Arith Design Document

Emily Yuan (eyuan01) & Changwoo Heo(cheo01)

Compression Step 0: Read Ppm & Trim Image

Description: This step involves reading in the provided ppm file and if the width or height is not even, it will trim the image dimension that is odd by 1 to make it even.

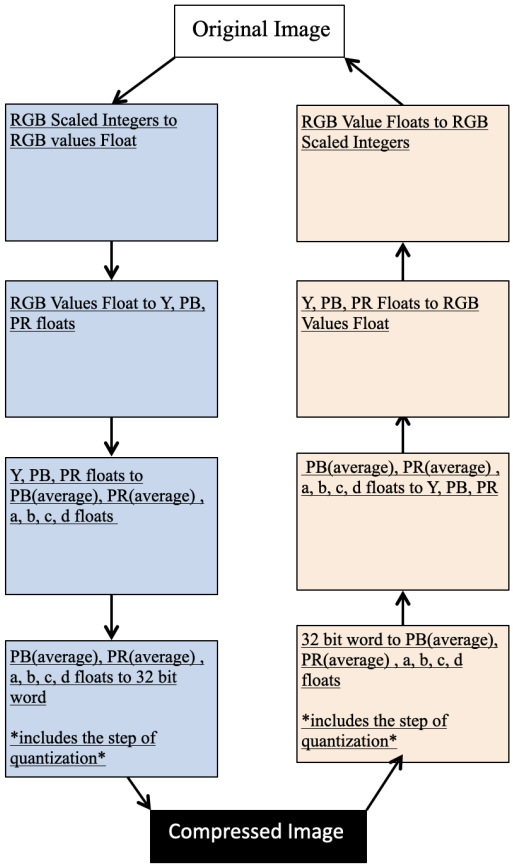
Input:

- File pointer to provided file (from command line or standard input)
- A pointer to method suite of type A2Methods_T

Output: Pnm_PPM constructed after reading and trimming, will have the new width/or new height and corresponding rgb information

Information Lost: There will be information lost when the width or height is not even as then we would have to trim one of the rows/columns. However, if the given image already has a width and height of size of even

Testing: Display the image and see if the image is the way it is intended to look like. If the process includes trimming, we can click on the information button (the i button) to check that the width and height are even numbers.



COMMON TESTING FOR EVERY STEP: RUN IT ON VALGRIND

Compression Step 1: RGB Scaled Integers to RGB values Float

Description: This step involves converting the RGB colors of each pixel from scaled integers, which range from 0-255 to floats that are from 0-1. Conversion to floats creates RGB color space, allowing us to conduct further mathematical operations such as when converting to the component video.

Input:

- An image containing pixels with RGB colors value as scaled integers, read from either standard input or specified on command line
- A pointer to method suite of type A2Methods_T

Output: A 2d array, which stores a struct that can hold three fields defined as float values each representing the color intensity of red, green and blue

Information Lost: There will be some information lost as floats in C can store up to seven decimal digits. Due to the following characteristics, that would mean all the decimal digits after the seventh one would not be stored and rounded up, producing a slightly imprecise data of the color intensity.

Testing: Assert float values are in range and printf or pipe rgb pixel values as floats and manually conduct the calculations and compare results to what our output is (not all of them, but only for a few to gain assurance)

Decompression Step 4: RGB Values Floats to RGB scaled integer

Description: This step involves converting the RGB pixels of each color from float values back to scaled integers. This makes the image compatible for outputting a standard ppm file.

Input:

- A 2d array, which stores a struct that can hold three float values which represents the color intensity of red, green and blue
- A pointer to method suite of type A2Methods_T

Output: Pnm_ppm converted_image, image containing pixels with RGB colors represented as scaled integers ranging from 0 - 255.

Information Lost: There will be information lost as when floats are converted back to integers, we are planning to multiply the denominator to the scaled values. In C, when floats are converted to integers, all the decimal points are left behind. For instance, if the value obtained from multiplying the denominator to the float is 210.9834… or 210.547 it would be converted to 210. Therefore, there would definitely be information lost, reducing the color intensity by a slight amount.

Testing: Use our implementation of ppmdiff to check the E value of the original image and then the image that went through compression step 1 and decompression step 4. The two images should relatively be similar since it was converted back to an integer and have an E value within the range of 0.1% to 1.5%, , if not debug decompression step 4 or compression step 1.

Compression Step 2: RGB Values Float to Y, P_B, P_R floats

Description: This step involves changing each pixel from RGB Color space as float values to component video color space Y, P_B, P_R floats. We would use the conversion equations provided in the spec to convert it to the corresponding values.

Input:

- the 2d array outputted by compression step 1 which contains a struct that contains three float values which represents the color intensity of red, green and blue as its element
- A pointer to method suite of type A2Methods_T

Output: A new instance of 2d array, which stores a struct that can hold three fields defined as float values each representing the Luminance, Chrominance (Blue) and Chrominance (Red)

Information Lost: There will be some information lost as floats in C can store up to seven decimal digits. When we conduct our calculations, since we are multiplying float pixel values with decimals to obtain the Y, P_B, P_R, all the decimal digits after the seventh one would not be stored and rounded up. This would produce a slightly imprecise data for the luminance and chromaticity values.

Testing: printf or pipe Y, P_B, P_R pixel values as floats and manually conduct the calculations and compare results to what our output is (not all of them, but for a few of them to gain assurance)

Decompression Step 3: Y, P_B, P_R floats to RGB Values Float

Description: This step involves converting the component video color space Y, P_B, P_R pixels back to the RGB values represented as floats. We would use the inverse conversion equations provided in the spec to convert it to the corresponding values.

Input:

- 2d array, which stores a struct that can hold three fields defined as float values each representing the Luminance, Chrominance (Blue) and Chrominance (Red)
- A pointer to method suite of type A2Methods_T

Output: new instance of 2d array that holds a struct storing three fields designs as float values each representing the color intensity of red, green, and blue

Information Lost: There will be some information lost as floats in C can store up to seven decimal digits. When we conduct our calculations, since we are now doing the inverse computation, we have to multiply the Y, P_B, P_R values with decimals. All the decimal digits after the seventh one would not be stored and rounded up. This would produce slightly imprecise data for retrieving the RGB values as floats.

Testing: Use our implementation of ppmdiff to check the E value of the original image and the image that went through compression step 1 and 2, then decompression step 4 and 3 in order. Assert that the value is within the range of 0.5% to 1.5%, if not debug decompression step 3 or compression step 2.

Compression Step 3: Y, P_B, P_R floats to $\overline{P_B}$, $\overline{P_R}$, a, b, c, d floats

Description: This step involves taking the average value of P_B, P_R, converting them to four bit values, and using the DCT to transform the four Y values into coefficients a,b,c,d.

Input:

- 2d array returned by compression step 2, which stores a struct that can hold three fields defined as float values each representing the Luminance, Chrominance (Blue) and Chrominance (Red) as its element
- A pointer to method suite of type A2Methods_T

Output: A new 2d array with the width and height of the UArray2_T blocks stored in the struct defined in UArray2b_T, where each cell, represents a 2 by 2 block, and stores a defined struct holding six fields defined as float values each representing $\overline{P_B}$, $\overline{P_R}$, a, b, c, d

Information Lost: Information can potentially be lost because when we average out the P_B, P_R values, all the P_B and P_R the value inside the block would be averaged out, meaning we would lose some information. However, when we are converting the Y values to a,b,c,d, using the DCT, we would not be losing information because the equation is rather straightforward and there is no need to round up since the values we are converting to hold floats as well.

Testing:

- Attempt to manually calculate values and compare with our own output (do on few)
- Ensure that we can print out every field in our struct for that new array and print its values, compare to manually calculated values

Decompression Step 2: $\overline{P_B}$, $\overline{P_R}$, a, b, c, d floats to Y, P_B, P_R floats

Description: This step involves retrieving the values of P_B, P_R, a, b, c, d and converting it back to the float values of P_B, P_R. We will retrieve the original Y float value using the inverse DCT calculation given in the spec.

Input

- the 2d array outputted by compression step 3 that stores a struct holding the six fields that we are trying to convert
- A pointer to method suite of type A2Methods_T

Output: A 2d array, which stores a struct that can hold three fields defined as float values each representing the Luminance, Chrominance (Blue) and Chrominance (Red)

Information Lost: Like the compression step, information would be lost because when we convert the average P_B, P_R values back to P_B, P_R, we would obtain the values that have been averaged out. Therefore, we would not have the original values that we started out with. Similarly, converting a, b, c, d into Y floats involves a straightforward equation, where both sides of the equation hold floats, so there should not be any information lost.

Testing: Use our implementation of ppmdiff to check the E value of the original image and the image that went through compression step 1, 2 and 3, then decompression step 4, 3, and 2 in order. Assert that the value is within the range of 0.5% to 1.5%. , If not debug the compression step 2 or compression step 3.

<u>Compression Step 4</u>: $\overline{P_B}$, $\overline{P_R}$, a, b, c, d floats to 32-bit words

Description: This step involves converting the float values for the color space components P_B, P_R and the DCT coefficients a, b, c, d back to scaled integers. We then use the bitpack interface to pack $\overline{P_B}$, $\overline{P_R}$, a, b, c, d into a 32 bit word sequence. Then, we can use this to print out the compressed image.

Input:

- A 2d array, outputted from compression step 3 that stores a struct that holds the float values of $\overline{P_B}$, $\overline{P_R}$, a, b, c, d for each block
- A pointer to method suite of type A2Methods_T

Output: A new 2d array of the width and height of the UArray2_T blocks stored in the struct defined in UArray2b_T, where each cell stores the 32 bit word converted from the six floats values calculated from packing the 2d array, given as input, into a block with the blocksize of 2.

Information Lost: There will be information lost as when quantization occurs, it has a hard barrier for the value that it can be quantized to. For example, for b, c, and d, if the quantized value is outside of the boundary of plus or minus 15, it is stored as 15, or -15 depending on the sign.

Testing:

- Print statements to check for correct conversion from float to scaled, should be valid numbers
- We can possibly separate this step into two, where we first convert values to scaled integers and then pack those integers into the bit word. Therefore, we can test one step at a time and by the end of compression when our bits have been packed, we would be able to unpack them and then compare those values to what we had before.

<u>Decompression Step 1</u>: 32 bit words to $\overline{P_B}$, $\overline{P_R}$, a, b, c, d floats

Description: This step involves unpacking the bit words and then retrieving the six elements, storing them in our struct and converting them back to float values.

Input:

- The compressed image file with a sequence of 32-bit code words, one for each 2-by-2 block of pixels

Output: A new 2d array that stores a struct that holds the float values of $\overline{P_B}$, $\overline{P_R}$, a, b, c, d for each block

Information Lost: There is information lost as anything quantized value that is outside the boundary of -15 to 15 is set to -15 or 15 based on the sign. Therefore, when converting it back to $\overline{P_B}$, $\overline{P_R}$, a, b, c, d, it won't be the exact same values as the ones produced from compression step 3.

Testing: Use our implementation of ppmdiff to check the E value of the original image and the image that went through compression step 1, 2, 3 and 4 then decompression step 4, 3, 2, and 1 in order. Assert that the value is within the range of 0.5% to 1.5%, if not debug the decompression step 1 or compression step 4.