar*threshold_vectorizer(threshold, M.inverse() * vector(image_in_greyscale_inverted))

    qwer = matrix(Mt)
    qwer32 = np.reshape(qwer, (width,width))

    return qwer32
```

```python
thresholds = [0,5,15,35]
image_names = ['oogway','linear','logo']


oogway = Image.open('oogway.png').convert('L')
linear = Image.open('linear.png').convert('L')
logo = Image.open('logo.png').convert('L')

images = [oogway,linear,logo]

for image in range(len(images)):
    for threshold in range(len(thresholds)):
        print(f"With a threshold of {thresholds[threshold]}, the compressed image for {image_names[image]} looks like ")
        compressed = compressor(thresholds[threshold],images[image])
        show(matrix_plot(compressed))

        image1 = Image.fromarray(compressed.astype(np.uint8))
        image1 = image1.convert('RGB')
        image1.save(f"{image_names[image]}{thresholds[threshold]}.png")
```

Figure 7: Code to compress images

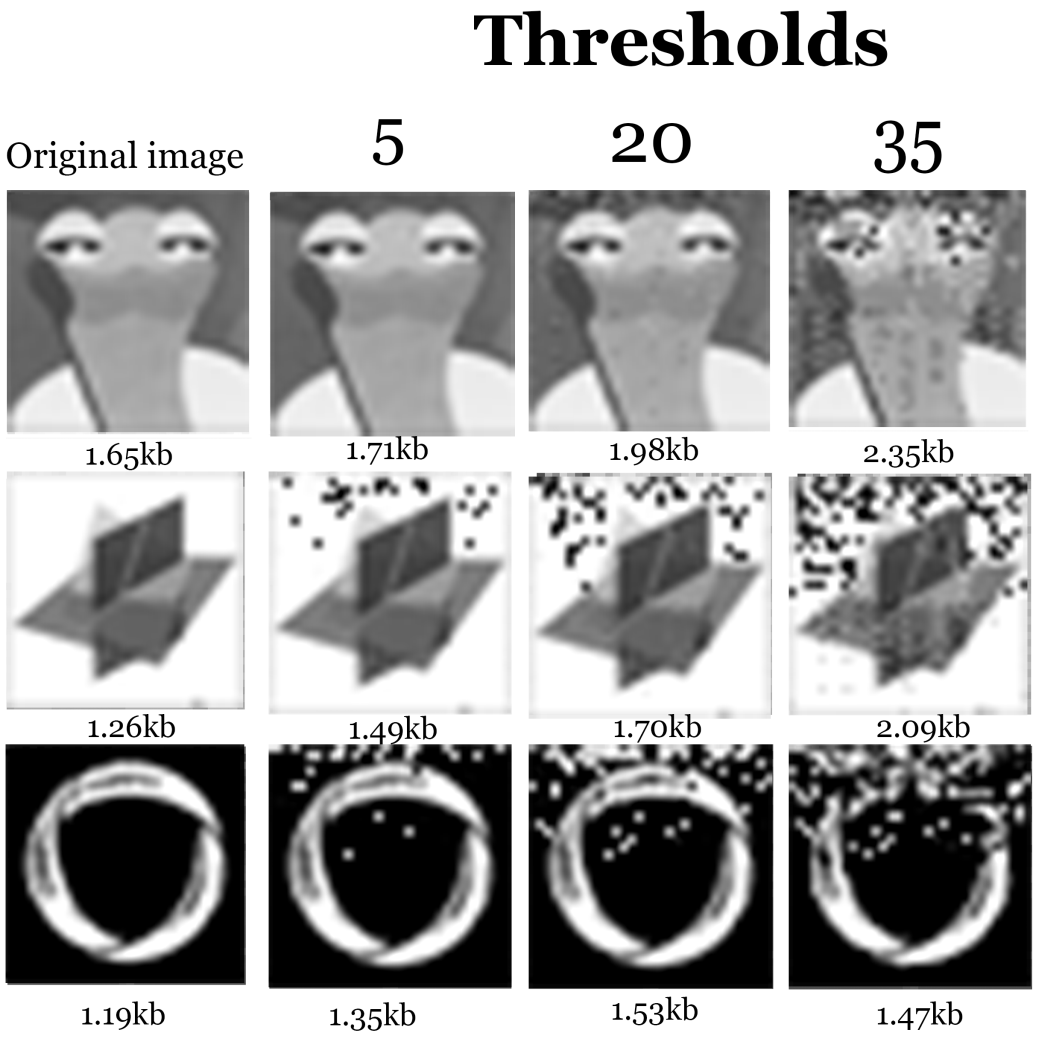


Figure 8: Final results from the experiment