POLITECNICO DI TORINO

Master degree course in Computer Engineering

Master Degree Thesis

Image raster to vector graphics: a case study of a Robot based IoT device able to manage physical contents

Supervisor
prof.ssa Marina Indri

Candidate
Yuri Gaito

Scribit
Company Supervisor
dott. ing. Andrea Bulgarelli

April 2019

Contents

Chapter 1

Introduction

The idea of a vertical plotter is not new but has existed for a long time. One of the first prototype was developed in 2001 by the VP Squared team from Cronell University [1]. As mentioned in the official website, the basic idea is to use the VP-space, a different way of representing space with respect to the Cartesian coordinates system. This plotter uses two stepper motors and a keypad to control the plotter; the marker is inserted in the centre of a small squared surface which is supported by two cables. This project was a pioneer in this field and opened doors to the implementation of new prototypes and artistic installations.

Nowadays, there is plenty of open source projects on the web to draw from, but the basic structure is the same as the one introduced by the VP Squared team, even if with some technological updates, i.e., keypad is replaced by different kinds of keyboards, such as microcontrollers or single-board computer (e.g., Raspberry Pi) to increase usability and improve functionalities.

In 2012, the first vertical plotter from *Carlo Ratti Associati* [2] presents this approach, with small design changes required for the presentation of the OSARC Manifesto (Open Source Architecture) [3].

In 2014, the prototype (both the software and the hardware part) is released as the open source project **Open Wall**; it presents a new configuration by mounting the two stepper motors inside the chassis attached to two pulleys, in which a cable is wound, thus requiring the presence of only two nails at the ends of the wall to hang up the plotter. The firmware is Arduino based and further software provides an algorithm to convert vectorial files into NC[1] instructions. Over the years, Open Wall becomes the backbone of Scribit improving electronics, mechanics and adding new features, such as new markers to support more colors, while the software remains more or less the same.

Moreover, more or less all the prototypes of vertical plotters are able to draw either on blackboards and canvases: in the first case it is possible to clean the blackboard and to rewrite it, in the second case the canvas can be used only once, it must be replaced to rewrite. Scribit wants to overcome this problem and implements an

[1]Numerical Control is the automated control of machining tools by means of computers.

automated erasing system thanks to a ceramic disk that, with increasing temperature, is able to evaporate a special water-based ink used in the stock markers supplied with the starter kit of the robot.

The idea is to have a user-friendly product that allows any user to employ the plotter without any prototyping skills. The aim is to extend the final target from makers to common people, such as artists, designers etc., and to have a commercial product.

This thesis is focused on a part of the front-end side of the software whose aim is to render any image in any format to SVG files in order to support the back-end side software, which expects as input only SVG files, making Scribit not very flexible. The main objective is to create a general purpose rendering software to then move on to a more specific goal: to use such a rendering process to get drawable SVG files, i.e., files with clear, well defined and simple contours. The developed software is JavaScript-based, a scripting object-oriented language widely used in Web, mobile and server development. It is a very flexible and easy language, it is portable and many frameworks, e.g., Angular and Ionic, are based on it or on a *typed* version of it (TypeScript).

The main contribution of this thesis work, therefore, is the developed software, which is based on a preliminary study phase. The rendering algorithms are part of a branch of computer science called **Computer vision**.

"*The goal of computer vision is to made useful decisions about real physical objects and scenes based on sensed images*"[4]. To make decisions about real objects, it is necessary to build some descriptions or models of them from the image.

Computer vision's critical issues are summarized as follows:

- **Sensing**: In which way are the images obtained? How do the images encode information such as shape, illumination and spatial relationship?
- **Encoded information**: How are informations about the 3D world yield by the images? How are objects identified?
- **Representations**: Which kind of representation should be used to store information, such as parts and properties, about the objects?
- **Algorithms**: Which are the approaches used to process the information to obtain knowledge, and to built object from real world images?

These issues and others will be discussed in Chapter 2, which describes the models, the algorithms and the representations studied and used for this thesis work, i.e., all the theoretical knowledge needed to better understand the overall work.

In particular, Section 2.1 shows which representation for the images is used, Section 2.2 to Section 2.4 describe the different algorithms involved in the process, and Section 2.5 introduces how information about contours is encoded.

Once the context and the tools are clear, Chapter 3 is aimed at describing in detail the main contribution of this thesis work, in particular it is split in proposal (Section 3.1) and implementation (Section 3.2 to Section 3.7).

Chapter 4 describes the tests that have been run to prove the validity of the proposed solution. Different kinds of test are proposed to show different scenarios and to highlight both the strength and the weakness of the proposal. At the very end, a simple benchmarking analysis is shown.

Chapter 5 summarizes the results of this thesis work, and describes possible future developments and improvements.

The following section describes the Scribit project and reports the new working paradigm of the creative industry, prototyping and design thinking, showing the functioning of the robot with particular attention to the software and the hardware.

1.1 Scribit

Scribit is the first vertical plotter robot in the world able to write and erase on any vertical wall. The project was born at the MIT Senseable City Lab, thanks to the intuition of the Turin director and architect Carlo Ratti and of the engineer Pietro Leoni; the project was funded on KickStarter and IndieGogo with a crowdfunding campaign that raised more than $2 million. The great interest in Scribit is due to its ability to write and delete any content that does not depend on the format of the files, nor even less from the surface: there are already several models of plotters and vertical plotters that can be easily created starting from a programmable board like the Arduino or ESP series, but the results are not comparable to the effectiveness and versatility of Scribit, making it the best available vertical plotter.

Thanks to its technology, Scribit can safely draw, erase and re-draw new images an infinite number of times and any vertical surface is a game: whiteboard, glass or standard plaster. Scribit can operate 4 colors at a time, and with 15 pen colors, the possibilities are endless. It is very easy to install: all that is needed are two nails and a power plug. Scribit can place itself at any point with great precision and uses markers to reproduce any type of content. Scribit is very easy to use: users can select or drop any image, customize their content, and then it draws. It uses G-code: a programming language that allows humans to control automated machine tools. In the following sections the structure and the general functioning of Scribit are described.

1.1.1 Hardware and mechanics

Scribit is an almost circular shaped robot and it consists of four main components: the **chassis**, the **drum**, two **stepper motors** and the actual hardware. The magnesium chassis has a diameter of 20 cm and it is drilled both at the bottom for the power cable and at the centre to accommodate the drum and its motor (Figure 1.1 - left). Furthermore, it has a hole on the back-end side for the ceramic disk in charge of erasing, and it is bend on both sides, on the upper part, to hold the stepper motors (Figure 1.1 - right).

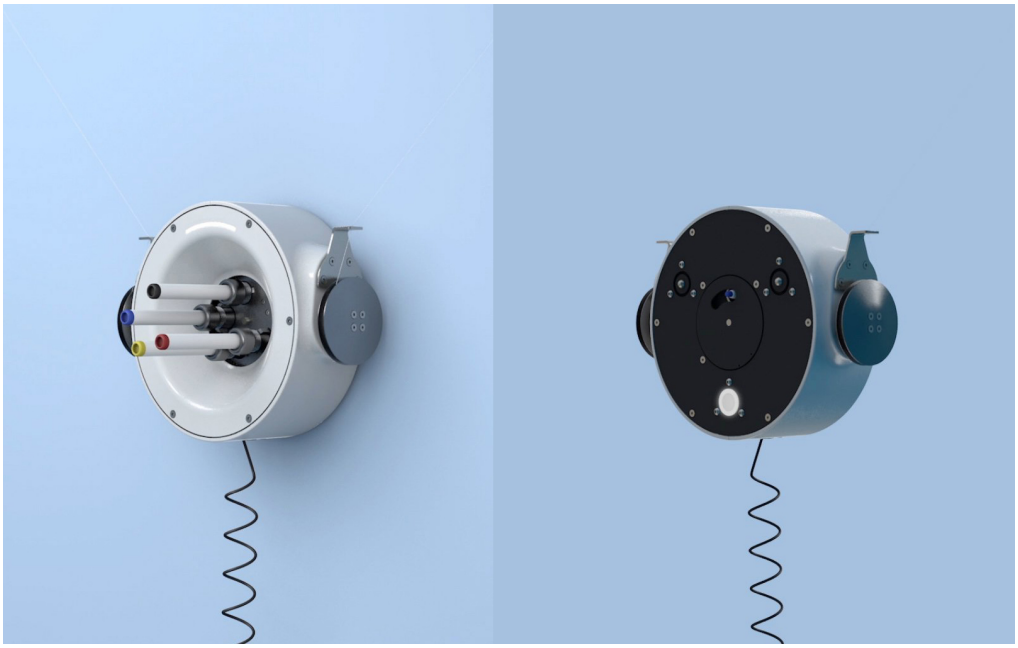


Figure 1.1. Scribit front and back details. (source: Scribit)

The two stepper motors are actually hosted on the two pulleys placed on both sides of the chassis, and in each of them a cable is rolled. Cables are used to fix Scribit to the wall with two nails, and they go through a **cord tensioner** whose purpose is to keep the cable stretched and prevent it from breaking, but also to keep Scribit as close as possible to the surface to avoid undesired movements during drawing activities.

The central drum holds the 4 markers that can be directly inserted if they have the correct size. In fact, each marker is hold by a **penholder** with an o-ring[2] to decide the diameter of the maker. It is possible to unscrew the penholder and replace the o-ring inside it to accommodate any other lower-sized marker. The central drum is also called *desmodromic*, since it is based on the stroke control mechanism both in opening and closing; obviously, this mechanism does not deal with managing the combustion of fuel as it does in desmo engines, rather it simultaneously assists the rotation of the drum with the insertion and the engagement

[2]An o-ring is a loop of elastomer with a round cross-section, designed to be seated in a groove and compressed during assembly between two or more parts, creating a seal at the interface.

of the markers. The desmodromic and deletion mechanisms have been patented by the company. The holed surface containing markers is actually a toothed wheel: it acts as the rack of a rack-pinion system where the pinion is controlled by the shaft of the central stepper motor, which allows rotations and therefore the functioning of the desmodromic system. A detailed view of the components is shown in Figure 1.2.

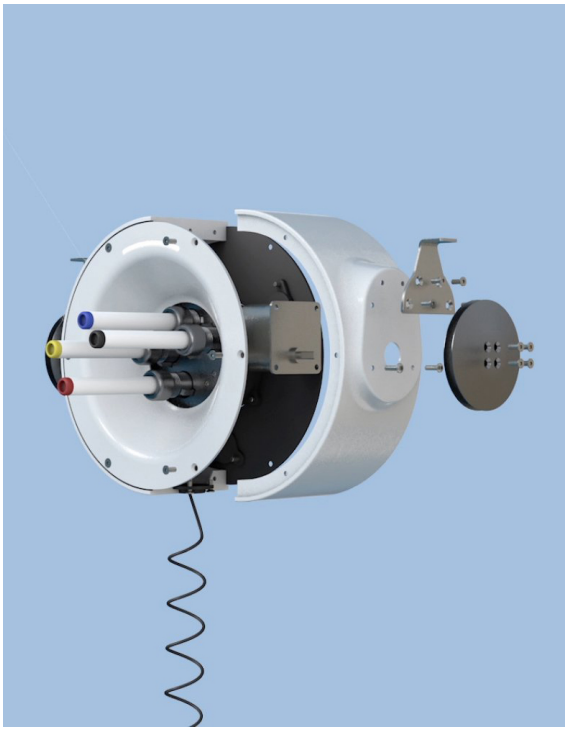


Figure 1.2. Scribit components' details. (source: Scribit)

The ceramic heater has a diameter of 3 cm, and it is hold by a cylinder made of insulating material that protrudes from the plotter of just 2 mm, to prevent the ceramic to directly touch the wall. It is used only during erasing procedures, according to precise rules, and it reaches very high temperature ($\sim 150°$) to be sure that erasing is successful, despite weather conditions that may alter the temperature of the wall. The insulating material is used for heat dissipation to protect both the chassis and the electronics. At the moment, erasing routine can be applied only to normal stuccoed wall, usage on other kinds of surface is not recommended.

The main electronic board is on the upper part of the chassis, far from the ceramic heater. It is customized on Scribit's needs both in shape and composition. It is based on two microprocessors: a SAMD (Arduino-based) used to drive the plotter and all its components through Marlin firmware[3] and an ESP32, used for connectivity utilities and to control the status LED, integrated on the board and visible from the outside, on the back. This LED is used as a reset button, too.

[3]Marlin is a open source firmware which runs on the 3D printer's control board and manages all of the machine's real-time activities including movement through the stepper drivers, heaters, sensors, lights, bed levelling, LCD displays and buttons.

1.1.2 Software

Scribit is basically a vertical plotter and the whole process is managed through a mobile application. The app is available either for Android and iOS devices; and it is actually published both on Play Store and Apple Store. This is because one of Scribit's objectives is to be user-friendly, easily accessible and for everyone. Nowadays with the smartphone it is possible to search, share and create contents, but also to control IoT devices both remotely and locally.

Scribit is a server-based IoT device, and users can both use pre-defined or built-in drawings, or create new contents accordingly to their creativity. Users can also link their personal accounts to multiple Scribit installations, potentially everywhere in the world. They can control each linked Scribit by means of the app.

On the app it is possible to view tutorials, and to read hints, e.g., for the installation process. To use Scribit users must subscribe and create their personal account on the app, and then they can link the robot(s) to their profiles through a **pairing process**. It is very common in IoT universe: Scribit generates its own access point and the user inserts its owns credentials. The communication is on local network, and once pairing is completed the robot is ready to be used.

After installation, **calibration** is needed: the user chooses the type of the wall (stuccoed, whiteboard or glass) and sets its width, then Scribit will move from the centre of the wall making movements aimed at understanding, given the inclination of the plotter, its position within the surface. At the end of the process it will go to the lower left corner of the surface.

Now that Scribit is really ready to be used, the user can access the app and select a design or a widget to print on the wall: except from the built-in drawings, it is possible to customize dynamic templates. For example, it is possible to print the daily weather forecast or for a week, the last 3 tweets published by a given user or linked to a specific hashtag, write text to be printed choosing font and dimension, or convert raster image into vectorial files (which is the focus of this thesis project). At the moment, some of these functionalities are still in beta and to be further tested. They will be available with the second and the third official releases of the app, scheduled for summer 2019.

Scribit uses **lambda functions** on a Google Cloud platform through http requests. The back-end infrastructure is NodeJS based, managed through micro services, and it provides REST API used both to manage exchange of information for drawing, such as user preferences and designs, and to manage communication with all the linked plotters. Once the design is chosen, the lambda function is invoked and the preview of the drawing is returned, then shown on the app. In case of built-in drawings, previews are already available on the app. Then the user can decide the number of colors to use, one or more if the design allows it, the scale factor (small, medium or large with respect to the drawable area), and whether to mirror the drawings or not in case of a transparent wall. Markers are inserted in the plotter, and finally an HTTP request is sent to the lambda function dedicated to the SVG/g-code conversion that returns the NC listing for both printing and erasing (if allowed), as well as useful information for the user, such as the estimated time and the distance in meters that each marker will cover.

NC listing is forwarded, through the server, to Scribit via MQTT broker, which is a standardized publish-subscribe based messaging protocol. When drawing starts, the status of the plotter changes, and it is visible both on the LED, on the upper part of the chassis, and on the app via a printing interface. It is possible either to pause or to stop the drawing: the first case is very useful to replace markers without postponing the drawing (when paused the state changes again and it is communicated to the server always via the MQTT broker). When the process ends Scribit returns in its initial position and the state changes again. Erasing routine can be launched using the app, if the last print was done on stuccoed wall and no other drawing was sent. If the user erased manually the drawing (the ink for stuccoed walls is easily removable with water or heat) and sends another drawing, the previous NC listing is deleted and replaced with the current one.

The aim of this thesis project is to show how customization is performed, i.e., how to go from any raster image to g-code. Conversion is divided into two main steps: from raster image to SVG image and from SVG image to g-code. This work is focused only on the first part while the other part has been developed by another master's candidate of the team.

Chapter 2

Background

2.1 The SVG format

The SVG (Scalable Vector Graphics) [6] is a technology whose purpose is to display vector graphic objects and, therefore, to manage dimensionally scalable images [5]. In particular it is an XML-based language, i.e a widespread metalanguage [1] used in web development. Moreover, SVG is an open standard developed by the World Wide Web Consortium (W3C)[2] since 1999.

SVG became a recognized standard in September 2001 after a rather hard process: before the SVG standard the two main competitors, Macromedia and Microsoft and Adobe e Sun Microsystems, had their own format with their specific features, so a good negotiation work was needed to find compromises between the parties.

SVG allows to treat three types of graphic objects:

- geometric shapes, i.e., lines consisting of straight line segments, curves and areas bounded by closed lines;
- images of raster graphics and digital images;
- explanatory texts, possibly clickable.

Graphic objects can be grouped into more comprehensive objects, equipped with style attributes and they can also be added to other previously constructed and visualized graphic objects. A text can be part of any XML namespace; this increases the searchability and the accessibility of SVG images. The figures expressed using the SVG format can be dynamic and interactive.

[1] In the logic and in the theory of formal languages metalanguage stands for a formally defined language whose purpose is the definition of other artificial languages, defined objective languages or object languages.

[2] The main international standards organization for the World Wide Web.

The potential of scalable vector graphics is remarkable:

- the geometry of each graphic element is mathematically defined (in terms of vectors), instead of being treated by rigid pixel frames;
- it is possible to resize any graphic element maintaining its quality. More specifically, when viewing a given graphic object on different devices (printer, video, mobile screen, etc.), you are sure to always get the highest quality that those supports can provide.

This potential affects all graphics applications that are not purely raster, i.e., based on pixel maps (in practice images coming from cameras or scans). On the other hand, the computational weight of a vector image is generally higher than that of the raster graphics, since the computer processor must substantially regenerate the image from scratch each time the display is resized. The difference between a raster image and a vector image is shown in Figure 2.1.

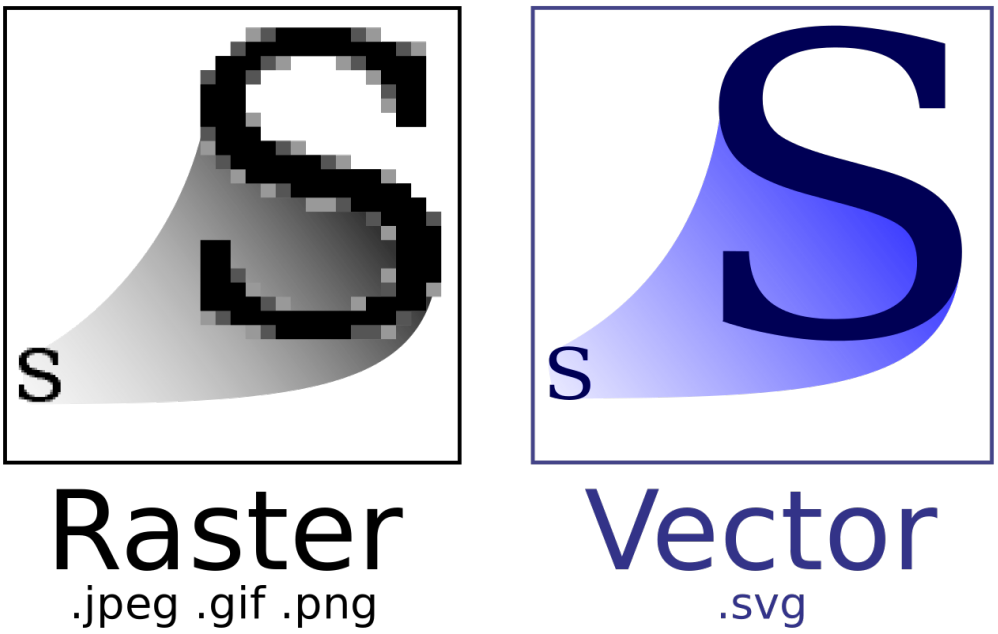


Figure 2.1. Raster and vector images (source: wikipedia).

The advantage of SVG compared to other vector graphics formats is its open standard nature: in this way, in principle, anyone who knows it is able to create SVG pages without the need for a dedicated commercial development environment. Being a format derived from XML, it inherits the ease of generation with automatic means and programming languages.

In the following, a brief description of a generic SVG file (Figure 2.2) is given to better understand how this standard language works.

As with an HTML file, an SVG file consists of a series of markers, called tags and indicated between the angle brackets (<>), which are the basic elements of an SVG image. It is possible to associate to each SVG tag a certain value or to characterize its appearance and behaviour through the use of particular attributes.

```
<?xml version="1.0" encoding="iso-8859-1" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//Dtd SVG 1.1//EN"
   "http://www.w3.org/Graphics/SVG/1.1/Dtd/svg11.dtd">
<svg width="300" height="200"
version="1.1" xmlns="http://www.w3.org/2000/svg">
<text x="10" y="100"
style="fill:red;font-family:times;font-size:18">
First SVG Example
</text>
</svg>
```

Figure 2.2. Basic SVG file

The fact that SVG, as mentioned before, is XML-based is clear from the **first line** of the example above. In fact, in this line there is the declaration of an XML file conforming to version 1.0.

In the **second line** of the example we find the declaration of the type of document. Through this command we inform the program that will take care to visualize our image that the SVG file has been written following the rules indicated in the version 1.1 of the language.

In the **third line** we find the declaration of the main element of an SVG file. As with the `<html>` tag, for web pages, which contains all the elements of the page, the `<svg>` tag is the one that contains all the graphic elements of the image. For this reason it is called root element of the document.

Now let's look at the meaning of the `<svg>` tag attributes used in this example. The width and height attributes indicate what will be the size of the image (in our case it is an image of size 300 x 200 pixels), while the version attribute indicates the version of SVG that is being used (in our case the version 1.1). The **xmlns** attribute is used to indicate the namespace to which the SVG file tags refer. In general, the namespace declaration, in an XML document, indicates to which language the tags used in the file belong. In our case we have declared that all tags and their attributes belong to the SVG syntax.

In the **fourth line** we find the declaration of the only graphic element that contains our example image: a textual element. Our example will therefore contain only the text *First SVG Example* and this writing, red, will be positioned with the first letter at a distance of 10 pixels from the right edge of the image and at a distance of 100 pixels from the top edge image. The position of the writing inside the image is specified by the x and y attributes of the `<text>` tag, while the color, the font of the writing and its size are indicated by the attribute style.

2.1.1 Coordinates system

All SVG elements are positioned within the drawing area called **SVG Canvas**. SVG provides a virtually infinite canvas, but on a practical level only a rectangular area of finite dimension is available. This area is called **SVG Viewport** and is defined through the width and height tags we have seen in the previous example. Dimensions are typically expressed in pixels, however SVG allows you to use different units of measurements such as `em, ex, px, py, cm, mm`, etc.

The graphic elements are placed within the drawing area by specifying their x and y coordinates relative to the SVG image coordinate system where the origin is the upper left corner. Figure 2.3 shows the coordinates system of our example.

Figure 2.3. Coordinates system.

It is possible to change and modify the coordinates system, when needed, by using **viewBox** and **preserveAspectratio** attributes.

2.1.2 Basic shapes

The basic shapes of an SVG image are the line, the polyline, the circle, the polygon, the rectangle and the ellipse.

- **The line**. To draw a line in SVG we can use the `<line>` tag and we must specify both the two starting and ending coordinates.
- **The polyline**. The polyline identifies an area defined by a sequence of lines, and the SVG tag to represent this sequence is `<polyline>`. It is possible to

specify the pairs of coordinates through which the sequence shall pass using the **points** attribute.

- **The circle**. In SVG, a circle is drawn using the `<circle>` tag. The **cx** and **cy** attributes indicate the coordinates of the centre of the circle, while the **r** attribute indicates the size of the radius of the circumference.
- **The polygon**. A polygon in SVG is obtained by using the `<polygon>` tag. Through the **points** attribute we can specify the coordinates of the edge points of the polygon. Note that, unlike in the case of the `<polyline>` tag, the first and last points of the attribute points will be connected so as to always form a closed surface.
- **The rectangle**. To draw a rectangle in SVG we must use the `<rect>` tag. The x and y attributes indicate the coordinates of the upper left corner of our rectangle, and the width and height of the rectangle can be specified using the width and height attributes. With a value greater than zero of the **rx** and **ry** attributes, it is possible to draw a rectangle with rounded edges.
- **The ellipse**. The SVG tag by which you can draw an ellipse is the `<ellipse>`. The cx and cy attributes express the coordinates of the centre of the ellipse, while the rx and ry attributes respectively indicate the radius of the ellipse along the X axis and along the Y axis.

2.1.3 Irregular shapes: paths

The path represents a closed or open geometrical figure, and consists of a sequence of segments that can be either straight or curved. To each segment corresponds a series of commands that allow to specify the characteristics of the segment in question. The basic commands for drawing a path are:

- `Moveto (M x, y)`: sets the origin of the path in the point x, y;
- `Lineto (L x, y)`: draws a line that joins the current point with the point of coordinates x, y;
- `Closepath (Z)`: draws a line that connects the current point with the origin of the path;
- `Horizontal lineto (H x)`: draws a horizontal line that joins the current point with the coordinate point x, leaving the y coordinate unchanged;
- `Vertical lineto (V y)`: draws a vertical line that connects the current point with the coordinate point y, leaving the coordinate x unchanged.

A segment can also be curved; to draw it the following curve commands must be used:

- `Curveto (C x1, y1 x2, y2 x, y)`: draws a cubic Bezier curve from the current point up to the point of coordinates x, y using the points $x1$, $y1$ and $x2$, $y2$ as control points at the beginning and at the end of the curve, respectively;

- `Smooth Curveto (S x2, y2 x, y)`: draws a cubic Bezier curve that joins the current point, with the coordinate point x, y. The point $x2$, $y2$ represents the second control point while the first control point is expressed by the symmetric of the second control point of the previous command, with respect to the current point;
- `Quadratic Bezier Curveto (Q x1, y1 x, y)`: draws a quadratic curve of Bezier that joins the current point with the point of coordinates x, y, using the point $x1, y1$ as control point.
- `Smooth Quadratic Bezier Curveto (T x, y)`: draws a quadratic curve of Bezier joining the current point with the coordinate point x, y. The control point is the symmetric of the control point of the previous command with respect to the current point.

All these commands can also be used in "relative"version. This allows you to express the coordinates of the points relatively to the coordinates of the current point. A relative command is indicated by writing the command itself in lower case.

For example, the `L (lineto)` command, in the relative version, becomes `l x, y` and in this case a line is drawn such that it joins the current point with a point that is x pixels along the X axis and y pixels along the Y axis.

2.1.4 Style

To define the style, you can insert the attributes that define the display properties within the single element, or use the attribute called **style**. The use of the style attribute therefore allows to group the definition of the style of the element within a single attribute.

The main attributes to define the style of an element are:

- `fill`: allows to indicate the fill color of the element. Colors can be expressed by indicating their name or their coding in RGB format;
- `opacity`: indicates the level of opacity of the element. The value 1 indicates the maximum opacity, while the value 0 indicates that the element is transparent;
- `fill-rule`: determines how the element's area will be filled. This attribute is very useful in the case of presence of complex paths within the image, as it establishes how the intersection areas will be filled. The possible values are **nonzero** and **evenodd**;
- `stroke`: allows to specify the color of the line that constitutes the edge of the element;
- `stroke-width`: indicates the measurement of the thickness of the line;
- `stroke-dasharray`: allows to specify the style of the line, for example normal or dashed;

- `stroke-linecap`: defines how the extremes of the line will be drawn and the possible values are **butt, round, square**;
- `stroke-linejoin`: defines the way in which the intersection between two lines will be drawn and the possible values are **miter, round, bevel**;
- `display`: allows to control the visibility of an element. Assigning the value to **none**, the element will not be displayed, while with the **inline** value the element will be displayed.

2.2 Gaussian Blur processing

Blurring is used in image processing to reduce noise and details [4]. Gaussian blur is the result of blurring an image by means of a Gaussian function. The result is a smoother blur resembling of viewing the image through a glass. This technique is widely used in image processing, in particular as a pre-processing phase in computer vision algorithms.

The idea is to convolve the image with a Gaussian function and, since the Fourier Transform of a Gaussian is a Gaussian itself, using Gaussian Blurring has the effect of reducing the image's high-frequency components, so it is a sort of **low pass filter**.

In two dimensions, the Gaussian function is the product of two Gaussian functions in one dimension:

$$G(x, y) = \frac{1}{2\pi\sigma^2} e^{\frac{x^2+y^2}{2\sigma^2}} \tag{2.1}$$

where x and y are the distance from the origin in the horizontal and vertical axis, respectively, and σ is the standard deviation. The two-dimensional Gaussian function is a surface whose contours are concentric circles with a Gaussian distribution. When convoluting the original image with this kind of function, each pixel's new value is set to a weighted average of that pixel's neighbourhood, in such a way that the pixel itself receives the highest weight, while its neighbouring pixels receive as smaller weights as the distance from the original pixel increases. The result is that this kind of blurring preserves boundaries and edges better than others. An example is shown in Figure 2.4

Figure 2.4. Example of Gaussian Blurring.

Furthermore, since the two-dimensional Gaussian function is circularly symmetric; it can be applied to two-dimensional image as two independent one-dimensional calculations. In computational terms this is a useful property, since the calculation can be performed in $O(w_{ker}w_{image}h_{image}) + O(h_{ker}w_{image}h_{image})$ instead of $O(w_{ker}h_{ker}w_{image}h_{image})$, where h is the height, w is the width and ker is the Gaussian kernel.

2.3 Clustering Algorithms

The objective is to find groups of objects such that the objects in a group are similar (or related) to one another and different from the objects in other groups. This means that the *intra-cluster* distances are minimized, while *inter-cluster* distances are maximized. Clustering techniques are explorative, they do not require any knowledge on data, and the analysis to find out the properties of the different groups is performed without any labels [7].

The applications of Cluster Analysis are typically **understanding**, to find out related groups when a huge data set has to be analysed (in this way the analysis considering related groups of data is simpler), or **summarization** through which it is possible to reduce the size of large data sets choosing some significant samples from the different groups.

A *clustering* is a set of clusters. It is possible to distinguish **partitional clusters** and **hierarchical clusters** (Figure 2.5). In the first case, each object belongs to at most one group (or subset), this means that groups are non-overlapped. Hierarchical clusters are a set of nested clusters, organized as a hierarchical tree, and subsets may be overlapped.

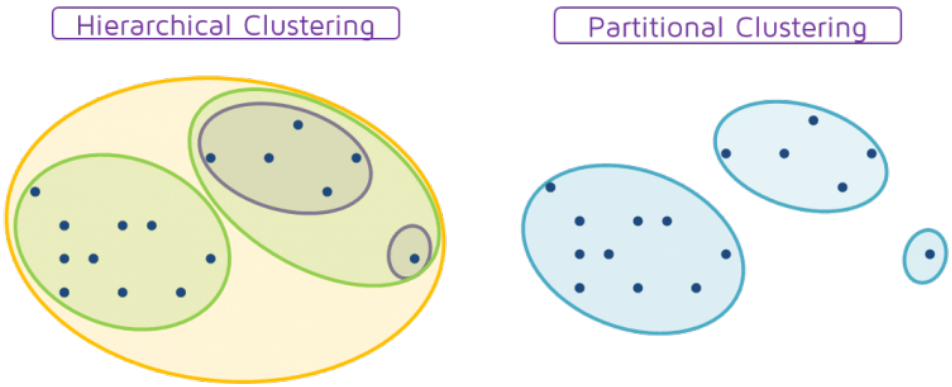


Figure 2.5. Difference between partitional and hierarchical clustering. (source: QuantDare)

Clusterings are classified in different ways; they could be:

- **Exclusive or non-exclusive**: a clustering is exclusive if each point belongs to one single cluster;
- **Fuzzy or non-fuzzy**: in fuzzy clustering a point belongs to every cluster with some weight between 0 and 1 (the sum of weights is equal to 1);

- **Partial or complete**: a clustering is complete if each point belongs at least to one cluster, partial if at least one point does not belong to any clusters;
- **Heterogeneous or homogeneous**: these measures are related to different dimensions such as size, shape and density. For example, clustering is homogeneous with respect to the size if all the clusters have the same size.

Moreover, there are different types of clusters:

- **Well-separated clusters**: clusters are disjointed;
- **Center-based**: a cluster is a set of objects such that an object in a cluster is closer to the centre of its cluster than to the centre of any other cluster. The centre of a cluster is often a centroid, the average of all the points in the cluster (it could be a point that is not in the data set), or a medoid, the most representative point of a cluster (it is a point of the data set);
- **Contiguous cluster**: a cluster is a set of objects such that an object in a cluster is closer to one or more other objects in the cluster than to any other object not in the cluster;
- **Density-based**: clusters are defined by looking at the density degree of the objects in the region. This approach is used when clusters are irregular;
- **Conceptual-based**: a cluster is a set of objects that share a common concept or that have some common properties.

The most common clustering algorithms are: K-means, Hierarchical clustering and Density-based clustering.

2.3.1 K-means clustering

It is a partitional complete clustering approach, this means that noise and outlier points are not detected. Each cluster is associated with a centroid and each point is assigned to the cluster with the closest centroid. This approach requires to specify the number k of clusters to extract. The basic algorithm is very simple: at the beginning k centroids are selected randomly, points are assigned to the closest centroid, and then the centroid of each cluster is recomputed. The algorithm ends after some iterations when the centroids do not change any more (Figure 2.6).

Due to randomly selection of initial centroid, the produced clusters vary from one run to another, so the algorithm should be executed various times to find the optimal solution. The algorithm converges in the first few iterations because often the stop condition is changed, when complex data sets are computed: the algorithm ends when relatively few points change clusters. Moreover, K-means clustering requires a measure for the closeness (Euclidean distance, cosine similarity, correlation etc.), and in order to compute it a common similarity measure should be adopted (data should be normalized according to the chosen measure).

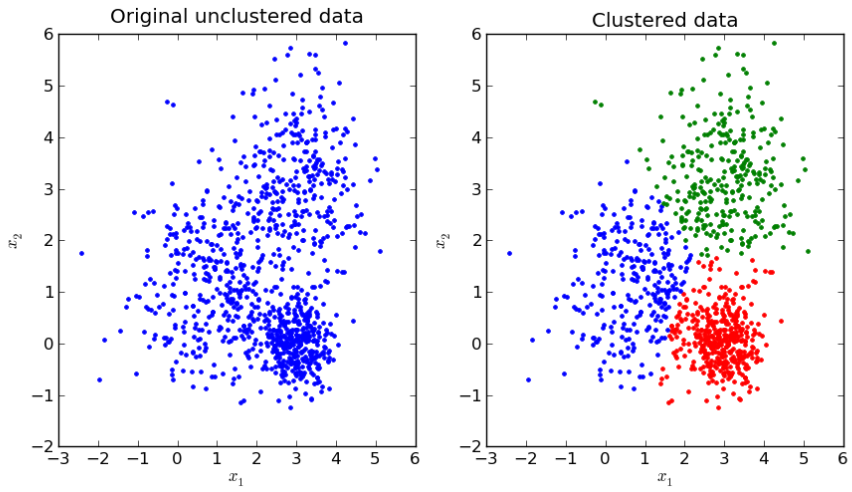


Figure 2.6. Example of K-means clustering. (source: mubaris)

To evaluate the output of the algorithm, in order to choose the best solution, the most common measure is the Sum of Squared Error (SSE):

$$SSE = \sum_{i=1}^{k} \sum_{x \in C_i} dist^2(m_i, x) \tag{2.2}$$

It is computed by taking into account the distance between the centroid (C_i) of the cluster and its points. High value of SSE means that the cluster is not very compact. After computing SSE for each solution, the best one is selected as the one having the lowest SSE (clusters are more compact). Clearly, increasing k, SSE reduces because clusters are smaller (the number of clusters is bigger), and so they are more compact.

While SSE is used to choose the best solution, the **Elbow Graph (Knee approach)** helps to choose the best k (number of clusters). The graph plots the quality measure trend (SSE) against k. The purpose is to identify the range of k values for which there is a significant improvement of the performance (i.e., a significant reduction of SSE). Sometimes, in the Elbow Graph increasing k the SSE increases itself. This is because when k is too high a large number of initial centroids must be selected, and it may happen that some of the centroids are chosen far from the data set, and so they generate empty clusters. To avoid empty cluster generation there are different strategies:

- Choose the point that contributes most to SEE, that is the farthest point from the centroid;
- Choose a point from the cluster with the highest SSE, that is choosing the less compact cluster;
- If there are several empty clusters, the above strategies can be repeated several times.

2.3.2 Bisecting K-means

K-means clustering is very simple; in order to improve performance, it is possible to *pre-process* data by eliminating outliers (they may cause a high value of SSE) and normalizing the data (to properly compute the distances). Furthermore, in *post-processing* it is possible to eliminate small clusters that may represent outliers, or split clusters with relatively high SSE (which are less compact), or merge clusters that are close and that have relatively low SSE (by merging them the resulting SSE is more or less the same).

These techniques could be used during the clustering process, and this is the strategy adopted by Bisecting K-means algorithm. It is a variant of K-means: at each step a cluster is chosen and it is divided into two smallest clusters, this means that Bisecting algorithm repeats different times the K-means algorithm using k=2, until the number of clusters is reached. The cluster to split is chosen by considering the one with the highest value of SSE. This version of the algorithm is better than the previous one, because it includes the post-processing operation during the computation of the clusters. Furthermore, because of the splitting, Bisecting algorithm produces also *hierarchical clustering*.

K-means has problems when clusters are of differing sizes, densities or have non-globular shapes. This is due to the fact that the algorithm tends to select globular clusters (points close to centroid); to avoid these problems a solution is to choose a higher value of k than the one really needed (in order to select more clusters), and then to post-process them to obtain the best solution.

2.3.3 Hierarchical clustering

This type of clustering produces a set of nested clusters organized as a hierarchical tree. The hierarchy can be visualized as a **dendogram** (Figure 2.7 - right), which is a tree like diagram that records the sequences of merges or splits at each step (in particular if at each step only one merge o split operation is recorded, the dendogram is *traditional*, otherwise it is called *non-traditional*). Hierarchical clustering could be *agglomerative*, if it starts with the points as individual clusters and at each step it merges the closest pair of clusters until one cluster is left, or *divisive*, if it starts with one cluster, including all points, and at each step it splits a cluster until each cluster contains a single point.

This type of clustering does not require to set any particular number of clusters (as it happens for K-means clustering), but any desired number of clusters can be obtained by cutting the dendrogram at the proper level (each level contains a particular number of clusters obtained by splitting or merging operations). Traditional hierarchical algorithms use a *similarity or distance matrix*.

The agglomerative clustering algorithm is the most used and it is very simple: first of all the *proximity matrix* is computed, then at each step the two closest clusters are merged and the proximity matrix is updated until one single cluster remains. The key point of this algorithm is the computation of the proximity of two clusters; the way the distance between two clusters is defined distinguishes the different versions of the algorithm.

Inter-cluster similarity (proximity) is defined in different ways:

- MIN: for each pair of points of each cluster (all possible combinations are considered) the distance is computed and then the minimum is selected;
- MAX: for each pair of points of each cluster (all possible combinations are considered) the distance is computed and then the minimum is selected;
- GROUP AVERAGE: the average distance is computed;
- Distance between centroids;
- Other methods driven by an objective function (such as SSE)

Using MIN similarity allows handling non-elliptical shapes (or non-globular shapes), but it is sensitive to noise and outliers: they are not detected and because of that some points are assigned erroneously. Instead, using MAX similarity avoids the problems of noise and outliers: they are not detected but they are assigned to one single cluster, other points are correctly assigned. In this case the limitation is that it tends to break large clusters, when the density is variable, into globular clusters like K-means clustering.

Sometimes hierarchical clustering is used in conjunction with K-means clustering: dendogram is cut at a desired level, and the initial centroids of K-means are selected using the most representative points of these clusters. The complexity of hierarchical clustering is $O(N^2)$ in space, since it uses the proximity matrix and N is the number of points, while $O(N^3)$ is the needed time in many cases, since there are N steps and at each step the proximity matrix must be updated.

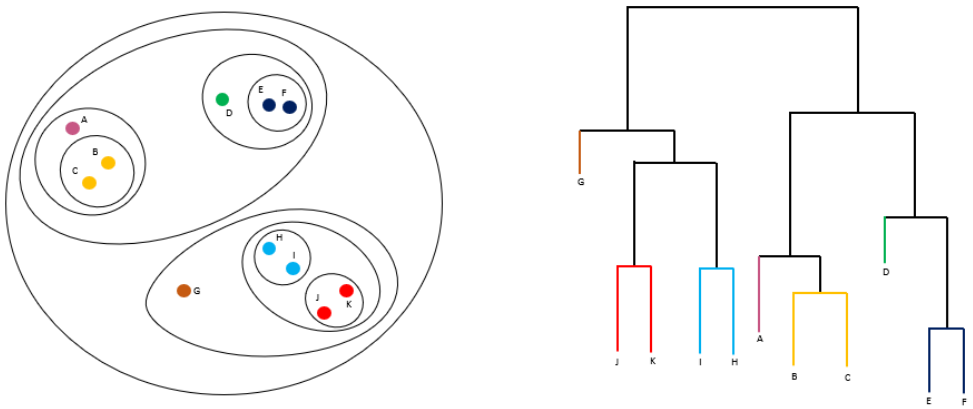


Figure 2.7. Hierarchical clustering (left) and its dendrogram (right)

2.3.4 DBSCAN (Density-based Clustering)

It is a partial clustering approach, this means that noise and outliers are detected (none of the techniques shown above are able to do that).The density is defined as the number of points within a specified radius (Eps). DBSCAN divides points into three categories:

- Core point: if within Eps the number of points is greater than a specified number of points (MinPts), chosen by the user;
- Border point: if it is not a core point but it is in the neighbourhood of one of them (it is next to a core point).
- Noise point: if it is neither a core point nor a border point.

At the beginning, the algorithm eliminates noise points, then considering all core points, each of them at a time, assigns them a label and analyses all the points in the Eps-neighbourhood by assigning them the label of the current core point. DBSCAN is resistant to noise and outliers, and can handle clusters of different shapes and sizes, while problems occur when there is a varying density and a high dimensional data: if eps is too high, clusters with higher density are not detected and merged with other clusters, while if eps is too low, only clusters with higher density are detected, while all the other ones are marked as outliers. To avoid this problem, typically the algorithm is iterated twice: the first one to detect the cluster with the higher density and the second one by clustering the ones that in the previous iteration have been signed as outliers.

The critical point of DBSCAN is to choose the correct value of Eps. The idea is that for points in a cluster, their k-th nearest neighbours are at roughly the same distance while noise points have the k-th nearest neighbour at farther distance, so plotting sorted distance of every point to its k-th nearest neighbour helps to choose Eps: like elbow graph the knee must be checked. If the graph has more knees, it means that there is a varying density, and so more iterations are needed.

2.3.5 Measures of Cluster Validity

The validation of clustering structure is the most difficult task; to evaluate the goodness of the resulting clusters, some numerical measures can be exploited. Measures are classified into two main classes:

- External index: used to measure the extent to which cluster labels match externally supplied class labels, that is to measure the equality between clustering algorithm and a priori knowledge;
- Internal index: used to measure the goodness of clustering structure without respect to external information (no a priori knowledge).

In real cases, clustering techniques are used without a priori knowledge, so internal measures are more interesting. The most common used measure are **cluster**

cohesion and **cluster separation**; they are similar to SSE and the purpose is to obtain a low value of cohesion (the points belonging to each cluster are similar) and a high value of separation (clusters are disjointed). In real systems, the most used measure is the *silhouette*, which takes into account both cohesion and separation. Silhouette can be computed over all data of the dataset (how appropriately the data has been clustered) or all over data of cluster (how tightly grouped all the data in the cluster are).

2.4 Interpolation

In mathematics, and in particular in numerical analysis, interpolation is a method to identify new points of the Cartesian plane starting from a finite set of data points, assuming that all points can refer to a function $f(x)$ of a given family of functions of a real variable [8].

In scientific and technological activities, and generally in quantitative studies of any phenomenon, it often happens that a certain number of points of the plan are obtained with a sampling or with measurement equipment, and they are considered good enough to identify a function that passes through all data points or at least in their vicinity. This is the case of **curve fitting**, where the problem is the approximation of a complicated function by a simple function. Suppose the formula for some given function is known, but too complicated to evaluate efficiently. A few data points from the original function can be interpolated to produce a simpler function which is still fairly close to the original one. The resulting gain in simplicity may outweigh the loss from interpolation error.

Let be given a sequence of n distinct real numbers x_k called **nodes**, and for each of them a second number y_k is given. The aim is to identify a function f of a certain family such that $f(x_k) = y_k$ for $k = 1, ..., n$

The pair (x_k, y_k) is the **given point** and f is called **interpolating function**, or simply **interpolating**, for data points. Sometimes the values y_k, when the interpolating function is a fairly defined function, are written as f_k.

The interpolation problem is summarized as follows: note some pairs of data (x, y), interpretable as points of a plane, the purpose is to find a function, called interpolating function, which is able to describe the relationship that exists between the set of values x and the set of values y.

There are many different methods of interpolation that differ from each other for some specific properties such as: accuracy, cost, number of data points needed, and smoothness of the resulting interpolating function. In the following, the data set shown in Figure 2.8 is considered as testbed.

2.4.1 Linear interpolation

Linear interpolation is a numerical method for finding the roots of a function. It requires the initial estimate of an interval (a, b), within which the root is calculated, such that $f(a)f(b) < 0$. It is also a first order method and therefore provides a slow convergence. Stability is thus guaranteed.

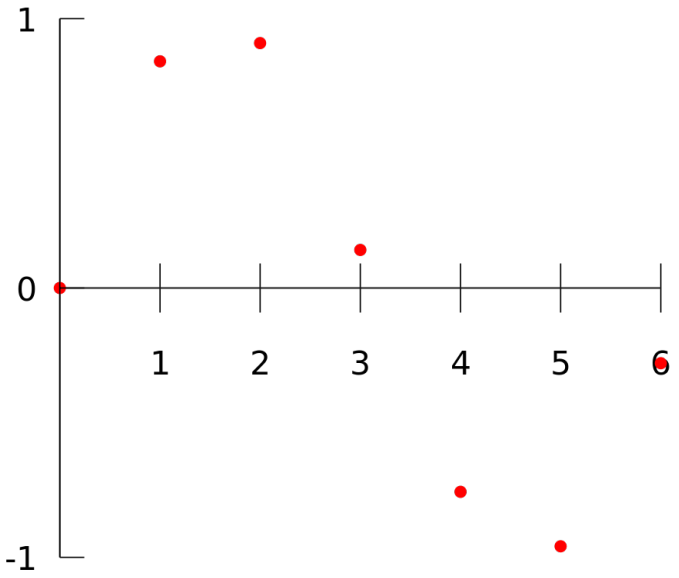


Figure 2.8. Plot of the data points considered as an example. (source: wikipedia)

Given the coordinates of the two known points, (x_0, y_0) and (x_1, y_1), the **linear interpolant** is the straight line between these points, given by the equation of the slopes derived geometrically:

$$\frac{y - y_0}{x - x_0} = \frac{y_1 - y_0}{x_1 - x_0} \tag{2.3}$$

Solving this equation for y:

$$y = y_0 + (x - x_0)\frac{y_1 - y_0}{x_1 - x_0} = \frac{y_0(x_1 - x) + y_1(x - x_0)}{x_1 - x_0} \tag{2.4}$$

This formula can also be interpreted as a weighted average. The weights are inversely related to the distance from the end points to the unknown point; a closer point has more influence than a farther point. Thus, the weights are $\frac{x - x_0}{x_1 - x_0}$ and $\frac{x_1 - x}{x_1 - x_0}$, which are normalized distances between the unknown point and each of the end points. This is clearer rearranging Equation 2.4:

$$y = y_0 \left(\frac{x_1 - x}{x_1 - x_0}\right) + y_1 \left(\frac{x - x_0}{x_1 - x_0}\right) \tag{2.5}$$

The linear interpolation of the example above is shown in Figure 2.9.

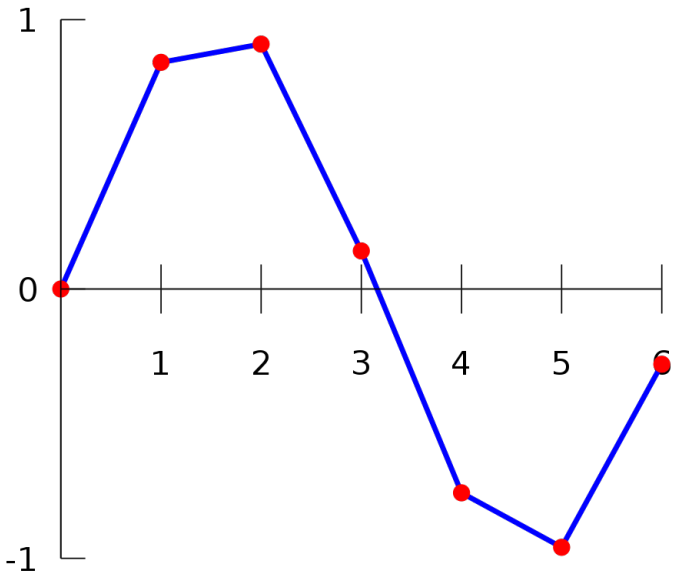


Figure 2.9. Plot of the data points with linear interpolation. (source: wikipedia)

The error estimate indicates that linear interpolation is not very precise: consider $g(x)$ as the interpolating function and suppose that x is between x_1 and x_0 and that $g(x)$ is twice differentiable. Then the error of linear interpolation is:

$$|f(x) - g(x)| \leq C(x_1 - x_0)^2 \quad \text{where} \quad C = \frac{1}{8} \max_{y \in |x_0, x_1|} g''(y) \tag{2.6}$$

Thus, the error is proportional to the square of the distance between the data points. The errors of some other methods, including the polynomial interpolation and the spline interpolation, are proportional to powers greater than the distance between the given points, and therefore are preferable. These methods also produce smoother interpolating functions.

2.4.2 Polynomial interpolation

A problem that frequently occurs in applied mathematics is that of the approximation of functions, which consists in determining one function g, belonging to a selected class of functions, which best approximates a given function f.

Polynomial interpolation is the interpolation of a series of values (for example, experimental data) with a polynomial function that passes through the data points. In particular, any set of $n + 1$ distinct points can always be interpolated by a polynomial of degree n which assumes exactly the value given at the initial points.

The function g, which is meant to approximate the function f is a polynomial of appropriate degree, whose coefficients are determined so that the approximation is the best possible, compatibly with the data available. To make the expression *the best possible* formally correct, it is necessary to define how to measure the distance

between function f and function g. The function g will then be determined in order to have the minimum possible distance from the function f. The distance between f and g is measured by means of the vector r, defined as:

$$r_i = f(x_i) - g(x_i), \quad for \;\; i = 0, \ldots, n \tag{2.7}$$

The searched coefficients are those that make the Euclidean distance of the vector r minimal. This method is called **least squares**.

Polynomial interpolation is a special case of polynomial approximation, where the vector r, or the vector of the squares of the distance between the value given in a point and the value of the approximate polynomial at that point, is zero. While for polynomial approximation the aim is to find a polynomial (generally of a low degree) that approximates the data points with a minimum error, with interpolation the aim is to find the polynomial (potentially of high degree) that passes *exactly* for those points.

An interpolation polynomial can be constructed by means of different methods, for example by using the Vandermonde matrix or the Lagrange interpolation formula. For example, according to the Lagrange interpolation formula, given a function $f(x)$ and $n+1$ points $a_0, a_1, a_2, \ldots, a_n$, for which $f(a_0), f(a_1), f(a_2), \ldots, f(a_n)$ are known, the Lagrange interpolation polynomial of the function f is the polynomial:

$$P(x) = \sum_{i=0}^{n} f(a_i) \prod_{j \neq i}^{n} \frac{x - a_j}{a_i - a_j} \tag{2.8}$$

Although the interpolation polynomial assumes the exact value in the given points, given the higher degree it will tend to oscillate more between one point and the other, thus giving a prediction of the value in those areas that will be worse than that given by a polynomial of lower degree that does not pass through all the data points. However, the interpolation error is proportional to the distance between the points given to power n. Moreover, this interpolant, being a polynomial, is indefinitely differentiable. Thus the polynomial interpolation, in principle, solves all the problems of linear interpolation, but on the other hand the computation of the coefficients of the interpolating polynomial is very expensive (in terms of execution time required by the computer and in terms of complexity of the elaborations). An example of polynomial interpolation is shown in Figure 2.10.

2.4.3 Quadratic spline interpolation

Spline interpolation is a particular interpolation method based on spline functions. It is a tool of numerical analysis used in many fields of application (for example in physics or statistics). Unlike polynomial interpolation, which uses a single polynomial to approximate the function over the entire definition interval, spline interpolation is achieved by dividing the interval into multiple sub-intervals and choosing for each of them a polynomial of degree k (usually small) [9]. It will then be imposed that two successive polynomials are smoothly sealed, i.e., observing the continuity of the first k-1s derivatives. The function that is obtained with a

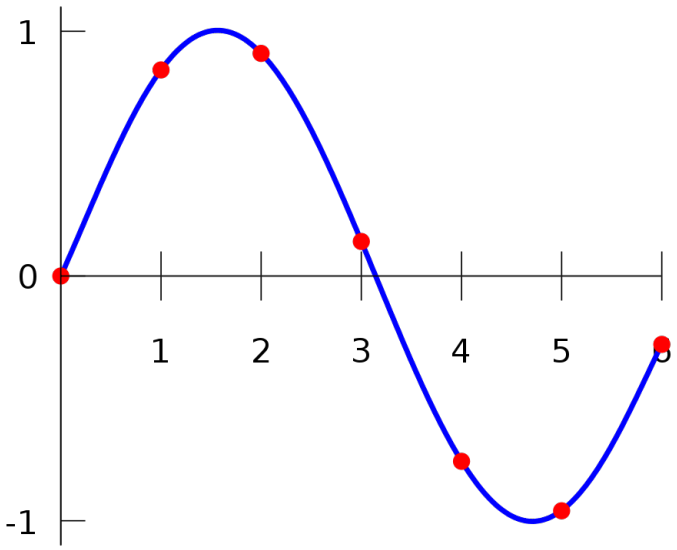


Figure 2.10. Plot of the data points with polynomial interpolation. (source: wikipedia)

procedure of this kind is called **spline function**. Linear interpolation, which uses a linear function, i.e., a grade 1 polynomial, on each sub-interval can be considered a special case of spline interpolation.

Suppose that $n+1$ points are known (**knots**) and that they satisfy the relation $a = x_0 < x_1 < \cdots < x_n = b$. A **spline function of degree k** having knots $x_0, x_1, \ldots, x_n$ is a function S such that:

- On each interval $[x_i - 1, x_i]$ is a polynomial of degree $\leq k$;
- S has a continuous $(k-1)^{st}$ derivative on $[x_0, x_n]$.

A **quadratic spline** $S_{2,n}(x)$ has the following:

$$S_{2,n}(x) = \begin{cases} p_1(x) = a_1 + b_1 x + c_1 x^2, & x \in [x_0, x_1], \\ p_2(x) = a_2 + b_2 x + c_2 x^2, & x \in [x_1, x_2], \\ \quad \vdots & \\ p_n(x) = a_n + b_n x + c_n x^2, & x \in [x_{n-1}, x_n] \end{cases} \tag{2.9}$$

$S_{2,n}(x)$ is thus *continuous* and has *continuous first derivative* everywhere in the interval $[a, b]$, in particular, at the knots. Also, to be an **interpolatory quadratic spline**, it must interpolate the data, that is:

$$S_{2,n}(x_i) = f_i, \quad i = 0,1,\ldots,n \tag{2.10}$$

Note that the function $S_{2,n}(x)$ has two quadratic pieces incident at the interior knot x_i; to the left of x_i, it is a quadratic $p_i(x)$, while to the right it is a quadratic

$p_{i+1}(x)$. Thus, a necessary and sufficient condition for $S_{2,n}(x)$ to have continuous first derivative is for these two quadratic polynomials incident at the interior knot to match in first derivative value. So a set of *smoothness conditions* is given, at each interior knot:

$$p_i'(x_i) = p_{i+1}', \quad i = 1,2,\ldots,n-1 \tag{2.11}$$

In addition, to interpolate data, a set of *interpolation conditions* is given, on the i-th interval:

$$p_i(x_{i-1}) = f_{i-1}, \;\; p_i(x_i) = f_i \quad i = 1,2,\ldots,n \tag{2.12}$$

The n quadratic splines have three unknown coefficients, so $S_{2,n}(x)$ involves $3n$ unknown coefficients. Interpolation imposes $2n$ linear constraints while the continuity of the first derivative imposes $(n-1)$ linear constraints; hence there are $3n-1$ linear constraints against the $3n$ unknowns. One more linear constraint is needed and it is typically chosen on the specific case.

The quadratic spline interpolation of the example above is shown in Figure 2.11.

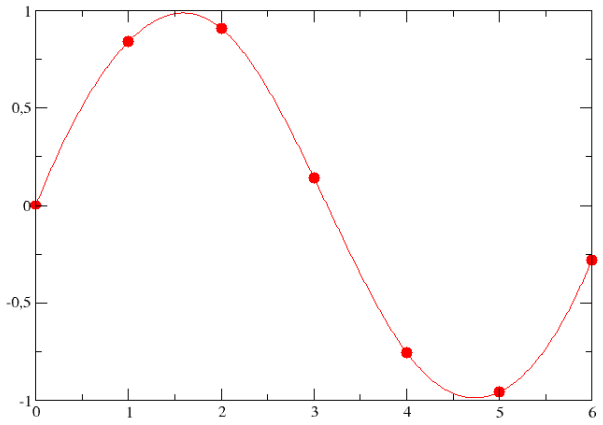


Figure 2.11. Plot of the data points with quadratic spline interpolation. (source: wikipedia)

The interpolating function obtained with spline interpolation is smoother than those obtained with other methods (for example with the polynomial interpolation), in the sense that it is the interpolating function with *minimal mean curvature*. Furthermore, the spline interpolation is easier to evaluate than the high degree polynomials required by the polynomial interpolation and does not suffer from the Runge phenomenon[3]. However, if the data to be interpolated have particular conformations (for example, they form steps), the interpolating spline may

[3]It is a problem related to polynomial interpolation on equispaced nodes with high degree polynomials. It consists in increasing the amplitude of the error near the extremes of the interval.

be subject to the Gibbs phenomenon[4], wide oscillations near a step. To overcome this problem, spline smoothing or spline tension are used.

2.5 Edge contour representation

When dealing with image processing or computer vision algorithms is essential to handle shapes and region representations. Objects can be described through their boundaries or *contours*, and the simplest representation of a contour is using an ordered list of its edge points. This leads to edge detection, which is a fundamental tool in image processing, particularly in the areas of feature detection and feature extraction. The aim of edge detection is to identify points in a digital image at which the image brightness changes sharply.

Edge contour representation, however, is not very compact and is not very effective for image analysis, although it is as accurate as the location estimates for the edge points. A more powerful representation is the chain-code representation [10].

2.5.1 Chain-code representation

This kind of representation is quite simple:

- Each edge point along the contour is specified through one single direction;
- Once the starting edge is chosen, go clockwise around the contour;
- The direction towards the next edge point is given by means of one of the four (or eight) directions.

Using this convention, it is possible to represent the shape of an object as a chain code, that is, a set of directional codes, with one code following another like links in a chain. Since chain codes provide an algorithm for representing the shape of an object, it is possible to build a machine that is capable of encoding that information. But algorithms cannot be implemented in a computer program using just any coding system. It must be a language that the computer can recognize. Since digital computers typically use a numerical system for storing information (everything is coded in "O's"and "1's") it will be a step in the right direction to modify the directional system to a numerical system rather than a letter system. Computer programmers who write chain codes based on an eight-way directional system, typically represent those eight directions using a numbering scheme. Some examples of chain-code representation using different numbering schemes are shown in Figure 2.12.

[4]It is the peculiar manner in which the Fourier series of a piecewise continuously differentiable periodic function behaves at a jump discontinuity.

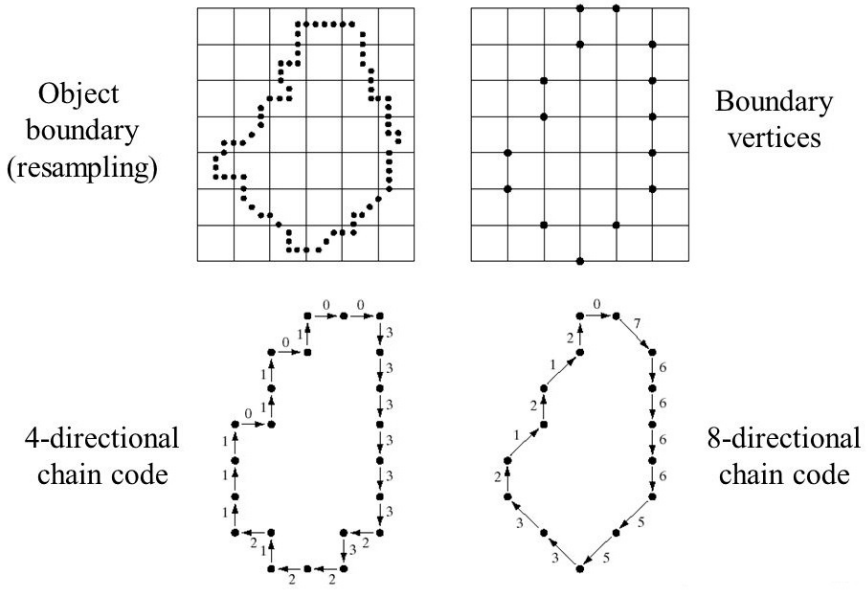


Figure 2.12. Examples of chain-code representation. (source: Machine vision by R.Jain et al.)

2.5.2 Curve interpolation vs curve approximation

A more powerful representation of contours is fitting an appropriate curve having some analytical description, and this typically leads to *curve fitting*. It is closely related to interpolation: a curve interpolates a list of points if it passes through the points, instead a curve approximates a list of points if it passes close to the points.

When using curve interpolation, the assumption is that the edge points have been extracted accurately, and the most common method used to obtain curve interpolation is polygonal representation, based of course on polynomial interpolation. In this case the contour is represented as a polygon and the edges are its vertices. Edge points identify a sequence of segments and each of them is fitted with a curve. There are three main ways to compute the polygonal approximation of a contour: Split algorithm, Merge algorithm and Split and Merge algorithm. The first approach is based on recursively splitting segments connecting edge nodes until the fitting error is below a given threshold, while the second one is based on least-squares and merges line segments by iteratively adding new edge points. To improve accuracy, the last method is an hybrid approach.

Another remarkable method is based on circular arcs. The assumption is that the edge list has been approximated by line segments, then subsequences of the line segments can be replaced by circular arcs. This kind of approach uses spline interpolation (typically it uses cubic splines, i.e., 3rd degree polynomials).

Curve approximation, instead, is meant to obtain a higher accuracy because the curve is not forced to pass through particular edge points. Approximation-based methods use **all the edge points** to find a good fit (in contrast to the previous methods that use only the interpolated edge points). The two main methods to approximate curves are *Least-square fit* and *Robust regression fit*. They depend on the reliability with which the edge points can be grouped into contours; for

instance the least-square approach can be used only if it is certain that the edge points grouped together belong to the same contour, instead the robust regression fit is more accurate with some grouping errors.

2.5.3 Least-square fit

The method of least squares is an optimization technique (or regression) that allows to find a function, represented by an optimal curve (or regression curve), which is as close as possible to a data set (typically points in the plan). In particular, the function found must be the one which minimizes the sum of the squares of the distances between the observed data and those of the curve that represent the function itself, also called **residuals**.

The mathematical formulation of the linear regression problem is summarized as follows: given the measurement of $n+1$ real variables $y(t)$, $u_1(t), \ldots, u_n(t)$ over a time interval (e.g., for $t = 1,2,\ldots,N$), find if possible the values of n real parameters $\theta_1, \theta_2, \ldots, \theta_n$ such that the following relationship holds:

$$y(t) = \theta_1 u_1(t) + \ldots + \theta_n u_n(t) \tag{2.13}$$

Thus, the least-square estimate is given by:

$$\hat{\theta} = [\Phi^T \Phi]^{-1} \ \Phi^T y \tag{2.14}$$

where $\mathbf{y}$ is a vector whose *i-th* element is the *i-th* observation of the dependent variable and Φ is the matrix whose *ij* element is the *i-th* observation of the *j-th* independent variable.

An example of linear least-square fit is shown in Figure 2.13.

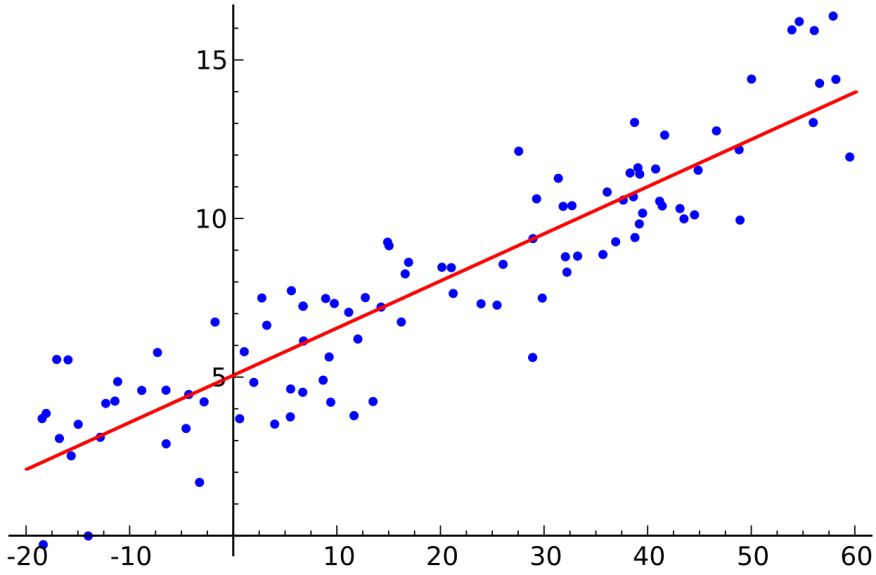


Figure 2.13. Random data points and their linear regression. (source: wikipedia)

2.5.4 Robust regression fit

When dealing with not small errors on the estimates and with outliers, i.e., observations which do not follow the pattern of the other observations, the least-squares estimates are highly sensitive, so other fitting methods, such as robust regression, must be exploited.

The idea is to find the relationship between one or more independent variables and a dependent variable. Robust regression methods are designed to try various subsets of the data points and to choose the subset that provides the best fit. The *iterative fitting* works as follows:

1. Find the curve that fits the data set;
2. Remove the point(s) with the worst error;
3. If the curve parameters do not change significantly, the fitting finishes otherwise repeat from step 1.

Other methods such as *Least-median of squares*, or Least trimmed square (LTS), are more accurate and robust. In this case, the LTS method attempts to minimise the sum of the squared residuals only over a subset k of the n considered points. The unused $n-k$ points do not influence the fit. A comparison between least-square fit and robust fit on the same data-set is shown in Figure 2.14.

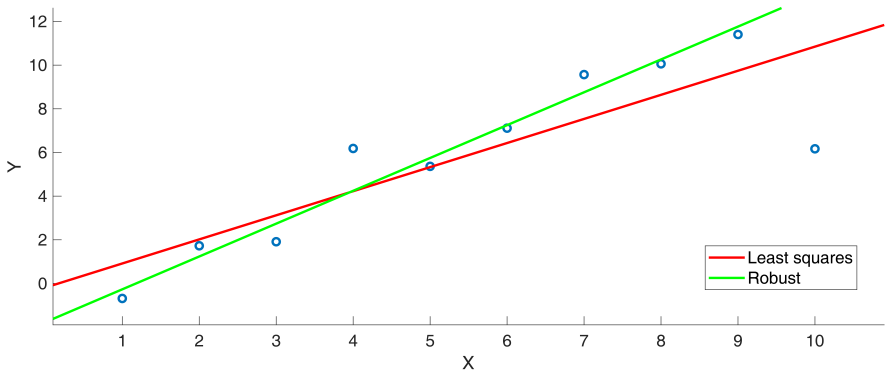


Figure 2.14. Random data points and comparison between least-square and robust fitting.

2.5.5 Evaluating the goodness of fitting

To evaluate the goodness of a fitting method many metrics are available. In the following the most common measures are listed:

- **Maximum absolute error**, which measures how much the points deviate from the curve in the worst case

$$MAE = max_i|d_i| \tag{2.15}$$

- **Mean squared error**, which gives an overall measure of the deviation of the curve from the edge points, in particular it measures the average of the squares of the errors, that is the average squared difference between the estimated values and what is estimated

$$MSE = \frac{1}{n}\sum_{i=1}^{n} d_i^2 \tag{2.16}$$

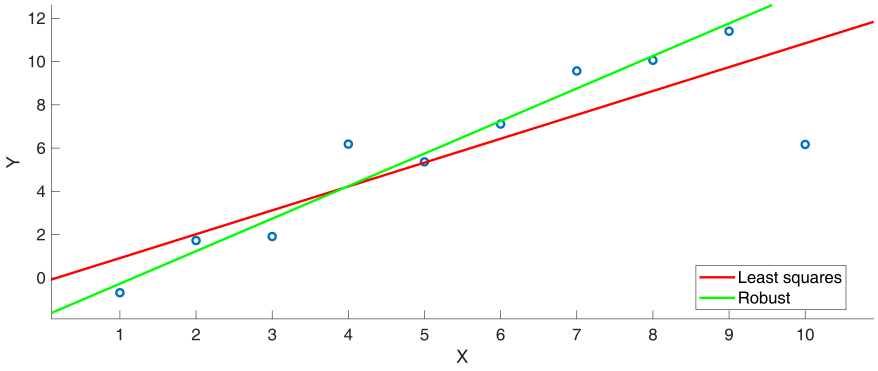
- **Root-mean squared error**, which is typically used instead of the Mean squared error, and is simply given by the root of the MSE

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n} d_i^2} = \sqrt{MSE} \tag{2.17}$$

- **Normalized maximum absolute error**, which is the ratio of the maximum absolute error to the length of the curve

$$NMAE = \frac{max_i|d_i|}{L} = \frac{MAE}{L} \tag{2.18}$$

Figure 2.15 shows the RMSE computed on the same data-set reported in Figure 2.14. This confirms that the robust fitting performs better than the least-squares fitting in case of outliers.



Least squares:	Y = -0.188327 + 1.10351*X	RMS error = 2.21375
Robust:	Y = -1.77278 + 1.50415*X	RMS error = 1.43663

Figure 2.15. RMSE evaluation of the previous example.

Chapter 3

Image rendering from any format to SVG format

This chapter describes in details the main contribution of the thesis project. As mentioned in Chapter 1, the aim is to make Scribit flexible and affordable for everyone, not only for those who are able to handle and create images in vector formats. This is why is offered the possibility to users to use images in any format. This requires software support, i.e., a conversion from any format to SVG format, since the expected input is an SVG file. This thesis project is focused on such a rendering process.

3.1 Proposal

The overall process is based on the Marching squares algorithm [11]. It is a computer graphics algorithm that generates contours for two-dimensional scalar fieldS. In particular, the contours can be of two kinds: *isolines*, i.e., lines following a single data level, or *isobands*, i.e., filled areas between isolines. The algorithm takes as input an image grid and processes each cell in the grid independently; this is why the algorithm is parallel and the computational overhead is not so high.

In this thesis project the algorithm is developed in JavaScript, a scripting object-oriented language widely used in Web, mobile and server development. The output is a collection of .js files which are part of a library, able to render any image into the specific SVG format needed for the translation to g-code, which means that IT is a slightly different SVG format, customized on Scribit's needs. The rendering process goes through 6 main steps: color quantization, layer separation and edge detection, pathscan, interpolation, tracing and SVG coordinates rendering. Table 3.1 lists the main user friendly functions provided, in particular:

- **imageToSVG** loads an image from aN URL, calls the tracing routine when loaded, and then executes callback with the scaled SVG string as argument
- **imagedataToSVG** traces imagedata, then returns the scaled svg string

- **imageToTracedata** loads an image from aN URL, calls the tracing routine when loaded, and then executes callback with tracedata as argument
- **imagedataToTracedata** traces imagedata, then returns tracedata (layers with paths, palette, image size)

Function name	Arguments	Returns
imageToSVG	image_url /*string*/ , callback /*function*/ , options /*optional object */	Nothing, callback will be executed
imagedataToSVG	imagedata /*object*/ , options /*optional object */	svgstring /*string*/
imageToTracedata	image_url /*string*/ , callback /*function*/ , options /*optional object */	Nothing, callback will be executed
imagedataToTracedata	imagedata /*object*/ , options /*optional object */	tracedata /*object*/

Table 3.1. API table

The overall rendering process is performed by the `imagedataToTracedata` function, which calls the vectorizing functions. The objects involved are detailed in each of the following sections, here is described only the `options` object needed to better understand the rendering process. It is basically a JavaScript object and Tables from 3.2 to 3.6 describe the parameters involved in the different steps of the process.

Option name	Default value	Meaning
`blurradius`	`0`	Set this to 1..5 for selective Gaussian blur preprocessing
`blurdelta`	`20`	RGBA delta threshold for selective Gaussian blur preprocessing

Table 3.2. Blur processing options

Option name	Default value	Meaning
`colorsampling`	`2`	0: disabled, generating a palette; 1: random sampling; 2: deterministic sampling
`numberofcolors`	`16`	Number of colors to use on palette if pal object is not defined
`mincolorratio`	`0`	Color quantization will randomize a color if fewer pixels than (total pixels*mincolorratio) has it
`colorquantcycles`	`3`	Color quantization will be repeated this many times

Table 3.3. Color quantization options

Option value	Default value	Meaning
`corsenabled`	`false`	Enable or disable CORS Image loading
`ltres`	`1`	Error treshold for straight lines
`qtres`	`1`	Error treshold for quadratic splines
`pathomit`	`8`	Edge node paths shorter than this will be discarded for noise reduction
`rightangleenhance`	`true`	Enhance right angle corners

Table 3.4. Tracing options

Option name	Default value	Meaning
`layering`	`0`	0: sequential ; 1: parallel

Table 3.5. Layering options

Option name	Default value	Meaning
`strokewidth`	`1`	SVG stroke-width
`linefilter`	`false`	Enable or disable line filter for noise reduction
`scale`	`1`	Every coordinate will be multiplied with this, to scale the SVG
`roundcoords`	`1`	rounding coordinates to a given decimal place. 1 means rounded to 1 decimal place like 7.3; 3 means rounded to 3 places, like 7.356
`viewbox`	`false`	Enable or disable SVG viewBox
`desc`	`false`	Enable or disable SVG descriptions

Table 3.6. SVG rendering options

3.2 Color Quantization

This is the first step of the overall process;the function `colorquantization` takes as inputs two objects: `options` (described in the previous section) and `imgdata`, i.e., an `ImageData` object [12]. It is used to manage images, and represents the underlying pixel data of an area. In particular it has three properties:

- `ImageData.data`, which is a `Uint8ClampedArray`, i.e., a typed array of 8-bit unsigned integers clamped to 0-255, representing a one-dimensional array containing the data in the RGBA order;
- `ImageData.height`, which is an unsigned long representing the actual height, in pixels, of the `ImageData`;
- `ImageData.width`, which is an unsigned long representing the actual width, in pixels, of the `ImageData`.

In `colorquantization`, the `palette` is also involved. It is used to represent the different colors of the image and the number of colors (different paths) of the final SVG image. It is simply an array of color objects and can be accessed through `options.pal`. A basic example of this object is shown in Figure 3.1.

```
pal = [
{r:0, g:0, b:0, a:255},
{r:23, g:55, b:127, a:255},
{r:255, g:255, b:255, a:255},
{r:15, g:15, b:21, a:255},
]
```

Figure 3.1. Example of pal object

The aim is to reduce the number of distinct colors of the image to have a simpler representation. The idea is to obtain an *indexed image*: the color information is split between an array of indices (one for each color), the *palette* mentioned before, and a matrix, where each element is the index of the corresponding color in the palette. This means that the image pixels do not contain the full information but only indexes of the palette. This kind of approach is typically used to manage digital images' colors, because it is not memory consuming, and to save file storage. An example of how indexed images work is shown in Figure 3.2.

The overall process is decomposed in two main phases: blur processing and K-means clustering.

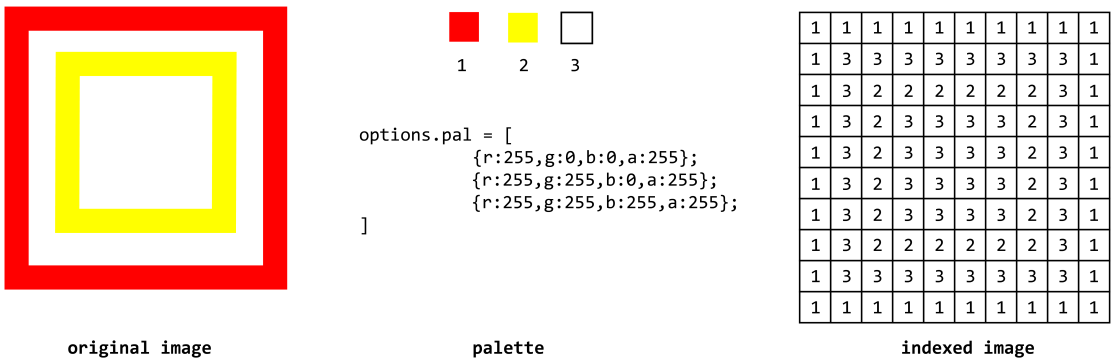


Figure 3.2. Example of indexed image.

3.2.1 Blur processing

In this case, the Gaussian blur's separable property is used to divide the blur process into two steps: at the beginning a one-dimensional kernel is used to blur the image in the horizontal direction only, then the vertical direction is processed using the same kernel. Each pixel of `imgd` is multiplied by the Gaussian kernel value, so that a blurred copy of the original indexed image is obtained.

Blurring is based on two parameters: `BLURRADIUS`, to define the number of neighbours to consider, and `BLURDELTA`, to implement a **selective blur**, deciding whether to consider the blurred pixel or not on the basis of RGBA[1] distance. This distance is used to compute the color similarity;given two pixels with RGBA values (r_i, g_i, b_i, a_i) with $i = 1,2$, the RGBA distance d is computed as follows:

$$d = abs(r_2 - r_1) + abs(g_2 - g_1) + abs(b_2 - b_1) + abs(a_2 - a_1) \tag{3.1}$$

If d is above `BLURDELTA` then the blurred pixel is discarded, at the very end the blurred `imgd` is returned.

3.2.2 K-means clustering

In this work, a simplified variant of this algorithm is used: the number k of clusters is defined a priori: it is the number of colors of the palette (palette's length). At the beginning of color quantization, in fact, the palette is chosen: customized on the user's needs, randomly or deterministically generated from the image or randomly generated, and there are three different functions intended to do that. Moreover, `colorquantization` is meant to be general purpose, but in this work the aim is to render an actual image so, despite `options.pal` is optional in general, in this case it is always an input. In particular, the open source project Color Thief [13] by Lokesh Dhakar is used. This library is able to grab the color palette from an image; an example of how it works is shown in Figure 3.3.

[1]RGBA stands for Red, Blue, Green and Alpha. It is an additive color model in which red, green and blue colors are added together in various ways to reproduce a broad array of colors. The alpha channel describes the degree of transparency/opacity of each pixel.

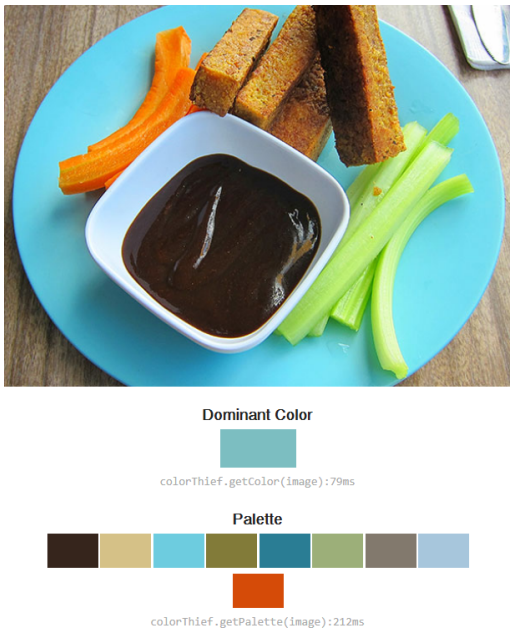


Figure 3.3. Example of palette extraction using Color thief. (source: ColorThief)

The algorithm uses NUMBEROFCOLORS in place of k and, since it is predefined, neither the Elbow Graph nor the SSE evaluation are needed. Clusters are basically the colors and the dataset is the indexed matrix (pixels).

The closeness measure adopted is the *Taxicab geometry* or *Rectilinear distance*, where the distance between two points is the sum of the absolute differences of their Cartesian coordinates. In this case, the distance is computed using RGBA values and at each iteration of the algorithm the closest color from the palette is chosen as cluster. The clustering algorithm is repeated COLORQUANTCYCLES times and at the very end both the averaged palette of colors and the indexed matrix are returned. The image inf Figure 3.4 is considered as reference example.

Figure 3.4. Reference example.

3.3 Layer Separation and Edge detection

The aim is to detect and separate as many layers as the number of colors of the palette. Each layer is represented as a multi-dimensional array and is strictly linked to one single color. The starting point is the indexed matrix and the objective is to assign at each pixel an edge node type (Figure 3.6 - right) going through the following steps:

1. Each pixel is considered as the lower right corner of a 4 pixels grid;
2. Each pixel of the 2x2 matrix has a given value (Figure 3.6 - left);
3. Walking around the cell in a clockwise direction, the edge node type is given by the sum of the four values: the value itself if the pixel has the same color as the inspected one, 0 otherwise.

In this way, each element of the layering matrix, representing the edge node type of that pixel, can have 16 possible values in the range 0-15. Layering can be sequential (default) or parallel: in the first case a two-dimensional matrix is created (one matrix for each color), while in the second case a unique cubic matrix is calculated. Note that the indexed matrix produced by color quantization is oversized, and it is such that the external contour is -1 to avoid out of memory indexing in edge detection computation. The following snippet shows how fast and easy is to compute edge node types:

```
// Looping through all pixels and calculating edge node type
// cnum is the current color
    for(j=1; j<ah; j++){
      for(i=1; i<aw; i++){
         layer[j][i] =
           ( ii.array[j-1][i-1]===cnum ? 1 : 0 ) +
           ( ii.array[j-1][i]===cnum ? 2 : 0 ) +
           ( ii.array[j][i-1]===cnum ? 8 : 0 ) +
           ( ii.array[j][i]===cnum ? 4 : 0 )
         ;
      }
     }
```

Figure 3.5. Edge node types calculation

Some examples of edge node types of the image shown in Figure 3.4 are listed in Figure 3.7. At the end of this step, which takes `imgd` and `options.pal` as inputs, two different outputs may be provided:

- `layer[imgd.width][imgd.height]`, a bi-dimensional matrix, computed by the `layeringstep` function, where each element is the edge node type with

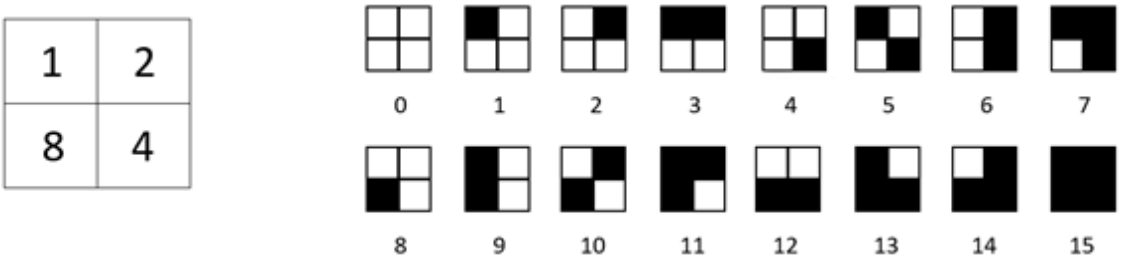


Figure 3.6. Pixel values (left) and Edge node types (right)

respect to the given color. It is used in case of *sequential layering*, so in this case the function is called `pal.length` times;

- `layers[k][imgd.width][imgd.height]`, where k is `pal.length`. It is a k-dimensional matrix computed by the function `layering`, called only once in case of *parallel layering*, and englobes the bi-dimensional matrices of each color.

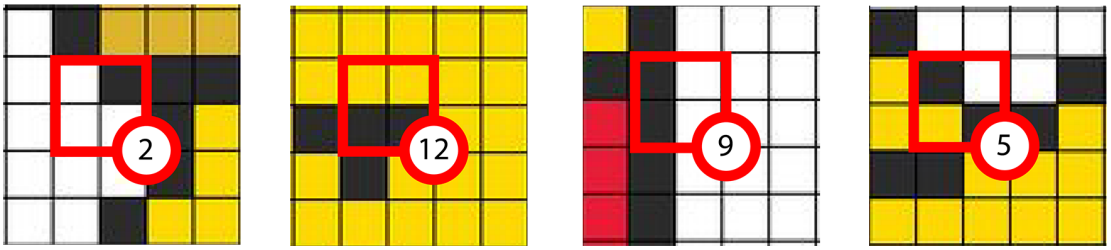


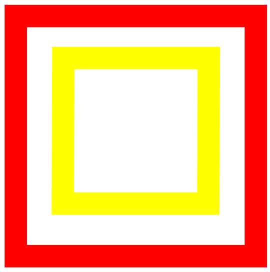
Figure 3.7. Edge node types examples

Considering the original image of Figure 3.2, the obtained layers for both red and yellow colors are shown in Figure 3.8.

3.4 Pathscan

The aim is to find and save all paths for each color, starting from an edge corner by finding chains of edge nodes (Figure 3.9 – red dots). In the following, the basic idea of the algorithm is described:

1. Scanning is performed clockwise, so the only feasible starting points are edges of type 4 or 11 and the default direction is right;
2. Directions are managed using the following notation: 0 stands for `RIGHT`, 1 stands for `UP`, 2 stands for `LEFT`, 3 stands for `DOWN`;
3. A pre-built lookup table is used to look for paths;



original image

4	12	12	12	12	12	12	12	12	8
6	0	0	0	0	0	0	0	0	9
6	0	0	0	0	0	0	0	0	9
6	0	0	0	0	0	0	0	0	9
6	0	0	0	0	0	0	0	0	9
6	0	0	0	0	0	0	0	0	9
6	0	0	0	0	0	0	0	0	9
6	0	0	0	0	0	0	0	0	9
6	0	0	0	0	0	0	0	0	9
2	3	3	3	3	3	3	3	3	1

layers[0] -> red

0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	4	12	12	12	12	12	0	0
0	0	6	0	0	0	0	9	0	0
0	0	6	0	0	0	0	9	0	0
0	0	6	0	0	0	0	9	0	0
0	0	6	0	0	0	0	9	0	0
0	0	2	3	3	3	3	1	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0

layers[1] -> yellow

Figure 3.8. Layers extraction examples

4. Using point's coordinates it is possible to detect the end of a path (if coordinates are equal to the starting point), and if it is a hole path or not (by checking the bounding box coordinates).

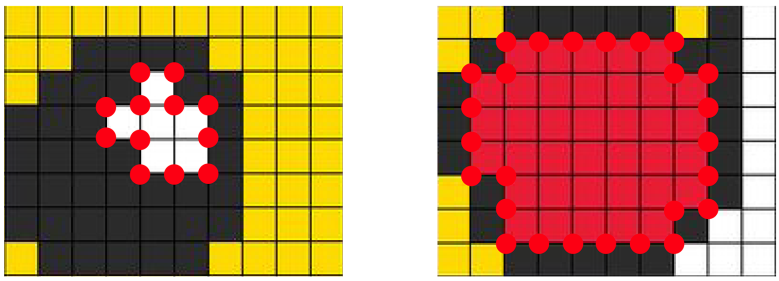


Figure 3.9. Edge node types examples

A lookup table is an array that replaces runtime computation with a simpler array indexing operation. The savings in terms of processing time can be significant, since retrieving a value from memory is often faster than undergoing an *expensive* computation or input/output operation. In this case the table has as many rows as the number of edges, and as many columns as the number of directions. Moreover, each column contains 4 values:

- next edge type (0-15);
- new direction (0-3);
- the increment of the coordinates of the next pixel (dx and dy).

so the overall table is a three-dimensional matrix. Each position of this table is indexed by the couple `(edge node type, direction)`: the first one selects the row, the second one indicates the column , i.e., the corresponding quadruple. Moreover, since edge node types `0` and `15` represent, respectively, an empty node and a full node are invalid indices.

Considering the red layer of Figure 3.8, the main accesses to the lookup table of the pathscan algorithm are listed in Figure 3.10:

```
4 -> [0, 0, 1, 0] //dir = 0 - right

12 -> [0, 0, 1, 0] //dir = 0 - right

8 -> [0, 3, 0, 1] //dir = 3 - down

9 -> [0, 3, 0, 1] //dir = 3 - down

1 -> [0, 2, -1, 0] //dir = 2 - left

3 -> [0, 2, -1, 0] //dir = 2 - left

2 -> [0, 1, -1, 0] //dir = 1 - up

6 -> [0, 1, 0, -1] //dir = 1 - up
```

Figure 3.10. Example of the usage of the lookup table.

The algorithm is very simple: the first node, from which the path starts, is node 4. Since it is the upper left node, and the default direction is right, the next direction is still right, and as the movement continues along the positive x-axis of the Cartesian system, only the x coordinate is increased. The path continues along the upper border of the image, so edge node types 12 are met, and the behaviour is the same as the edge node type 4 until the upper right corner is reached. In this case the direction changes from right to down, and the movement is along the positive y-axis of the Cartesian system, so only the y coordinate is increased. Basically, when moving forward (right and down), coordinates are increased, while when moving backward (left and up) coordinates are decreased.

This step of the process takes the layers as input and uses path object. It is a quite complex object, with different properties, as shown in Figure 3.11.

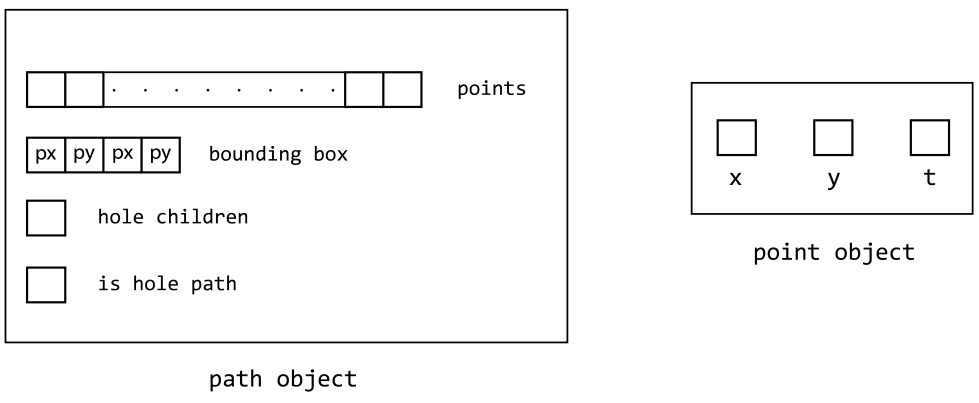


Figure 3.11. Path object and point object structure.

Each `path object` has the collection of all the points belonging to the path, the coordinates of the upper left corner and the lower right corner, i.e., the `bounding box`, the number of hole children (`holechildren`), and if the path has hole children, i.e., `isholepath` is a boolean set to `false` only if `holechildren` is equal to 0. Each point is an object as well, and it has three properties: `x` coordinate, `y` coordinate and the edge node type `t`.

Considering the path generated from the red layer of Figure 3.8, the output is reported in Figure 3.12.

```
path = [
   boundingbox : [0, 0, 9, 9],
   isholepath: false,
   holechildren: 0,
   points:[
           {x=0, y=0, t=4},
           {x=1, y=0, t=12},
           {x=2, y=0, t=12},
           {x=3, y=0, t=12},
           {x=4, y=0, t=12},
           {x=5, y=0, t=12},
           {x=6, y=0, t=12},
           {x=7, y=0, t=12},
           {x=8, y=0, t=12},
           {x=9, y=0, t=8},
               .......
           {x=9, y=9, t=1},
           {x=8, y=9, t=3},
           {x=7, y=9, t=3},
           {x=6, y=9, t=3},
           {x=5, y=9, t=3},
           {x=4, y=9, t=3},
           {x=3, y=9, t=3},
           {x=2, y=9, t=3},
           {x=1, y=9, t=3},
           {x=0, y=9, t=2},

               .......
     ]
  ]
```

Figure 3.12. Example of path object.

At the very end of this step an array of path objects, containing all the founded paths for a given color, is returned. Paths shorter than `PATHOMIT` are neglected.

3.5 Interpolation

The aim is to interpolate the obtained paths to make them smoother. Each line segment in the path is associated to one of the 8 possible directions: North, South, East, West, North-East, North-West, South-East, South-West, so the 8-directional chain code representation is used (see Section 2.5.1). Each direction is represented as a number from 0 to 7 and is computed considering the current pixel and its next. The code snippet reported in Figure 3.13 shows how directions are calculated:

```
this.getdirection = function( x1, y1, x2, y2 ){
       var val = 8;
       if(x1 < x2){
          if (y1 < y2){ val = 1; } // SouthEast
          else if(y1 > y2){ val = 7; } // NE
          else { val = 0; } // E
       }else if(x1 > x2){
          if (y1 < y2){ val = 3; } // SW
          else if(y1 > y2){ val = 5; } // NW
          else { val = 4; } // W
       }else{
          if (y1 < y2){ val = 2; } // S
          else if(y1 > y2){ val = 6; } // N
          else { val = 8; }// center, this should not happen
       }
       return val;
}
```

Figure 3.13. Example of path object.

At this level a very simple linear interpolation is used (based on pixels' coordinates), but right angles interpolation is thus enhanced to help the following steps. To identify right angles 5 points are considered: both the two following and the two preceding the current point and the current point itself. If a right angle is detected, and `RIGHTANGLEENHANCE` option is `true`, the current point is neglected, then the previous and the following points are interpolated using a median (Figure 3.14).

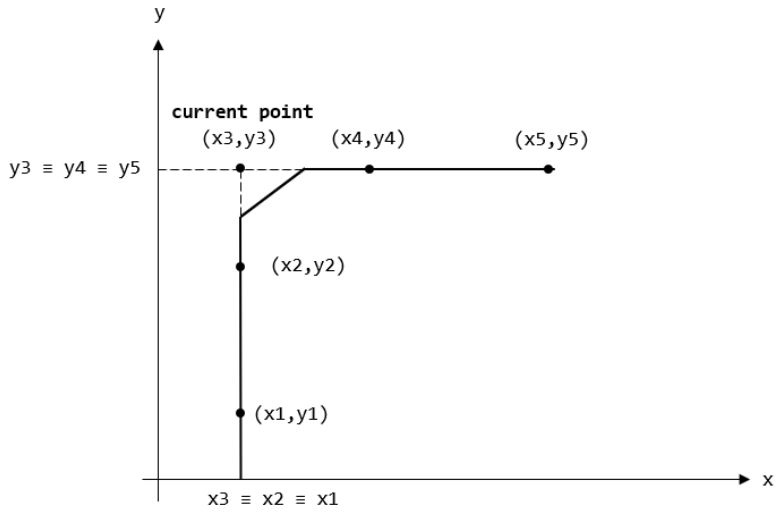


Figure 3.14. Right angle enhanced.

Interpolation is also used to thicken points to have better accuracy in the final result. So the overall purpose is to interpolate points and to assign at each line segment one direction. The output is still a path object, but in this case there is the linesegment type in place of the edge node type.

Considering the path generated from the red layer of Figure 3.8, and supposing that `RIGHTANGLEENHANCE = false`, the interpolated path is as given in Figure 3.15.

```
path = [
   boundingbox : [0, 0, 9, 9],
   isholepath: false,
   holechildren: 0,
   points:[
         {x=0.5, y=0, linesegment=0},
         {x=1.5, y=0, linesegment=0},
         {x=2.5, y=0, linesegment=0},
         {x=3.5, y=0, linesegment=0},
                  .......
                  .......
                  .......
         {x=2.5, y=9.5, linesegment=4},
         {x=1.5, y=9.5, linesegment=4},
         {x=0.5, y=9.5, linesegment=4},
               .......
    ]
 ]
```

Figure 3.15. Example of interpolated path object

3.6 Tracing and SVG coordinates generation

The aim of this step is to fit linear or quadratic spline segments on the 8 direction inter-node paths, splitting the interpolated paths into sequences with only two directions. The algorithm works as follows:

- Try to fit a straight line between the starting and ending points of the sequence;
- Calculate the distance between points belonging to the linear spline and the actual sequence of points;
- Use `LTRES` as threshold to decide if the straight line is good enough or not, i.e., evaluating if the error is below the threshold or not;
- If the straight line does not fit the sequence, select the point with the biggest error;
- Fit a quadratic spline through the control point (project this to get the *control point*), then measure errors on every point in the sequence;
- Use `QTRES` as threshold to decide if the quadratic spline fits or not the sequence;
- If the quadratic spline does not fit, find the point with the biggest error and mark it as *splitting point*;
- Split the sequence in two: starting point – splitting point and splitting point – ending point;
- Recursively apply the fitting algorithm to each sub-sequence.

At the very end of the fitting algorithm each path is marked as straight or quadratic. To help SVG string creation two different markdowns are used to create a `segment-type`:

- `L`: straight line going from (x_s, y_s) to (x_e, y_e);
- `Q`: quadratic line going from (x_s, y_s) to (x_e, y_e) and passing through an intermediate point, the *splitting point*, (x_i, y_i).

The fitting algorithm takes the `path object`, `ltres`, `qtres` and both the starting and ending points, i.e., (x_s, y_s) and (x_e, y_e) as inputs. To evaluate the goodness of the fitting sequence towards the threshold, RMSE is used as measure.

At the end of tracing step, involving the fitting algorithm for each of the previously computed paths, the `path object` is still returned, but it has an array of `segment-types` in place of `points`. Considering the example of the interpolated path shown in Figure 3.15, the `path object` returned at the very end of this step would be as reported in Figure 3.16.

```
path = [
   boundingbox : [0, 0, 9, 9],
   isholepath: false,
   holechildren: 0,
   segments:[
      {type= "L", x1=0.5, y1=0, x2=9, y2= 0},
      {type= "L", x1=9, y1=0.5, x2=8.5, y2= 9},
      {type= "L", x1=8.5, y1=9, x2=0, y2= 7.5},
      {type= "L", x1=0, y1=7.5, x2=0.5, y2= 0}
   ]
]
```

Figure 3.16. Example of traced path object.

At this point the original image has been divided into layers, one for each color. For each of these layers a series of paths have been identified, and from each of these paths a sequence of segments has been extracted. Finally, each segment has information on its type, linear or curvilinear, and on the start and end points.

SVG coordinates, collected in the SVG file, are just a string obtained using the syntax explained in Chapter 2.1:

- The string is initialized with `<svg>` tag;
- `width` and `height` properties are set;
- Each path starts with `<path>` tag;
- For each path `fill`, `stroke` and `stroke-width` properties are set using the palette generated in the first steps;
- Each segment of the path is rendered using `moveto ('M')`, `lineto ('L')` or `curveto ('C')` depending on the `segment-type`;
- Each path is closed with `<Z>` tag.

so the overall structure of the generated SVG file is as reported in Figure 3.17.

```
<svg id="svg-image" width="214" height="213">
 <g id="m1">
  <path fill = "rgb(255,255,255)" stroke="rgb(255,126,126)"
     stroke-width="1" opacity="1" d="M 194.5 18 L 194.5 19 L
     194.5 18 Z "/>
  <path fill = "rgb(255,255,255)" stroke="rgb(255,126,126)"
     stroke-width="1" opacity="1" d="M 194.5 18 L 194.5 19 L
     194.5 18 Z "/>
     .......
 </g>

 <g id=m8">
   <path fill = "rgb(255,255,255)" stroke="rgb(255,126,126)"
      stroke-width="1" opacity="1" d="M 194.5 18 L 194.5 19 L
      194.5 18 Z "/>
   <path fill = "rgb(255,255,255)" stroke="rgb(255,126,126)"
      stroke-width="1" opacity="1" d="M 194.5 18 L 194.5 19 L
      194.5 18 Z "/>
      .......
 </g>
</svg>
```

Figure 3.17. General structure of the generated SVG file.

As mentioned before, the SVG file is customized on Scribit's needs, in particular colors are grouped, i.e., one group for each color (layer), using the `<g>` tag. Moreover, a unique id is associated to each group, representing the marker to use. At the moment, Scribit does not support fills, i.e., the assumption is that only the contours are to be drawn. This is due to back-end conversion in g-code but also for timing reasons: right now drawing filled images requires too much time. To remove fill, the `fill` property of the `<path>` is set equal to `rgb(255,255,255)`, i.e. is set to white. White fills are discarded by the conversion algorithm performed on the back-end side.

Furthermore, it is possible to decide between straight line, dashed-line and pointed-line. Thanks to the `stroke` properties, it is possible to render different kinds of lines. The code snippet showing how it is done in this case is reported in Figure 3.18.

```
if (line == 0)
  var add = "";
else if (line == 1)
  var add = 'stroke-dasharray="10,10" ';
else
  var add = 'stroke-dasharray="1,5" stroke-linecap="round" ';
```

Figure 3.18. Setting of the type of the line.

Here, `line = 0` stands for straight, `line = 1` stands for dashed and `line = 2` stands for pointed. These properties are then added to the `<path>` tag.

3.7 SVG - GCode conversion

For the sake of completeness, in the following a brief description of the SVG/gcode conversion, developed by the other master's candidate, is provided.

The idea behind the conversion algorithm lies on some consideration about SVG paths: each SVG file is made up by both shapes and paths and their coordinates are directly accessible from their specific tags. The main difference between them is essentially in the way in which the points are defined: `shape` has only the starting point of the path (the upper left corner in case of lines and polygons), or the centre in the case of circumferences and ellipses, while `path` has both the starting and end points and the control point of the curves. The output file, i.e., the NC listing, must contain discrete commands point by point, so first of all geometric shapes are converted into paths and then only paths are considered when actual conversion is performed.

The second step is to scale the working area to the desired size of the design on the wall. These information are available in the `<svg>` tag, via `width`, `height` and `viewBox` attributes. The scale factor is computed using the actual dimensions and the desired ones, and is used to modify the `width`, the `height` and the `viewBox` of the SVG file. After scaling, it is possible to further modify the file, if required by the user, mirroring it with respect to the Y axis. This option has been made available to users who want to print something on windows and transparent surfaces in general. Scribit, in fact, is designed to work indoor, and a drawing that must be appreciated and/or understood from the outside (e.g., in stores) will be mirrored if printed on the inside of the wall.

Now it is possible to proceed with the extraction of the commands within the paths, and consequently all the points: each instruction of each path must be treated separately and translated into GCODE according to its meaning. For example, all the commands related to the points (`moveTo`, `lineTo` etc.) will be translated into `G1` instructions: it is the instruction of a linear movements and takes as inputs the Cartesian coordinates. It is essential to distinguish between absolute or relative coordinates, because it is necessary to know at every instant the current point in

which the marker is located to properly add the relative point to coordinates. A separate case is represented by the curves, that exposing only the end point and control points, need to be interpolated in several linear movement commands, dividing the curve into segments and their length (and consequently the total number of them) represents the degree of approximation in inverse proportionality.

After GCode file generation, the script creates the file used for erasing, if requested by the user. To erase a drawing, an SVG image is created containing only horizontal and vertical segments that trace the scaled drawing.

Chapter 4

Results

The overall rendering algorithm is designed to be very efficient, i.e., to have a rendered SVG image as close as possible to the original image, at least on paper. Performance tests have been executed to measure how much good the proposed solution is. They can be divided into two groups: tests with SVG fill and tests without SVG fill.

Each test has been run considering two main parameters: the `blurdelta` and the `numberofcolors`. Moreover, a "difference image", i.e., a sort of negative[1], is used to compare the color similarity between the original image and the SVG image. The difference image (`dimgd`) is built pixel by pixel considering the RGBA distance between the original image (`oimgd`) and the rendered one (`imgd`), as reported in Figure 4.1.

```
for(var j=0; j<imgd.height; j++){
  for(var i=0; i<imgd.width; i++){
   dimgd.data[idx ] = Math.abs( oimgd.data[idx ] -
       imgd.data[idx ] ); //R
   dimgd.data[idx+1] = Math.abs( oimgd.data[idx+1] -
       imgd.data[idx+1] ); //G
   dimgd.data[idx+2] = Math.abs( oimgd.data[idx+2] -
       imgd.data[idx+2] ); //B
   dimgd.data[idx+3] = Math.abs( oimgd.data[idx+3] -
       imgd.data[idx+3] ); //A
  }
}
```

Figure 4.1. Difference image computation.

[1] In photography, a negative is an image in which the lightest areas of the photographed subject appear darkest and the darkest areas appear lightest.

Since the RGBA values for `black` and `white` are, respectively, `(0,0,0,0)` and `(255,255,255,0)`, the more the RGBA distance is small, i.e., the color of the SVG image is as similar as possible to the original one, the more the difference image tends to black, and vice versa.

Furthermore, to better interpret the results some measures, such as *tracing time*, *rendering time* and *RGBA difference*, are computed.

4.1 Tests with SVG fill

The first set of tests takes as reference image a very simple image, i.e., a cartoon style image.

The first test (Figure 4.2) is run with `blurdelta = 100` and `numberofcolors=4`.

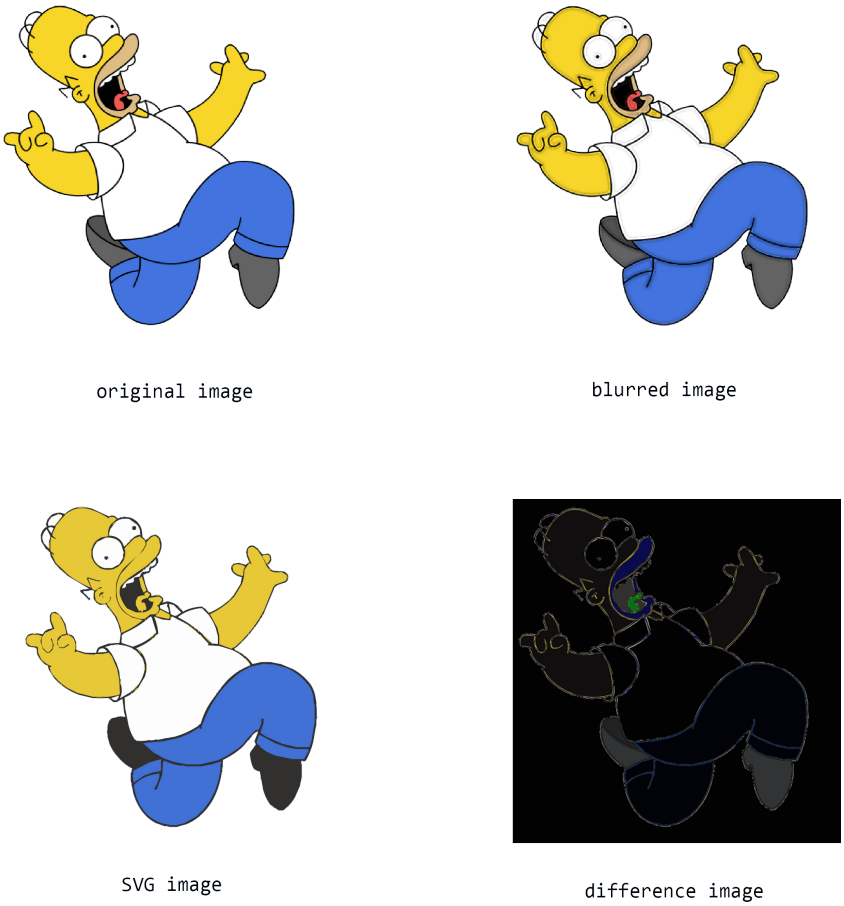


Figure 4.2. Test 1

Since the image is very simple, blurring has no effect, and the rendered image is quite good except the tongue and the beard (see Figure 4.2 - difference image). This is because `numberofcolors = 4`, and the averaged palette colors (Chapter 3.2.2.) is `yellow` for the skin, `blue` for jeans, `white` for the shirt and `grey` for shoes, so the beard and the tongue are clustered with the nearest color, in a RGBA sense, i.e., the yellow.

For this reason, a second test has been run (Figure 4.3) with `blurdelta = 10` and `numberofcolors = 16`. As expected, now the tongue and the beard are correctly rendered since `numberofcolors` is higher than in the previous test (note that `blurdelta = 10` since blurring has no effect).

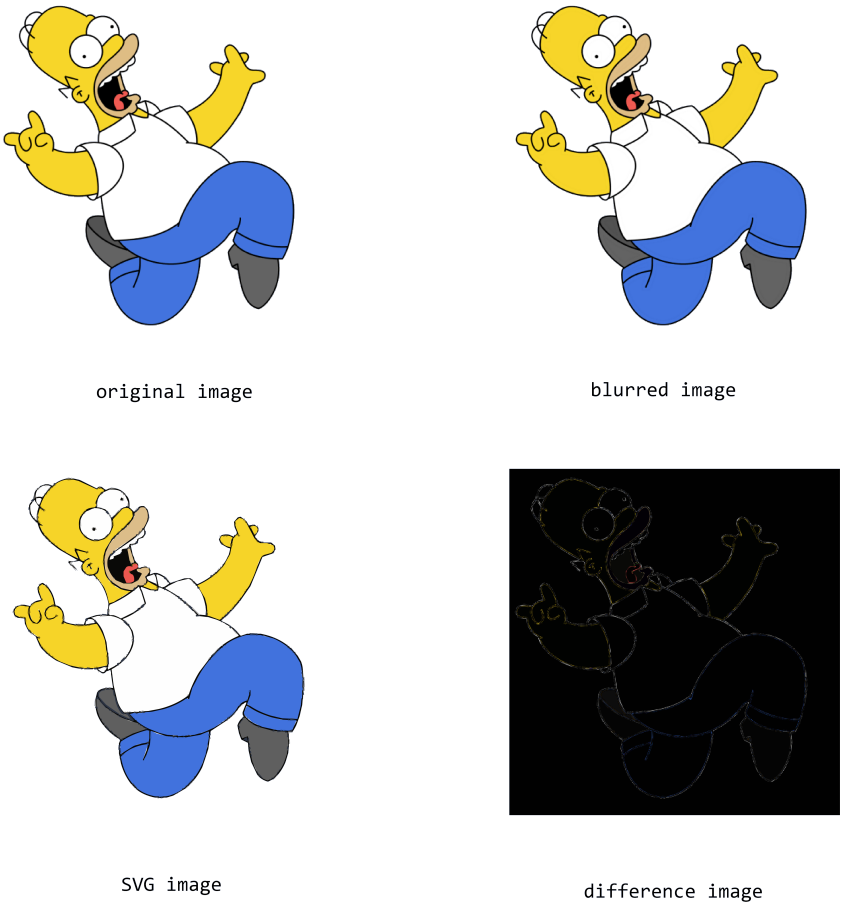


Figure 4.3. Test 2

As mentioned before, some numerical measures are computed to better interpret and compare the tests. Table 4.1 and 4.2 show, respectively, the results for both Test 1 and Test 2. On the one hand, since the second image is more accurate, because of the tongue and the beard, the RGBA difference is lower, on the other hand both the tracing and the rendering time are higher, i.e., higher accuracy is more time consuming.

Tracing time (ms)	**Rendering time (ms)**	**RGBA difference**	**Nr. of paths**
650	5	6.54	676

Table 4.1. Test 1 measures

Tracing time (ms)	Rendering time (ms)	RGBA difference	Nr. of paths
995	17	2.86	676

Table 4.2. Test 2 measures

The second set of tests takes as reference image a more complex image, with more colors and more details. The first test (Figure 4.4) is run with `blurdelta = 10` and `numberofcolors = 4`; `blurdelta = 10` to prevent loss of details, since the image is quite complex.

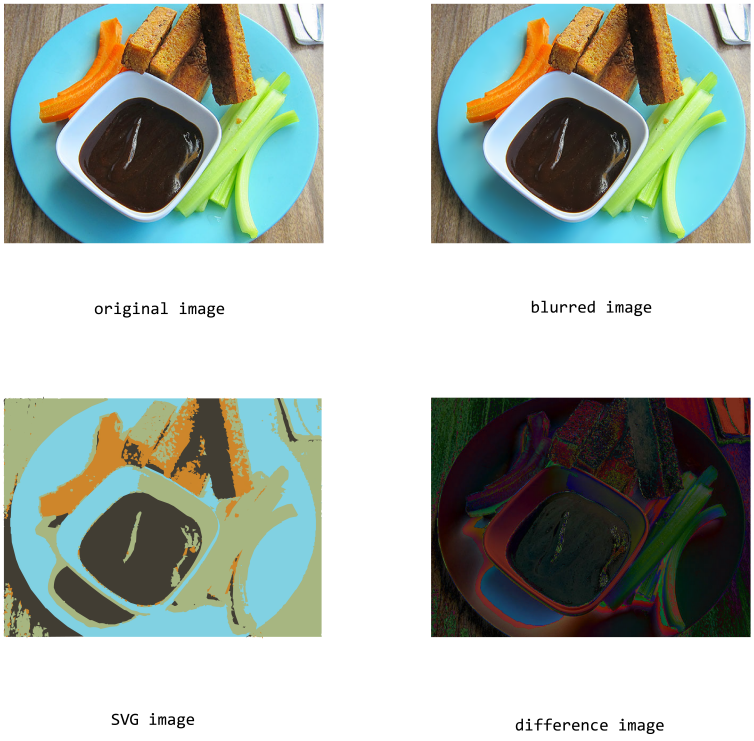


Figure 4.4. Test 3

The resulting SVG image is quite complex as the original image, although some details are lost because of the small size of the palette. The aim is to have as simple as possible SVG images because the final target is the plotter, i.e., the clearer the SVG image is, the better and the faster the plotting is. So, to simplify the image a higher `blurdelta` can be used and Figure 4.5 shows the results of this test.

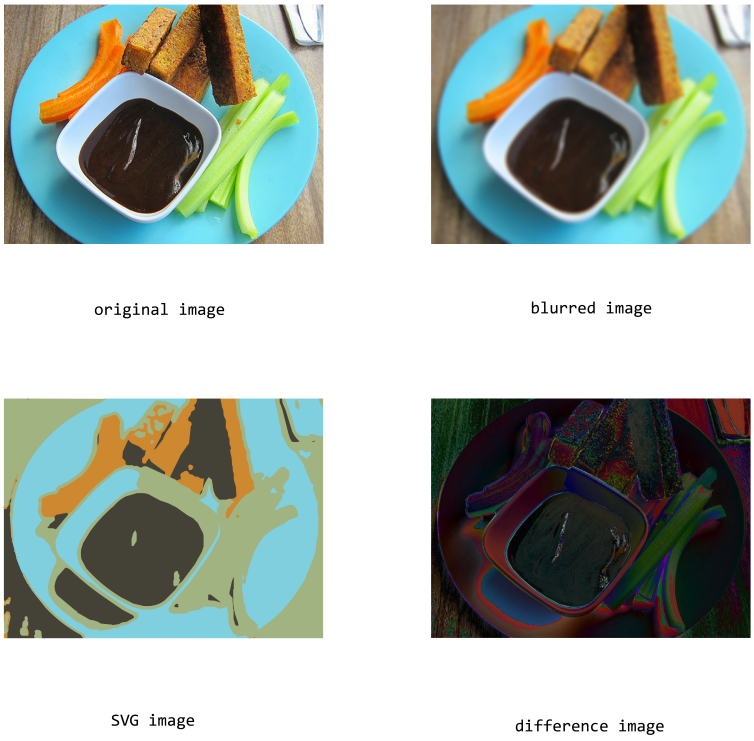


Figure 4.5. Test 4

In this case the SVG image is more smoothed and simple, but comparing Table 4.3 and Table 4.4 is clear that simpler image, i.e., lower rendering time and lower number of paths, means higher RGBA difference.

Tracing time (ms)	**Rendering time (ms)**	**RGBA difference**	**Nr. of paths**
680	33	20.9	1615

Table 4.3. Test 3 measures

Tracing time (ms)	**Rendering time (ms)**	**RGBA difference**	**Nr. of paths**
715	5	22.34	103

Table 4.4. Test 4 measures

It is possible to reduce the RGBA distance by increasing the size of the palette. So, another test is run with `blurdelta = 1024` and `numberofcolors = 16`; Figure 4.6 shows the results.

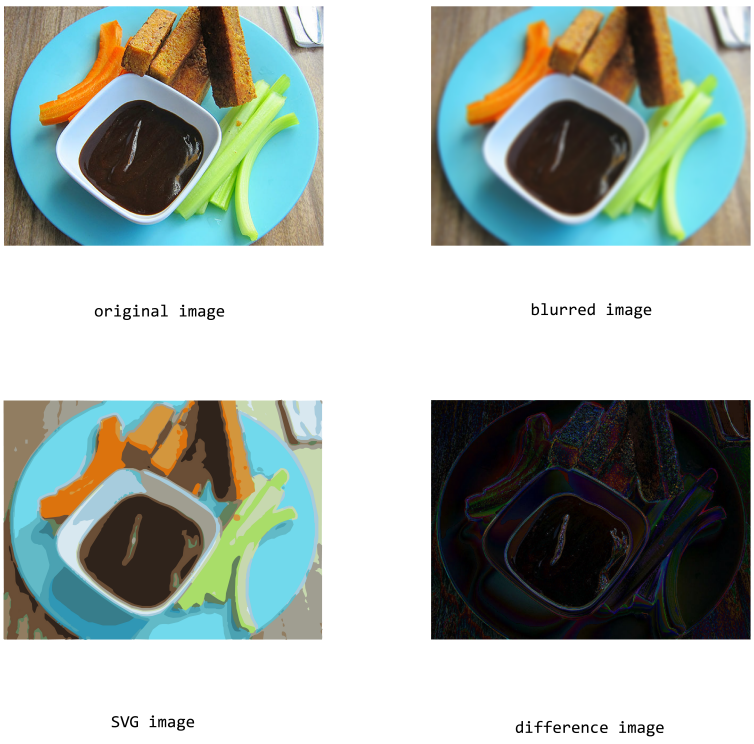


Figure 4.6. Test 5

As Table 4.5 shows, the RGBA distance is halved but the SVG image is more complex and both the tracing and the rendering time are increased.

Tracing time (ms)	**Rendering time (ms)**	**RGBA difference**	**Nr. of paths**
910	15	11.9	360

Table 4.5. Test 5 measures

To try to find a good trade-off, one last test is run with `blurdelta = 10` and `numberofcolors = 16`.

Figure 4.7 shows that in this case the rendered SVG image is very realistic and much closer to the original one, but Table 4.6 shows that both the tracing time and the number of paths is very high, despite the RGBA difference is further decreased.

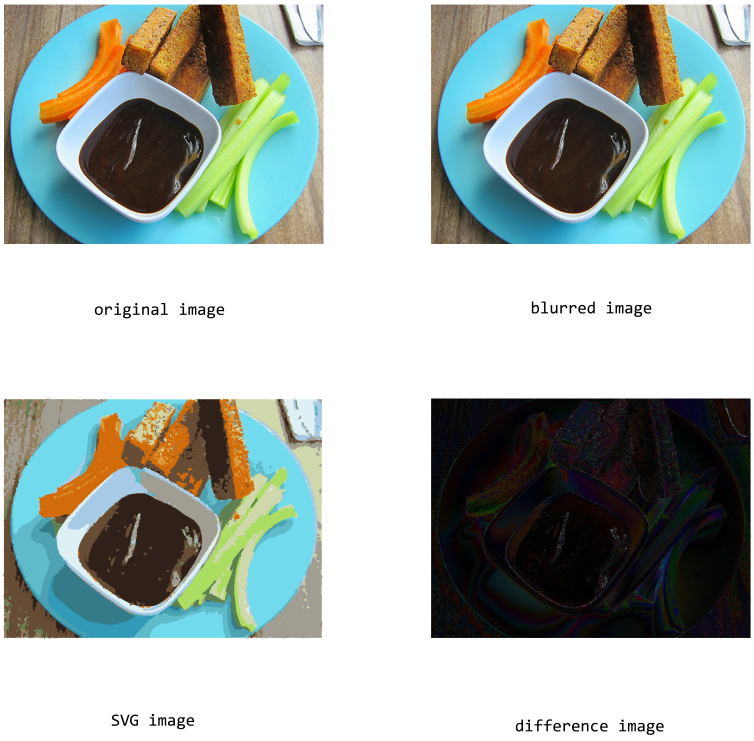


Figure 4.7. Test 6

Tracing time (ms)	Rendering time (ms)	RGBA difference	Nr. of paths
1216	62	10.1	4032

Table 4.6. Test 6 measures

The last set of tests takes as reference image a more complex one.

To have the resulting SVG image as accurate as possible, the first test is run with `blurdelta = 10` and `numberofcolors = 16`. As expected, Figure 4.8 shows that the result is very accurate and all the details are preserved. This is further confirmed by Table 4.7, which shows a very low RGBA difference, although both the tracing and the rendering time are very high, i.e., this test is more time consuming.

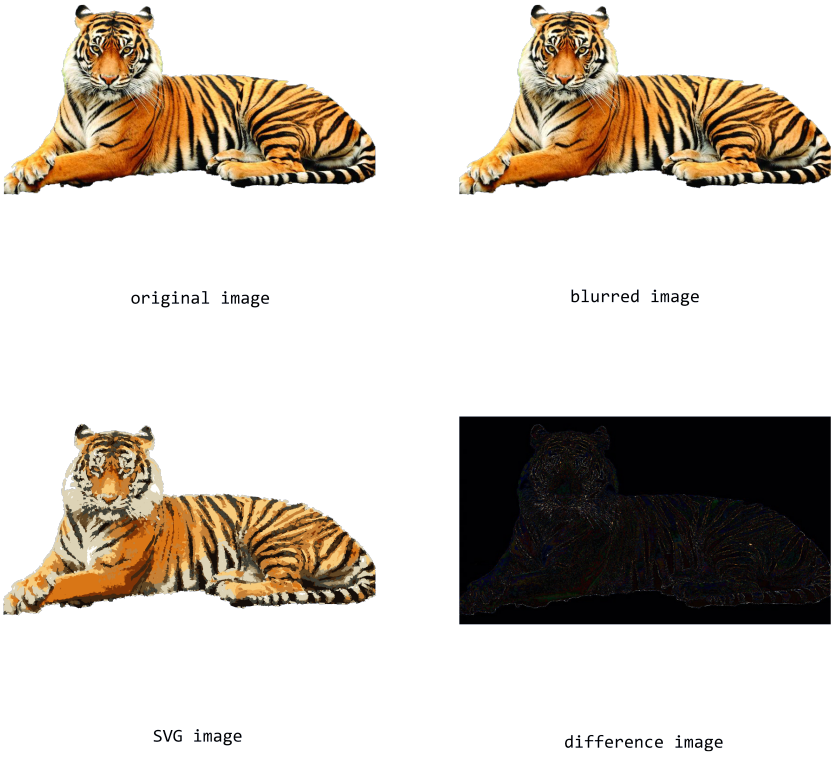


Figure 4.8. Test 7

Tracing time (ms)	Rendering time (ms)	RGBA difference	Nr. of paths
1001	105	7.35	8619

Table 4.7. Test 7 measures

To reduce the complexity and the rendering time, it is possible to use a higher `blurdelta` and a lower `numberofcolors`. So, the last test is run with `blurdelta = 100` and `numberofcolors = 4`. Table 4.8 shows that the RGBA difference is increased but either the tracing and the rendering time are decreased, as desired.

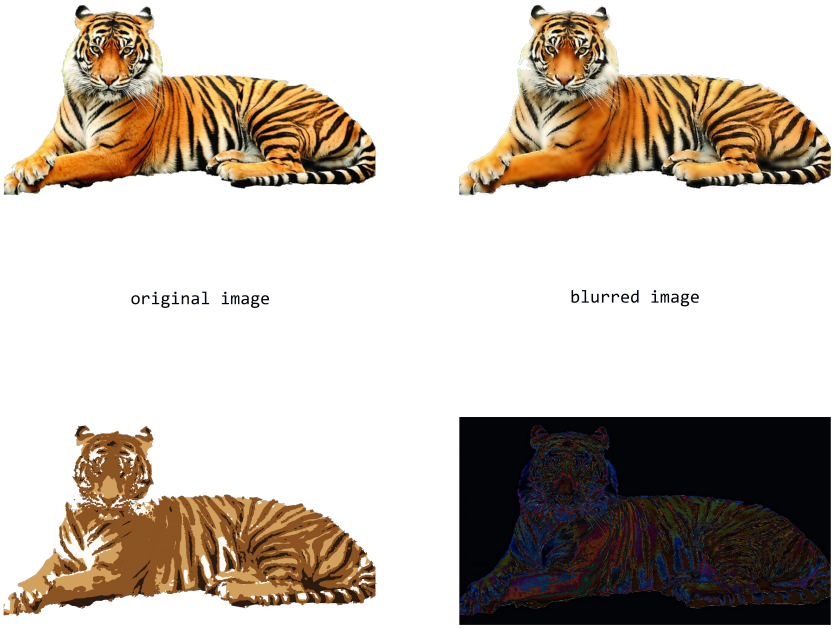


Figure 4.9. Test 8

Tracing time (ms)	Rendering time (ms)	RGBA difference	Nr. of paths
895	42	12.8	2168

Table 4.8. Test 8 measures

4.2 Tests without SVG fill

The rendering algorithm has been developed to be general purpose and to obtain a traced image as close as possible to the original one. The tests shown so far prove that it is possible to get different results, depending on the specific needs, by finding the right trade-off between the involved parameters.

The context of this thesis work, on the other hand, is more specific. The rendering algorithm must be used for a vertical plotter (Scribit), and therefore it is necessary to customize it on Scribit's needs. In particular, the goal of Scribit is to draw images that are clear and recognizable, and in the shortest possible time. For these reasons, at the moment fills are not supported, i.e., the gcode conversion algorithm expects SVG images that have only the contours. Moreover, the central drum (Chapter 1.1.1) can hold only 4 markers, i.e., the rendering algorithm should consider only 4 colors.

In the following, the `fill` property of the `path` tag is set to `(255,255,255,0)`, i.e., white, and the `numberofcolors` is assumed to be always 4. The images of the previous section are still used as reference images, and `blurdelta` is the only involved parameter. For these tests the RGBA difference is useless, since the fills are removed and the difference would be reasonably high.

The first test is run with `blurdelta = 10` and Figure 4.10 shows the result.

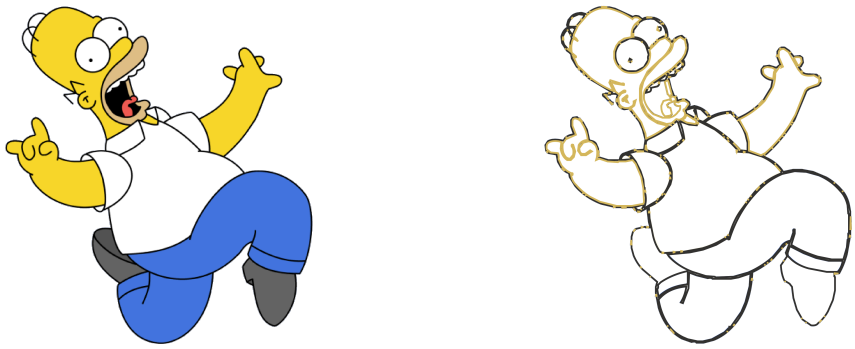


Figure 4.10. Test 9

In this case the image is very simple and its contours are well defined so the overall result is quite good. If a more complex image is considered, with less defined contours, and with the same `blurdelta`, the result is slightly less good. This is the case shown in Figure 4.11.

original image SVG image

Figure 4.11. Test 10

The resulting SVG image is not well defined and too complex to be drawn. To improve the result and to have a better defined contours, i.e., a simplified version of the original image, it is possible to increase the `blurdelta`. Figure 4.12 shows the result of the test run with `blurdelta = 1024`.

original image SVG image

Figure 4.12. Test 11

It is quite evident that removing fills causes loss of details and the images are not always fully recognizable. This is more clear by looking at Figure 4.13.

original image SVG image

Figure 4.13. Test 12

In this case, either the image and its contours are very complex and despite a quite low `blurdelta` is used, the rendered image is not recognizable at all.

4.3 Further tests: portraits and selfies

The previous section shows that, as a matter of fact, the proposed solution is not very effective when images have complex contours. This is further confirmed when the input image is a selfie or a portrait.

In this section, the picture of a famous actress, Angelina Jolie, is taken as a reference. Figure 4.14 shows the result with and without the SVG fill property.

original image SVG with fill SVG without fill

Figure 4.14. Portrait Test 1

It is clear that the SVG without the fill cannot be drawn, it is too complex and not recognizable, although the render with the full fill is very remarkable. This depends on the fact that a face has very detailed and complex features, and it should be as simple as possible, without losing its recognizability, to be drawable. This requires a very accurate edge-detection algorithm which is able to extract all the details and to create smoothed, simple, and clear lines.

The need to render portraits, or better still selfies, comes from the kind of experience that Scribit wants to offer its users: nowadays it is common practice to take selfies, and it cannot be excluded that the ordinary user is fascinated by the possibility of seeing his selfie drawn on the wall. For these reasons, the opportunity of designing your own selfie was announced as one of the additional features to be released soon.

To overcome the current limitations of the developed rendering algorithm, the last part of this thesis work is focused on the research and the study of existing solutions to draw faces and, generally, selfies, that are recognizable and with a "Scribit style".

There are plenty of solutions, but the ones that are more suited to Scribit's needs are *Coherent Line Drawing* [14] and *Line Draw* [15]. The first solution is more methodological, has a mathematical approach to the problem, dealing with vector fields, flows and gradients, and provides satisfactory results on paper (Figure 4.15).

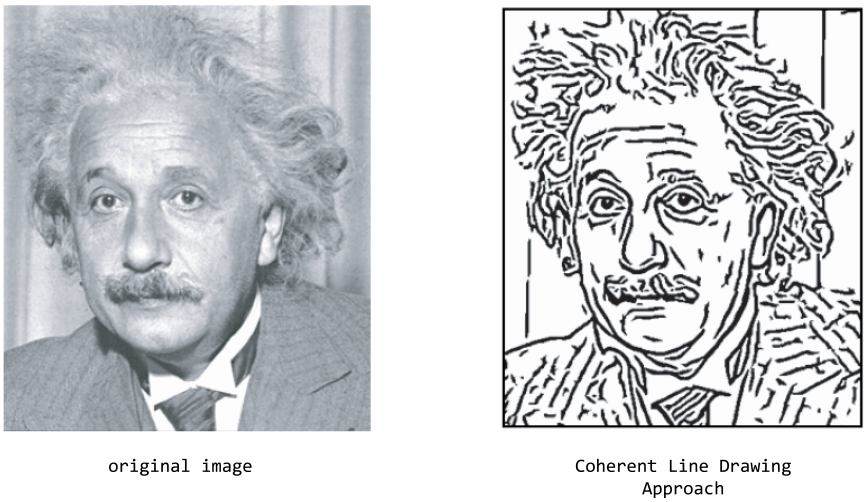


Figure 4.15. Coherent Line Drawing example (source: Coherent Line Drawing)

However, the second solution was chosen because, although it is simpler and technology based, it still provides good results. It is based on OpenCV [16], a multi-platform software library for real-time computer vision. It is a free software library originally developed by Intel, and, as the official website mention, it has more than 47 thousand people of user community and estimated number of downloads exceeding 14 million. Usage ranges from interactive art, to mines inspection, stitching maps on the web or through advanced robotics.

The `linedraw` algorithm is developed in python and it is basically based on the Canny Edge Detector [17]. It was designed in 1986 by John F. Canny and it uses a multi-stage calculation method to find the contours of many of the types normally found in real images. Canny algorithm aims at satisfing three main criteria:

- Good recognition: the algorithm must identify as many contours as possible in the image;

- Good localization: the identified contours must be as close as possible to the actual contours of the image;
- Minimum response: a given contour of the image must be marked only once, and, if possible, the noise present in the image must not cause the recognition of false contours.

To support the edge detection, the image is first transformed into black and white, then a function is applied to correct the contrast so as to make the difference between black and white clearer and more evident. Figure 4.16 shows this process.

Figure 4.16. Line Draw pre-processing

Figure 4.17 shows the final result, which is quite good with respect to the result shown in Figure 4.14: the image is clear, recognizable and drawable.

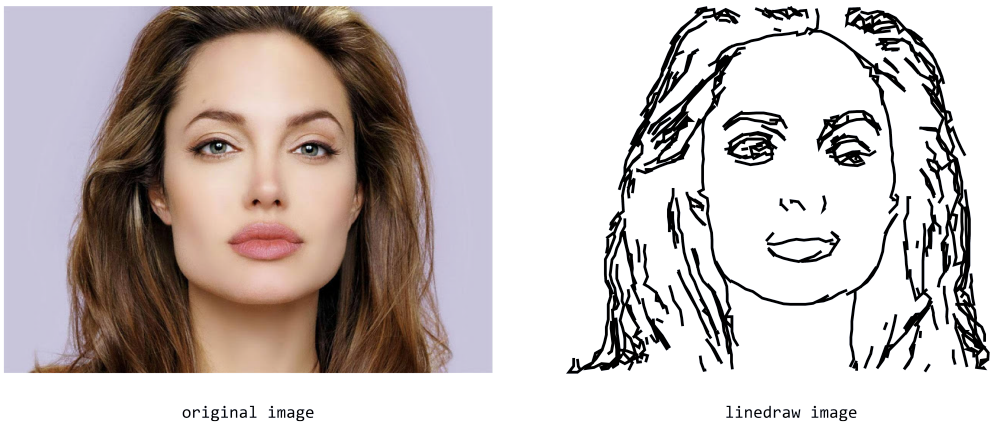


Figure 4.17. Line Draw Test 1

So far, this approach seems to be suited to Scribit's needs, but to further evaluate the goodness of this algorithm another test has been run with a real selfie as input image. As Figure 4.18 shows, the result is still good.

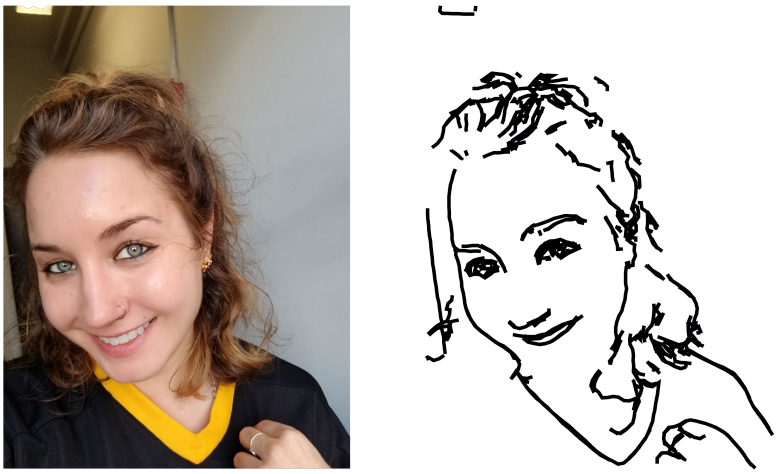


Figure 4.18. Line Draw Test 2

In this case either the background and the selfie itself are quite simple, so the result is clear and recognizable. Conversely, if the background is complex the result is not good as expected (Figure 4.19).

Figure 4.19. Line Draw Test 3

However, this algorithm provides a further feature that can be used to improve the result: it is possible to insert a stylized filling using a hatching algorithm. It consists in a series of horizontal lines that enhance the recognizability of the image, as shown in Figure 4.20.

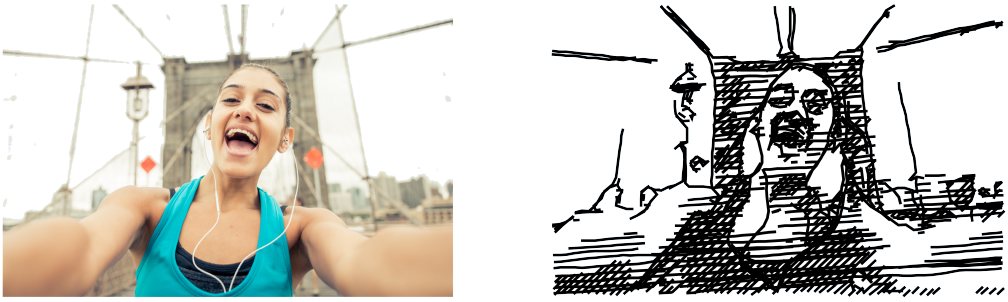


Figure 4.20. Line Draw Test 4

Obviously, the clearer the selfie the better the result. Nevertheless, hatching adds complexity and the required time to draw the image increases significantly, so the last effort of this thesis work focused on the optimization and the simplification of the hatching algorithm.

The key concept arise from the black and white conversion: the python library uses the ITU-R 601-2 luma transform [18], which encodes the RGBA values with a single value L. In this case, the value 16 is used for black and 235 for white, so to optimize the hatching algorithm it is possible to discard the background, i.e., the pixels above a given threshold that are near to 235 (white). Moreover, to simplify the hatching it is possible to decrease the number of lines. Figure 4.21 shows the final result of the optimization process. To be noticed that in this case a non-uniform background was chosen to further test this algorithm, contrary to the released guidelines.

Figure 4.21. Line Draw Test 5

4.4 Benchmarking

Benchmark means one or more tests specifically designed to evaluate the performance of a device, or the effectiveness of a technical process, or a financial instrument, with respect to a reference standard.

For software, benchmarking is not straightforward as it may seems, since too many parameters are involved. For instance, software development is often based on heuristics and programmers' choices, and over years different rigorous methodologies have been developed to investigate the behaviour of the algorithms.

In the case of computer vision, there is no real reference standard, and typically tests are based on comparing different approaches to the same problem. For this reason, in this section the proposed solution is compared with two vectorizing tools: Inkscape and Png or Jpg to SVG converter. Inkscape is a free and open source vector graphic editor for Windows, Mac OS X and Linux, Png or Jpg to SVG converter is one of the most used online converters, i.e., one of the first addressed by Google.

The three images used in the previous tests are still taken as a reference. To better highlight the differences between the different approaches, `numberofcolors` is assumed to be always 16, and only the case of images with fills is considered.

The first comparison is with Inkscape, and it considers the cartoon style image of Test 1 as example. Figure 4.22 shows that, except for the contours, the proposed solution works better than Inkscape since the resulting image seems to be very opaque, as if it had been blurred too much (it is not possible in Inkscape to chose neither the type of blurring nor the radius).

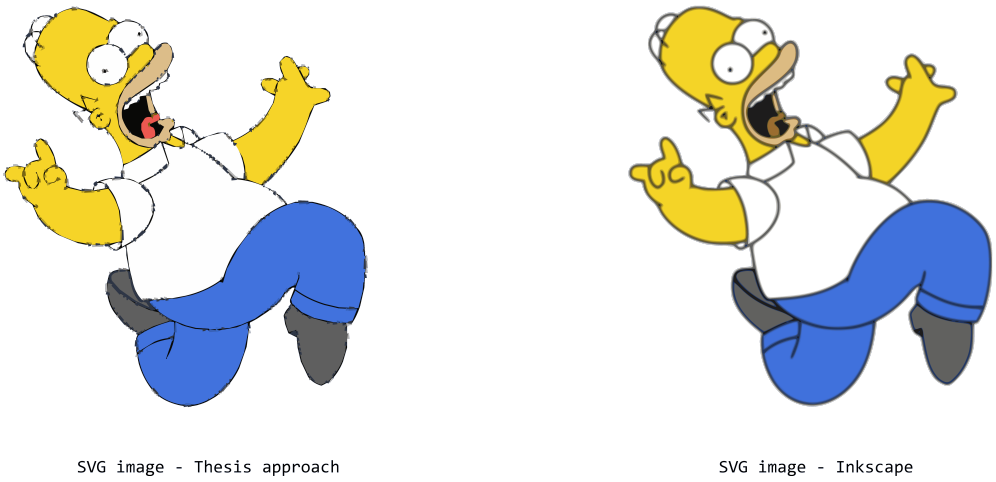
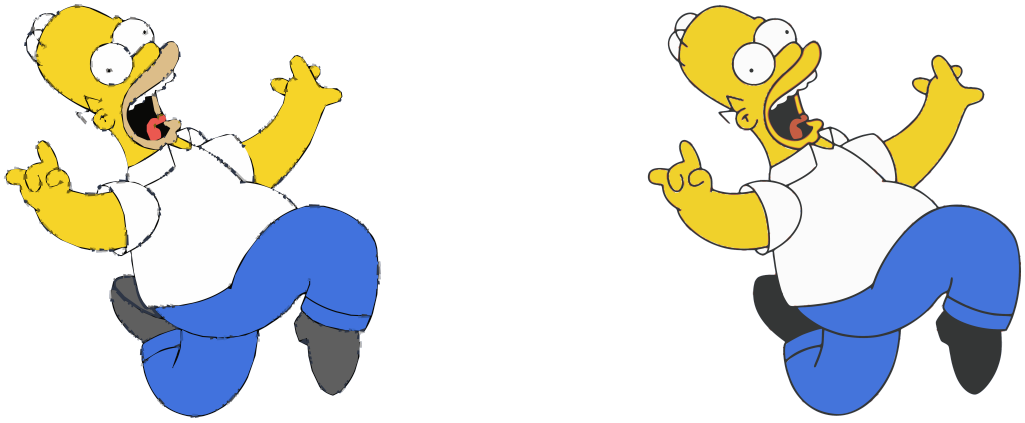


Figure 4.22. Comparison 1

If considering the online converter (Figure 4.23) the result is in principle the same, except for the beard which is correctly recognized only by this thesis approach.



SVG image - Thesis approach

SVG image - Online converter

Figure 4.23. Comparison 2

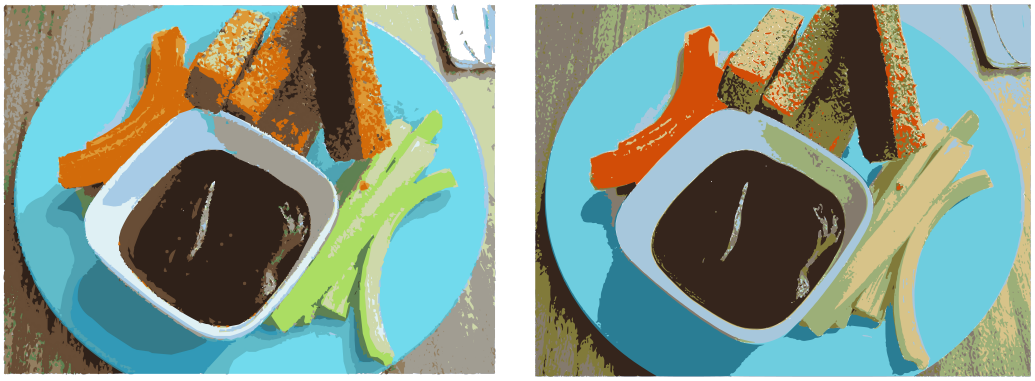
If considering the image of Test 3 as test image, there are very few differences between this thesis approach and Inkscape (Figure 4.24).

SVG image - Thesis approach

SVG image - Inkscape

Figure 4.24. Comparison 3

Instead, Figure 4.25 shows that in this case the result of the online converter is slightly different but mostly for the palette, so the overall result is still good.



SVG image - Thesis approach

SVG image - Online converter

Figure 4.25. Comparison 4

The last comparison takes the image of Test 5 as example. In this case Figure 4.26 and Figure 4.27 show that the behaviour is similar to that of the first comparison.

SVG image - Thesis approach

SVG image - Inkscape

Figure 4.26. Comparison 5

SVG image - Thesis approach

SVG image - Online converter

Figure 4.27. Comparison 6

Chapter 5

Conclusions

The subject of this thesis has been challenging for a number of reasons.

Surely there are very efficient and accurate SVG rendering algorithms, but Scribit wants the user experience to be the best possible, providing exactly what is needed, without too many frills or complications. Using third-parties services makes the user experience slower and more cumbersome, hence the need to have custom solutions to offer a complete 360° experience.

The first result to highlight is the efficiency of the algorithm: both the tracing and rendering time, as shown in the previous chapter, are in the order of milliseconds, regardless of the web browser used.

However, the main objective of this thesis work was never to have a rendering algorithm as fast as possible, but to have the resulting SVG file as faithful as possible to the original image. Indeed, the goodness of the overall rendering algorithm is very remarkable: Chapter 4 shows clearly that the result is very accurate, either with simple images or with complex images. The limitation is surely when fills are removed: if the image is too complex, with not well defined contours, the result is not clear and recognizable.

A viable solution would be to prefer a different edge-detection approach to have more smoothed and clearer contours. An idea could be to take as an example the python algorithm shown at the very end of Chapter 4: it is based on the Canny edge detector, which is a well-known and open algorithm, so it would be possible to provide a new implementation of this algorithm using JavaScript. In fact, the know how and the experience gathered about edge detection during the development and testing stages of this thesis work would be a good starting point.

Moreover, it would be interesting to exploit different color quantization approaches using other algorithms, e.g, octree. Color quantization could be further improved because at the moment colors with few pixels are randomized. Besides, other interpolation solutions, such as cubic splines or other curves, could be preferred.

Besides, it would be possible to play with the different parameters shown in Chapter 3 to provide more interesting and creative filters to further enrich the user experience.

In conclusion, the main goal this thesis achieved is to prove that such a rendering algorithm is not only possible, but also efficient, effective and usable. It also proved that the overall process can be improved by showing its limitations and some real solutions. Moreover, thanks to this thesis work there was the possibility to learn the fundamental of computer vision, such as color quantization, blurring and edge detection. This knowledge is very useful and it is the starting point for possible future research work in different application field such as recognition, robotics, autonomous driving and connectivity.

Bibliography

[1] VP Squared - Vertical plotting solutions
https://people.ece.cornell.edu/land/courses/ece4760/
FinalProjects/s2001/vp2/vp2miss.html
[2] Carlo Ratti Associati, https://carloratti.com
[3] Open Source Architecture, http://senseable.mit.edu/osarc
[4] L.Shapiro, G.Stockman, "Computer Vision"
Pearson College Div, 2001
[5] Scalable Vector Graphics (SVG), https://www.w3.org/Graphics/SVG/
[6] Guida SVG, https://www.html.it/guide/guida-svg/
[7] P. Tan, M. Steinbach, V. Kumar, "Introduction to Data Mining"
Pearson College Div., 2005
[8] R. Kress, "Numerical analysis"
Springer New York, 1998
[9] Interpolation by spline
https://www.math.uh.edu/ jingqiu/math4364/spline.pdf
[10] R. Jain, R. Kasturi, B. G. Schunck, "Machine Vision"
McGraw-Hill, 1995
[11] C. Maple, "Geometric design and space planning using the marching squares
and marching cube algorithms"
International Conference on Geometric Modeling and Graphics
London, July 2003
[12] MDN web docs
https://developer.mozilla.org/en-US/docs/Web/API/ImageData
[13] Color Thief, https://lokeshdhakar.com/projects/color-thief
[14] H. Kang, S. Lee, C. K. Chui, "Coherent Line Drawing"
http://umsl.edu/mathcs/about/People/Faculty/HenryKang/coon.pdf
[15] L. Dong, "Line Draw", https://github.com/LingDong-/linedraw
[16] OpenCV, https://opencv.org/
[17] Canny Edge Detector
https://docs.opencv.org/2.4.13.7/doc/tutorials/imgproc/imgtrans/
canny_detector/canny_detector.html
[18] ITU-R 601-2 standard, https://en.wikipedia.org/wiki/Rec._601