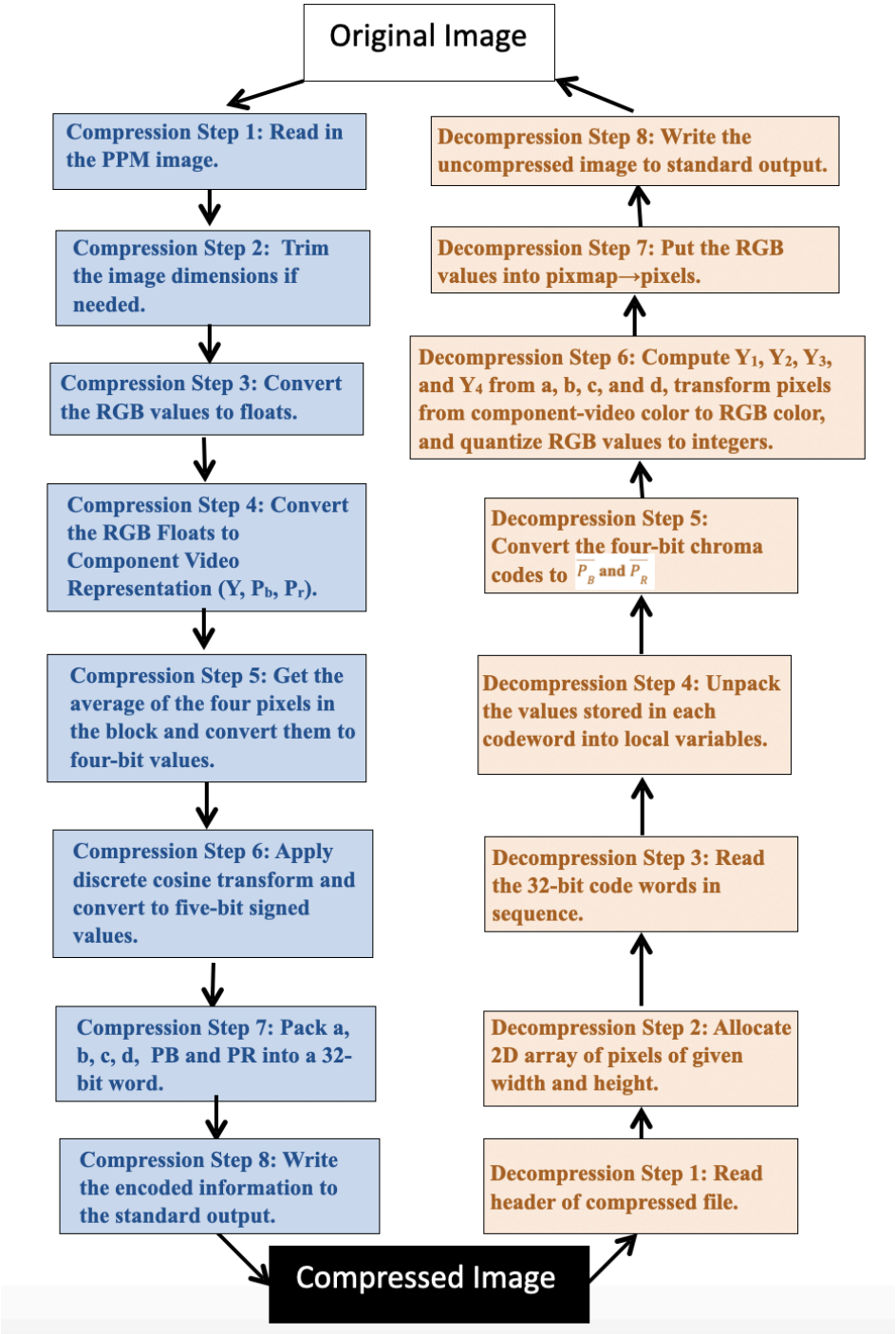
CS40 Arith Design Doc

Aoife O'Reilly (aoreil02) and Griffin Faecher (gfaech01)

February 23rd, 2025



Corresponding Details to the Steps

Compression Step 1: Read in the PPM image.

- Description:
  - Extract the width, height, and denominator from the header of the image file.
  - Read the image raster in from a file specified on the command line or from standard input.
  - Store the image using a UArray2_T.
- Input:
  - A FILE pointer to a PPM image.
- Output:
  - Pnm_ppm struct representing the image being read in.
- Information Lost:
  - None.

Compression Step 2: Trim the image dimensions if needed.

- Description:
  - If the width and height of the image are odd numbers, then trim the last row and/or column of the image so that they are even.
- Input:
  - Pnm_ppm struct representing the image.
- Output:
  - Pnm_ppm struct representing the image, but with the dimensions of the image possibly altered depending on if the dimensions were odd or even.
- Information Lost:
  - If the image has odd numbered dimensions, one column and one row of data will be lost. This is seen as valid in our ppmdiff function, so it is an allowable loss of data.

Compression Step 3: Convert the RGB values to floats.

- Description:
  - Scale each pixel's R, G, and B values to the range [0, 1] using:

$$R_{scaled} = \frac{R}{denominator}$$
$$G_{scaled} = \frac{G}{denominator}$$
$$B_{scaled} = \frac{B}{denominator}$$

  - Note that each denominator remains constant. Only the RGB values are scaled.
- Input:
  - Pnm_ppm struct representing the image.
- Output:
  - A UArray2d containing Pnm_float structs (defined below). The input and output arrays will be the same dimensions
  - `struct Pnm_float { float R; float G; float B; };`
- Information Lost:

- Converting to floats can result in data loss because floats cannot perfectly represent certain numbers. Even if double precision floats are used, depending on the length of the number being stored, the computer may have to round the number at some point to fit in the number of bits specified. This loss, in most cases, is very small and will not affect our program as a whole.

Compression Step 4: Convert the RGB Floats to Component Video Representation (Y, P_b, P_r).

- Description:
  - Use the linear transformations below to convert to component video representation for gamma-corrected signals (y, pb, pr):

$$y = 0.299 * r + 0.587 * g + 0.114 * b;$$
$$pb = -0.168736 * r - 0.331264 * g + 0.5 * b;$$
$$pr = 0.5 * r - 0.418688 * g - 0.081312 * b;$$

  - Take these new values, place them into a struct, and create a new 2b array with them.
- Input:
  - UArray2 containing Pnm_float structs from the previous step.
- Output:
  - New UArray2b containing structs detailed below. These will be organized in size two blocks.
    - `struct YPP_pixel { float y; float pb; float pr; };`
- Information Lost:
  - Once again, converting to floats can result in data loss because floats cannot perfectly represent certain numbers. There can be an overflow in the significand if the number cannot be perfectly represented, meaning some rounding has to occur.

Compression Step 5: Get the average of the four pixels in the block and convert them to four-bit values.

- Description:
  - Computes the average value ($\overline{P_B}$ and $\overline{P_R}$) of our four color difference values in the block.

    Then, convert these averages into 4 bit unsigned quantized values. This is done in the function below:

    ```
    unsigned Arith40_index_of_chroma(float x);
    ```

  - These quantized values, along with the original Y values, are placed into a struct and are placed in a new UArray2.
- Input:
  - UArray2b containing structs with component video pixels.
- Output:
  - Returns a UArray2 that is a quarter of the size of the input UArray2b. This output UArray2 contains intermediary structs detailed below. What used to be in each block of the previous UArray2b is now contained all in one element of the UArray2 inside a struct.
  - `struct intermediate_YPP { float Y1, Y2, Y3, Y4; uint8_t avg_Pb, avg_Pr;};`

- Information Lost:
    - By computing this average value and then quantizing the result, some data is inherently lost in the rounding and averaging process. We are no longer using the original pixel values.

Compression Step 6: Apply discrete cosine transform and convert to five-bit signed values.

- Description:
    - Use the discrete cosine transform (DCT) to transform the four Y values of the pixels into cosine coefficients a, b, c, and d which are defined as follows:
        - a = average brightness of the image
          $= (Y_4 + Y_3 + Y_2 + Y_1) / 4.0$
        - b = the degree to which the image gets brighter moving from top to bottom
          $= (Y_4 + Y_3 - Y_2 - Y_1) / 4.0$
        - c = the degree to which the image gets brighter moving from left to right.
          $= (Y_4 - Y_3 + Y_2 - Y_1) / 4.0$
        - d = the degree to which the pixels on one diagonal are brighter than the pixels on the other diagonal.
          $= (Y_4 - Y_3 - Y_2 + Y_1) / 4.0$
    - Then, convert only b, c, and d into five bit signed values, only keeping datum that are in between -.3 and .3. Wrap all of these values up into a struct, and make a new UArray2 to hold them.
- Input:
    - The UArray2 containing intermediate_YPP structs
- Output:
    - A new UArray2 containing processed data points in a struct detailed below. The output array should be the same size as the input array.
    - `struct processed_data { Uint16_t a; int_8t b, c, d; uint8_t avg_Pb, avg_Pr;};`
    -
- Information Lost:
    - Because values greater than ± .3 are thrown away, some data is lost. However, there are not many instances where the provided values are greater than .3, so we can forgive this data loss for increased accuracy.

Compression Step 7: Pack a, b, c, d, $\overline{P_B}$ and $\overline{P_R}$ into a 32-bit word.

- Description:
    - Takes values a, b, c, d, $\overline{P_B}$, and $\overline{P_R}$, and inserts them into a 32-bit word. Following the table below, we place these values into the 32 bit word using big endian for each element.

Value	Type	Width	LSB
a	Unsigned scaled integer	9 bits	23

b	Signed scaled integer	5 bits	18
c	Signed scaled integer	5 bits	13
d	Signed scaled integer	5 bits	8
$\overline{P_B}$	Unsigned	4 bits	4
$\overline{P_R}$	Unsigned	4 bits	0

- Input:
  - UArray2 contains proccessed_data structs containing all processed image data.
- Output:
  - An output UArray2 containing the 32 bit words containing the data to be output. These words are stored as ints.
- Information Lost:
  - No information is lost in this step.

Compression Step 8: Write the encoded information to the standard output.

- Description:
  - Write the header of the file, followed by all of the encoded information to standard output using Bitpack.
- Input:
  - All of the encoded information (i.e the 32 bit words)
- Output:
  - Nothing directly back to the code. Prints to standard output.
- Information Lost:
  - No information is lost in this step.

Decompression Step 1: Read header of compressed file.

- Description:
  - Use `fscanf` to read input from a file stream and parse through the header to obtain the width and height of the original PPM.
- Input:
  - FILE pointer to a file that stores a compressed PPM image.
- Output:
  - The height and width of the decompressed image (possibly trimmed from the original image).
- Information Lost:
  - No information is lost in this step.

Decompression Step 2: Allocate 2D array of pixels of given width and height.

- Description:

    - Create a 2D array using the following local variable struct:

```
struct Pnm_ppm pixmap = {
        .width = width,
        .height = height,
        .denominator = denominator,
        .pixels = array,
        .methods = methods };
```

    - Use the width and the height found in the previous step.
    - Use a denominator that is large enough to prevent quantization artifacts in the image but small enough to keep the PPM file size manageable.
    - The size of the initialized array should be the size of a colored pixel.
- Input:
    - Width and height of the original PPM.
- Output:
    - The 2D array.
- Information Lost:
    - No information is lost in this step.

Decompression Step 3: Read the 32-bit code words in sequence.

- Description:
    - Use a loop that iterates over the width and height to sequentially read in codewords from the compressed image file. Each codeword will be processed using a function defined in decompression step 4, and the extracted information will be stored in a struct, which will then be placed in a UArray2.
    - Check whether the file is too short, meaning there are too few codewords for the specified width and height or if the final codeword is incomplete.
- Input:
    - FILE pointer to a file that stores a compressed PPM image.
- Output:
    - A UArray2 that stores the unpacked data from each codeword.
- Information Lost:
    - No information is lost in this step.

Decompression Step 4: Unpack the values stored in each codeword into local variables.

- Description:
    - Use arithmetic operations (shifts) and masking to unpack the values a, b, c, d, and the coded $\overline{P_B}$ and $\overline{P_R}$ into local variables for each codeword.
- Input:
    - The 32-bit word to unpack.
- Output:
    - The four cosine coefficients a, b, c, d and the two four-bit chroma codes $\overline{P_B}$ and $\overline{P_R}$.
- Information Lost:

- No information is lost in this step.

Decompression Step 5: Convert the four-bit chroma codes to $\overline{P_B}$ and $\overline{P_R}$.

- Description:
  - Use the following formula to convert the four-bit chroma codes to $\overline{P_B}$ and $\overline{P_R}$ (the average values of the four pixels in each 2x2 block):

```
float Arith40_chroma_of_index(unsigned n);
```

- Input:
  - 2 four-bit chroma codes.
- Output:
  - 2 floats $\overline{P_B}$ and $\overline{P_R}$.
- Information Lost:
  - No information is lost in this step.

Decompression Step 6: Compute Y_1, Y_2, Y_3, and Y_4 from a, b, c, and d, transform pixels from component-video color to RGB color, and quantize RGB values to integers.

- Description:
  - Use the inverse of the discrete cosine transform below to compute Y1, Y2, Y3, and Y4 (pixel space) from a, b, c, and d (DCT space):

    $Y_1 = a - b - c + d$
    $Y_2 = a - b + c - d$
    $Y_3 = a + b - c - d$
    $Y_4 = a + b + c + d$
  - Use each Y , $\overline{P_B}$,and $\overline{P_R}$ to obtain red, green, and blue pixel integer values.
- Input:
  - Floats a, b, c, and d.
- Output:
  - Y1, Y2, Y3, and Y4, $\overline{P_B}$ and $\overline{P_R}$ all as integers.
- Information Lost:
  - Information will be lost because floating point arithmetic is not exact, so the resulting Y floats will not be the same as the values for the original image. Information will also be lost during the quantization.

Decompression Step 7: Put the RGB values into pixmap→pixels.

- Description:
  - Put the three RGB integer values into a pixel struct.
  - Pass the struct into the 2D array pixmap.
- Input:
  - Y1, Y2, Y3, and Y4, $\overline{P_B}$ and $\overline{P_R}$ all as integers
- Output:
  - None.

- Information Lost:
    - No information is lost in this step.

Decompression Step 8: Write the uncompressed image to standard output.

- Description:
    - Use ppmwrite to write the uncompressed image to standard output.
- Input:
    - The updated pixmap.
- Output:
    - Nothing directly back to the code. Prints to standard output.
- Information Lost:
    - No information is lost in this step.

Implementation Plan

*** As a note, whenever we implement a compression step, we will implement the decompression step that reverses it. We will use ppmdiff to check and make sure the images are the same (unless their dimensions differ by 1).

1. **Compression Step 1: Read in the PPM image.**
   Testing: Write the Pnm_PPM struct to stdout and use display to see if we have accurately captured the image in our 2D array.

2. **Decompression Step 1: Read header of compressed file.**
   Testing: Print the values we captured from `fscanf` and make sure they're what we expect them to be.

3. **Compression Step 2: Trim the image dimensions if needed.**
   Testing: Input images with odd widths, heights, and both odd widths and heights. After processing, print the widths and heights of the images and make sure that they're at most one less than what they used to be. Then, display the image and make sure it was an edge row or column that was deleted.

4. **Decompression Step 2: Allocate 2D array of pixels of given width and height.**
   Testing: Allocate a small test image (e.g., 4x4 pixels) and ensure proper memory allocation without segmentation faults. Print out the images widths and heights to make sure it has the expected dimensions.

5. **Compression Step 3: Convert the RGB values to floats.**

Testing: Print all pixels after conversion. If any print above 1.0, throw an error message. Then, Use an image with all red pixels (255, 0, 0) and ensure that all converted red values are 1.0 while green and blue are 0.0.

6. **Decompression Step 3: Read the 32-bit code words in sequence.**
   Testing:
   Read a compressed file and print the binary representation of the first few codewords.

7. **Compression Step 4: Convert the RGB Floats to Component Video Representation (Y, P_b, P_r).**
   Testing: Convert a pure grayscale image (same R, G, B values for all pixels) and verify that P_b and P_r are 0 for all pixels. Then convert a colored image and verify that P_b and P_r are not 0 for all pixels.

8. **Decompression Step 4: Unpack the values stored in each codeword into local variables.**
   Testing: Print unpacked (a, b, c, d, P_B, P_R) values from a known compressed file and compare with expected values. Then, verify that negative b, c, d values are correctly extracted as signed 5-bit numbers.

9. **Compression Step 5: Get the average of the four pixels in the block and convert them to four-bit values.**
   Testing: Input a 2x2 block with Pb/Pr values that are pre-specified and compute the expected average manually. Print and verify the computed result.

10. **Decompression Step 5: Convert the four-bit chroma codes to $\overline{P_B}$ and $\overline{P_R}$**
    Testing: Convert Pb/Pr indices back to floating-point values and compare with expected values from the compression step.

11. **Compression Step 6: Apply discrete cosine transform and convert to five-bit signed values.**
    Testing: Input a 2x2 block with identical Y values and verify that b, c, and d all become 0 (since no variation exists).

12. **Decompression Step 6: Compute Y_1, Y_2, Y_3, and Y_4 from a, b, c, and d, transform pixels from component-video color to RGB color, and quantize RGB values to integers.**
    Testing: Convert the block back to RGB and verify range [0,255] is maintained. If not, throw an error statement.

13. **Compression Step 7: Pack a, b, c, d, $\overline{P_B}$ and $\overline{P_R}$ into a 32-bit word.**
    Testing: Pack manually selected values for a, b, c, d, P_B, P_R into a 32-bit word and verify that each field is stored correctly by printing out the total value and checking the 32-bit word to make sure it represents the same bits as we expect.

14. **Decompression Step 7: Put the RGB values into pixmap→pixels.**

Testing: Print the first few decompressed RGB pixel values and compare them with the expected outputs from the compression test.

15. **Compression Step 8: Write the encoded information to the standard output.**
    Testing: Write a compressed file and verify that the output matches the expected format.

16. **Decompression Step 8: Write the uncompressed image to standard output.**
    Testing: Use diff and ppmdiff to ensure that our image has been decompressed correctly.