Title: Comparison of the AVIF and JFIF lossy image codec.

Research Question: To what extent is AVIF a better lossy image codec than JFIF (JPEG)?

Word count: 3999

Personal code: [redacted]

Session: November 2023

Discipline: Computer science

Table of contents

Introduction

Modern displays contain thousands of pixels, each of which are capable of displaying colours by varying the amount of red, green or blue that is output. In order to store an image digitally, each pixel is a colour represented by three numbers that correspond to an amount of red, green and blue. In a 24 bit colour system, a digital representation of a single pixel may encode each colour's component with 8 successive bytes. For example,  (in R→G→B order) is: 00110011 11100011 00010000.

In practice, this digital representation of images is inefficient. The same (or similar) pixel may be encoded many times in parts of an image where there are simple correlations between pixels. Image compression is a process where we attempt to minimise the amount of data used to encode an image, whilst preserving its visual quality as much as possible. We may do this 'losslessly', without any quality degradation, or 'lossily', with quality degradation. This essay will focus on the lossy image compression algorithms of JFIF and AVIF.

Lossy image compression may have two possible benefits based on its goal: to maximise visual quality whilst minimising file size. These benefits have a number of advantages making the research question worthy of investigation. Images often make up the majority of a website's total size (Lu 2019). Reducing their size can thus improve the speed at which a website loads, which is proven to increase the number of users converting into customers on business websites (Cloudflare n.d.). Smaller image sizes may allow businesses to use less storage, saving costs in hardware and bandwidth.

Computer science was chosen for this research question since image compression is a topic that features disciplinary concepts such as: algorithms and encoding; and, decoding.

JFIF and AVIF are two codecs worthy of consideration. JFIF is considered a *de facto* technology. Consequently, it must be improved upon considerably if it is to be ousted. Many standards have been created in an attempt to supersede JFIF but have failed due to: lack of industry support — in the case of JPEG XL (Sneyers 2022); providing little benefit over JFIF — in the case of WEBP (Mozilla 2013); or, due to complicated licences, royalty and patent requirements — in the case of HEIF (Chiariglione 2018). AVIF presents a greater challenge to JFIF's omnipresent status because it is developed by an alliance of key industry players, has advanced encoding techniques that promise to challenge JFIF, and is royalty free. This makes a comparison between JFIF and AVIF worthy information for users making image compression technology decisions. Since JFIF provides the same licence benefits and has tremendous industry support, the comparison hinges on whether AVIF's encoding techniques provide enough benefit over JFIF.

To answer the research question, JFIF and AVIF will be analysed using their respective specification documents, related literature and experimented upon using two metrics: quality and encode time. Previous comparisons of JFIF and AVIF have been conducted in order to provide context for examining other algorithms, rather than as a focus, or have been conducted in a casual manner, relying on qualitative observation or a low sample size. For example, *On the hunt for the best image quality per byte* (Fronius, 2020) uses a single image and qualitative observations to form a conclusion.

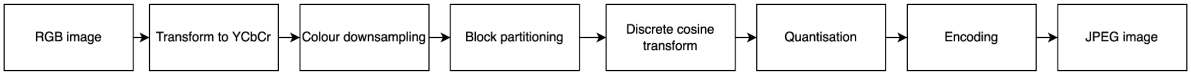
An experimental and theoretical process for evaluation was chosen because lossy image compression is an operation that produces results dependent on unpredictable input data. The two image compression algorithms considered could be made to output images that subvert theoretical conclusions by engineering the input data to suit (or not suit) a particular algorithm. So, an experimental approach where conclusions are made based upon common types of images to draw conclusions for 'most' images is critical to supplement theoretical conclusions.

Lossy image compression

JPEG File Interchange Format (JFIF) algorithm

The JPEG compression algorithm used by JFIF, released in 1991, can be explained in steps (Figure 1) since the steps are reversible and the same for each image.

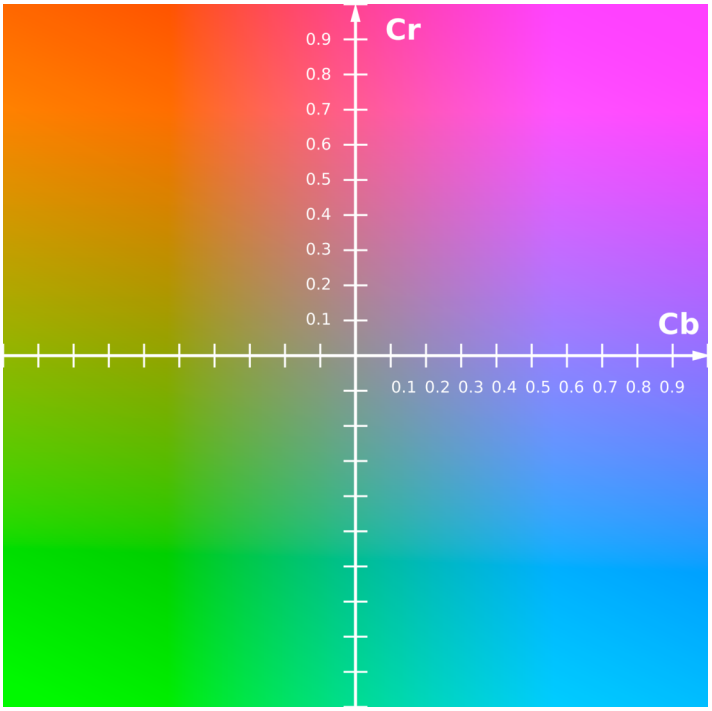
Figure 1: *Steps of JPEG compression* (Authors Own, 2023)



YCbCr conversion and colour downsampling

The first step of JPEG compression is to transform the inputted 24 bit RGB image to the 24 bit YCbCr colour space (Pound 2015). The Y in YCbCr contains the luminance (brightness) information about a colour. An image with only the Y component would be greyscale. The Cb (chroma blue) and Cr (chroma red) components contain the chrominance information as seen in Figure 2.

Figure 2: *Cb and Cr components visualised at constant luminance* (Eugster 2010)



The conversion to YCbCr and subsequent separation of chrominance and luminance allows the image to be downsampled in one component. In JFIF, the chrominance components are severely downsampled since the human eye has more regard for the luminance of an image (Pound 2015).

Block partitioning and 'Discrete Cosine Transform' (DCT)

In preparation for the discrete cosine transform, the JPEG algorithm splits an image into 8x8 pixel blocks (International Telecommunication Union 1993).

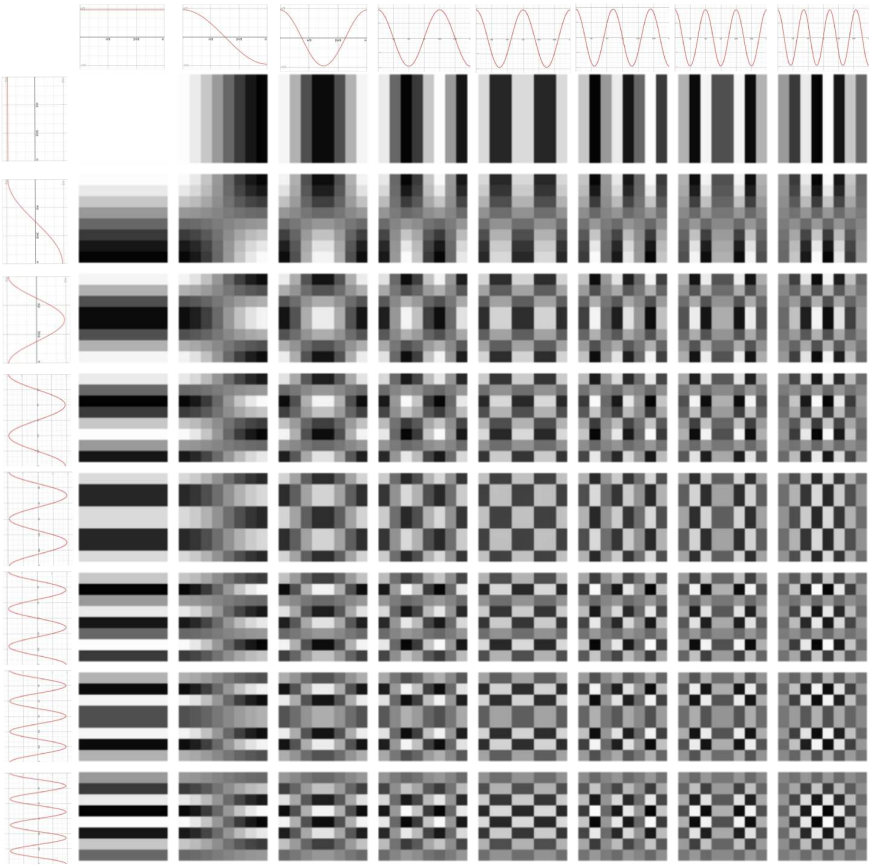
Operating on individual blocks and individual components, the goal of the DCT is to separate the highest frequency data in an image from the lowest frequency, most fundamental data (Pound 2015). The DCT does so by representing the data as a weighted sum of DCT frequencies.

To generate a single, one dimensional DCT frequency, a cosine wave is generated with a restricted domain of π. The domain is divided evenly into the number of samples in the given one dimensional data. The midpoints in each of the domains is taken and saved for use as the final DCT frequency. To generate all the DCT frequencies, this process is repeated starting with a cosine frequency of 0, incrementing to the number of samples given. The first row of DCT frequencies in Figure 3 show this process for given data with 8 samples.

For two dimensional data, such as images, the DCT frequency generation process is repeated for both dimensions. The midpoints are multiplied together to generate the final DCT frequencies. The entirety of Figure 3 shows the two dimensional DCT frequencies used in JPEG compression. Since JFIF uses 8x8 block sizes, the sample size used in the JFIF DCT is 8x8 (8 in each dimension, 64 total samples).

After generating frequencies, the algorithm determines which weighted combination of the two dimensional frequencies perfectly represents the input block (Pound 2015), outputting a grid of coefficients such as shown in Figure 4. The coefficient corresponding to the DCT frequency of 0 in both dimensions is the 'DC coefficient' and all others are 'AC coefficients'.

Figure 3: *Two-dimensional DCT frequencies with cosine graph annotations* (Author's own, 2023)



Quantisation

Quantisation takes DCT coefficients (such as those in Figure 4) as input. Then, using integer division, divides each coefficient with a corresponding value in the quantisation table that varies with the value of the quality encoder parameter. The effect of quantisation is that many of the DCT coefficients will be set to 0, resulting in quality loss. Figure 5 shows the luminance quantisation table used by libjpeg, the JFIF encoder created by the Independent JPEG Group.

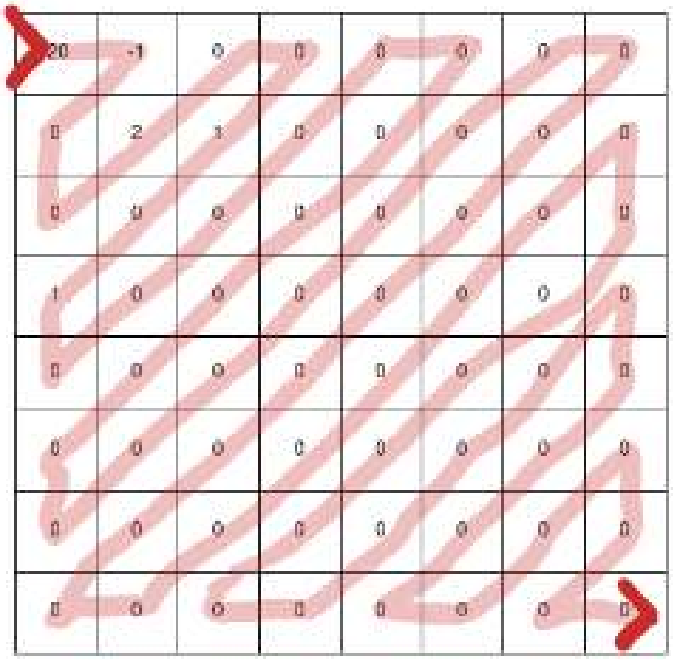
Figure 4: *Example DCT coefficients* (Authors Own, 2023)

-330	-23	-3.5	-3.2	-5.3	-0.2	-0.3	-2.3
-220	32.1	-24.5	-12.2	-3.2	-9.9	-3.5	-6.3
-58	-8.3	-7.5	-6.3	-8.5	-2.2	0.4	0.6
-23	-6.6	0.5	-4.3	0.2	-0.5	-0.3	-0.6
-6.3	-8.2	-3.6	-0.2	0.1	0.5	3.0	0.5
-5.2	-7.5	-2.5	-0.5	0.1	-0.2	-0.1	0.1
-0.3	-3.5	0.3	-0.9	0.5	0.1	0.2	0.2
-12	-29	-0.03	0.1	0.2	-0.1	0.2	1.2

Figure 5: *Independent JPEG Group luminance quantisation coefficients* (Authors Own, adapted from Richter 2023)

16	11	10	16	24	40	51	61
12	12	14	19	26	58	60	55
14	13	16	24	40	57	69	56
14	17	22	29	51	87	80	62
18	22	37	56	68	109	103	77
24	35	55	64	81	104	113	92
49	64	78	87	103	121	120	101
72	92	95	98	112	100	103	99

Figure 6: *Example output of quantization step with encoding order overlay* (Author's own, 2023)



Coding

In the coding step, the JPEG algorithm reads quantised AC coefficients in a 'zig-zag' fashion (Figure 6). This method results in many 0's being encoded in a row, allowing for more effective lossless compression using huffman coding. DC coefficients are encoded separately as a residual calculated between successive blocks.

AV1 Image Format (AVIF) algorithm

AVIF is derived from the AV1 video compression algorithm (Concolato 2022). AV1 compresses data both inter-frame (between frames) and intra-frame (within a single frame). AVIF uses only intra-frame compression (Concolato 2022).

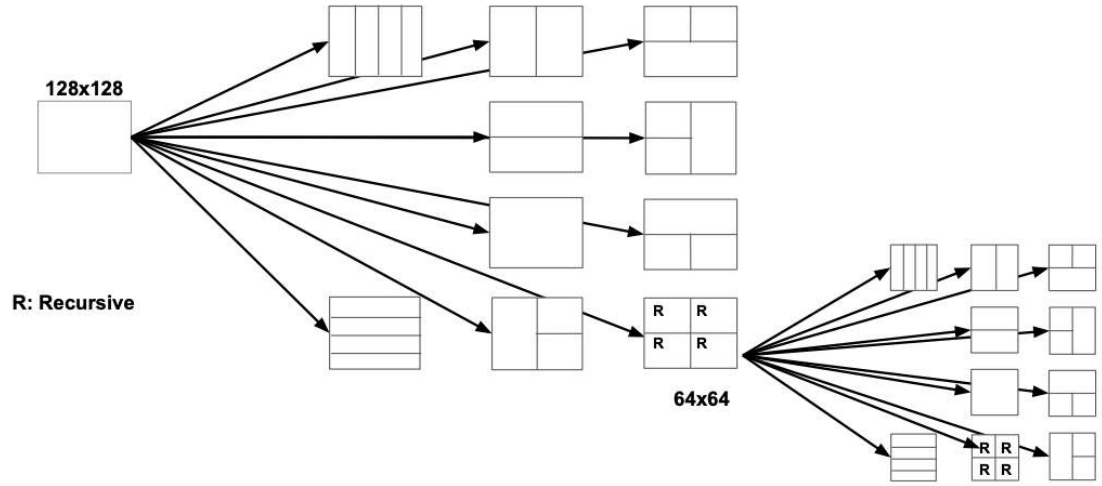
Before an image can go through the AV1 intra-frame compression algorithm, the image must be converted into the YCbCr colour space and partitioned into blocks, like JFIF. The AV1 partitioning differs in that blocks can be in different shapes and sizes (Terriberry 2019).

Block partitioning

AV1 block partitioning is more complicated than JFIF. Rather than splitting the image evenly into 8x8 blocks, AVIF divides the image so that the most detailed areas are allocated the most blocks. To do so, AV1 starts by dividing the image into the largest group of blocks called 'tiles'. Tiles do not depend on each other and can thus be compressed on separate CPU threads. Each tile contains at least one superblock of size 128x128 or 64x64 pixels (Han *et al.* 2021). The superblock is further partitioned into blocks. The encoder chooses one of the 10 shapes in Figure 7. One of these shapes is recursive, marked by R. If the recursive shape is chosen, the partitions created by the

shape are partitioned again by the same algorithm. The minimum block size is 4x4 samples (Han *et al.* 2021).

Figure 7: *The recursive block partition tree in AV1* (Han *et al.* 2021)



Intra prediction

'Intra prediction' refers to the process of creating prediction models based upon original image data on a block by block basis. This process is absent from JFIF and its inclusion has upsides and downsides. Prediction can drastically increase the effectiveness of compression in images; however, it can introduce dependencies between blocks (Han et al. 2021) that mean portions of the image cannot be shown to the user as it is rendered. In JFIF, there are no dependencies between blocks, so the JPEG can be shown progressively, creating the illusion of a faster loading image. This speed improvement could have a greater effect on user experience than reduced file size. However, AVIF's prediction step may be favoured in circumstances where bandwidth remains a greater concern than user experience. AVIF's intra-prediction works best with simple spatial correlations

like colour gradients. By breaking complex parts of an image into smaller blocks, variable block sizes assists the prediction step.

To implement intra-prediction, AVIF uses a number of different algorithms called 'predictors' that usually work on a single colour component (Han *et al.* 2021).

Table 1: Different AVIF intra-predictors and their description

Predictor	**Brief description**	**Theoretical use case**
Directional intra-predictor	The neighbouring top and left pixels are encoded with an angle along which to extrapolate the colour component for each pixel in the block (Montgomery 2018). A 'DC' mode is included where the average of the neighbouring pixels is taken and set to all the block pixels (Terriberry 2019).	Sharp edges or flat areas.
Non-directional smooth intra-predictor	The neighbouring top and left pixels are encoded along with a mode: vertical, horizontal and general (Correa *et al.* 2019). Neighbouring pixels in the same row or column of the current pixel and the top right or bottom left neighbouring pixel depending on the mode are multiplied by weightings inherent to the AVIF encoder/decoder and added together.	Gradients and blooms (Correa *et al.* 2019)
Recursive intra-predictor	Blocks are split into 4x2 sub-blocks, each with top and left neighbouring pixel dependencies (Correa *et al.* 2019). An array of neighbouring pixel coefficient arrays for each pixel is inherent to the encoder/decoder. 5 modes are included with different coefficients. The coefficient arrays include a coefficient for each neighbouring pixel; these values are multiplied by the neighbouring pixel and summed to create the value for the pixel.	Luminance information (Correa *et al.* 2019)
Chroma from luma predictor	A chrominance only predictor that performs linear regression on the luminance and chrominance	Chroma values that correlate with

	values to create chroma from luma model that is encoded for the block (Correa *et al.* 2019).	luminance values
Intra block copy	Specifies a block other than itself to copy.	Screenshots, digital images with large repeating areas
Colour palette	A unique luma & chroma predictor that chooses 2-8 unique colours with high representation in the block and each pixel is assigned the colour that it best fits with (Correa *et al.* 2019).	Screenshots, digital images with low amounts of unique colours.

Transform coding

The transform coding step is similar to the DCT step of JFIF. However, where JFIF's DCT is given the raw pixel values of blocks, AVIF's various transformation modes (including DCT) are given the residual between original and predicted pixels. Similar to JFIF, the transform coefficients are quantised and losslessly compressed. AVIF uses arithmetic rather than Huffman coding. In decoding, the output of the inverse transformation step is added to the predicted pixels to produce a result.

Filtering

Filtering is the process of improving the image's perceived quality by removing artefacts introduced by the prediction and transform coding process. It is absent from JFIF. There are a number of filters tailored to different artefacts detailed in table 2.

Table 2: Different AVIF filters

Filter	Brief description	Theoretical use case
Deblocking filter	Block based compression methods often suffer from blocking artefacts, where the borders of blocks are clearly visible. The deblocking filter essentially blurs edges together to reduce the effect (Gurtovaya & Barnov 2021).	When the amount of compression applied is high
Constrained directional enhancement filter	The constrained directional enhancement filter aims to reduce ringing artefacts caused by the directional intra-predictor (Montgomery 2018).	When the amount of compression applied is high
Film grain synthesis	Grain and other noisy details can often be removed by the compression process (Han et al. 2021). Film grain synthesis adds slight random variations to pixels to reintroduce grain and noisy details. Film grain synthesis is not enabled automatically, only with an encoder flag.	For images with a high amount of noise

Investigation

AVIF and JFIF were experimentally compared using two metrics: quality; and, encode time. Decode time was considered as a metric, but ruled out due to being insignificant relative to the time to download an image over the internet (Sneyers 2022). Quality was included due to its significance as a major goal of compression algorithms and for its benefits outlined in the introduction (bandwidth costs, storage costs, decreased loading times). Encode time was deemed to be significant because it can accumulate significant costs when an image Content Delivery Network (CDN) is used. For example, the popular 'fastly' CDN charges $0.000045 per vCPU second (Fastly n.d.). This amount could add to be significant over many images and seconds of encoding time.

Quality

To evaluate quality, a primary investigation was undertaken. Eleven uncompressed photos were taken to represent a range of different uses of image compression. These included a selfie (to represent social media sites), webcam photo (video conferencing), sunset (social media, various), landscape (social media, various), 3d render (product websites), cityscape (social media, various), portrait (profile photo, various), and close up and extreme close up (social media, various). Photos were taken on a 'Google Pixel 4XL' smartphone using the RAW capture mode that does not use any lossy compression; the 3d render was created using blender; and the screenshot and webcam photo were taken on a 13-inch Macbook Air from 2020. The photos were developed using the 'Snapseed' app and exported as lossless PNGs to avoid lossy compression.

To evaluate the effectiveness of image compression algorithms, either quality or file size must be constant. It was decided that file size would be made constant. To achieve this in an experiment, a Python script was developed to compress images to a given size by varying only the quality encoder parameter of both AVIF and JFIF. Once the image was compressed to size, the script would decode the image into a bitmap and analyse it using the Structural Similarity Index (SSIM) algorithm. Various libraries were used, detailed in table 3. The images were compressed at 50,000 bytes, 100,000 bytes, 150,000 bytes, 200,000 bytes, 250,000 bytes and 300,000 bytes.

Table 3: *Table of libraries used in quality experiment*

Library used	Significance
Pillow (Clark *et al.* 2023)	To import and encode images from the file system. To encode JFIF images, Pillow uses the Independent JPEG Group software, libjpeg. AVIF is not natively supported.
pillow-avif-plugin (Dintino 2023)	Add AVIF support to Pillow using libaom, the AVIF encoder from the creators of AVIF, the Alliance for Open Media (Dintino 2023)
PyTorch (Paszke *et al.* 2019)	A dependency of PIQ and used to format images correctly for use in PIQ.
PyTorch Image Quality (Kastryulin *et al.* 2023)	To calculate image quality using a number of different metrics.
scikit-image (Schönberger *et al.* 2023)	To read images from the file system in a format for PyTorch.

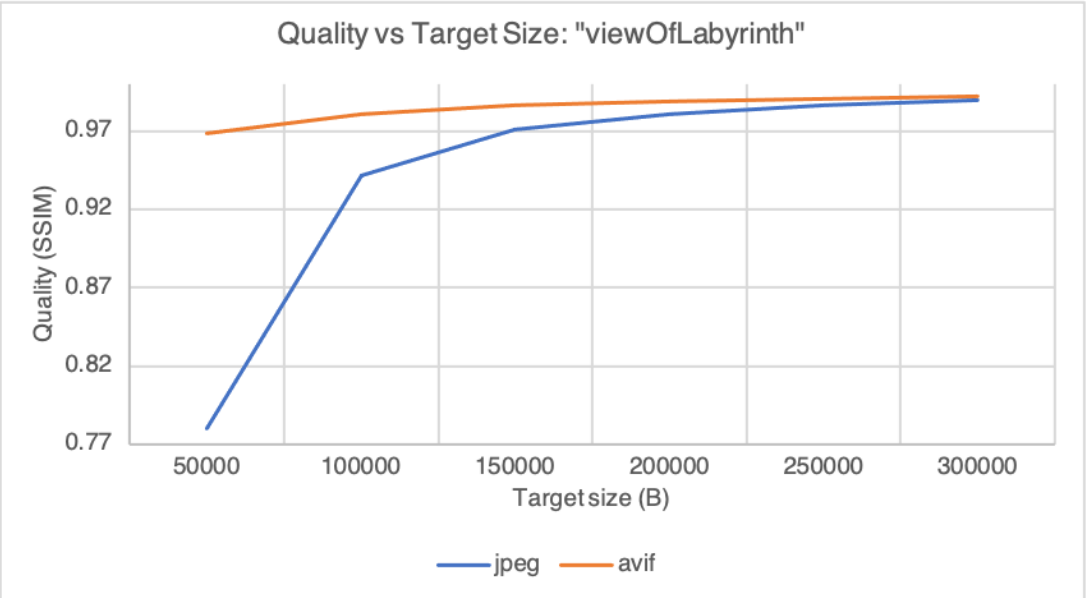
Measuring image quality is a difficult task since it requires a deep understanding of human vision in order to correlate well with human perceptions. The Structural Similarity Index (SSIM) is the *de facto* standard for image quality analysis (Ding *et al.* 2020). SSIM supersedes Mean Squared Error (MSE), the simplest way to calculate the quality of a transformed image with reference to the original. According to Ding *et al.*, SSIM is different to Mean Squared Error (MSE) in that, instead of treating all variance from the reference as equal error, SSIM normalises the luminance and contrast of the image so that a globally changed luminance or contrast does not affect the quality index in the same way as localised changes in the image.

Results and analysis

Landscapes and longshots

Figure 8: *'viewOfLabyrinth' image. Losslessly compressed.*

Figure 9: *Graph of quality vs target size for 'viewOfLabyrinth' image.*



In the first category of images, landscapes and longshots (Appendices E, G, M, U, W) AVIF consistently compressed images to a higher quality than JFIF due to its variable block sizes and prediction modes. In 'viewOfLabyrinth' (Figure 8, Appendix W), the quality at 50,000 bytes was 0.77 for JFIF, lower than the average for all JFIF images of 0.93. This indicates that 'viewOfLabyrinth' was an image that was difficult for JFIF to compress. 'viewOfLabyrinth' had a large region of simple spatial correlation (the top region) and a large region of complex, random spatial correlation (the bottom region). AVIF's variable block size allowed it to dedicate more blocks to regions of low spatial correlation, allowing it to compress this image more effectively. This is in comparison to JFIF, whose uniform block size resulted in an excessive use of bytes to encode the correlationally simple sky. In addition, AVIF's prediction modes allow it to represent gradients more efficiently and accurately, such as those in the sky seen in Figure 10. This pattern of large regions of simple spatial correlation allowing more bytes to be allocated toward regions with higher structural complexity ensured AVIF's advantage in 'busStop' (Appendix E) and 'farmFromBridge' (Appendix G). In 'market' (Appendix M), the image had much smaller regions of simple spatial correlation, making it an exception to this rule and resulting in the lowest average quality across all target sizes for AVIF of any image tested. However, AVIF prevailed in its lead in this image likely due to the prediction step that allows it to encode general correlations within blocks with less bytes than would be used solely by a transform encoder (such as JFIF). This allows AVIF to use the transform encoder to represent details in a block that are not part of the more general spatial correlations in the block, 'spending' bytes encoding detail rather than broad spatial correlation. It should be noted that AVIF's prediction steps and variable block sizes can have the opposite effect in images where blocks are concentrated in one area and not in others. This effect is visible in

'busStop' (Appendix E), where in Figure 11 AVIF's image appears smooth whilst JFIF's blocking artefacts restore the original texture in the image. Despite this, AVIF compressed images to a higher quality in this category due to its variable block sizes and prediction step.

Figure 10: *'viewOfLabyrinth' compressed with JFIF, AVIF at target size 100,000 bytes. Annotated.*

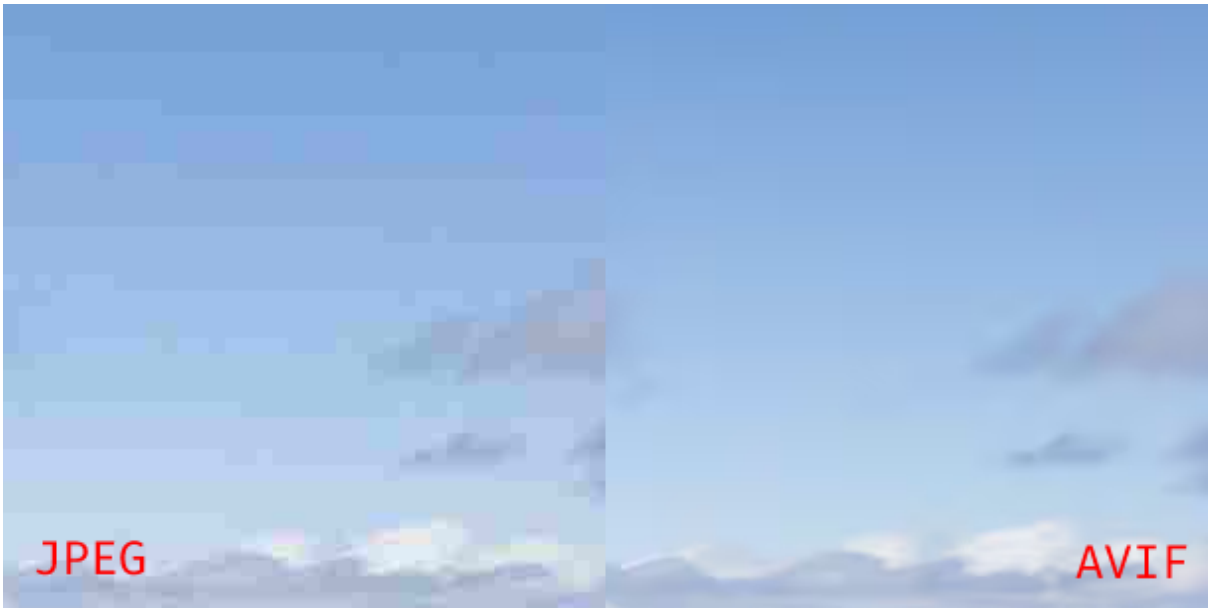


Figure 11: *'busStop' compressed to 100,000 bytes with JFIF (left) and AVIF (middle).*

Medium shot

Figure 12: *'person' image. Losslessly compressed.*

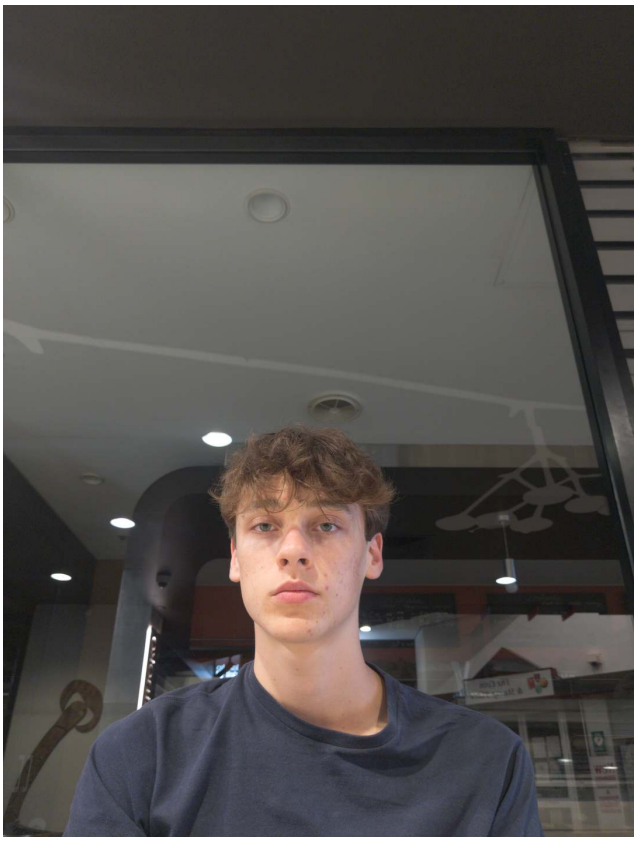
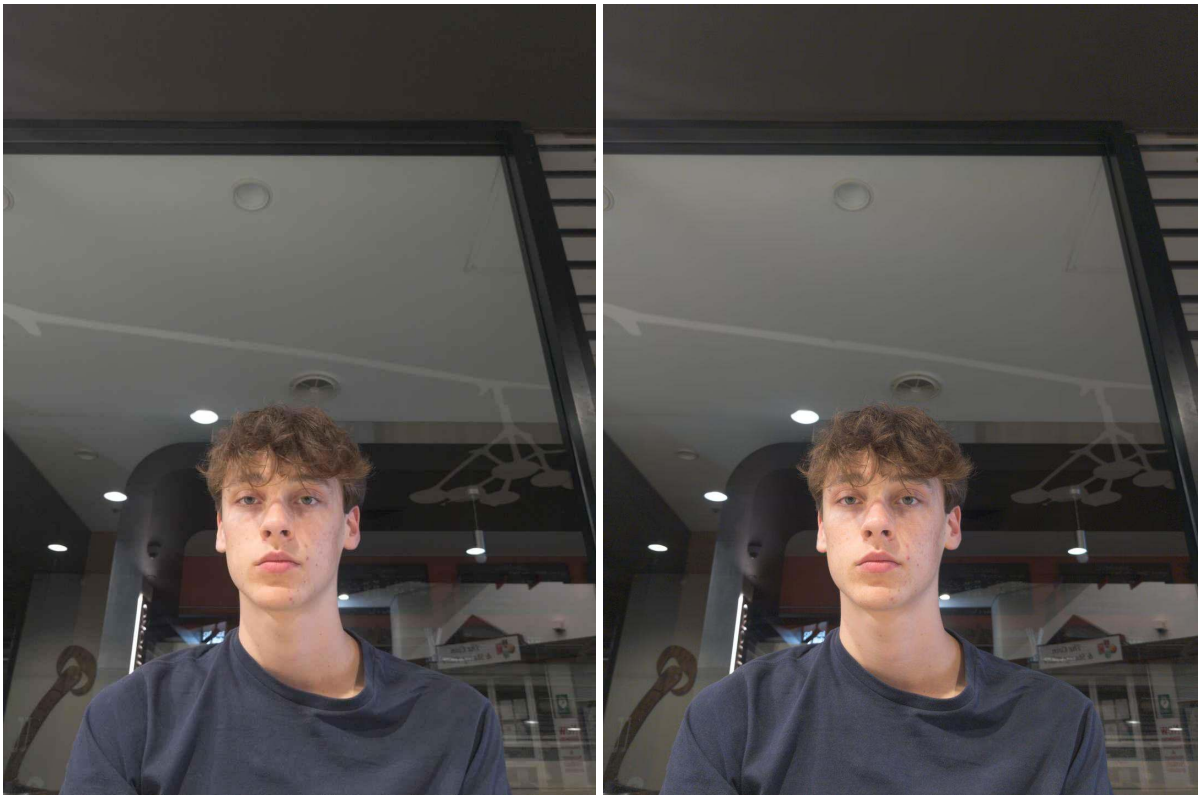


Figure 13: *Graph of quality vs target size for 'person image'.*

'person' (Appendix O) was compressed more effectively by AVIF than JFIF, but both algorithms compressed this image well. The simple spatial correlations in the image were compressed efficiently by AVIF's large block sizes and predictors, as seen in Figure 15. JFIF's compression struggled with these simpler spatial correlations, creating a banding effect as seen on the ceiling in Figure 14. Areas of complex correlation were compressed well by both AVIF and JFIF, indicating the suitability of transform coding algorithms for images with complex spatial correlation. The lack of banding in AVIF's image and its greater SSIM score over all target sizes indicates that it remained the 'better' compression algorithm in terms of quality for this image.

Figure 14 (left) and 15 (right): *'person' compressed to 50,000 bytes with JFIF (left) and AVIF (right)*



Close ups

Figure 16: *'ipodClose' image. Losslessly compressed.*

Figure 17: *Graph of quality vs target size for 'ipodClose' image.*

The two close ups (appendices I, K) tested were compressed more effectively by AVIF, but JFIF was not as far behind as in other categories. These two images had large areas of high spatial correlation with 'noisy' (slight random variations in colour from pixel to pixel) regions on the metal of the iPod. These noisy regions are difficult to encode efficiently due to their random nature. Figure 18 shows a portion of a noisy region compressed with AVIF and JFIF. In JFIF, blocking artefacts and individual DCT frequencies can be observed whilst the noise in AVIF appears to be more authentic. This could be explained by: variable block size, which makes blocking artefacts less obvious and increases the intricacy of predictions and transforms; a prediction step in combination with a transform step, which makes DCT frequencies less obvious due to their combination with some form of prediction; and/or a deblocking filter that smooths over remaining artefacts. Film grain synthesis could improve the quality of the noise in this image further, but it was not enabled in the encoder. AVIF's features helped it to more effectively compress noise than JFIF in these images.

Figure 18: *'ipodClose' compressed to 50,000 bytes using JFIF (left) and AVIF (right)*

Screenshot and 3d model

Figure 19: *'phone' image. Losslessly compressed.*

Figure 20: *Graph of quality vs target size for 'phone' image.*

Figure 21: *'screenshot' image. Losslessly compressed.*

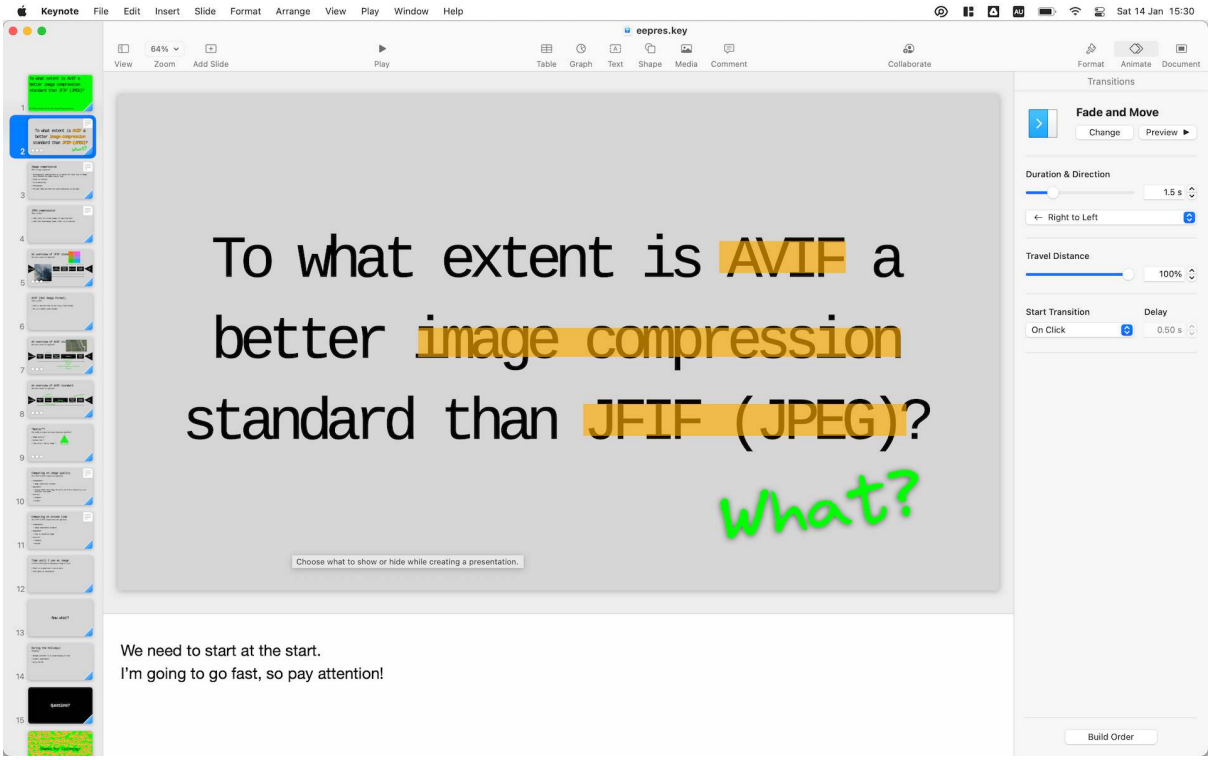
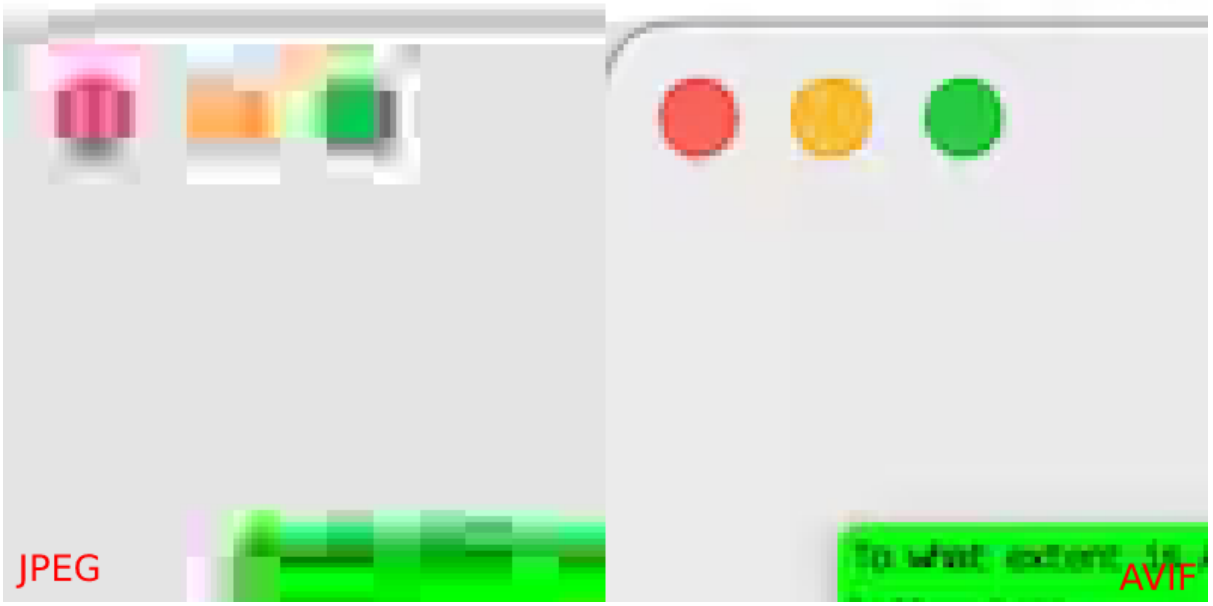


Figure 22: *Graph of quality vs target size for 'screenshot' image.*

The screenshot and 3d model (appendices S and Q) were compressed very effectively by AVIF and JFIF, however, at low target sizes AVIF is more effective. The low structural complexity of these images and large areas of single colours allowed JFIF and AVIF to rely on DC coefficients and prediction modes. Few compression artefacts were visible in 'phone' for both AVIF and JFIF (Appendix Q). However, the SSIM index for this image reported that JFIF consistently compressed to a lower quality. In 'screenshot' (Appendix S), AVIF's intra block copy and colour palette modes excelled in compressing the image. This was the case even at the lowest target size where it was compressed to a SSIM index of 0.9982 where the average for this target size for AVIF was 0.9768. AVIF's variable block sizes allowed it to effectively compress smaller details in an image, such as in Figure 23, where the JFIF block size was too large to compress the three dots well. AVIF more effectively compressed this category of images.

Figure 23: *'screenshot' compressed to 50,000 bytes with JFIF (left) and AVIF (right).*



Webcam

Figure 24: *'webcam' image. Losslessly compressed.*

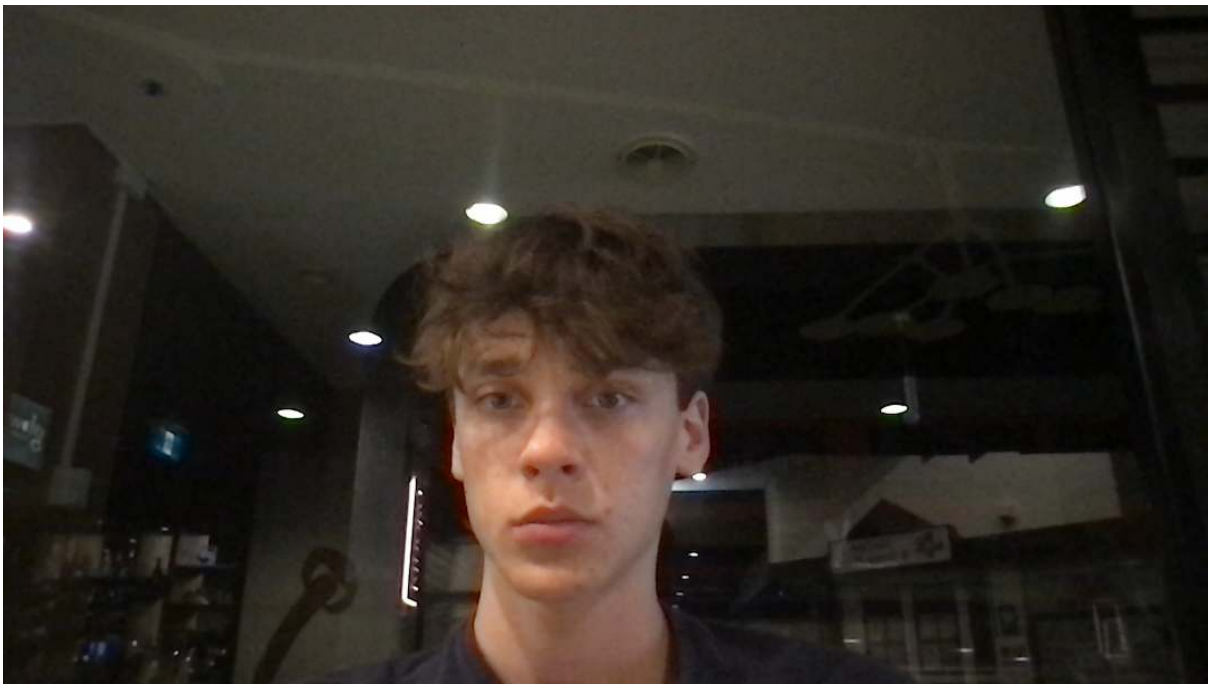


Figure 25: *Graph of quality vs target size for 'webcam' image.*

The webcam image (Appendix Y) was compressed similarly by both AVIF and JFIF, however, the SSIM index reports that AVIF compressed to a higher quality in the lower target sizes. The image featured a high amount of noise and thus low spatial correlation that made the AVIF prediction step less effective than in other images.

Overall results

Overall, AVIF performed better than JFIF in the quality experiment. To make judgments, the SSIM values for all images at all target sizes were normalised using min-max feature scaling, where the minimum and maximum were found in the quality values from both AVIF and JFIF. These normalised values were averaged for both codecs to produce a chart (Figure 26) that represents the average quality across all images for both codecs. Figure 28 and 29 show images with similar normalised SSIM qualities as the averages for AVIF and JFIF. In these images, the difference between AVIF and JFIF can be seen, compression artefacts are visible in Figure 28 but are difficult to discern in Figure 29.

Figure 26: *JFIF and AVIF image quality comparison (normalised)*

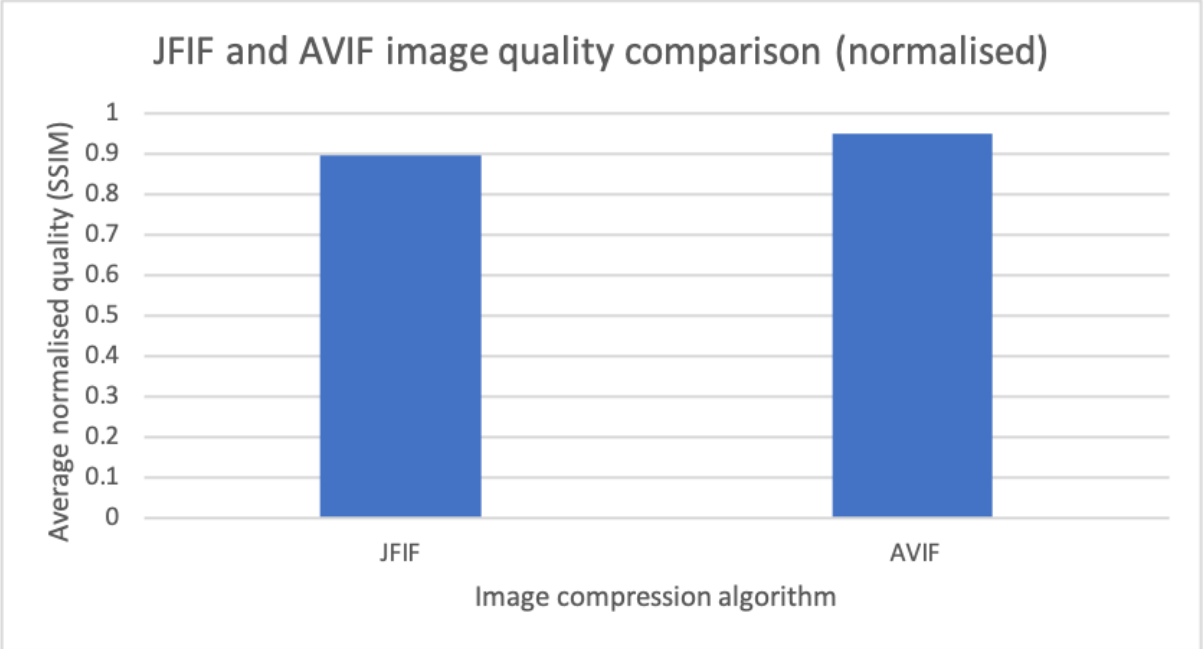


Figure 27: *JFIF and AVIF image quality comparison*

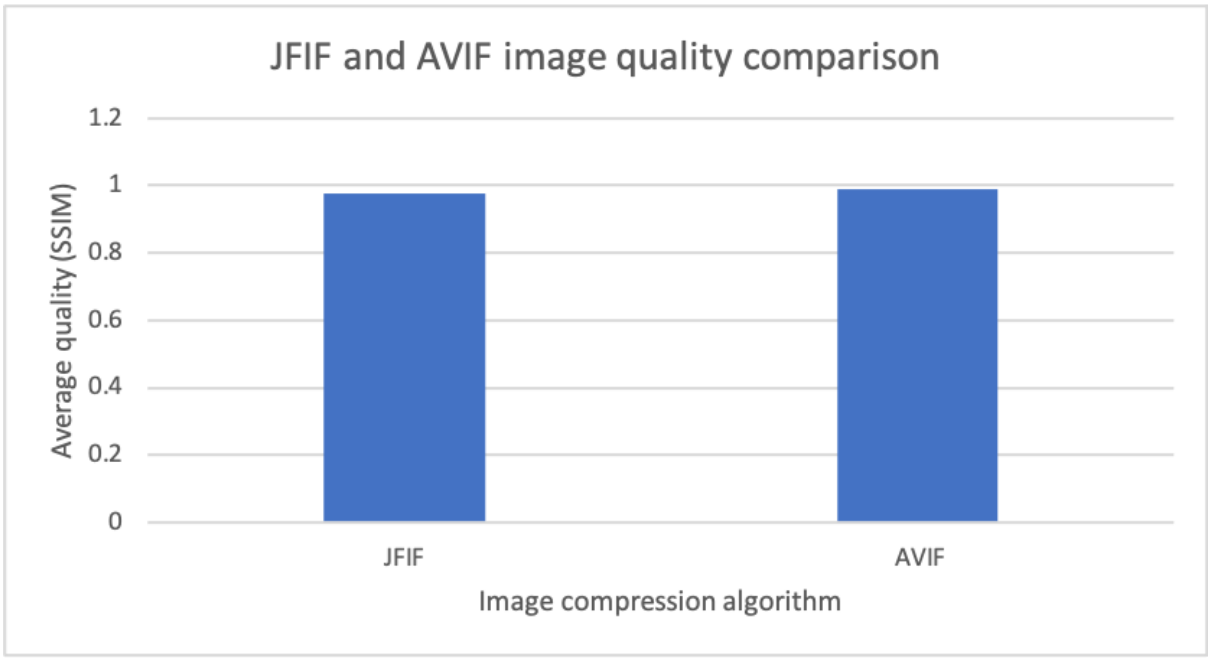


Figure 28: *'ipodClose' compressed to 50,000 bytes using JFIF, the image with the closest normalised quality (0.8954) to average normalised JFIF quality (0.8958)*

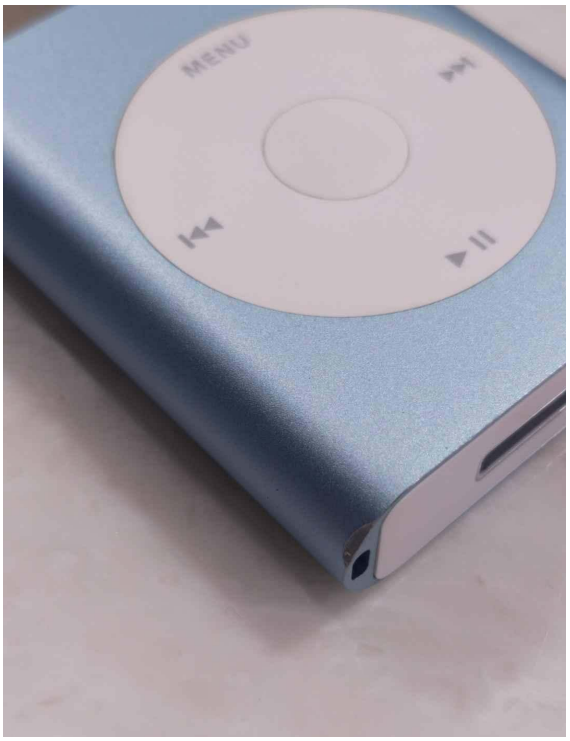


Figure 29: *'market' compressed to 300,000 bytes using AVIF, the image with the closest normalised quality (0.9449) to average normalised AVIF quality (0.9497)*

Encoding time

To analyse encoding time, a primary investigation was undertaken. The same 11 uncompressed photos used in the quality investigation were used to analyse encoding time. Since encode time will be dependent on the computer the algorithms run on, the experiment was conducted on the cloud to allow simple replication of the hardware the algorithms ran on. The code was run on Amazon Web Services EC2 using a 't2.micro' instance. The experiment's code was written in golang to avoid the runtime overhead of interpreted languages. The code relied on a number of libraries, detailed in table 4. libjpegturbo was used instead of libjpeg because it includes a number of performance optimisations that do not compromise on JFIF specification compliance. It is therefore an obvious choice in performance critical scenarios, like on an image CDN, that this experiment seeks to mimic.

Table 4: *Table of libraries used in encoding time experiment*

Library used	**Significance**
go-avif (Hiiragi 2023)	To add AVIF encoding support to golang using libaom.
golibjpegturbo (Antonini 2022)	To add JFIF encoding support to golang using libjpegturbo.

In order to record the time to encode images using each compression algorithm, the current time is recorded, then, in the following line of code, the images are compressed and finally elapsed time is recorded. This process is repeated 20 times for each image and compression algorithm.

The average time for each image and compression algorithm was calculated. The time was divided by the amount of megapixels in the image to generate average seconds per megapixel in an image.

Results and analysis

Across the 20 trials, JFIF's superiority in encode speed became abundantly clear. The relative range for AVIF was 1.59, whereas JFIF had a relative range of 0.28. This indicates that AVIF was much less consistent in the time it took to compress images. In JFIF, the seconds per megapixel varied slightly. On average, JFIF was 125.5 times faster than AVIF in encoding images.

Table 5: *Seconds per megapixel*

	Average seconds per megapixel (s/MP)	
image	**AVIF**	**JFIF**
person	3.73	0.04
screenshot	2.50	0.03
webcam	3.26	0.05
farmFromBridge	7.17	0.04
market	9.93	0.04
busStop	8.28	0.04
sky	3.26	0.04
ipodStanding	2.58	0.04
phone	2.26	0.04
ipodClose	4.26	0.04

viewOfLabyrinth	5.78	0.04

Limitations

Quality experiment

In image compression, one of either image quality or file size must be constant and the other used to measure the effectiveness of the codec. Prior to conducting the experiment, file size was chosen to be constant but this raised an issue. The properties of the image, such as resolution and structural complexity, affected the quality degradation caused by compression. This limited the visibility of compression artefacts, making comparisons difficult to draw. Compressing the image to constant quality would theoretically make images appear to have equal compression artefacts. However, this approach would limit human interpretation since human comparisons would not be at constant file size, which is important to supplement alongside image quality metrics that attempt to mimic human quality perception.

The images were only evaluated for quality using SSIM, which is now a relatively old image quality metric. Machine learning metrics, such as the Deep Image Structure and Texture Similarity metric may yield better correlations with human perceptions, however in testing proved to be unreliable.

Encode time experiment

A data centre virtual machine was chosen to run the experiment because large scale image compression operations, like image CDNs are run in such environments. However, such large scale

operations may use specialised hardware, drastically reducing the encode time. The use of such hardware encoders could be considered in future research.

The headless ‘Ubuntu’ virtual machine did not have the overhead of a graphical interface, but may have had some background processes, randomly affecting its performance. A more lightweight operating system could have been chosen to limit the impact of this issue. Other virtual machines on the same physical machine may have affected its performance. This issue could have been rectified by running the experiment on standardised, dedicated hardware.

Conclusion

AVIF is a 'better' lossy image codec than JFIF in limited aspects. The quality experiment clearly showed that AVIF was better than JFIF at maximising the quality of an image within a constrained file size. In general analysis of the images used, AVIF produced images that had a SSIM index 0.01 higher than JPEG on average. In no image or target file size tested did JFIF outperform AVIF on the SSIM index. The reasoning for AVIF's quality superiority is likely due to the introduction of a prediction step and variable block sizes, both of which allow AVIF to compress images more effectively. However, AVIF's prediction and variable block sizes have significant time costs as seen in the encode time experiment, where JFIF compressed images 125.5 times faster than AVIF. This discrepancy, on AWS US East t.2 micro pricing in June of 2023 would cost USD$0.12 per 1,000,000 images for JPEG and USD$15 per 1,000,000 images for AVIF (Amazon Web Services n.d). This makes JFIF seem the more economical option, however, if bandwidth cost and costs associated with a degraded user experience are considered, AVIF is a better option. JFIF's user experience may be greater in environments where images are loaded and shown to a user progressively, in which case it may be a better choice.

References

Alex Clark, Andrew Murray, & Hugo van Kemenade 2010, 'Pillow', viewed March 6, 2023, <https://pillow.readthedocs.io/en/stable/index.html>.

Alliance for Open Media 2022, 'AV1 Image File Format (AVIF)', viewed January 22, 2023, <https://aomediacodec.github.io/av1-avif/>.

Antonini, E 2022, 'antonini/golibjpegturbo', viewed March 6, 2023, <https://github.com/antonini/golibjpegturbo>.

Chiariglione, L 2018, 'A crisis, the causes and a solution', *Leonardo's Blog*, viewed May 31, 2023, <https://blog.chiariglione.org/a-crisis-the-causes-and-a-solution/>.

Cloudflare 'How website performance affects conversion rates', *Cloudflare*, viewed June 8, 2023, <https://www.cloudflare.com/learning/performance/more/website-performance-conversion-rates/>.

Colourspaces (JPEG Pt0)- Computerphile 2015, viewed October 21, 2022, <https://www.youtube.com/watch?v=LFXN9PiOGtY>.

Ding, K, Ma, K, Wang, S & Simoncelli, EP 2020, 'Image Quality Assessment: Unifying Structure and Texture Similarity', *IEEE Transactions on Pattern Analysis and Machine Intelligence*, viewed March 5, 2023, <http://arxiv.org/abs/2004.07728>.

Dintino, F 2023, 'pillow-avif-plugin', viewed March 6, 2023, <https://github.com/fdintino/pillow-avif-plugin>.

Fastly 'Edge cloud pricing', viewed January 25, 2023, <https://www.fastly.com/pricing>.

Fronius, L 2020, 'Comparing image formats. JPEG vs WebP vs AVIF.', viewed January 20, 2023, <https://fronius.me/articles/2020-10-14-comparing-image-formats-jpg-webp-avif.html>.

Gurtovaya, P & Barnov, A 2021, 'Decoding AVIF: Deep dive with cats and imgproxy—Martian Chronicles, Evil Martians' team blog', *evilmartians.com*, viewed June 1, 2023, <https://evilmartians.com/chronicles/decoding-avif-deep-dive-with-cats-and-imgproxy>.

Han, J, Li, B, Mukherjee, D, Chiang, C-H, Grange, A, Chen, C, Su, H, Parker, S, Deng, S, Joshi, U, Chen, Y, Wang, Y, Wilkins, P, Xu, Y & Bankoski, J 2021, 'A Technical Overview of AV1', viewed October 21, 2022, <http://arxiv.org/abs/2008.06091>.

Hiiragi, K 2023, 'go-avif', viewed March 6, 2023, <https://github.com/Kagami/go-avif>.

International Telecommunication Union 'Recommendation T.81 (09/92): Information technology—Digital compression and coding of continuous-tone still images—Requirements and guidelines', viewed January 20, 2023, <https://www.wikidata.org/wiki/Q26289072>.

Kastryulin, S, Zakirov, J & Prokopenko, D 2023, 'photosynthesis-team/piq', viewed March 6, 2023, <https://github.com/photosynthesis-team/piq>.

Lu, L 2019, 'Image policies for fast load times and more', *web.dev*, viewed January 19, 2023, <https://web.dev/image-policies/>.

Montgomery, C 'AV1: next generation video – The Constrained Directional Enhancement Filter – Mozilla Hacks - the Web developer blog', *Mozilla Hacks – the Web developer blog*, viewed January 23, 2023, <https://hacks.mozilla.org/2018/06/av1-next-generation-video-the-constrained-directional-enhancement-filter>.

Mozilla 2013, 'Studying Lossy Image Compression Efficiency', *Mozilla Research*, viewed May 31, 2023, <https://research.mozilla.org/2013/10/17/studying-lossy-image-compression-efficiency>.

Paszke, A, Gross, S, Massa, F, Lerer, A, Bradbury, J, Chanan, G, Killeen, T, Lin, Z, Gimelshein, N, Antiga, L, Desmaison, A, Kopf, A, Yang, E, DeVito, Z, Raison, M, Tejani, A, Chilamkurthy, S, Steiner, B, Fang, L, Bai, J, Chintala, S, Wallach, H,

Larochelle, H, Beygelzimer, A, d'Alché-Buc, F, Fox, E & Garnett, R 2019, 'PyTorch: An Imperative Style, High-Performance Deep Learning Library', *Advances in Neural Information Processing Systems 32*, viewed March 6, 2023, <http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf>.

Pound, M, *JPEG DCT, Discrete Cosine Transform (JPEG Pt2)- Computerphile* 2015, viewed August 16, 2022, <https://www.youtube.com/watch?v=Q2aEzeMDHMA>.

Pound, M, *JPEG 'files' & Colour (JPEG Pt1)- Computerphile* 2015, viewed October 21, 2022, <https://www.youtube.com/watch?v=n_uNPbdenRs>.

Pound, M, *Resizing Images - Computerphile* 2016, viewed January 23, 2023, <https://www.youtube.com/watch?v=AqscP7rc8_M>.

Pound, M, *The AV1 Video Codec* 2019, viewed October 21, 2022, <https://www.youtube.com/watch?v=qubPzBcYCTw>.

Richter, T 2023, 'libjpeg', viewed June 20, 2023, <https://github.com/thorfdbg/libjpeg>.

Schönberger, J, Walt, S van der & Nunez-Iglesias, J 2023, 'scikit-image: Image processing in Python', viewed March 6, 2023, <https://github.com/scikit-image/scikit-image>.

Sneyers, J 2022, 'The Case for JPEG XL', *Cloudinary Blog*, viewed May 31, 2023, <https://cloudinary.com/blog/the-case-for-jpeg-xl>.

Sneyers, J 2022, 'Contemplating Codec Comparisons', *Cloudinary Blog*, viewed January 25, 2023, <https://cloudinary.com/blog/contemplating-codec-comparisons>.

Terriberry, T, *The Problem with JPEG - Computerphile* 2015, viewed October 21, 2022, <https://www.youtube.com/watch?v=yBX8GFqt6GA>.

Appendices

Appendix A — functions to compress images.

```
import os
from PIL import Image
import pillow_avif
from skimage.io import imread
import uuid

def compressImageToSize(pathRoot, outputFolderName, file, desiredSize, step, format, extension):
        # instantiate class
        class imageInfo:
        quality = [50, 50, 50] # an array of the past 3 qualities of outputs
        size = None # the size of the output images
        sizeDelta = None # the difference between the output size and the desired size
        path = "" # image path
        imageSizeEqual = False
        counter = 0

        outputPath = os.path.join(pathRoot + outputFolderName + "/", str(desiredSize/1000) + "kb" + "/")
        try:
        os.mkdir(outputPath)
        except:
        print("directory already exists")

        os.chdir(pathRoot)
        while not imageSizeEqual:
        # compress the image using the desired format and extension, save to desired location
```

```
        with Image.open(file) as im:
                im.save(outputPath + "/" + file + extension, format,
quality=imageInfo.quality[counter%3])
                imageInfo.path = (outputPath + "/" + file + extension)
        imageInfo.size = os.path.getsize(imageInfo.path)

        # check if image is below, above or equal to the desired size and adjust quality
accordingly
        if imageInfo.size < desiredSize:
                imageInfo.quality[counter%3] = imageInfo.quality[counter%3-1] + step
        elif imageInfo.size > desiredSize:
                imageInfo.quality[counter%3] = imageInfo.quality[counter%3-1] - step
        else:
                imageSizeEqual = True
        print(imageInfo.quality)

        # set flag to true if the quality of the compressions is bouncing between two
values
        if imageInfo.quality[counter%3] == imageInfo.quality[counter%3-2]:
                imageSizeEqual = True

        if ((imageInfo.quality[counter%3] == 0) or (imageInfo.quality[counter%3] ==
100)):
                imageSizeEqual = True

        counter += 1

        # calculate the delta
        imageInfo.sizeDelta = abs(desiredSize - imageInfo.size)
        # return the image info
        return imageInfo
```

```
## takes in BMP image file paths and outputs quality
def calcImageQuality(imgXPath, imgYPath):
        return(0)

def convertBackToBMP(imageInfo, tmpFolderName):
        name = str(uuid.uuid4())
        with Image.open(imageInfo.path) as im:
        im.save(tmpFolderName + "/" + name + ".bmp", "BMP")
        return(tmpFolderName + "/" + name + ".bmp")

def    getCompressedQualityOfImage(file,    pathRoot,    outputFolderName,    tmpFolderName,
fileSize, qualityStep, format, extension):
        return             calcImageQuality(convertBackToBMP(compressImageToSize(pathRoot,
outputFolderName,  file,  fileSize,  qualityStep,  format,  extension),  tmpFolderName),
(file))
```

Appendix B — main loop

```
from compressImage import getCompressedQualityOfImage
import csv
import os, glob

pathRoot = "" ## path to experiment folder. path removed to protect candidate privacy.
outputFolderName = "output"
tmpFolderName = "tmp"
fileSizes = [50_000, 100_000, 150_000, 200_000, 250_000, 300_000]
formats = [["JFIF", ".JFIF"], ["AVIF", ".avif"]]

os.chdir(pathRoot)
for file in glob.glob("*.bmp"):
        with open(file + ".csv", 'w') as csvFile:
        header = []
        for format in formats:
                header.append(format[0])
                header.extend("" for x in range(len(fileSizes)-1))
        csvRows = [header, fileSizes]
        csvWriter = csv.writer(csvFile)

        row = []
        for fileSize in fileSizes:
                for format in formats:
                row.append(getCompressedQualityOfImage(file,   pathRoot,   outputFolderName,
tmpFolderName, fileSize, 1, format[0], format[1]))
        print(row)

        csvRows.append(row)
        csvWriter.writerows(csvRows)
```

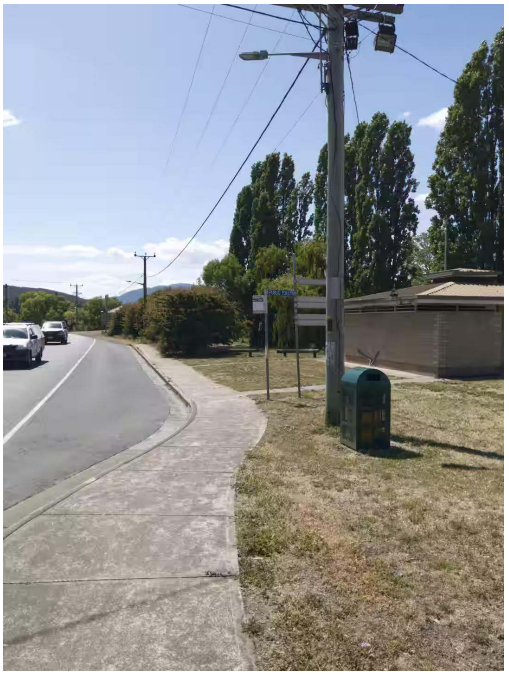
Appendix C — raw quality data table

	quality (SSIM)																					
	viewOfLabyrinth		person		webcam		busStop		sky		farmFromBridge		ipodClose		ipodStanding		market		phone		screenshot	
target size	JFIF	AVIF	JFIF	AVIF	JFIF	AVIF	JFIF	AVIF	JFIF	AVIF	JFIF	AVIF	JFIF	AVIF	JFIF	AVIF	JFIF	AVIF	JFIF	AVIF	JFIF	AVIF
50000	0.7801	0.9689	0.9653	0.9792	0.9874	0.9919	0.8901	0.9593	0.9833	0.9912	0.8945	0.9505	0.9770	0.9896	0.9788	0.9887	0.8187	0.9278	0.9995	0.9998	0.9550	0.9982
100000	0.9419	0.9806	0.9787	0.9852	0.9937	0.9953	0.9605	0.9764	0.9893	0.9948	0.9559	0.9692	0.9903	0.9933	0.9882	0.9930	0.9302	0.9612	0.9997	0.9998	0.9962	0.9991
150000	0.9708	0.9867	0.9832	0.9885	0.9952	0.9953	0.9780	0.9831	0.9925	0.9961	0.9717	0.9776	0.9925	0.9948	0.9912	0.9950	0.9600	0.9743	0.9998	1.0000	0.9987	0.9991
200000	0.9809	0.9892	0.9860	0.9901	0.9952	0.9953	0.9858	0.9878	0.9938	0.9972	0.9789	0.9833	0.9936	0.9956	0.9927	0.9959	0.9718	0.9817	0.9998	1.0000	0.9990	0.9991
250000	0.9870	0.9909	0.9885	0.9920	0.9952	0.9953	0.9890	0.9902	0.9955	0.9972	0.9835	0.9873	0.9946	0.9968	0.9939	0.9967	0.9782	0.9853	0.9998	1.0000	0.9991	0.9991
300000	0.9902	0.9927	0.9898	0.9934	0.9952	0.9953	0.9912	0.9919	0.9955	0.9972	0.9859	0.9899	0.9954	0.9968	0.9950	0.9967	0.9821	0.9879	0.9998	1.0000	0.9992	0.9991

Appendix D — raw encode time data table

	time (s)																																			
	Trial 1		Trial 2		Trial 3		Trial 4		Trial 5		Trial 6		Trial 7		Trial 8		Trial 9		Trial 10		Trial 11		Trial 12		Trial 13		Trial 14		Trial 15		Trial 16		Trial 17		Trial 18		Trial 19		Trial 20		Average	
image	AVIF	JPEG	AVIF	JPEG	AVIF	JPEG	AVIF	JPEG	AVIF	JPEG	AVIF	JPEG	AVIF	JPEG	AVIF	JPEG	AVIF	JPEG	AVIF	JPEG	AVIF	JPEG	AVIF	JPEG	AVIF	JPEG	AVIF	JPEG	AVIF	JPEG	AVIF	JPEG	AVIF	JPEG	AVIF	JPEG	AVIF	JPEG
person	6.11	0.06	5.76	0.06	5.78	0.06	5.79	0.06	5.78	0.06	5.85	0.06	5.81	0.06	5.77	0.06	5.80	0.06	5.78	0.06	5.80	0.06	5.77	0.06	5.80	0.06	5.78	0.06	5.76	0.06	5.81	0.06	5.80	0.06	5.82	0.06	5.80	0.06	5.80	0.06	5.81	0.06
screenshot	5.88	0.08	5.82	0.08	5.70	0.08	5.75	0.08	5.75	0.09	5.78	0.08	5.74	0.08	5.75	0.08	5.73	0.08	5.76	0.08	5.74	0.08	5.75	0.08	5.77	0.08	5.76	0.08	5.77	0.08	5.78	0.08	5.73	0.08	5.76	0.08	5.76	0.08	5.76	0.08	5.76	0.08
webcam	2.14	0.03	2.14	0.02	2.14	0.02	2.15	0.02	2.16	0.02	2.15	0.02	2.13	0.02	2.13	0.03	2.14	0.03	2.15	0.03	2.14	0.02	2.13	0.02	2.14	0.02	2.14	0.02	2.15	0.02	2.15	0.03	2.14	0.02	2.14	0.02	2.15	0.02	2.15	0.02	2.14	0.03
farmFromBridge	11.1	0.06	11.2	0.06	11.2	0.06	11.1	0.06	11.2	0.06	11.2	0.06	11.2	0.06	11.1	0.06	11.2	0.06	11.2	0.06	11.2	0.06	11.1	0.06	11.2	0.06	11.2	0.06	11.2	0.06	11.1	0.06	11.2	0.06	11.2	0.06	11.2	0.06	11.2	0.06	11.2	0.06
market	15.5	0.06	15.5	0.06	15.5	0.06	15.5	0.06	15.5	0.06	15.4	0.06	15.5	0.06	15.4	0.06	15.5	0.06	15.5	0.06	15.5	0.06	15.4	0.06	15.5	0.06	15.5	0.06	15.5	0.06	15.4	0.06	15.5	0.06	15.4	0.06	15.5	0.06	15.5	0.06	15.5	0.06
busStop	12.9	0.06	13.0	0.06	12.9	0.06	12.9	0.06	12.9	0.06	12.9	0.06	12.9	0.06	12.9	0.06	12.9	0.06	12.9	0.06	12.9	0.06	12.9	0.06	12.9	0.06	12.9	0.06	12.9	0.06	12.9	0.06	12.9	0.06	12.9	0.06	12.9	0.06	12.9	0.06	12.9	0.06
sky	5.10	0.06	5.09	0.06	5.08	0.06	5.08	0.06	5.06	0.06	5.10	0.06	5.07	0.06	5.07	0.06	5.08	0.06	5.07	0.05	5.05	0.05	5.10	0.06	5.07	0.06	5.20	0.08	5.06	0.05	5.05	0.06	5.06	0.06	5.08	0.06	5.07	0.06	5.08	0.06	5.08	0.06
ipodStanding	4.03	0.06	4.03	0.06	4.05	0.06	4.06	0.06	4.02	0.05	4.04	0.06	4.02	0.06	4.04	0.06	4.04	0.06	4.03	0.06	4.01	0.06	4.03	0.06	4.03	0.05	4.04	0.06	4.01	0.06	4.05	0.06	4.01	0.05	4.06	0.06	4.04	0.06	4.04	0.06	4.03	0.06
phone	2.37	0.04	2.37	0.04	2.37	0.04	2.38	0.04	2.38	0.04	2.38	0.04	2.37	0.04	2.39	0.04	2.37	0.04	2.37	0.04	2.36	0.04	2.38	0.04	2.37	0.04	2.36	0.04	2.38	0.04	2.37	0.04	2.36	0.04	2.39	0.04	2.38	0.04	2.39	0.04	2.37	0.04
ipodClose	6.65	0.06	6.66	0.06	6.67	0.06	6.64	0.06	6.66	0.06	6.65	0.06	6.67	0.06	6.63	0.06	6.68	0.06	6.64	0.06	6.64	0.06	6.62	0.06	6.64	0.06	6.65	0.06	6.65	0.06	6.64	0.05	6.64	0.06	6.66	0.06	6.66	0.06	6.67	0.06	6.65	0.06
viewOfLabyrinth	16.0	0.10	15.9	0.10	15.9	0.10	15.9	0.10	17.1	0.10	15.9	0.10	15.9	0.10	15.8	0.10	16.2	0.10	15.8	0.10	15.9	0.10	15.9	0.10	15.9	0.10	15.9	0.10	15.9	0.10	15.9	0.10	15.9	0.10	15.9	0.10	15.9	0.10	15.9	0.10	16.0	0.10

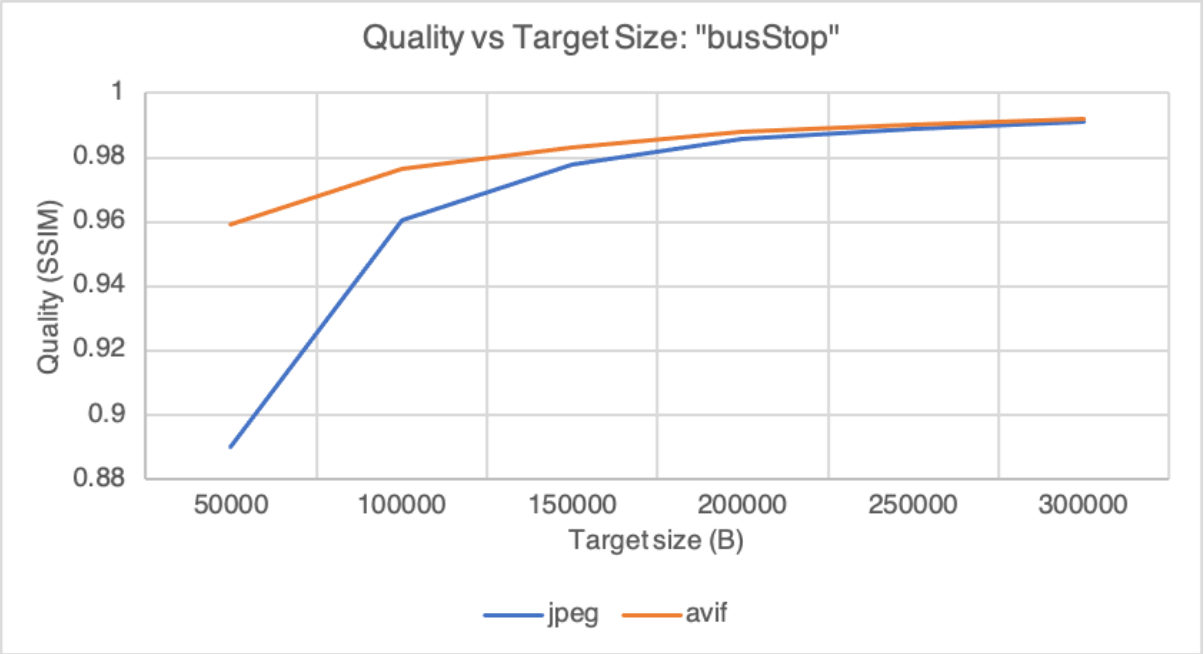
Appendix E — all target sizes levels for 'busStop'

Size (bytes)	JFIF	AVIF
50,000		
100,000		

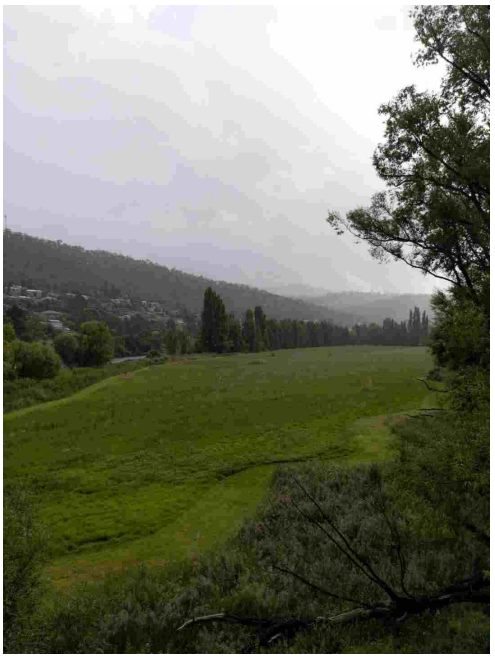
150,000		
200,000		

250,000		
300,000		

Appendix F — quality vs target size for 'busStop'



Appendix G — all target sizes for 'farmFromBridge'

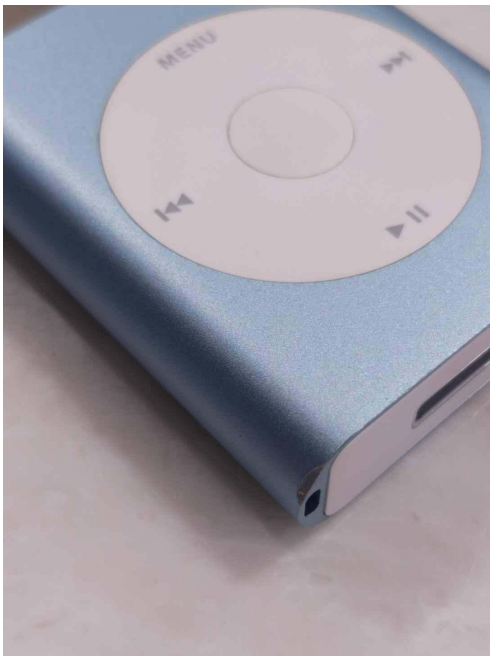
Size (bytes)	JFIF	AVIF
50,000		
100,000		

150,000		
200,000		

250,000		
300,000		

Appendix H — quality vs target size for 'farmFromBridge'

Appendix I — all target sizes for 'ipodClose'

Size (bytes)	JFIF	AVIF
50,000		
100,000		

150,000		
200,000		

250,000		
300,000		

Appendix J — quality vs target size for 'ipodClose'

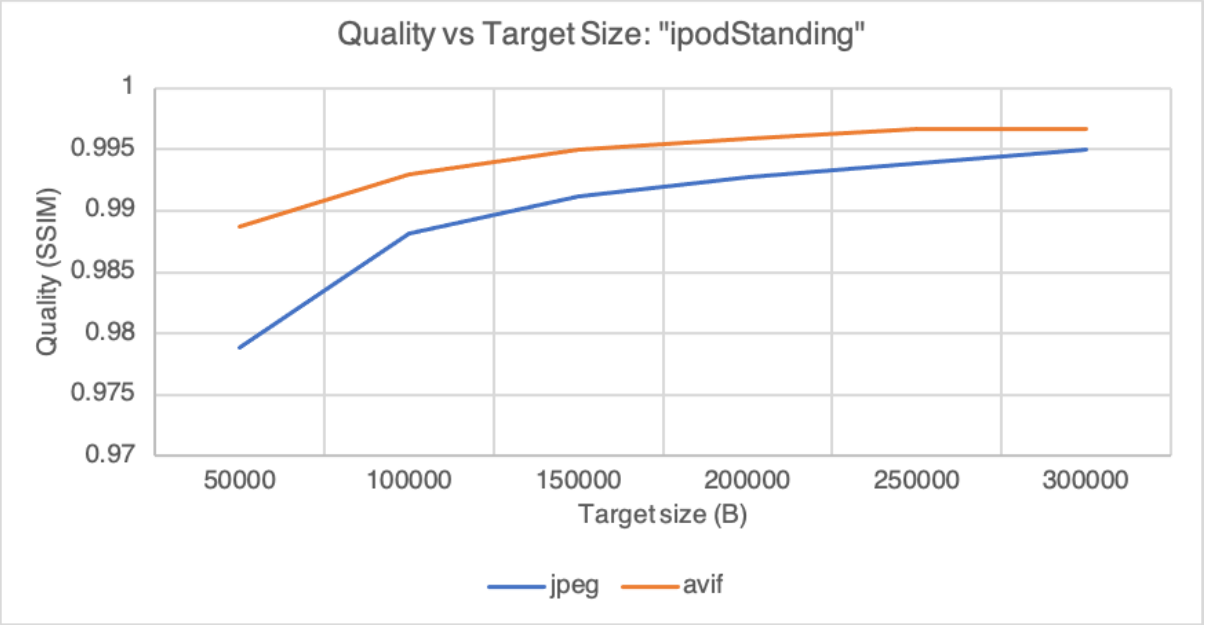
Appendix K — all target sizes for 'ipodStanding'

Size (bytes)	JFIF	AVIF
50,000		
100,000		

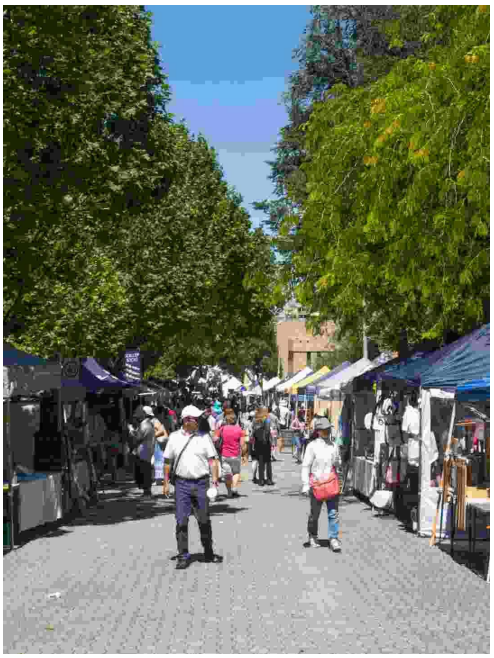
150,000		
200,000		

250,000		
300,000		

Appendix L — quality vs target size for 'ipodStanding'



Appendix M — all target sizes for 'market'

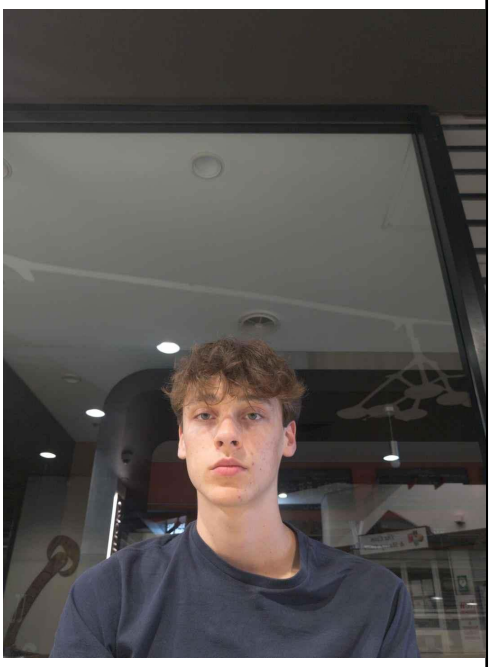
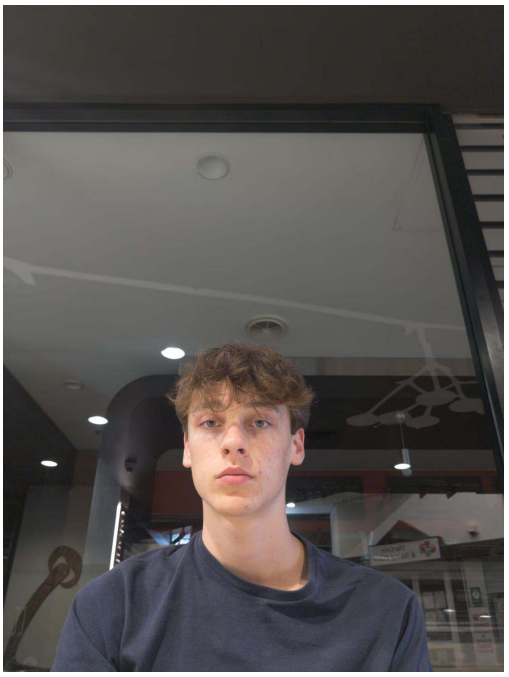
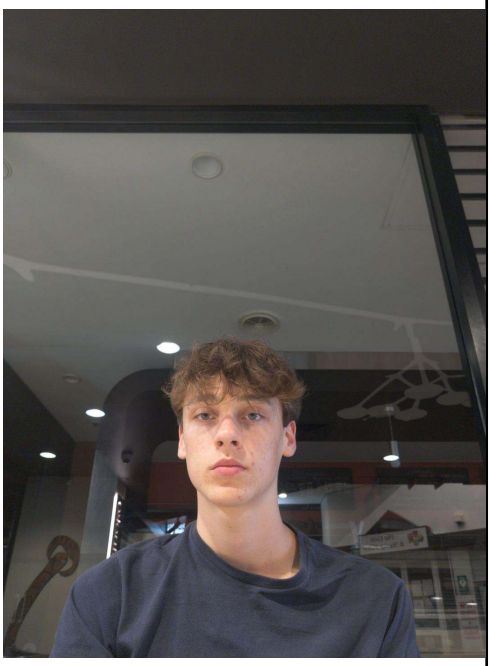
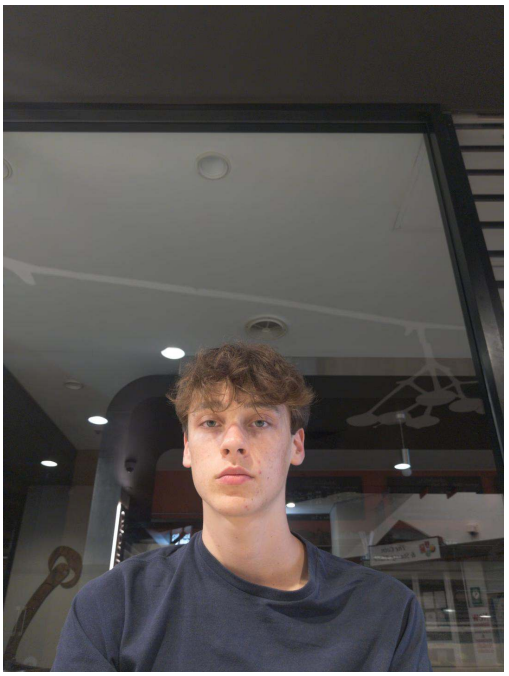
Size (bytes)	JFIF	AVIF
50,000		
100,000		

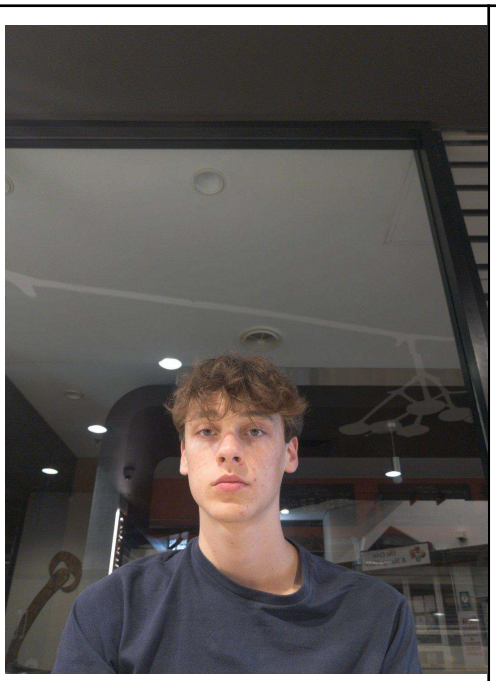
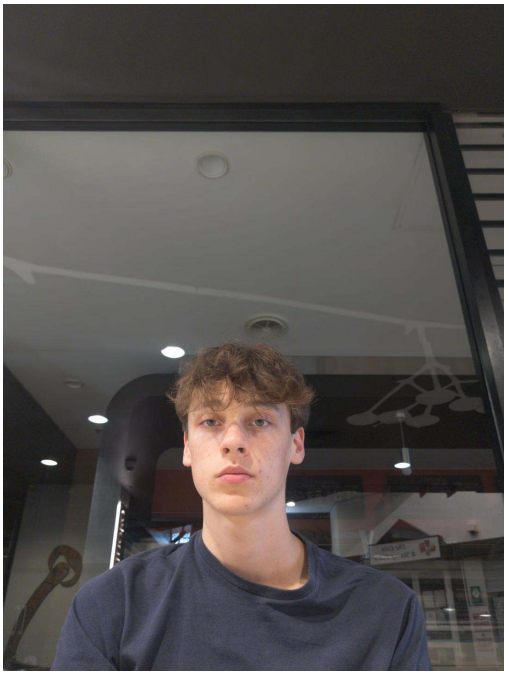
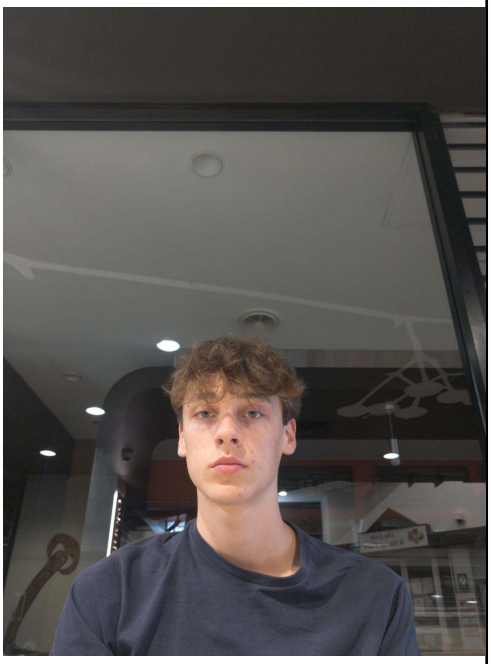
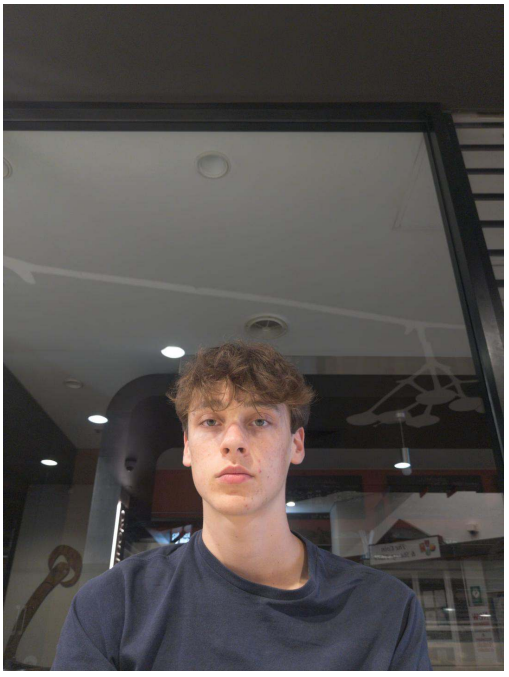
150,000		
200,000		

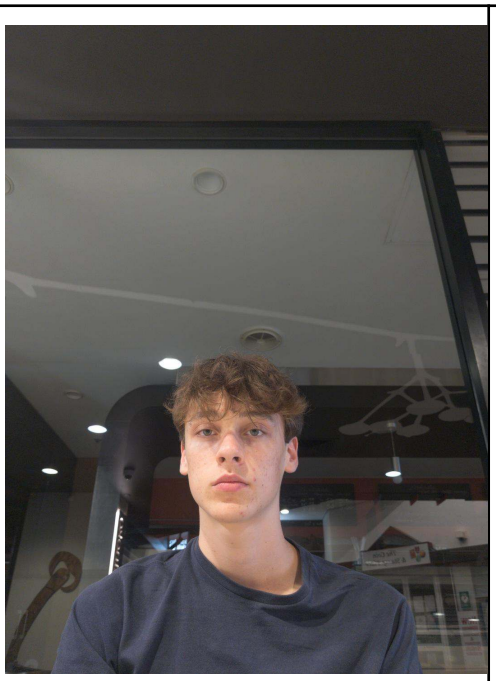
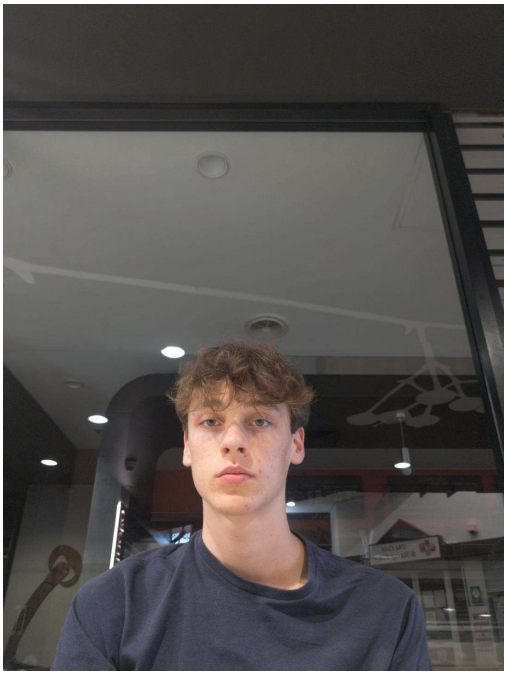
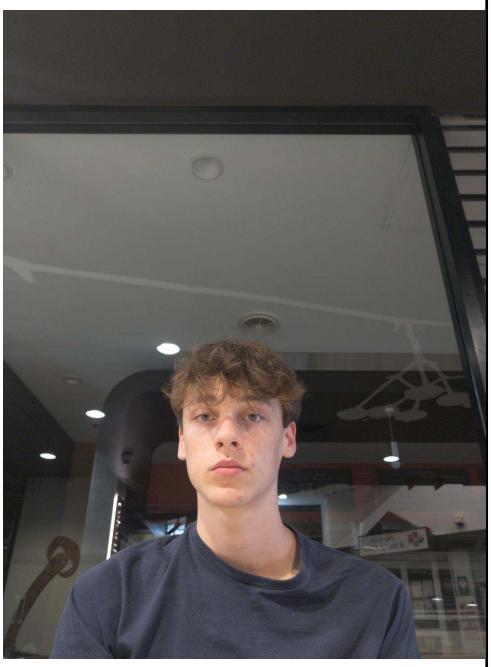
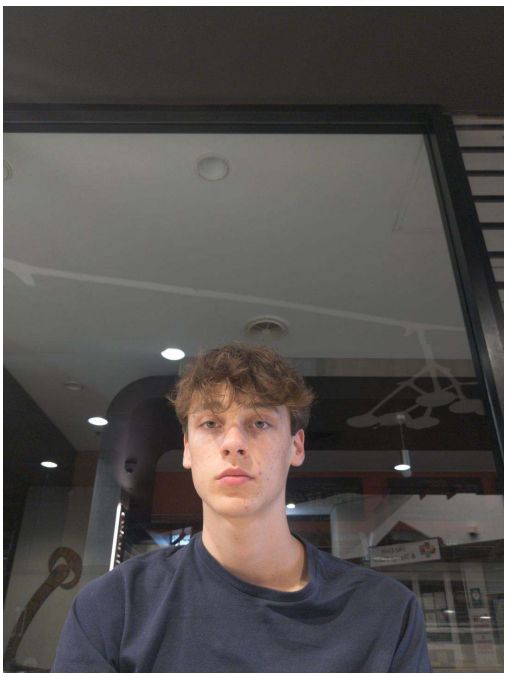
250,000		
300,000		

Appendix N — quality vs target size for 'market'

Appendix O — all target sizes for 'person'

Size (bytes)	JFIF	AVIF
50,000		
100,000		

150,000		
200,000		

250,000		
300,000		

Appendix P — quality vs target size for 'person'

Appendix Q — all target sizes for 'phone'

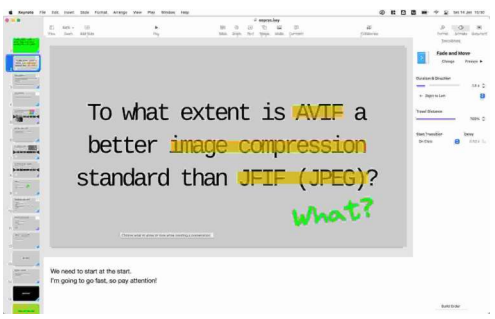
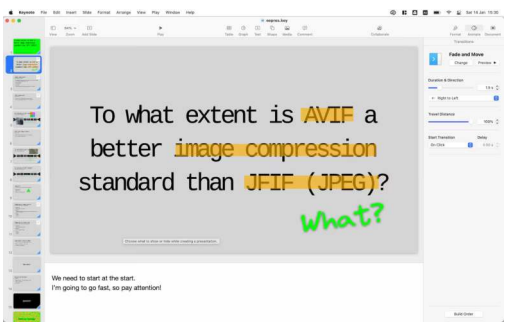
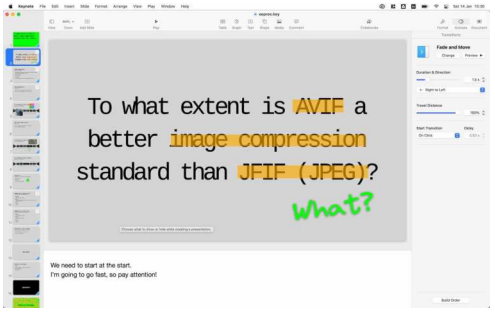
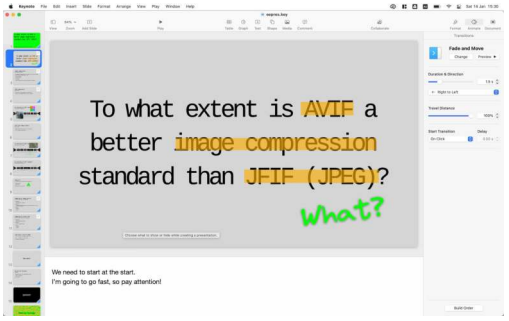
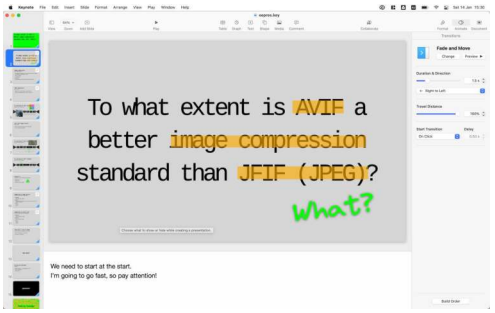
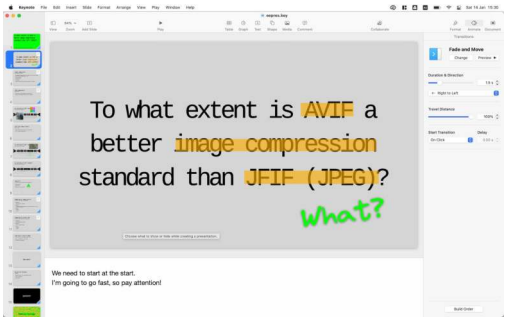
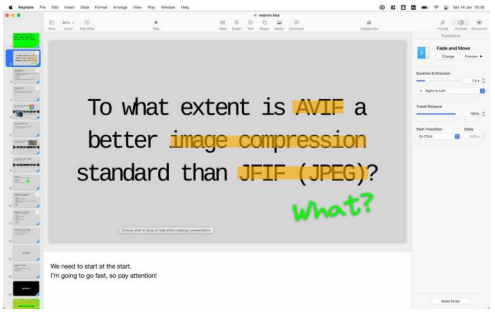
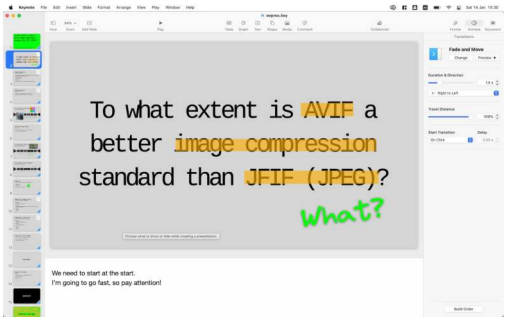
Size (bytes)	JFIF	AVIF
50,000		
100,000		

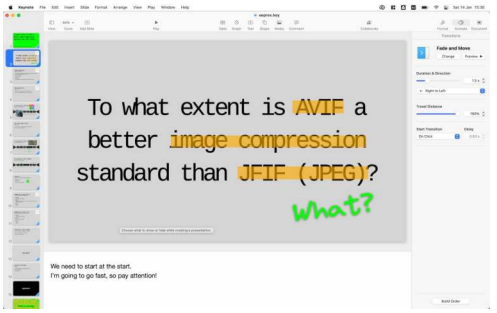
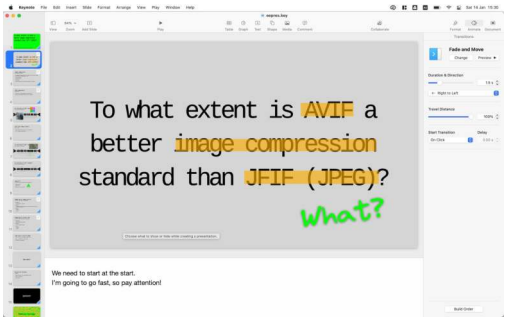
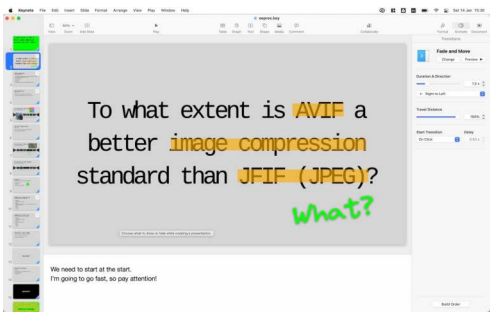
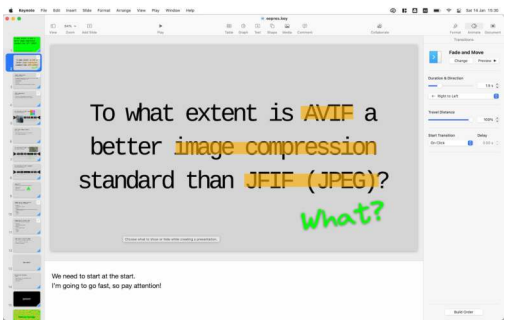
150,000		
200,000		
250,000		

300,000		

Appendix R — quality vs target size for 'phone'

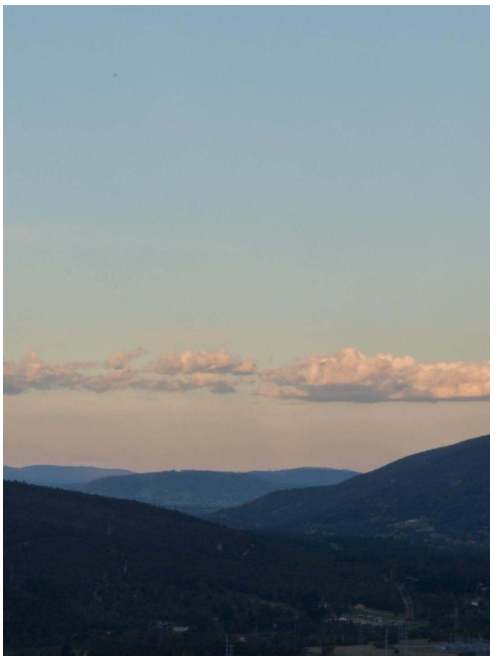
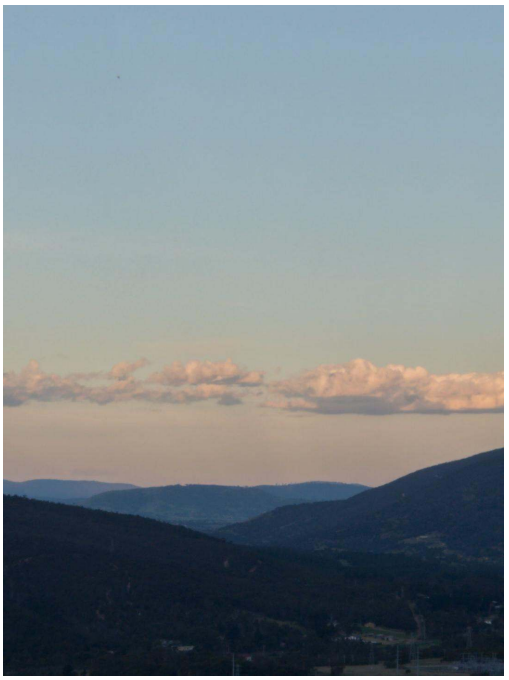
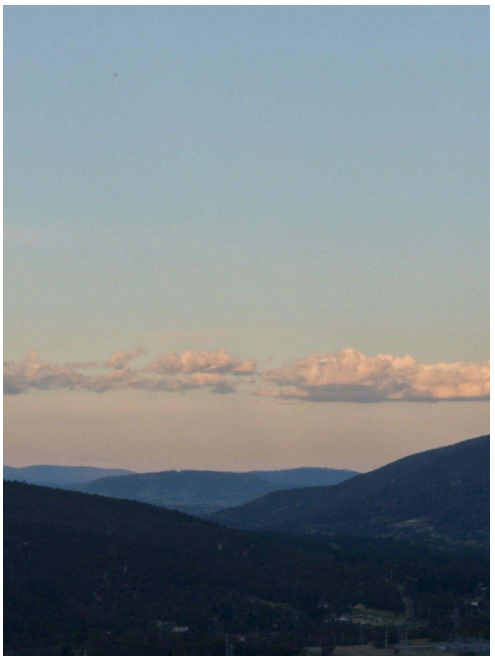
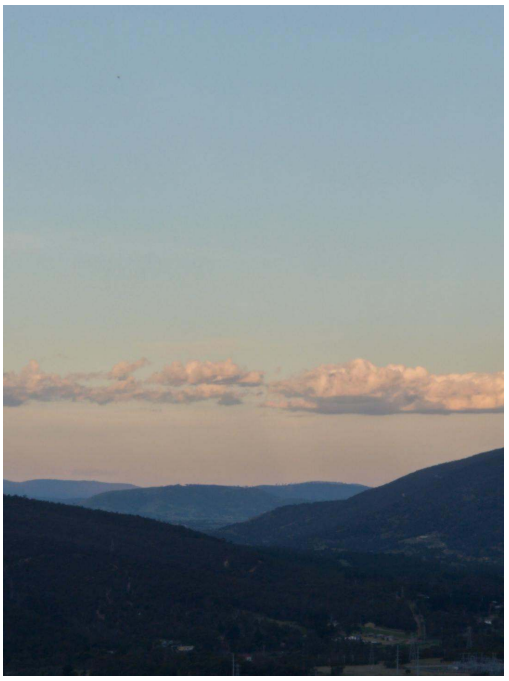
Appendix S — all target sizes for 'screenshot'

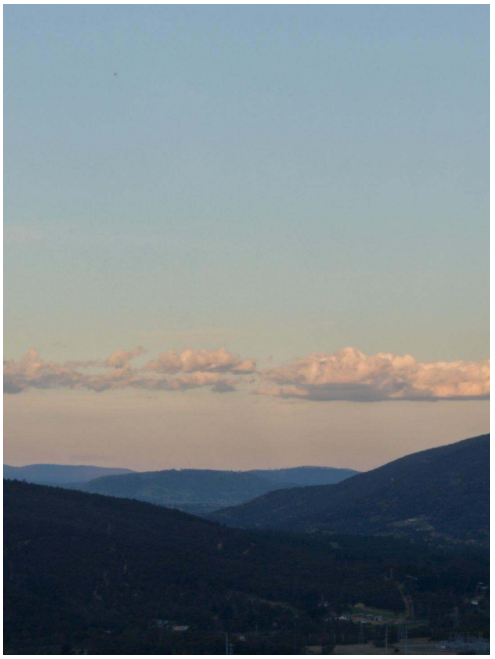
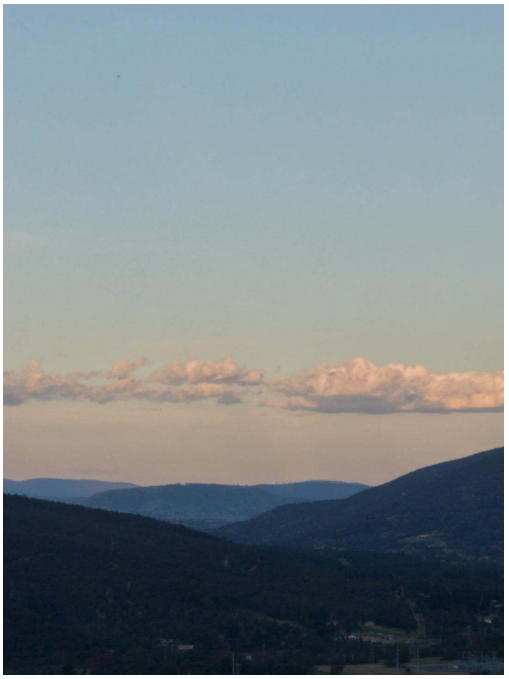
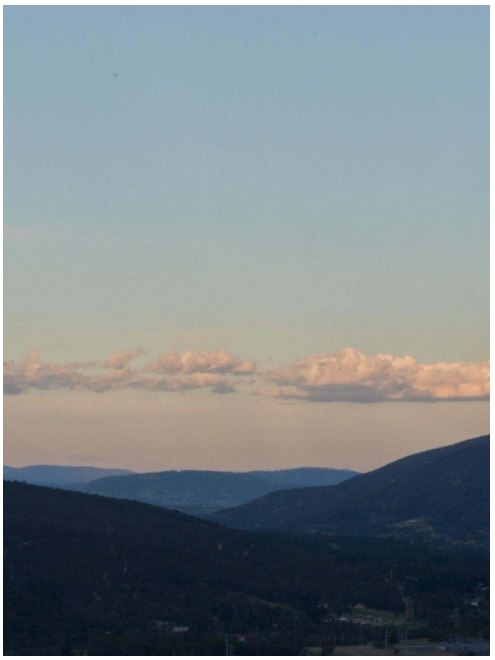
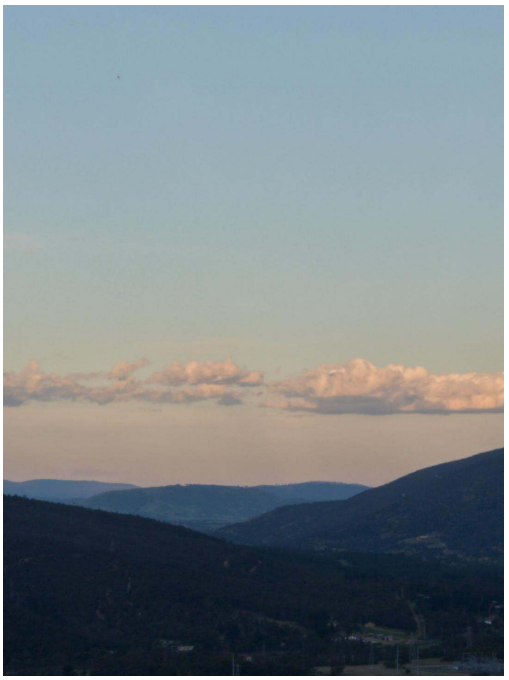
Size (bytes)	JFIF	AVIF
50,000		
100,000		
150,000		
200,000		

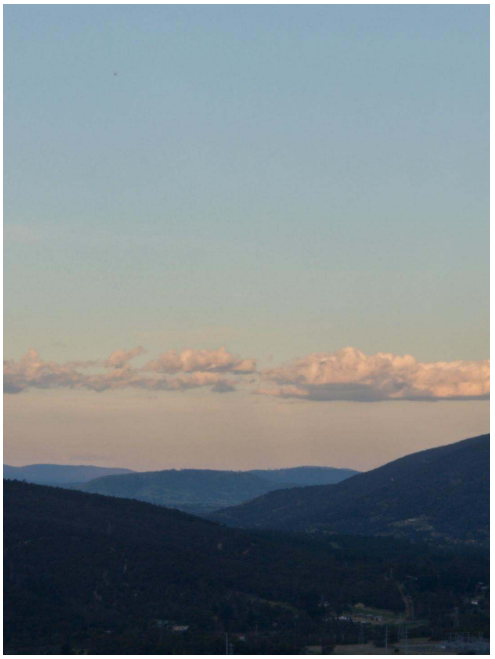
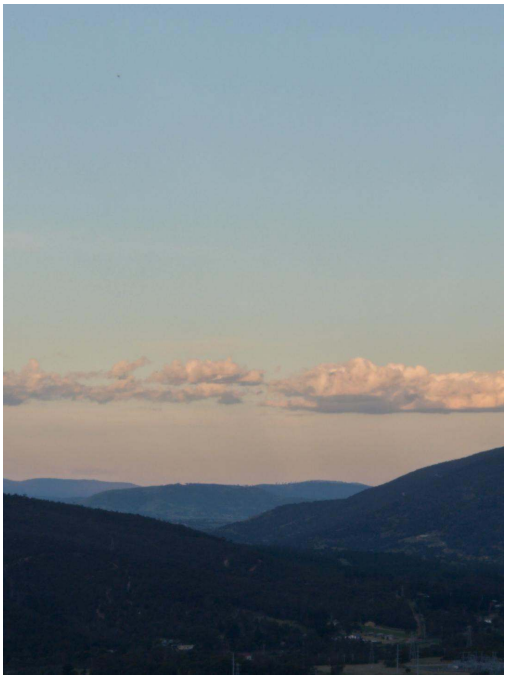
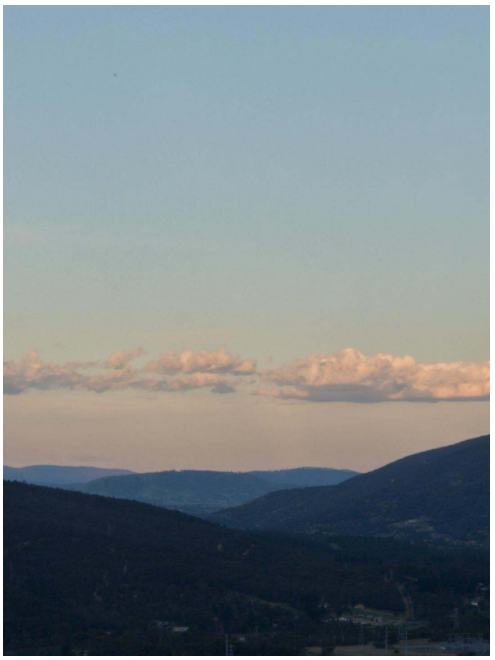
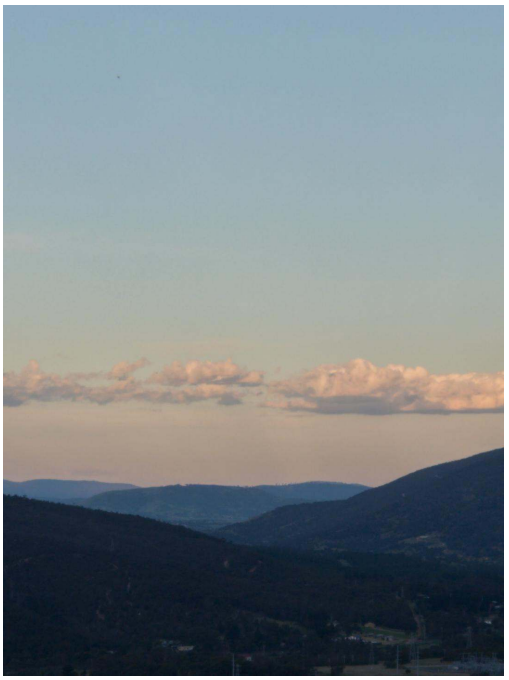
250,000		
300,000		

Appendix T — quality vs target size for 'screenshot'

Appendix U — all target sizes for 'sky'

Size (bytes)	JFIF	AVIF
50,000		
100,000		

150,000		
200,000		

250,000		
300,000		

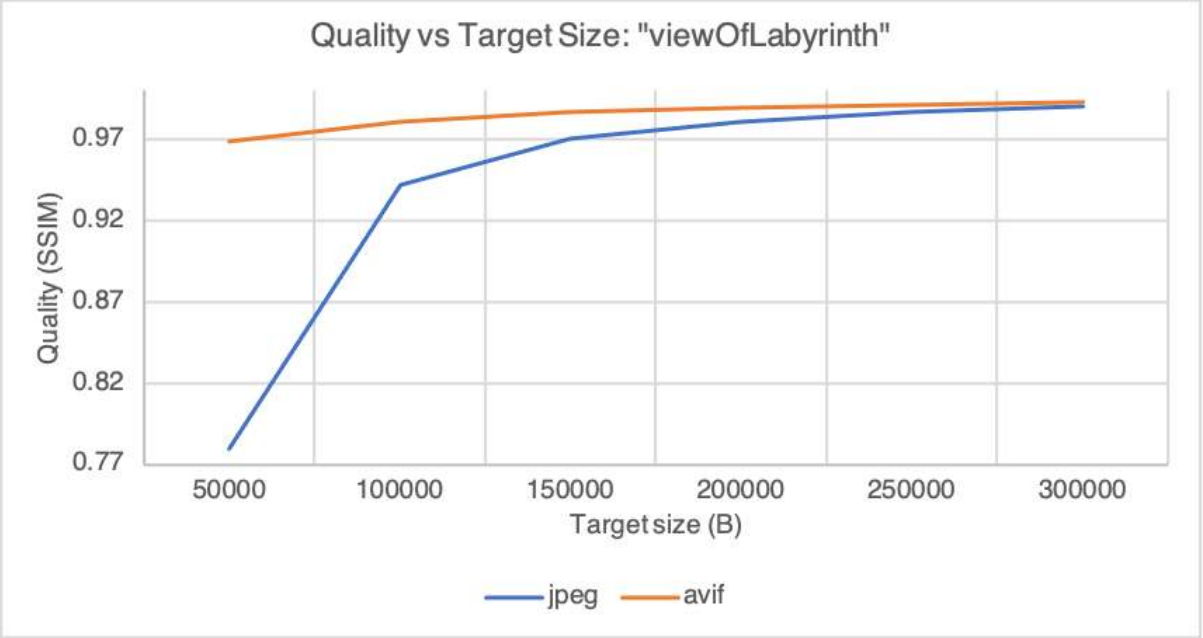
Appendix V — quality vs target size for 'sky'

Appendix W — all target sizes for 'viewOfLabyrinth'

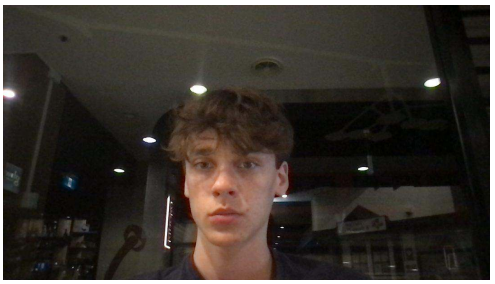
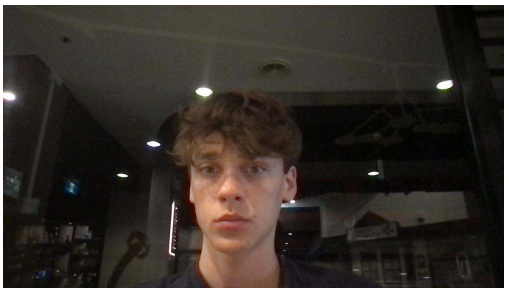
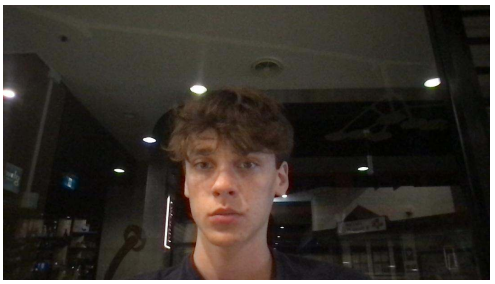
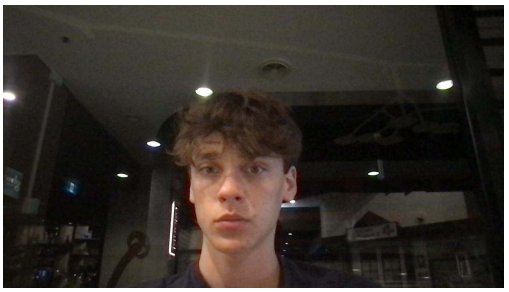
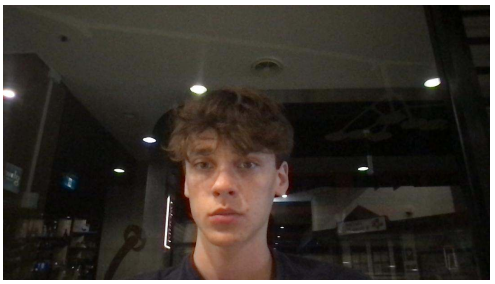
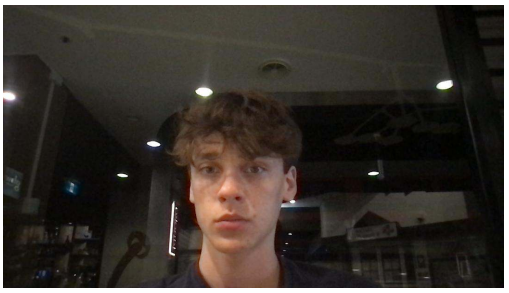
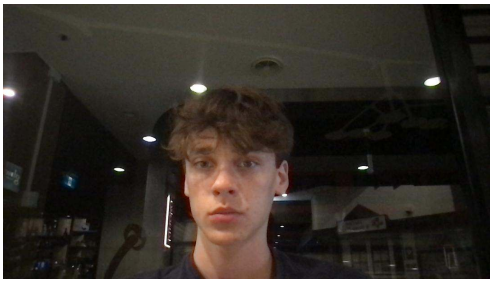
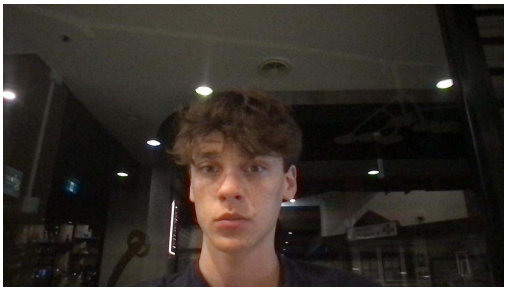
Size (bytes)	JFIF	AVIF
50,000		
100,000		
150,000		

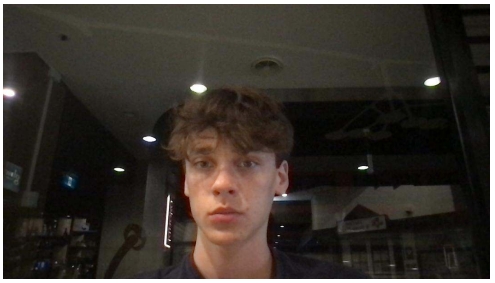
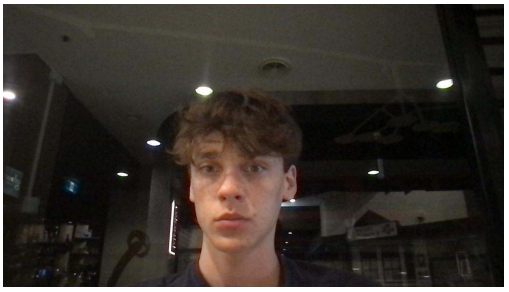
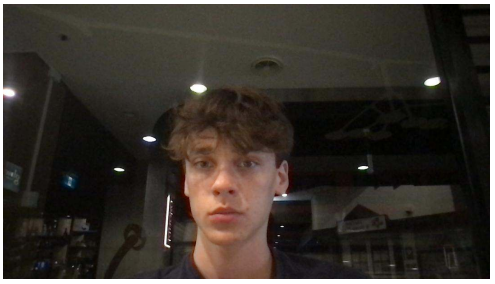
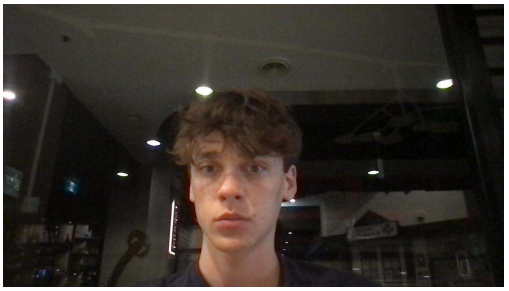
200,000		
250,000		
300,000		

Appendix X — quality vs target size for 'viewOfLabyrinth'



Appendix Y — all target sizes for 'webcam'

Size (bytes)	JFIF	AVIF
50,000		
100,000		
150,000		
200,000		

250,000		
300,000		

Appendix Z — quality vs target size for 'webcam'