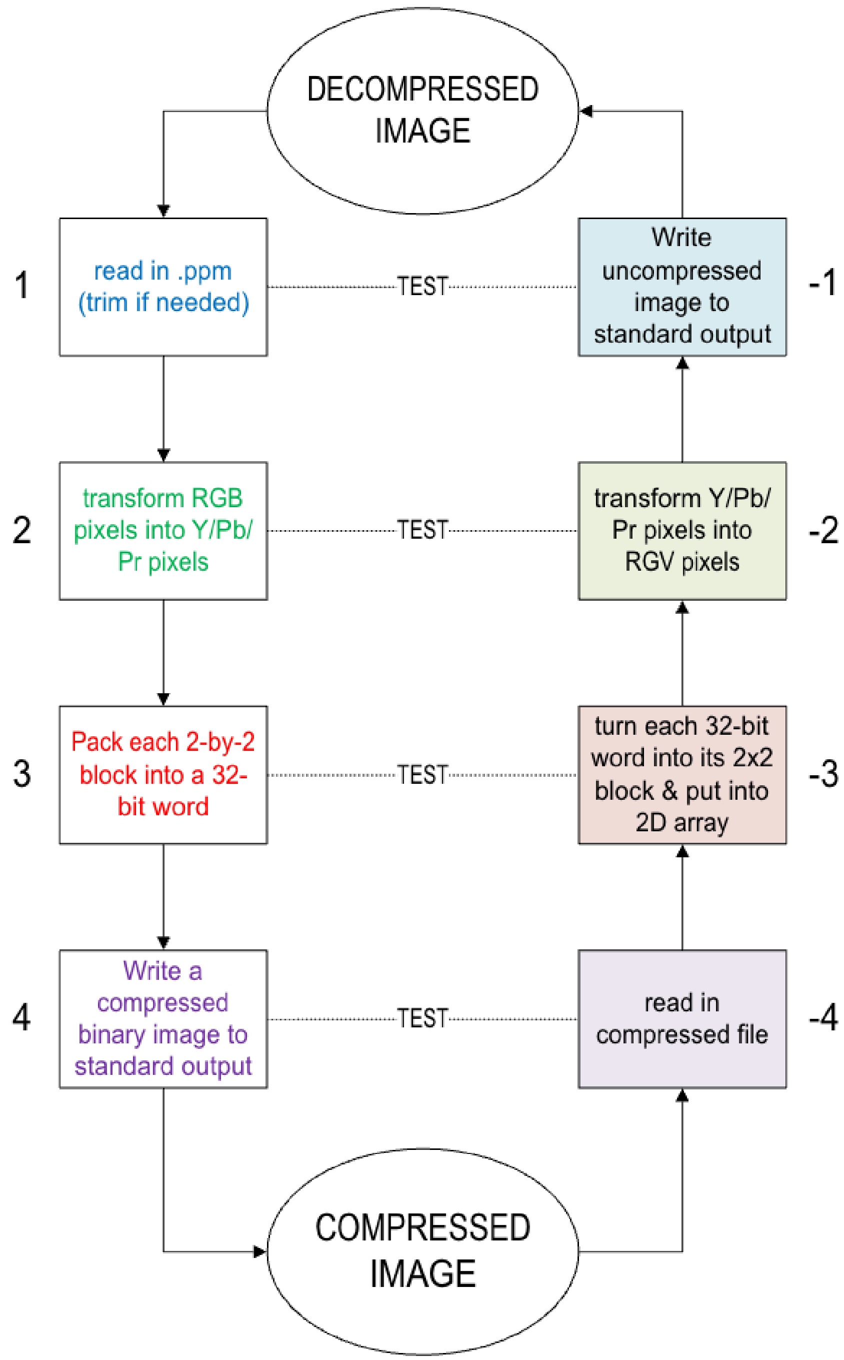
Design doc

Marielle Cibella (mcibel01) & Erica Huang (ehuang02)



Note: all compression steps [1 to 4] will be in its own file (compression.c) and all decompression steps [-1 to -4] will be in its own file (decompression.c). We will use functions from the bitpack file for components of step 3 and -3. We will also have a main file (main.c) that uses both our compression and decompression files. We will also further modularize our code where we see it is needed.

Compression	Decompression
Step 1: Read PPM and trim **Input/output:** ● Input: .ppm file ● Output: pnm_ppm instance where the width and height are even integers **Is information lost?:** ● yes if we have to trim the edges **Implementation Summary:** ● Read a PPM image from a file specified on the command line or from standard input ○ Note: given main file to handle commands ○ Use a FILE *file variable to open the given file or stdin if no file supplied ○ Use pnm_ppmread ● trim width or height if either is an odd number ○ Check the width and height in header, if width and or height is an odd number create a new 2d array with the trimmed width/ height, copy the necessary pixels over, create a new pnm_ppm instance from this, and free the old pnm_ppm with the untrimmed edges ○ if widht and height are even j use original pnm_ppm	Step -1: Output Uncompressed PPM Image **Input/output:** ● Input: a UArray2 of RGB vals (pnm_rgb struct) ● Output: an uncompressed ppm file **Is information lost?:** ● No **Implementation Summary:** ● create Pnm_ppm from the destination 2d arry (bc is type A2Methods_UArray2) ● Use Pnm_ppmwrite(stdout, pixmap) to output the image as a standard PPM.
Step 2: Change to floating point and transform RGB to (Y/Pb/Pr) Input/output: ● Input: pnm_ppm containing pnm_rgb pixels, where each pixel has red,	Step -2: Y/Pb/Pr to RGB Conversion **Input/output:** ● Input: a UArry2 of YPbPr vals (ypbpr struct) ● Output: a UArray2 of RGB vals

<table>
<tr>
<td>

green, and blue values as unsigned integers
● Output: A UArray2 containing YPbPr structs for each pixel, where Y, Pb, and Pr are floating-point values

Is information lost?:
○ no

Implementation Summary:
● for each pnm_rgb in pixel pnm_ppm calculate (Y/Pb/Pr)
○ Within a for loop:
■ Get one pixel (pnm_rgb struct) using at()
■ change r, g, & b to floating point
■ Change r, g, & b to Y, Pb, and Pr using calculations which get stored in a struct
■ Store the YPbPr struct in a UArray2
● Ex function:
○ struct_ypbpr newFormat(struct Pnm_rgb) → returns ypbpr struct
■ float r = rgb->red
■ float g = rgb->green
■ float b = rgb->blue
● Calculations
○ y = 0.299 * r + 0.587 * g + 0.114 * b;
○ pb = -0.168736 * r - 0.331264 * g + 0.5 * b;
○ pr = 0.5 * r - 0.418688 * g - 0.081312 * b;

</td>
<td>

(pnm_rgb struct)

Is information lost?:
● No

Implementation Summary:
● In a for loop
○ Read the Y/Pb/Pr values of one instance in the UArray2 from the compressed data that is stored in a struct
○ Inverse transformation from Y/Pb/Pr values back to RGB integers using the inverse transformation formulas
■ R = Y + 1.402 x Pr
■ G = Y − 0.344136 x Pb − 0.714136 x Pr
■ B = Y + 1.772 x Pb
○ Ensure (probably using an assert statement that the calculated R, G, B values are between 0 and 255
○ Store each RGB value in a Pnm_rgb struct and create a 2D array for the uncompressed RGB pixels
■ once we make the uarray2 we can make a pnm_ppm struct bc pixels is A2Methods_UArray2

</td>
</tr>
<tr>
<td>

Step 3: Pack each 2-by-2 block into a 32-bit word

Input/output:
● Input: UArray2 where each element is a yPbPr struct
● Output: UArray2 of 32bit words

</td>
<td>

step -3: unpack 32 bits into a 2d array

Input/output:
● Input: A 2D array (UArray2) storing each 32-bit word as an element. (width and height of this 2d array will b half of the img bc each element is a

</td>
</tr>
</table>

Is information lost?:

- yes when we take the average Pb and Pr for the block we lose the precise values for each pixel
- yes when we turn bcd into 5 bit signed vals because we put their floats into buckets that determine their 5bit val
  - (if a bucket was -.1 to -.2 then -.12 and -.19 would be the same val in a 5bit val)
  - if a float is greater than .3 or less than -.3 then we say it is -.3 or .3 so we lose that info too
- yes when we pack Pb and Pr into 4 bit vals bc putting into buckets
- yes when we multiply a by 511 and round bc can't unround a number

Implementation Summary:

3a - get Pb and Pr and convert into 4bit val

- Use Arith40_index_of_chroma(float x) to quantize the floating-point Pb and Pr values and store the value in an unsigned int variable
  - Output: Although it's stored as 4 bytes, the value is between 0 and 15 which fits in 4 bits
- Ex code:
  - unsigned pb_index = Arith40_index_of_chroma(pb);
  - unsigned pr_index = Arith40_index_of_chroma(pr);

3b: use DCT to turn the 4 Y values into into a, b, c, d & turn bcd into 5 bit signed vals

- Calculate a, b, c, and d as floating-point values using the DCT formulas
  - $a = (Y4 + Y3 + Y2 + Y1)/4.0$
  - $b = (Y4 + Y3 - Y2 - Y1)/4.0$
  - $c = (Y4 - Y3 + Y2 - Y1)/4.0$
  - $d = (Y4 - Y3 - Y2 + Y1)/4.0$
- Trim values to the 5-bit signed range if needed [-15, 15]
  - Note: make a function for this

2x2 block of pixels)

- Output: a UArry2 of YPbPr vals (ypbpr struct). width and height of this 2d array will be the same as the img bc unboxed so each pixel is its own element

Is information lost?:

- No

Implementation Summary:

-3a: allocate 2D (blocked) array of struct_ypbpr to hold decompressed img

- width and height is that of compressed img
- size is size of a colored pixel (pnm_rgb)
- choose a denominator

-3b: read in 32 bit code words (from 2d arr)

- for each word unpack a, b, c, d, Pb, Pr
  - Use LSB chart to know where in the word each value starts and read in each value
    - USE BITPACK
  - create function to turn packed b,c, & d back into their float vals (from 5 bit signed ints)
  - turn a back into its float val (divide by 511 and cast as float)
  - turn Pb and Pr back into floats (from 4bit vals) -> use float Arith40_chroma_of_index(unsigned n);
  - store values in a struct
- get Y1-4 from abcd (as floats)
  - $Y1 = a - b - c + d$
  - $Y2 = a - b + c - d$
  - $Y3 = a + b - c - d$
  - $Y4 = a + b + c + d$
- now have (Yi, Pb, Pr) for each of the 4 blocks
- add each pixel from block to destination 2d array
  - use tiny for loop and at() for each block

<table>
<tr>
<td>

|b|, |c|, and |d| <= 0.3 → 5-bit scaled integers (-15 to 15)
|b|, |c|, and |d| > 0.3 → code it as if it were +0.3 or −0.3, whichever is closer → signed integer +15 or -15

multiply a by 511 and cast as int (so it rounds)
Store or return the values for packing

3c: Pack a, b, c, d. Pb, and Pr, into a 32-bit word

Use BITPACK functions → Bitpacking is its own file that we will call functions from

Create 32 bit word variable
Pack a (9-bits, unsigned)
Pack b, c, d (5-bits each, signed)
Pb and Pr (4 bits each, unsigned)

</td>
<td></td>
</tr>
<tr>
<td>
Step 4: Write a compressed binary image to standard output
Input/output:

Input: A UArray2 containing 32-bit words
Output: A compressed binary image written to stdout that includes a header with the width and height and a sequence of 32-bit words written in big-endian order

Is information lost?:

No because just writing

Implementation Summary:

Write header of compressed with: printf("COMP40 Compressed image format 2\n%u %u", width, height);

width and height AFTER trimming

</td>
<td>
Step -4: read in compressed file
Input/output:

Input: A compressed file containing a header with the image format as well as a sequence of 32-bit words
Output: A 2D array (UArray2) storing each 32-bit word as an element. (width and height of this 2d array will b half of the img bc each element is a 2x2 block of pixels)

Is information lost?:

No

Implementation Summary:

use fscanf("COMP40 Compressed image format 2\n%u %u", width, height)
Begin decompression using steps above

</td>
</tr>
</table>

• header should be followed by a newline and then a sequence of 32-bit code words, one for each 2-by-2 block of pixels
• write each word to stdout (disk) in big-endian order (most significant byte first)
    ○ write single byte using putchar()
    ○ use row-major (will go (0,0) → (2,0) not (1,0) bc each word is a 2x2 block)
	• maybe store each word in an element of a 2d array
	    ○ width and height would be half as big as decompressed img bc each element is the 2x2 block

Order of implementation/testing:
Note: implementation order is in alphabetical order
Note: create executable file used to trim original images and return the trimmed version → can use this for testing in Steps 1, -1, 2, and -2 bc the only info lost in these steps is the trimmed edges. will allow us to diff

A. Read PPM and trim - **(Step 1)**
   a. Testing:
      i. Test with images both even and odd dimensions
         1. Ensure images with even dimensions are read without modification
         2. Verify images with odd dimensions are trimmed correctly
      ii. Ex:
         1. 4x4 → 4x4 (No change)
         2. 5x4 → 4x4 (Trim 1 column)
         3. 4x5 → 4x4 (Trim 1 row)
         4. 5x5 → 4x4 (Trim 1 row and 1 column)
      iii. Edge cases:
         1. Very large dimensions
         2. Very small dimensions
         3. 0 x 0
      iv. Check the output Pnm_ppm instance to verify width and height (use print statements)
         1. Ensure the pixel data matches the trimmed dimensions

B. Output Uncompressed PPM Image - **(Step -1)**
   a. Testing:
      i. Verify that the output matches the trimmed PPM image dimensions and pixel data - **use ppmdiff.c to identify whether or not the margin of**

difference is appropriate

1. due to lossness it won't be OG dimensions (unless width and height are even bc then no trimming)
2. with odd width/ height manually compare (use small file size so can easily see a lack of width or height) (can't do diff bc losing dimensions)

ii. Edge case:

1. Test with an empty array
2. really big array (width and height are even)

C. Change to floating point and transform RGB to (Y/Pb/Pr) - **(Step 2)**

a. Testing:

i. with a small array, get the Y/Pb/Pr of each pixel and compare that with manual calculation

1. Ensure Y is

ii. with a bigger array

D. Y/Pb/Pr to RGB Conversion - **(Step -2)**

a. Testing:

i. when width/ height are even can diff with original fil

ii. if width/ height is odd can trim og img then diff

iii. cases:

1. empty array
2. odd width even height
3. even width odd height
4. odd width odd height
5. even width even height
6. giant arrays
7. small arrays

iv. **Use ppmdiff.c to identify whether or not the margin of difference is appropriate**

E. Pack each 2-by-2 block into a 32-bit word - **(Step 3)**

a. Testing:

i. Verify each value is packed into the correct bit position using the LSB chart

ii. Use Bit masks to manually check each section of the 32-bit word

iii. for each substep, print out results to a file to late use for manual comparison when writing step -3

iv. Edge cases:

1. Maximum or minimum values for a, b, c, d, Pb, Pr
2. Ensure proper rounding and no overflow for numbers that slightly exceed the range of a or b, etc.

F. unpack 32 bits into a 2d array **(step -3)**

a. Note: will need to make a lot of files in step 3 to compare with our output in this step

b. Note: a lot of manual comparison will have to take place due to the information lost during compression

c. Testing:

i. step: allocate 2D (blocked) array of struct_ypbpr to hold decompressed img

1. valgrind to make sure arr was allocated
2. print width, height, and size to confirm correct

ii. step: read in 32 bit code words (from 2d arr)
    1. with small image, print out whole wordand manually compare
    2. with small image, separate a, b, c, d, Pb, Pr and print out separately then manually compare (check if were splitting it correctly)

iii. step: create function to turn packed b,c, & d back into their float vals (from 5 bit signed ints) and turn a back into its float val (divide by 511 and cast as float)
    1. with small image, turn into float, print, and manually compare with b, c, d vals before compression

iv. step: turn Pb and Pr back into floats (from 4bit vals) -> use float Arith40_chroma_of_index(unsigned n);
    1. manually compare

v. step: store unpacked values in a struct
    1. valgrind to make sure structs are allocated correctly
    2. print out values of struct

vi. step: get Y1-4 from abcd (as floats)
    1. compare with Y values before comparison (shoudl be pretty close in val)

vii. step: add each pixel from block to destination 2d array
    1. after adding to 2d array print out contents of arry
    2. valgrind to make sure each element exists after trying to add it
    3. check width and height of 2d array to make sure its correct
    4. try adding just one block then testing to make sure you add it correctly (keeps block shape in array)

viii. **Use ppmdiff.c to identify whether or not the margin of difference is appropriate**

G. Write a compressed binary image to standard output - **(Step 4)**

a. Testing:
    i. Check the header format
        1. Should be in format:
            COMP40 Compressed image format <number>
            <width> <height>
    ii. Ensure big-endian order (most significant bit first)
        1. try printing one byte then reading it in, making sure its read in correctly
    iii. Edge cases:
        1. Test with very small (minimum) and very large (maximum) images to check performance and valgrind

H. read in compressed file **(Step -4)**

a. Testing:
    i. Ensure the header is read correctly and the width and height are extracted accurately
    ii. Confirm each 32-bit word is stored in the correct 2D array element
    iii. **Use ppmdiff.c to identify whether or not the margin of difference is appropriate for the entire file**

AT THIS POINT EVERYTHING SHOULD BE IMPLEMENTED SO DO OVERALL TESTS USING PPMDIFF.C AND COMPARING THE COMPRESSED AND DECOMPRESSED FILES WITH EACH OTHER MANUALLY WHILE TAKING INTO ACCOUNT LOSS