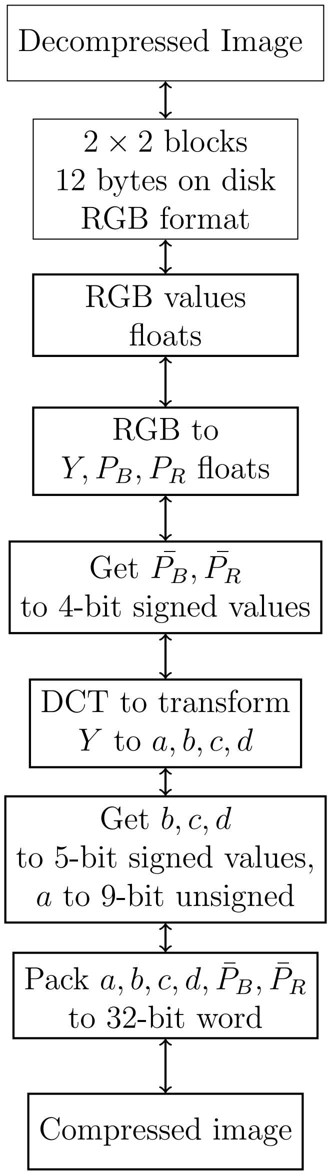
1 Compression Diagram



1.1 Decompressed Image $\rightarrow 2 \times 2$ blocks, 12 bytes on disk, RGB format

Summary: Read in the image using the PNM reader and store in a 2D blocked, unboxed array (blocksize = 2 for 2×2 blocks).
Inverse: Print the image by passing the 2D, blocked, unboxed array to the PNM writer.
Input: File/stream containing the PPM image.
Output: 2D blocked, unboxed array with PPM image.
Information Lost: Potentially a row and column if the PPM does not have even dimensions.
Tests:

- Give a 4×4 PPM image, expect a 4×4 PPM image back with no information lost.
- Give a 3×3 PPM image, expect a 2×2 PPM image back.
- Run with valgrind, ensure no memory leaks.

1.2 2×2 blocks, 12 bytes on disk RGB format $\rightarrow$ RGB values floats

Summary: Divide by the denominator for each component.
Inverse: Muiply by the denominator for each component, casting the result to unsigned integers.
Input: 4 *Pnm_rgb* structs with unsigned ints for each component.
Output: 4 structs now with floats for each component.
Information Lost: None. A 32-bit float with a 23-bit mantissa can store a 16-bit integer without rounding (because the denominator may be at most $2^{16} - 1 = 65535$).
Tests:

- Give a 2×2 block of *Pnm_rgb* structs, expect a 2×2 block of structs with floats for each component.
- Give a 1×1, expect a CRE.
- Run with valgrind, ensure no memory leaks.

1.3 RGB values floats $\rightarrow$ RGB to Y, P_B, P_R floats

Summary: Converts RGB values to component video values using the given equations.
Inverse: Convert component video values to RGB values, using the given equations solved for the RGB values.
Input: A struct with floats for each component of a RGB triple.
Output: A struct with floats for each component of a component video triple.
Information Lost: Potentially. The coefficients for the conversion are rounded and most likely not exact.
Tests:

- Give a struct with floats for each component of a component video triple via an image. Pass it through the function and the inverse function. Send the results all the way back up the decompression side and diff two images.
- Give a data type that is not the expected struct, expect a CRE if NULL or RGB values not between 0 and 1.
- Run with valgrind, ensure no memory leaks.

1.4 Y, P_B, P_R floats $\rightarrow$ $\bar{P_B}, \bar{P_R}$ to 4-bit signed values

Summary: Compute $\bar{P_B}, \bar{P_R}$ and then converts the results to 4-bit signed values using $Arith40\_chroma\_of\_index()$.
Inverse: Compute the set of 4 P_B, P_R values from $\bar{P_B}, \bar{P_R}$ using $Arith40\_index\_of\_chroma()$.
Input: 4 structs with floats for each component of a component video triple.
Output: Two 4-bit singed values representing $\bar{P_B}, \bar{P_R}$ (store in unsigned val).
Information Lost: Yes. We are converting a float to a 4-bit value and are therefore scaling down the precision.
Tests:

- Give a struct with floats for each component of a RGB triple, pass it through the function and the inverse function, expect roughly the same values back.
- Give a data type that is not the expected 4 values, expect a CRE if NULL or values are not between -0.5 and 0.5. Assert that the given functions work as intended.
- Run with valgrind, ensure no memory leaks.

1.5 Y, P_B, P_R floats $\rightarrow$ Y to a, b, c, d

Summary: Preform DCT to transform the 4 Y values to a, b, c, d using the given equations.
Inverse: Invert DCT transform, converting a, b, c, d to the 4 Y values.
Input: 4 luminesce values (Y) from a 2×2 block.
Output: Floats representing a, b, c, d.
Information Lost: None.
Tests:

- Give a 4 luminesce values, pass it through the function and the inverse function, expect the exact same values back.
- Give a data type that is not the expected struct, expect a CRE if NULL or values are not between -0 and 1. Assert that the given functions work as intended.
- Run with valgrind, ensure no memory leaks.

1.6 Get b, c, d to 5-bit signed values, a to 9-bit unsigned

Summary: Convert b, c, d (-0.3 to 0.3) to 5-bit signed values (-15 to 15). Convert a to a 9-bit unsigned value (multiply by 511 and round).
Inverse: Convert the 5-bit signed values representing b, c, d to float values between -0.3 to 0.3. Convert the 9-bit unsigned value representing a back to a by dividing by 511 and stashing as a float.
Input: b, c, d values between -0.3 and 0.3. a value between -0.5 and 0.5
Output: b, c, d in 5-bit signed values, a in 9-bit signed value.
Information Lost: Yes. We are rounding a and the mapping of b, c, d should not be exactly invertible.

- Give a, b, c, d values. Make sure the outputs are within the ranges of their specified bit sizes.
- Give a, b, c, d values outside of the expected range. Make sure they map to the min/max of the range.
- Pass output through the inverse function. The results should resemble the initial inputs.
- Run with valgrind, ensure no memory leaks.

1.7 Pack $a, b, c, d, \bar{P}_B, \bar{P}_R$ to 32-bit word

Summary: Place $a, b, c, d, \bar{P}_B, \bar{P}_R$ in a 32-bit word with big-endian order. This process should involve bitpack.c.
Inverse: Read each 32-bit word, separating $a, b, c, d, \bar{P}_B, \bar{P}_R$ using our implementation of bitpack.c.
Input: $a, b, c, d, \bar{P}_B, \bar{P}_R$ in a struct.
Output: An integer representing the 32-bit word.
Information Lost: None.
Tests:

- Give $a, b, c, d, \bar{P}_B, \bar{P}_R$ values. Using bit shifting we should be able to get the exact values back from the resulting int.
- Give $a, b, c, d, \bar{P}_B, \bar{P}_R$ values. Run through function and its inverse, expect the exact values back.
- Unit test bitpack.c.
- Run with valgrind, ensure no memory leaks.

1.8 32-bit words $\rightarrow$ Compressed image

Summary: Print the compressed image to stdout using the provided header and the 32-bit words we got from the previous function. We will print the image in row-major order in raw-byte format.
Inverse: Read in the compressed image, storing it in a 2D unboxed array. Process each 32-bit word at a time.
Input: A 2D unboxed array with the 32 bit words stored as its elements.
Output: The compressed image to stdout.
Information Lost: None.
Tests:

- Give some arbitrary array with ints as its elements. Try printing it out in plain text first and make sure it has done so correctly (the raster should be character encodings of the ints).
- Both directions should work after this step, so we should be able to run the program, compressing and then decompressing an image. We will use the diff program to compare the result with the original.
- Run with valgrind, ensure no memory leaks.