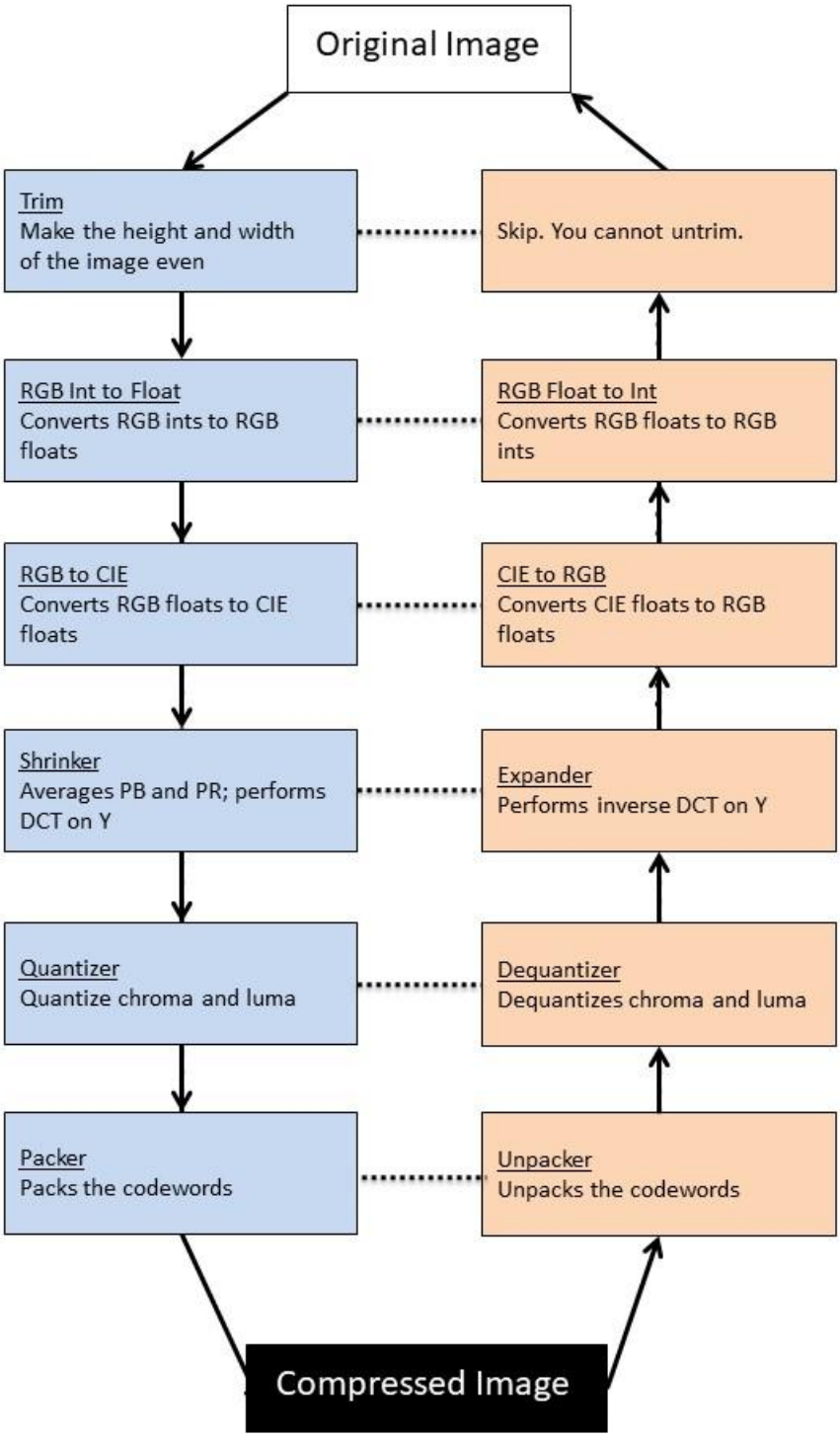
CS40 HW4 Arith Design

Ivi Fung (sfung02) and Matthew Wong (mwong14)



Module Descriptions

Compression	Decompression
Trim *Purpose:* Make the height and width of the image even *Input:* An image in the RGB int format *Output:* An image in the RGB int format with even dimensions *Lossy?* Yes, at most one row and one column is irreversibly removed from the image.	*(SKIP)* *Purpose:* There is no recovery of the trimmed edges *Input:* An image in the RGB int format *Output:* An image in the RGB int format, unaltered *Lossy?* No, as this operation does nothing.
RGB int to float *Purpose:* Converts RGB ints to RGB floats *Input:* An image with RGB ints (and denominator) as each pixel *Output:* An image with RGB floats as each pixel *Lossy?* Yes, as some precision is lost when converting an int to a float, because floats reserve bits for the sign and exponents.	*RGB float to RGB int* *Purpose:* Converts RGB floats to RGB ints *Input:* An image with RGB floats as each pixel *Output:* An image with RGB ints (and denominator) as each pixel *Lossy?* Yes, as it will lose some precision when going back to an int. In addition, some decimals will get cut off when going back to an int.
RGB to CIE *Purpose:* Converts RGB floats to CIE floats *Input:* An image with RGB floats as each pixel *Output:* An image with CIE floats as each pixel *Lossy?* Yes, as some precision is lost while conducting floating point multiplication.	*CIE to RGB* *Purpose:* Converts CIE floats to RGB floats *Input:* An image with CIE floats as each pixel *Output:* An image with RGB floats as each pixel *Lossy?* Yes, as some precision is lost while conducting floating point multiplication.
Shrinker *Purpose:* Averages P_B and P_R; performs DCT on Y *Input:* An image with CIE floats as each pixel *Output:* An image with luma (a, b, c, d) and chroma (P_B and P_R) as floats for each 2x2 block *Lossy?* Yes, as the chroma are irreversibly averaged together.	*Expander* *Purpose:* Performs inverse DCT on Y *Input:* An image with luma (a, b, c, d) and chroma (P_B and P_R) as floats for each 2x2 block *Output:* An image with CIE floats as each pixel *Lossy?* Yes, as the luma are put together via floating point math.

Quantizer *Purpose:* Quantize chroma and luma *Input:* An image with luma (a, b, c, d) and chroma (P_B and P_R) as floats for each 2x2 block *Output:* An image with luma (a, b, c, d) and chroma (P_B and P_R) as unsigned ints for each 2x2 block *Lossy?* Yes, as quantization buckets values in the same region.	*Dequantizer* *Purpose:* Dequantizes chroma and luma *Input:* An image with luma (a, b, c, d) and chroma (P_B and P_R) as unsigned ints for each 2x2 block *Output:* An image with luma (a, b, c, d) and chroma (P_B and P_R) as floats for each 2x2 block *Lossy?* Yes, as you go from ints to floats.
Packer *Purpose:* Packs into codewords *Input:* An image with luma (a, b, c, d) and chroma (P_B and P_R) as unsigned ints for each 2x2 block *Output:* A 2D array of codewords *Lossy?* No, as the bits are packed as if they are unsigned ints.	*Unpacker* *Purpose:* Unpacks the codewords *Input:* A 2D array of codewords *Output:* An image with luma (a, b, c, d) and chroma (P_B and P_R) as unsigned ints for each 2x2 block *Lossy?* No, as the bits are packed as if they are unsigned ints.

Addendum:

- All of the "rows" of the column will be linked together in a struct (which is called zip) that will allow us to make an array in main which will determine order of the decoding. We will do the same exporting trick as we did with a2plain.c and a2blocked.c

```
struct zip {
        void compress(Pnm_ppm image);
        void decompress(Pnm_ppm image);
};
```

- Each step takes in a Pnm_ppm, which has a different format for its UArray2 (described in its box above). They will have the compression step and decompression step together, so compression would read the list from left to right, and the decompressor will read the list from right to left. This works, as the modules all alter the Pnm_ppm in place. The functions can run sequentially without knowing anything but the format of the PPM they work on.
- The reason that DCT and averaging P_B and P_R are in the same step is because they both "shrink" the image size, meaning they have to be done at the same time, otherwise the size of 2D arrays to represent the data won't match (i.e. if we separate them, we would have to maintain different sized arrays). In other words, a secret that the size has changed is shared between them.
- We will have a file reader/writer for uncompressed images (via pnm.h), but we will create a compressed image reader and writer. The flag will determine which is run.

Implementation Plan

1. Trim
2. RGB int to float
3. RGB float to int
4. RGB to CIE
5. CIE to RGB
6. Shrinker
7. Expander
8. Quantizer
9. Dequantizer
10. Packer
11. Unpacker
12. Writer for codewords
13. Reader for codewords

Testing Plan

When testing, we assume that the previous modules (which have been rigorously tested) are working. This is because we will run the previous modules to get to the module we are testing, then we will decompress from that module. (We do "smaller loops" of the compression process to make sure it is working at every step). Note that the compression and decompression are tested at the same time, as otherwise we wouldn't be able to test/look at it, because it would be in an unrecognizable format.

General Module Plan

We will have a bash script that will go through a list of ppm images (specified in the bash script), which will use ppmdiff to find the difference between our compression (which we run incrementally with the added modules) and the jpeg compression, then output both the ppmdiff of the jpeg and the ppmdiff our compression and decompression. The images will mostly be sourced from the /comp/40/bin/images, but we create some of our own (using software). Will run after every module's custom tests are done. We will take a closer look at the software when the difference from jpeg's compression exceeds 0.05.

Custom Test Plan

1. Trim (Note: n and m are "large" numbers, greater than or equal to 250)
    a. 0 x 0 ppm -> empty image (0 x 0 ppm)
    b. 1 x 1 ppm -> empty image (0 x 0 ppm)
    c. 1 x n ppm where n is even -> empty image (0 x 0 ppm)
    d. n x 1 ppm where n is even -> empty image (0 x 0 ppm)
    e. 2 x 2 ppm -> same 2 x 2
    f. n x m ppm where n is even and m is odd -> image crop of n x (m - 1)
    g. n x m ppm where n is odd and m is even -> image crop of (n - 1) x m
    h. n x m ppm where n is even and m is even -> n x m image
    i. n x m where n is odd and m is odd -> image crop of (n - 1) x (m - 1)
2. RGB int to float / RGB float to int
    a. Red square -> red square
    b. Blue square -> blue square
    c. Green square -> green square
    d. Black square -> black square
    e. White square -> white square
    f. Color bar test -> color bar test (no big artifacts)
    g. Gradient test -> gradient (no big artifacts)
    h. High contrast checkerboard (no adjacent pixels are the same color) -> same
    i. Check float range (int -> float only) -> all should be in [0, 1]
    j. Check int range (float -> int only) -> all should be in [0, denominator]
3. RGB to CIE / CIE to RGB
    a. Red square -> red square
    b. Blue square -> blue square
    c. Green square -> green square
    d. Black square -> black square
    e. White square -> white square
    f. Color bar test -> color bar test (no big artifacts)
    g. Gradient test -> gradient (no big artifacts)
    h. High contrast checkerboard (no adjacent pixels are the same color) -> same
    i. Check P_B and P_R value range (RGB -> CIE only) -> all should be in [-0.5, 0.5]
    j. Check Y value range (RGB -> CIE only) -> all should be in [0, 1]
    k. Check RGB value range (CIE -> RGB) -> all should be [0, 1]

4. Shrinker / Expander
    a. Red square -> red square
    b. Blue square -> blue square
    c. Green square -> green square
    d. Black square -> black square
    e. White square -> white square
    f. Color bar test -> color bar test with artifacting
    g. Gradient test -> blocky gradient
    h. High contrast checkerboard (no adjacent pixels are the same color) -> uniform
    i. Black circle on white background -> edges of circle less nice
    j. Monochrome gradient -> preserves luminosity for the most part
    k. Monochrome dither -> preserves the detailed dither pattern
    l. Monochrome large blocks -> Keeps the image the same
    m. 0 x 0 ppm -> empty image (0 x 0 ppm)
    n. 1 x 1 ppm -> empty image (0 x 0 ppm)
    o. 1 x n ppm where n is even -> empty image (0 x 0 ppm)
    p. n x 1 ppm where n is even -> empty image (0 x 0 ppm)
    q. 2 x 2 ppm -> same 2 x 2
    r. n x m ppm where n is even and m is odd -> image crop of n x (m - 1)
    s. n x m ppm where n is odd and m is even -> image crop of (n - 1) x m
    t. n x m ppm where n is even and m is even -> n x m image
    u. n x m where n is odd and m is odd -> image crop of (n - 1) x (m - 1)
    v. Check value range of (a) (Shrinker only) -> all in range [0, 1]
    w. Check value range of (b, c, d) (Shrinker only) -> all in range [-0.5, 0.5]
    x. Check image size (Shrinker only) -> ¼ of the original
    y. Check value range of P_B and P_R -> all in range [-0.5, 0.5]
    z. Check value range of Y (Expander only) -> all in range [0, 1]
    aa. Check image size (Expander only) -> 4 times the the original
5. Quantizer / Dequantizer
    a. Check the P_B and P_R quantization (given) -> both fit into the proper byte size (4)
    b. Check that (a) quantization -> fit into the proper byte size (9)
    c. Check that (a) range -> all in [0, 511]
    d. Check that (b, c, d) quantization -> fit into the proper byte size (5)
    e. Check that (b, c, d) signed range -> all in [-15, 15]
    f. Check that quantization size -> fit into proper byte size (32)
    g. Check the P_B and P_R dequantization (given) -> all in range [-0.35, 0.35]
    h. Check the Y dequantization -> all in range [0, 1]
    i. Red square -> red square
    j. Blue square -> blue square
    k. Green square -> green square
    l. Black square -> black square
    m. White square -> white square
    n. Color bar test -> color bar test with artifacting
    o. Gradient test -> blocky gradient
    p. High saturation image -> worse quality
    q. Colored photo -> high quality
    r. High contrast checkerboard (no adjacent pixels are the same color) -> uniform
    s. Black circle on white background -> edges of circle less nice
    t. Monochrome gradient -> preserves luminosity for the most part
    u. Monochrome dither -> preserves the detailed dither pattern
    v. Monochrome large blocks -> Keeps the image the same

6. Packer / Unpacker
    a. Check that (a) size -> fit into the proper byte size (9)
    b. Check that (b, c, d) size-> fit into the proper byte size (5)
    c. Check order -> in big-endian order
    d. Check pack size -> fit into proper byte size (32)
7. Codewriter / Codereader
    a. Pipe images in, then pipe them to the reader, and check images