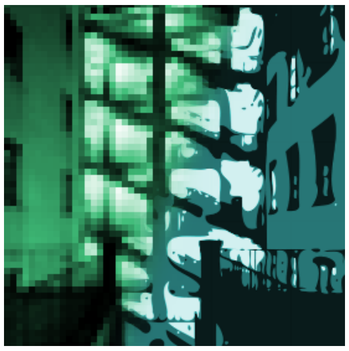


VECTRABOOL

M1 Integrated project — École Normale Supérieure de Lyon

Final report

Vladan Jovičić
(project leader)

Rémy Cerda
Rémi Coudert
Etienne Desbois
Quentin Guilmant
Emirhan Gürpınar
Ha Nguyen
Manon Philibert
Enguerrand Prebet
Hippolyte Signargout
Dewi Sintiari

Website: `https://vladan-jovicic.github.io/Vec-Lib/`
Code: `https://github.com/vladan-jovicic/Vec-Lib`

Contents

1 Project Summary

The aim of the project is to make a library for vectorization of a bitmap images, i.e., given an image in the bitmap format as input, transform it into vector image format SVG [1].

1.1 Motivation

Our aim was basically to get a simple image in a vectorized form, given a bitmap one as an input. Who did not already say once "well, it would be better if I could resize this image without losing the quality". Thanks to our project, one may be able to get a good approach of this image in vectorized format and then manipulate it as necessary.

Obviously we say simple image, the motivation was not about photography but mostly to have a tool that can cope with logos, diagrams... For instance if one needs to put one's institution logo on any LaTeX report or article but does not have any access to one with an appropriate size, it will be possible to vectorize it and resize it with great quality and, we aim to, accuracy.

1.2 Existing solutions

There are several softwares (commercial and open source) that can vectorize a bitmap image. Some of them are:

1. Commercial software:
   - Vector Magic
   - Printy
   - Image Vectorizer
2. Open source software:
   - Potrace
   - Vectorization.org
   - Autotrace

A natural question is why do we need then another software. Each mentioned software has some disadvantages. For example, Potrace and Vectorization.org work only with black and white images. Autotrace can handle also color images, but the output is not optimistic in some cases (see the results section). So, the answer on the above question is that we need a library that is combination of all features provided by the mentioned softwares, which is easy to use and which anyone can use.

1.3 Approach

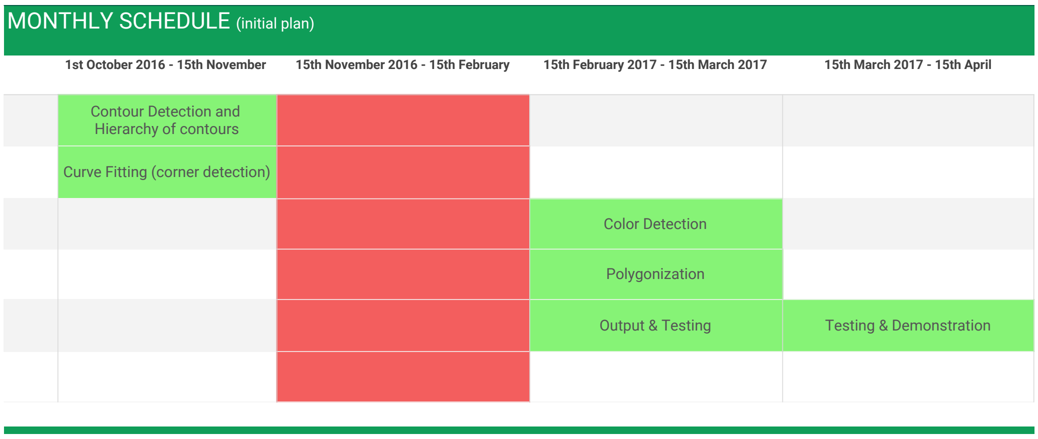
Our approach to this problem is to firstly detect different contours and make a hierarchy of contours. The obtained contours are approximated with Bezier curves since they can be represented on an easy way in the SVG format. Since each contour defines some region, based on the obtained contours we detect color

[1] See `https://en.wikipedia.org/wiki/Scalable_Vector_Graphics`.

inside each of them. We will restrict ourselves on monochromatic regions and regions that are colored with a transition from one color to the other. Many problems can occur in this package (check the initial project proposal[2]). When the above is done we plan to try to detect if some of the contours represents a polygon which can be represented in the SVG format. We call this polygonization. At the end, we need to merge obtained and to export to the SVG format.

Guided by our approach, we split our library into several parts (packages):

1. Contour detection and contour hierarchy detection
2. Curve fitting
3. Color detection
4. Polygonization
5. Output



MONTHLY SCHEDULE (initial plan)

	1st October 2016 - 15th November	15th November 2016 - 15th February	15th February 2017 - 15th March 2017	15th March 2017 - 15th April
	Contour Detection and Hierarchy of contours	🟥		
	Curve Fitting (corner detection)	🟥		
		🟥	Color Detection	
		🟥	Polygonization	
		🟥	Output & Testing	Testing & Demonstration
		🟥		

Figure 1: The initial schedule

[2]See https://docs.google.com/document/d/1rv0qQLA3jWaatWc40hIRV-x8QutbySnuLv5yGqzValU/edit?usp=sharing.

1.4 Technical details

We use the OpenCV library as a support. It provides many algorithms for an easy image processing. We use C++ (g++ 5.4.0 compiler) and Python 2.7 to implement packages. There are a few dependencies. A detailed specification is given in the code description (Github code). We also encourage the reader to check user guide ().

2 Contour Detection

In order to be able to draw the vectorized image, we first needed to be able to extract its contours.

2.1 Preprocessing

The image needs to be preprocessed first. There are mutliple possible ways of doing it such as Blurring, Thresholding, Sharpening for example. We chose to apply a Gaussian Blur.

First let's recall the (1D) Gaussian function :

$$g(x;\sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{x^2}{2\sigma^2}}$$

The Image is a function $f : \mathbb{Z}^2 \to \mathbb{Z}$ (for each color). In order to apply a Gaussian Blur to it, we first need to compute a $n \times n$ matrix (n odd) corresponding to the discretized version of g. Then we compute $(f * K_n) : \mathbb{Z}^2 \to \mathbb{Z}$, the convolution of f with K_n as :

$$(f * g)(x,y) = \sum_{i=-n}^{n} \sum_{j=-n}^{n} f(x-i, y-j) K_n(i,j)$$

It Can be done with two passes of the 1-D Version, and has to be done for all colors.

2.2 The Canny Algorithm

This Algorithm for contour detection was introduced by John F. Canny in 1986. Its principle is to detect edges by computing the image gradients. It works in 5 steps :

1. A Smoothing of the image (with Gaussian Blur here), with resulting image $h : \mathbb{Z}^2 \to \mathbb{Z}$
2. Computation of the image gradient in both direction and retrieve gradient norm and direction
3. Thresholding of the image with T, $h_T(x,y) = \begin{cases} h(x,y) & \text{if } h(x,y) > T \\ 0 & \text{otherwise} \end{cases}$
4. Suppression of non-maxima pixels to thin edges
5. Hysteresis phase : with two new thresholds (upper and lower), accept a pixel if either :
   - its gradient value is bigger than the upper threshold

- its gradient value is between upper and lower threshold and it is connected to a pixel whose gradient value is bigger than the upper threshold

The implementation was done thanks to OpenCV

2.3 Postprocessing

After the edge map have been computed by the Canny Algorithm, We apply a last processing to the Image. We chose to use a closing. A Closing is a Dilation followed by an erosion. The intuition behind the idea of dilation is to "push" the edges with with a small element such as a square of a circle, erosion is the converse.

Using a closing permits to close edges seen as different by the Canny Algorithm that are very close. This allowed us to be able to get a better edge map. We applied the closing with a 3×3 square.

3 Polgonization

3.1 Context

When we do the contour detection step, we use polylines. The major problem that one can notice is that a polyline may have a huge number of points. At the corner detection step we work thanks to those points. If there are too many of them, the complexity would increase and the accuracy may drop down. Basically, the algorithm may detect corner that do not exist in the actual image. To avoid that, we have to decrease the number of points. So, the polygonization problem can be formulated as follows: what is the smallest number of points on polyline such that we can approximate the polyline within a pre-defined error. The only assumption for this package is that points of a polyline are sorted in clockwise or counterclockwise order.

3.2 Integer programming

The first thing one can notice is that the problem can be expressed as an integer programming problem. Let $S \subseteq \mathbb{R}^2$. Let d be the usual euclidean distance over $\mathbb{R}^2$. Suppose we are given a distance δ such that each initial point of the input set of point must have a distance to the output polygon (or polyline) P at most δ, in other words

$$\forall e \in S \qquad d(e,P) \leq \delta$$

We also define for all $e \in S$, the interger variable x_e corresponding the the node e of S. Then the condition will be

$$\forall e \in S \quad \exists f, f' \in S \quad x_f = x_{f'} = 1 \qquad d(e,[f,f']) \leq \delta$$

Suppose now we are given e, f and f'. Let h be the orthogonal projection of e on $[f,f']$. Then

$$\vec{fh} = \vec{ff'} \frac{\langle \vec{fe}, \vec{ff'} \rangle}{\|\vec{ff'}\|^2}$$

Thanks to the projection h we can simplify and say

$$d(e,[f,f'])^2 = \langle \vec{fe}, \vec{fe} \rangle - \langle \vec{fh}, \vec{fh} \rangle$$

One may notice that the quantities we are working on are positive. Then squaring them does not change anything. The condition now becomes

$$\begin{array}{rll} & \forall e \in S & \min_{f,f'\in S}((\langle \vec{f}e, \vec{f}e\rangle - \langle \vec{f}h, \vec{f}h\rangle - \delta^2)x_f x_{f'}) < 0 \\ \Leftrightarrow & \forall e \in S & \sum_{f,f'\in S}(\langle \vec{f}e, \vec{f}e\rangle - \langle \vec{f}h, \vec{f}h\rangle - \delta^2)x_f x_{f'} < \sum_{f,f'\in S}|\langle \vec{f}e, \vec{f}e\rangle - \langle \vec{f}h, \vec{f}h\rangle - \delta^2|x_f x_{f'} \\ \Leftrightarrow & \forall e \in S & \sum_{f,f'\in S}(|\langle \vec{f}e, \vec{f}e\rangle - \langle \vec{f}h, \vec{f}h\rangle - \delta^2| - \langle \vec{f}e, \vec{f}e\rangle + \langle \vec{f}h, \vec{f}h\rangle + \delta^2)x_f x_{f'} > 0 \end{array}$$

Let $\tilde{a}_{e,f,f'}$ be the quantity

$$\tilde{a}_{e,f,f'} = |\langle \vec{f}e, \vec{f}e\rangle - \langle \vec{f}h, \vec{f}h\rangle - \delta^2| - \langle \vec{f}e, \vec{f}e\rangle + \langle \vec{f}h, \vec{f}h\rangle + \delta^2$$

We now need to have a fresh variable z for each tuple (f, f') that will represent the product of the variables corresponding to f and f'. Because all those variables can be seen as elements of $\{0,1\}$, one can express that zf, f' is the product of both the varaibles with three constraints and get the following problem

$$\begin{array}{lll} & \min \sum_{e\in S} x_e & \\ \text{respect to} & \forall e \in S & \sum_{f,f'\in S} \tilde{a}_{e,f,f'} z_{f,f'} > 0 \\ & \forall f, f' \in S & z_{f,f'} \leq x_f \\ & \forall f, f' \in S & z_{f,f'} \leq x_{f'} \\ & \forall f, f' \in S & z_{f,f'} \geq \frac{x_f + x_{f'}}{2} - \frac{1}{2} \\ & \forall e \in S & x_e \in \{0,1\} \\ & \forall f, f' \in S & z_{f,f'} \in \{0,1\} \end{array}$$

To simplify the problem we can relax the problem with $\tilde{a}_{e,f,f'} \in \{0,1\}$. In fact the goal of those variable are to say wheter or not e is close enough to $[f, f']$ to be forgetten we we take them in the final polyline or not. It is 0 if they are not good candidates, positive otherwise. Then let $a_{e,f,f'}$ defined as 1 if $\tilde{a}_{e,f,f'} > 0$ and 0 otherwise. Then we get the problem

$$\begin{array}{lll} & \min \sum_{e\in S} x_e & \\ \text{respect to} & \forall e \in S & \sum_{f,f'\in S} a_{e,f,f'} z_{f,f'} > 0 \\ & \forall f, f' \in S & z_{f,f'} \leq x_f \\ & \forall f, f' \in S & z_{f,f'} \leq x_{f'} \\ & \forall f, f' \in S & z_{f,f'} \geq \frac{x_f + x_{f'}}{2} - \frac{1}{2} \\ & \forall e \in S & x_e \in \{0,1\} \\ & \forall f, f' \in S & z_{f,f'} \in \{0,1\} \end{array}$$

This integer program was interesting but is difficult to solve. Basically, even if we get ride of the $\frac{1}{2}$ coefficients, if δ is big, the matrix is (trivially) not totally unimodular and then a sufficient condition to have integer vertices of the polytope is not satisfied. Then we will do a simpler algorithm.

3.3 Ramer-Douglas-Peucker algorithm

We still use a threshold disance ε. We will proceed recusivly. We chose the farthest point from le straight line that links the first and the last point and we call recursively on the two part we have drawn. The algorithm is as follows

```
Function RDP(points,epsilon) : point array
    Determine $i_{max} \geq 2$ the index of the farthest point from $[points[0]points[|points|-1]]$
    Let $d_{max}$ be the corresponding distance.
    If $(d_{max} \geq epsilon)$ then
        return $RDP(points[0:i_{max}-1])||RDP(points[i_{max}:|points|-1])$
    else
        return $points$
    end If
End
```

One can notice that it is a quicksort. One choose a pivot and then divides the original set into two parts thanks to this pivot and then proceed recursively. Then, the complexity is the same and we get

- average: $O(n \log n)$.
- worst case $\Theta(n^2)$.

4 Corner Detection

The next thing to do, when we have the polylines, is to find which of the polylines corners are real corners, and which ones could be contained in curves. This is what we call corner detection. For this we use the ShortStraw algorithm, designed by Wolin, Eoff and Hammond in 2008 [2]. We assume that points describing a polylines are sorted either clockwise or counterclockwise. The algorithm consists of three main steps:

- Resampling: Resample the points on the stroke so that each pair of consecutive points has the same distance between them;
- Bottom-Up Corner Finding: Compute the length of all straws between two points separated by a certain number of points, and set as corners the middle point of straws of locally minimal length;
- Top-Down Corner Checking: Verify that corners are corners and that lines are lines by checking the maximal distance between the stroke and a straight line between two corners.

4.1 Resampling

The idea of the algorithm is to fix a distance, draw straight lines between points at that distance on the stroke from one another, and say that there are corners where the length of the line is the smallest. For this, we need a way to calculate easily the stroke-distance between two points, and we will do this by resampling the points so that they are evenly spaced apart, so that the stroke-distance can be approximated by the number of points between two points.

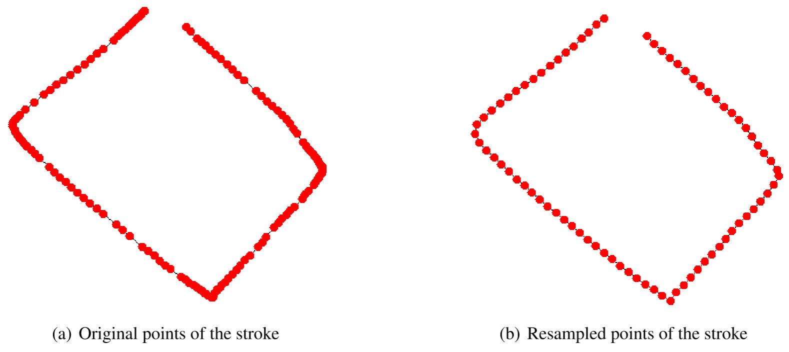

(a) Original points of the stroke (b) Resampled points of the stroke

Figure 2: Resampling of unevenly spaced points on a stroke

4.2 Bottom-Up Corner Finding

Once we have a stroke with points evenly spaced apart, we can start finding possible corners. We just fix w an integer and define $straw_i = |p_{i-w}, p_{i+w}|$, the Euclidian distance between the two points at stroke-distance w from p_i. The we compute the median straw length, set a parameter $t \in]0,1[$, and set p_k to be a corner is $straw_k$ is a local minimum below $median * t$.

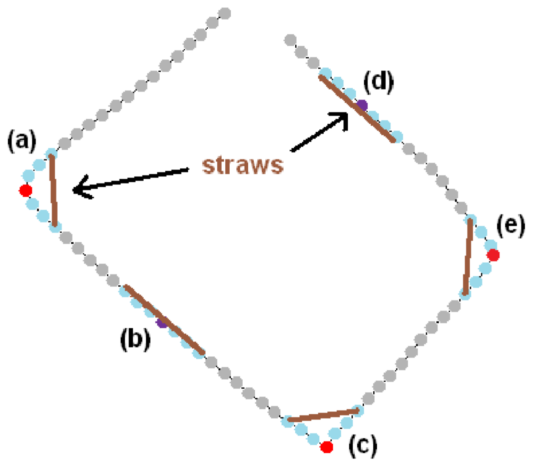


Figure 3: Example of straws on a stroke for $w = 3$

Our implementation does not fix nor find parameters t and w, it lets the user choose them.

4.3 Top-Down Corner Checking

After the initial set of corners is found by taking the shortest straws, some higher-level processing is run on the stroke to find missed corners and remove false positives.

The first thing is to make sure we have not missed corners. For each pair of consecutive corners, we compute $r = \frac{|a,b|}{stroke-distance(a,b)}$ and say there is a missing corner if $r < t$. If it is the case, we define the point c with straw between a and b and with minimum straw length to be a corner, and the verify that there are no missing corners between a and c and between c and b.

Once we have done that, we want to check that we do not have false positives, corners which should be part of lines, which is done by checking for three consecutive corners a, b, and c if $\frac{|a,c|}{|a,b|+|b,c|} > t$, and removing b from the corners if it is the case.

5 Curve fitting

The next step is to link the corners with curves, as close as possible to the stroke while avoiding overfitting. We chose to use cubic Bezier curves.

5.1 Cubic Bezier Curves

A Bézier curve of degree n is defined in terms of Bernstein polynomials:

$$Q(t) = \sum_{i=0}^{n} V_i B_i^n(t) \qquad t \in [0,1]$$

Where the V_i are the control points, and the $B_i^n(t)$ are the Bernstein polynomials of degree n:

$$B_i^n(t) = \binom{n}{i} t(1-t)^{n-i} \qquad 0 \leq i \leq n$$

When $n = 3$, the curves are called cubic.

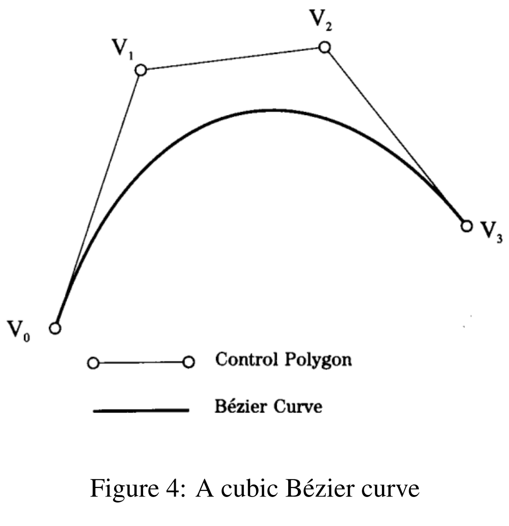


Figure 4: A cubic Bézier curve

5.2 Conditions

Now that we have defined the tools we are going to use, here are the conditions that we put on the curves so that they fit the stroke between two corners as well as possible:

- V_0 and V_3, the first and last control points, are given: they are set to be equal to the ends of the region we want to approximate, that is, the two corners;
- Define $\hat{t}_l$ and $\hat{t}_2$ the unit tangent vectors at V_0 and V_3, respectively;
- $V_l = \alpha_1 \hat{t}_1 + V_0$, and $V_2 = \alpha_2 \hat{t}_2 + V_3$; that is, the two inner control points are each some distance α from their the nearest end control point, in the tangent vector direction, so that the tangent of the curve at the corners are $\hat{t}_l$ and $\hat{t}_2$.

5.3 Solution

The idea is to find α_1 and α_2 minimizing S, the total quadratic distance between the curve Q and the stroke points p_i. That is, to find α_1 and α_2 for which $\frac{\delta S}{\delta \alpha_1} = 0$ and $\frac{\delta S}{\delta \alpha_2} = 0$

So, $\hat{t}_l$ and $\hat{t}_2$ can be found by minimizing square distances in the neighborhood of V_0 and V_3, and α_1 and α_2 can be found with a closed-form solution, in order to get, for each pair of corners, V_1, V_2, V_3 and V_4. These quadruples of Control points are what will be stored by our program.

5.4 Other implemented algorithms

We have also tried to implement other well-known algorithms for corner detection. One of them gave a reasonable results: Harris corner detection algorithm. We will not describe the algorithm in this report since an easy way to use that algorithm as a part of the library is not provided. However, all details can be found in article [1]. We don't provide a way to use the algorithm because of two reasons.

1. It is not the best solution for our problem since we want to find actually discontinuity points of a curve for which we know only several points. The notion of corner in the article describing Harris corner detection algorithm is much more general.
2. Using several tricks we were able to transform our problem to a special instance for Harris corner detection algorithm. However, the running time of the algorithm was not very optimistic: $O(k * width * length)$ where $width, length$ represent the width and length of the input image and k is the number of contours in the input image. Since value of k can be large for a very complex images and the performance obtained by the Harris corner detection algorithm is not better than the ShortStraw algorithm, we don't use it as a part of the library.

6 Color Detection

We considered two algorithms for the color detection. Indeed, the implemented algorithm is a very simple and naive, it gives a good results. Here we explain two algorithms that we try to implemented.

6.1 First algorithm

At the beginning we planned to implement algorithm to check the colour of the region given the polygon as its borders (although the borders are not exactly the polygon).

This algorithm runs as follow :

- For each polygon as contour, take three corners (y,x) on the contour
- For each corner, check its neighbourhood by taking some close points such as $(y+2,x)$, $(y-2,x)$, $(y,x+2)$ and $(y,x-2)$
- Check if these points are in the polygon using an openCV function named **pointPolygonTest**
- If it's in the polygon, consider its colour to determine the colour of the region by a majority vote.
- If none of these points are in the polygon, we would generate some random points and check whether they are in the polygon.

For not losing too much time to attend enough points in the polygon, we would compute the mean $\mu_x\mu_y$ and the standard deviation $\sigma_x\sigma_y$ of the coordinates of the corners of the polygon and take random points with coordinates in $\mu_x \pm 2\sigma_x$ and $\mu_y \pm 2\sigma_y$. This helps taking random points near the region rather than possibly far parts of the entire image. Afterwards we proceed in the same way with majority vote.

An additional feature of this algorithm was to be able to check if a point is in a region R and not in any region contained in the contour of the region R. To have this property, we wanted to use the hierarchy tree of contours. The idea was to check for each child (corresponding to inner contours) of the node (corresponding to R), whether the point is in or out. To be in the region R, a point must satisfy two conditions : being inside the contour of R and outside the contour of its children. Unfortunately, a lot of unexpected situations and cases appeared while trying to handle all the structures. So, we decided to use a similar algorithm but easier to implement.

6.2 Algorithm implemented

Consequently we used another algorithm for the entire colour detection package. The general idea of the algorithm is as the following:

Algorithm: Colour Detection

```
1:  Define a structure for keeping different colors and their number of occurrences
2:  for a point in the contour do
3:      Draw a square with center at point and side length 3
4:      for a point pt in the square do
5:          if pt belongs to contour then
6:              check the color of pt
7:              add it to the structure (or increase its number of occurrences)
8:          end if
9:      end for
10:     return the majority color
11: end for
```

To check whether the point is in the polygon we use **Ray Casting** algorithm, although the coordinates are not integers but floating point numbers. An advantage of this algorithm is that depending on the contour detection, this algorithm is efficient in all cases. However it only works for monochromatic regions.

6.3 Comments

We haven't use other colour detection algorithms we know since they have other cons such as using all the pixels. It is hard to compare these algorithms, nevertheless for monochromatic case we are efficient. On one hand this is not the restricting package for the speed of the project, on the other hand working only on monochromatic images is a big restriction.

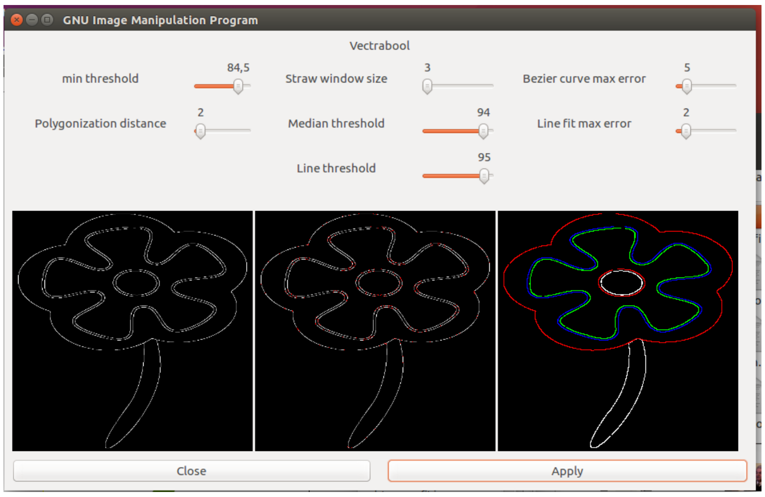
7 Results and user guide

In this section, we will present some examples and the results we achieved. Before we start with examples, let us describe the layout of the gimp plugin (see figure 5). In the upper part, there are various parameters that can be adapted for a different images. In the first group (the leftmost group) one can change lower threshold for the contour detection algorithm and the maximum error for the polygonization algorithm. In the middle group of upper part, one can set parameters for the corner detection algorithm: the length of the straw, the thresholds for median and line. The last group contains parameters for the curve fitting algorithm. One can set there the maximum error that is allowed when approximating polylines with either lines or Bezier curves.

In the lower part, there are three windows for displaying an intermediate results. The first window shows detected contours of the image, the second one shows contours together with corners (red points). The last window shows the output stroke of the image after approximation of polylines.

Now we will present some concrete examples and compare them with other softwares. We will also try to explain the values of parameters used for a particular images.

Remarks: To install the plugin, please follow the instructions provided in the code readme. To apply the plugin, open an image, go to *Filters-VUI* and the vectrabool window will appear. The output image is saved in the home directory named *output.svg*. All the examples presented here can be found in the code repository.

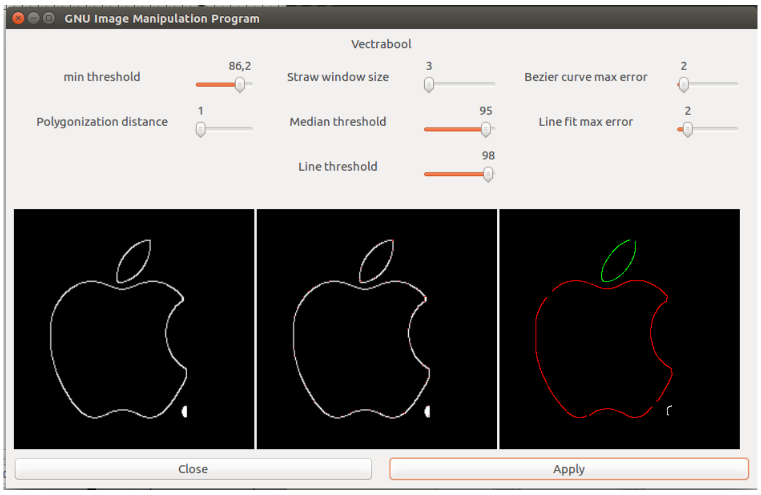


min threshold 84,5
Straw window size 3
Bezier curve max error 5
Polygonization distance 2
Median threshold 94
Line fit max error 2
Line threshold 95

Figure 5: The layout of gimp plugin

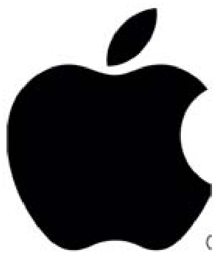
8 A simple example

We start by a simple example. For this example, each of the softwares produces a good result.



min threshold 86,2
Straw window size 3
Bezier curve max error 2
Polygonization distance 1
Median threshold 95
Line fit max error 2
Line threshold 98

Figure 6: Parameters for simple example


(a) The input image

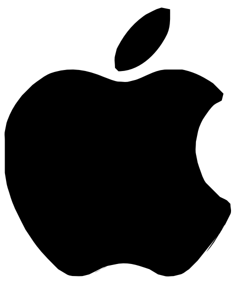
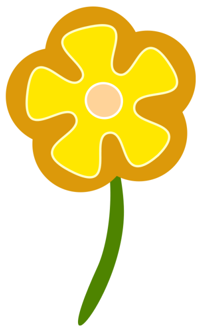

(b) The output image

Figure 7: A simple example

9 The flower example


(a) The input image

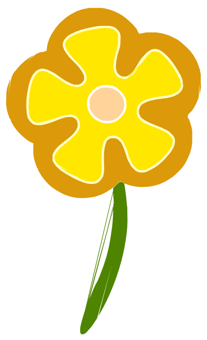
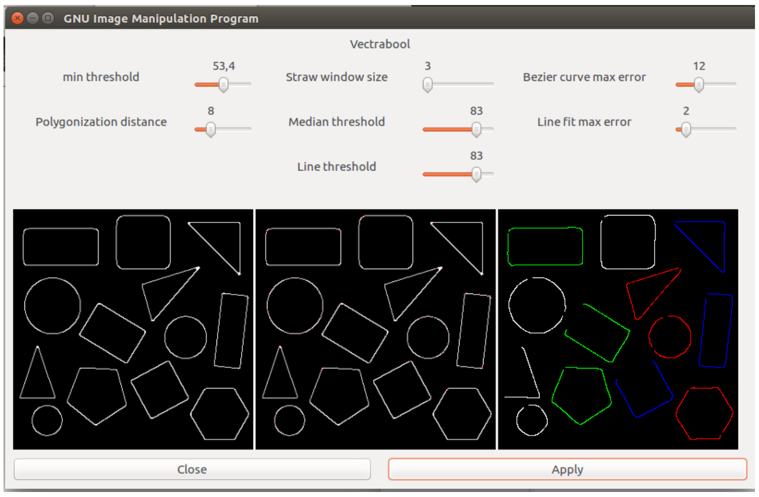

(b) The output image

Figure 8: The flower example

10 The objects example


GNU Image Manipulation Program

Vectrabool

min threshold 53,4

Polygonization distance 8

Straw window size 3

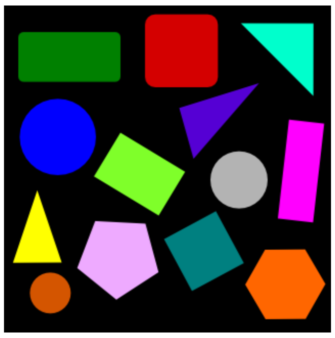
Median threshold 83

Line threshold 83

Bezier curve max error 12

Line fit max error 2

Figure 9: Parameters for objects example


(a) The input image

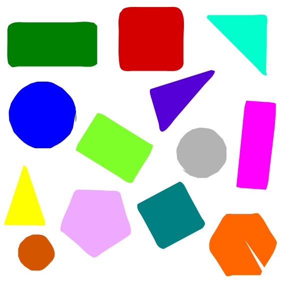
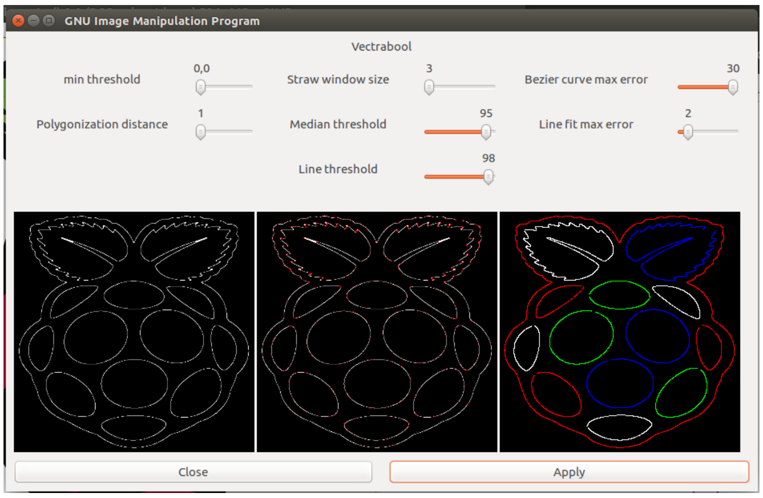

(b) The output image

Figure 10: The objects example

11 A very precise output example

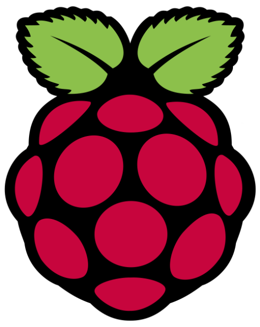
In this example, we make a comparison between the input image and output image when both are zoomed in.


GNU Image Manipulation Program

Vectrabool

min threshold 0,0
Polygonization distance 1
Straw window size 3
Median threshold 95
Line threshold 98
Bezier curve max error 30
Line fit max error 2

Figure 11: Parameters for the image


(a) The input image

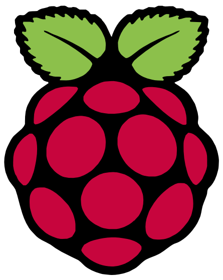
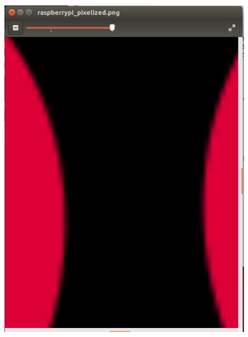
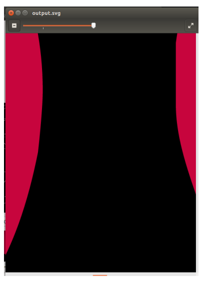

(b) The output image

Figure 12: A very precise example



(a) The input image when it is zoomed in

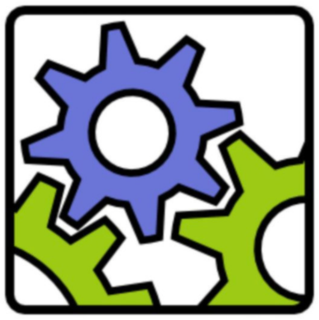


(b) The output image when it is zoomed it

Figure 13: A comparison of input and output image when both are zoomed n

12 Better than autotracer or not

This example shows that our library is not perfect. It also shows some bugs in the color detection algorithm. We encourage the reader to try to test this example with autotracer or vectorization.org and compare the results.


(a) The input image

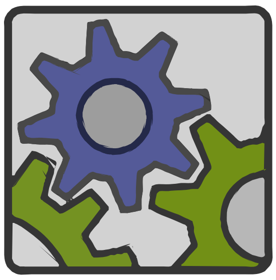

(b) The output image

Figure 14: An example where the color detection algorithm does not work properly

13 Blurring trick

If the input image has a very low resolution, one should blur it as much as possible (but keeping it understandable) in order to achieve the best results. In such cases, the color detection algorithm performs very poorly. But, compared to other mentioned softwares, only our library produces a reasonable result.

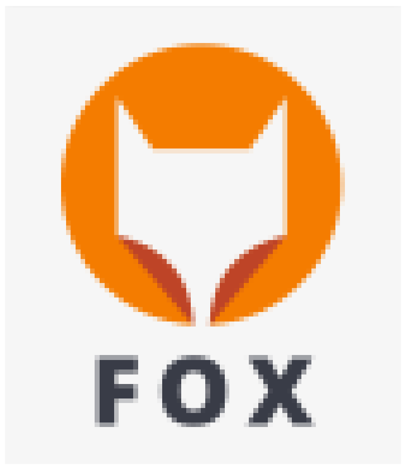
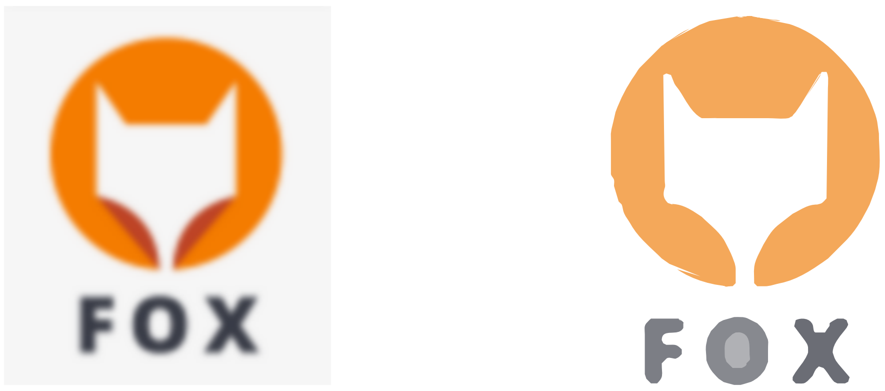


Figure 15: The input image



(a) The input image when blurred (b) The output image

Figure 16: An example where the color detection algorithm does not work properly

14 Further work

There are several things we plan to do in order to improve the quality of the library. We can divide all of them into to groups: technical improvements and algorithmic improvements. The most important technical improvements are removing opencv as a dependency while keeping all features the same, adapting the gimp plugin for inkscape software and developing a web application. We plan also to improve algorithms for corner detection and color detection. Although we have a good performance with the current implementations, we think that finding a better algorithms could improve the library in terms of the quality of output image as well as in terms of time complexity.

References

[1] Konstantinos G Derpanis. The harris corner detector. *York University*, 2004.

[2] Aaron Wolin, Brian Eoff, and Tracy Hammond. Shortstraw: A simple and effective corner finder for polylines. In *SBM*, pages 33–40, 2008.