## Problem Statement (part 1)

**Description:** DCT transformation refers to the transform between image spatial domain and frequency domain. In the first part of the assignment, you are required to realize a simplified version of DCT based compression.

**Consider the following sequence of values:**

10 11 12 11 12 13 12 11
10 -10 8 -7 8 -8 7 -7

(a) Transform each row separately using an eight-point DCT. Plot the resulting 16 transform coefficients
(b) Combine all 16 numbers into a single vector and transform it using a 16-point DCT. Plot the 16 transform coefficients
(c) Compare the results of (a) and (b). For this particular case would you suggest a block size of 8 or 16 for greater compression? Justify your answer

A simple way of doing it (at least for me!) is by creating a pseudo-code algorithm for a nested for-loop, the outer one being for the 'u' value and inner being for the sum 'i' values. So:

```
for (u = 0; u < N; u++)
        for (i = 0; i < N; i++)
                dct[u] += alpha(u) * cosine(PI * u * (2*i + 1) * 1/2N) * f(i);
```

'u' being the result value of the dct coefficient.
f(i) = the value of the current element we are working on from the matrix given.
alpha(u) = 1/sqrt(8) for u=0 and sqrt(2/8) for u != 0

(a) Table for eight point DCT

| F(0) | F(1) | F(2) | F(3) | F(4) | F(5) | F(6) | F(7) |
|---|---|---|---|---|---|---|---|
| 32.52 | -1.2814 | -1.3065 | 0.4499 | -1.4142 | 1.1716 | 0.5211 | 0.2548 |
| 0.35355 | 4.2405 | 0.3498 | 5.0462 | 2.4748 | 1.1716 | 1.7685 | 20.388 |

(b) Result table for a 16 point DCT

| F(0) | F(1) | F(2) | F(3) | F(4) | F(5) | F(6) | F(7) |
|---|---|---|---|---|---|---|---|
| 23.25 | -3.9117 | -0.6764 | 8.0641 | 0.75 | -6.1407 | 1.6332 | -14.2364 |
| **F(8)** | **F(9)** | **F(10)** | **F(11)** | **F(12)** | **F(13)** | **F(14)** | **F(15)** |
| 2.2873 | 14.2364 | -1.6332 | 6.1407 | -0.75 | 1.1716 | 0.6764 | 3.9117 |

(c) I would suggest using the 8 point DCT because it will hold more low frequency values therefore giving a better image quality for the amount of compression made. The 16 point will hold less low frequency values and more high frequency values, so using a 16 point can discard the high frequency values, technically having high compression but terrible quality.

## Problem Statement (part 2)

- Description: 2D image compression.
- Implementation: Convert image f1 from RGB to HSI, obtain intensity image I.

## Algorithm Design

I created the function 'OnAssignment3Hsi' where there are mainly 6 variable that obtain the results needed: Integers red, blue, green and doubles hue, saturation, and intensity. All of the pixels of the image are obtained and each value of every pixel is assigned to their corresponding color variable (red goes to int red, green goes to int green, etc). The intensity function, being the easiest and less computational needed for it, was calculated first. This is done by grabbing the average of the RGB values, so simply by adding them together and dividing them by 3.

```
blue = pImg[i*iWidth*3+j*3];      //B
green = pImg[i*iWidth*3+j*3+1];     //G
red = pImg[i*iWidth*3+j*3+2];      //R
all = red+green+blue;
//intensity
if (all != 0)
        intensity = all/3;
```

where intensity is between [0, 255]

Once that was computed, I proceeded to compute the hue by using the formula:

```
//hue
param = (red-green)*(red-green) + (red-blue)*(green-blue);
hue = (0.5*(red+red-green-blue))/(sqrt(param));
if (blue <= green) hue = acos(hue);
else hue = 2*PI - acos(hue);
hue = hue*180/PI;
```

After applying the formula, I normalized the result by converting it from gradient to degrees (multiplying by 180/PI).
Finally the saturation was computed after obtaining the intensity by using:

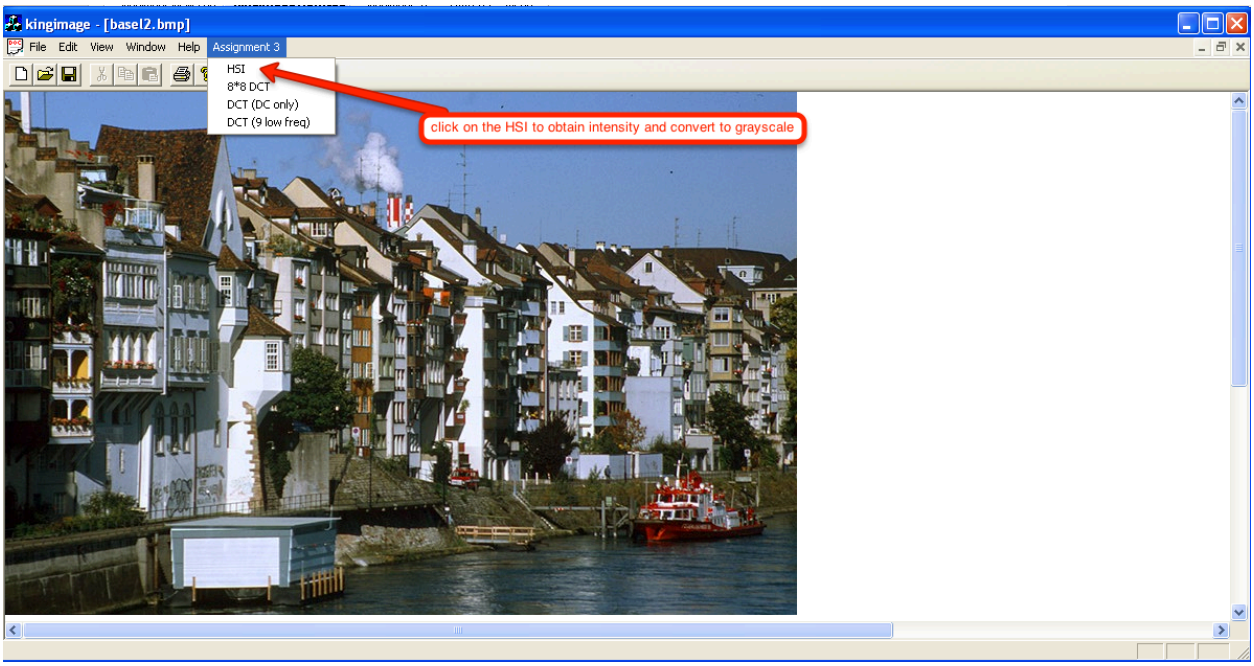
```
//saturation
saturation = 1 - ((min(red,(min(blue,green))))/intensity);
```

where saturation was between [0,1].

**NOTE: In order to show the resulting intensity, I applied the intensity to all RGB values, making the image convert from RGB to grayscale.

## Manual

Load an image through the default open file. Once the file has been loaded, click on the menu 'Assignment 3' and then on the submenu 'HSI'.




## Problem Statement (part 2)

- Description: 2D image compression.
- Implementation: Apply 8*8 DCT transform to image I, obtain frequency domain image F

## Algorithm Design

Created the function 'OnAssignment39()' in which 3 global variables are the main elements. I decided to create them globally static for the sake of performance. Without them, it would've taken about 4-5 seconds to process each image, but now it takes less than 1 second. These are an 'intensity' array that maintains the intensity value for each pixel of the image. This array is recalculated in every function and I do repeat myself, but again for the sake of performance instead of calling a function. Another static global variable is a cosine array that maintains the 8*8 values of all the cosine computations, that way it only computes the values once throughout the whole program. And lastly, a dct 8*8 array that would hold all of the coefficients.
So what this function does is the following. Enters a loop of the size of the image but in intervals of 8, that way the main loop gets reduced by n/8. Inside the loop, the dct array is calculated by the sum of the product of the pixel values times the cosine values that have already been precomputed. So:

```
for (int x=0; x<8; ++x)
for (int y=0; y<8; ++y)
{
        dct[x][y] = 0;
        for (int u=0; u<8; ++u)
        for (int v=0; v<8; ++v)
        {
                dct[x][y] += intense[i+u][j+v]*cosine[x][u]*cosine[y][v];
        }
        dct[x][y] = alpha(x)*alpha(y)*dct[x][y];
}
```

Finally I multiply the current result by the static function alpha(u):

```
static double alpha(int i){
        if (i==0) return 1.0/sqrt(8.0);
        return 0.5;
}
```

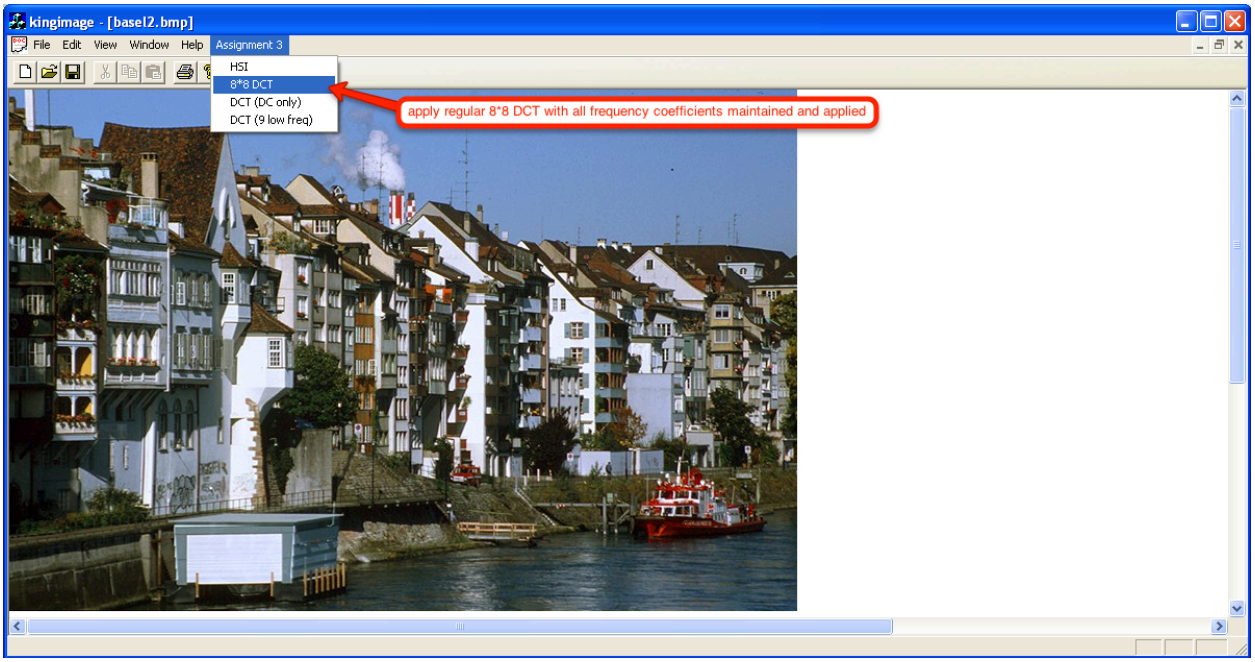
which returns 0.3535 for 0 and 0.5 for all other values.
Finally, once the DCT table is calculated, I calculate the IDCT in order to project on the screen the result. Theoretically, the image should remain the same except for the color which I converted to grayscale.

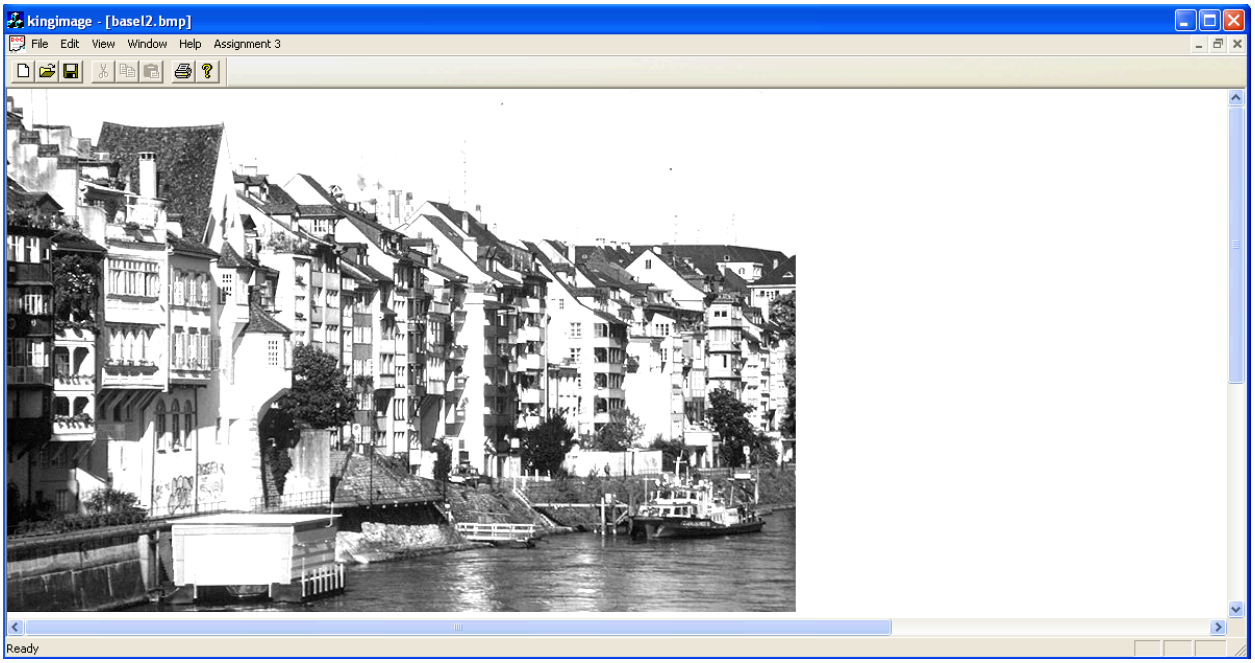
**Bugs:**

However, I believe there is a bug in my code that I just cannot figure out, where the resulting image returns much brighter than it should. I know my formulas are correct (tested them in many different ways) and I know that my math is right. The problem is in the DC component of the DCT coefficients, it doesn't seem to be correctly calculated even though all other coefficients are right. If I remove the DC from the DCT matrix, the image is the correct color but wouldn't be the correct compression. So I commented out an if-statement that would correct this in case someone wants to test it out.
Also the images have to be an 8*8 compatible image, that is, iHeight % 8 = 0 and iWidth % 8 = 0. The program would not crash otherwise, but the algorithm will ignore the remaining pixels.

## Manual

Load an image through the open module and select a compatible image multiple of 8 pixels. The image **must** be a multiple of 8 to work perfectly!


kingimage - [basel2.bmp]
File Edit View Window Help Assignment 3
HSI
8*8 DCT
DCT (DC only)
DCT (9 low freq)
apply regular 8*8 DCT with all frequency coefficients maintained and applied


kingimage - [basel2.bmp]
File Edit View Window Help Assignment 3
Ready

## Problem Statement (part 3)

- Description: 2D image compression.
- Implementation: In each 8*8 frequency block, keep DC component and remove all other frequency components from F, obtain frequency domain image D1.

## Algorithm Design

Function 'OnAssignment39()' takes care of this part of the assignment. This description will be a little vague because it is the exact same method as described in the previous problem. I grab the static global variables and reuse them by loading them with the current image values. Once I get to the computation of the DCT matrix, is where the difference comes in. The basic idea behind using only the DC component and removing all others, consists in keeping the very first value computed in the matrix and set all others to 0. Since all of the values of the matrix have already been initialized to 0 at the beginning of the loop, the only value needed to be computed is dct[0][0]. I do this by setting the loop to only repeat once:

```
for (int x=0; x<1; ++x)
for (int y=0; y<1; ++y)
{
	for (int u=0; u<8; ++u)
	for (int v=0; v<8; ++v)
	{
		dct[x][y] += intense[i+u][j+v]*cosine[x][u]*cosine[y][v];
	}
	dct[x][y] = alpha(x)*alpha(y)*dct[x][y];
}
```

So I compute that exactly the same way as previous problem, and finally, once I have the computed DCT matrix, I can compute the IDCT to display my results. I do this by leaving everything unmodified from the previous function and let it do its work.

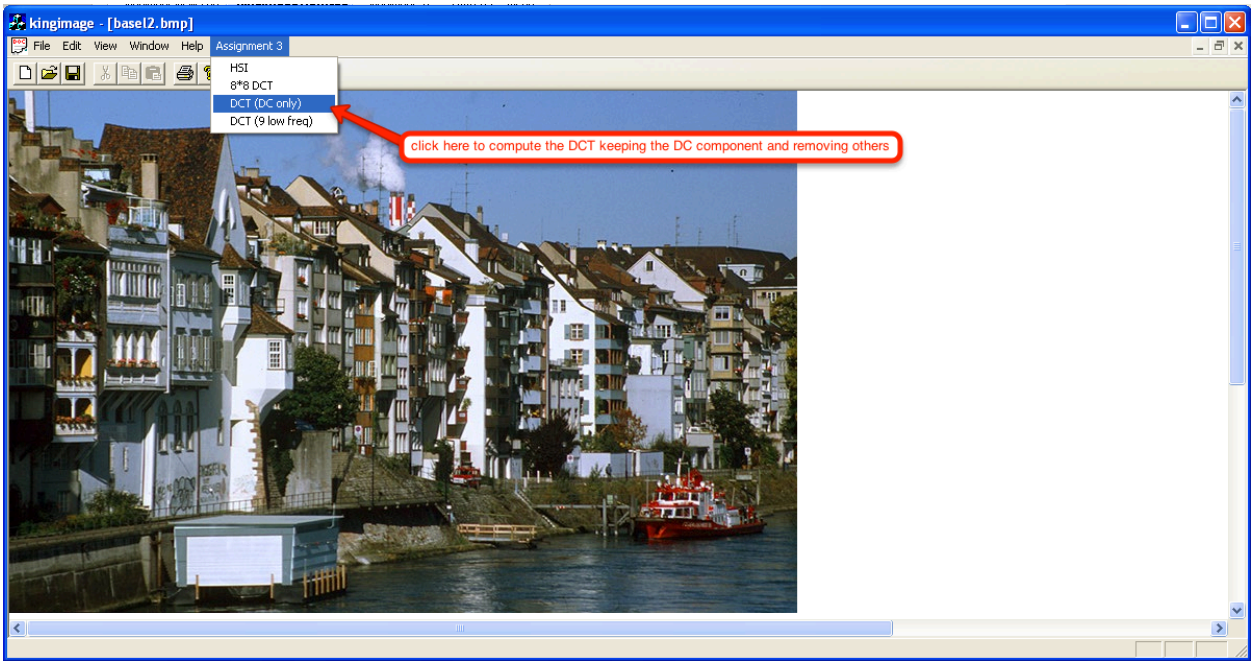
**Bugs:**

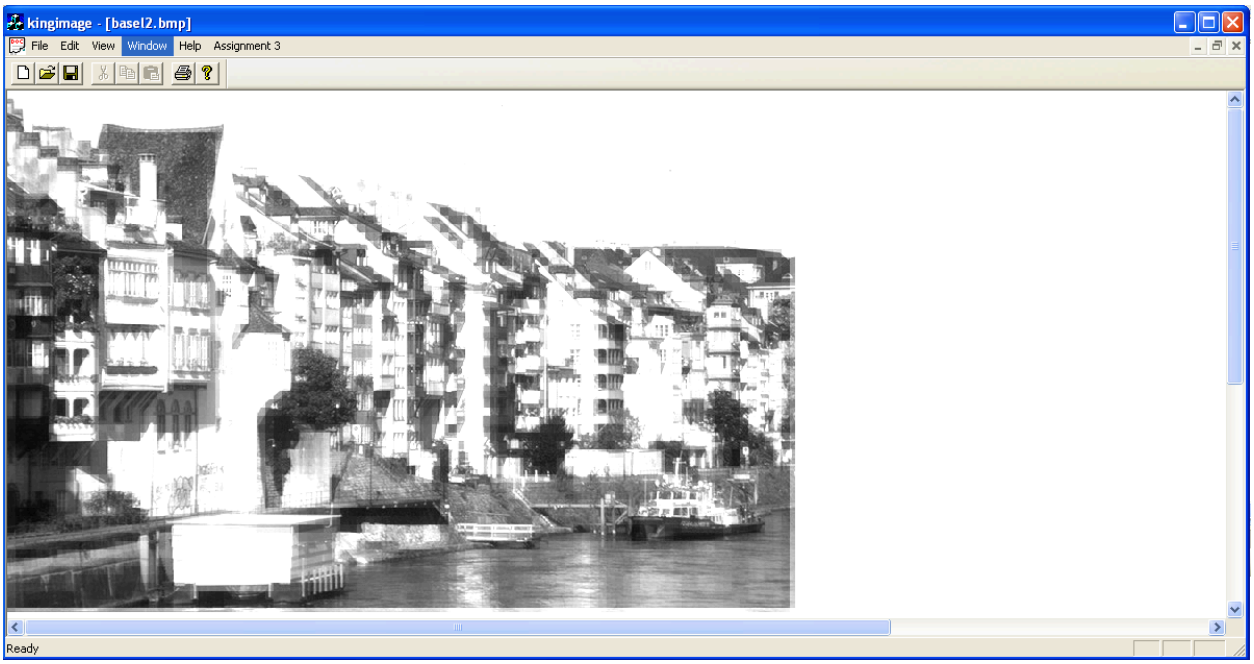
It also has the same bugs as problem 2. The image needs to be a multiple of 8 both in height and width and the image is also brighter than it should be because I am only using the DC component (which is the cause of this bug).

```
for (int u=0; u<8; ++u)
for (int v=0; v<8; ++v)
{
	for (int x=0; x<8; ++x)
	for (int y=0; y<8; ++y)
	{
		intense[i+u][j+v] += alpha(x)*alpha(y)*dct[x][y]*cosine[x][u]*cosine[y][v];
	}
	if (intense[i+u][j+v] > 255) intense[i+u][j+v] = 255;
	if (intense[i+u][j+v] < 0) intense[i+u][j+v]= 0;
}
```

# Manual

Load an 8*8 image through the default open module. Then click on Assignment 3, DCT (DC only) menu option.



## Problem Statement (part 4)

- Description: 2D image compression.
- Implementation: Similar to (3), in each 8*8 frequency block, keep first 9 low frequency components and remove all other high frequency components from F, obtain frequency domain image D2.

## Algorithm Design

This part of the assignment is handled in function 'OnAssignment310()'. The main idea behind this problem is to create the DCT coefficients and use only the 9 low frequency components. That is, use the first 3x3 matrix elements inside the DCT array. Similar to problem 3, I do this by limiting the loop to be only the first 3 elements of both x and y; this grabs the first 3x3 elements from the 8x8 matrix and sets all other elements to 0 since they were already set to 0.

```
for (int x=0; x<3; ++x)
for (int y=0; y<3; ++y)
{
        dct[x][y] = 0;
        for (int u=0; u<8; ++u)
        for (int v=0; v<8; ++v)
        {
                dct[x][y] += intense[i+u][j+v]*cosine[x][u]*cosine[y][v];
        }
        dct[x][y] = alpha(x)*alpha(y)*dct[x][y];
}
```

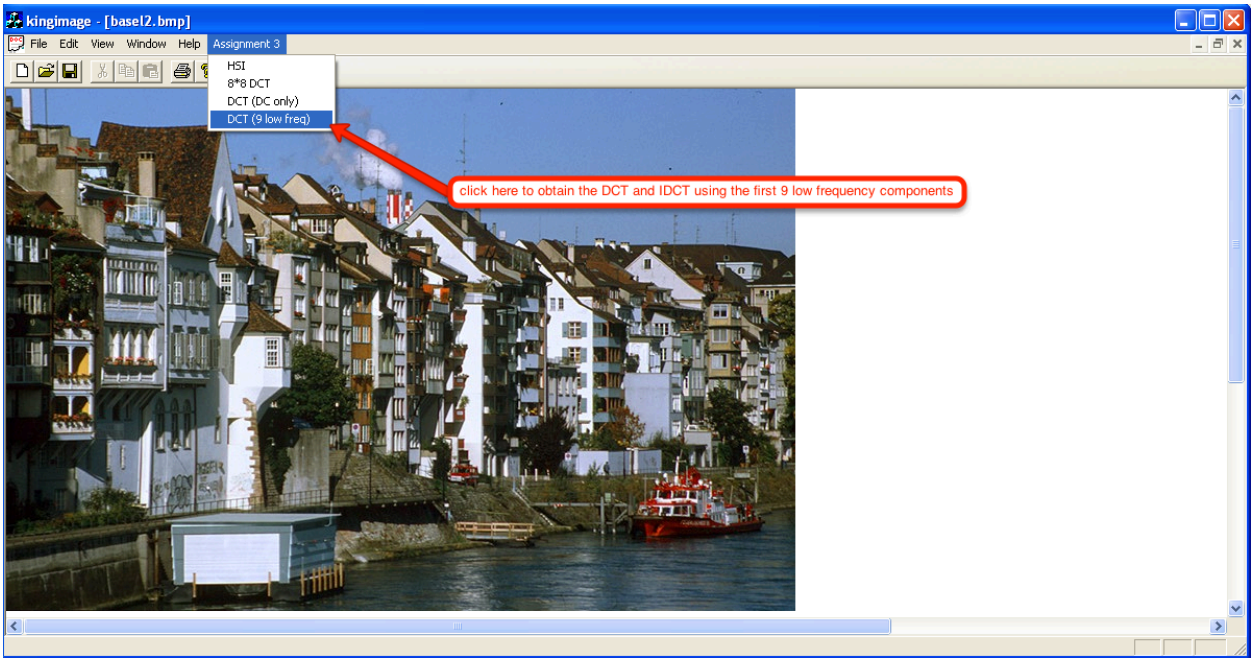
Everything else (besides the matrix) in this part of the assignment is identical to part 2 and 3. After obtaining the DCT, I proceed by obtaining the IDCT and displaying the resulting image in the window.

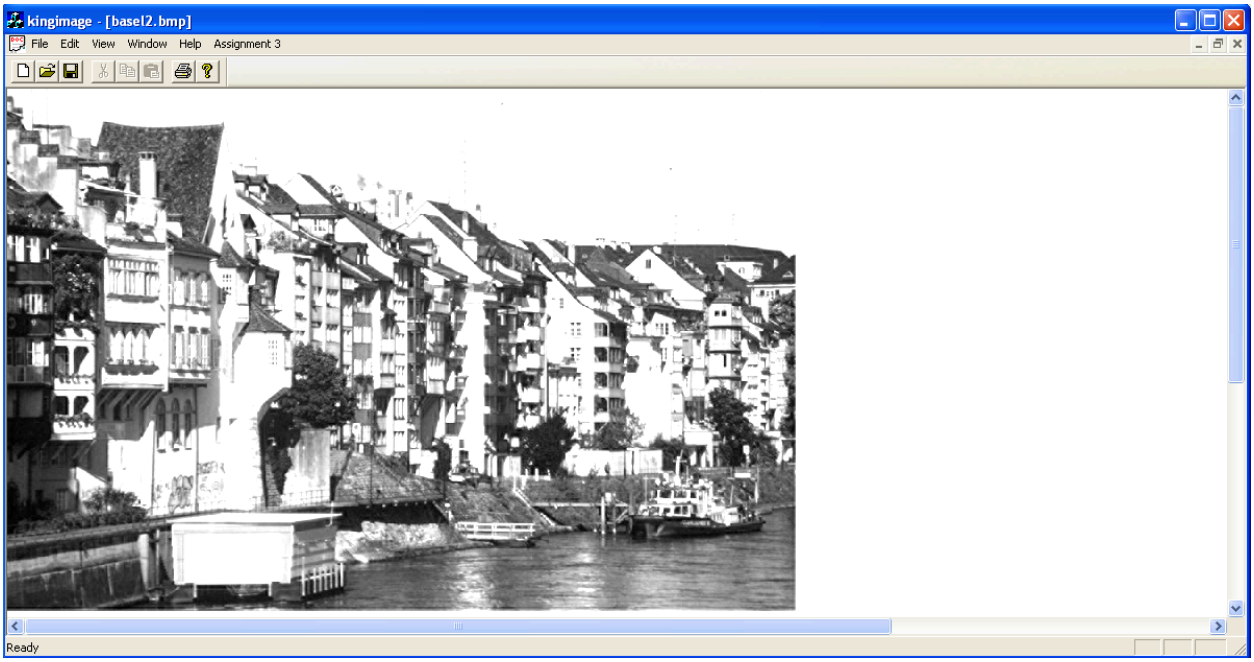
**Bugs**:

Same bugs as in part 2 and 3. The image must be a multiple of 8 in both height and width in order to work perfectly. Also, the resulting image has that increased brightness that I cannot figure out how to fix.

## Manual

Using the open module provided by the main frame window, open a compatible, multiple of 8 pixel image.




## Problem Statement (part 4)

- Description: 2D image compression.
- Implementation: Apply IDCT on D1 and D2, obtain image R1 and R2. Explain which image is more blurred, and why.

I already applied the IDCT to all images but I'll explain which is more blurred and why.
Since for part 3, we only used the DC part of the DCT matrix, the resulting image is much more 'pixelized' than any other, that is, the 8*8 blocks were really obvious and the quality was fairly low because the compression was really high. On the other hand, the other image was using the 9

low frequency components, which means that it would have a higher quality and more detailed image than the previous one. So R2 is more blurred (less blocks) than R1.