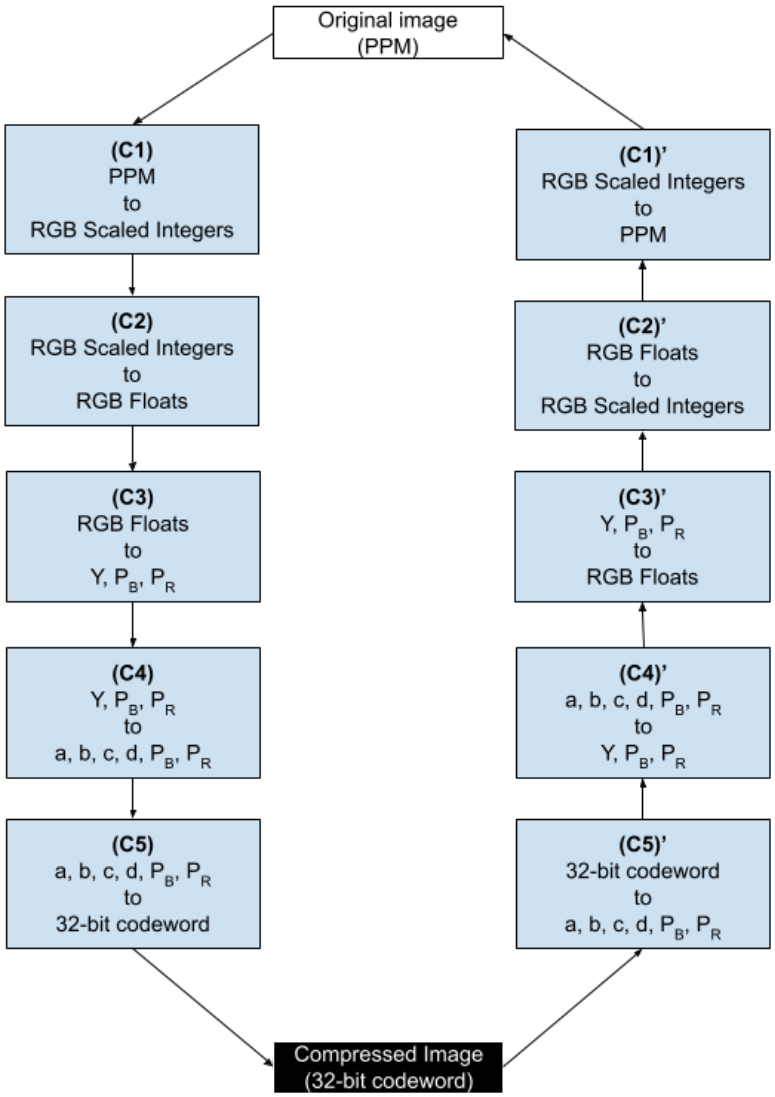

General Testing Plan (applies to all steps)

- For each component, we will verify successful compilation with Makefile very often.
- We will write unit tests to test for invariants and that every function works as intended.
- We will use print statements between lines in the function to locate sources of error.
- We will run the program with valgrind after each step to ensure find and remove memory leaks.
- Round Trip Testing
    - We will implement the steps in pairs (the compression step, C, and its corresponding decompression inverse, C').
        - After each implementation pair (C, C'), we will run the program on an image up to the implementation step in progress, compressing and decompressing.

        - We will then use ppmdiff to compare the original image with the result from the program.
        - We will use the resulting E value to verify the success of the steps.
            - E is “small enough” when it is roughly < 0.03, but will vary depending on potential information loss at certain steps.
- General Test cases for every step
    - Empty image
    - Large image
    - Very small (2 x 2) images
    - Small + medium images
    - Originally even dimensions
    - Originally odd dimensions
    - Originally mixed-parity dimensions
    - Pure black + pure white
    - Pure red/green/blue
    - Real images from internet

C1 & C2: Conversion from PPM to RGB Scaled Integers then to RGB Floats

- *Description*:
    - Uses Pnm_ppmread to a Pnm_ppm image.
    - Trims last row and/or column to make width and height of image even.
    - Traverses image, reading each pixel's scaled unsigned integer red, green, and blue values into a UArray2b of block size 2 of structs holding the RGB float values (and other necessary information).
        - (This is the C2 part) Divide each scaled integer by the denominator (maxval)
- *Intended input and output*:
    - Overall Input and Output:
        - Input: File pointer to PPM file
        - Output: UArray2b of structs containing RGB float values
    - Intermediate Input and Output:
        1. Read in file using Pnm_ppm reader
            a. Input: File pointer to the PPM file
            b. Output: Pnm_ppm image
        2. Trim file (If necessary)
            a. Input: Pnm_ppm image
            b. Output: Pnm_ppm image (with even width and height)
        3. Read RGB values into structs
            a. Input: Pnm_ppm image
            b. Output: UArray2b of structs containing RGB float values

- *Information Loss?*:
    - At most, one row and/or column of the original image may be irreversibly lost due to trimming to an even width and column.
    - Dividing to convert scaled integers to floats may lose negligible amounts of information due to rounding.
- *Testing Plan for Specific Cases*:
    - Test with a PPM image with even width and height.
    - Test with a PPM image with odd width and/or height.
        - Verify the width and height are even and one less after trimming.
    - Test with empty PPM image edge case.
        - We expect the program to run without exiting.

C3: Conversion from RGB Floats to Component Video Color Space

- *Description*:
    - Makes sure struct for each pixel holds floats that represent the Y, P_B, P_R
    - Calculates Y, P_B, P_R for each pixel using given formulas
    - Uses the four P_B and P_R (DC component) values of each block to find block average $\overline{P_B}, \overline{P_R}$. Stores those in the block's struct.
    - Quantizes $\overline{P_B}, \overline{P_R}$ (using given Arith40_index-of_chroma function) to 4-bit values and store in the block's struct
- *Intended input and output*:
    - Overall Input and Output:
        - Input: UArray2b of structs containing RGB float values
        - Output: 4-bit quantized representations of $\overline{P_B}, \overline{P_R}$ for each block
    - Intermediate Input and Output:
        1. Using given formulas to calculate Y, P_B, P_R for each pixel
            a. Input: RGB float values
            b. Output: Y, P_B, P_R float values
        2. Finding block averages $\overline{P_B}, \overline{P_R}$
            a. Input: Four P_B, P_R float values for pixel in each block
            b. Output: $\overline{P_B}, \overline{P_R}$ float values for each block
        3. Quantization to 4-bit representations
            a. Input: $\overline{P_B}, \overline{P_R}$ float values for each block
            b. Output: $\overline{P_B}, \overline{P_R}$ unsigned 4-bit values for each block
- *Information Loss?*:
    - There is potential information loss during floating point arithmetic when calculating the luminance and chroma component video values. Because floating

point values of r, g, and b could have different decimals places after being scaled from the image, the resulting Y, P_B, P_R may have lost information in decimal truncation since floating point arithmetic addition is not associative.

    - Quantizing $\overline{P_B}, \overline{P_R}$ from floats to 4-bit values essentially "rounds" a 32-bit representation to a 4-bit representation. Since many 32-bit representations map to the same 4-bit representation, we irreversibly sacrifice precision for space efficiency.
    - Negligible amounts of information may be lost due to rounding when finding averages.
- *Testing Plan for Specific Cases*:
    - Y, P_B, P_R
        - Test with small normal image example and verify the Y, P_B, P_R for each pixel
        - Test with fully black image and verify that the Y, P_B, P_R are all 0.0
        - Test with fully white image and verify Y is 1.0 and P_B, P_R are 0.0
        - Test with fully red/green/blue and verify Y, P_B, P_R match with expected outputs
    - $\overline{P_B}, \overline{P_R}$
        - Set the P_B, P_R for each pixel in the block and verify the average function correctly calculates the average
    - Quantization
        - Test that the quantization function correctly outputs a 4-bit unsigned integer.
    - We will use unit tests for every function using a hard-coded example in order to verify its correct implementation.

C4: Conversion from Component Video Color Space to Cosine Coefficients

- *Description*:
    - Transforms the Y values (luminances) of each 2x2-pixel block from pixel space to DCT space using the DCT formulas.
        - This represents brightness data in a more compression-friendly way using cosine coefficients *a*, *b*, *c*, and *d*.
- *Intended input and output*:
    - Overall Input and Output:
        - Input: The four floating-point Y (luminance) values for each 2x2 block.
        - Output: Updated UArray2b structs for each block, now containing the cosine coefficients *a*, *b*, *c*, and *d*.
- *Information Loss?*:

    - There is potential information loss during floating point arithmetic when calculating the cosine coefficients. Because the floating point values of r, g, and b could have different decimals places after being scaled from the image, the resulting luminances could also have different decimals places. Since floating point arithmetic addition is not associative, adding a Y with more decimal places to a Y with less may lose the information of the difference in decimal places.
- *Testing Plan for Specific Cases*:
    - Test with fully black image and verify the coefficients are all 0
    - Test with fully white image and verify that *a* is 1, and *b*, *c*, and *d* are 0
    - Test with fully red image and verify the coefficients match expected values

C5: Conversion from Cosine Coefficients to 32-bit word

- *Description*:
    - Converts *b*, *c*, and *d* for each block into 5-bit signed values using our bitpack.c implementation.
        - If necessary, force their values in the range [-0.3, 0.3] to be able to pack into 5 bits.
        - Quantize the b, c, and d coefficients by mapping the float interval [-0.3, 0.3] to the 5-bit signed integer range [-15, 15].
    - Packs in the following into a 32-bit word for each block:
        - 9-bit representation of *a* (found by multiplying *a* by 511 and rounding)
        - 5-bit representations of *b*, *c*, and *d*
        - 4-bit representations of index($\overline{P_B}$) and index($\overline{P_R}$)
            - Index operation implemented by Arith40_index_of_chroma
        - Logic: (assuming Bitpack.c fully works)
            - Declare uint64_t (not necessarily important if 32-bit codeword is unsigned or signed)
            - Place the necessary representation values in 32-bit codeword using Bitpack interface
    - Output the compressed binary image to stdout with correct header and format
        - Header followed by 32-bit codewords in row major in big-endian
        - Width and height are after trimming for potential odd columns/rows
- *Intended input and output*:
    - Overall Input and Output:
        - Input: UArray2b of structs now with the luminances for each block
        - Output: stdout of the formatted compressed binary image
    - Intermediate Input and Output:
        1. b, c, d conversion from floats to 5-bit signed values
            a. Input: Float representation of each variable
            b. Output: 5-bit representation of each variable

2. Packing each block into 32-bit codeword
    a. Input: All needed bit representations for 32 code word specified by table
    b. Output: Fully initialized block structs in UArray2b
3. Output compressed binary image to stdout
    a. Input: UArray2b with block structs containing 32-bit code words
    b. Output: stdout of formatted compressed binary image

- *Information Loss?*:
    - Irreversible data loss occurs when we round *a* to turn it into 9-bit representation
    - Irreversible data loss occurs if we force *b*, *c*, and *d* (which initially may be in the range [-0.5, 0.5]) into the range [-0.3, 0.3].
    - We quantize the coefficients when rounding to the nearest integer after multiplication by 50. Since many 32-bit representations map to the same 5-bit representation, we irreversibly sacrifice precision for space efficiency.
- *Testing Plan for Specific Cases*:
    - Test with white image
        - Make sure b, c, d are still 0 after calculation
    - Test with our own created image files
    - Test with the CS 40 image files
    - Test with empty image
    - Test with huge image

(C5)': Conversion from 32-bit word to Cosine Coefficients

- *Description*:
    - Reads header of compressed file.
        - Store width, height
    - Declare UArray2b to hold pixels with given width and height. Place this in a pixmap struct, along with width, height, denominator, and the methods.
    - Reads in the 32-bit codewords with the most significant byte first.
        - Unpack the coefficients *a*, *b*, *c*, and *d*, as well as the index($\overline{P_B}$) and index($\overline{P_R}$) and store in each 2x2 block struct
- *Intended input and output*:
    - Overall Input and Output:
        - Input: Compressed image file
        - Output: *a*, *b*, *c*, *d*, *index*($\overline{P_B}$), *index*($\overline{P_R}$) stored in 2x2 block structs
- *Information Loss?*:
    - No information loss expected given correct Bitpack implementation and usage. However, new values will reflect the information lost in C5
- *Testing Plan for Specific Cases*:

- Test we are reading the compressed header correctly.
  - Verify width and height variables are correctly set and even with very small and large dimensions.
- Test that we are reading the 32-bit code words in big-endian order.
- Manually create known code word and verify correct output
  - Code word with all 0s, verify all values are 0
- Test alongside corresponding compression step C5.
  - We expect no difference, as bit-packing and un-packing is lossless.

(C4)': Conversion from Cosine Coefficients to Component Video Color Space

- *Description*:
  - Converts the coded *index*($\overline{P_B}$), *index*($\overline{P_R}$) to $\overline{P_B}$, $\overline{P_R}$ using Arith40_chroma_of_index(unsigned n) and store within each pixel block's struct
  - Uses the inverse of the discrete cosine transform (DCT) to compute the luminance values of each pixel in the block and store within each pixel's struct.
- *Intended input and output*:
  - Overall Input and Output:
    - Input: *a*, *b*, *c*, *d*, *index*($\overline{P_B}$), *index*($\overline{P_R}$) of the image stored in 2x2 block structs
    - Output: Each pixel's luminance stored in its pixel struct and the $\overline{P_B}$, $\overline{P_R}$ stored in each block struct
  - Intermediate Input and Output:
    1. Decoding *index*($\overline{P_B}$), *index*($\overline{P_R}$)
       a. Input: coded *index*($\overline{P_B}$), *index*($\overline{P_R}$)
       b. Output: $\overline{P_B}$, $\overline{P_R}$
    2. Inverse DCT
       a. Input: a, b, c, d
       b. Output: Y_1, Y_2, Y_3, Y_4
- *Information Loss?*:
  - No information loss expected given correct Bitpack implementation and usage
- *Testing Plan for Specific Cases*:
  - Chroma Decoding
    - Assert that the de-quantized chroma values are within the valid range of [-0.5, 0.5].
  - Inverse DCT
    - Test with monochrome blocks.
      - We expect the calculated luminances to be identical.

        - Test with all coefficients 0 (black) and verify all Y values are 0
        - Test with a = 1, b = c = d = 0 (white) and verify all Y values are 1
    - Verify all pixels in each block share the same $\overline{P_B}$, $\overline{P_R}$ values

(C3)': Conversion from Component Video Color Space to RGB Floats

- *Description*:
    - Converts the four pixels in the block to the RGB color space using the inverse transformation formulas.
- *Intended input and output*:
    - Overall Input and Output:
        - Input: Each pixel's luminance stored in its pixel struct and the $\overline{P_B}$, $\overline{P_R}$ stored in each block struct
        - Output: Each pixel's RGB floats stored in its pixel struct.
- *Information Loss?*:
    - No information loss expected given correct Bitpack implementation and usage. However, new values will reflect the information lost in C3 when taking the average of the chroma values.
- *Testing Plan for Specific Cases*:
    - Unit test with black image by providing inputs Y = 0.0, $\overline{P_B}$ = 0.0, $\overline{P_R}$ = 0.0. Assert that the resulting RGB float values are all approximately 0.0.
    - Unit test with white image by providing inputs Y = 1.0, $\overline{P_B}$ = 0.0, $\overline{P_R}$ = 0.0. Assert that the resulting RGB float values are all approximately 1.0.

(C2)': Conversion from RGB Floats to RGB Scaled Integers

- *Description*:
    - Quantize RGB values to integers in range 0 to xxx for denominator and put these values in the structs of each pixel in the pixmap→pixels array, the UArray2b
- *Intended input and output*:
    - Overall Input and Output:
        - Input: Each pixel's RGB floats stored in its pixel struct
        - Output: pixmap with scaled integer representation of RGB stored in each pixel struct
- *Information Loss?*:
    - There will be irreversible information lost due to quantizing the RGB floats to integers
        - From the multiple quantizations from previous components, values potentially falling outside of 0 and 1 will be lost

    - There may be irreversible information lost due to rounding when converting the float to an integer, as multiple different float values can round to the same integer.
- *Testing Plan for Specific Cases*:
    - Test that negative floats are quantized to 0.
    - Test that floats greater than 1 are quantized to xxx.

(C1)': Conversion from RGB Scaled Integers to PPM

- *Description*:
    - Using the pixmap struct holding the UArray2b holding the pixel structs with their scaled integer RGB values along with other necessary information (width, height, etc.), format the printed decompressed PPM to stdout
- *Intended input and output*:
    - Overall Input and Output:
        - Input: pixmap with scaled integer representation of RGB stored in each pixel struct
        - Output: Prints the decompressed image to standard output using Pnm_ppmwrite(stdout, pixmap).
- *Information Loss?*:
    - No information loss expected given correct Bitpack implementation and usage.
- *Testing Plan for Specific Cases*:
    - Redirect program output run on given image to output and verify correct header information
        - Width and height are dimensions after potential trimming
        - Denominator is the one we chose and is valid range of 1 to 65535
    - Verify all RGB values are in range 0 to denominator

Module Architecture and File Organization

bitpack.c and bitpack.h (provided)
Contains: Bit manipulation operations
In: C5, (C5)'

rgb_cv_convert.c and rgb_cv_convert.h
Contains: Functions to convert between RGB and component video
In: C3 (conversion), (C3)' (inverse conversion)

dct.c and dct.h
Contains: Functions that facilitate the discrete cosine transform
- converts Y_1, Y_2, Y_3, Y_4 to a, b, c, d
- converts a, b, c, d to Y_1, Y_2, Y_3, Y_4

In: C4 (forward DCT), (C4)' (inverse DCT)

quantize.c and quantize.h
Contains: Functions to quantize and dequantize
- *b*, *c*, *d* to 5-bit signed integers
- 5-bit signed to floats
- *a* to 9-bit unsigned integer
- dequantizes 9-bit unsigned to float

In: C5 (quantization), (C5)' (dequantization)

codeword.c and codeword.h
Contains: Functions packing *a*, *b*, *c*, *d*, $p_b$index, $p_r$index into 32-bit word and unpacking 32-bit word into components
In: C5 (packing), (C5)' (unpacking)

image_format.c and image_format.h
Contains: Functions that trim dimensions, write PPM to stdout, reads and writes file headers, reads and writes 32 bit words in big endian

C1 (read/trim), C2 (scale to floats), C5 (write compressed), (C5)' (read compressed), (C2)' (scale to integers), (C1)' (write PPM)

40image.c (provided)
Contains: main() function for command-line handling
Calls: respective compression and/or decompression functions