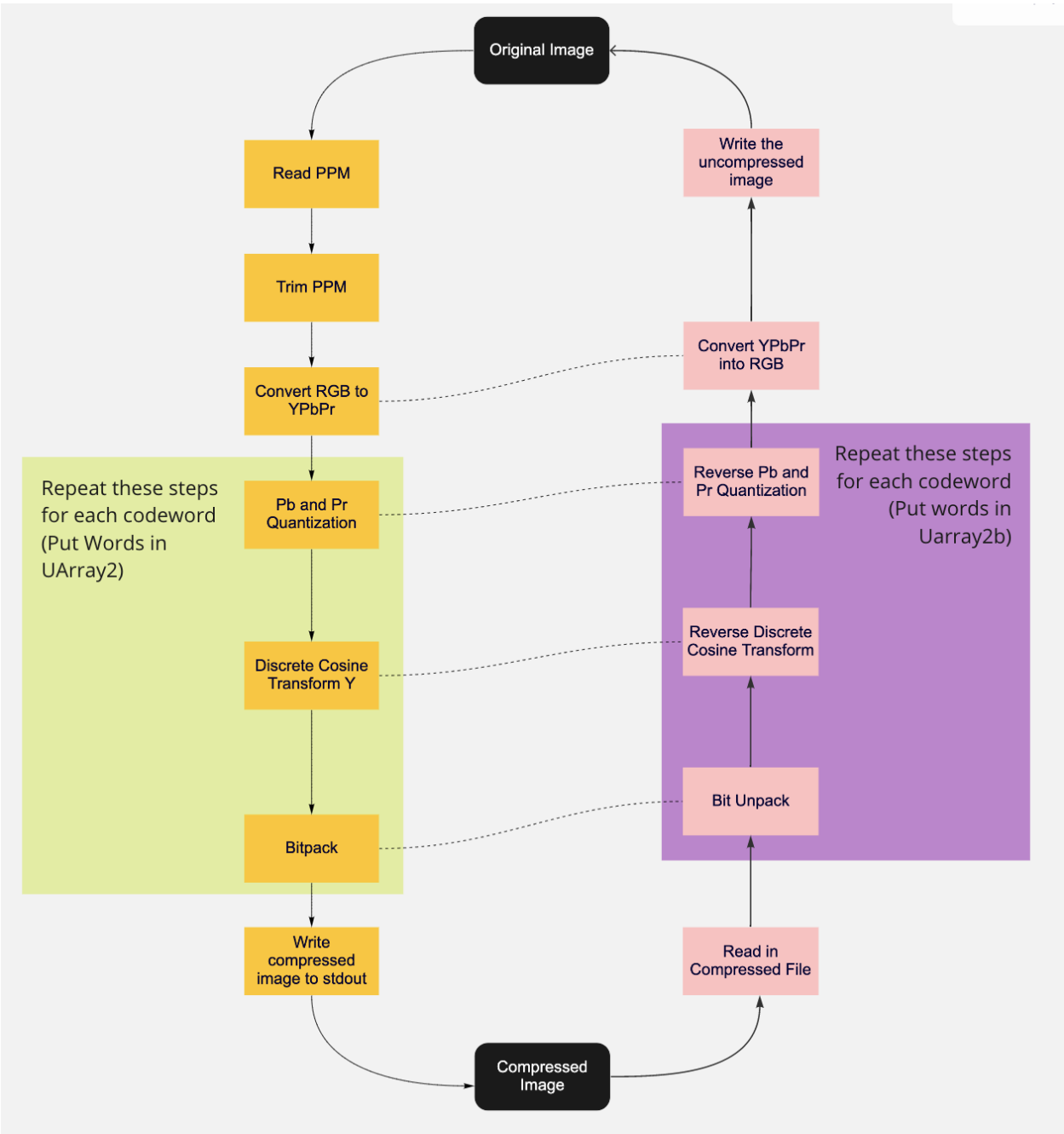
Overall Design



Note: On the diagram above, dotted lines indicate steps that will be part of a shared module.

Compression

1. Read in PPM

a. Input: PPM file
b. Output: Pnm_ppm

In this function we will take in a ppm file from either the command line or standard input. We will then use the pnm.h read function to read the ppm data into an actual Pnm_ppm structure, then return that Pnm_ppm.

2. Trim odd width/height to be even (data lost)
    a. Input: Pnm_ppm
    b. Output: Pnm_ppm ppm with trimmed width/height pixels array

    In this function we will take the Pnm_ppm returned in the previous function and subtract 1 from its width and/or height if they are odd numbers. We will then return the Pnm_ppm with the altered values. If the height and/or width of the inputted ppm is odd, this step will cause data to be lost because the pixels on the corresponding edges will be deleted.

3. Convert RGB to YPbPr (data lost to float division)
    a. Input: ppm pixels array
    b. Output: UArray2b of struct YPbPr

    In this function we will convert the RGB data stored in the pixels array of our Pnm_ppm into component video color space, YPbPr. We will return a 2D blocked array (blocksize = 2) of the YPbPr values of each pixel in the ppm. Data is lost in this step due to the limited precision of floats.

4. Put Words in UArray2 **(Wrapper function of steps 5 - 8)**
    a. Input: Our UArray2b
    b. Output: A UArray2 of codewords

    This function performs a block-major traversal over our UArray2b and runs the functions constituted by steps 5 - 8 on each 2 by 2 block. After all of these functions are run on a block, the result is a codeword, which will be stored at the appropriate index of the UArray2 based on the index of the current block.

Allocate a struct codeword:

Unsigned a
Signed b
Signed c
Signed d
Unsigned Pb
Unsigned Pr

5. Pb and Pr Quantization

a. Input: the current block of YPbPr structs (stored as a UArray), as well as the current codeword struct
b. Output: the quantized unsigned 4 bit representations of Pb and Pr are initialized in the struct
c. Helper functions:
    i. Return the average of either Pb or Pr value (forced to be in range -0.5 to 0.5)
        1. Expected Input: four floats representing either the Pb or the Pr values of the four pixels in a block
        2. Output: average of the four floats (either the average Pb or the average Pr) with the return value constrained to the range -0.5 and 0.5

    This helper function will simply find the average of four float values, which is then constrained to be in the range -0.5 to 0.5. We will pass it either a Pb or a Pr value and will get the average Pb or Pr value back. Data is lost in this step due to the limited precision of floats.

    ii. Arith40_index_of_chroma(float x) (data is lost)
        1. Input: Pb or Pr averages with values between -0.5 and 0.5
        2. Output: Index of quantized Pb or Pr average in an internal table

    We will use the Arith40_index_of_chroma function to quantize the Pb and Pr values returned from the previous function. Data is lost in this step because the quantization of these values will round them into the nearest region in the quantized representation, of which there are 16.

Struct now has Pb and Pr initialized

6. Discrete cosine transform (Y)
    a. Input: the current block of YPbPr structs (stored as a UArray), as well as the current codeword struct
    b. Final Output – the following values are set in the codeword struct:
        i. a (unsigned integer)
        ii. b, c, and d (signed integers)
    c. Helper functions:
        i. Convert 4 Y values into abcd (data lost to quantization) using conversion math
            1. Input: 2 by 2 block of Y values
            2. Output: a, b, c, d
        ii. Force bcd into 5-bit signed values (data lost to quantization)
            1. Input: bcd
            2. Output: 5-bit signed bcd (constrained to -0.3 to 3.0)

This function converts the 4 Y values in a 2 by 2 block into a, b, c, and d using the provided math. With a being an unsigned scaled integer and b, c, and d all being signed scaled integers. Data is lost in this step when the Y values are converted using DCT because of the imprecise nature of float division. Then data is lost again when the values are scaled into their respective bit sizes, as this quantization requires rounding, which loses the precision of the longer value being rounded.

7. Bitpacking
    a. Pack a,b,c,d, Pb, and Pr into a 64-bit word (called on each 2-by-2 block)
        i. Input: a, b, c, d, Pb and Pr
        ii. Output: A 64-bit codeword (uint64_t), with 32 bits used on the codeword and the other 32 set to 0

    In this function we will pack the four encoded values (a, b, c, and d), as well as the average Pb and Pr values into a 64-bit codeword. We will use the Bitpack class to do the bitpacking, following a consistent convention about which values will be encoded in which part of the word (this convention is outlined in the table on Page 5 of the specification). Although the codewords are technically 32 bits long, we will fit them in 64 bit words with half of the bits unused to keep with convention.

8. Write compressed image to stdout
    a. Input: UArray2 of codewords
    b. Output: Compressed image (header and binary words printed to stdout)

    In this function we will write the codewords to stdout by traversing the UArray2 of codewords in row-major order, converting each codeword into four chars, and then printing each char in big-endian order. To do this we will use the uint64_t Bitpack_getu function to get the unsigned values from the codeword, and the int64_t Bitpack_gets function to get the signed values from the codeword. We will then print them individually to ensure they are in big-endian order.

Decompression

1. Read in compressed image
    a. Input: File pointer to the compressed image, a Pnm_ppm in which to store the width, height, and denominator
    b. Output: a UArray2 of codewords

    This function takes a pointer to a file with a compressed image and uses fscanf to read the width and height and set them within the Pnm_ppm object that it is passed. It also reads in the bytes from the compressed image using getchar(), then puts them in a codeword using the Bitpack_newu and Bitpack_news functions based on which coded value we are packing, finally storing each codeword in a UArray2 of codeword structs

that it then returns. If the compressed file is too short (i.e. there aren't enough bytes to read in), the function will throw a Checked Runtime Error.

2. Put Words in UArray2b **(Wrapper function of steps 3–5)**
    a. Input: A UArray2 of codeword structs
    b. Output: A UArray2b of struct YPbPr

    This function contains a for loop that iterates the UArray2 and runs the functions constituted by steps 3–5 on each codeword. After all of these functions are run on a codeword, the result is a 2 by 2 block (UArray of size 4), which will be stored at the i-th location of a UArray2b, based on the value of i in the for loop.

3. Bit unpacking
    a. Input: A 64-bit codeword (uint64_t), with 32 bits used on the codeword and the other 32 set to 0
    b. Output: a, b, c, d, Pb and Pr

    This function will unpack the current codeword using the get functions in the Bitpack class, storing the output values in the codeword struct.

4. Reverse DCT - convert abcd to 4 Y values (data lost to quantization)
    a. DCT space to pixel space
        i. Input: a, b, c, d as ints
        ii. Output: Sets the Y value of each of the 4 pixels in the block that is being iterated using their respective YPbPr struct.

    This function will take the a, b, c, and d values and convert them to Y values using the given math. The Y fields in each of the YPbPr structs of each of the 4 pixels will then be filled with their respective Y values.

5. Reverse Quantization of Pb and Pr
    a. Input: Indices of Pb and Pr in the internal quantized chroma table
    b. Output: Pb and Pr

    This function will convert the values of Pb and Pr from their quantized form to the nearest float values, which we will then use as the respective Pb and Pr values for each of the 4 pixels in the block.

6. Turn YPbPr into RGB
    a. Input: UArray2b of struct YPbPr (passed from the function in Step 2)
    b. Output: A UArray2 with the RGB values of each pixel

This function will create a UArray2 of Pnm_rgb structs that will be used as the pixels value of the final ppm. Each value will be quantized to a value between 0 and a denominator of our choice using range-remapping math.

7. Write the uncompressed image to stdout using Pnm_ppmwrite
    a. Input: UArray2 of Pnm_rgb structs, Pnm_ppm with proper header
    b. Output: Writes the complete Pnm_ppm to stdout

    This function updates Pnm_ppm->pixels to the output UArray from Step 6, then calls the Pnm_ppmwrite from pnm.h to write the uncompressed image to stdout.

Implementation Plan

1. Read PPM
    a. Test: Find two ppm files of different sizes, test_small.ppm and test_large.ppm. Print header info of input ppm to stdout.
        i. Input: test_small.ppm
        ii. Expected output: the ppm file's header information
        iii. Compare output with the expected header information
        iv. Use valgrind to ensure no memory leaks
        v. Repeat previous steps with test_large.ppm

2. Write Uncompressed Image
    a. Test: Take two different ppms that have the same header and swap their pixel arrays and write them to stdout, running the function twice to test if both have been properly swapped.
        i. Input: Two ppm files with the same header
        ii. Expected output: the opposite ppm image
        iii. Verify by comparing images
        iv. Use valgrind to ensure no memory leaks

3. Trim PPM
    a. Test: We will test this function with all 4 possible parities of width and height, as well as with a 1 by 1 ppm. Print the new width and height.
        i. Input Cases:
            1. a ppm with odd width and even height
            2. a ppm with odd width and odd height
            3. a ppm with even width and even height
            4. a ppm with even width and odd height
            5. a ppm with width/height 1
        ii. Expected outputs: Properly trimmed, even width and height values. In the case that width/height is 1, we will throw an exception.
        iii. Use valgrind to ensure no memory leaks

4. Convert RGB to YPbPr
    a. Test: Take a ppm and run it through the function, checking that its conversions from RGB to YPbPr are correct, and the resulting UArray2b is properly filled in order with the YPbPr structs by using the block-major mapping function to print the YPbPr pixels stored in each block.
        i. Input: test_small.ppm
        ii. Expected output: converted YPbPr values, printed to stdout in block-major order
        iii. Use valgrind to ensure no memory leaks
        iv. Repeat previous steps with test_large.ppm

5. Convert YPbPr to RGB
    a. Test 1: Reverse of the previous test: run the UArray2b of YPbPr structs from Step 4 through the function and make sure that it properly outputs a UArray2 of RGB structs, using a row-major mapping function to print the RGB pixels. This checks that all the values are in the correct order and the YPbPr to RGB math is executed properly.
        i. Input: test_small.ppm
        ii. Expected output: converted RGB values, printed to stdout in row-major order
        iii. Use valgrind to ensure no memory leaks
        iv. Repeat previous steps with test_large.ppm
    b. Test 2: Replace the pixels array of the input ppm (which has a width and height that were updated by Trim PPM) with this array of converted RGB pixels and use our Write Uncompressed Image function to output the ppm. Then use ppm_diff to test accuracy and fidelity to the original image.
        i. Input: test_small.ppm
        ii. Expected output: a ppm that appears identical
        iii. Verify the image is (almost) identical using ppmdiff
        iv. Use valgrind to ensure no memory leaks
        v. Repeat previous steps with test_large.ppm

6. Pb and Pr Quantization
    a. Test: Pass a UArray of 4 YPbPr structs to the function and check that the averages of the 4 Pb and 4 Pr values are properly calculated and forced to be values between -0.5 and 0.5 by printing this data to stdout.
        i. Input: a UArray of 4 YPbPr structs
        ii. Expected output: Pb and Pr values are properly calculated and forced into values between -0.5 and 0.5 printed to stdout
        iii. Use valgrind to ensure no memory leaks
7. Reverse Pb and Pr Quantization
    a. Test 1: Take in the index of the respective Pb and Pr values and run them through the chroma_of_index function. Then print values to stdout to check how they compare to the average Pb and Pr values from the beginning of step 6.

        i. Input: Indices of Pb and Pr values in the internal table
        ii. Expected output: Pb and Pr values (floats between -0.5 and 0.5) printed to stdout
        iii. Use valgrind to ensure no memory leaks
    b. Test 2: Run steps 1, 3, 4, and 6 (i.e. the compression steps) on a test ppm file, then run this step followed by steps 5 and 2 to reverse the compression steps.
        i. Input: test_small.ppm
        ii. Expected output: a ppm that appears identical
        iii. Verify the image is (almost) identical using ppmdiff
        iv. Use valgrind to ensure no memory leaks
        v. Repeat previous steps with test_large.ppm
8. Discrete Cosine Transform Y
    a. Test: Take in a UArray of 4 YPbPr structs and an empty codeword struct. We will then check if our function properly converts the 4 Y values to a, b, c, and d using the given math by printing those values to stdout and using the given reverse math. If our function works properly we'll get 4 Y values that are close to the 4 originally inputted to the function, they will be slightly different due to lost data. We will then check if our b, c, and d values get properly constrained to 5 bits, and that a, b, c, and d are all properly scaled.
        i. Input: UArray of 4 YPbPr structs and empty codeword struct
        ii. Expected output: calculated values of a, b, c, and d printed to stdout
9. Reverse Discrete Cosine Transform
    a. Test 1: Convert ints a, b, c, and d to their respective Y values and print the converted Y values, comparing them to the Y values from the beginning of step 8
        i. Input: UArray of 4 YPbPr structs and ints a, b, c, and d.
        ii. Expected output: a ppm that appears identical
        iii. Verify the image is (almost) identical using ppmdiff
        iv. Use valgrind to ensure no memory leaks
        v. Repeat previous steps with test_large.ppm
    b. Test 2: Run steps 1, 3, 4, and 8 (i.e. the compression steps) on a test ppm file, then run this step followed by steps 7, 5, and 2 to reverse the compression steps.
        i. Input: test_small.ppm
        ii. Expected output: a ppm that appears identical
        iii. Verify the image is (almost) identical using ppmdiff
        iv. Use valgrind to ensure no memory leaks
        v. Repeat previous steps with test_large.ppm
10. Bitpacking and Bit Unpacking
    a. Test: We will take in a fully initialized codeword struct with reasonable values for a, b, c, d, Pb and Pr. We will then pack each value into one 64 bit codeword using the Bitpack_newu and Bitpack_news functions repeatedly on each value (depending on if they are signed or unsigned) and the same codeword. After this, we will use the Bitpack_getu and Bitpack_gets functions to print each value from the codeword and check if they match the original input from the codeword struct.
        i. Input: fully initialized codeword struct

    ii. Expected output: correct a, b, c, d, Pb and Pr values printed to stdout
    iii. Use valgrind to ensure no memory leaks
11. Write Compressed Image to stdout
    a. Test: We will take in a UArray2 of codewords and print them to stdout in row major order. We will test to make sure that each codeword is properly printed in big endian order, as well as if the UArray2 itself is being properly mapped over and printed to stdout.
        i. Input: A UArray2
        ii. Expected Output: A compressed image printed to stdout
        iii. Use valgrind to ensure no memory leaks
12. Read in Compressed File
    a. Test Case 1:
        i. Input: File with valid header
        ii. Expected Output: Reads in compressed image and successfully prints out the decompressed ppm
        iii. Use valgrind to ensure no memory leaks
    b. Test Case 2:
        i. File with invalid header
        ii. Expected Output: throws a checked runtime error
    c. Test Case 3:
        i. File with not enough bytes
        ii. Expected Output: throws a checked runtime error