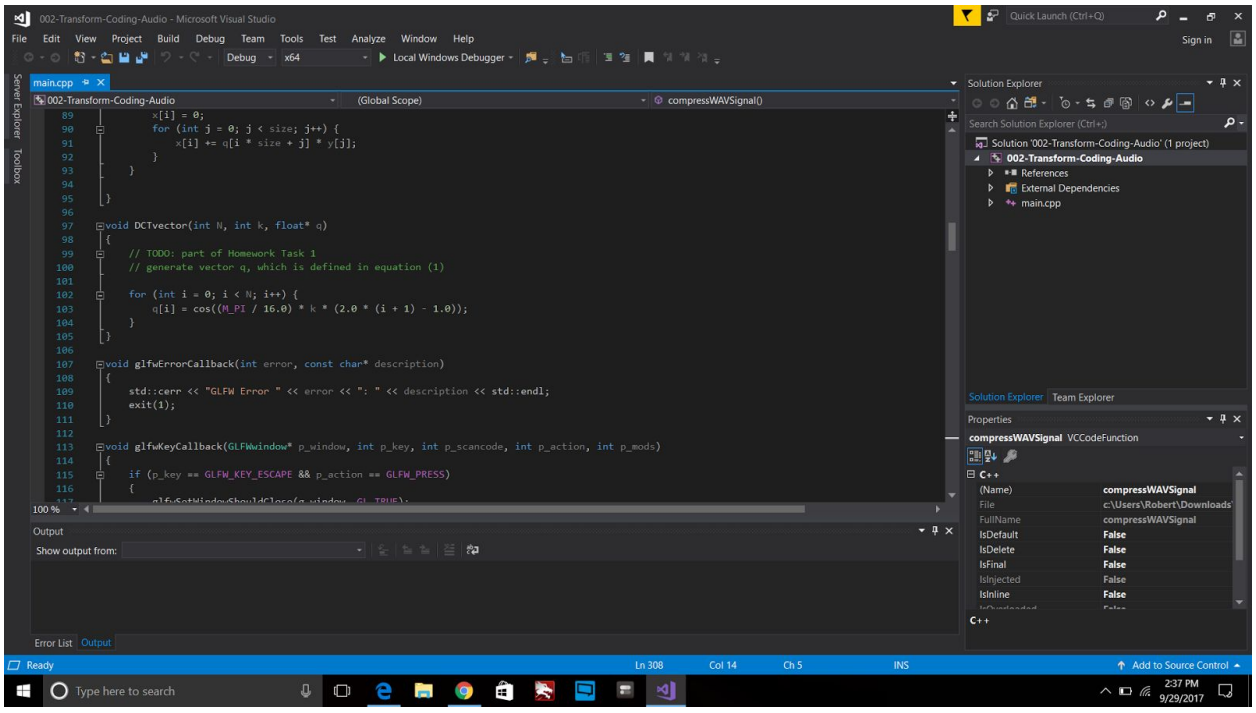
Robert Weischedel
CS 4600
9/29/17

Assignment 2: Transform Coding

Audio Compression

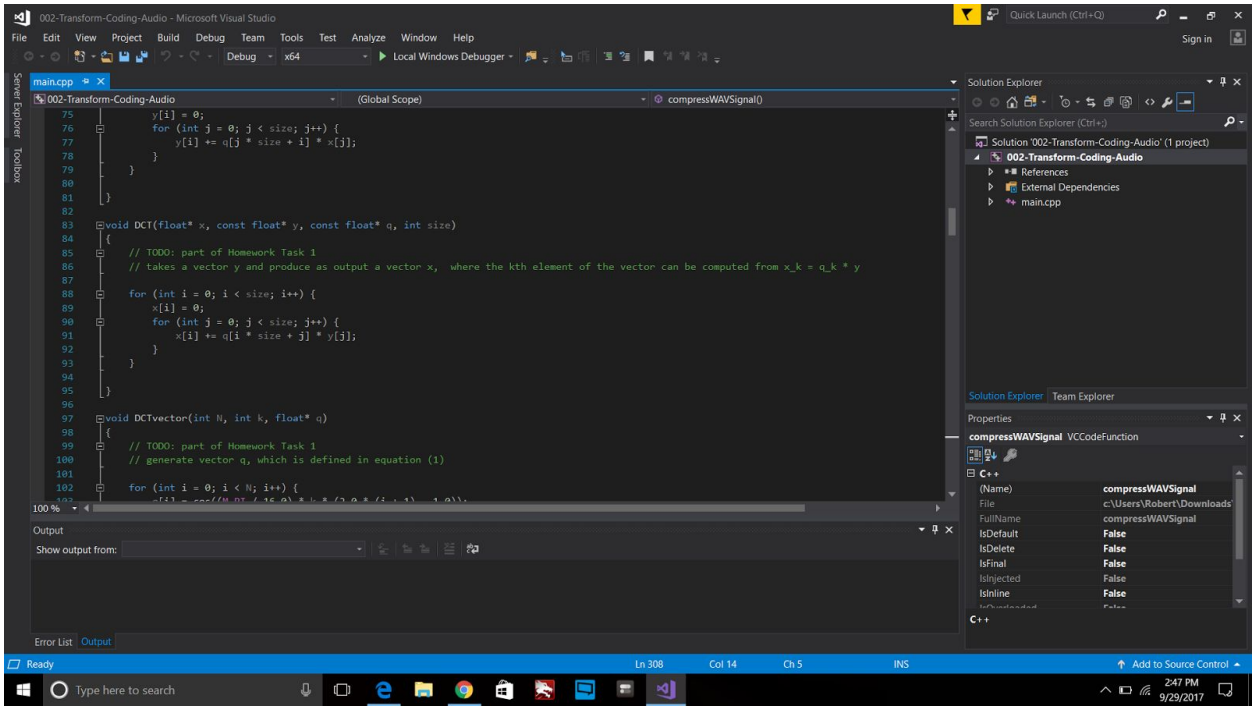
For this part of the assignment I followed the equations and walkthrough that were given during lecture for solving a 1D DCT. Let us take a look at each of the three methods that I wrote and also the little snippet of code I added in compressWAVSignal.

DCTvector Function



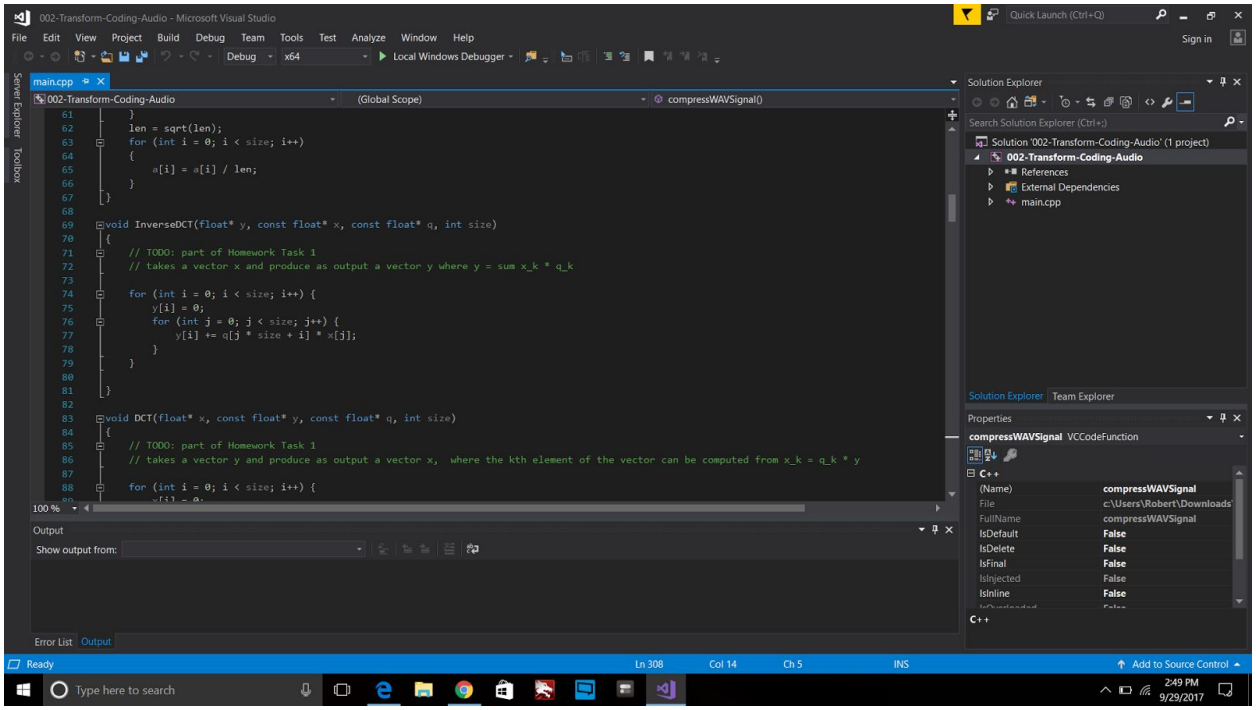
For this function I followed closely to equation 1 which was given in the assignment handout. In order to perform a DCT Transformation I need to construct the 8 orthonormal vectors that span R8. I do this by using this function by creating each of the q_i vectors one by one. This function loops through a size of N, which is the size of each q_i vector I need and store the all results of the equation in the vector at each indice.

DCT Function



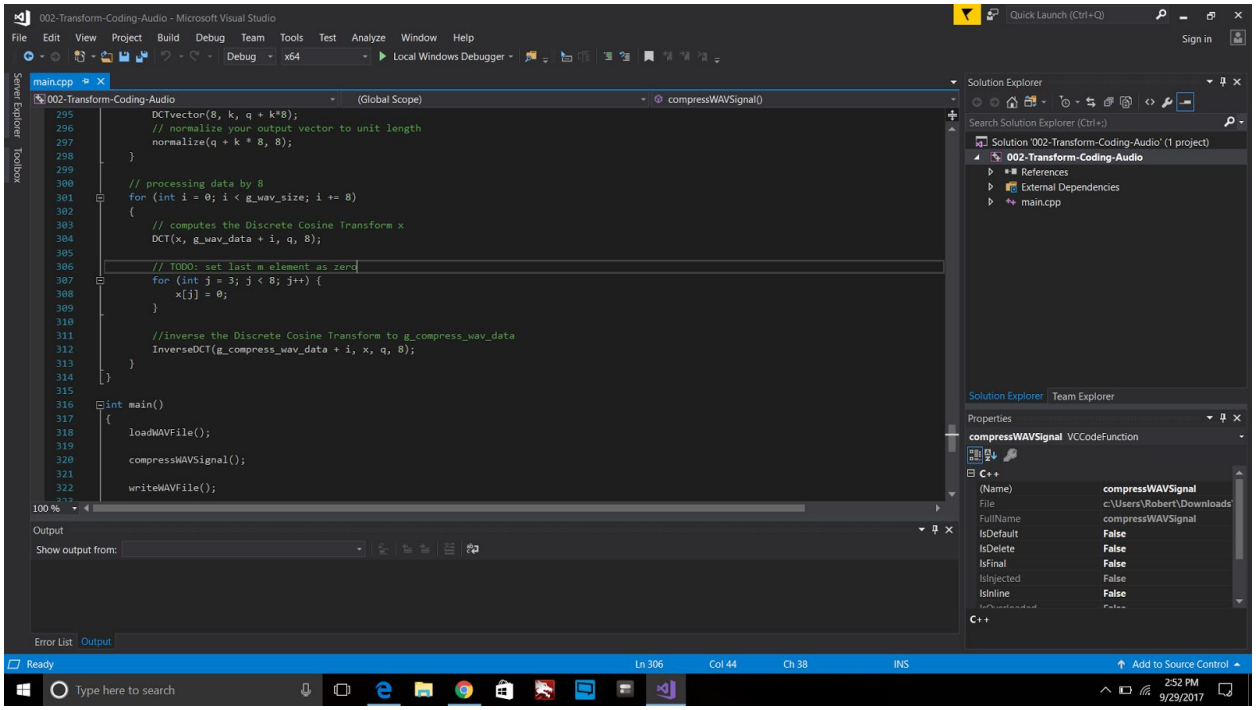
This function is the heart of the DCT Transformation, as it actually performs the transformation and compression of the audio image. For my solution I used a double nested for loop design that would allow me to reach all the correct values in each of the three arrays we are working with in this function. The first for loop or the i loop, is where I index the x[] or the output array, in this loop I also ensure to zero out the x[] at that particular location first to ensure no garbage data makes its way into the compression process. Then I move into my inner loop or my j loop, which is used to index the correct g_wavdata and q_i and multiply those values together and then store that value into the compressed array x[].

InverseDCT Function



This function is the exact opposite of the DCT function. It takes the compressed image array and converts it back to its original state. Or at least as close to the original as we possibly can, since the transformations we are doing are not lossless. So for this function I used the same array set up except I switched the indice math and the arrays that I performed in the original DCT function and returned the “uncompressed” values back into the y array which is a uncompressed audio image array from the x array which contained the compressed audio image.

compressWAVSignal Function



While this function was mostly filled out, we were asked to write a small snippet of code to ensure that the last m elements are zero. Or in other words each of the q_i vectors return a wave, and for this transformation we are only concerned with the first 3 waves, q_1, q_2, q_3. And not the rest so, we simple just loop through the x[] array and set the last 5 q_i vector values equal to zero.

Output:
For this first experiment I have to say that I am impressed and excited at my results I was successfully able to compress and uncompress the audio file, while maintaining the heart of the audio. As you listen to the out file you will hear some slight distortion, but the audio integrity is almost completely there. Here are the audio images graphs, on the left is the original file and on the right is the uncompressed file.

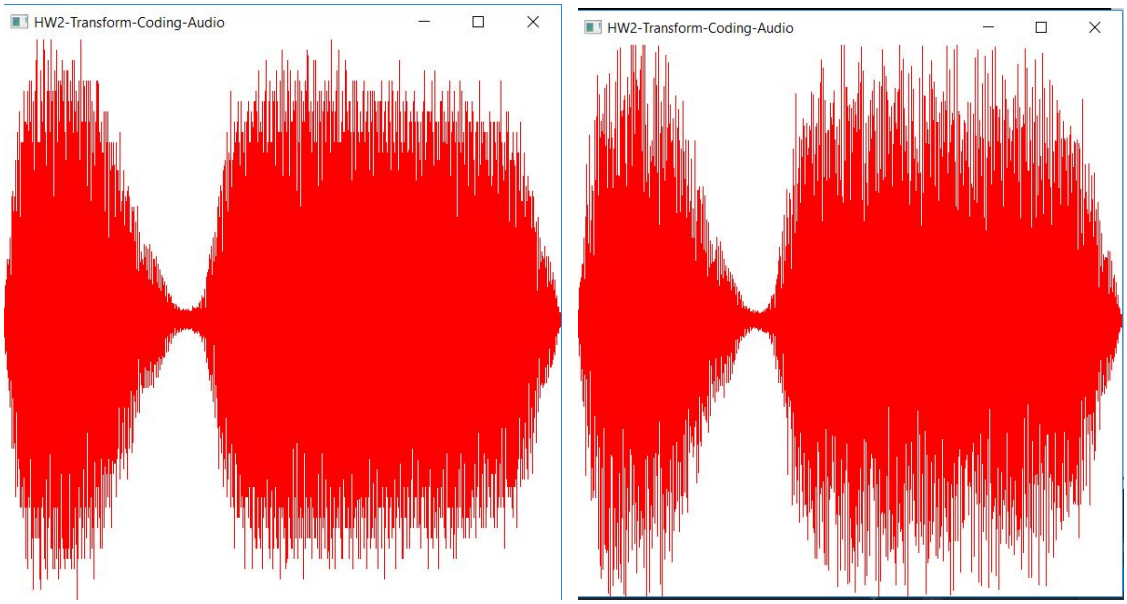
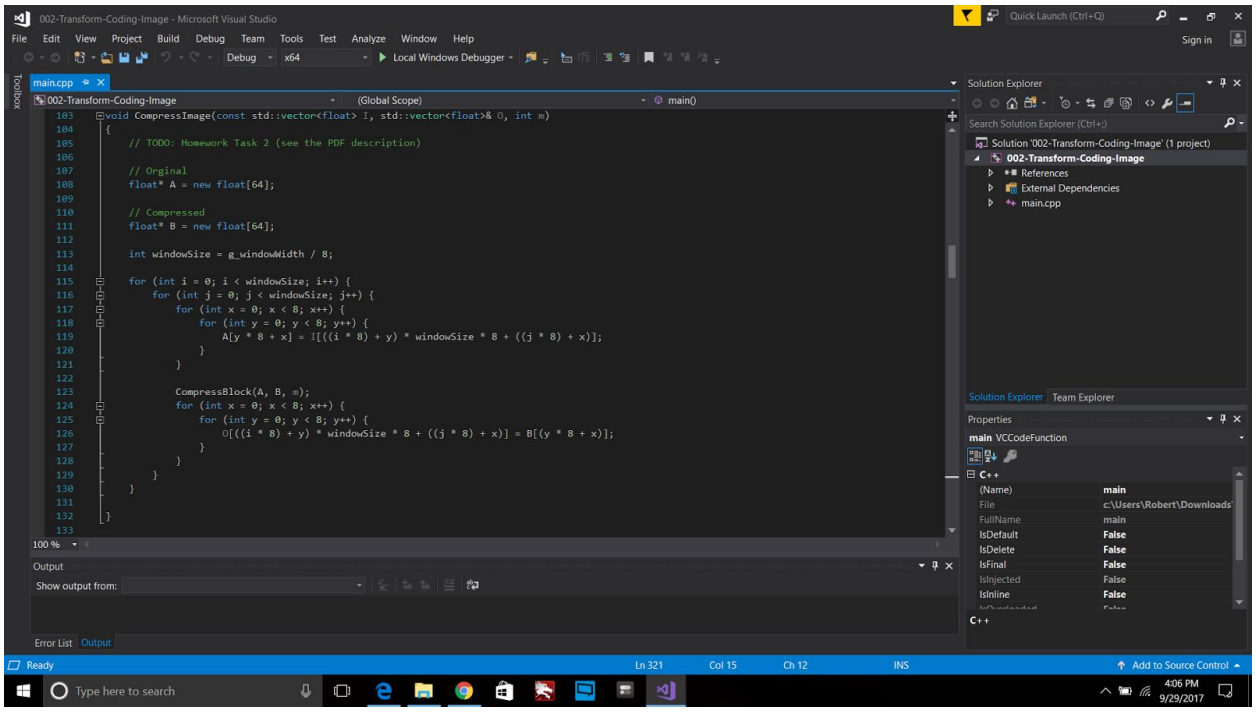


Image Compression

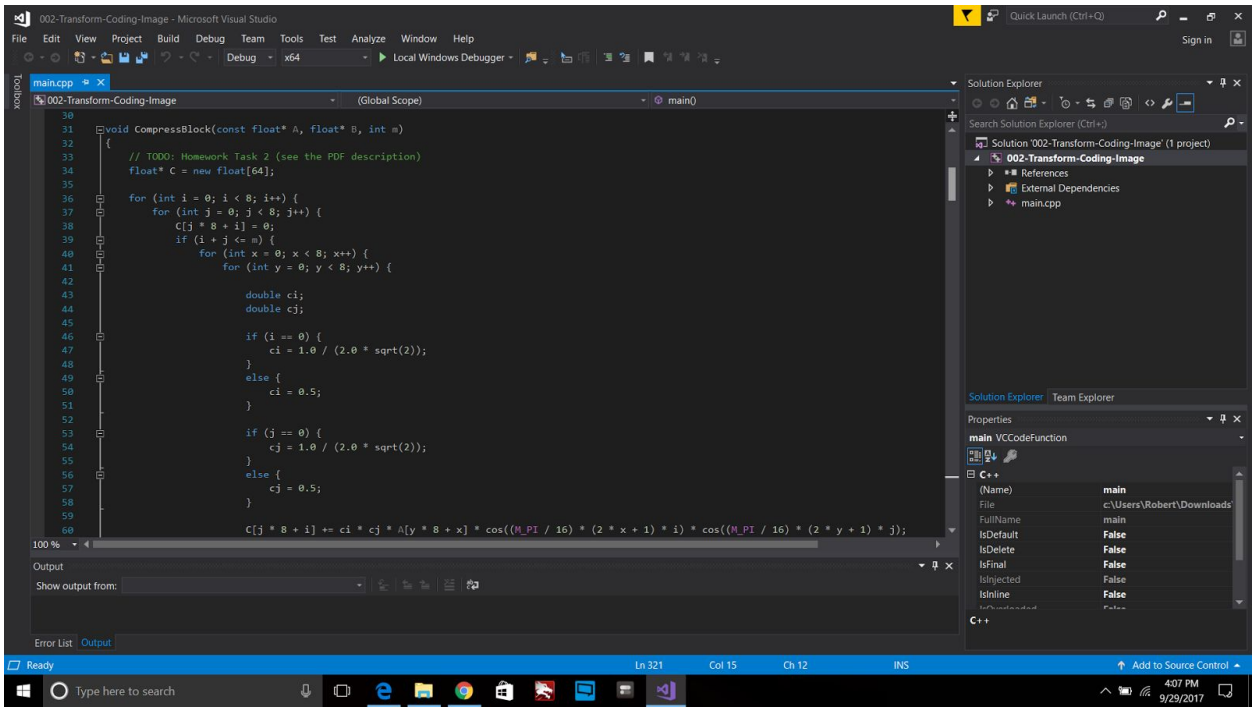
For this portion of the assignment I had to rely a bit more on the slides for help than I did for the first portion of the assignment. Specifically I used the lecture 5 slides as they cover the heart of the 2D version of the DCT Transformation equation. Let us take a look at the two larger methods I had to complete for this portion.

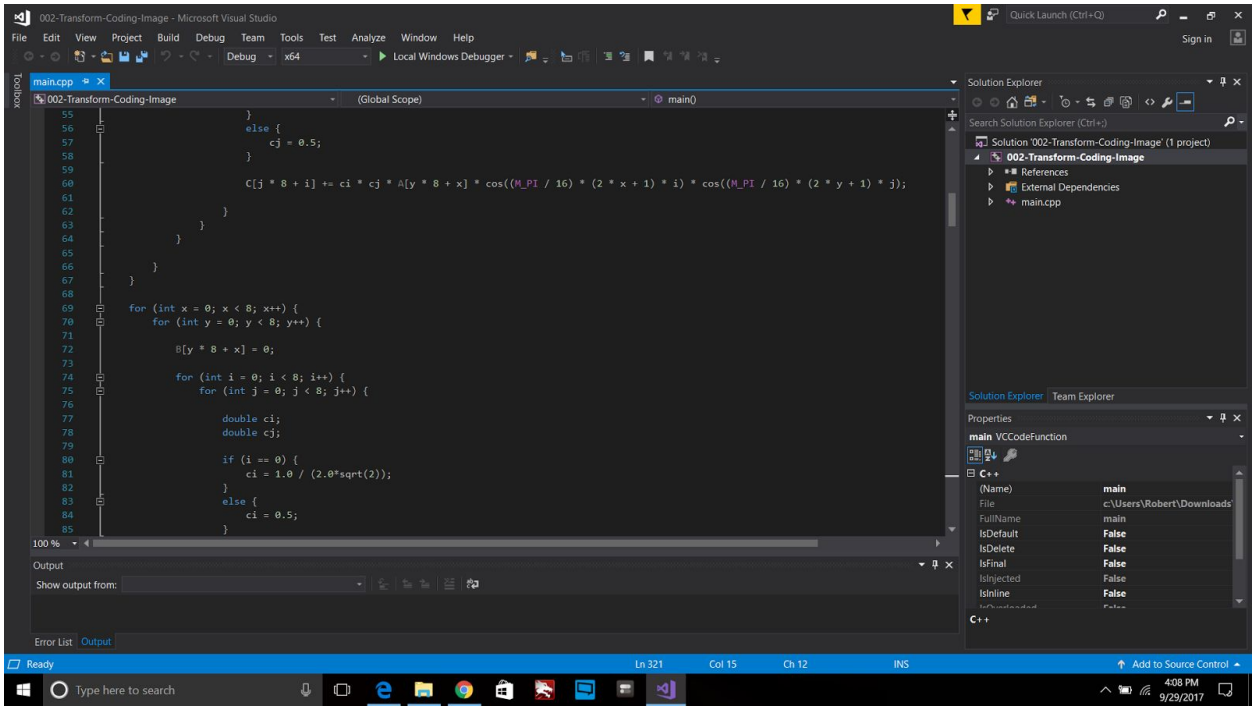
CompressImage Function

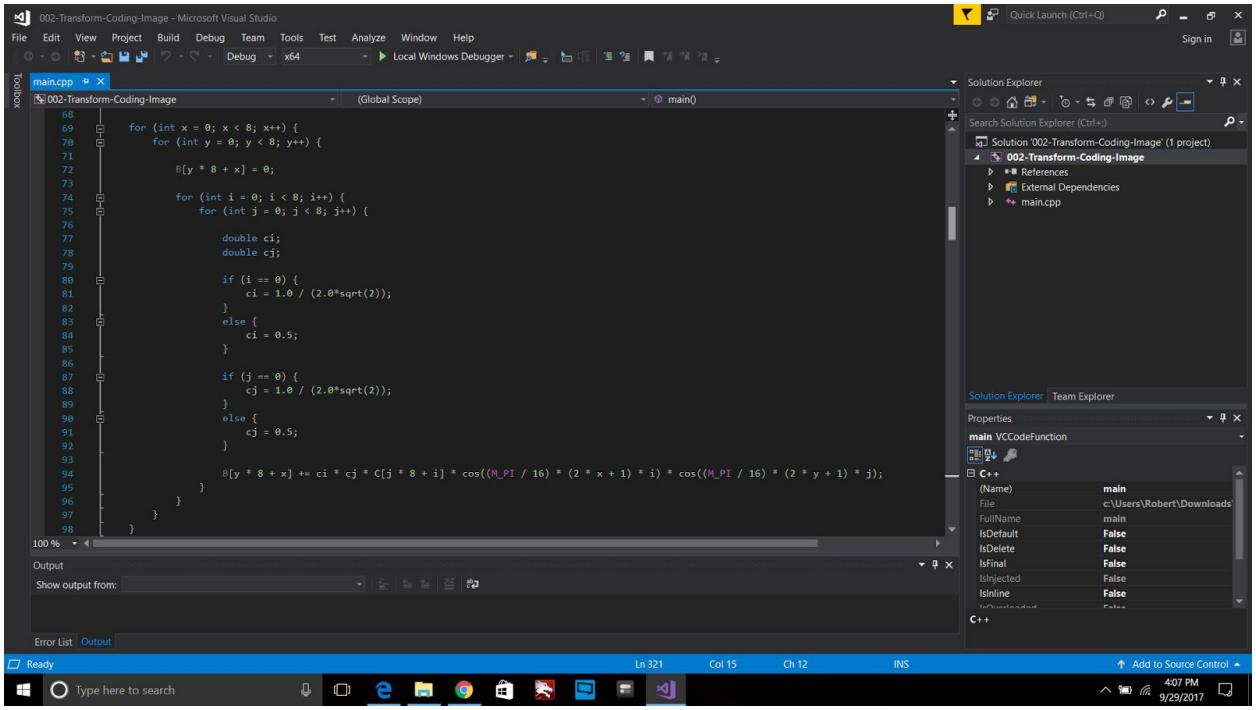


This function is used to help compress the entire image by breaking down the image I into blocks for the CompressBlock function to work on. Then after the CompressBlock function is finished it returns the compressed block and places that compressed output into the O or output array. For my implementation I used two arrays A, B to represent the original and compressed versions. The first two loops break the total image up into the 8 * 8 blocks and then the other two loops assist in accessing the correct portion of the input array in order to be stored into the A array. After CompressBlock is called and the compressed block is returned, I use two more loops to store the 8 * 8 block into the output array.

CompressBlock Function







This function is the heart of what I had to do for the compression and uncompression of the image, it takes the individual blocks of image and compresses it and then turns around and immediately decompresses it. The first set of for loops takes the 8 * 8 block that is given from the CompressImage function, break it down to its individual components, calculate the variables needed for the summation equation, perform the required summation for compression, and then store it into the C or coefficients array. The summation equation I use, I derived from the slides.

Then I simply do the opposite to decompress each block. Which takes up the second portion of the function, and looks almost identical to the first half, except the indices math is a bit different, and we are performing the summation on the coefficients, and storing it into the output array B.

Output:
I am happy to report that the methods work exactly as they should. The more you increase the m value the cleaner the outputted uncompressed image file is. After about 7 or 8 it becomes harder to tell the differences between the two and after 10 the changes are almost negligible. See below for the examples for the different m values.

Original

0

2

4

10

16