import numpy as np

from collections import deque, namedtuple

from itertools import product

from math import ceil, cos, pi

from scipy.interpolate import griddata

from typing import Callable, Tuple, Union

from pathlib import Path

# JPEG markers (for our supported segments)

SOI = bytes.fromhex("FFD8") # Start of image

SOF0 = bytes.fromhex("FFC0") # Start of frame (Baseline DCT)

SOF2 = bytes.fromhex("FFC2") # Start of frame (Progressive DCT)

DHT = bytes.fromhex("FFC4") # Define Huffman table

DQT = bytes.fromhex("FFDB") # Define quantization table

DRI = bytes.fromhex("FFDD") # Define restart interval

SOS = bytes.fromhex("FFDA") # Start of scan

DNL = bytes.fromhex("FFDC") # Define number of lines

EOI = bytes.fromhex("FFD9") # End of image

# Restart markers

RST = tuple(bytes.fromhex(hex(marker)[2:]) for marker in range(0xFFD0, 0xFFD8))

# Containers for the parameters of each color component

ColorComponent = namedtuple("ColorComponent", "name order vertical\_sampling horizontal\_sampling quantization\_table\_id repeat shape")

HuffmanTable = namedtuple("HuffmanTable", "dc ac")

class JpegDecoder():

def \_\_init\_\_(self, file:Path) -> None:

# Open file

with open(file, "rb") as image:

self.raw\_file = image.read()

self.file\_size = len(self.raw\_file) # Size (in bytes) of the file

self.file\_path = file if isinstance(file, Path) else Path(file)

# Check if file is a JPEG image

# (The file needs to start with bytes 'FFD8FF')

if not self.raw\_file.startswith(SOI + b"\xFF"):

raise NotJpeg("File is not a JPEG image.")

print(f"Reading file '{file.name}' ({self.file\_size:,} bytes)")

# Handlers for the markers

self.handlers = {

DHT: self.define\_huffman\_table,

DQT: self.define\_quantization\_table,

DRI: self.define\_restart\_interval,

SOF0: self.start\_of\_frame,

SOF2: self.start\_of\_frame,

SOS: self.start\_of\_scan,

EOI: self.end\_of\_image,

}

# Initialize decoding paramenters

self.file\_header = 2 # Offset (in bytes, 0-index) from the beginning of the file

self.scan\_finished = False # If the 'end of image' marker has been reached

self.scan\_mode = None # Supported modes: 'baseline\_dct' or 'progressive\_dct'

self.image\_width = 0 # Width in pixels of the image

self.image\_height = 0 # Height in pixels of the image

self.color\_components = {} # Hold each color component and its respective paramenters

self.sample\_shape = () # Size to upsample the subsampled color components

self.huffman\_tables = {} # Hold all huffman tables

self.quantization\_tables = {} # Hold all quantization tables

self.restart\_interval = 0 # How many MCUs before each restart marker

self.image\_array = None # Store the color values for each pixel of the image

self.scan\_count = 0 # Counter for the performed scans

# Main loop to find and process the supported file segments

"""NOTE

We are sequentially looking for markers on the file. Once a recognized

marker is found, control is passed to a method to handle it. The method

then gives the control back to the main loop, which continues from where

the method stopped.

Each marker begins with 0xFF. If this byte is followed by a 0x00, then

it is escaped.

If the marker isn't recognized, then its data segment is just skipped.

我們正在按順序查找檔上的標記。一旦被認可

找到標記，將控制權傳遞給處理它的方法。方法

然後將控制權交還給主迴圈，主循環從哪裡繼續

該方法已停止。

每個標記都以 0xFF 開頭。如果此位元節後跟一個0x00，則

它被逃脫了。

如果無法識別標記，則會跳過其數據段。

"""

while not self.scan\_finished:

try:

current\_byte = self.raw\_file[self.file\_header]

except IndexError:

del self.raw\_file

break

# Whether the current byte is 0xFF

if (current\_byte == 0xFF):

# Read the next byte

my\_marker = self.raw\_file[self.file\_header : self.file\_header+2]

self.file\_header += 2

# Whether the two bytes form a marker (and isn't a restart marker)

if (my\_marker != b"\xFF\x00") and (my\_marker not in RST):

# Attempt to get the handler for the marker

my\_handler = self.handlers.get(my\_marker)

my\_size = bytes\_to\_uint(self.raw\_file[self.file\_header : self.file\_header+2]) - 2

self.file\_header += 2

if my\_handler is not None:

# If a handler was found, pass the control to it

my\_data = self.raw\_file[self.file\_header : self.file\_header+my\_size]

my\_handler(my\_data)

else:

# Otherwise, just skip the data segment

self.file\_header += my\_size

else:

# Move to the next byte if the current byte is not 0xFF

self.file\_header += 1

def start\_of\_frame(self, data:bytes) -> None:

"""Parse the information on the Start of Frame segment: scan mode,

image dimensions, color space, sampling, quantization tables used."""

data\_size = len(data)

data\_header = 0

"""NOTE

The byte structure of the segment (in order):

- 2 bytes: length of the segment

- 1 byte: sample precision

- 1 byte: image height

- 1 byte: image width

- 1 byte: amount of color components

- For each color component:

- 1 byte: ID of the component

- 4 bits: horizontal sample

- 4 bits: vertical sample

- 1 byte: ID of the quantization table used on the component

If there are 3 color components, the image is considered to be in the YCbCr

color space. The first component of the segment is Y, the next Cb, and the

last is Cr.

If there is a single component, then the image is considered to be greyscale.

段的位元組結構（按順序）：

- 2 位元組：段的長度

- 1 位元組：採樣精度

- 1 位元組：圖像高度

- 1 位元組：影像寬度

- 1 位元組：顏色分量的數量

- 對於每個顏色元件：

- 1 位元組：元件的ID

- 4 位：水平採樣

- 4 位：垂直採樣

- 1 位元組：元件上使用的量化表的ID

如果有 3 個顏色分量，則認為圖像在 YCbCr 中

色彩空間。該段的第一個分量是 Y，下一個分量是 Cb，而

最後是Cr。

如果存在單個元件，則圖像被視為灰度

"""

# Check encoding mode

# (the marker used for the segment determines the scan mode)

mode = self.raw\_file[self.file\_header-4 : self.file\_header-2]

if mode == SOF0:

self.scan\_mode = "baseline\_dct"

print("Scan mode: Sequential")

elif mode == SOF2:

self.scan\_mode = "progressive\_dct"

print("Scan mode: Progressive")

else:

raise UnsupportedJpeg("Encoding mode not supported. Only 'Baseline DCT' and 'Progressive DCT' are supported.")

# Check sample precision

# (This is the number of bits used to represent each color value of a pixel)

precision = data[data\_header]

if precision != 8:

raise UnsupportedJpeg("Unsupported color depth. Only 8-bit greyscale and 24-bit RGB are supported.")

data\_header += 1

# Get image dimensions

self.image\_height = bytes\_to\_uint(data[data\_header : data\_header+2])

data\_header += 2

"""NOTE

If the height is specified as zero here, then it means that the height value

is going to be specidied on the DNL segment after the first scan.

This is for the case when the final height is unknown when the image is

being created, for example when a scanner is generating the image.

"""

self.image\_width = bytes\_to\_uint(data[data\_header : data\_header+2])

data\_header += 2

print(f"Image dimensions: {self.image\_width} x {self.image\_height}")

if self.image\_width == 0:

raise CorruptedJpeg("Image width cannot be zero.")

# Check number of color components

components\_amount = data[data\_header]

if components\_amount not in (1, 3):

if components\_amount == 4:

raise UnsupportedJpeg("CMYK color space is not supported. Only RGB and greyscale are supported.")

else:

raise UnsupportedJpeg("Unsupported color space. Only RGB and greyscale are supported.")

data\_header += 1

if components\_amount == 3:

print("Color space: YCbCr")

else:

print("Color space: greyscale")

# Get the color components and their parameters

components = (

"Y", # Luminance

"Cb", # Blue chrominance

"Cr", # Red chrominance

)

try:

for count, component in enumerate(components, start=1):

# Get the ID of the color component

my\_id = data[data\_header]

data\_header += 1

# Get the horizontal and vertical sampling of the component

sample = data[data\_header] # This value is 8 bits long

horizontal\_sample = sample >> 4 # Get the first 4 bits of the value

vertical\_sample = sample & 0x0F # Get the last 4 bits of the value

data\_header += 1

# Get the quantization table for the component

my\_quantization\_table = data[data\_header]

data\_header += 1

# Group the parameters of the component

my\_component = ColorComponent(

name = component, # Name of the color component

order = count-1, # Order in which the component will come in the image

horizontal\_sampling = horizontal\_sample, # Amount of pixels sampled in the horizontal

vertical\_sampling = vertical\_sample, # Amount of pixels sampled in the vertical

quantization\_table\_id = my\_quantization\_table, # Quantization table selector

repeat = horizontal\_sample \* vertical\_sample, # Amount of times the component repeats during decoding

shape = (8\*horizontal\_sample, 8\*vertical\_sample), # Dimensions (in pixels) of the MCU for the component

)

# Add the component parameters to the dictionary

self.color\_components.update({my\_id: my\_component})

# Have we parsed all components?

if count == components\_amount:

break

except IndexError:

raise CorruptedJpeg("Failed to parse the start of frame.")

# Shape of the sampling area

# (these values will be used to upsample the subsampled color components)

sample\_width = max(component.shape[0] for component in self.color\_components.values())

sample\_height = max(component.shape[1] for component in self.color\_components.values())

self.sample\_shape = (sample\_width, sample\_height)

# Display the samplings

print(f"Horizontal sampling: {' x '.join(str(component.horizontal\_sampling) for component in self.color\_components.values())}")

print(f"Vertical sampling : {' x '.join(str(component.vertical\_sampling) for component in self.color\_components.values())}")

# Move the file header to the end of the data segment

self.file\_header += data\_size

def define\_huffman\_table(self, data:bytes) -> None:

"""Parse the Huffman tables from the file.

"""

data\_size = len(data)

data\_header = 0

"""NOTE

The Huffman tables are used to compress and decompress the image data.

For an explanantion of how they work in general:

https://www.youtube.com/watch?v=NjhJJYHpYsg

A JPEG image has up to 4 of Huffman tables: 2 for luminance and 2 for chrominance.

In the context of a JPEG image, these are the very basics of where those tables come from:

- When an image is encoded to JPEG, it is divided in blocks of 8x8 pixels.

- The luminance and chrominance values of those pixels are stored on an 8x8 matrix,

for each block.

- The top left value of each matrix is called DC, and all other values AC.

- When compressing the values, all the DC values are grouped and compressed

separately from the AC values.

- So the 4 tables refer to:

- Luminance DC

- Luminance AC

- Chrominance DC

- Chrominance AC

Here is a more in-depth explanation:

https://www.impulseadventure.com/photo/jpeg-huffman-coding.html

In progressive JPEG, a Huffman table might be overwritten by a new one after

a scan. Thus allowing the encoder to create tables that are optimized for the

next scan.

"""

# Get all huffman tables on the data

"""NOTE

Each Huffman table begins with the 0xFFC4 marker, followed by two bytes that

indicate the lenght (in bytes) of the section, and then 1 byte to indicate

the destination that the table refers to.

The lower nibble of the destination is the ID of the table, and the upper

nibble is if the table is for DC values (0x0) or AC values (0x1).

"""

while (data\_header < data\_size):

table\_destination = data[data\_header]

data\_header += 1

# Count how many codes of each length there are

"""NOTE

Then the next 16 bytes following the destination indicate the bit lenghts

of the elements stored on the table: first byte of this sequence is the amount

of elements that are 1 bit long, second byte the amount of elements 2 bits long,

and so on.

"""

codes\_count = {

bit\_length: count

for bit\_length, count

in zip(range(1, 17), data[data\_header : data\_header+16])

}

data\_header += 16

# Get the Huffman values (HUFFVAL)

"""NOTE

The bytes following the lengths are the values themselves in order of

increasing bit-length.

"""

huffval\_dict = {} # Dictionary that associates each code bit-length to all its respective Huffman values

for bit\_length, count in codes\_count.items():

huffval\_dict.update(

{bit\_length: data[data\_header : data\_header+count]}

)

data\_header += count

# Error checking

if (data\_header > data\_size):

# If we tried to read more bytes than what the data has, then something is wrong with the file

raise CorruptedJpeg("Failed to parse Huffman tables.")

# Build the Huffman tree

"""NOTE

A Huffman tree starts from a root node (depth 0), and each node has two

nodes bellow it (depths 1, 2, 3, ...). On a JPEG file, the tree goes up

to depth 16.

The tree contains elements that appear on the original data (before

compression). Elements are assigned to some nodes in a way that the most

common elements take shorter to navigate to than the least common. The

path used to navigate throug the tree is the codeword that is stored

pn the compressed data

To navigate through the tree: starting from the root node, if you go to

the left node you append the value of 0 to the codeword, and if you go

right you append the value of 1. When you get to a non-empty node you

stop, the value contained on the node is the value that the codeword

represents. The amount of steps you took to get to the element is the

bit-length of the codeword.

The JPEG file stores the amount of elements of each bit-length, which

we have already extracted to self.huffman\_tables. In order to build the

tree from this data:

1. Starting from the root node (depth 0), add two empty nodes (depth 1).

2. Depth 1: the empty nodes are filled from left to right with the

elements bit-length 1.

3. To the remaing empty nodes (depth 1) add 2 nodes to each (creating

depth 2)

4. Steps 2 and 3 are repeated, with depth N being filled with elements

of bit-depth N.

5. The process stops when all elements are used

Our Huffman tree will be stored in a Python dictionary. It associates the

codeword (as a string containg only '0' and '1') with its respective value.

"""

huffman\_tree = {}

code = 0

for bit\_length, values\_list in huffval\_dict.items():

code <<= 1

for huffval in values\_list:

code\_string = bin(code)[2:].rjust(bit\_length, "0")

huffman\_tree.update({code\_string: huffval})

code += 1

# Add tree to the Huffman table dictionary

self.huffman\_tables.update({table\_destination: huffman\_tree})

print(f"Parsed Huffman table - ", end="")

print(f"ID: {table\_destination & 0x0F} ({'DC' if table\_destination >> 4 == 0 else 'AC'})")

"""NOTE

Depending on the encoder, all the Huffman tables might be in the same

segment of the file, or each table can be in its own segment.

If the tables are in the same segment, then the bytes of the next table

immediatelly follows the bytes of the previous one. The order remains

the same: destination, lengths, values.

"""

# Move the file header to the end of the data segment

self.file\_header += data\_size

def define\_quantization\_table(self, data:bytes) -> None:

"""Parse the quantization table from the file.

"""

data\_size = len(data)

data\_header = 0

"""NOTE

The color values of a JPEG image are not stored directly, but rather as

a table of frequencies (DCT coefficients). The quantization table is used

to "cut" those coefficients deemed "unecessary" for how the human eye

will perceive the image.

The quantization table is a 8 x 8 matrix. The quantization is the lossy

step of the JPEG encoding. The image is broken in blocks of 8 x 8 pixels.

The DCT coefficients of the block are calculated, and then they are

divided element-wise by the quantization table. The results are rounded

to the nearest integer.

The quantization essentially decreases the resolution of the coefficients.

It assumes that the human eye cannot perceive quick variation of details

within a small area. The coefficients closer to the top left have a bigger

imact on how the eye will perceive the image, because they represent

smaller frequencies of details (smaller frequency means larger wavelength),

which the eye should perceive them better.

It is worth noting that the smaller DCT coefficients are going to become 0.

That will be the case of most, if not all, of the coefficients of the lower

right section of the block. This large amount of repeated values make

the data to be better compressed.

The smaller values of the other coefficients also aids in compression,

because those values can be represented using less bits.

IMPORTANT: The 8 x 8 block of quantized coefficients are stored in zig-zag

order, starting from the top left. This makes the zero values to be mostly

grouped together in the end of the sequence, which helps with compression.

The sequence is (from 0 to 63):

0 1 5 6 14 15 27 28

2 4 7 13 16 26 29 42

3 8 12 17 25 30 41 43

9 11 18 24 31 40 44 53

10 19 23 32 39 45 52 54

20 22 33 38 46 51 55 60

21 34 37 47 50 56 59 61

35 36 48 49 57 58 62 63

"""

# Get all quantization tables on the data

while (data\_header < data\_size):

table\_destination = data[data\_header]

data\_header += 1

"""NOTE

The quantization table segments on the file begin with the marker 0xFFDB.

The next two bytes represent the length of the segment. The next byte is

the ID of the table. And the 64 bytes afterwards are the 64 values of

the quantization table in zig-zag order.

"""

# Get the 64 values of the 8 x 8 quantization table

qt\_values = np.array([value for value in data[data\_header : data\_header+64]], dtype="int16")

try:

quantization\_table = undo\_zigzag(qt\_values)

except ValueError:

raise CorruptedJpeg("Failed to parse quantization tables.")

data\_header += 64

# Add the table to the quantization tables dictionary

self.quantization\_tables.update({table\_destination: quantization\_table})

print(f"Parsed quantization table - ID: {table\_destination}")

"""NOTE

Each quantization table can come each in its own segment on the file.

Or all tables in the same segment, one imediately after the other,

following the same byte structure (ID, then the 64 values).

"""

# Move the file header to the end of the data segment

self.file\_header += data\_size

def define\_restart\_interval(self, data:bytes) -> None:

"""Parse the restart interval value."""

self.restart\_interval = bytes\_to\_uint(data[:2])

self.file\_header += 2

print(f"Restart interval: {self.restart\_interval}")

"""NOTE

The JPEG standart allow to restart markers to be added to the encoded image data.

Those are meant to aid in error correction. The restart markers, when present,

are added each a certain amount of MCUs. This amount is specified in the

"Define Restart Interval" (DRI) segment, which starts after the 0xFFDD marker.

The restart markers are the bytes from 0xFFD0 to 0xFFD7. They are used sequentially,

and wrap back to 0xFFD0 after 0xFFD7.

It is worth noting that the MCUs encoded on the data stream are not necessarily

aligned to the byte boundary (8-bits). So after reaching the amount of MCUs specified

on the restart interval, it is necessary to move the bits header to the begining of

the next byte, by taking the modulo of the position,if the header isn't already there:

if (header\_position % 8) != 0:

header\_position += 8 - (header\_position % 8)

header\_position += 16

We also need to jump the marker itself, which is 16-bits long. So we also added 16

to the header position.

It is worth noting that the restart interval can be defined again after a scan.

The latest defined value is what counts for each scan.

"""

def start\_of\_scan(self, data:bytes) -> None:

"""Parse the information necessary to decode a segment of encoded image data,

then passes this information to the method that handles the scan mode used."""

data\_size = len(data)

data\_header = 0

"""NOTE

The structure of the Start of Scan header is (in order):

- 2 bytes: length of the segment

- 1 byte: amount of color components in the current scan

- For each color component in the scan:

- 1 byte: ID of the component

- 4 bits: ID of the Huffman table for DC values of the component

- 4 bits: ID of the Huffman table for AC values of the component

- 1 byte: Start of the spectral selection

- 1 byte: End of the spectral selection

- 4 bits: Successive approximation (high)

- 4 bits: Successive approximation (low)

Note: Spectral selection and successive approximation are relevant for

the progressive scan. They have no meaning for baseline scan.

"""

# Number of color components in the scan

components\_amount = data[data\_header]

data\_header += 1

# Get parameters of the components in the scan

my\_huffman\_tables = {}

my\_color\_components = {}

for component in range(components\_amount):

component\_id = data[data\_header] # Should match the component ID's on the 'start of frame'

data\_header += 1

# Selector for the Huffman tables

tables = data[data\_header]

data\_header += 1

dc\_table = tables >> 4 # Should match the tables ID's on the 'detect huffman table'

ac\_table = (tables & 0x0F) | 0x10

"""NOTE

The ID of the AC tables is a byte value which the first hexadecimal digit is 1.

Usually: 0x10 and 0x11 (16 and 17, in decimal).

On the other hand, the ID of the DC tables begin with hexadecimal digit 0.

Usually: 0x00 and 0x01.

"""

# Store the parameters

my\_huffman\_tables.update({component\_id: HuffmanTable(dc=dc\_table, ac=ac\_table)})

my\_color\_components.update({component\_id: self.color\_components[component\_id]})

# Get spectral selection and successive approximation

if self.scan\_mode == "progressive\_dct":

spectral\_selection\_start = data[data\_header] # Index of the first values of the data unit

spectral\_selection\_end = data[data\_header+1] # Index of the last values of the data unit

bit\_position\_high = data[data\_header+2] >> 4 # The position of the last bit sent in the previous scan

bit\_position\_low = data[data\_header+2] & 0x0F # The position of the bit sent in the current scan

"""NOTE

The data unit is formed by blocks of 8 x 8 pixels (64 values in total, indexed from 0 to 63).

The progressive scan breaks the values in different scans. And the bits of those values can

also be broken in separate scans. It is up to the encoder to decide how to split the data.

After all scans, it should be possible to reconstruct all values of all data units.

"""

data\_header += 3

# Move the file header to the begining of the entropy encoded segment

self.file\_header += data\_size

# Define number of lines

if self.image\_height == 0:

dnl\_index = self.raw\_file[self.file\_header:].find(DNL)

if dnl\_index != -1:

dnl\_index += self.file\_header

self.image\_height = bytes\_to\_uint(self.raw\_file[dnl\_index+4 : dnl\_index+6])

else:

raise CorruptedJpeg("Image height cannot be zero.")

# Dimensions of the MCU (minimum coding unit)

"""NOTE

If there is only one color component in the scan, then the MCU size is always 8 x 8.

If there is more than one color component, the MCU size is determined by the component

with the highest resolution (considering all the components of the image, not only the

components in the scan).

"""

if components\_amount > 1:

self.mcu\_width:int = 8 \* max(component.horizontal\_sampling for component in self.color\_components.values())

self.mcu\_height:int = 8 \* max(component.vertical\_sampling for component in self.color\_components.values())

self.mcu\_shape = (self.mcu\_width, self.mcu\_height)

else:

self.mcu\_width:int = 8

self.mcu\_height:int = 8

self.mcu\_shape = (8, 8)

# Amount of MCUs in the whole image (horizontal, vertical, and total)

"""NOTE

If the image dimensions are not multiples of the MCU dimensions, then

then the image right and bottom borders are padded with enough pixels

to fit a full MCU (normally by just repeating the pixels on the borders).

A JPEG decoder will just disregard those padding pixels when rendering

the image.

"""

if components\_amount > 1:

self.mcu\_count\_h = (self.image\_width // self.mcu\_width) + (0 if self.image\_width % self.mcu\_width == 0 else 1)

self.mcu\_count\_v = (self.image\_height // self.mcu\_height) + (0 if self.image\_height % self.mcu\_height == 0 else 1)

else:

component = my\_color\_components[component\_id]

sample\_ratio\_h = self.sample\_shape[0] / component.shape[0]

sample\_ratio\_v = self.sample\_shape[1] / component.shape[1]

layer\_width = self.image\_width / sample\_ratio\_h

layer\_height = self.image\_height / sample\_ratio\_v

self.mcu\_count\_h = ceil(layer\_width / self.mcu\_width)

self.mcu\_count\_v = ceil(layer\_height / self.mcu\_height)

self.mcu\_count = self.mcu\_count\_h \* self.mcu\_count\_v

# Create the image array (if one does not exist already)

if self.image\_array is None:

# 3-dimensional array to store the color values of each pixel on the image

# array(x-coordinate, y-coordinate, RBG-color)

count\_h = (self.image\_width // self.sample\_shape[0]) + (0 if self.image\_width % self.sample\_shape[0] == 0 else 1)

count\_v = (self.image\_height // self.sample\_shape[1]) + (0 if self.image\_height % self.sample\_shape[1] == 0 else 1)

self.array\_width = self.sample\_shape[0] \* count\_h

self.array\_height = self.sample\_shape[1] \* count\_v

self.array\_depth = len(self.color\_components)

self.image\_array = np.zeros(shape=(self.array\_width, self.array\_height, self.array\_depth), dtype="int16")

# Setup scan counter

if self.scan\_count == 0:

self.scan\_amount = self.raw\_file[self.file\_header:].count(SOS) + 1

print(f"Number of scans: {self.scan\_amount}")

# Begin the scan of the entropy encoded segment

if self.scan\_mode == "baseline\_dct":

self.baseline\_dct\_scan(my\_huffman\_tables, my\_color\_components)

elif self.scan\_mode == "progressive\_dct":

self.progressive\_dct\_scan(

my\_huffman\_tables,

my\_color\_components,

spectral\_selection\_start,

spectral\_selection\_end,

bit\_position\_high,

bit\_position\_low

)

else:

raise UnsupportedJpeg("Encoding mode not supported. Only 'Baseline DCT' and 'Progressive DCT' are supported.")

def bits\_generator(self) -> Callable[[int, bool], str]:

"""Returns a function that fetches the bits values in order from the raw file.

"""

bit\_queue = deque()

# This nested function "remembers" the contents of bit\_queue between different calls

def get\_bits(amount:int=1, restart:bool=False) -> str:

"""Fetches a certain amount of bits from the raw file, and moves the file header

when a new byte is reached.

"""

nonlocal bit\_queue

# Should be set to 'True' when the restart interval is reached

if restart:

bit\_queue.clear() # Discard the remaining bits

self.file\_header += 2 # Jump over the restart marker

# Fetch more bits if the queue has less than the requested amount

while amount > len(bit\_queue):

next\_byte = self.raw\_file[self.file\_header]

self.file\_header += 1

if next\_byte == 0xFF:

self.file\_header += 1 # Jump over the stuffed byte

"""NOTE

In order to prevent a sequence to be mistaken for a marker, when a 0xFF byte

appears on the image data, the encoder adds a 0x00 afterwards. This is called

'byte stuffing', and it is up to the decoder to remove it before decoding the

stream.

"""

bit\_queue.extend(

np.unpackbits(

bytearray((next\_byte,)) # Unpack the bits and add them to the end of the queue

)

)

# Return the bits sequence as a string

return "".join(str(bit\_queue.popleft()) for bit in range(amount))

# Return the nested function

return get\_bits

def baseline\_dct\_scan(self, huffman\_tables\_id:dict, my\_color\_components:dict) -> None:

"""Decode the image data from the entropy encoded segment.

The file header should be at the beginning of said segment, and at

the after the decoding the header will be moved to the end of the segment.

"""

print(f"\nScan {self.scan\_count+1} of {self.scan\_amount}")

print(f"Color components: {', '.join(component.name for component in my\_color\_components.values())}")

print(f"MCU count: {self.mcu\_count}")

print(f"Decoding MCUs and performing IDCT...")

# Function to read the bits from the file's bytes

next\_bits = self.bits\_generator()

# Function to decode the next Huffman value

def next\_huffval() -> int:

codeword = ""

huffman\_value = None

while huffman\_value is None:

codeword += next\_bits()

if len(codeword) > 16:

raise CorruptedJpeg(f"Failed to decode image ({current\_mcu}/{self.mcu\_count} MCUs decoded).")

huffman\_value = huffman\_table.get(codeword)

return huffman\_value

# Function to perform the inverse discrete cosine transform (IDCT)

idct = InverseDCT()

# Function to resize a block of color values

resize = ResizeGrid()

# Number of color components in the scan

components\_amount = len(my\_color\_components)

# Decode all MCUs in the entropy encoded data

current\_mcu = 0

previous\_dc = np.zeros(components\_amount, dtype="int16")

while (current\_mcu < self.mcu\_count):

"""NOTE

The decoding process goes through all MCUs in the image.

The MCUs are encoded sequentially on the stream, starting from the top

left of the image, then going left-to-right and then top-to-bottom.

Each MCU has 64 elements, the first being the DC value and the other 63

being the AC values.

If the image is greyscale, the color values are sequential: you get the

64 values of the Luminance MCU, and then move to the next MCU.

If the image is colored, the color values are interleaved: first you get

the values of the Luminance MCU, followed by the values of the Blue

Chrominance MCU, and then the values of the Red Chrominance MCU. Both

Red and Blue chrominances share the same Huffman trees.

However the amount of values that each MCU has depend on the chroma subsampling.

The subsampling is made by taking a number adjascent pixels, then averaging

their chromaminance values, and treating the result as a single pixel.

So each 8x8 block of chrominance values end up representing an area larger

than 8x8 pixels.

In order to compensate for that, when decoding the data stream, you first get

(consecutively) a number of 8x8 blocks of luminance values enough to

cover the area of the chrominance blocks. And only after those luminance blocks

you get one 8x8 blue chrominance block, and then one 8x8 red chrominance block.

And then the whole process repeats until all the data is scanned.

The area covered by the 8x8 luminance blocks starts from the top left, then

goes from left to right, and finally from top to bottom. This area starts

from the top left of the image, and also follows left to right and top to bottom.

"""

# (x, y) coordinates, on the image, for the current MCU

mcu\_y, mcu\_x = divmod(current\_mcu, self.mcu\_count\_h)

# Loop through all color components

for depth, (component\_id, component) in enumerate(my\_color\_components.items()):

# Quantization table of the color component

quantization\_table = self.quantization\_tables[component.quantization\_table\_id]

# Minimum coding unit (MCU) of the component

if components\_amount > 1:

my\_mcu = np.zeros(shape=component.shape, dtype="int16")

repeat = component.repeat

else:

my\_mcu = np.zeros(shape=(8, 8), dtype="int16")

repeat = 1

"""NOTE

When there is more than one color component in the scan, the components are

interleaved. And the component with the highest resolution is repeated enough

times to cover the area of the subsampled components.

For example, if you subsampled the chrominance by 2 pixels in the vertical and

in the horizontal, then a 8 x 8 block of the chrominance layer actually covers

an area of 16 x 16 in the image. While the luminance blocks still cover an

area of 8 x 8. Four of those blocks are necessary to cover an area of 16 x 16,

So in this case, while decoding we get 4 blocks of luminance followed by 1 block

of blue chrominance and then 1 block of red chrominance. All this set of blocks

is the MCU (Minimum Coding Unit).

The repeated blocks cover the MCU area starting from the top left, then moving

from left to right and from top to bottom. The MCUs themselves also cover the

image following the same pattern (left to right, then bottom to top).

"""

for block\_count in range(repeat):

# Block of 8 x 8 pixels for the color component

block = np.zeros(64, dtype="int16")

# DC value of the block

table\_id = huffman\_tables\_id[component\_id].dc

huffman\_table:dict = self.huffman\_tables[table\_id]

huffman\_value = next\_huffval()

"""NOTE

For the DC values decoding, the huffman\_value represents the bit-length

of the next DC value.

"""

dc\_value = bin\_twos\_complement(next\_bits(huffman\_value)) + previous\_dc[depth]

previous\_dc[depth] = dc\_value

block[0] = dc\_value

"""NOTE

The DC value is delta encoded in relation to the previous DC value of the

same color component.

Delta encoding is the difference between two consecutive values. So the

decoded value is just added to the previous DC value in order to find

the current value.

For the first DC value, the previous value is considered to be zero.

"""

# AC values of the block

table\_id = huffman\_tables\_id[component\_id].ac

huffman\_table:dict = self.huffman\_tables[table\_id]

index = 1

while index < 64:

huffman\_value = next\_huffval()

"""NOTE

The huffman\_value is one byte long. For the AC value decoding, the eight bits

of the huffman value are in the following format:

RRRRSSSS

Where:

RRRR represents the amount of zeroes before the next non-zero AC value

SSSS represents the bit-length of the next AC value

A huffman\_value of 0x00, however, has a different meaning: it marks the end of

the block (all remaining AC values of the block are zero).

"""

# A huffman\_value of 0 means the 'end of block' (all remaining AC values are zero)

if huffman\_value == 0x00:

break

# Amount of zeroes before the next AC value

zero\_run\_length = huffman\_value >> 4

index += zero\_run\_length

if index >= 64:

break

# Get the AC value

ac\_bit\_length = huffman\_value & 0x0F

if ac\_bit\_length > 0:

ac\_value = bin\_twos\_complement(next\_bits(ac\_bit\_length))

block[index] = ac\_value

# Go to the next AC value

index += 1

# Undo the zigzag scan and apply dequantization

block = undo\_zigzag(block) \* quantization\_table

# Apply the inverse discrete cosine transform (IDCT)

block = idct(block)

# Coordinates of the block on the current MCU

block\_y, block\_x = divmod(block\_count, component.horizontal\_sampling)

block\_y, block\_x = 8\*block\_y, 8\*block\_x

# Add the block to the MCU

my\_mcu[block\_x : block\_x+8, block\_y : block\_y+8] = block

# Upsample the block if necessary

if component.shape != self.sample\_shape:

my\_mcu = resize(my\_mcu, self.sample\_shape)

"""NOTE

Linear interpolation is performed on subsampled color components.

"""

# Add the MCU to the image

x = self.mcu\_width \* mcu\_x

y = self.mcu\_height \* mcu\_y

self.image\_array[x : x+self.mcu\_width, y : y+self.mcu\_height, component.order] = my\_mcu

# Go to the next MCU

current\_mcu += 1

print\_progress(current\_mcu, self.mcu\_count)

# Check for restart interval

if (self.restart\_interval > 0) and (current\_mcu % self.restart\_interval == 0) and (current\_mcu != self.mcu\_count):

next\_bits(amount=0, restart=True)

previous\_dc[:] = 0

"""NOTE

When the Restart Interval is reached, the previous DC values are reseted to zero

and the file header is moved to the byte boundary after the marker.

"""

self.scan\_count += 1

print\_progress(current\_mcu, self.mcu\_count, done=True)

def progressive\_dct\_scan(self,

huffman\_tables\_id:dict,

my\_color\_components:dict,

spectral\_selection\_start:int,

spectral\_selection\_end:int,

bit\_position\_high:int,

bit\_position\_low:int) -> None:

# Whether to the scan contains DC or AC values

if (spectral\_selection\_start == 0) and (spectral\_selection\_end == 0):

values = "dc"

elif (spectral\_selection\_start > 0) and (spectral\_selection\_end >= spectral\_selection\_start):

values = "ac"

else:

raise CorruptedJpeg("Progressive JPEG images cannot contain both DC and AC values in the same scan.")

"""NOTE

In sequential JPEG both DC and AC values come in the same scan, however in progressive JPEG

they must come in different scans.

"""

# Whether this is a refining scan

if bit\_position\_high == 0:

refining = False

elif (bit\_position\_high - bit\_position\_low) == 1:

refining = True

else:

raise CorruptedJpeg("Progressive JPEG images cannot contain more than 1 bit for each value on a refining scan.")

"""NOTE

The first scan of a value sends a certain amount of the value's most significant bits.

The following scans of the same value send the next bits, in order, one bit per scan.

Those scans are called "refining scans".

"""

print(f"\nScan {self.scan\_count+1} of {self.scan\_amount}")

print(f"Color components: {', '.join(component.name for component in my\_color\_components.values())}")

print(f"Spectral selection: {spectral\_selection\_start}-{spectral\_selection\_end} ({values.upper()})")

print(f"Successive approximation: {bit\_position\_high}-{bit\_position\_low} ({'refining' if refining else 'first'} scan)")

print(f"MCU count: {self.mcu\_count}")

print(f"Decoding MCUs...")

# Function to read the bits from the file's bytes

next\_bits = self.bits\_generator()

# Function to decode the next Huffman value

def next\_huffval() -> int:

codeword = ""

huffman\_value = None

while huffman\_value is None:

codeword += next\_bits()

if len(codeword) > 16:

raise CorruptedJpeg(f"Failed to decode image ({current\_mcu}/{self.mcu\_count} MCUs decoded).")

huffman\_value = huffman\_table.get(codeword)

return huffman\_value

# Beginning of scan

current\_mcu = 0

components\_amount = len(my\_color\_components)

if (values == "ac") and (components\_amount > 1):

raise CorruptedJpeg("An AC progressive scan can only have a single color component.")

"""NOTE

A DC progressive scan can have more than one color component, while an AC progressive

scan must have only one color component.

"""

# DC values scan

if values == "dc":

# First scan (DC)

"""NOTE

For the most part, the first DC scan on progressive mode is the same as on baseline mode.

The only difference is that the decoded value needs to have to undergo through a left

bit shift by the amount specified in 'bit\_position\_low', because the progressive scan

only gives this amount of the first bits of the value on the first scan.

"""

if not refining:

# Previous DC values

previous\_dc = np.zeros(components\_amount, dtype="int16")

while (current\_mcu < self.mcu\_count):

# Loop through all color components

for depth, (component\_id, component) in enumerate(my\_color\_components.items()):

# (x, y) coordinates, on the image, for the current MCU

x = (current\_mcu % self.mcu\_count\_h) \* component.shape[0]

y = (current\_mcu // self.mcu\_count\_h) \* component.shape[1]

# Minimum coding unit (MCU) of the component

if components\_amount > 1:

repeat = component.repeat

else:

repeat = 1

# Blocks of 8 x 8 pixels for the color component

for block\_count in range(repeat):

# Coordinates of the block on the current MCU

block\_y, block\_x = divmod(block\_count, component.horizontal\_sampling)

delta\_y, delta\_x = 8\*block\_y, 8\*block\_x

# First scan of the DC values

if not refining:

# DC value of the block

table\_id = huffman\_tables\_id[component\_id].dc

huffman\_table:dict = self.huffman\_tables[table\_id]

huffman\_value = next\_huffval()

# Get the DC value (partial)

dc\_value = bin\_twos\_complement(next\_bits(huffman\_value)) + previous\_dc[depth]

previous\_dc[depth] = dc\_value

"""NOTE

The DC value is delta encoded in relation to the previous DC value of the

same color component.

Delta encoding is the difference between two consecutive values. So the

decoded value is just added to the previous DC value in order to find

the current value.

For the first DC value, the previous value is considered to be zero.

"""

# Store the partial DC value on the image array

self.image\_array[x+delta\_x, y+delta\_y, component.order] = (dc\_value << bit\_position\_low)

"""NOTE

'bit\_position\_low' is the position of the last value's bit sent in the scan.

So the partial value has its bits left-shifted by this amount.

"""

# Refining scan for the DC values

else:

new\_bit = int(next\_bits())

self.image\_array[x+delta\_x, y+delta\_y, component.order] |= (new\_bit << bit\_position\_low)

"""NOTE

A refining scan of the DC values just sent the next bit of each value, in the

same order as the partial values were sent.

So the bit is just OR'ed to the existing values, in the position the bit belongs.

"""

# Go to the next MCU

current\_mcu += 1

print\_progress(current\_mcu, self.mcu\_count)

# Check for restart interval

if (self.restart\_interval > 0) and (current\_mcu % self.restart\_interval == 0) and (current\_mcu != self.mcu\_count):

next\_bits(amount=0, restart=True)

if not refining:

previous\_dc[:] = 0

"""NOTE

When the Restart Interval is reached, the previous DC values are reseted to zero

and the file header is moved to the byte boundary after the marker.

"""

# AC values scan

elif values == "ac":

"""NOTE

This scan always has one color component, and the MCU always is 8x8 pixels.

All the AC values are considered to be in one contiguous band.

The band starts with the values specified on the spectral selection for the first MCU,

then those values for the next MCU, and so on until the whole image is covered.

The order of the MCUs is: left-to-right starting from the top left, then top-to-bottom.

"""

# Spectral selection

spectral\_size = (spectral\_selection\_end + 1) - spectral\_selection\_start

# Color component

(component\_id, component), = my\_color\_components.items()

# Huffman table

table\_id = huffman\_tables\_id[component\_id].ac

huffman\_table:dict = self.huffman\_tables[table\_id]

# End of band run length

eob\_run = 0

"""NOTE

It is the amount of bands that the decoder needs to skip during decoding.

(a band is the section of a MCU, as specified in the spectral selection)

On the first scan, all values in those bands are considered to be zero.

On refining scans, the non-zero values that were skiped will be refined.

"""

# Zero run length

zero\_run = 0

"""NOTE

This is the amount of zero values to be skipped. If the run goes beyond

one band, then the run is finished.

On refining scans, the non-zero values found along the way will be

refined (those values do not decrease the zero\_run counter).

"""

# Refining function

def refine\_ac() -> None:

"""Perform the refinement of the AC values on a progressive scan

"""

nonlocal to\_refine, next\_bits, bit\_position\_low, component

# Fetch the bits that will be used to refine the AC values

# (the bits come in the same order that the values to be refined were found)

refine\_bits = next\_bits(len(to\_refine))

# Refine the AC values

ref\_index = 0

while to\_refine:

ref\_x, ref\_y = to\_refine.popleft()

new\_bit = int(refine\_bits[ref\_index], 2)

self.image\_array[ref\_x, ref\_y, component.order] |= new\_bit << bit\_position\_low

ref\_index += 1

# Queue of AC values that will be refined

to\_refine = deque()

# Decode and refine the AC values

current\_mcu = 0

while (current\_mcu < self.mcu\_count):

# Coordinates of the MCU on the image

x = (current\_mcu % self.mcu\_count\_h) \* 8

y = (current\_mcu // self.mcu\_count\_h) \* 8

# Loop through the band

index = spectral\_selection\_start

while index <= spectral\_selection\_end: # The element at the end of the band is included

# Get the next Huffman value from the encoded data

huffman\_value = next\_huffval()

run\_magnitute = huffman\_value >> 4

ac\_bits\_length = huffman\_value & 0x0F

# Determine the run length

if huffman\_value == 0:

# End of band run of 1

eob\_run = 1

break

elif huffman\_value == 0xF0:

zero\_run = 16

elif (ac\_bits\_length == 0):

# End of band run (length determined by the next bits on the data)

# (amount of bands to skip)

eob\_bits = next\_bits(run\_magnitute)

eob\_run = (1 << run\_magnitute) + int(eob\_bits, 2)

break

"""NOTE (EOB run)

If the upper lower of the Huffman value is 0x0, and the upper nibble is from 0x0 to 0xE,

then a End of Band Run (EOB run) is defined. The EOB run tells the decoder how many

bands to skip.

This amount is determined by the Huffman value and the bits following it. The upper nibble

of the Huffman value determines the amplitude (N) of the EOB run (2^N). Then the next N bits

on the data (following the Huffman value) determine the length to be added to the EOB run.

Those bits are converted from binary to decimal and added to 2^N:

EOB run = 2^N + length

"""

else:

# Amount of zero values to skip

zero\_run = run\_magnitute

"""NOTE (Zero run)

If the lower nibble of the Huffman value greater than 0x0 (except for 0xF0), then a zero run

is defined. The zero run is the amount of zero values to skip within a band. This amount is

determined directly by the value of the upper nibble of the Huffman value.

The lower nibble determines the bit-length (N) of the next non-zero AC value.

The next N bits on the data (following the Huffman value) are the next AC value,

in a binary two's complement representation.

A Huffman value of 0xF0 defines a zero run of length 16, with no AC value bits following it.

"""

# Perform the zero run

if not refining and zero\_run: # First scan

index += zero\_run

zero\_run = 0

"""NOTE

On the first scan, all AC values skipped are considered to be zero.

"""

else:

while zero\_run > 0: # Refining scan

xr, yr = zagzig[index]

current\_value = self.image\_array[x + xr, y + yr, component.order]

if current\_value == 0:

zero\_run -= 1

else:

to\_refine.append((x + xr, y + yr))

index += 1

"""NOTE

On a refining scan, only the zero AC values decrease the zero run counter.

The decoder keeps moving to the next AC value until the counter is depleted.

The non-zero values found along the way are enqueued to be refined.

"""

# Decode the next AC value

if ac\_bits\_length > 0:

ac\_bits = next\_bits(ac\_bits\_length)

ac\_value = bin\_twos\_complement(ac\_bits)

# Store the AC value on the image array

# (the zig-zag scan order is undone to find the position of the value on the image)

ac\_x, ac\_y = zagzig[index]

# In order to create a new AC value, the decoder needs to be at a zero value

# (the index is moved until a zero is found, other values along the way will be refined)

if refining:

while self.image\_array[x + ac\_x, y + ac\_y, component.order] != 0:

to\_refine.append((x + ac\_x, y + ac\_y))

index += 1

ac\_x, ac\_y = zagzig[index]

"""NOTE

On a refining scan, when the lower nibble of the Huffman value is not zero

then a new AC value is created. However the new AC value cannot be created

in the spot where there is an existing AC value. So if the decoder happens

to be at a non-zero AC value, then it moves to the next spot until a zero

is found. The non-zero values found along the way are enqueued to be refined.

"""

# Create a new ac\_value

self.image\_array[x + ac\_x, y + ac\_y, component.order] = ac\_value << bit\_position\_low

# Move to the next value

index += 1

# Refine AC values skipped by the zero run

if refining:

refine\_ac()

"""NOTE

Following the bits of the AC value (on the image data), come the bits of all

values enqueued to be refined. One bit per value, in the same order the values

were found. So if N values are going to be refined, then N bits will follow.

"""

# Move to the next band if we are at the end of a band

if index > spectral\_selection\_end:

current\_mcu += 1

if refining:

# Coordinates of the MCU on the image

x = (current\_mcu % self.mcu\_count\_h) \* 8

y = (current\_mcu // self.mcu\_count\_h) \* 8

# Perform the end of band run

if not refining: # First scan

current\_mcu += eob\_run

eob\_run = 0

"""NOTE

In the first scan, all the skipped AC values are consideded to be zero.

If the EOB run is called when a band has been partially processed, then

only the remaining values on the band are considered zero (this band

stills count for the EOB run counter).

"""

else: # Refining scan

while eob\_run > 0:

xr, yr = zagzig[index]

current\_value = self.image\_array[x + xr, y + yr, component.order]

if current\_value != 0:

to\_refine.append((x + xr, y + yr))

index += 1

if index > spectral\_selection\_end:

# Move to the next band

eob\_run -= 1

current\_mcu += 1

index = spectral\_selection\_start

# Coordinates of the MCU on the image

x = (current\_mcu % self.mcu\_count\_h) \* 8

y = (current\_mcu // self.mcu\_count\_h) \* 8

"""NOTE

On a refining scan, the non-zero values that were skipped are enqueued

to be refined. If the EOB run begins on a partially processed band, then

only the remaining values on the band are considered (this band still

decreases the EOB run counter).

"""

# Refine the AC values found during the EOB run

if refining:

refine\_ac()

"""NOTE

On the image data stream, the bits following the Huffman value that defined

an EOB run will be used to refine the non-zero AC values that were skipped

during the run. If N non-zero values were skipped, then N bits will follow

to refine them. One bit per value, in the same order the values were found.

"""

print\_progress(current\_mcu, self.mcu\_count)

# Check for restart interval

if (self.restart\_interval > 0) and (current\_mcu % self.restart\_interval == 0) and (current\_mcu != self.mcu\_count):

next\_bits(amount=0, restart=True)

"""NOTE

When a restart interval is reashed, then the decoder moves to the next byte

boundary (if not already at one), and then jumps over the restart marker (2 bytes long).

"""

print\_progress(current\_mcu, self.mcu\_count, done=True)

# Check if all scans have been performed and perform the IDCT

self.scan\_count += 1

if self.scan\_count == self.scan\_amount:

# Function to perform the inverse discrete cosine transform (IDCT)

idct = InverseDCT()

# Function to resize a block of color values

resize = ResizeGrid()

# Perform the IDCT once all scans have finished

dct\_array = self.image\_array.copy()

print("\nPerforming IDCT on each color component...")

for component in self.color\_components.values():

# Quantization table used by the component

quantization\_table = self.quantization\_tables[component.quantization\_table\_id]

# Subsampling ratio

ratio\_h = self.sample\_shape[0] // component.shape[0]

ratio\_v = self.sample\_shape[1] // component.shape[1]

# Dimensions of the MCU of the component

component\_width = self.array\_width // ratio\_h

component\_height = self.array\_height // ratio\_v

# Amount of MCUs

mcu\_count\_h = component\_width // 8

mcu\_count\_v = component\_height // 8

mcu\_count = mcu\_count\_h \* mcu\_count\_v

# Perform the inverse discrete cosine transform (IDCT)

for current\_mcu in range(mcu\_count):

# Get coordinates of the block

x1 = (current\_mcu % mcu\_count\_h) \* 8

y1 = (current\_mcu // mcu\_count\_h) \* 8

x2 = x1 + 8

y2 = y1 + 8

# Undo quantization on the block

block = dct\_array[x1 : x2, y1 : y2, component.order]

block \*= quantization\_table

# Perform IDCT on the block to get the color values

block = idct(block.reshape(8, 8))

# Upsample the block if necessary

if component.shape != self.sample\_shape:

block = resize(block, self.sample\_shape)

x1 \*= ratio\_h

y1 \*= ratio\_v

x2 \*= ratio\_h

y2 \*= ratio\_v

# Store the color values of the image array

self.image\_array[x1 : x2, y1 : y2, component.order] = block

print\_progress(current\_mcu+1, mcu\_count, header=component.name.ljust(2))

print\_progress(current\_mcu+1, mcu\_count, header=component.name.ljust(2), done=True)

def end\_of\_image(self, data:bytes) -> None:

"""Method called when the 'end of image' marker is reached.

The file parsing is finished, the image is converted to RGB and displayed."""

# Clip the image array to the image dimensions

self.image\_array = self.image\_array[0 : self.image\_width, 0 : self.image\_height, :]

"""NOTE

When the image dimensions are not multiples of the MCU size, padding pixels are

added to the right and bottom edges of the image in order to make the dimensions

multiples.

Then it is the duty of the decoder to remove those extra pixels.

"""

# Convert image from YCbCr to RGB

if (self.array\_depth == 3):

self.image\_array = YCbCr\_to\_RGB(self.image\_array)

elif (self.array\_depth == 1):

np.clip(self.image\_array, a\_min=0, a\_max=255, out=self.image\_array)

self.image\_array = self.image\_array[..., 0].astype("uint8")

self.scan\_finished = True

self.show()

del self.raw\_file

def show(self):

"""Display the decoded image in a window.

"""

try:

import tkinter as tk

from tkinter import ttk

except ModuleNotFoundError:

self.show2()

return

try:

from PIL import Image

from PIL.ImageTk import PhotoImage

except ModuleNotFoundError:

print("The Pillow module needs to be installed in order to display the rendered image.")

print("For installing: https://pillow.readthedocs.io/en/stable/installation.html")

print("\nRendering the decoded image...")

# Create the window

window = tk.Tk()

window.title(f"Decoded JPEG: {self.file\_path.name}")

try:

window.state("zoomed")

except tk.TclError:

window.state("normal")

# Horizontal and vertical scrollbars

scrollbar\_h = ttk.Scrollbar(orient = tk.HORIZONTAL)

scrollbar\_v = ttk.Scrollbar(orient = tk.VERTICAL)

# Canvas where the image will be drawn

canvas = tk.Canvas(

width = self.image\_width,

height = self.image\_height,

scrollregion = (0, 0, self.image\_width, self.image\_height),

xscrollcommand = scrollbar\_h.set,

yscrollcommand = scrollbar\_v.set,

)

scrollbar\_h["command"] = canvas.xview

scrollbar\_v["command"] = canvas.yview

# Button for saving the image

save\_button = ttk.Button(

command = self.save,

text = "Save decoded image",

padding = 1,

)

# Convert the image array to a format that Tkinter understands

my\_image = PhotoImage(

Image.fromarray(

np.swapaxes(self.image\_array, 0, 1)

)

)

# Draw the image to the canvas

canvas.create\_image(0, 0, image=my\_image, anchor="nw")

# Add the canvas and scrollbars to the window

canvas.pack()

scrollbar\_h.pack(

side = tk.BOTTOM,

fill = tk.X,

before = canvas,

)

scrollbar\_v.pack(

side = tk.RIGHT,

fill = tk.Y,

before = canvas,

)

# Add the save button to the window

save\_button.pack(

side = tk.TOP,

before = canvas,

)

# Open the window

window.mainloop()

def show2(self):

"""Display the decoded image in the default image viewer of the operating system.

"""

try:

from PIL import Image

except ModuleNotFoundError:

print("The Pillow module needs to be installed in order to display the rendered image.")

print("For installing it: https://pillow.readthedocs.io/en/stable/installation.html")

return

img = np.swapaxes(self.image\_array, 0, 1)

Image.fromarray(img).show()

def save(self) -> None:

"""Open a file dialog to save the image array as an image to the disk.

"""

from PIL import Image

from tkinter.filedialog import asksaveasfilename

# Open a file dialog for the user to provide a path

img\_path = Path(

asksaveasfilename(

defaultextension = "png",

title = "Save decoded image as...",

filetypes = (

("PNG image", "\*.png"),

("Bitmap image", "\*.bmp"),

("All files", "\*.\*")

),

initialfile = self.file\_path.stem,

initialdir = self.file\_path.parent,

)

)

# If the user has canceled, then exit the function

if img\_path == Path():

return

# Make sure that the saved image does not overwrite an existing file

count = 1

my\_stem = img\_path.stem

while img\_path.exists():

img\_path = img\_path.with\_stem(f"{my\_stem} ({count})")

count += 1

# Convert the image array to a PIL object

my\_image = Image.fromarray(np.swapaxes(self.image\_array, 0, 1))

# Save the image to disk

try:

my\_image.save(img\_path)

except ValueError:

img\_path = img\_path.with\_suffix(".png")

count = 1

my\_stem = img\_path.stem

while img\_path.exists():

img\_path = img\_path.with\_stem(f"{my\_stem} ({count})")

count += 1

my\_image.save(img\_path, format="png")

print(f"Decoded image was saved to '{img\_path}'")

class InverseDCT():

"""Perform the inverse cosine discrete transform (IDCT) on a 8 x 8 matrix of DCT coefficients.

"""

# Precalculate the constant values used on the IDCT function

# (those values are cached, being calculated only the first time a instance of the class is created)

idct\_table = np.zeros(shape=(8,8,8,8), dtype="float64")

xyuv\_coordinates = tuple(product(range(8), repeat=4)) # All 4096 combinations of 4 values from 0 to 7 (each)

xy\_coordinates = tuple(product(range(8), repeat=2)) # All 64 combinations of 2 values from 0 to 7 (each)

for x, y, u, v in xyuv\_coordinates:

"""NOTE

xyuv\_coordinates are all the combinations of 4 values, ea

"""

# Scaling factors

Cu = 2\*\*(-0.5) if u == 0 else 1.0 # Horizontal

Cv = 2\*\*(-0.5) if v == 0 else 1.0 # Vertical

# Frequency component

idct\_table[x, y, u, v] = 0.25 \* Cu \* Cv \* cos((2\*x + 1) \* pi \* u / 16) \* cos((2\*y + 1) \* pi \* v / 16)

"""NOTE

For an in-depth explanation on how the transform works, please refer to chapter 7 of this book:

https://research-solution.com/uplode/book/book-26184.pdf

Compressed Image File Formats, John Miano

"""

def \_\_call\_\_(self, block:np.ndarray) -> np.ndarray:

"""Takes a 8 x 8 array of DCT coefficients, and performs the inverse discrete

cosine transform in order to reconstruct the color values.

"""

# Array to store the results

output = np.zeros(shape=(8, 8), dtype="float64")

# Summation of the frequecies components

for x, y in self.xy\_coordinates:

output[x, y] = np.sum(block \* self.idct\_table[x, y, ...], dtype="float64")

# Return the color values

return np.round(output).astype(block.dtype) + 128

"""NOTE

128 is added to the result because, before the foward DCT transform, 128

was subtracted from the value (in order to center around zero values

from 0 to 255).

"""

class ResizeGrid():

"""Resize a grid of color values, performing linear interpolation between of those values.

"""

# Cache the meshes used for the interpolation

mesh\_cache = {}

indices\_cache = {}

def \_\_call\_\_(self, block:np.ndarray, new\_shape:Tuple[int,int]) -> np.ndarray:

"""Takes a 2-dimensional array and resizes it while performing

linear interpolation between the points.

"""

# Ratio of the resize

old\_width, old\_height = block.shape

new\_width, new\_height = new\_shape

key = ((old\_width, old\_height), (new\_width, new\_height))

# Get the interpolation mesh from the cache

new\_xy = self.mesh\_cache.get(key)

if new\_xy is None:

# If the cache misses, then calculate and cache the mesh

max\_x = old\_width - 1

max\_y = old\_height - 1

num\_points\_x = new\_width \* 1j

num\_points\_y = new\_height \* 1j

new\_x, new\_y = np.mgrid[0 : max\_x : num\_points\_x, 0 : max\_y : num\_points\_y]

new\_xy = (new\_x, new\_y)

self.mesh\_cache.update({key: new\_xy})

"""NOTE

The mesh has the same shape as the resized grid.

The mesh contains the indices of the original grid, but interpolated to the new shape.

"""

# Get, from the cache, the indices of the values on the original grid

old\_xy = self.indices\_cache.get(key[0])

if old\_xy is None:

# If the cache misses, calculate and cache the indices

xx, yy = np.indices(block.shape)

xx, yy = xx.flatten(), yy.flatten()

old\_xy = (xx, yy)

self.indices\_cache.update({key[0]: old\_xy})

# Resize the grid and perform linear interpolation

resized\_block = griddata(old\_xy, block.ravel(), new\_xy)

return np.round(resized\_block).astype(block.dtype)

# ----------------------------------------------------------------------------

# Helper functions

def bytes\_to\_uint(bytes\_obj:bytes) -> int:

"""Convert a big-endian sequence of bytes to an unsigned integer."""

return int.from\_bytes(bytes\_obj, byteorder="big", signed=False)

def bin\_twos\_complement(bits:str) -> int:

"""Convert a binary number to a signed integer using the two's complement."""

if bits == "":

return 0

elif bits[0] == "1":

return int(bits, 2)

elif bits[0] == "0":

bit\_length = len(bits)

return int(bits, 2) - (2\*\*bit\_length - 1)

else:

raise ValueError(f"'{bits}' is not a binary number.")

def undo\_zigzag(block:np.ndarray) -> np.ndarray:

"""Takes an 1D array of 64 elements and undo the zig-zag scan of the JPEG

encoding process. Returns a 2D array (8 x 8) that represents a block of pixels.

"""

return np.array(

[[block[0], block[1], block[5], block[6], block[14], block[15], block[27], block[28]],

[block[2], block[4], block[7], block[13], block[16], block[26], block[29], block[42]],

[block[3], block[8], block[12], block[17], block[25], block[30], block[41], block[43]],

[block[9], block[11], block[18], block[24], block[31], block[40], block[44], block[53]],

[block[10], block[19], block[23], block[32], block[39], block[45], block[52], block[54]],

[block[20], block[22], block[33], block[38], block[46], block[51], block[55], block[60]],

[block[21], block[34], block[37], block[47], block[50], block[56], block[59], block[61]],

[block[35], block[36], block[48], block[49], block[57], block[58], block[62], block[63]]],

dtype=block.dtype

).T # <-- transposes the array

"""NOTE

The array is transposed so the code above matches the (x, y) positions of the elements

in the 8 x 8 block of pixels:

array[x, y] = value on that pixel position

"""

# List that undoes the zig-zag ordering for a single element in a band

# (the element index is used on the list, and it returns a (x, y) tuple

# for the coordinates on the data unit)

zagzig = (

(0, 0), (1, 0), (0, 1), (0, 2), (1, 1), (2, 0), (3, 0), (2, 1),

(1, 2), (0, 3), (0, 4), (1, 3), (2, 2), (3, 1), (4, 0), (5, 0),

(4, 1), (3, 2), (2, 3), (1, 4), (0, 5), (0, 6), (1, 5), (2, 4),

(3, 3), (4, 2), (5, 1), (6, 0), (7, 0), (6, 1), (5, 2), (4, 3),

(3, 4), (2, 5), (1, 6), (0, 7), (1, 7), (2, 6), (3, 5), (4, 4),

(5, 3), (6, 2), (7, 1), (7, 2), (6, 3), (5, 4), (4, 5), (3, 6),

(2, 7), (3, 7), (4, 6), (5, 5), (6, 4), (7, 3), (7, 4), (6, 5),

(5, 6), (4, 7), (5, 7), (6, 6), (7, 5), (7, 6), (6, 7), (7, 7)

)

def YCbCr\_to\_RGB(image\_array:np.ndarray) -> np.ndarray:

"""Takes a 3-dimensional array representing an image in the YCbCr color

space, and returns an array of the image in the RGB color space:

array(width, heigth, YCbCr) -> array(width, heigth, RGB)

"""

print("\nConverting colors from YCbCr to RGB...")

Y = image\_array[..., 0].astype("float64")

Cb = image\_array[..., 1].astype("float64")

Cr = image\_array[..., 2].astype("float64")

R = Y + 1.402 \* (Cr - 128.0)

G = Y - 0.34414 \* (Cb - 128.0) - 0.71414 \* (Cr - 128.0)

B = Y + 1.772 \* (Cb - 128.0)

output = np.stack((R, G, B), axis=-1)

np.clip(output, a\_min=0.0, a\_max=255.0, out=output)

return np.round(output).astype("uint8")

def print\_progress(current:int, total:int, done:bool=False, header:str="Progress") -> None:

"""Print a progress percentage on the screen. If the process is not done yet,

the line is updated instead of moving to the next line.

"""

if not done:

print(f"{header}: {current}/{total} ({current \* 100 / total:.2f}%)", end="\r")

else:

print(f"{header}: {current}/{total} ({current \* 100 / total:.0f}%) DONE!")

# ----------------------------------------------------------------------------

# Decoder exceptions

class JpegError(Exception):

"""Parent of all other exceptions of this decoder."""

class NotJpeg(JpegError):

"""File is not a JPEG image."""

class CorruptedJpeg(JpegError):

"""Failed to parse the file headers."""

class UnsupportedJpeg(JpegError):

"""JPEG image is encoded in a way that our decoder does not support."""

# ----------------------------------------------------------------------------

# Run script

if \_\_name\_\_ == "\_\_main\_\_":

from sys import argv

try:

from tkinter.filedialog import askopenfilename

import tkinter as tk

dialog = True

except ModuleNotFoundError:

dialog = False

# Get the JPEG file path

# If a path was provided as a command line argument, then use it

if len(argv) > 1:

jpeg\_path = Path(argv[1])

command = True

else:

command = False

while True:

# Open a dialog to ask the user for a image path

if not command:

if dialog:

window = tk.Tk()

window.state("withdrawn")

print("Please choose a JPEG image...")

jpeg\_path = Path(

askopenfilename(

master = None,

title = "Decode a JPEG image",

filetypes = (

("JPEG images", "\*.jpg \*.jpeg \*.jfif \*.jpe \*.jif \*.jfi"),

("All files", "\*.\*")

)

)

)

window.destroy()

# Check if the user has chosen something

# (if the window was closed, then it returns an empty path)

if jpeg\_path == Path():

print("No file was selected.")

jpeg\_path = None

break

# If no GUI is available, then use the command prompt to ask the user for a path

else:

jpeg\_path = Path(input("JPEG path: "))

# Check if the provided path exists

if jpeg\_path.exists():

break

# Ask the user to try again if the path does not exist

else:

command = False

print(f"File '{jpeg\_path.name}' was not found on '{jpeg\_path.parent.resolve()}'")

# Ask yes or no

while True:

user\_input = input("Try again with another file? [y]es / [n]o: ").lstrip().lower()[0]

if user\_input in "yn":

break

# Break or continue the "get file path" loop

if user\_input == "y":

continue

elif user\_input == "n":

jpeg\_path = None

break

# Decode the image

if jpeg\_path is not None:

JpegDecoder(jpeg\_path)

print("Program finished. Have a nice day!")