An Evaluation of the Quite Ok Image Compression Algorithm

Westly Bouchard

# Table of Contents

[**Table of Contents**](#_majgdqb1e1ix) **1**

[**Introduction**](#_g1frn730e4qu) **3**

[Research Question](#_igtsxxc687rm) 3

[Background](#_hcsj5lczv2at) 3

[How Computers Store Data](#_3mco3p593d4b) 4

[How Images are Stored](#_z8rnsnm0l2cz) 4

[Image Compression](#_tsfkzf8ey8gf) 5

[A general Overview](#_m35wdyd7pb8x) 5

[The Industry Standard](#_9i5n3lrk2ad) 6

[QOI Image Compression](#_e3u1y7l6g81b) 7

[Methodology](#_afory0w3obv7) 8

[**A Deep Dive Into QOI**](#_ej6o87q5kwq3) **8**

[Compressing Images](#_195n3ovgjflu) 9

[Decompressing Images](#_z6uiio5ow5lv) 10

[Comparison to PNG](#_6mzvrcoa3avy) 11

[**Analysis and Comparison**](#_i1tw2yrb9645) **12**

[Data Collection](#_n09zghw4vu6b) 12

[Analysis](#_5wb3lhdsvp1p) 12

[Encode time](#_30xnye26dwwy) 13

[Decode Time](#_sq954ir5ebcz) 15

[Compression Rate](#_yphw1ctyjt65) 16

[**Conclusions**](#_3sv2z8y6y8ke) **17**

[Results](#_6n4vi5jwkqzd) 17

[Real World Implications](#_dpg9t03672jz) 18

[Successes](#_6gpgiwys6xdc) 19

[Limitations](#_q4pi7e67gyhq) 20

[**Works Cited**](#_4y0t0qikqk9u) **22**

[**Appendix A: QOI Implementation?**](#_k2pmqs9q1lyd) **23**

[**Appendix B: Data processing script for provided benchmark data**](#_voxbtyz4vewu) **23**

[**Appendix C: Code for generating graphs**](#_4jj0u1qkyggl) **26**

[Generating the Encode Time graph](#_vkunb8146wxc) 26

[Generating the Decode Time graph](#_vq7ylt435e69) 27

[Generating the Compression Rate graph](#_b3jxhyjqhart) 27

# 

# Introduction

This extended essay focuses on techniques for compressing images. Specifically, I will explore and compare two competing compression algorithms: the Portable Network Graphic standard (PNG), and the Quite Ok Image format (QOI). This essay will consist of two main sections: an exploration and comparison into the design of each algorithm, and the theory behind why certain design decisions were made, and a real world statistical comparison of the algorithms on a set of test images.

## Research Question

The research question for this extended essay is: How does the Quite Ok Image (QOI) format compare to industry standard image formats in terms of compression speed and efficiency?

## Background

Images are an integral part of our experience when using computers or browsing the internet. From setting a desktop background, to liking someone’s vacation photos on Instagram, images are just as common as text in modern day computers.

However, as image capture technology progresses, allowing for higher and higher resolution images, those images get more and more difficult to store effectively. For example, a modern smartphone camera may have a sensor that can capture 24 megapixels (24 million pixels) of image information in one photograph. This can lead to image files in excess of 48 megabytes (4.8\*10^7 bytes). At these sizes, transmitting images across the internet (especially on slow connections) can take incredible amounts of time.

Hence, methods of compressing image information to make files smaller were developed.

### How Computers Store Data

Let us first take a look, however, at how computers store information. At the lowest level, computers store all of their information as binary numbers. Everything that a computer displays, be it text, images, audio, or video, is stored as very long sequences of binary. Programmers and software developers define information **schemas**that allow translation between these numbers and whatever we are storing.

A schema outlines a set of rules or guidelines that describe how a certain type of data will be stored in binary.

Binary is a base two number system, the only two digits in binary are 1 and 0. Computers use this numbering system because it is extremely easy to implement; representing two unique states is as simple as flipping a switch to turn an electrical connection on or off.

Computers group binary data together in powers of two, however most subdivisions are beyond the scope of this essay. Importantly: a single binary digit is referred to as a **bit** while a set of eight binary digits is referred to as a **byte**.

### How Images are Stored

An image is made up of thousands or millions of individual pixels. Associated with each pixel is a red, green, and blue color value. These values represent the corresponding amount of red green or blue light a display must create to correctly produce the color of the image at that particular location. Some images also have an alpha value associated with each pixel, which denotes its transparency.

Usually, each color value is assigned one byte, allowing it to take on any value between 0 and 255. Hence, a single pixel is three bytes long for a normal image or four bytes long for an image with transparency.

All of this together means that, when a computer stores raw image information, all it is doing is storing a very long sequence of three or four byte chunks.

## Image Compression

### A general Overview

The goal of all data compression is to make the same amount of data take up less space on a computer. There are many different methods of going about this, some are specific to the type of data being compressed, while others are somewhat portable. Regardless of the type, compression schemes can be characterized as either lossless or lossy.

Lossy compression allows for parts of the original data to be discarded during compression, in ways that will not be noticeable to a user when the file is decompressed. Many image compression schemes are lossy, this is because slightly reducing the resolution or sharpness of an image is barely noticeable to the average user, so it makes sense to discard this unneeded data.

Lossless compression is the opposite of lossy compression. Where the original data can be reproduced exactly as it was before it was compressed. Lossless compression is a necessity for compressing data like text. If a text compression scheme were allowed to use lossy compression to compress data, the resulting decompressed data would be illegible.

Essentially, lossless compression relies on the idea that any file will have repeated patterns that can be collectively represented with less data than the pattern itself, resulting in a smaller file.

As mentioned previously, this essay focuses specifically on lossless image compression schemes.

### The Industry Standard

One of the most widely used lossless image compression techniques is the Portable Network Graphics (PNG) algorithm. The PNG specification was first published by the W3C in October of 1996, and was later updated in 2003.

PNG compression consists of three major steps: Pre-Filtering, LZ77 Coding, and Chunking.

First, a scan line serialization is performed, where the image is separated into rows of pixels, each row being a string of bytes [1].

Each line is then filtered. For each byte of information in the scan line, the filter uses information provided by 3 neighboring bytes in the datastream to calculate a difference, or predicted value of sorts, for the byte.

As mentioned above, data compression relies on the idea that there are many repeating patterns in the source data to take advantage of. This filtering step helps to put the data into a form where patterns can be more easily recognized, which makes the compression process more efficient.

The next step is to actually compress the filtered data. This is done with an algorithm called Deflate, which is based on a coding algorithm called LZ77. Created by Lempel and Ziv in 1977 (hence the name LZ77), the algorithm reads data into a moving window. As each new piece of data is read, the algorithm looks backwards into the window for matches. If it finds a match, instead of writing the data to the output stream, it writes a reference to the match. In most cases, this reference is much shorter than the data itself and so compression is achieved. [3]

Then, the compressed data is broken down into **chunks** (small groups of a few bytes each). This allows for the algorithm to add redundancy between pieces of compressed for the purpose of error correction. Additional chunks are also added to the beginning and end of the datastream to encode **metadata** (information about the data) like image width and height. Finally, the data is written to a file and saved.

PNG allows for high quality images to be sent over the internet much faster and more efficiently. However, as discussed above, as images get bigger and bigger, so do their compressed counterparts. Hence, the problem is not so much solved as it is postponed.

## QOI Image Compression

With that in mind we move to the Quite Ok Image format, a lossless format aimed at replacing PNG. The main goals of QOI are to improve compression ratios, speed, and complexity when compared to PNG.

A compression ratio is a metric used to describe the effectiveness of a compression algorithm. It is calculated by dividing the original size of the file that was compressed by the compressed size, which represents how much smaller the file became, and through that how well the compression algorithm performed.

Speed is a measure of how fast a compression algorithm can compress a file. It is a metric that can be affected by many different factors within the algorithm itself, like the amount of error correction that is performed or the efficiency of the compression step itself.

Complexity in the context of image compression refers to the number and length of each of the steps involved in the compression process. PNG is considered a fairly complex image compression algorithm, while QOI is considered much simpler, as will be discussed later.

## Methodology

To address the research question, I will first be examining the design of each algorithm and evaluating how certain decisions affect the efficiency of the algorithms. Secondly I will be collecting benchmark data provided by the QOI specification page, and analyzing the relative performance of the algorithms.

# A Deep Dive Into QOI

A compressed QOI image is fairly simple. It starts with a 14 byte header that provides some basic information about the image, like the colorspace and the number of channels, three for a red green and blue (RGB) image, four for an image that contains an alpha (transparency) channel (RGBA). This header also contains information about the height and width of the image.

Following the header is the compressed image data. To signal the end of the file, a unique 8 byte termination sequence is placed at the end. This sequence tells the program that is processing the compressed data that the end of the image has been reached.

## Compressing Images

When compressing an image, an encoder will write a series of chunks to the file. In the QOI scheme, there are six possible types of chunks that an encoder can write: rgba, rgb, luma, diff, run, and index.

An RGBA chunk is the largest chunk an encoder can write. It encodes a complete pixel using its red, green, blue, and alpha channel values. This takes five bytes of memory: an identification byte at the beginning which describes what type of chunk it is, and then the four bytes for each channel in the image.

An RGB chunk encodes a complete pixel using its red, green, and blue color values. The alpha channel value is assumed to be the same as for the pixel before it. Hence, this chunk takes 4 bytes of memory, one less than an RGBA chunk.

In an image without transparency, then this chunk is used to encode entire pixels and the RGBA chunk is not used at all. In images with transparency, both chunk types are used, as repeated alpha values are common, and the RGB chunk is smaller than the RGB chunk.

A Luma chunk is used when the previous pixel and the current pixel are somewhat similar. This chunk uses two bytes to encode the difference between the red, green, and blue channels in two pixels. Because the pixels are somewhat similar, less data can be used when encoding their difference rather than the two entire pixels.

A Diff chunk is used when the current pixel and previous pixel are extremely close in color. Because the pixels are so similar, this chunk can be encoded with only one byte.

A Run chunk can be written when the current and previous pixels are exactly the same. This is encoded using only one byte of memory. Run chunks are useful for large portions of one solid color, because they allow the pixels in that space to ¼ of the original space.

Finally, an Index chunk is used when a pixel matches a previously seen pixel, but when that pixel is not directly behind that current pixel. Like PNG, the QOI algorithm uses a sliding window to achieve greater compression. As QOI iterates over the raw pixel data, it calculates a **hash** (a unique mathematical representation of each pixel) and uses it to store the pixel information in a list. As QOI traverses through data, it compares incoming data to the stored hash; if it finds a match, it will refer to the previous value by its index in the array rather than writing the pixel information. This uses only one byte of data instead of the four that would be required otherwise.

When compressing an image, an encoder will look at every pixel in the original image, and decide which chunk to write to the output stream by first testing to see if a pixel fits the criteria for a one byte chunk, then the larger chunks.

## Decompressing Images

When decompressing, or decoding, an image, a program will read through the compressed data, and essentially perform the inverse operation that the encoder performed for each chunk in the datastream. As the decoder reads the compressed data, it will slowly rebuild the original image from the chunks.

## Comparison to PNG

There are a couple of important differences between PNG and QOI that allow QOI to achieve its goals of being more efficient in terms of compression ratios and speed, and simpler to implement and understand in comparison to PNG.

The first of which is that QOI can write run and index chunks with less data than PNG. The two algorithms are similar in that they use sliding windows to reference previously seen data, but QOI takes this even further. In PNG, all chunks are four bytes long, which means that when PNG writes a reference chunk, it is using four bytes of data to do so. QOI can use one byte of memory to write a reference chunk or a run chunk. This means that in situations where there are many large blocks of solid colors, or repeating patterns, QOI can achieve higher compression ratios than PNG. Additionally, QOI can write difference chunks, which allow for similar pixels to be represented with less data. These are very useful in images of the real world, as color gradients are very common. These differences in chunk structure allow for QOI to compress data more efficiently with high compression ratios than PNG can achieve.

Finally, in terms of complexity, the design of QOI allows for much more straightforward encoders and decoders to be developed. While PNG has many pre filtering steps like sub-imaging and filtering, QOI compresses the image data as it is. This removal of steps counteracts the small amount of complexity that QOI adds in its chunk structure and allows for a much more straightforward compression process. These improvements also help the algorithm run faster than PNG, meaning that QOI, in theory, succeeds in its design goals.

# Analysis and Comparison

## Data Collection

The [www.qoiformat.org](http://www.qoiformat.org) website provides a repository of benchmark images and corresponding compression data for both the QOI and PNG algorithms. This will serve as the primary data source for analysis.

As there are over 2000 data points in this dataset, a python script was used to collect and analyze the data. The full script can be found in appendix B.

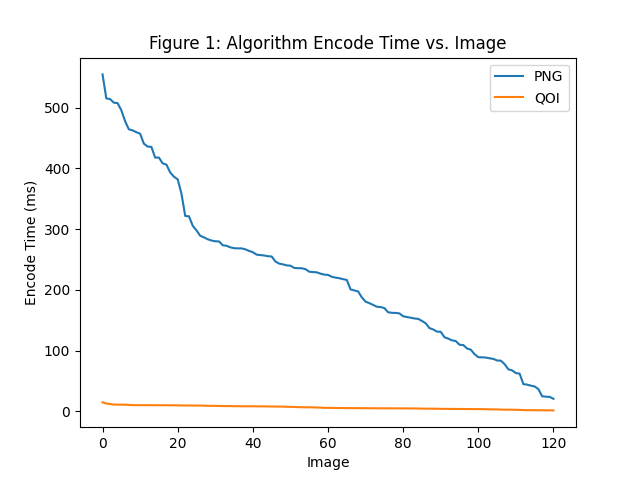
All of the visualizations seen in this essay were generated using python code written by me. The scripts for the generation can be found in appendix C.

## Analysis

There are three important metrics for assessing the effectiveness of these compression algorithms. The first is to encode time, which measures the time it takes the algorithm to compress a stream of raw image data. The second is decode time, which is the time it takes for an algorithm to take a compressed image, and extract the original image data for display. These are two of the most important metrics to look at when comparing algorithms as these happen every time you take, or even view an image. The third metric measured in this experiment is the compression ratio achieved by each algorithm. This is important to look at because storing images is expensive, and so the less space that each image can take up on a storage disk, the better.

The benchmark dataset has data for many different sizes of images, however, the graphs were generated for images of resolution 1024x1024. This size of image was chosen because it gives these algorithms the best possible opportunity to perform. A larger image size allows for longer runs to be written, as well as for more complex data references to be made, which can increase compression.

### Encode time



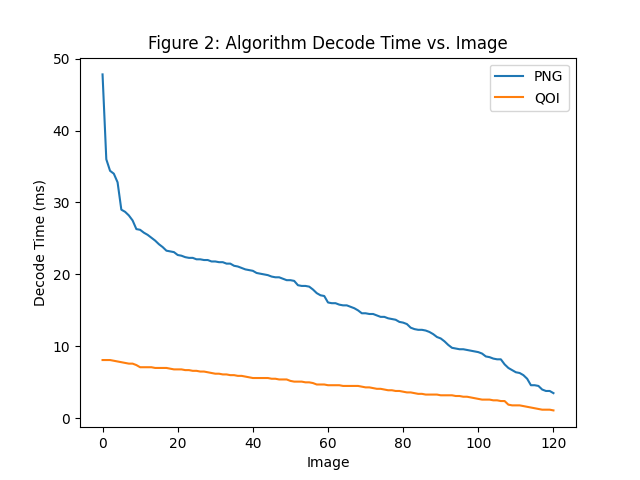
This graph shows the encode time for 120 different 1024x1024 images in the dataset. The blue line shows the PNG time for each image, while the yellow line shows the QOI time for each image. From this graph it is clear that the time it takes for QOI to encode an image is considerably shorter than that for PNG. Another interesting note is that, while there is an incredible amount of variance for the PNG encoder, the QOI encoder has a relatively low variance.

Analyzing the data further yields the following statistics:

| Algorithm | Encode Average (ms) | Encode Standard Deviation (ms) |
| --- | --- | --- |
| Quite Ok Image | 6.53 | 2.93 |
| Portable Network Graphic | 227.2 | 130.4 |

The average encode time for PNG is an order of magnitude larger than for that of QOI. Perhaps more interesting, though, is the extremely low standard deviation for the QOI encoding time. This means that QOI is extremely consistent, even when images differ greatly in their actual content. This has important implications for the application of QOI that will be discussed later. In terms of encode time, however, QOI is the clear winner against PNG.

### Decode Time

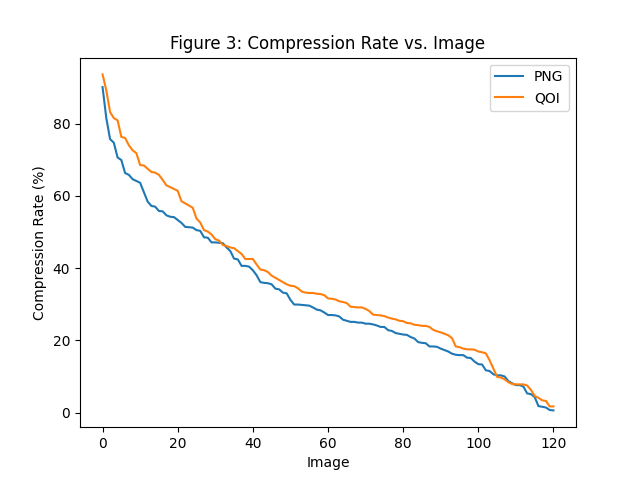


As mentioned above, decode time is another important metric to consider when evaluating image compression algorithms. This graph shows the same 120 images of resolution 1024x1024 on the x axis, but instead shows the decode time for both PNG and QOI on the y axis. From an initial impression of the graph, it can easily be seen that QOI takes the win here as well. In this graph, QOI exhibits the same characteristics that we’ve seen before, not only is the overall decompression rate lower, but it is much more consistent:

| Algorithm | Decode Average (ms) | Decode Standard Deviation (ms) |
| --- | --- | --- |
| Quite Ok Image | 4.71 | 1.89 |
| Portable Network Graphic | 16.8 | 7.69 |

These impressions from the graph are confirmed through this analysis. While numbers are lower across the board, QOI still takes a decisive victory over PNG in both average decode time and in decode time standard deviation.

### Compression Rate



This graph shows the compression ratio achieved by each algorithm for each of the 1024x1024 images analyzed. The two algorithms trade blows throughout this data, but there are some clear wins for QOI, shown by the high yellow peaks. These mean that, for that specific image, QOI was able to obtain a substantially higher compression ratio when compared to PNG.

| Algorithm | Compression Rate Average (%) | Compression Rate Standard Deviation (%) |
| --- | --- | --- |
| Quite Ok Image | 36.0 | 21.3 |
| Portable Network Graphic | 32.1 | 19.7 |

QOI edges out the win here in average compression rate, but only by a few percentage points. In this test, PNG achieved a higher, but more slightly consistent compression rate. It is difficult to declare a winner in this situation, as these results are incredibly close. Due to the fact that the standard deviations for these datasets are so close, the average rate of the QOI algorithm suggests that it could take the win.

# Conclusions

## Results

When looking at each metric individually, it could be argued that PNG does have some advantages over QOI, especially in the final compression ratio of the compressed image data.

However, when the metrics are viewed as a whole, QOI is clearly the more efficient image compression algorithm. While it only offers slight improvements in final compression ratio over PNG, it does this while drastically reducing both encode and decode time for images.

On top of this the QOI algorithm itself is much simpler, and easier to implement than PNG. This has a number of advantages. If an algorithm is easier and simpler to implement, any given implementation of it will be less prone to errors, and so can be more trustworthy and reliable.

### Real World Implications

These results are important for many reasons, the main one being efficiency. While QOI may offer a nice performance boost on a mobile phone, or personal computer when viewing images, these efficiency gains will be most prominent in the server and datacenter space.

Big tech companies spend incredible amounts of money on specialized computers called servers, which are used for processing large amounts of information, or for mass data storage.

For example, when a user loads a webpage on their phone, a request is made over the internet to a server containing all of the page content (the text, links, videos, and images). The server processes this request by retrieving the information from its storage drives, and sending it back over the internet to the device. The better compression ratio offered by QOI means that these content retrieval requests can take less time, because the server is transmitting a smaller amount of data over the internet.

Another application that could see great efficiency gains from this algorithm is cloud based applications and image editors. When a user is editing an image in their browser, and wishes to save it to their computer for other uses, the application must export (encode) the image before it can be sent to the user’s computer. For a site with millions of users, the efficiency gains in encoding with QOI could mean a smoother user experience as well as millions of dollars saved.

Finally, another implication to consider here is the improved consistency of the QOI algorithm. In many ways, this may be more important than the efficiency gains offered, as greater consistency means that computer systems can be more reliable and robust. For any tech company, an inconsistent user experience can wreak havoc on reputation and on brand confidence. QOI offers greater reliability, allowing users to place more trust into a company, improving its reputation, and its business.

## Successes

This experiment was successful in that it allowed these algorithms to be compared while controlling a number of important variables. These algorithms were tested on the same images, in the same programming language, and on the same machine.

Ensuring that each algorithm was tested on the same images is extremely important, as differences between images can lead to drastically different results across an algorithm. If an image is just one million pixels (roughly the number of pixels in a 1024x1024 image) of the exact same color, it will compress very differently than a picture of a dog, for example, which contains many different colors and lots of random variation across the image.

Both of these algorithms were tested in the C programming language according to the benchmark data. This is important as different languages can perform different optimizations when compiling code, and due to differing design paradigms across languages, can have a great impact on the performance of the resulting program.

Finally, it is important that these tests were run on the same machine because the hardware is what actually runs the programs. If the algorithms were tested across two machines, one could have slight differences in its hardware performance that would influence the perceived performance of the algorithms, skewing the data.

## Limitations

There were a number of limitations associated with this experiment.

First of all, the benchmark data used was from an external source. While this is not inherently detrimental, it could lead to inaccuracies or uncontrolled variables that could skew the results. According to the [qoiformat.org](http://qoiformat.org) website, the benchmarks were run using an Intel i7-6700k processor, but that is all of the information that is given. This experiment could be much improved with a custom QOI implementation. This would allow for more finetuned control of the algorithm, and would also allow for the collection of much more detailed data on each image. This could also benefit from running on more recent hardware, as the data provided was generated on a processor that is 7 generations old. When discussing the real world implications of a technology it is important to consider the hardware the technology would be running on, and with a custom QOI implementation, it would be possible to determine how the differing processor architectures used in the current generation of processors impacts the efficiency of these two algorithms.

Another limitation that comes from this dataset is that the largest image size tested was 1024x1024 pixels. This is an important limitation because, as image capture technology progresses, the size of images will continue to increase. For an algorithm that has the potential to be an industry standard, it is important to know whether it will be able to hold up as images get larger and larger. This experiment could be improved if images of larger resolutions were added to the dataset. Particularly, as 4k slowly becomes more prominent in the industry, performance results for QOI at 4k would give insight into how this algorithm will perform in the future. The resolution of a 4k image is 3840x2160 pixels, which is a far cry from the 1024x1024 tested in the benchmark data.

All in all, this experiment could be much improved by testing a custom implementation of the algorithm, rather than relying on externally generated data.

# Works Cited

[1] “Portable Network Graphics (PNG) Specification (Second Edition).” World Wide Web Consortium. <https://www.w3.org/TR/PNG/> (accessed August 11, 2022).

[2] C. Erdmann. “Finally Understanding PNG.” Compress-Or-Die. <https://compress-or-die.com/Understanding-PNG> (accessed August 11, 2022).

[3] D. Budhrani. “How data compression works: exploring LZ77.” Towards Data Science. <https://towardsdatascience.com/how-data-compression-works-exploring-lz77-3a2c2e06c097> (accessed August 11, 2022).

Szablewski, D. (2022, January 5). *The Quite Ok Image Format.* QOI. <https://qoiformat.org/qoi-specification.pdf>

CompuServe Incorporated. (1987, June 15). *G I F (tm) Graphics Interchange Format (tm) A standard defining a mechanism for the storage and transmission of raster-based graphics information.* World Wide Web Consortium. <https://www.w3.org/Graphics/GIF/spec-gif87.txt>

srinidhi2810. (2022, January 12). *What is Image Compression.* GeeksforGeeks. <https://www.geeksforgeeks.org/what-is-image-compression/>

# Appendix A: QOI Implementation?

# Appendix B: Data processing script for provided benchmark data

| import requests import matplotlib.pyplot as plt import numpy as np  # Helper class used to store data class Datapoint:  # Constructor takes in arrays of the various parsed data, and stores each statistic in its own variable  def \_\_init\_\_(  self,  resolution\_x,  resolution\_y,  lbpng\_data,  stbi\_data,  qoi\_data,  ):  self.resolution\_x = resolution\_x  self.resolution\_y = resolution\_y  self.lbpng\_decode\_ms = lbpng\_data[0]  self.lbpng\_encode\_ms = lbpng\_data[1]  self.lbpng\_decode\_mpps = lbpng\_data[2]  self.lbpng\_encode\_mpps = lbpng\_data[3]  self.lbpng\_size\_kb = lbpng\_data[4]  self.lbpng\_rate = lbpng\_data[5]  self.stbi\_decode\_ms = stbi\_data[0]  self.stbi\_encode\_ms = stbi\_data[1]  self.stbi\_decode\_mpps = stbi\_data[2]  self.stbi\_encode\_mpps = stbi\_data[3]  self.stbi\_size\_kb = stbi\_data[4]  self.stbi\_rate = stbi\_data[5]  self.qoi\_decode\_ms = qoi\_data[0]  self.qoi\_encode\_ms = qoi\_data[1]  self.qoi\_decode\_mpps = qoi\_data[2]  self.qoi\_encode\_mpps = qoi\_data[3]  self.qoi\_size\_kb = qoi\_data[4]  self.qoi\_rate = qoi\_data[5]   # Representation method to help with debugging, just prints out all of the data associated with the point  def \_\_repr\_\_(self):  return f'Image Resolution: {x\_res}x{y\_res}\n\  LibPng: {self.lbpng\_decode\_ms}\t {self.lbpng\_encode\_ms}\t{self.lbpng\_decode\_mpps}\t{self.lbpng\_encode\_mpps}\t{self.lbpng\_size\_kb}\t{self.lbpng\_rate}\n\  STBI: {self.stbi\_decode\_ms}\t {self.stbi\_encode\_ms}\t{self.stbi\_decode\_mpps}\t{self.stbi\_encode\_mpps}\t{self.stbi\_size\_kb}\t{self.stbi\_rate}\n\  QOI: {self.qoi\_decode\_ms}\t {self.qoi\_encode\_ms}\t{self.qoi\_decode\_mpps}\t{self.qoi\_encode\_mpps}\t{self.qoi\_size\_kb}\t{self.qoi\_rate}'  # Grabbing the data from the webpage content = requests.get("https://qoiformat.org/benchmark")  # Filtering out unnecessary stuff at the beginning content = content.text[content.text.find("Individual", 5000, 6000):]  # Split the content over the '.png' string to separate it into manageable chunks # Start at the fourth element in the list, as the first three are not data points content = content.split(".png")[3:]  # Helper function to extract the statistics from a single line of the response def extract\_data(line):  # Split the line over the spaces to separate the numbers into an array  data = line.split(" ")   # Strip the array of all of the empty elements, so we're left with just the numbers  # Here we also remove extraneous symbols like the `%` on the rate statistic, and convert the data from strings to integers  return [float(x.strip("%")) for x in list(filter(None, data))[1:]]   # Structure for holding all of the `Datapoint` objects as data is parsed data = []  # For every entry in the content we grabbed from the webpage for entry in content:  # This is more of a sanity check  # If the entry does not contain the `libpng` text, then it is not an actual datapoint and should be skipped  if "libpng" not in entry:  continue   # From the start of the resolution, grab everything until the `x` character of the resolution  x\_res = entry[5:entry.find("x")]   # Converting the resolution we just parsed into an integer  x\_res = int(x\_res)   # Logic for finding the y resolution  # Find the index of the `x` character  curr\_idx = entry.find("x")   # Iterate over the entry until we find the newline character, which signals the end of the y resolution  while entry[curr\_idx] != '\n':  curr\_idx += 1   # Grab the text that has the characters for the y resolution  y\_res = entry[entry.find("x") + 1:curr\_idx]   # Convert the y resolution from a string to an integer  y\_res = int(y\_res)   # Now we split the entry over the lines so we can parse the data points individually using the `extract\_data()` function  lines = entry.split('\n')[2:]   # Pass each line to the function separately and store the results in a variable  libpng\_data = extract\_data(lines[0])  stbi\_data = extract\_data(lines[1])  qoi\_data = extract\_data(lines[2])   # Construct a new datapoint with all of the parsed statistics and append it to our list of datapoints  data.append(Datapoint(  x\_res,  y\_res,  libpng\_data,  stbi\_data,  qoi\_data  ))  # Sanity check to make sure we have all of the data print(len(data)) |
| --- |
|  |

# Appendix C: Code for generating graphs

## Generating the Encode Time graph

| # Collecting all data for tests run on 1024x1024 images i1024 = []  # For each point in our list of points for point in data:  # If the x resolution is 1024, we add it to the list  if point.resolution\_x = 1024:  i1024.append(point)  # Extract specific statistics for each point in the list lbpng\_encode\_ms = [i.lbpng\_encode\_ms for i in i1024] qoi\_encode\_ms = [i.qoi\_encode\_ms for i in i1024]  # Plot using MatplotLib plt.plot([i for i in range(len(i1024))], lbpng\_encode\_ms, label="PNG") plt.plot([i for i in range(len(i1024))], qoi\_encode\_ms, label="QOI") plt.xlabel("Image") plt.ylabel("Encode Time (ms)") plt.legend() plt.show() |
| --- |

## Generating the Decode Time graph

| # Collecting all data for tests run on 1024x1024 images i1024 = []  # For each point in our list of points for point in data:  # If the x resolution is 1024, we add it to the list  if point.resolution\_x = 1024:  i1024.append(point)  # Extract specific statistics for each point in the list lbpng\_decode\_ms = [i.lbpng\_decode\_ms for i in i1024] qoi\_decode\_ms = [i.qoi\_decode\_ms for i in i1024]  # Plot using MatplotLib plt.plot([i for i in range(len(i1024))], lbpng\_decode\_ms, label="PNG") plt.plot([i for i in range(len(i1024))], qoi\_decode\_ms, label="QOI") plt.xlabel("Image") plt.ylabel("Decode Time (ms)") plt.legend() plt.show() |
| --- |

## Generating the Compression Rate graph

| # Collecting all data for tests run on 1024x1024 images i1024 = []  # For each point in our list of points for point in data:  # If the x resolution is 1024, we add it to the list  if point.resolution\_x = 1024:  i1024.append(point)  # Extract specific statistics for each point in the list lbpng\_rate = [i.lbpng\_rate for i in i1024] qoi\_rate = [i.qoi\_rate for i in i1024]  # Plot using MatplotLib plt.plot([i for i in range(len(i1024))], lbpng\_rate, label="PNG") plt.plot([i for i in range(len(i1024))], qoi\_rate, label="QOI") plt.xlabel("Image") plt.ylabel("Compression Rate (%)") plt.legend() plt.show() |
| --- |