Project 1: ASCII Art Conversion

David Aparco Cardenas
Rosa Yuliana Gabriela Paccotacya Yanque
Rolan Alexander Valle Rey Sánchez

***Abstract*—An ASCII art can be described as a set of characters, defined by the ASCII standard, arranged as a bidimensional matrix that aims to reproduce an image. ASCII Art stemmed from the lack of graphics ability from early printers, and nowadays it is commonly used to represent pseudo gray-scale images in text-based messages. In the present work, we describe an ASCII Art pipeline to obtain an effective representation of an input image in grayscale and color using only characters of the ASCII standard. This pipeline is comprised of a procedure of either image upsampling or downsampling as required, which is carried out with our own implementation of image convolution using several filters like the median filter. Afterward, each pixel of the resultant image is compared with ASCII characters based on its density in order to find the most effective match. This procedure is performed aiming to obtain the best representation of the original image.**

I. Introduction

An *ASCII art* can be defined as a set of characters arranged as a bidimensional matrix in a way that it achieves to represent the original image texture effectively. Many approaches have been proposed to accomplish this task using different techniques and each one with particular constraints. For instance, O'Grady et al. [1] propose to treat automatic ASCII art conversion of binary images as an optimization problem while Takeuchi et al. [2] address this problem inspired by local exhaustive search in order to optimize binary images for printing.

The main objective of ASCII art conversion is to facilitate printing in cases when printers do not have enough graphics ability and also as a mean of communication as an alternative to graphics in situations when the latter is unfeasible. ASCII art can be roughly classified into two broad categories: the tone-based ASCII art and the structure-based ASCII art [3]. In the tone-based ASCII art, a gray-scale image is converted into a matrix of ASCII characters with the intention of reproducing the intensity levels. The standard procedure involves partition of the grayscale image into blocks of character size, subsequently, a character is attributed to each block such that the intensity level of that region is preserved. On the other side, the structure-based ASCII art consists of converting an original grayscale image into a matrix of ASCII characters with the objective of reproducing the shapes of the original image. For this purpose, a character is attributed to each block aiming to preserve the shape of the block [1].

In the present work, we explore the tone-based ASCII art aiming to represent an image through a subset of ASCII characters. This representation must preserve the distribution of pixel intensities from the original image allowing the human eye to perceive almost the same information than in the original image. Furthermore, we will extract color information from the original image in order to present a color ASCII art.

Firstly, we will propose several methodologies to address the problem of implementing the image convolution with a given kernel. We will assess the performance of each methodology by measuring their execution time and analyzing the convoluted image output. Moreover, since we require that the dimensions of the convoluted image coincides with the dimensions of the original image, we will also need to address the problem of handling image borders.

Next, we will propose a pipeline to produce ASCII art from either a gray-scale or color image. First, we give as input the desired dimensions in terms of the number of characters of the ASCII art image that will be output. Then, with this information in mind, we will proceed to perform the up-sampling or down-sampling of the original image as required.

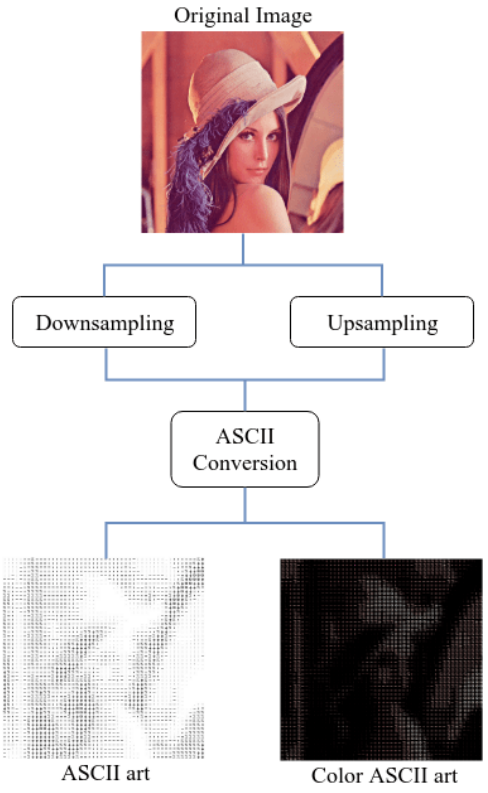


Fig. 1: Basic flowchart for the ASCII art pipeline

Afterward, we will end up with an image with dimensions equal to the dimensions specified in terms of the number of

RA: 230293 - d230293@dac.unicamp.br
RA: 263068 - r263068@dac.unicamp.br
RA: 230254 - r230254@dac.unicamp.br

characters at the beginning. The next step in this pipeline is to map an ASCII character from a subset of ASCII characters called *alphabet* to each pixel in the resized image. This mapping will be carried out taking into account the intensity of each pixel and the density of each ASCII character within the alphabet when it is projected into a bidimensional image.

The basic flowchart of this procedure is show in figure 1.

The code used in this project was implemented using Python 3.7.3. Table I summarizes the libraries utilized in this work and its respective versions.

Library	Version
Python	3.7.3
NumPy	1.17.0
OpenCV-Python	4.1.1-dev

TABLE I: Python libraries used in this project.

The present report is organized as follows. First, a brief introduction was done in this section. The section II presents our implementation of image convolution and the alternative approaches we propose in order to reduce the high computational and time complexity the traditional algorithm incurs. Moreover, some experiments were carried with the objective of comparing the proposed techniques. Next, in section III, the proposed pipeline to convert an image into an ASCII art is presented. Finally, Experiments and Conclusions are given.

II. Convolution

Convolution can be defined as the process of applying a filter. The procedure of *linear filtering* is *shift-invariant* — implicating that the value of the output relies on the pattern within an image neighborhood rather than on the position of the neighborhood — and *linear* — meaning that the output for the sum of two images is equivalent to the sum of separated outputs for the images. The pattern of weights used to perform a linear filtering is usually referred to as a *kernel* of the filter [4].

Let $\mathcal{H}$ be a filter kernel and $\mathcal{R}$ the result of the convolution of the kernel with image $\mathcal{F}$. The i, jth component of $\mathcal{R}$ is given by

$$\mathcal{R}_{ij} = \sum_{u,v} \mathcal{H}_{i-u,j-v}\mathcal{F}_{u,v} \tag{1}$$

In other words, "we can say that $\mathcal{H}$ has been convolved with $\mathcal{F}$ to yield $\mathcal{R}$" [4].

```
function CONVOLUTION(I, K)
  R ← ZEROS(h, w)
  for i ← n - 1 to h - 1 do
    for j ← m - 1 to w - 1 do
      a ← 0
      for k ← 0 to n - 1 do
        for l ← m - 1 do
          a ← a + I[i-k, j-l]K[k, l]
      R[i - n + 1, j - m + 1] ← a
  return R
```

Algorithm 1: 2D convolution between an image and a kernel.

Algorithm 1 shows the algorithm for a classical implementation of the image convolution with four nested loops. Although easy to implement, this algorithm can be highly expensive in terms of computer power and time. This issue is more evident when the dimensions of either the image or the kernel are high. Figure 2 shows the process of image convolution, when a kernel is applied to a pixel neighborhood producing an output pixel.

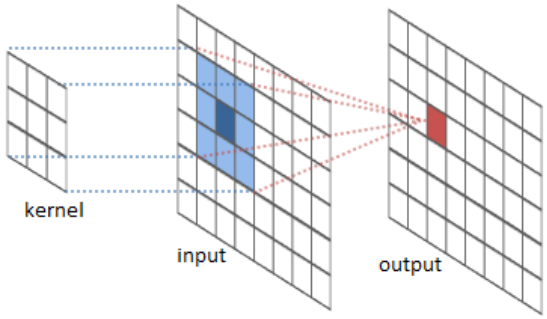


Fig. 2: 2D convolution. Reprinted from [6]

A. Image convolution techniques

In order to propose a solution to this issue, we introduce three different approaches that pursue to optimize this task. In the following algorithms, the input parameters are I and K which refers to the original input image with handled borders and the kernel of the filter respectively. Other general variables we can also find are h and w, which represent the height and width of I respectively; n, m, which refers to the height and width of the kernel respectively. $\mathcal{R}$ represents the convolved image output which is returned once the algorithm has completed its execution.

1) Vectorized element-wise product: In this approach we leverage the vectorized operations that NumPy provides to reduce dramatically the execution time of the image convolution. As we can see in Algorith 2, we iterate over all pixels of I and proceed to apply a vectorized element-wise product between the current pixel neighborhood and the kernel. Afterward, we sum all values of the resultant product and we store it in $\mathcal{R}$.

```
function VECTORIZED_CONVOLUTION(I, K)
  R ← ZEROS(h, w)
  for i ← 0 to h - n + 1 do
    for j ← 0 to w - m + 1 do
      a ← np.sum(np.multiply(I[i:i+n][j:j+m]),K)
      R[i, j] ← a
  return R
```

Algorithm 2: 2D convolution using vectorized element-wise products.

2) Batch matrix multiplication: This approach aims to reduce the number of matrix multiplications that is performed in the vectorized element-wise product. For this purpose we will store each pixel neighborhood, in its vector form, in an array. Therefore, we will end up with a matrix where each row represents each pixel neighborhood and the columns represents the number of elements within the pixel's neighborhood, that is, the number of kernel elements. The size of this matrix

is $BSIZE \times |\mathrm{K}|$. Hence, the number of multiplications is reduced by a factor of $BSIZE$ reducing the computational time of the algorithm.

```
function BATCH_MATRIX_MULTIPLICATION(I, K, BSIZE)
  R ← []
  aa ← []
  bb ← K.reshape((n*m,1))
  for i ← 0 to h - n + 1 do
    for j ← 0 to w - m + 1 do
      aa.append(I[i:i+n,j:j+m].reshape(n*m))
      if len(aa) == BSIZE do
        mm ← np.matmul((aa,bb))
        R ← np.concatenate((R,mm),axis=None)
        aa ← []
  mm ← np.matmul(aa, bb)
  R ← np.concatenate((R,mm),axis=None)
  R ← R.reshape((h-n+1,w-m+1))
  return R
```

Algorithm 3: 2D convolution using batch matrix multiplication.

3) Dynamic element-wise product: This algorithm aims to reduce the time complexity of the traditional image convolution algorithm by using the previous multiplications that were realized with previous pixels. The procedure of multiplying element-wise the kernel a given pixel neighborhood can be optimized by using the previous multiplications that were carried out with neighbor pixels. We can take advantage of the previous multiplications by storing the column-wise sums in an array. This will lead to a reduction of number of multiplications that need to be performed for each pixel. Therefore, we will only be required to compute the multiplication of the last column of the kernel.

A drawback of this algorithm is that only works with kernels with some constraints. The kernel must have either all columns or all rows equal. A good application for this approach are image convolutions with a mean filter, since all elements within the kernel are equal.

```
function DYNAMIC_CONVOLUTION(I, K)
  R ← ZEROS(h, w)
  for i ← 0 to h - n + 1 do
    S ← ZEROS(n,1)
    for j ← 0 to w - m + 1 do
      if j == 0 do
        S ← np.sum(np.multiply(I[i:i+n,j:j+m],K),0)
      else do
        S ← np.delete(S, 0)
        S ← np.append(S, np.dot(I[i:i+n,j+m-1],
    K[:,m-1]))
      R[i,j] ← np.sum(S)
  return R
```

Algorithm 4: 2D convolution using dynamic element-wise product.

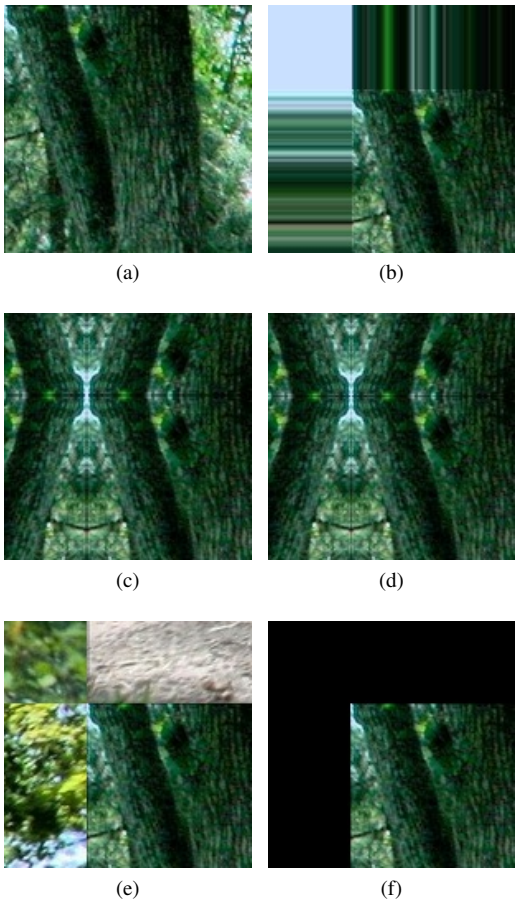
B. Border Handling

The resultant output image after performing convolution must have the same dimensions than the original image according to the project requirements. For this purpose, we explore five techniques to handle image borders with this objective in mind.

Figure 3a shows the left corner of an original image while 3(b)-(f) show the left corner of the same image with handled borders using the following techniques: replicate, reflect, reflect 101, wrap and constant m, with m equal a zero, respectively.

Technique	Example of padded pixels		
	Pad	Image	Pad
Replicate	a a a a	a b c d e f	f f f f
Reflect	d c b a	a b c d e f	f e d c
Reflect 101	e d c b	a b c d e f	e d c b
Wrap	c d e f	a b c d e f	a b c d
Constant m	m m m m	a b c d e f	m m m m

TABLE II: Border handling techniques



(a) (b) (c) (d) (e) (f)

Fig. 3: Border handling techniques. (a) original image, (b) replicate, (c) reflect, (d) reflect 101, (e) wrap, and (f) constant

From this figure we can see that the techniques that best preserve the global information of the image in the borders are reflect and reflect 101 (figure 3(c)-(d)). Also, libraries like OpenCV use this border handling technique as default for their filter operations.

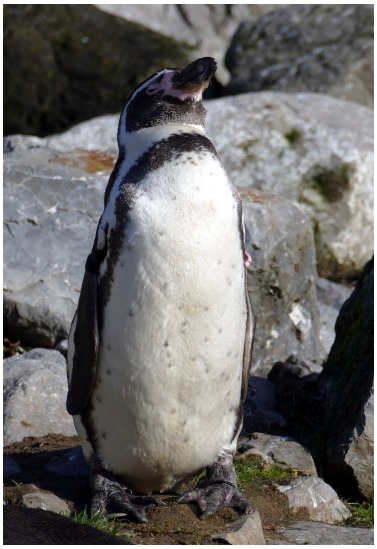
to choose for subsequently applying convolution to the image is reflect and reflect 101, since

C. Experiments

Since the input image is represented as a set of discrete pixels, we will use discrete convolution kernels in the presents work.

We will explore the convolution on the images shown in 6. This images were chosen due to the variety of textures and colors they represent within their boundary. Another factor of choosing this images is because of the variety of dimensions they have.

(a) Kids



(b) Penguin

(c) Lion

Fig. 4: Original images. The dimensions of the images are (a) 530×128, (b) 1316×1920, and (c) 423×640.

Since our goal in this experiment is to measure the effectiveness of the algorithms we implemented in comparison with the convolution function provided by OpenCV we will use the mean filter for this purpose due its ease of generation.

We can create a mean kernel of size $n \times n$ easily using NumPy by the following command:

```
mean_kernel = np.ones((n, n),dtype=np.float32)/(n*n)
```

Algorithm 5: Creation of a mean kernel of size n.

The convolution of color images is performed by convoluting each channel and then stacking them up as a color image again.

The following tables summarize the execution times for the different methods proposed in this section for different sizes of mean filters. The methods we use are classical algorithm (CA), vectorized element-wise product (VEP), batch matrix multiplication (BMM), dynamic element-wise product (DEP), and OpenCV.

(a)

(b)

(c)

Fig. 5: Results of image convolution on *Kids* using the mean filter of sizes (a) 3×3, (b) 5×5, and (c) 15×15.

Method	**3×3 (s)**	**7×7 (s)**	**15×15 (s)**
CA	16.1587	76.9876	373.8280
VEP	12.5822	13.2467	15.1602
BMM	3.9368	4.4151	5.8124
DEP	4.8271	4.8968	4.6723
OpenCV	0.0364	0.0442	0.2912

TABLE III: Execution times of image convolution on *Kids* (530×1280) picture using mean filters of different sizes.

(a) (b) (c)

Fig. 6: Results of image convolution on *Penguin* using the mean filter of sizes (a) 3×3, (b) 5×5, and (c) 15×15.

From these results, we can see that the fastest method is OpenCV's convolution in all the scenarios. From our implementations, we can conclude that the fastest algorithms

Method	3×3 (s)	7×7 (s)	15×15 (s)
CA	62.9441	363.4438	1631.1801
VEP	48.8443	68.5162	72.7996
BMM	16.7851	20.4449	26.6859
DEP	4.9222	6.2870	5.5230
OpenCV	0.1541	0.2031	0.4349

TABLE IV: Execution times of image convolution on the *Penguin* (1920×1360) picture using mean filters of different sizes.

(a) (b) (c)

Fig. 7: Results of image convolution on *Lion* using the mean filter of sizes (a) 3×3, (b) 5×5, and (c) 15×15.

Method	3×3 (s)	7×7 (s)	15×15 (s)
CA	8.4613	40.5164	180.3480
VEP	7.7034	6.3626	8.8895
BMM	2.1436	2.5935	3.3216
DEP	6.8240	5.4045	7.1882
OpenCV	0.0164	0.0194	0.0547

TABLE V: Execution times of image convolution on the *Lion* (530×1280) picture using mean filters of different sizes.

for small images are batch matrix multiplication (BMM) and vectorized element-wise product (VEP), since these algorithms takes advantage of the optimized functions for matrix multiplication that NumPy provides, also it tries to reduce the number of such multiplications giving a really fast algorithm. On the other hand, we see that the dynamic element-wise product algorithm works really well when the image has large dimensions.

In these experiments we used the replicate 101 technique for border handling, since they preserve the visual continuity of the image by mirroring regions closest to the borders.

From now on, we will use the vectorized element-wise product (VEP) in all the convolutions since it provides better execution times.

III. ASCII Art Pipeline

A. General Algorithm

Our proposed ASCII art conversion algorithm begins by receiving a grayscale or color image along with the desired dimensions in terms of number of characters and our alphabet. For simplifying purposes, we are considering only the desired width as input in order to keep the dimensional ratio from the original image.

Once we have the information regarding the width of our ASCII art image in terms of the number of characters we proceed with either the upsampling or downsampling of the image. This procedure aims to obtain an image with width dimension equal to the number of characters given as input. In other words, the number of pixels that define the width of the sampled image must be equal to the number of characters width we entered as input.

Next, we will proceed to compare each pixel of our sampled image with the characters in our alphabet. In order to proceed with the comparison we need to know information about our characters, that is, which character can represent better a given pixel intensity. For this reason, we first need to order the characters in terms of density. Hence, the character with the lowest density will be mapped to the pixel of biggest density, and so on. This procedure will be performed in each pixel of our sampled image resulting with an ASCII art representation of our original image with the desired dimensions entered as input.

In order to save the generated ASCII art as image we proceed to draw the characters over a grid of white pixels of fixed size. Then, this "small" images are joined together into our final ASCII art image. On the other hand, for a color ASCII art image we proceed to draw the characters over a grid of black pixels of fixed size. Next, we attribute to the character the color of the pixel from it was mapped.

In the case of a color ASCII art image the mapping of characters to pixels is performed in the opposite way than in the binary ASCII art image, that is, characters with high density will be mapped to pixels with high density and so on,due to the background in color ASCI art image is black.

A general view of this algorithm is presented in Algorithm 6.

```
function ASCIIFY(I, w, ∑)
    equalize_image(I)
    if I.w < w do
        I* ← image_upsampling(I, w)
    elif I.w > w do
        I* ← image_downsampling(I, w)

    ∑* = rank_alphabet_on_density(∑)

    A = []
    for each pixel p in I* do
        A.append(map_character_to_pixel(∑*, p))

    AI = create_image(w)
    for each character c in A do
        AI.append(draw_character(c))

    return AI
```

Algorithm 6: General ASCII art conversion algorithm.

B. Methodology

1) Upsampling and Downsampling: The methodology to upscale and downscale an image is presented in Figure 8. In short, to upscale the image, first we resize the image duplicating the columns and rows until complete the desired size, then a filter is applied. In this work we explored the *median filter*, *bilinear interpolation* and the *Gaussian filter*. To downscale the process, a similar procedure is followed, but in the reverse way, first we apply a filter, specifically the Gaussian filter to avoid aliasing, then the resizing is performed removing rows and columns.

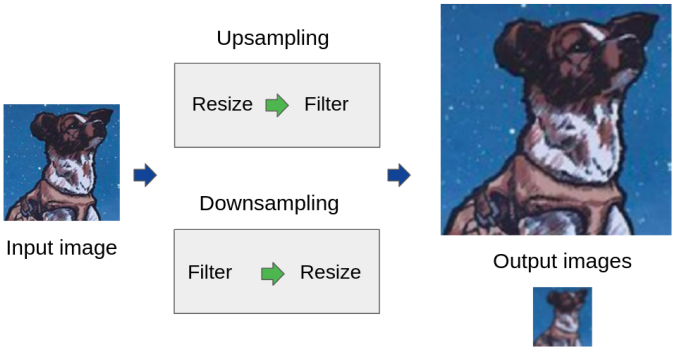


Fig. 8: Methodology to upscale and downscale an image

The filters used to perform image upscaling and downscaling are presented as follows. The first two filters were used applying convolution. The size of the filter kernels are computed based on the ratio scale.

- *Bilinear interpolation*. The bilinear interpolation is the most common method used for upsampling, this filter assigns to the pixel the weighted average sum of their nearest known pixels.For example,

$$\frac{1}{3} * \begin{bmatrix} 0.25 & 0.5 & 0.25 \\ 0.5 & 1 & 0.5 \\ 0.25 & 0.5 & 0.25 \end{bmatrix}$$

is a kernel when doubling the size of a image, and it is generated when a multiplication of their base form and its transpose is done:

$$\frac{1}{3} * \begin{bmatrix} 0.5 \\ 1 \\ 0.5 \end{bmatrix} \begin{bmatrix} 0.5 & 1 & 0.5 \end{bmatrix}$$

Similarly, when we want to triple the size the base form of the kernel is:

$$\begin{bmatrix} 1/3 & 2/3 & 3 & 2/3 & 1/3 \end{bmatrix}$$

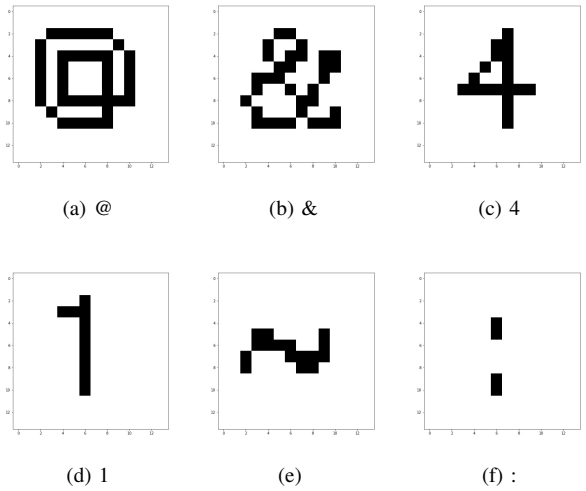
The Bilinear Interpolation is the method by default that OpenCV uses to resize images.

- *Gaussian filter*. The Gaussian filter is a low-pass filtering. In our implementation, the variance for the Gaussian kernel is always 1, and the kernel size is calculated according to the ratio scale when upsampling or downsampling the image. Specifically for downsamplig the literature suggest that the size of the kernel should double for each $1/2$ size reduction. The following equation defines the Gaussian filter.

$$G_\sigma(x, y) = \frac{1}{2\pi\sigma^2} \exp - \frac{x^2 + y^2}{2\sigma^2}$$

- *Median filter*. The median filter assigns to a pixel the middle value of their nearest known pixels. The filter size is according to the ratio scale.

2) Ranking of ASCII characters and mapping: The ranking of ASCII characters is critical step in our algorithm since from this information we attribute a character to each pixel based on its intensity. This procedure works by drawing a character in a grid of white pixels of dimensions 14×14 and counting the number of black pixels within its boundary, the density of a character is based on this number. In other words, the character with most black pixels in a 14×14 regions is the most dense. Figure 9 shows several characters projected into a grid of 14×14 being the character in (a) the most dense and (f) the least dense.



(a) @ (b) & (c) 4

(d) 1 (e) (f) :

Fig. 9: Characters projected into a grid of 14×14.

Next, we will divide the whole range of pixel intensities into equal size intervals. The number of intervals is given by the number of characters. For instance, in figure 9 we show the pixel intensities range from 0 to 255 which has been divided into 5 equal size intervals, in accordance with the number of characters of the alphabet. Therefore, the size of each interval is $255/5 = 51$. Then, for a new pixel we determine in which interval its intensity will fall and we assign the correspondent character in that interval.

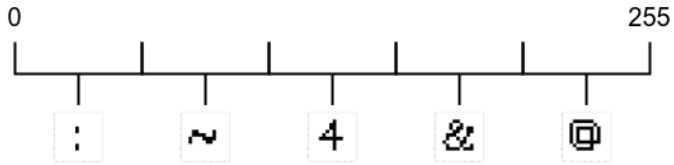


Fig. 10: Equal size intervals in the range of pixel intensities from 0 to 255 (Characters are ordered based on its density).

The aforementioned procedure is carried out on each pixel in the resized image. Figure 11 shows this methodology in action.

Since that the mapping to characters is done in the range from 0 to 255, we need to have the pixels in the image in that range too, otherwise some characters from the alphabet given will not used leading to not the best possible visualization. So, before resizing the image, an histogram equalization for all the channels in the input image is done.

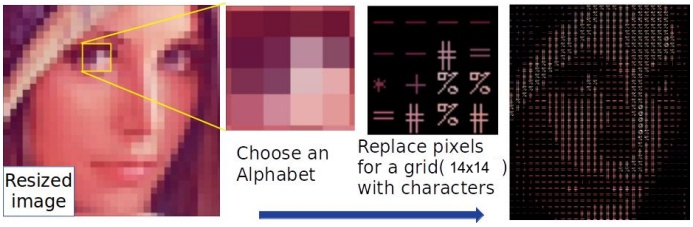
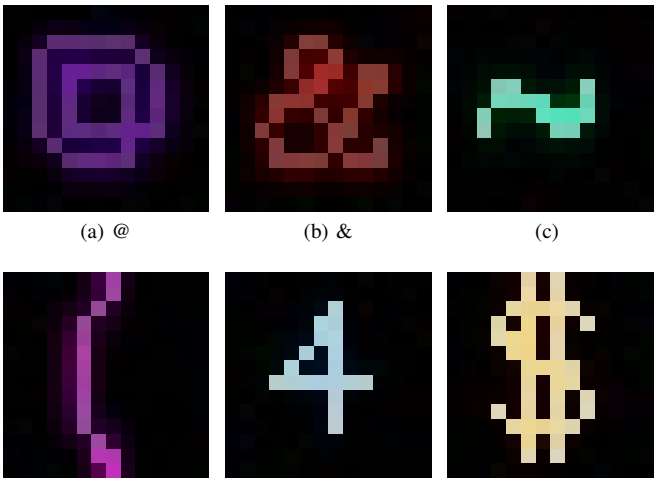


Fig. 11: Mapping of pixels to characters.

C. Construction of ASCII art images

Once the characters have been mapped to the pixels we proceed to construct the ASCII art images. These step is performed by "drawing" each character in a 14×14 white or black grid of pixels either for binary or color ASCII art respectively. For binary images we simply use the characters represented in figure 9 while for color images we use the characters shown in figure 12.



(a) @ (b) & (c)

(d) ((e) 4 (f) $

Fig. 12: Color characters projected into a grid of 14×14.

These figures are appended together, forming in this way the final ASCII art image.

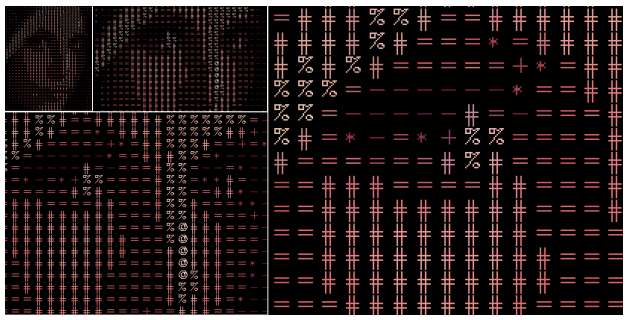


Fig. 13: Color character figures being appended.

Figure 13 shows how characters are appended together to obtain the final ASCII art image.

The final color and binary ASCII art images are shown in figure 14. The results in quality vary depending on the dimensions given in terms of the number of characters. If we give as input a large width, we may end up with an image with better visual quality but a drawback arises in terms of the size of the resultant ASCII art image.

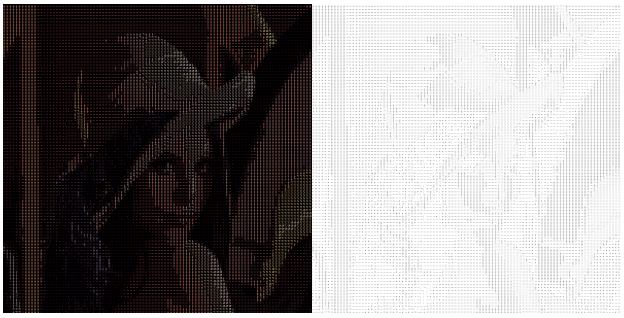


Fig. 14: Color and binary ASCII art images.

Therefore it is important to consider an effective size as width, having a balance between visual quality and size of the image.

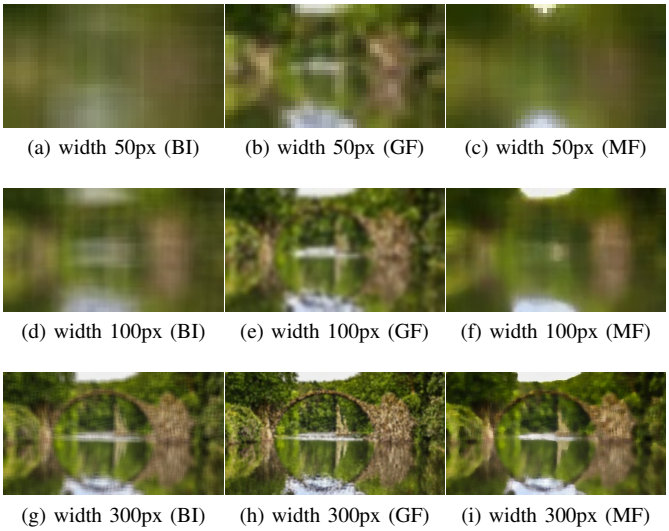
The next section presents the experiments carried out regarding the upsampling and downsampling of images and the conversion of natural images to ASCII art.

IV. Experiments and Discussion of Results

This sections shows the experiments that were performed aiming to explore and analyze the results of the conversion of images to ASCII images for different images.

A. Downsampling and upsampling of images

In this section some experiments regarding upsampling and downsampling were performed.



(a) width 50px (BI) (b) width 50px (GF) (c) width 50px (MF)

(d) width 100px (BI) (e) width 100px (GF) (f) width 100px (MF)

(g) width 300px (BI) (h) width 300px (GF) (i) width 300px (MF)

Fig. 15: Downsampling of an image of dimensions 1920×1080. First columns shows the downsampling performed using bilinear interpolation (BI), second column using the Gaussian filter G (GF) and third column using the median filter (MF).

Figure 15 shows how our downsampling implementation performs with different filters and different sizes. Here, we can notice that the usage of median filter and bilinear interpolation

leads to aliasing in the resulting image. Hence, for image downsampling is strongly suggested to apply the Gaussian filter before resizing.

Figure 16 follows the aforementioned procedure with different filters and different sizes. In one hand, the usage of bilinear interpolation leads to a light smoothed image, without noticing a pixelated image. On the other hand, with the Gaussian Filter the upscaled image contains more details looking a little pixelated. Then, with the median filter the results looks as if the image were cartoonized. Therefore, due to the objective of this project and because we map each pixel to an ASCII character, we choose to perform the upsampling of images with the Gaussian filter from now on, in order to preserve the details of the original image.

(a) width 300px (BI) (b) width 300px (GF) (c) width 300px (MF)

(d) width 500px (BI) (e) width 500px (GF) (f) width 500px (MF)

Fig. 16: Upsampling of an image of dimensions 100×100. First columns shows the downsampling performed using bilinear interpolation (BI), second column using the Gaussian filter G (GF) and third column using the median filter (MF).

B. ASCII art conversion

In this section we present the results of our experiments regarding ASCII art conversion of natural images. We discuss the size-variation and alphabet-variation experiments carried out during the experimental procedure.

For this purpose, we use 4 alphabets, each one comprised of different number of characters. We present them in table VI:

Alphabet	**Characters**
Alphabet 1	, ” ! 1 ¡ = V U 5 R B W @
Alphabet 2	whitespace . , ‘ ! \ 1 ¡ (= V 4 2 E 0 R B W & # @
Alphabet 3	# @ % = * : - . whitespace
Alphabet 4	C O M P U T E R V I S N F o c i m p .

TABLE VI: Alphabets used for the experiments in this project.

This alphabets were chosen from the set of ASCII characters shown in table VII. We constructed the alphabets selecting characters of different densities, considering numbers, uppercase letters, lowercase letters, and special symbols.

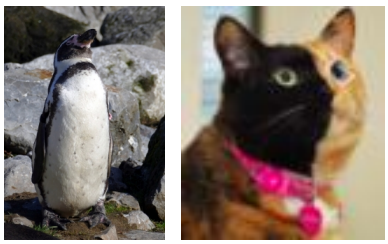
Printable ASCII codes of characters				
32 space ←	51 ← 3	70 ← F	89 ← Y	108 ← l
33 ← !	52 ← 4	71 ← G	90 ← Z	109 ← m
34 ← ”	53 ← 5	72 ← H	91 ← [	110 ← n
35 ← #	54 ← 6	73 ← I	92 ← \	111 ← o
36 ← $	55 ← 7	74 ← J	93 ←]	112 ← p
37 ← %	56 ← 8	75 ← K	94 ← ˆ	113 ← q
38 ← &	57 ← 9	76 ← L	95 ← _	114 ← r
39 ← ’	58 ← :	77 ← M	96 ← ‘	115 ← s
40 ← (	59 ← ;	78 ← N	97 ← a	116 ← t
41 ←)	60 ← <	79 ← O	98 ← b	117 ← u
42 ← *	61 ← =	80 ← P	99 ← c	118 ← v
43 ← +	62 ← >	81 ← Q	100 ← d	119 ← w
44 ← ,	63 ← ?	82 ← R	101 ← e	120 ← x
45 ← -	64 ← @	83 ← S	102 ← f	121 ← y
46 ← .	65 ← A	84 ← T	103 ← g	122 ← z
47 ← /	66 ← B	85 ← U	104 ← h	123 ← {
48 ← 0	67 ← C	86 ← V	105 ← i	124 ← —
49 ← 1	68 ← D	87 ← W	106 ← j	125 ← }
50 ← 2	69 ← E	88 ← X	107 ← k	126 ← ∼

TABLE VII: ASCII characters

We use the images shown in figure to perform the subsequent experiments.

(a) 200 × 237 (b) 128 × 128

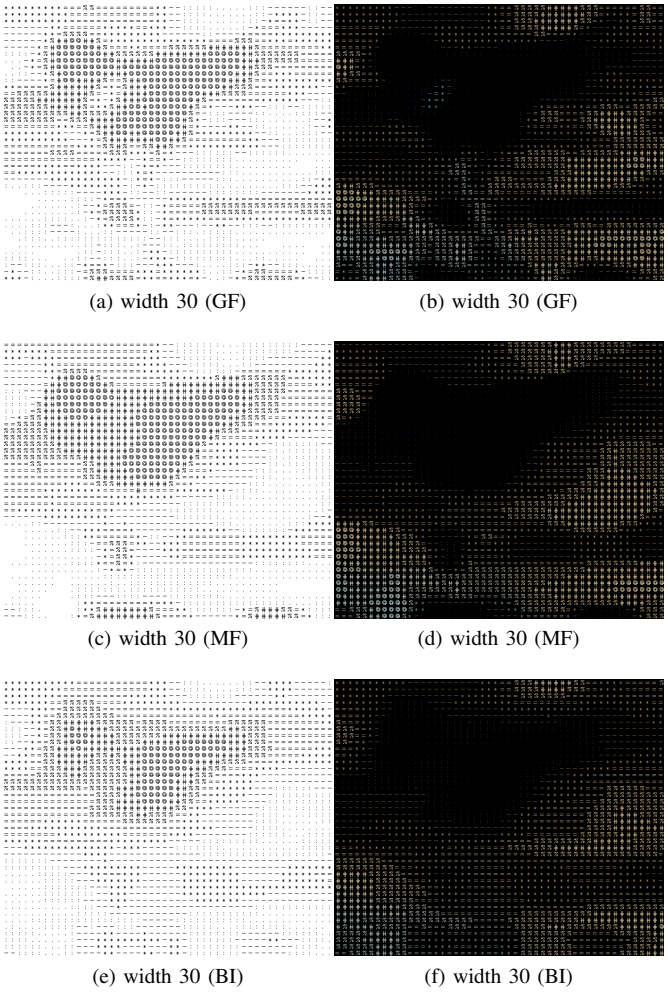
(c) 100 × 100



(d) 336 × 230 (e) 86 × 75

Fig. 17: Images used for the ASCII art conversion experiments with their respective dimensions.

1) Size-variation experiments: In this section we performed several experiments comparing several aspects from our implementation, we tested three filters to do the upsampling or downsampling as required: Gaussian filter, median filter and bilinear interpolation. Also we experimented with the following width size for all images: 50, 90, 150, and 210. In this experiment the alphabet 3 was used.



(a) width 30 (GF) (b) width 30 (GF)

(c) width 30 (MF) (d) width 30 (MF)

(e) width 30 (BI) (f) width 30 (BI)

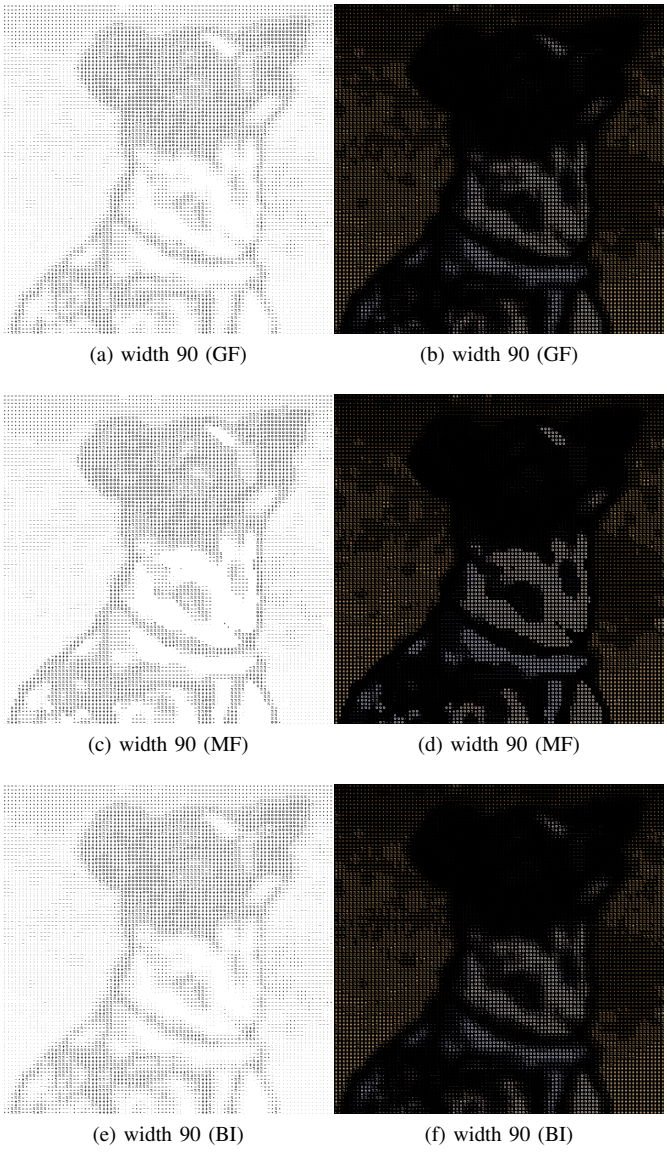
Fig. 18: ASCII art images with alphabet 3 from figure 17a with width of 30 characters (Downsampling). (a)-(b) Gaussian filter, (c)-(d) median filter, and (e)-(f) bilinear interpolation.

From figure 18, we can see that for downsampling purposes bilinear interpolation performs poor and the Gaussian filter is the best one, showing more details in the pictures.

From figure 19, it can be seen that the Gaussian filter is still performing better than its colleagues. Notice that when nedian filter is applied, the figure seems to be cartoonized therefore the change in tones is sharp. (See how different is Laika's chest in the middle row compared with the others).

As discussed before the Gaussian Filter for downsampling is the best option.Also, it can be seen from figures 20 and 21 in this experiment that as we increase the dimensions of the ASCII art images, the details can be seen more clearly.

From Figures 20 and 21, in upsampling we noticed that ASCII Art with bilinear interpolation and Gaussian filter are similar, however Gaussian filter shows more details slightly,



(a) width 90 (GF) (b) width 90 (GF)

(c) width 90 (MF) (d) width 90 (MF)

(e) width 90 (BI) (f) width 90 (BI)

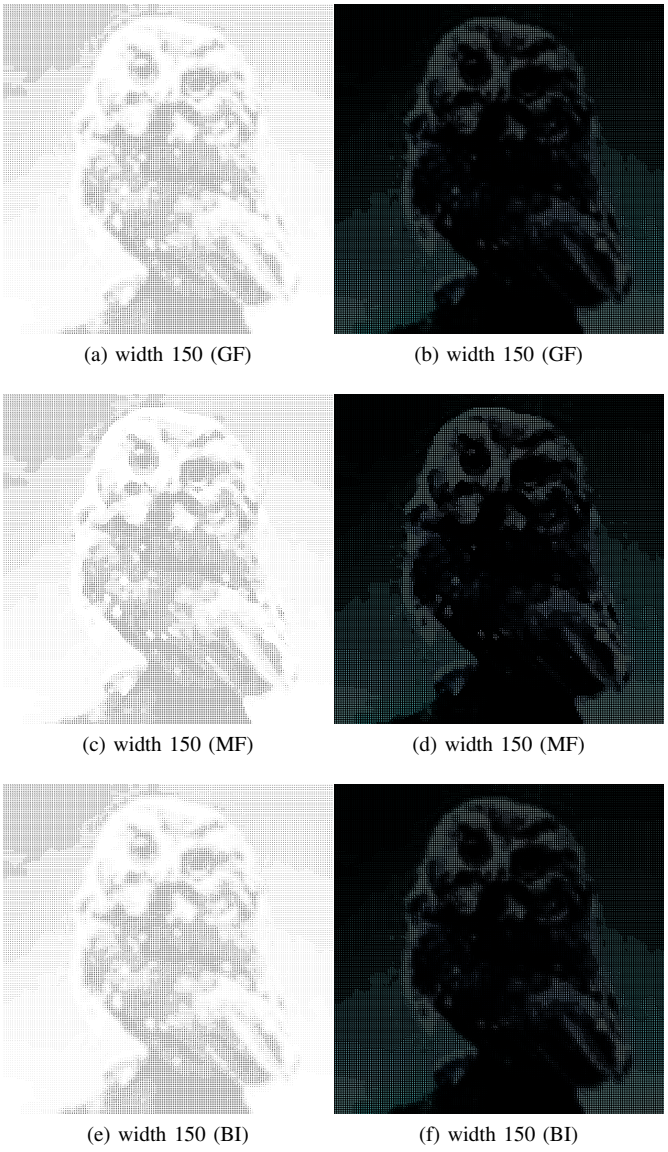
Fig. 19: ASCII art images from figure 17b (Laika) with alphabet 3 and width of 90 characters (Downsampling). (a)-(b) Gaussian filter, (c)-(d) median filter, and (e)-(f) bilinear interpolation.

as occurred when downsamplig. In the same way, the median filter seems to be cartooning the image, showing sharp changes in the characters intensity.

To follow the figures in the repository we use the naming convention suggested by the professor. For instance, if one image is named 0-4-a-2-a-1.png, then the third character points out to the image, the fourth points out to the size, the fifth points out to the kernel, and lastly the sixth points out if the image is binary or colored.

2) Alphabet-variation experiments: In this experiment we test the four aforementioned alphabets from table VI with the images shown in figure 17. We also test width lengths of 50 and 200 characters.

In this experiment we only use the Gaussian filter to perform



(a) width 150 (GF) (b) width 150 (GF)

(c) width 150 (MF) (d) width 150 (MF)

(e) width 150 (BI) (f) width 150 (BI)

Fig. 20: ASCII art images from figure 17d with width of 150 characters (Upsampling). (a)-(b) Gaussian filter, (c)-(d) median filter, and (e)-(f) bilinear interpolation.

(a) width 210 (GF) (b) width 210 (GF)

(c) width 210 (MF) (d) width 210 (MF)

(e) width 210 (BI) (f) width 210 (BI)

Fig. 21: ASCII art images from figure 17d with width of 210 characters (Upsampling). (a)-(b) Gaussian filter, (c)-(d) median filter, and (e)-(f) bilinear interpolation.

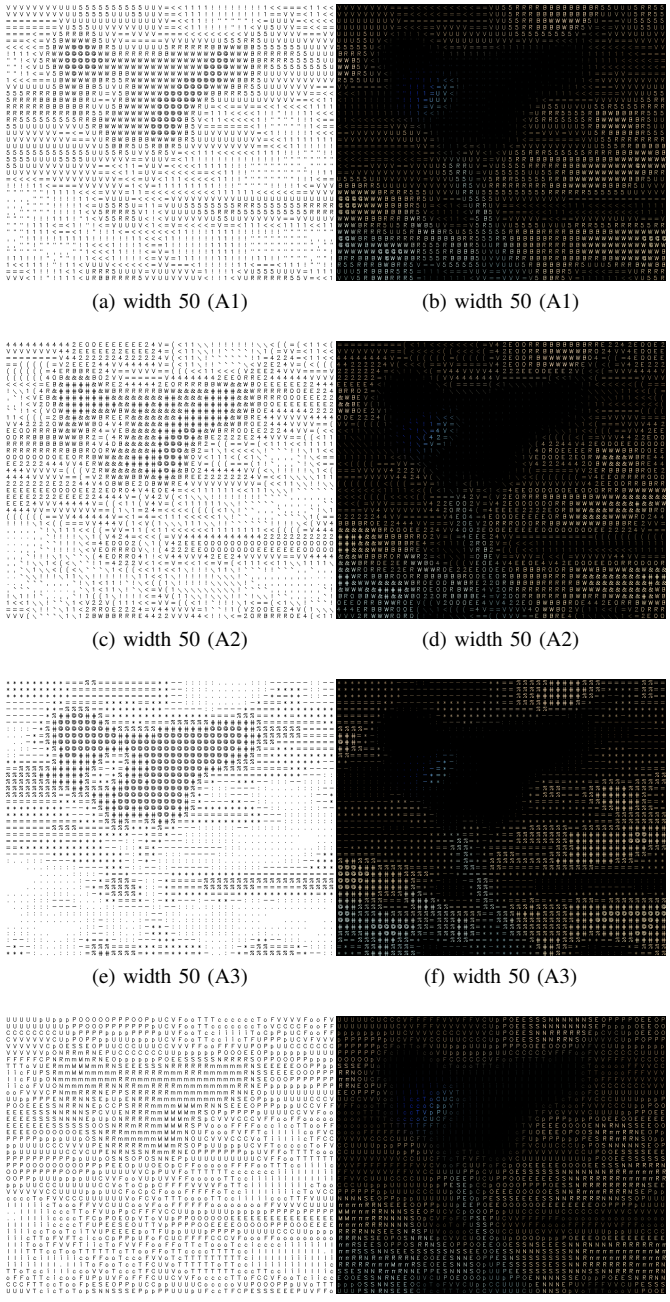
the upsampling and downsampling.

From these figures 22 and 23, we can see that Alphabet 3 gives as output the best visual quality images, while Alphabet 4 is the one the output ASCII art of poor quality. It can be due to the variety of symbols it contains from different densities. Therefore we can see that as our characters in our alphabet have more variety in density then the visual result of the ASCII art image will be better. In Alphabet 4, several characters have almost the same density, for instance the uppercase letters are responsible for the poor quality of the ASCII art images for Alphabet 4 in both cases.

To follow the figures in the repository we use the naming convention suggested by the professor. For instance, if one image is named 0-4-a-0-a-0.png, then the third character points out to the image, the fourth points out to the alphabet, the fifth points out to the size, and lastly the sixth points out if the image is binary or colored.

V. Conclusions

In this project, we achieved to represent an image in ASCII art, with different sizes and alphabets. The convolution operation was an important part for the project since it was used when resizing an image, affecting the run-time. In view of this, we implemented and compared different approaches with different computational costs. Similarly, for resizing a comparison between different filters to smooth the image was done. Likewise, for experiments transforming to the ASCII



(a) width 50 (A1) (b) width 50 (A1)

(c) width 50 (A2) (d) width 50 (A2)

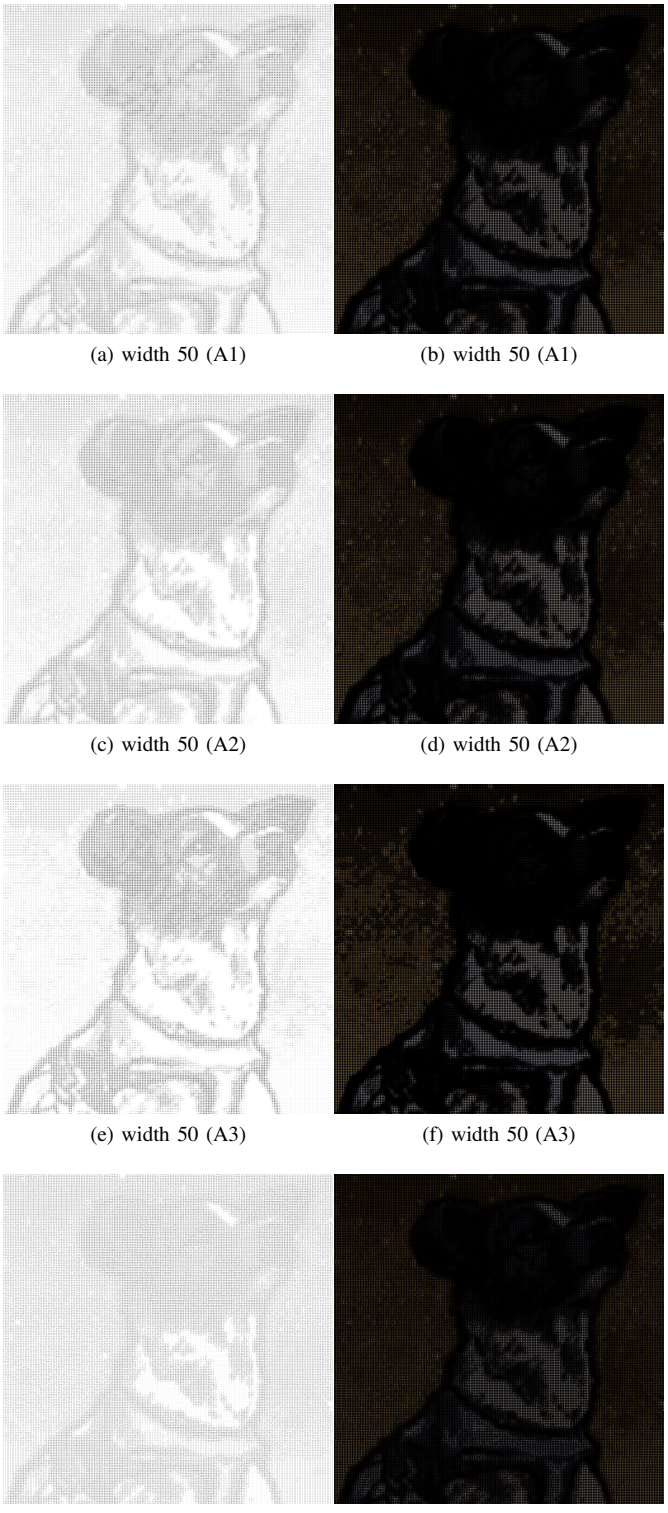
(e) width 50 (A3) (f) width 50 (A3)

(g) width 50 (A4) (h) width 50 (A4)

Fig. 22: ASCII art images from figure 17a with width of 50 characters (Downsampling). (a)-(b) Alphabet 1, (c)-(d) Alphabet 2, (e)-(f) Alphabet 3, and (g)-(h) Alphabet 4.

art four suitable alphabets were chosen, being the best the one suggested by the professor.

We conclude that the use of a quickly convolution algorithm is critical for the purposes of upsampling and downsampling of images. The Gaussian filter gave the best results in these tasks. Also, the alphabet that gave the best results was the Alphabet 3 proposed by the professor.



(a) width 50 (A1) (b) width 50 (A1)

(c) width 50 (A2) (d) width 50 (A2)

(e) width 50 (A3) (f) width 50 (A3)

(g) width 50 (A4) (h) width 50 (A4)

Fig. 23: ASCII art images from figure 17b with width of 200 characters (Downsampling). (a)-(b) Alphabet 1, (c)-(d) Alphabet 2, (e)-(f) Alphabet 3, and (g)-(h) Alphabet 4.

REFERENCES

[1] P. D. O'Grady and S. T. Rickard, "Automatic ASCII art conversion of binary images using non-negative constraints," IET IRish Signals and Systems Conference (ISSC 2008), pp. 186-191, 2008

[2] Y. Takeuchi, D. Takafuji, Y. Ito, and Koji Nakano, "ASCII Art Generation Using the Local Exhaustive Search on the GPU," 2013 First International Symposium on Computing and Networking, pp. 194-200, 2013

[3] X. Xu, L.Zhang, and T.-T Wong, "Structure-based ASCII art," *ACM*

Transactions on Graphics (SIGGRAPH 2010 issue), vol 29, no. 4, pp. 52:1-52:9, Jul. 2010
[4] D. A. Forsyth, J. Ponce, “Computer Vision: A Modern Approach,” 2nd Edition, Pearson, 2012
[5] L. G. C Harney, ”A Functional Approach to Border Handling in Image Processing,” 2015 International Conference on Digital Image Computing: Techniques and Applications (DICTA), 2015
[6] Bringing Parallelism to the Web with River Trail, `http://intellabs.github.io/RiverTrail/tutorial/`