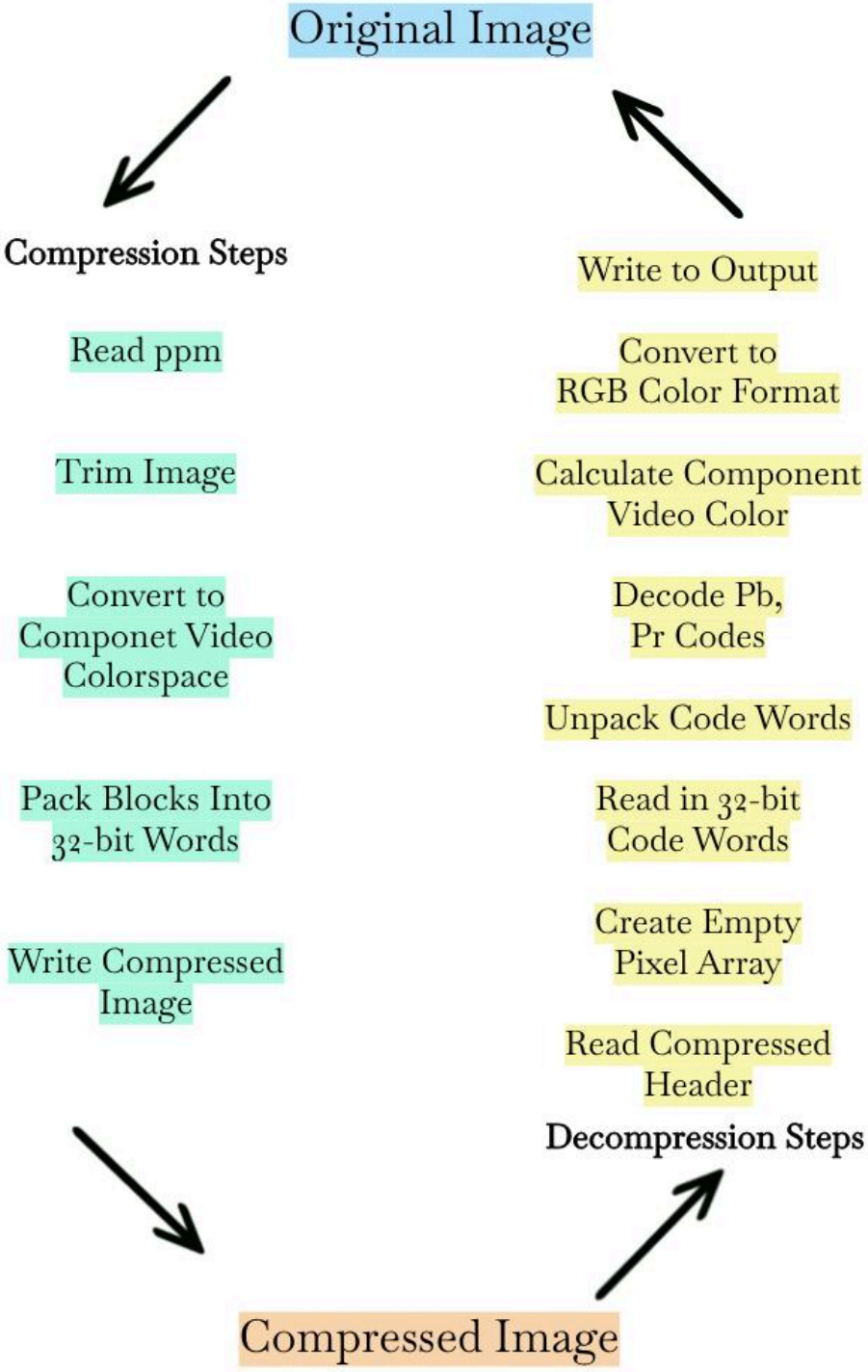
Jack Adkins (jadkin05)
Arshiya Lall (alall01)

Image Compression & Decompression Design Doc

For this assignment, our order of implementation will be box by box through each step of the compression process, but in between each one, we will implement the corresponding decompression step(s) that undo the compression step we just implemented. This will allow us to see the functionality of the program better as we build it and will be a crucial part of our testing throughout the entire assignment to make sure we aren't missing any small errors. We will also be using our ppmdiff implementation to help with this so we can see if/how the image is changing after going through our compression/decompression process. We have our entire compression and decompression process mapped out in the chart at the end, with explanations for each step detailed in the pages before.


Original Image
Compression Steps
Read ppm
Trim Image
Convert to Componet Video Colorspace
Pack Blocks Into 32-bit Words
Write Compressed Image
Compressed Image
Decompression Steps
Read Compressed Header
Create Empty Pixel Array
Read in 32-bit Code Words
Unpack Code Words
Decode Pb, Pr Codes
Calculate Component Video Color
Convert to RGB Color Format
Write to Output

COMPRESSION

Compression Step 1: Read in a ppm image

- Use the pnm.h interface to assist with error handling
- Do it in row-major order
- **Input:** A provided file or stdin
- **Output:** A pnm_ppm object containing the given pnm image
- **Information Loss:** No information lost in this step

Testing:

- Use a simple printing apply function to print of the ppm in row-major order and compare it to the given input
- Check for improper input by passing different versions of bad input (empty, non-ppm files etc.)

Compression Step 2: Trim image dimensions

- Ensure that width and height of image are even numbers by trimming the last row/column if needed
- **Input:** pnm_ppm with width and height
- **Output:** Trimmed pnm_ppm with even width and height dimensions
- **Information Loss:** Will lose the information from the last row or column of the ppm image if the # rows or #cols is odd

Testing:

- Test this function on an image with odd width and height to check the dimensions become even
- Test this function on an image with even width and height to check that the dimensions remain unchanged

---

Compression Step 3: Convert each pixel into Component Video Colorspace format

- Use the row-major mapping function to access each pixel individually
- Use the formulas in the spec to calculate the y, Pb, and Pr values from the pixel's previous RGB values
- Set the pixel's new value to be y, Pb, and Pr
- **Input:** A pnm_ppm object with RGB pixel values
- **Output:** A pnm_ppm object with Component Video pixel values

- **Information Loss:** No information lost, same color just represented in a different format

Testing:

- To ensure the formula is correct, do manual/by-hand calculations of what we expect the y, Pb, and Pr values to be for a few different RGB inputs and make sure they align with what our program is getting
- Test extreme case of white (255,255,255) and black (0,0,0)
- Assertions for the possible ranges of the converted values (0 to 255 for y, and -127.5 to 127.5 for Pb and Pr)

Compression Step 4: Pack 2x2 block into 32-bit word

- Use row major mapping to access each pixel
    - If the pixel is a “top-left” pixel of a block (both col an row % 2 == 0) access the rest of the 2x2 block (c + 1, r) (c, r+ 1) (c+1, r+1)
    - If the pixel is not a “top-left” pixel, move on with the mapping function
- For every 2x2 block of pixels, take the DC value (average value of 4 pixels in block)
- Quantize the chroma value (PB and PR) by taking the average values and convert them into 4-bit values using *unsigned Arith40_index_of_chroma(float x)*
- Use DCT to transform the Y values of pixels into cosine coefficients
- Quantize DCT coefficients b, c, d to 5-bit signed values
- Pack values a, b, c, d, quantized_PB, quantized_PR into 32-bit word
- Use the bitpack module to assist in these operations
- **Input:** Y/PB/PR values for 2x2 block of pixels
- **Output:** 32-bit packed word with quantized chroma and DCT coefficients
- **Information Loss:** information is lost in this step because of the quantization of PB, PR and Y coefficients to help compress data

Testing

- Test the quantization of PB, PR values by comparing both the original and quantized values to verify their difference
- Use the inverse DCT function on Y values to revert transformed data back to original form

---

Compression Step 5: Write the Compressed pnm to output

- Order the 32-bit words in Big Endian order
- Print out the corresponding header
- Write each word on a new line to stdout

- **Input:** The compressed pnm_ppm file in the form of 32-bit words
- **Output:** The collection of 32-bit words along with their corresponding header
- **Information Loss:** No information lost in this step

Testing:

- Print out the list of 32-bit words to manually check if they are in Big Endian order
- Calculate the expected number of 32-bit words and compare it with the actual number output

---

DECOMPRESSION

Decompressor Step 1: Read compressed image header

- Parse in the compressed image's header to receive information on width and height of image by using fscanf with string used to write header
- **Input:** compressed file
- **Output:** width and height of image
- **Information Loss:** none in this step

Testing plan:

- Test the reading of the header by providing valid and invalid header information
- Ensure correct width and height is outputted

Decompressor Step 2: Create an empty 2D array of pixels

- Allocate space for the pnm_ppm object
- Set the width and height according to the data collected from fscanf
- Also assign the size, denominator, pixels array, and methods suite to the object
- **Input:** Variables containing the values for the pnm's dimensions (width, height, size, etc.)
- **Output:** An empty pnm_ppm object ready to have pixel values read into it
- **Information Loss:** No information lost in this step

Testing:

- Experiment with denominator values so that the pnm isn't too large and the image doesn't show quantization artifacts
- Create temporary apply functions to access each pixel in the pnm_ppm
- Print out the width and height dimensions and compare them to their expected values

- Access the methods suite through the pnm_ppm object to ensure they can be called

---

Decompressor Step 3: Read 32-bit words into a sequence

- Unpack the coefficients for a, b, c, d, and quantized PB/PR values for every 32-bit word
- If number of code words is too low for a given width/height, fail with CRE
- **Input:** 32-bit word
- **Output:** Y, PB, PR values for a 2x2 block
- **Information lost:** no information lost as we are simply unpacking values

Testing:

- Test on a variety of images with varying block sizes
- Verify unpacking by packing, then unpacking and verifying values
- Pass in file that has incomplete last code word, or the number of codewords is too low for dimensions and ensure failure with CRE

Decompressor Step 4: Unpack 32-bit Code Words

- Unpack 32-bit code words and finds values of a, b, c, d, and PB, PR (quantized chroma values)
- Use the bitpack module to assist in these operations
- **Input:** 32-bit code word
- **Output:** values for a, b, c, d, and coded PB and PR
- **Information lost:** slight loss of information, because of quantization during the packing process. This is because of limitations in bit-wise storage.

Testing:

- Select a 32-bit code word that is known and test unpacking function by comparing program-outputted values of a, b, c, d, and PB/PR against known values
- Check with ‘0’ or ‘maxval’ to test upper/lower bounds

Decompressor Step 5: Decode the Pb and Pr codes

- Use the provided function to do the actual decoding
- Store the Pb and Pr floats in the color variables in each pixel’s struct
- **Input:** 2 unsigned four-bit chroma codes for each pixel
- **Output:** The Pb and Pr float values for each pixel

- **Information Loss:** No information lost. We are just converting from chroma codes to floats

Testing:

- Give the function 4-bit chroma codes of known floats to ensure the given formula is correct
- Pass invalid input types and impossible codes to check for runtime errors

---

Decompressor Step 6: Calculate every pixel's component video color

- Calculate the four lumina (Y) values, one for each pixel in a 2x2 block
- Use the DCT equations provided in the spec with a, b, c, and d as variables
- Store each each pixel lumina value in its struct similar to the Pb and Pr codes from the previous step
- **Input:** a, b, c, and d values for each 2x2 pixel block
- **Output:** The lumina value for each pixel in the same 2x2 block
- **Information Loss:** No information loss in this step, another conversion between similar variables

Testing

- Give the function practice variables to convert and check it against the conversion we do by hand
- Give the function variables not within the range of possible inputs and ensure runtime errors are returned

---

Decompressor Step 7: Convert from component video to RGB colors

- For every pixel in the 2x2 block, this function converts Y, PB, PR values to RGB.
- Quantizes RGB values, making sure they are in the expected range
- **Input:** Y values– Y1, Y2, Y3, Y4, PB and PR
- **Output:** RGB values for every pixel in a given 2x2 block
- **Information Loss:** information may be lost at this step, as during quantization, RGB values are converted to integers

Testing:

- Test to ensure that program is rounding RGB values correctly with large values
- Take known Y, PB, PR values and convert them; check that the resulting RGB values are within the appropriate range

---

Decompressor Step 8: Write pixel map to output

- Function will write decrompessed image (PPM) to stdout
- Pixel array will be in PPM form with appropriate header
- **Input:** pnm_ppm struct with decompressed pixel data
- **Output:** PPM written to stdout
- **Information loss:** no information lost in this step

Testing

- Check to see if decompressed image matches PPM format
- Compare output to original image to see if compression and decompression has happened correctly

IMPLEMENTATION PLAN ORDER

1. Read ppm (Compression)
2. Write to output (Decompression)
3. Trim Image (Compression)
4. Convert to Component Video Colorspace (Compression)
5. Convert to RGB color format (Decompression)
6. Pack Blocks into 32-bit words (Compression)
7. Create an Empty Pixel Array (Decompression)
8. Read in 32-bit Code words (Decompression)
9. Unpack Code Words (Decompression)
10. Decode Pb, Pr codes (Decompression)
11. Write Compressed Image (Compression)
12. Read Compressed Header (Decompression)