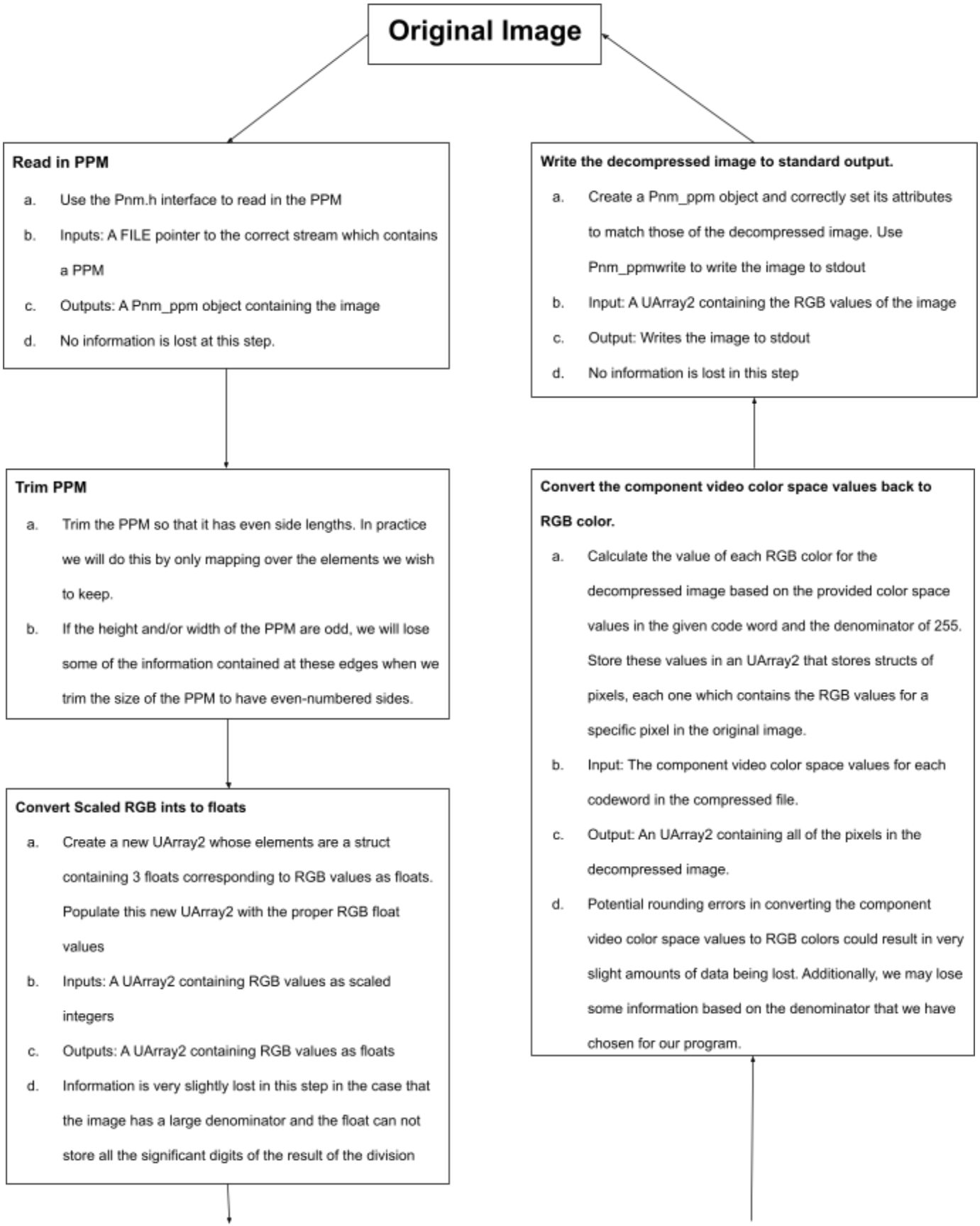
Original Image
Read in PPM
a. Use the Pnm.h interface to read in the PPM
b. Inputs: A FILE pointer to the correct stream which contains a PPM
c. Outputs: A Pnm_ppm object containing the image
d. No information is lost at this step.
Trim PPM
a. Trim the PPM so that it has even side lengths. In practice we will do this by only mapping over the elements we wish to keep.
b. If the height and/or width of the PPM are odd, we will lose some of the information contained at these edges when we trim the size of the PPM to have even-numbered sides.
Convert Scaled RGB ints to floats
a. Create a new UArray2 whose elements are a struct containing 3 floats corresponding to RGB values as floats. Populate this new UArray2 with the proper RGB float values
b. Inputs: A UArray2 containing RGB values as scaled integers
c. Outputs: A UArray2 containing RGB values as floats
d. Information is very slightly lost in this step in the case that the image has a large denominator and the float can not store all the significant digits of the result of the division
Convert the component video color space values back to RGB color.
a. Calculate the value of each RGB color for the decompressed image based on the provided color space values in the given code word and the denominator of 255. Store these values in an UArray2 that stores structs of pixels, each one which contains the RGB values for a specific pixel in the original image.
b. Input: The component video color space values for each codeword in the compressed file.
c. Output: An UArray2 containing all of the pixels in the decompressed image.
d. Potential rounding errors in converting the component video color space values to RGB colors could result in very slight amounts of data being lost. Additionally, we may lose some information based on the denominator that we have chosen for our program.
Write the decompressed image to standard output.
a. Create a Pnm_ppm object and correctly set its attributes to match those of the decompressed image. Use Pnm_ppmwrite to write the image to stdout
b. Input: A UArray2 containing the RGB values of the image
c. Output: Writes the image to stdout
d. No information is lost in this step

Convert RGB floats to component video color space

A. Create a new UArray 2 whose elements are a struct containing 3 floats corresponding to Y, P_b, P_r values. Populate this new UArray2 with the converted values.

B. Inputs: A UArray2 containing the RGB values as floats

C. Outputs: A UArray2 containing the converted color space values

D. Information is very slightly lost due to rounding errors in the calculation of the Y, P_b, P_r values

Compute 32-bit word of each 2x2 block of pixels

A. Create a new UArray2 whose side lengths are half of the trimmed image. Each element of this UArray2 is a 32 bit char array, thus it is 8 chars long. Start by computing the average P_b, P_r values and Discrete Cosine Transform of 2x2 blocks, then use the bitpack interface to create the code words.

B. Inputs: A UArray2 containing color space values

C. Outputs: A UArray2 containing the 32-bit words corresponding to each 2x2 block of pixels

D. Information is lost in this step. When we average and quantize the values P_b and P_r for the four pixels, information is lost on the original values. When we compute the DCT coefficients, we still have enough information to recreate the Y values as we have 4 equations for 4 unknowns, but we lose information by truncating the values of "b, c and d" if they lie outside of the range [-0.3, 0.3]. Converting "a" into a 9 bit representation may result in lost information due to rounding errors.

Printing the compressed image

a. Print out the header and then the code words in row major order and big-endian form.

b. Inputs: A UArray2 with the packed code words

c. Outputs: The code words will be printed to stdout

d. No information is lost in this step.

Compute the Y values for the four pixels that are contained in the given codeword

a. Calculate the inverse of the discrete cosine transformation for the Y values for each pixel in the codeword, using the a, b, c, d provided from the given codeword.

b. Input: Each codeword in the compressed file will be inputted in this step.

c. Output: The Y values of each pixel that the codeword has stored information for.

d. Very small amounts of information could be lost due to rounding errors as we compute each Y value, especially since we are dealing with different types of variables (i.e. unsigned integers and signed integers).

Read the header of the compressed file and ensure that it is properly formatted. Create an UArray2 to store the contents of the decompressed image and put in a Pnm_ppm struct.

a. Allocate the 2D array using the width and height given from the header and the size of a pixel struct. Create a new Pnm_ppm struct and add the array to this struct.

b. For selecting our denominator, we will begin by choosing 255, as it is the typical denominator used for images.

Compressed Image

Implementation Plan

For each step, we will implement the compression and then implement the corresponding decompression step so that we can test each step of the program by compressing and decompressing test image files.

1. Reading and Writing the PPM (Compression Step 1 and Decompression Step 4)
    a. Test on images of different sizes both in dimension and file size. Test different methods of inputting an image. Use ppmdiff to ensure that no information has been lost.
    b. Test for invalid input files that can not be opened and ensure that proper error handling is done.
2. Converting between RGB and Color Space, and converting between Color Space and RGB (Compression Steps 2 - 4, and Decompression Step 3)
    a. Test on images of different sizes and via different input methods. Use ppmdiff to ensure that only very little information is being lost. Also test on images with different denominators to see how that affects our program.
3. Converting between Color Space and Code Words (Compression Step 5 and Decompression Steps 1 and 2)
    a. Test on images of different sizes and via different input methods. Use ppmdiff to ensure that not too much information is being lost. Check that the compressed file is around 3 times smaller than the source ppm as noted by the spec.
    b. For decompression, test that proper error handling occurs if the given file is too short, meaning that we stop in the middle of a codeword.

Finally we will test the whole program on various different inputs such as blank images, images with very disproportionate dimensions, small 1 by 1 images, and other edge cases we come up with. We will use ppmdiff to make sure that the decompressed images do not show significant loss. We will also unit test our other helper functions and the functions in the bitpack interface. Additionally, we will utilize valgrind to ensure any memory allocated during the program is freed, and that no memory leaks occur.