# Haar Transform
# CS 430
# ©Denbigh Starkey

## Background

The Haar transform has a rather messy looking definition, so I'll give the definition and will then go carefully through an example.

The transform is important for a couple of reasons:

- It leads to efficient image compression schemes, both lossless and lossy, which I'll describe later in these notes, and
- it provides a lead-in to wavelets, which I'll be covering in the next set of notes.

I'll continue to assume the image is $N \times N$, where $N$ is a power of 2, and that $J = \log_2 N$.

In the pyramid scheme we created a number of new images, leading to an increase of pixels with an upper bound of a 33% increase. With Haar our transformed image will have the same dimensions as the original, and so any compression savings are immediately gained.

# Computing the Haar Transform

Given an $N \times N$ image $F$, the Haar transform will be computed as:

$T = HFH^{\mathrm{T}}$. (Note: Gonzalez incorrectly omits the transpose.)

where $H$ contains the *Haar basis functions*. The matrix $H$ will be defined with the structure:

$$H_N = \begin{bmatrix} h_0(0) & h_0(\frac{1}{N}) & \cdots & h_0(\frac{N-1}{N}) \\ h_1(0) & h_1(\frac{1}{N}) & \cdots & h_1(\frac{N-1}{N}) \\ \cdots & \cdots & \cdots & \cdots \\ h_{N-1}(0) & h_{N-1}(\frac{1}{N}) & \cdots & h_{N-1}(\frac{N-1}{N}) \end{bmatrix}$$

The definition of these $h_k(z)$ blending functions first needs a side definition of two additional variables, $p$ and $q$, for each row $k$. They are defined using four rules:

$k = 2^p + q - 1$
$0 \le p \le J - 1$
if $p = 0$, then $q = 0$ or 1
if $p \ne 0$, then $1 \le q \le 2^p$

These are used to define the $h_k(z)$ functions as follows.

$$h_0(z) = \frac{1}{\sqrt{N}}$$

$$h_k(z) = h_{pq}(z) = \begin{cases} \frac{1}{\sqrt{N}} 2^{p/2} & \text{if } \frac{q-1}{2^p} \le z < \frac{q-\frac{1}{2}}{2^p} \\ -\frac{1}{\sqrt{N}} 2^{p/2} & \text{if } \frac{q-\frac{1}{2}}{2^p} \le z < \frac{q}{2^p} \\ 0 & \text{otherwise} \end{cases} .$$

I'll build the $H$ matrix slowly using the $4 \times 4$ matrix as an example, where $N = 4$ and so $J = 2$. First I'll get the $a$ and $b$ values that we need for $k = 1$, 2, and 3. For all of these cases the second rule for $p$ and $q$ states that $p$ is either 0 or 1, since $J = 2$.

If $k = 1 = 2^p + q - 1$, then either $p = 0$ and so $q = 1$, or $p = 1$ and so $q = 0$. The second option violates the fourth rule and so $p = 0$ and $q = 1$.

If $k = 2 = 2^p + q - 1$, then either $p = 0$ and so $q = 2$, or $p = 1$ and so $q = 1$. The first option violates the third rule and so $p = 1$ and $q = 1$.

If $k = 3 = 2^p + q - 1$, then either $p = 0$ and so $q = 3$, or $p = 1$ and so $q = 2$. The first option violates the third rule and so $p = 1$ and $q = 2$.

Now we need the matrix:

$$H_4 = \begin{bmatrix} h_0(0) & h_0(\frac{1}{4}) & h_0(\frac{1}{2}) & h_0(\frac{3}{4}) \\ h_1(0) & h_1(\frac{1}{4}) & h_1(\frac{1}{2}) & h_1(\frac{3}{4}) \\ h_2(0) & h_2(\frac{1}{4}) & h_2(\frac{1}{2}) & h_2(\frac{3}{4}) \\ h_3(0) & h_3(\frac{1}{4}) & h_3(\frac{1}{2}) & h_3(\frac{3}{4}) \end{bmatrix}.$$

Note that all of these values are multiples of $\frac{1}{\sqrt{N}}$, which is $\frac{1}{\sqrt{4}}$ in this case, so we can pull this value ($\frac{1}{2}$) outside the matrix.

The first row is easy, since every element is defined as being $\frac{1}{\sqrt{N}}$.

The second row has $k = 1$. As we saw above, $p = 0$ and $q = 1$. So the definition says that:

$$h_1(z) = h_{01}(z) = \begin{cases} \frac{1}{2} & \text{if } 0 \le z < \frac{1}{2} \\ -\frac{1}{2} & \text{if } \frac{1}{2} \le z < 1 \\ 0 & \text{otherwise} \end{cases}.$$

So $h_1(0) = \frac{1}{2}$, $h_1(\frac{1}{4}) = \frac{1}{2}$, $h_1(\frac{1}{2}) = -\frac{1}{2}$, and $h_1(\frac{3}{4}) = -\frac{1}{2}$.

Now consider the third row, with $k = 2$. As we saw above, $p = 1$ and $q = 1$. The definition says that:

$$h_2(z) = h_{11}(z) = \begin{cases} \frac{1}{2}\sqrt{2} & \text{if } 0 \le z < \frac{1}{4} \\ -\frac{1}{2}\sqrt{2} & \text{if } \frac{1}{4} \le z < \frac{1}{2} \\ 0 & \text{otherwise} \end{cases}.$$

So $h_2(0) = \frac{1}{2}\sqrt{2}$, $h_2(\frac{1}{4}) = -\frac{1}{2}\sqrt{2}$, $h_2(\frac{1}{2}) = 0$, and $h_2(\frac{3}{4}) = 0$.

For $k = 3$ we found that $p = 1$ and $q = 2$. So, using the definition,

$$h_3(z) = h_{12}(z) = \begin{cases} \frac{1}{2}\sqrt{2} & \text{if } \frac{1}{2} \le z < \frac{3}{4} \\ -\frac{1}{2}\sqrt{2} & \text{if } \frac{3}{4} \le z < 1 \\ 0 & \text{otherwise} \end{cases}.$$

So $h_3(0) = 0$, $h_3(\frac{1}{4}) = 0$, $h_3(\frac{1}{2}) = \frac{1}{2}\sqrt{2}$, and $h_3(\frac{3}{4}) = -\frac{1}{2}\sqrt{2}$.

Putting this all together, we finally get:

$$H_4 = \frac{1}{2}\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ \sqrt{2} & -\sqrt{2} & 0 & 0 \\ 0 & 0 & \sqrt{2} & -\sqrt{2} \end{bmatrix}.$$

(Note that Gonzalez and Woods have one of these values wrong.)

The $2 \times 2$ Haar transform is easier to develop since $J = 1$, and so $0 \le p \le 0$. I.e., $p$ can only be zero. For $k = 1$ this gives $1 = 2^0 + q - 1$, and so $q = 1$.

$h_0(z) = \frac{1}{\sqrt{2}}$ for all $z$, which fixes the first row.

$$h_1(z) = h_{01}(z) = \begin{cases} \frac{1}{\sqrt{2}} & \text{if } 0 \le z < \frac{1}{2} \\ -\frac{1}{\sqrt{2}} & \text{if } \frac{1}{2} \le z < 1 \\ 0 & \text{otherwise} \end{cases}.$$

The $h$ functions are evaluated at $z = 0$ and $z = \frac{1}{2}$, giving the matrix:

$$H_2 = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}.$$

Gonzalez and Woods, in Figure 7.8, shows the result of applying the $H_4$ Haar discrete wavelet transform to an image, but doesn't explain a lot of concepts that are going on in the figure. E.g., one might get the impression from that figure that Haar will produce a reduced copy of the original image in the top left corner of the transformed image, but this doesn't happen. As I'll discuss below, this reduced image is actually scattered through the transformed array, but Gonzalez and Woods have reorganized the array structure to make it more visually intuitive. They also don't show how easy it is to take the transformed image and restore the original. I'll look at the restoration problem first, and then describe what is going on in Figure 7.8 and all other transformed images. Finally I'll look at how to use Haar in lossless and lossy compression schemes.

## Restoring the Original Image from the Transform

Look at the rows of $H_4$ and $H_2$ as vectors. If we take the dot product of any pair of these row vectors then the result is zero, which means that in $H_4$ the four rows form orthogonal vectors in 4-space and in $H_2$ they form orthogonal vectors in 2-space. In addition the magnitude of each row vector (remembering to include the $\frac{1}{\sqrt{N}}$ multiplier outside the matrix) is equal to 1. Technically this means that these (and all other Haar matrices) are orthonormal, which also means that the inverse of each Haar matrix is its transform.

The discrete Haar transform formula is, as mentioned above,

$$T = HFH^{T}.$$

Because $H$ is orthonormal this can be rewritten as:

$$T = HFH^{-1},$$

and so

$$H^{-1}TH = F.$$

Reusing the orthonormal property means that we can restore $F$ from $T$ using:

$$F = H^{T}TH.$$

## Producing the Transform Matrix

As discussed above, the image shown in Figure 7.8 of Gonzales and Woods is not the Haar transform image, but has been rearranged for visual convenience. This rearrangement is very standard, but you should know what is being done to get the image. I'll use $H_2$ in the discussion below, since $2\times2$ leads to simpler descriptions, but the extension to, say, $H_4$ and $4\times4$ is trivial. First, remember that:

$$H_2 = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}.$$

We divide the matrix up into $2\times2$ blocks, and apply the transform to each block to get the block in the transformed image. E.g., if the $2\times2$ block has $a$ and $b$ in its first row and $c$ and $d$ in its second row the transform will be

$$H_2\begin{bmatrix} a & b \\ c & d \end{bmatrix}H_2{}^{\mathrm{T}} = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}\begin{bmatrix} a & b \\ c & d \end{bmatrix}\frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

Noting that the transpose of $H_2$ is the same as $H_2$, which is not true for other Haar matrices. This gives the transformed $2\times2$ block:

$$\frac{1}{2}\begin{bmatrix} a+b+c+d & a-b+c-d \\ a+b-c-d & a-b-c+d \end{bmatrix}.$$

Look at what is going on here. The top left is a lowpass or mean filter. The top right is an average horizontal gradient, and the bottom left an average vertical gradient. The bottom right gives diagonal curvature.

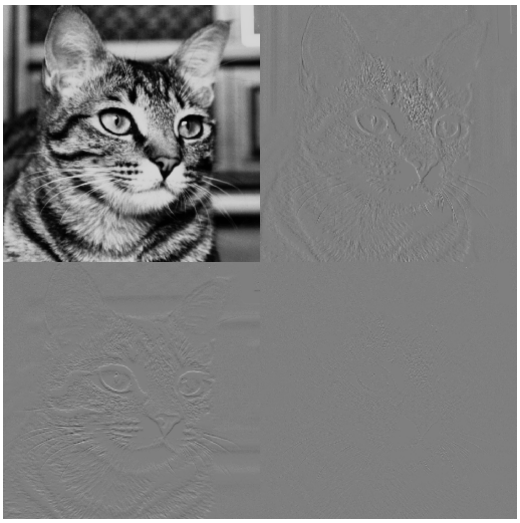
Applying this to an image of Peppy on the left, below, gives the result on the right, which doesn't have any of the structure shown in, say, Figure 7.8 of Gonzalez and Woods. However the cat is clearly recognizable for two reasons. One is that every fourth pixel in the new image is an average of four pixels in the original, and the other is that the image also includes the vertical, horizontal, and diagonal curvature filter information. The Peppy image was $512\times512$ with 256 gray levels. The transform produced values between -255 (e.g., $a = b = 0$, $c = d = 255$ on the lower left) and +510 (all four variables = 255) on the top left. So the transformed image was

normalized to be between 0 and 255 for display by adding 255 and dividing by 3.

Peppy

Peppy with 2×2 Haar Transform

The form in Gonzalez and Woods Figure 7.8 is the result of moving the pixels around. In this 2×2 case we'll get four tiles, each 256×256, where the top left corner is all of the top left pixels in each block, the top right is all of the top right pixels in each block, etc. So we get the result shown below:



Peppy with tiled Haar Transform

I've done the color balancing differently here. The top left has an intensity range of 0 to 510, so I've halved those intensities to get 0 to 255. The other

three tiles have a range of -255 to +255, and so for them I've added 255 and then halved. I.e., the adjustment is not consistent through the image. The relevant code from my program for getting the original transform and the tiled transform, assuming obvious definitions and with `SIZE` #defined as 512 is:

```
for (i = 0; i < SIZE; i = i + 2) {
    for (j = 0; j < SIZE; j = j + 2) {
        a = inarray[i][j];
        b = inarray[i][j + 1];
        c = inarray[i + 1][j];
        d = inarray[i + 1][j + 1];

        outarray[i][j] = (a + b + c + d) / 2;
        outarray[i][j + 1] = (a - b + c - d) / 2;
        outarray[i + 1][j] = (a + b - c - d) / 2;
        outarray[i + 1][j + 1] = (a - b - c + d) / 2;

        out0to255[i][j] = (outarray[i][j] + 255) / 3;
        out0to255[i][j + 1] = (outarray[i][j + 1] + 255) / 3;
        out0to255[i + 1][j] = (outarray[i + 1][j] + 255) / 3;
        out0to255[i + 1][j + 1] = (outarray[i + 1][j + 1] + 255) / 3;

        iby2 = i / 2;
        jby2 = j / 2;
        sizeby2 = SIZE / 2;
        prettyout[iby2][jby2] = outarray[i][j] / 2;
        prettyout[iby2][jby2 + sizeby2] =
                  (outarray[i][j + 1] + 255) / 2;
        prettyout[iby2 + sizeby2][jby2] =
                  (outarray[i + 1][j] + 255) / 2;
        prettyout[iby2 + sizeby2][jby2 + sizeby2] =
                  (outarray[i + 1][j + 1] + 255) / 2;
    }
}
```

If I take the `outarray` values and use them as input to the same program I get Peppy back, as expected.

## Using Haar for Lossless Compression

I'll look at lossless compression here and then I'll look at lossy compression in the next part of these notes. For lossless compression I'll use $H_2$ as the example again, although as we'll see the compression is much better for the higher $H$ transforms.

With $H_2$ we get four tiles using the tiled view. The top left tile doesn't compress any better than the input image, but the other three tiles are clustered very close to zero. E.g., for Peppy over 80% of the pixels in these three tiles are between -10 and +10, leaving only 18% of the pixels with absolute values between 11 and 255. Even better, there are only seven pixels with absolute values greater than 127, out of a total of 196,608 pixels. So using some kind of variable length coding like Huffman coding some huge gains can be achieved for the three filter tiles.

If we were the use a higher $H$ transform like, say, $H_8$, then the reduced tile in the top left will now only contain $\frac{1}{64}$ of the pixels in the image, and so the compression will be over the remaining $\frac{63}{64}$ of the pixels. This means that significant compression can be achieved with Huffman coding or similar systems.

## Using Haar for Lossy Compression

This is where one can win spectacularly using Haar. Most of the output of the Haar transform is, as I've discussed before, centered around zero. E.g., even for the Peppy image, which has some extreme differences in intensity, for $H_2$ about 16% of the pixels in the three modified tiles are at zero, and 81.9% are between -10 and +10. Since these are representing relatively low change areas, one can take a band of them around zero and change them all to zero without affecting image quality when the original image is restored. E.g., for the image below I applied $H_2$ to Peppy and then changed all values in the three filter tiles that were between -10 and +10 to zero. Then, when I restored it I got the image shown.

I've looked carefully at this image and the original Peppy side by side on my screen at $512 \times 512$ each, and haven't been able to see any differences at all.

This means that using $H_2$ with this change over 80% of the values in the 75% of the image that represents filters have the same value (zero), and so any image compression scheme like Huffman or run-length encoding will be able to make huge gains.

If I were to use, say, $H_8$, instead, I'd be able to get similar compression savings, but now over $\frac{63}{64}$ of the file.