Arith homework design document

Authors: Ella Hou (ehou02) and Darius-Stefan Iavorschi (diavor01)

This document describes a step-by-step process for compressing a PPM image and later decompressing it back into an image. The overall process is divided into two parts—compression and decompression—with thorough testing at each stage to verify correctness.

Order of implementation:

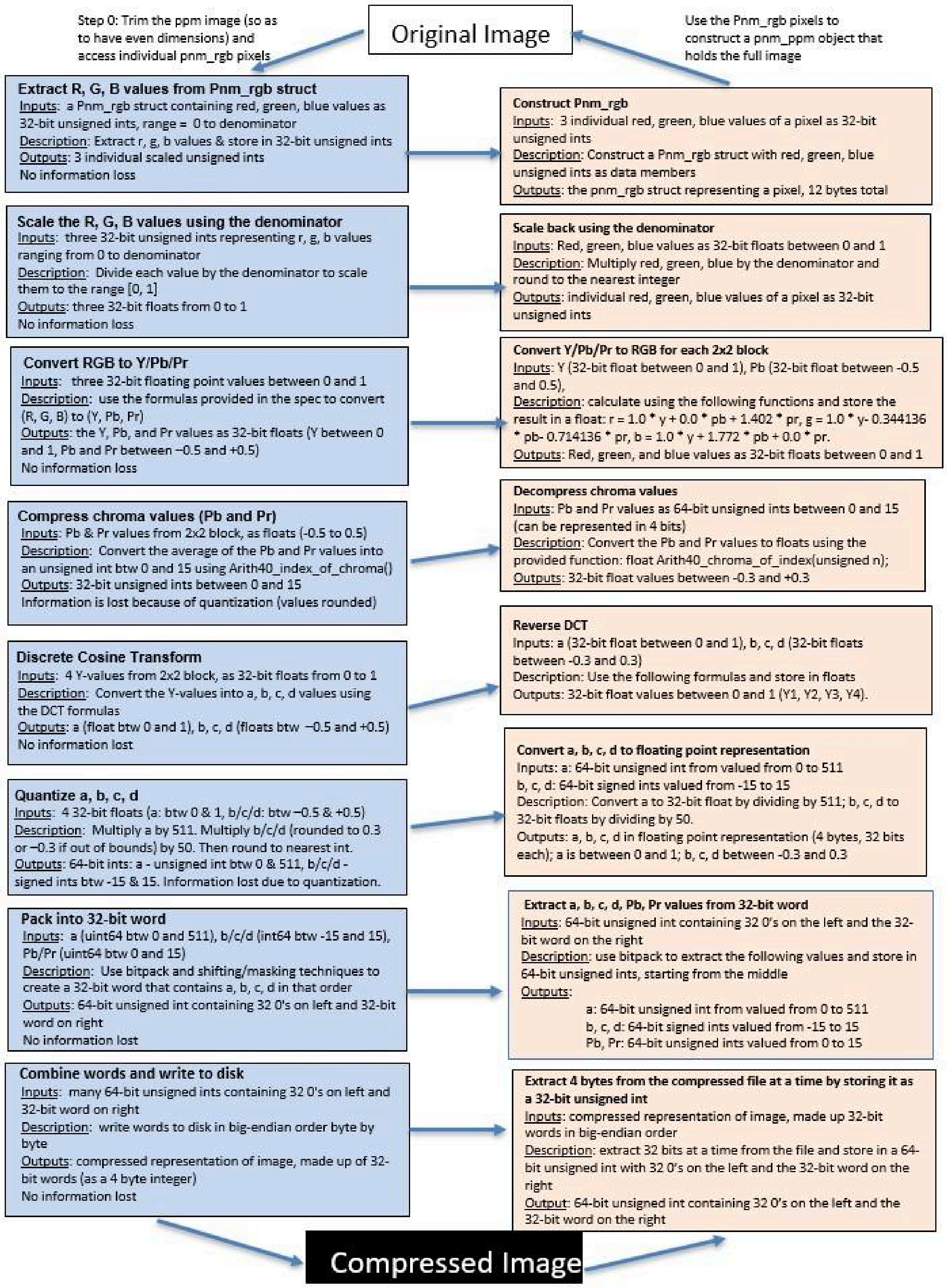
- The compression steps are implemented in order, from 0 to 8.
- Each compression step is implemented in conjunction with its decompression counterpart step at a time and their behavior is tested together.

General testing notes:

- Develop functions that take in signed and unsigned integers, printing them as 32-bit or 64-bit binary strings to help visualize at each step.
    - Do this by masking with 1 and shifting right
- Files we are going to use:
    - Basic small example: feep.ppm
    - Files with odd dimensions
    - Files with even dimensions
    - Files with different denominators (255, 65536, 10, etc)
    - Files with pixel blocks that result in the b, c, and d values above 0.3 or below -0.3
    - Files specifically selected to create either zero or maximum possible information loss

Overall Testing patterns for each step (for both compression and decompression):

- Print Verification:
    - Print values in both decimal and binary to confirm correct transformations (for signed and unsigned ints). Printing in binary will require our utility function described above.
- Type & Size Checks:
    - Use typeof() and sizeof() to ensure proper data types and sizes.
- Edge Case Handling:
    - Test with values that are out of bounds, incorrect types, or intentionally produce high information loss or no information loss.
    - See specific examples under each step
- Cross-Validation:
    - Ensure that each compression step has a corresponding decompression step that accurately recovers the data (within acceptable loss limits where appropriate), using ppmdiff.



Step 0: Trim the ppm image (so as to have even dimensions) and access individual pnm_rgb pixels

Extract R, G, B values from Pnm_rgb struct
Inputs: a Pnm_rgb struct containing red, green, blue values as 32-bit unsigned ints, range = 0 to denominator
Description: Extract r, g, b values & store in 32-bit unsigned ints
Outputs: 3 individual scaled unsigned ints
No information loss

Scale the R, G, B values using the denominator
Inputs: three 32-bit unsigned ints representing r, g, b values ranging from 0 to denominator
Description: Divide each value by the denominator to scale them to the range [0, 1]
Outputs: three 32-bit floats from 0 to 1
No information loss

Convert RGB to Y/Pb/Pr
Inputs: three 32-bit floating point values between 0 and 1
Description: use the formulas provided in the spec to convert (R, G, B) to (Y, Pb, Pr)
Outputs: the Y, Pb, and Pr values as 32-bit floats (Y between 0 and 1, Pb and Pr between −0.5 and +0.5)
No information loss

Compress chroma values (Pb and Pr)
Inputs: Pb & Pr values from 2x2 block, as floats (-0.5 to 0.5)
Description: Convert the average of the Pb and Pr values into an unsigned int btw 0 and 15 using Arith40_index_of_chroma()
Outputs: 32-bit unsigned ints between 0 and 15
Information is lost because of quantization (values rounded)

Discrete Cosine Transform
Inputs: 4 Y-values from 2x2 block, as 32-bit floats from 0 to 1
Description: Convert the Y-values into a, b, c, d values using the DCT formulas
Outputs: a (float btw 0 and 1), b, c, d (floats btw −0.5 and +0.5)
No information lost

Quantize a, b, c, d
Inputs: 4 32-bit floats (a: btw 0 & 1, b/c/d: btw −0.5 & +0.5)
Description: Multiply a by 511. Multiply b/c/d (rounded to 0.3 or −0.3 if out of bounds) by 50. Then round to nearest int.
Outputs: 64-bit ints: a - unsigned int btw 0 & 511, b/c/d - signed ints btw -15 & 15. Information lost due to quantization.

Pack into 32-bit word
Inputs: a (uint64 btw 0 and 511), b/c/d (int64 btw -15 and 15), Pb/Pr (uint64 btw 0 and 15)
Description: Use bitpack and shifting/masking techniques to create a 32-bit word that contains a, b, c, d in that order
Outputs: 64-bit unsigned int containing 32 0's on left and 32-bit word on right
No information lost

Combine words and write to disk
Inputs: many 64-bit unsigned ints containing 32 0's on left and 32-bit word on right
Description: write words to disk in big-endian order byte by byte
Outputs: compressed representation of image, made up of 32-bit words (as a 4 byte integer)
No information lost

Use the Pnm_rgb pixels to construct a pnm_ppm object that holds the full image

Construct Pnm_rgb
Inputs: 3 individual red, green, blue values of a pixel as 32-bit unsigned ints
Description: Construct a Pnm_rgb struct with red, green, blue unsigned ints as data members
Outputs: the pnm_rgb struct representing a pixel, 12 bytes total

Scale back using the denominator
Inputs: Red, green, blue values as 32-bit floats between 0 and 1
Description: Multiply red, green, blue by the denominator and round to the nearest integer
Outputs: individual red, green, blue values of a pixel as 32-bit unsigned ints

Convert Y/Pb/Pr to RGB for each 2x2 block
Inputs: Y (32-bit float between 0 and 1), Pb (32-bit float between -0.5 and 0.5),
Description: calculate using the following functions and store the result in a float: r = 1.0 * y + 0.0 * pb + 1.402 * pr, g = 1.0 * y- 0.344136 * pb- 0.714136 * pr, b = 1.0 * y + 1.772 * pb + 0.0 * pr.
Outputs: Red, green, and blue values as 32-bit floats between 0 and 1

Decompress chroma values
Inputs: Pb and Pr values as 64-bit unsigned ints between 0 and 15 (can be represented in 4 bits)
Description: Convert the Pb and Pr values to floats using the provided function: float Arith40_chroma_of_index(unsigned n);
Outputs: 32-bit float values between -0.3 and +0.3

Reverse DCT
Inputs: a (32-bit float between 0 and 1), b, c, d (32-bit floats between -0.3 and 0.3)
Description: Use the following formulas and store in floats
Outputs: 32-bit float values between 0 and 1 (Y1, Y2, Y3, Y4).

Convert a, b, c, d to floating point representation
Inputs: a: 64-bit unsigned int from valued from 0 to 511
b, c, d: 64-bit signed ints valued from -15 to 15
Description: Convert a to 32-bit float by dividing by 511; b, c, d to 32-bit floats by dividing by 50.
Outputs: a, b, c, d in floating point representation (4 bytes, 32 bits each); a is between 0 and 1; b, c, d between -0.3 and 0.3

Extract a, b, c, d, Pb, Pr values from 32-bit word
Inputs: 64-bit unsigned int containing 32 0's on the left and the 32-bit word on the right
Description: use bitpack to extract the following values and store in 64-bit unsigned ints, starting from the middle
Outputs:
a: 64-bit unsigned int from valued from 0 to 511
b, c, d: 64-bit signed ints valued from -15 to 15
Pb, Pr: 64-bit unsigned ints valued from 0 to 15

Extract 4 bytes from the compressed file at a time by storing it as a 32-bit unsigned int
Inputs: compressed representation of image, made up 32-bit words in big-endian order
Description: extract 32 bits at a time from the file and store in a 64-bit unsigned int with 32 0's on the left and the 32-bit word on the right
Output: 64-bit unsigned int containing 32 0's on the left and the 32-bit word on the right

More detailed steps and testing below:

Step 0: Trim the Image

- Inputs: the original image of pixels in rgb, made of ppm_rgb structs
- Description: Ensure that the image dimensions are even by trimming an extra row and/or column if necessary
- Output: an image with even dimensions, held in a UArray2_T object
- Information loss: yes if the height and/or width was odd, requiring rows and/or columns to be trimmed
- Testing edge cases:
    - Odd height and width -> trim both
    - Odd height, even width -> trim height
    - Even height, odd width -> trim width
    - Even height, even width -> no trimming
- No decompression step to test in conjunction with (lost rows/columns are never recovered)

Step 1: Extract R, G, B values from each Pnm_rgb struct

- Inputs: a Pnm_rgb struct containing red, green, and blue values stored in unsigned ints from 0 to *denominator*. Total size: 12 bytes.
- Description: Extract the red, green, and blue values and store into unsigned ints (4 bytes, 32 bits each)
- Outputs: 3 individual unsigned ints representing the red, green, blue values (4 bytes. 32 bits each)
- No information lost in this step
- Testing edge cases:
    - red, green, blue values above the denominator or below 0 -> error
    - Input is not a pnm_rgb struct of size 12 bytes -> errors
- Test in conjunction with: Decompression Step 8

Step 2: Scale the R, G, B values using the denominator

- Inputs: 3 unsigned ints for the red, green, and blue values. Between 0 and *denominator*, 4 bytes (32 bits) each.
- Description: Divide each value by the denominator to scale them to the range [0, 1] and store in a 32-bit float.
- Outputs: 3 floating point values from 0 to 1 (representing the red, green, and blue values), each of size 4 bytes (32 bits)
- No information is lost in this step
- Testing edge cases:
    - Inputs that are out of bounds of 0 to *denominator* -> error
    - *Denominator* is not between 0 and 65536 -> error
    - Inputs that are not unsigned ints -> error

    - Inputs where the denominator is very large (up to 65536) -> should still work
- Test in conjunction with: Decompression Step 7

Step 3: Convert RGB to Y/Pb/Pr
- Inputs: 3 floating point values representing the R, G, B values of a pixel, each 4 bytes (32 bits)
- Description: use the provided formulas to convert (R, G, B) to (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;
- Output: the Y, Pb, and Pr values as floats (4 bytes, 32 bits each)
    - Y value is between 0 and 1, and the Pb and Pr values are between -0.5 and 0.5.
- No information is lost in this step
- Testing edge cases:
    - r, g, b values outside of range 0-1 -> error
    - r, g, b values are not floating points -> error
    - R, g, and b values are all 0 and 1 respectively
    - Create 2 * 2 blocks containing R, G, B values that correspond to extreme Pb and Pr values (since we are taking the average, the information loss will be maximum)
        - Pixel 1: R and g are 0 while b is 1 (Pb will get the maximum value 0.5)
        - Pixel 2: R is 1 while g and b are 0 (Pr will get the maximum value 0.5)
        - Pixel 3: R and g are 1 while b is 0 (Pb will get the minimum value -0.5)
        - Pixel 4: R is 0 while g and b are 1 (Pr will get the minimum value -0.5)
    - Create 2 * 2 blocks containing R, G, B values that correspond to the same Pb and Pr values (no information loss by taking the average)
        - Pick 4 identical pixels
- Test in conjunction with: Decompression Step 6

Step 4: Compress chroma values (Pb and Pr)
- Inputs: 4 sets of Pb and Pr values from 2x2 block of pixels, in floating point representation (-0.5 and +0.5). Size is 4 bytes (32 bits) each.
- Description:
    - Take the average of the 4 Pb and Pr values in a 2x2 block
    - Convert the average into 4-bit values using the provided function: unsigned Arith40_index_of_chroma(float x);
- Outputs: quantized representation of the chroma value, 32-bit unsigned ints between 0 and 15 (can be represented using 4 bits)
- There is information lost because the chroma values are quantized within the function
- Testing edge cases:
    - Any input value is not between -0.5 and 0.5 -> error
    - Any input value is not a floating point of 4 bytes -> error
    - Test in conjunction with: Decompression Step 5

Step 5: Discrete Cosine Transform

- Inputs: 4 Y-values from 2x2 block of pixels, in floating point representation (from 0 to 1)
- Description: Apply the formulas
    - a =(Y4 +Y3+Y2+Y1)/4.0, representing the average brightness of the image.
    - b = (Y4 +Y3 −Y2−Y1)/4.0, representing the degree to which the image gets brighter as we move from top to bottom;
    - c =(Y4 −Y3 +Y2−Y1)/4.0, representing the degree to which the image gets brighter as we move from left to right;
    - d =(Y4 −Y3−Y2+Y1)/4.0, the degree to which the pixels on one diagonal are brighter than the pixels on the other diagonal;
- Output: a, b, c, d in floating point representation (4 bytes, 32 bits each)
    - a: between 0 and 1;
    - b, c, d are between -0.5 and 0.5
- No information is lost in this step
- Testing edge cases:
    - Y-values not between 0 and 1 -> error
    - Y-values not floating points of size 32 bits (4 bytes) -> error
    - Choose 3 different edge cases for Y1, Y2, Y3, and Y4 so as b, c, and d are one at a time out the range [- 0.3, 0.3]
    - Choose equal Y values
- Test in conjunction with: Decompression Step 4

Step 6: Quantize a, b, c, d

- Inputs: floating point values (4 bytes, 32 bits each)
    - a: between 0 and 1
    - b, c, d are between -0.5 and 0.5
- Description:
    - represent a using 9 bits by multiplying by 511 and rounding to the nearest integer
    - Represent b, c, d using 5 bits by multiplying by 50 and rounding to the nearest unsigned int integer
        - If the original value of b, c, and/or d is outside of the range [-0.3, +0.3], first round to -0.3 or +0.3, then multiply by 50
    - Store the results in 64-bit ints (unsigned for a, signed for b, c, d)
- Outputs:
    - The quantized values (specifics below):
        - a: 64-bit unsigned int between 0 and 511 (can be represented in 9 bits)
        - b, c, d: 64-bit signed ints between -15 and 15 (can be represented using 5 bits)
- Information lost: yes, through quantization
- Testing edge cases:
    - Inputs of b, c, d not in the range of [-0.3 to +0.3] -> should round and still work, despite high information loss
    - Inputs of b, c, d out of range of [-0.5, +0.5] -> error
    - Input of a out of range [0, 1] -> error

- Test in conjunction with: Decompression Step 3

Step 7: Pack into 32-bit word
- Inputs: 64-bit ints:
    - a - unsigned int between 0 and 511
    - b, c, and d: signed ints between -15 and 15
    - Pb and Pr: unsigned ints between 0 and 15
- Description: use bitpack to create a 32-bit word in a 64-bit unsigned int where:
    - bits 31-23 are a
    - bits 22-18 are b
    - bits 17-13 are c
    - bits 12-8 are d
    - bits 7-4 are Pb
    - bits 3-0 are Pr
- Output: a 64-bit unsigned int with 32 0's on the left half and a 32-bit word on the right half
- No information is lost in this step
- Testing edge cases:
    - Input of a is not an unsigned int between 0 and 511 -> error
    - Inputs of b, c, d are not signed ints between -15 and 15 -> error
    - Inputs of Pb and Pr are not unsigned ints between 0 and 15 -> error
- Test in conjunction with: Decompression Step 2

Step 8: Write to disk
- Inputs: many 64-bit unsigned ints containing 32 0's on the left half and a 32-bit word on the right half
- Description: start in the middle of the 64-bit unsigned int. Write byte-by-byte to disk in big-endian order, using putchar() in row-major order
- Output: compressed representation of image, made up 32-bit words in big-endian order
- No information is lost in this step
- Testing edge cases:
    - Inputs are not 64-unsigned ints with the first 32 values being 0s -> error
- Test in conjunction with: Decompression Step 1

<u>Decompression:</u>
- No information loss happens at any point within the decompression process

Step 1: Extract 4 bytes from the compressed file at a time by storing it as a 32-bit unsigned int;
- Inputs: compressed representation of image, made up 32-bit words in big-endian order
- Description: extract 32 bits at a time from the file and store in a 64-bit unsigned int with 32 0's on the left and the 32-bit word on the right
- Output: 64-bit unsigned int containing 32 0's on the left and the 32-bit word on the right

- Testing edge cases:
    - Number of codewords is too low for the stated width and height -> CRE
    - Excess data after the last codeword -> should still work
- Test in conjunction with Step 1 of Compression (using the input files we compressed) by printing the binary version (using our testing function) to verify it matches

Step 2: Extract a, b, c, d, Pb, Pr values from 32-bit word

- Inputs: 64-bit unsigned int containing 32 0's on the left and the 32-bit word on the right
- Description: extract the following values and store in 64-bit unsigned ints
    - a: first 9 bits (spots 31 to 23)
    - b: next 5 bits (spots 22 to 18)
    - c: next 5 (spots 17 to 13)
    - d: next 5 bits (spots 12 to 8)
    - Pb: next 4 bits (spots 7 to 4)
    - Pr: last 4 bits (spots 3 to 0)
- Outputs:
    - a: 64-bit unsigned int from valued from 0 to 511
    - b, c, d: 64-bit signed ints valued from -15 to 15
    - Pb, Pr: 64-bit unsigned ints valued from 0 to 15
- Testing edge cases:
    - The input is not a 64-bit unsigned int with 32 0's in spots 32-63 -> error
- Test in conjunction with Compression Step 7

Step 3: convert a, b, c, d to floating point representation

- Inputs:
    - a: 64-bit unsigned int from valued from 0 to 511
    - b, c, d: 64-bit signed ints valued from -15 to 15
- Description:
    - Convert a to 32-bit float by dividing by 511
    - Convert b, c, d to 32-bit floats by dividing by 50
- Outputs: a, b, c, d in floating point representation (4 bytes, 32 bits each)
    - a is between 0 and 1
    - b, c, d between -0.3 and 0.3
- Testing edge cases:
    - a is not an unsigned int between 0 and 511 -> error
    - b, c, d are not signed ints between -15 and 15 -> error
- Test in conjunction with Compression Step 6

Step 4: Convert a, b, c, d to Y-values

- Inputs: a (32-bit float between 0 and 1), b, c, d (32-bit floats between -0.3 and 0.3)
- Description: Use the following formulas and store in floats
    - $Y1 = a-b-c+d$
    - $Y2 = a-b+c-d$

    - Y3 = a+b−c−d
    - Y4 = a+b+c+d
- Outputs: 32-bit float values between 0 and 1 (Y1, Y2, Y3, Y4). However, based on the information loss, the Y-values may turn out to be out of range [0, 1]. We must see if this is the intended behavior or an error.
- Testing edge cases:
    - a is not a float between 0 and 1 -> error
    - b, c, d are not floats between -0.3 and 0.3 -> error
- Test in conjunction with Step 5 of Compression

Step 5: Decompress chroma values
- Inputs: Pb and Pr values as 64-bit unsigned ints between 0 and 15 (can be represented in 4 bits)
- Description: Convert the Pb and Pr values to floats using the provided function:
    - float Arith40_chroma_of_index(unsigned n);
- Outputs: 32-bit float values between -0.3 and +0.3
- Testing edge cases:
    - Input Pb and Pr values are not unsigned ints between 0 and 15 -> error
- Test in conjunction with Step 4 of Compression

Step 6: Convert Y/Pb/Pr to RGB for each 2x2 block
- Inputs: Yi (32-bit float between 0 and 1), Pb (32-bit float between -0.5 and 0.5),
- Description: calculate using the following functions and store the result in a float:
    - r = 1.0 * y + 0.0 * pb + 1.402 * pr;
    - g = 1.0 * y- 0.344136 * pb- 0.714136 * pr;
    - b = 1.0 * y + 1.772 * pb + 0.0 * pr;
- Outputs: Red, green, and blue values as 32-bit floats between 0 and 1
- Testing edge cases:
    - Input Yi value is not a float between 0 and 1 -> error
    - Input Pb and Pr values are not floats between -0.3 and +0.3
- Test in conjunction with Step 3 of Compression

Step 7: Scale back using the denominator
- Inputs: Red, green, blue values as 32-bit floats between 0 and 1
- Description: Multiply red, green, blue by the denominator and round to the nearest integer
- Outputs: individual red, green, blue values of a pixel as 32-bit unsigned ints
- Testing edge cases:
    - Input r, g, b values are not floats between 0 and 1 -> error
    - The denominator is very large (around 65536) -> should still work
- Test in conjunction with Step 2 of Compression

Step 8: Construct Pnm_rgb
- Inputs: individual red, green, blue values of a pixel as 32-bit unsigned ints

- Description: Construct a Pnm_rgb struct with red, green, blue unsigned ints as data members
- Outputs: the pnm_rgb struct representing a pixel, 12 bytes total
- Testing edge cases:
    - Input values are not 32-bit unsigned ints -> error
    - Input values are not between 0 and denominator -> error
- Test in conjunction with Step 2 of Compression

Final step:

- Put all pixels into a pnm_ppm object to represent the entire image!
- Testing:
    - Use ppmdiff: the final decompressed image should be at most 2.5% different from the original one
    - Display the image: check that they look roughly the same to the human eye