CARLETON UNIVERSITY

HONOURS PROJECT

COMP4905

Evaluation of Vectorization Techniques for Freehand Raster Sketches

Author:
Patricia Joanne FOSTER

Supervisor:
Dr. Oliver VAN KAICK

April 21, 2017

Abstract

Vectorization involves turning a pixelated raster image into a series of smooth mathematical curves. It has applications in animation, industrial design, and illustration. Accurate vectorization faces many challenges, including noise, ambiguous junctions, and superfluous details; even commercial software leaves much to be desired in terms of accuracy. This project examines several state-of-the-art vectorization techniques and uses them to implement a simple vectorization program. It also assesses the strength & weaknesses of its algorithms and provides concrete suggestions on how the software could be improved in the future, both in terms of robustness and accuracy. While the resulting vectorized images lack in detail and accuracy, the software nonetheless manages to capture the overall outline of each image. In addition, several original improvements to the algorithms are described, including a solution to address filled areas, the implementation of a minimum speed factor, and the possibility of using corners to improve centreline extraction.

Acknowledgements

Most of the algorithms cited in this project are based on the work of Noris et al. (2013). The distinction between these algorithms and original work is noted in each section.

I would also like to thank the artists Julie Fiala, Victoria Levesque, Alexei Vidal, Sarah Koudelka, Claude Charron, and Chloe Couture for providing original sketches to be used as test inputs to the program.

Contents

List of Figures

List of Tables

Introduction

1.1 Motivation

The term "lineart" is used to denote a clean, finalized drawing that an artist has refined from their messy initial sketch. While necessary, it is a time-consuming and often tedious procedure. Furthermore, artists do not always have access to the necessary hardware required to create high-quality digital lineart (such as scanners, graphic tablets, or cintiqs). Automating the lineart process would be beneficial to many artists from diverse disciplines such as animation, illustration, and technical sketching, especially if the solution can be used anywhere and at any time. However, it is imperative that the vectorization software does not compromise the quality of the final drawing. Unfortunately, even state-of-the-art software leaves much to be desired in terms of precision and smoothness, and the majority of artists prefer to do their lineart manually.

1.2 Current Vectorization Methods

Currently, there are two main approaches to vectorizing freehand sketches. The first involves using traditional image processing techniques that evaluate each picture using mathematical tools. A recent example of this method is described in *"Topology-Driven Vectorization of Clean Line Drawings"* (Noris et al., 2013), which was first published in 2013. It involves pixel clustering, topology extraction, Gaussian smoothing and a novel "Reverse Drawing" procedure. Combined, they create a vectorized image from a raster sketch. Many variations of these methods exist, including least-squares line fitting, bezier curve fitting, Gabor & Kalman filters, etc.

The second approach leverages recent developments in the field of artificial intelligence, specifically Convolutional Neural Networks (CNNs). The paper *"Learning to Simplify: Fully Convolutional Networks for Rough Sketch Cleanup"* (Simo-Serra et al., 2016) explores this new technology and showcases results that are more accurate and aesthetically pleasing than state-of-the-art programs such as Adobe Live Trace and Potrace. However, as this method is neural network-based, its success relies entirely on the quality of the dataset. Furthermore, the program described required three weeks of training using a specialized GPU in order to achieve high-quality results. It is a promising approach, especially as it is a relatively new field. However, there unquestionably difficult resource constraints on its implementation.

1.3 Project Goals

The goal of this project was to implement a simple vectorization software and evaluate its strengths and weaknesses. Given the time constraints of the project, the traditional vectorization approach was selected, specifically the work of (Noris et al., 2013). This paper was chosen because it is recent, its algorithms are new, and its results improve upon the status-quo.

Specifically, the goal of this project are as follows:

1. Implement the described algorithms and create a simple vectorization software. Since the original paper had limited space, it did not cover implementation details. Therefore, different approaches to every step will be considered and analyzed.
2. Conduct an in-depth assessment of the strengths and weaknesses of the program
3. Suggest improvements

In particular, the authors note that their method is intended for ’clean’ raster sketches. Since there is no single traditional vectorization method that can be appropriate for all sketches, given that levels of cleanliness, roughness, and detail change which vectorization approach is the most appropriate for a particular sketch, it would be beneficial to know the exact limits of the proposed solution.

1.4 Structure of Report

The report is divided into seven sections. First, there is a run-through of the current methods used in traditional vectorization. Next, there is a summary of the original paper, *Topology-Driven Vectorization for Clean Line Drawings* (Noris et al., 2013). This section provides an overview of the vectorization algorithm as well as a few illustrative examples.

Aftterwards, a detailed look is taken at the three main components of the vecotorization algorithm: Pixel Clustering, Topology Extraction, and Smoothing / Curve Fitting. Each step is evaluated in its own section. Note that the Reverse-Drawing is not considered in this project. Each of these sections considers the implementation details of the algorithm and discusses the strengths and weaknesses of the resulting program.

Finally, the Evaluation considers the time and space requirements of the program as a whole.

The report ends with a conclusion.

Background Information

2.1 Vectorization

Vectorization of freehand sketches is a challenging problem, since a computer has no sense of the intention of the image. Many images have a mix of long lines (used to create outlines and convey the overall shape of the image) combined with small details. Paying too much attention to one typically has consequences for the accuracy of the other. Furthermore, certain details (such as ambiguous junctions and overly sharp corners) are easy to distort during vectorization.

From a high-level perspective, vectorization involves cleaning up an image (i.e. separating meaningful data from non-meaningful data) and then attempting to fit the extracted data into a series of simplified shapes, whether these are lines, arc, or piece-wise polynomial functions.

2.2 Filtering

Various methods exist to ’pre-process’ a raster image in order to make it easier to vectorize. The most important step is to remove ’noise’, which obfuscates important data and can skew the vectorization results. From an artistic perspective, noise is typically the product of smudges, partially erased lines, or accidental scribbles. Unfortunately, it is almost impossible for a computer to meaningfully differentiate between an accidental scribble and a small but purposeful detail using traditional methods (assuming the program has no bonus knowledge of the picture and the final goal).

Gaussian filtering is the most basic form of noise removal and is recommended as a first step for most image processing algorithms. For example, Gaussian Filtering is built-in to the opencv implementation of the Canny Edge Detector.

A more sophisticated filtering method is used by Sezgin and Davis (2004), involving geometric approximations that handle overtraced lines. Bartolo et al. (2007) takes a different approach and implements Gabor filtering to group and transform overtraced lines into simpler lines, which can then be vectorized. This approach exclusively addresses rough sketches, and is not necessarily appropriate for inputs where the exact shape of the lines need to be preserved.

Too much filtering can cause details to be lost, whereas insufficient filtering leads to noise in the end product. Therefore, the correct amount of filtering typically depends on the input image.

2.3 Curve Fitting

Curve fitting is frequently (but not always) a part of vectorization. Curve fitting involves finding a function to approximate a multidimensional - in this case, 2D - series of points. The earliest work with curve fitting was done using exclusively straight lines; an example of this is the Ramer-Douglas-Peucker algorithm, which is used in the opencv implementation of **approxpolydp**. Another method for creating straight lines is the Hough Transform, which was leveraged by Hilaire and Tombre (2006). Finally, combining straight lines with arcs is also popular, especially in programs used for industrial design.

More recently, **splines** and **Bezier Curves** have been used, such as in Chang and Yan (1998). These are examples of piecewise polynomial functions that can smoothly approximate complex shapes. The popularity of splines is the ease of their construction and the efficiency of the curve-fitting algorithm.

There are several difficulties with vectorization. The first lies in determining which point segments correspond to which curve (i.e. figuring how how to partition the point set into different curves). Next, there is a question of how precisely a curve should math the point set, and balancing between oversimplification and over complication.

Overview

This project was primarily based on the work of Noris et al. (2013). Their vectorization algorithm contained 4 distinct steps:

1. **Pixel Clustering**. Each pixel above a certain intensity threshold is treated as a 2D point in continuous space. Using the x and y image gradients as its direction vector, each point is then moved iteratively towards its 'centerline', i.e. the space where bands of opposing pixels meet. In other words, thick bands of pixels are thinned down to single lines.

   There are several advantages to this method. First of all, it homogenizes strokes widths, which makes it easier to extract structural information from the picture. Next, the pixel clustering method provides an estimate of the 'local stroke width' at each point. Finally, the method is highly accurate and helps to reduce noise.

2. **Topology Extraction.** To evaluate the topology of the picture, the clustered set of points in transformed into a graph. Each point becomes a vertex which is connected to every other vertex within the local stroke width. Next, the *Minimum Spanning Tree* of the graph is calculated. This MST is iteratively pruned until the underlying topology is extracted. At this stage, only vertices and edges that contribute directly to the picture's structure remain. This makes it possible to separate the picture into distinct curves, or 'base centrelines'.

   Centreline extraction is accomplished by location all of the graphs endpoints (vertices of degree 1) and junctions (vertices of degree 3 or higher) and using these as separation points for the curves. Note that if the graph is not properly pruned, it will have many spurious junctions and endpoints, which in turn creates unnecessary curves.

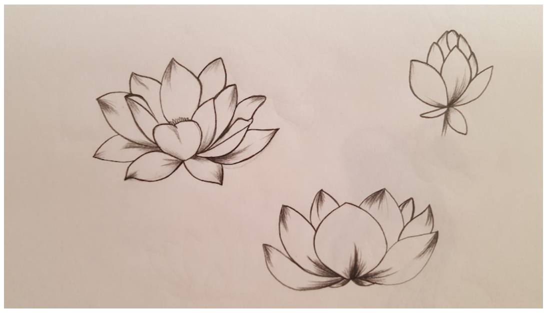
3. **Smoothing.** Although curves have been extracted, they are not always optimally smooth. This may be due to imperfections in the original image, or innaccuries introduced in the previous two steps. This phase of the algorithm iteratively applies a *Gaussian Smoothing Operator* to each curve in order to correct any errors and create the most mathematically optimal approximation of the original points.

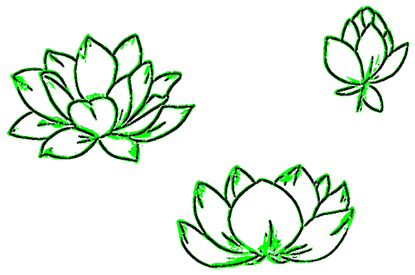
4. **Reverse Drawing.** One of the most challenging aspects of vectorization is dealing with what 2013 refer to as 'ambiguous junction regions'. Junctions - where different lines meet - often contain a lot of subtle detail that is easily lost during vectorization. The Reverse Drawing algorithm identifies the ambiguous region, deletes it, and then tries fitting a series of pre-defined junctions to the original images. The junction that fits the data the best is kept.

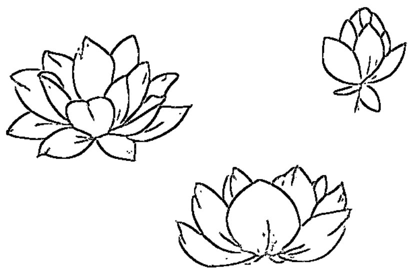
While this project did not implement *Reverse Drawing*, it did have a fourth step that was not included in Noris et al. (2013)'s original paper: Curve Fitting. Smoothing involves making minor adjustments to points in a curve in order to obtain a more flowing line. Curve fitting, on the other other, attempts to derive a mathematical equation to fit this curve. Therefore, fitted curves are the most condensed representation of an image.

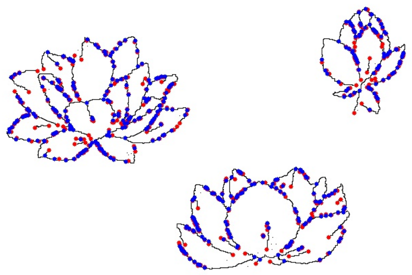
The vectorization program is implemented in Python, using the OpenCV library as well as python packages such as NumPy, SciPy, and matplotlib. Python was chosen because of its flexibility and simplicity, as well as its compatibility with OpenCV, which contains numerous image processing & computer vision algorithms whose implementations are efficient and robust. MATLAB was used during the curve fitting step because of its extensive and efficient curve fitting library.

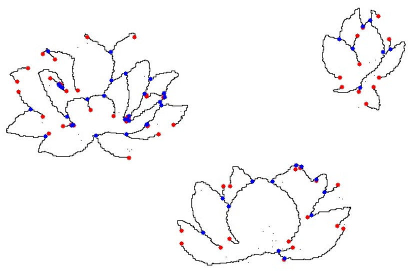
Results

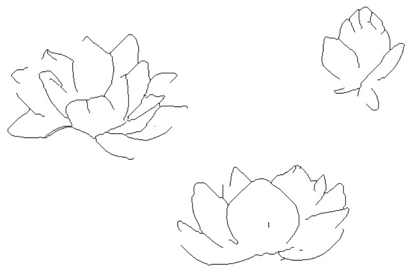

(a) Original Image by Alexei Vidal.

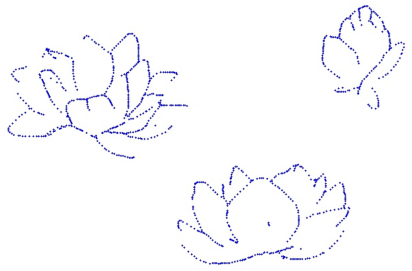

(b) Third round of pixel clustering algorithm.

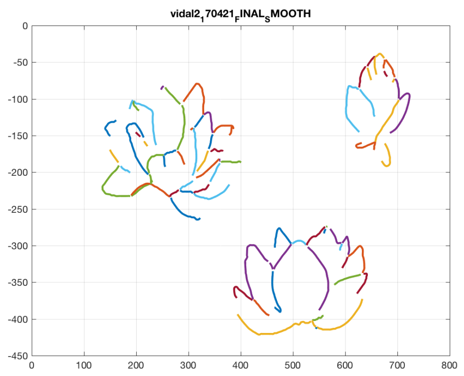

(c) End of pixel clustering algorithm.


(d) First round of iterative pruning during topology extraction. Blue points mark the junctions; red points are endpoints.


(e) Last round of iterative pruning.


(f) Image after base centreline (curve) extraction.

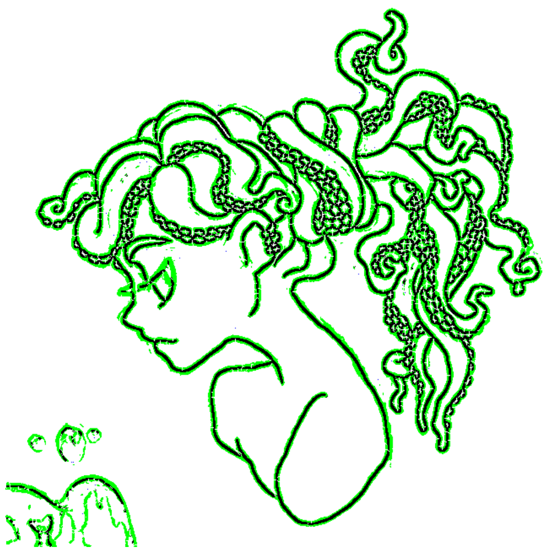

(g) Image after 200 rounds of smoothing.

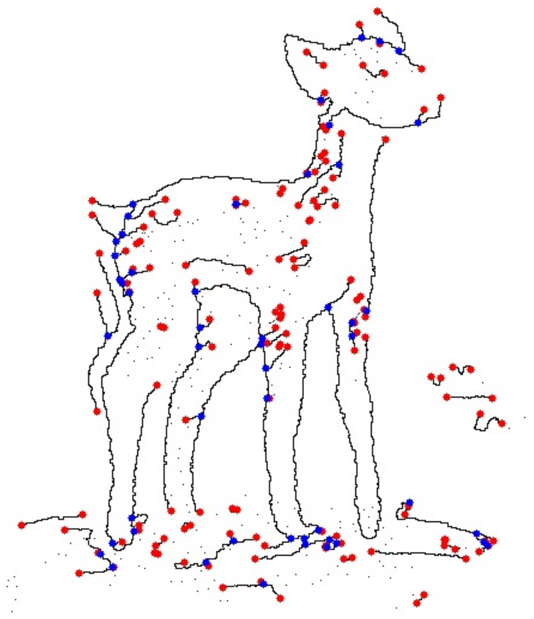


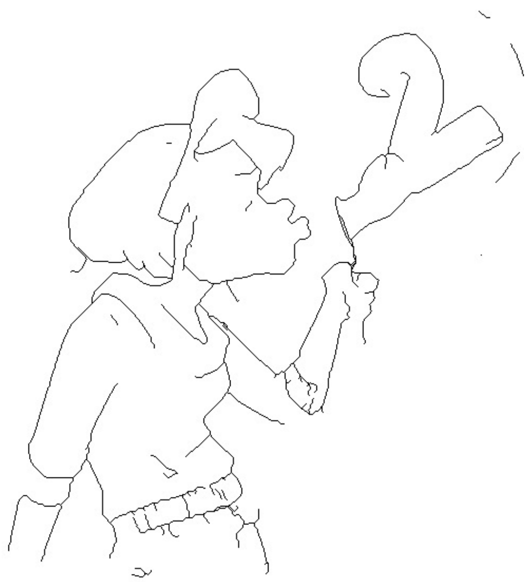
(h) Smoothed image vectorized using MATLAB natural spline. (Note that image scale has been skewed).

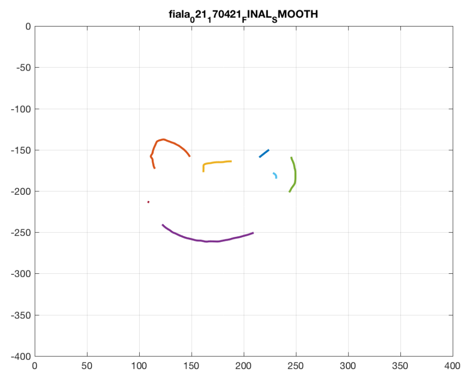
Figure 3.1: **Lotus.** Step by step overview of vectorization process.

While the software is relatively efficient, and effectively captures the overall shape of the image, there are still numerous shortcomings in terms of smoothness and accuracy. Some of these are gaps are due to missing pieces in the algorithm - such as the calculation of local minimum spanning trees, or the correction of loops. Some shortcomings are implementation quirks, while others are the result of limitations in the original algorithm by Noris et al. (2013). This project will explore all three in detail, and suggest possible ways of overcoming them.


(a) **Octopus Hair** by Sarah Koudelka. 2 rounds of pixel clustering.


(b) **Fawn** by Victoria Levesque. 7 rounds of pruning.


(c) **2014** by Chloe Couture. End of centreline extraction.



(d) **Kitty** by Julie Fiala. Vectorized using a natural spline.

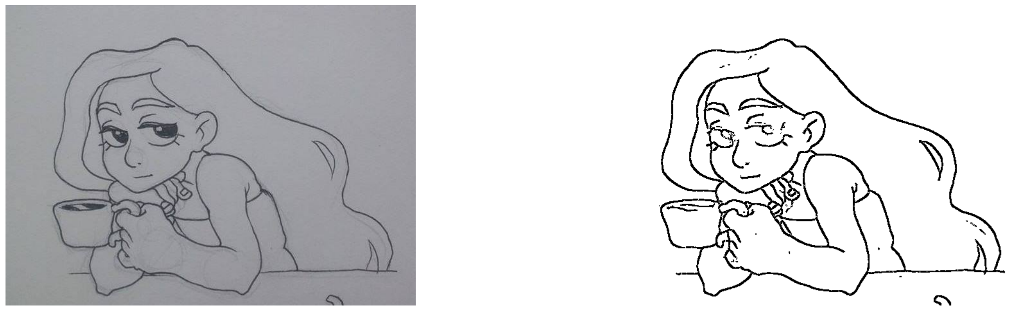
Figure 3.2: Example outputs at various stages.

Pixel Clustering

4.1 Pre-Processing

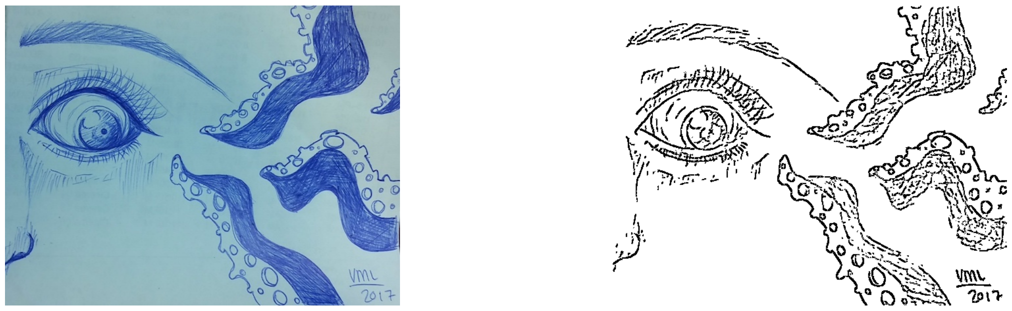
In order to make the pixel clustering more effective, a pre-processing step is executed that converts the image to greyscale. In addition, a Gaussian Blur filter is added to reduce noise.

While not currently implemented, a useful improvement would involve a special step to deal with 'filled' spaces. This is the most obvious weakness of the original vectorization of Noris et al. (2013). In all fairness, the algorithm was never intended to be used by such sketches. Currently, the algorithm's treatment of filled spaces is somewhat random. Sometimes, they are completely subsumed by other lines (Figure 9.7). Other times, they are converted into random squiggles that were clearly not the original intention of the artist. (Figure 4.2).



(a) Original Image by Sarah Koudelka. (b) After pixel clustering.

Figure 4.1: **Coffee**. Comparison before and after Pixel Clustering.



(a) Original Image by Victoria Levesque. (b) After pixel clustering.

Figure 4.2: **Eye**. Comparison before and after Pixel Clustering.

One possibility for correcting this is a step that detects 'filled' areas and isolates them from the rest of the image. Next, using an edge finder, such as the Canny Edge Detector, the contours could be determined and then reconnected to the original image. Consequently, the original picture would have uniform edges suitable for vectorization.

4.2 Clusering Algorithm

As previously mentioned, pixel clustering is somewhat analogous to 'line thinning'. The algorithm described by Noris et al. (2013) reduces lines to their thinnest, most basic shape, thereby simplifying the sketch and allowing for effective topological extraction.

ALGORITHM 1: PIXEL CLUSTERING

1. **The image gradient is calculated**. Where ∇_i is the gradient at pixel p_i

2. **Each pixel is placed into one of two categories: moving pixels (M) or stationary pixels (S).** The classes are defined as follows:

$$M = \{p_i || |\nabla_i|| < \epsilon\} \tag{4.1}$$

$$S = \{p_i || |\nabla_i|| \geq \epsilon\} \tag{4.2}$$

   Where ϵ is 10% the width of the gradient. However, Noris et al. (2013) note that ϵ should be large enough to remove all noise from the picture. For simplicity's sake, each pixel is assumed to have a height and a width of 1. Using this sizing method, the image's pixels are mapped to a set of moving points:

   Code Snippet:

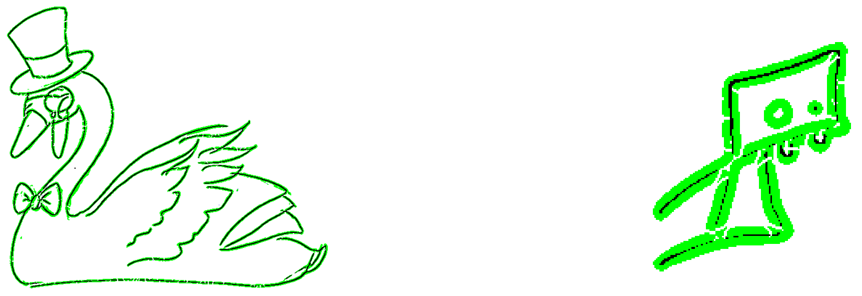
```
pointset = []
# STEP 1: MEASURE GRADIENT MAGNITUDES + GET THRESHOLD
max = grad_mag.max()
thresh = max / 10

# STEP 2: ITERATE THROUGH PIXELS
for i in range(img.shape[0]):
    for j in range(img.shape[1]):

        # THRESHOLD TEST
        if grad_mag.item(i,j) > thresh:
            # ADD POINT TO POINTSET
            p = util.Pixel(i, j, grady.item(i,j), gradx.item(i,j))
            pointset.append(p)
```

3. **The position of each moving pixel is iteratively updated until all pixels cluster around the centerline.** To accomplish this, the pixels move following their gradient direction, and move with a speed proportional to their gradient magnitude. A motion vector (m_i) is defined as follows:

$$m_i = \delta t \nabla_i \tag{4.3}$$

   Where δt is a constant speed factor.

   The following figures are images after a single round of pixel clustering. The black pixels have been stopped. The green pixels are still moving.



(a) **Swan**. (b) **Dinosaur**.

Figure 4.3: Images after a single round of pixel clustering.

4. **The loop stops when 99% of the pixels have finished moving.** An individual pixel can be 'stopped' under two conditions.

    (a) The pixel has less than two neighbours. This indicates that this particular point is likely noise and does not contribute to the overall topological structure of the image.

    (b) The pixel has moved passed another pixel in the 'opposing band'. A band is any connected group of pixels moving in the same direction; the opposing band moves in the reverse direction. Centerlines occur where opposing bands meet. The equation for the stopping condition is given as follows:

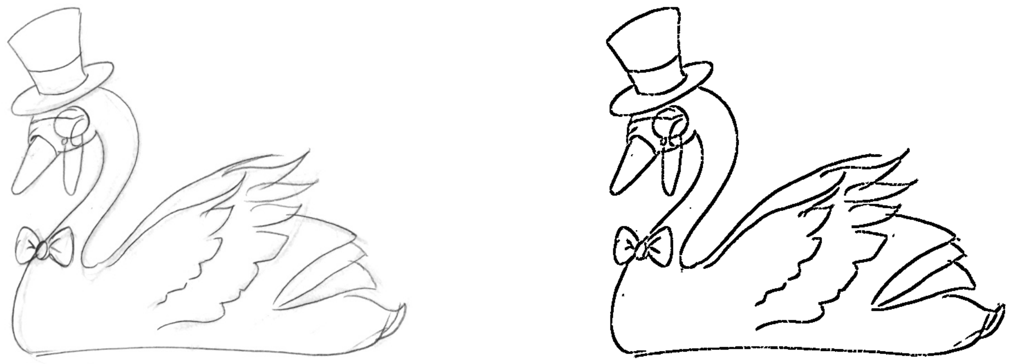
    For each pixel $p_j \in N_i, p_j \neq p_i$:
    $$(p_i - p_j) \cdot \nabla_i < 0 \tag{4.4}$$

5. **Calculate Local Stroke Thickness**.Once the clustering algorithm is complete, it is possible to calculate the *local stroke thickness* at each pixel. This property is used during topology extraction and smoothing. Intuitively, the local stroke thickness should reflect the distance travelled by each pixel (defined as r_i). This property is stored in the Pixel object. Ultimately, the local stroke thickness (LST) is calculated by taking a conservative estimate as follows:

    $$LST = max\{r_j | r_j \in N_i\} \tag{4.5}$$

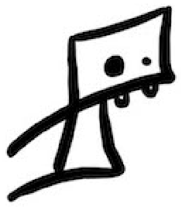
    Where N_i is the local neighbourhood of pixel p_i.

The following are the two sample images before and after the pixel clustering process. Notice how the swan's lines remain the same thickness (if anything, they appear thicker) while the dinosaur's lines have shrunk considerably. The lines, which were very different at the outset, are now approximately the same thickness.



(a) Original Image by Patricia Foster. (b) Image after pixel clustering algorithm.

Figure 4.4: **Swan**. Comparison of image before and after the pixel clustering process.


(a) Original Image by Julie Fiala.

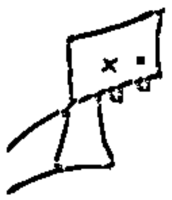

(b) Image after pixel clustering algorithm.

Figure 4.5: **Dinosaur**. Comparison of image before and after the pixel clustering process.

4.3 Implementation Details

Pixel Class

In order to keep track of all the information about each point, a 'Pixel' class was created with the following attributes: current x and y coordinates (floats), x gradient and y gradient values (floats), the initial x and y coordinate (ints), the local stroke width (float), and a boolean value indicating whether or not the pixel has 'stopped'.

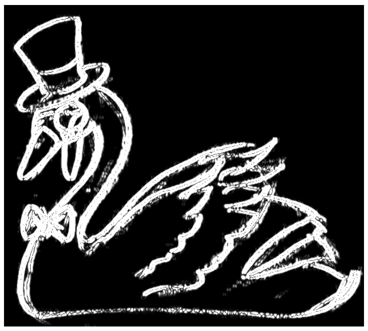
Gradient

Although Noris et al. (2013) do not mention which gradient they use (Sobel, Laplacian), their algorithm requires both magnitude and direction. Therefore, in this implementation, the Sobel X and Y gradients were chosen for simplicity and efficiency. Essentially, the Sobel Operator is a discrete approximation of an image's intensity gradient. It is calculated by convolving the original image with two $N \times N$ kernels; one for the horizontal derivative (G_x), and one for the vertical derivative (G_y):

$$G_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix}$$

Note that the size of G_x and G_y can vary; a larger kernel lends itself to more accuracy. In the following example, a 5x5 kernel was used:

(a) Original Image by Patricia Foster.


(b) Sobel X gradient.

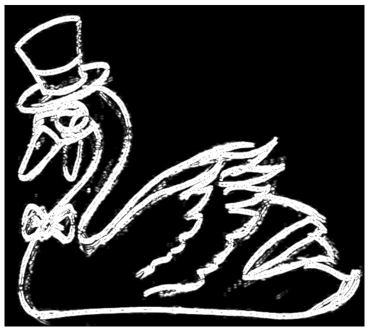

(c) Sobel Y gradient.

Figure 4.6: **Swan.** Original Image and Sobel gradients.

Code Snippet:

```
blur = cv2.GaussianBlur(img, (7,7), 1)
sobelx = cv2.Sobel(blur, cv2.CV_64F, 1, 0, ksize=5)
```

```
sobely = cv2.Sobel(blur, cv2.CV_64F, 0, 1, ksize=5)
grad_mag = np.sqrt(np.square(sobelx) + np.square(sobely))
```

Finding Neighbours

In the movement phase (steps 3&4) of the Pixel Clustering Algorithm, where pixels are slowly being drawn towards their centrelines, locating a pixel's neighbours is key. Unfortunately, this is a tricky task, because the moving points are stored in a list. Furthermore, the coordinates of each point shift slightly at every iteration, meaning that a point's neighbours must be re-calculated at every iteration.

There are two possible approaches to finding neighbours. The first involves a **k-d tree**, and the second involves a **hashed grid**.

The advantage of using k-d trees is the simplicity of the method. k-d trees are data structures used to partition k-dimensional points, allowing for efficient nearest-neighbour searches in this k-dimensional space. The k-d tree is constructed by splitting points into sorted bings along the x plane. Inside each of these bins, it then splits the points into sorted bins along the y planes. This process continues until the bins are a satisfactory size; at each level, the k-d tree alternates between dimensions.

However, as all the pixels are constantly moving, the k-d tree must be created from scratch at any iteration. This leads to considerable slowdown in the process, as a k-d tree is constructed in $O(n \log n)$ time but searched in $O(\log n)$. However, immediately updating the node of each k-tree (but removing a pixel and replacing it with its updated version) would skew the clustering process towards the pixels that were processed first.

For implementation purposes, the SciPy 'spatial' package was used to create the k-d tree. The neighbourhood radius is set at $\sqrt{2}$; this corresponds to the distance between the 'centers' of two different pixels.

Code Snippet: Implementation of the movement phase (steps 3&4) using a k-d tree:

```
movement_phase = True
counter = 0
while movement_phase:
       counter += 1

        # movement phase
        for i in xrange(len(moving_points)):

        x_speed = getMinSpeed(SPEED_FACTOR * moving_points[i].gradx)
        y_speed = getMinSpeed(SPEED_FACTOR * moving_points[i].grady)

        if moving_points[i].stopped: continue
          moving_points[i].x = moving_points[i].x - x_speed
          moving_points[i].y = moving_points[i].y - y_speed

        # make the kd_tree
        coords = util.Pixel.pixel2coords(moving_points)
        kdtree = spatial.KDTree(coords) # 'spatial' is a scipy package

        # evaluation phase
        for p in moving_points:
            stop = stop_pixel(kdtree, moving_points, p)
            if stop:
                p.stopped = stop
```

The second solution - **the hashed grid** - involved blending a discrete structure (where it is easy to find neighbours) with a continuous structure (where it is possible to make small incremental changes to the positions of individual points).

While converting the matrix to a list of 2D points, another data structure (a Hashmap, known as a Dictionary in Python) was constructed at the same time. The rules are as follows:

1. **Each pixel in the original image is treated as a bin.** An entry in the dictionary is created for each pixel, and a tuple containing the row, column indices is used as a key. (In Python, tuples are immutable, and thus hashable.) The associated value is an empty list.
2. **Points are sorted into appropriate bins.** The x and y coordinates are rounded up to find the appropriate indices. Then, the point is appended to the list of the appropriate dictionary key.
3. **To find neighbours, simply create a master list of all points in the bins adjacent to the current point.** Bins can be checked in constant ($O(1)$) time. Points that are not within $\sqrt{2}$ are rejected.
4. **The location of each point must be updated immediately after that pixel's movement phase.** If the pixel's rounded x and y coordinates no longer fall within the same bin, the pixel is removed from the previous bin and added to the new one. This is also accomplished in constant time. However, immediately updating pixels has the potential of skewing results slightly in favour of pixels that are processed first. In practice, this was not noticeable, and may not be a problem.

The trickiest part of the implementation is checking for image's edges. Any pixel that moved into a non-defined was immediately removed from the set.

The images obtained through this method were practically identical, and the algorithm runtime's was over 10 times faster. Thus, grid method was chosen in favour of k-d trees.

Code Snippet (finding neighbours):

```
def get_neighbours(point_map, p, shape):

    i = util.R(p.x)
    j = util.R(p.y)

    if i < 0 or j < 0 or i >= shape[0] or j >= shape[1]: return []

    neighbours = []
    neighbours.extend(point_map[(i, j)])
    if j != shape[1]-1: neighbours.extend(point_map[(i, j+1)])
    if j != 0: neighbours.extend(point_map[(i, j-1)])

    if i != shape[0]-1: neighbours.extend(point_map[(i + 1, j)])
    if i != shape[0]-1 and j != shape[1]-1: neighbours.extend(point_map[(i + 1, j + 1)])
    if i != shape[0]-1 and j != 0: neighbours.extend(point_map[(i + 1, j - 1)])

    if i != 0: neighbours.extend(point_map[(i - 1, j)])
    if i != 0 and j != shape[1]-1: neighbours.extend(point_map[(i - 1, j + 1)])
    if i != 0 and j != 0: neighbours.extend(point_map[(i - 1, j - 1)])

    return neighbours
```

Using the Hashmap also allows for another possible optimization, although it was not implemented in this project. The idea would be to create a *scarce point set* that would potentially speed up later steps in the algorithm.

From a visual perspective, having one versus multiple points in the same bin will not change the appearance of the final image, since the pixel is the smallest visual unit in an image. Thus, once the pixel clustering phase was complete, 30% or 50% of the pixels in a given bin (always rounded up) could be selected at random to be removed. This may be useful for very large images with very large graphs. On the tested inputs, however, the runtime was not large enough for this optimization to be necessary.

Minimum Speed Factor

Noris et al. (2013) define δ_t as 10% the width of the pixel. However, in experimental trials, the maximal gradient length frequently had values of 7,000 or even 9,000. Therefore, a pixel's motion vector could have values of 700 or 900, which would be a ridiculous increment. Most likely, this indicates a difference in either the type of gradient used or the properties of the image being processed. In order to compensate, δt was set to 0.0001. The intuition is that no single point should move more than 1 pixel during a single iteration. This avoids points moving too quickly and ending up farther than the desired centerline.

Unfortunately, since the speed factor (δt) is so much smaller than expected, the outermost pixels with smaller gradients now move at a crawl. The smallest gradient is roughly 10% the length of the maximal gradient; this means that many, many iterations are required before the pixels converge.

In order to counter this, a *minimum speed factor* was implemented. Essentially, the movement vector on any single pixel is restricted from being any smaller than this value. To determine the impact of the Minimum Speed Factor (and thus its optimal value), values between 0.2 and 1 (in steps of 0.1) were experimentally tested and the algorithm's runtime was recorded for each one.

The following graphs demonstrate the runtime (in sections) as a function of the Minimum Speed Factor (MSF). As shown, the runtime decreases in an exponential manner as the Minimum Speed Factor is increased.

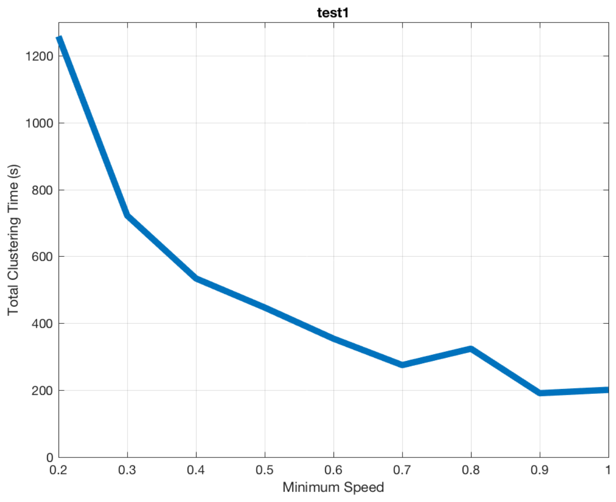


Figure 4.7: Graph of time vs. MSF for **Swan**.

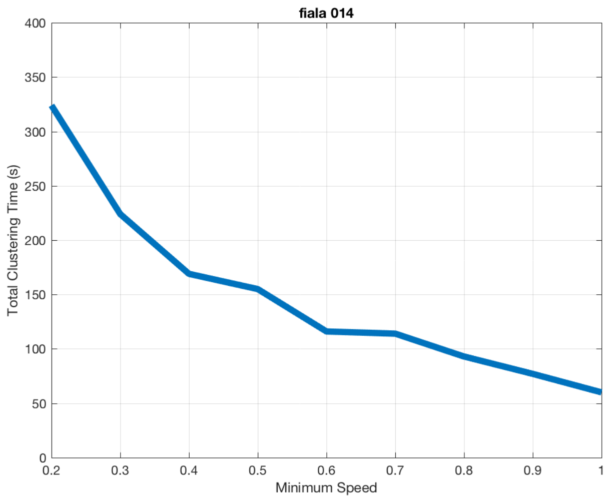
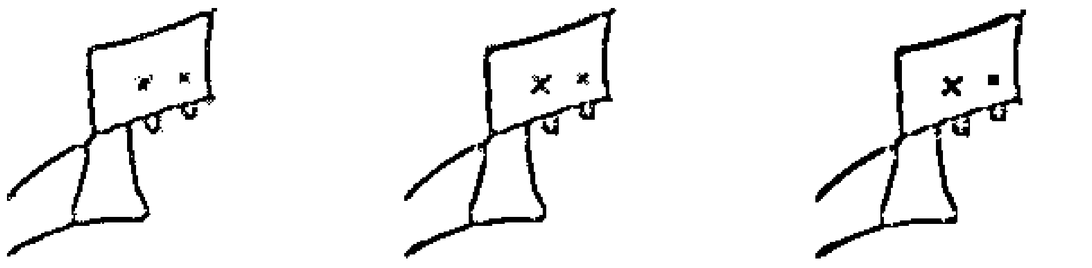


Figure 4.8: Graph of time vs. MSF for **Dinosaur**.

The number of moving pixels, on the other hand, does not vary in any clear, logical way as a function of the Minimum Speed Factor, and remains relatively stable. Overall, there is minimal impact on accuracy, which is not to say that the resulting images are identical.



(a) Minimum Speed Factor: 0.2 (b) Minimum Speed Factor: 0.7 (c) Minimum Speed Factor: 1.0

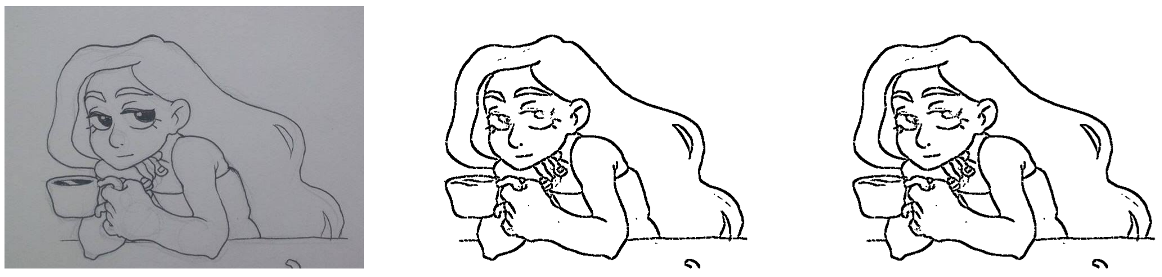
Figure 4.9: **Dinosaur**. Image after pixel clustering with different Minimum Speed Factors.

As the Minimum Speed Factor increases, the lines in the image become bolder (as the resulting pixels are more spread out) and sharper, as there is less time for gentle arcs to form. Overall, this is a negligible difference, except in the case of small, tight curves, such as the dinosaur's eyes, which are turned into Xs or stars. Another benefit of higher minimum speed factors is a reduction is noise.

As a compromise between speed and accuracy, the default Minimum Speed Factor was chosen as 0.5. However, for pictures with a lot of moving pixels (over 15,000) it is increased to 0.7, and for pictures with over 50,000 pixels it is set at 1.0.

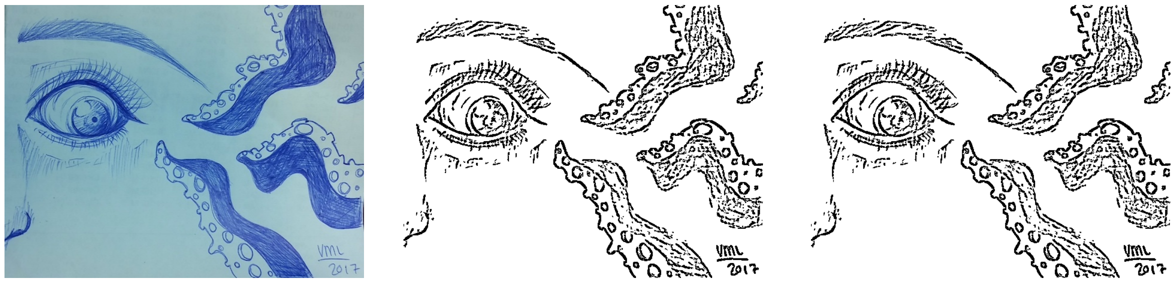
Pixel Stopping Condition

Another modification from the original algorithm was lowering the pixel stopping condition from 99% to 95%. The difference in quality between the two figures was negligible, and the 4% decrease let to an almost 50% increase in clustering speed.



(a) Original Image. (b) Pixel Clustering at 95%. (c) Pixel Clustering at 99%

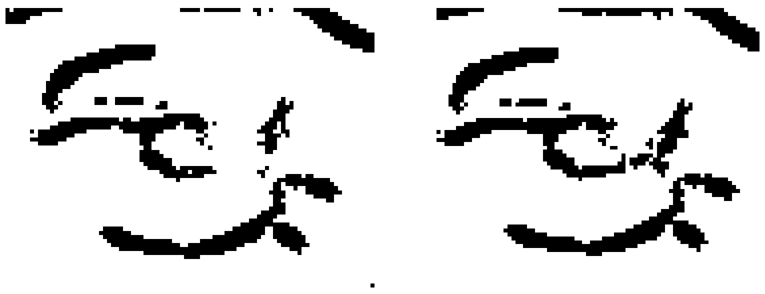
Figure 4.10: **Coffee**. Results of varying the stopping percentage. Image by Sarah Koudelka.



(a) Original Image. (b) Pixel Clustering at 95%. (c) Pixel Clustering at 99%

Figure 4.11: **Eye**. Results of varying the stopping percentage. Image by Victoria Levesque.

From far away, the images appear almost identical. However, at a closeup, a small improvement is evident in the image clustered at 99%. The only places that genuinely benefit, however, from this extra precision, are sections of the image that were already 'filled'. The problem is that the clustering algorithm by Noris et al. (2013) were not intended to deal with 'filled' sketches or 'messy' sketches. Essentially, the behaviour of the pixel clustering algorithm when faced with these spaces is almost random. In the case of 'Coffee', the filled section becomes a gap. In 'Eye', the filled section of the tentacles are interpreted into a series of meandering squiggles.



(a) Pixel Clustering at 95%. (b) Pixel Clustering at 99%

Figure 4.12: Closeup of the righthand eye in **Coffee**.

On the other hand, there is a drastic difference in the runtime of the algorithm depending on the percentage. Typically, the 95% takes 0.5 ± 0.1 less time.

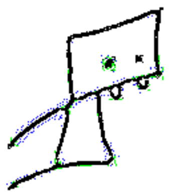
Image Name	Speed for 95% (s)	Speed for 95% (s)	Difference (%)
Coffee	447	897	0.498
Eye	884	2266	0.391
Dinosaur	130	197	0.660
Autumn Vignette	929	1873	0.496
Wave	468	917	0.510

Table 4.1: Speed of Pixel Clustering Algorithm at 95% and 99%

Cleaning Phase

Once the clustering process is done, any points that are still 'moving' and any points that were stopped under the first condition (not having neighbours) are removed from the pointset. Experimental trials showed that these pixels typically contribute to noise and not to the overall topology of the picture.

(a) **Swan**. Image by Patricia Foster


(b) **Dinosaur**. Image by Julie Fiala

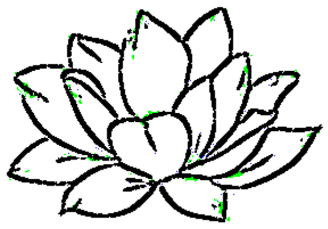

(c) **Lotus**. Image by Alexei Vidal

Figure 4.13: Images during the last round of pixel clustering.

In Figure 4.13, blue dots are pixels that have been stopped because they have less than two neighbours. Green dots are pixels that are still 'moving'.

Topology Extraction

5.1 Creating the Graph

In order to extract the topology of each image, Noris et al. (2013) transform the point set into a graph and employ the well-known *Minimum Spanning Tree* algorithm. However, first, the *Complete Cluster Graph* is constructed.

Essentially, each pixel becomes a vertex. Edges are added between any two pixels in the same neighbourhood. The local stroke thickness of each pixel is used to determine the radius of said local neighbourhood.

The hashed grid method is employed to locate each pixel's nearest neighbours. Meanwhile, the **NetworkX** (NX) Python package is used to model the cluster graph. The NX implementation of *Kruskal's Algorithm* is then applied to create the minimum spanning tree.

5.2 Iterative Pruning

The goal of this step is to remove all the unnecessary nodes and edges that do not contribute to the stroke's core shape. In other words, to 'prune' all of the twigs jutting off of the main branches.

ALGORITHM 2: ITERATIVE PRUNING

---

1. **All the leaf nodes in the tree are identified**. A leaf node is any node with a degree of 1.
2. **Starting at a leaf node, the entire branch is pruned until a stopping condition is reached.** Once a leaf node is removed, the node it was connected to becomes a leaf, and is pruned in its turn. The stopping conditions are as follows:
   (a) The length of the branch removed exceed the *maximum pruning radius*. Typically, this is the local stroke radius. For more accurate (if slower results), $\sqrt{2}$ can be used. This step is used to prevent real endpoints from being pruned.
   (b) The main branch is reached. This occurs when the node reached has a degree ≥ 2.

   One small improvement to Noris et al. (2013)'s original algorithm involves tweaking the pruning radius. Since there is a lot of variation within local stroke width, some lines are pruned very quickly (and thus greatly shortened) while other lines are pruned so slowly that the final product contains clumps of endpoints & junctions that do not reflect the topological structure. However, using $\sqrt{2}$ for all pixels led to longer runtimes and ultimately more gaps. A compromise turned out to be the bes solution: the pruning radius is set to the maximum value between the local stroke width and $\sqrt{2}$. This led to the highest quality results.
3. **Repeat steps 1-2 until all unnecessary branches are removed.** Unfortunately, it is impossible to tell with absolute certainty when the pruning process is done. In addition, the pruning process is slightly destructive. If it is repeated too many times, strokes are shortened, and some lines can become disconnected or even completely destroyed.

In order to determine when the loop should stop, the number of endpoints (nodes of degree 1) and junctions (degree ≥ 3) are counted. In the first few rounds of pruning, this number is incredibly high; close to the total number of moving points, in fact. The pruning continues until the number of junctions & endpoints is over 300 times smaller than the total number of moving points.

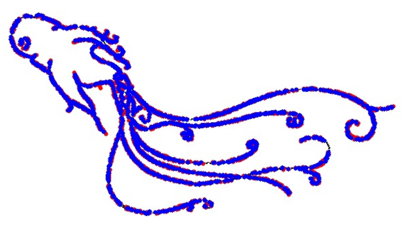
The magic number 300 is the **scale factor**. It was determined experimentally, by running the pruning algorithm on many different inputs. It is the optimal compromise between minimizing the number of junctions & endpoints and preserving the original topology of the picture.

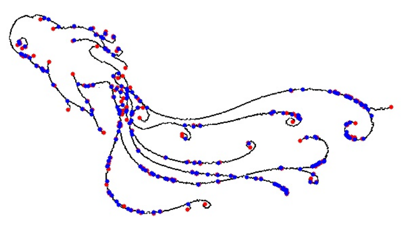
Code Snippet (Iterative Pruning Algorithm)

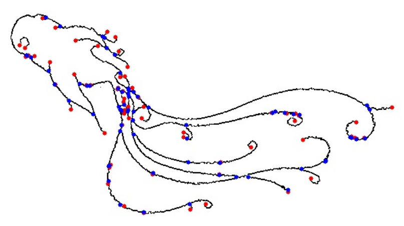
```
counter = 0
endpoints = []
junctions = []
itr_pruning_start = time.time()
while num_spc_pts >= len(moving_points) // SCALE_FACTOR:

    counter += 1
    leaf_nodes = [x for x in mst.nodes_iter() if mst.degree(x) == 1]

    for leaf in leaf_nodes:
        total_weight = 0
        curr_node = leaf
        while total_weight < PRUNING_RADIUS:
            n = mst.neighbors(curr_node)
            if len(n) != 1: break # used to be == 0
            e = mst.get_edge_data(curr_node, n[0])
            if total_weight + e['weight'] < math.sqrt(2):
                total_weight += e['weight']
                mst.remove_node(curr_node)
                curr_node = n[0]
            else:
                break

    endpoints = [x for x in mst.nodes_iter() if mst.degree(x) == 1]
    junctions = [x for x in mst.nodes_iter() if mst.degree(x) >= 3]

    num_spc_pts = len(endpoints) + len(junctions)
```

However, pictures with a lot of details and connecting edges will have a higher ratio of fractions. Consequently, the ratio may never drop beneath 1/300, and the program will be stuck in an infinite loop. In order to avoid this, the number of junctions + endpoints from the previous iteration is recorded. If no additional points were pruned in between two iterations, then the loop automatically exits.

The following sample image (*Koi Fish*) was provided by Alexei Vidal. In order to facilitate visualization of the pruning process, each endpoint was marked with a red circle, and each junction was marked with a blue circle.


(a) 1 round of pruning. captionsetupwidth=0.8


(b) 3 rounds of pruning.


(c) 5 rounds of pruning.

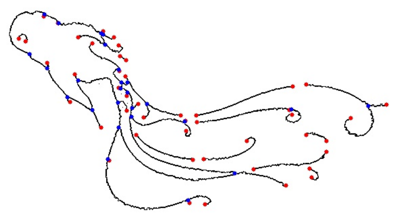

(d) Final image. (13 rounds of pruning).

Figure 5.1: **Koi Fish.** Various iterations of the Pruning Phase.

At the end, gaps have appeared in the tail of the Koi Fish. This is a byproduct of the minimum spanning-tree algorithm, which will create a gap for every loop in the picture.

In other cases - especially images with lots of detail - the process is quite destructive:

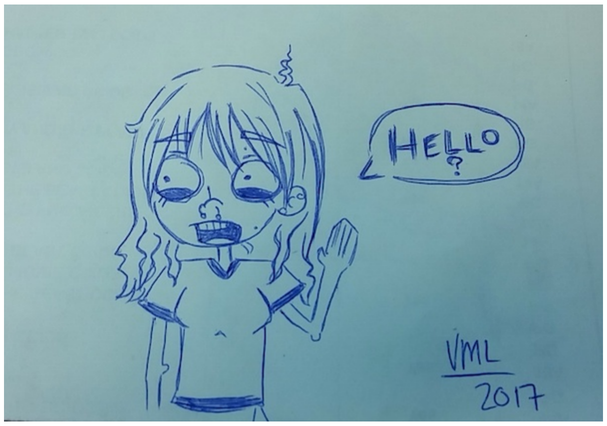


Figure 5.2: **Wave**. Original sketch by Victoria Levesque.

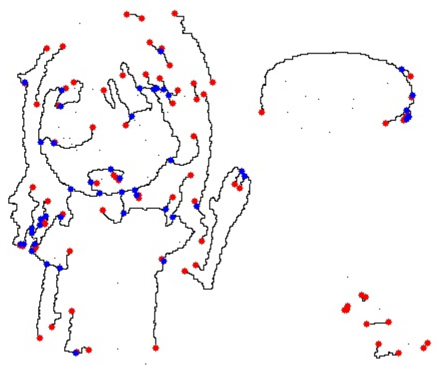


Figure 5.3: **Wave**. Picture at the end of pruning algorithm.

To correct this, Noris et al. (2013) calculate Local Minimum Spanning Trees and then merge the two graphs. Their algorithm looks something like this:

ALGORITHM 3: LOCAL MINIMUM SPANNING TREES

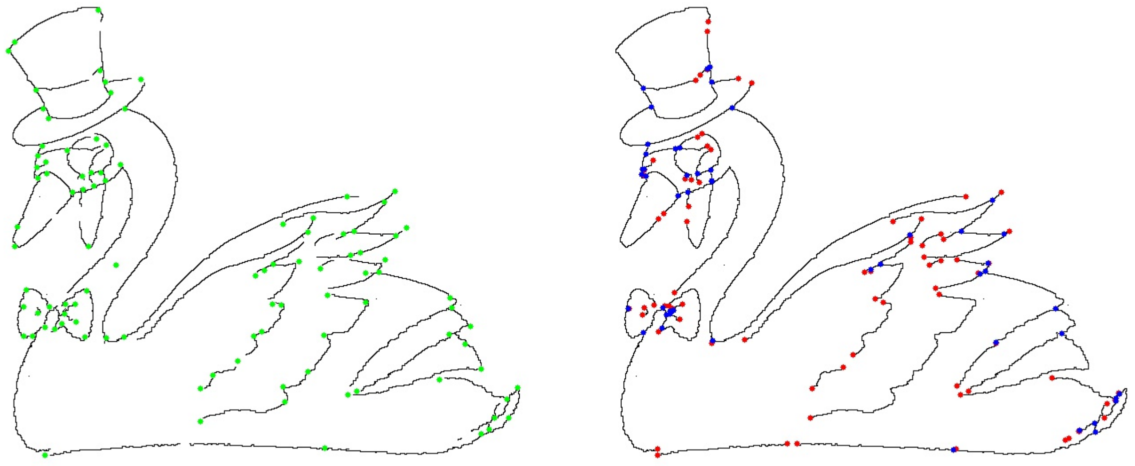
1. **For each endpoint, find all points and edges within the "local area", using the full cluster graph.** Noris et al. (2013) uses a 'conservative' radius of $4r_i$, where r_i is the local stroke thickness at endpoint p_i. For this particular implementation, the constant should be equal to 2 times the number of rounds conducted in the pruning phase (+1), since that is the maximal length that could have pruned from.
2. **Calculate a local minimum spanning tree.**
3. **Merge global and local minimum spanning trees.** This involves detecting which nodes are the same, adding nodes that don't exist, and filling in the relevant edges.

Note that the intuition of the algorithm is explained in the original paper, as well as certain details (such as the $4r_i$ radius), but no implementation is given. This step of the vectorization process was not implemented in this project.

Possible Improvement

Currently, endpoints and junctions are used to separate groups of pixels into distinct curves. However, given the current limitations of the project's pruning algorithm, sometimes important separation points aren't including, and sometimes unnecessary junctions are added when the curve in question would be more effectively treated as a single unit.

Another possible candidate for 'separation points' are 'corners', sometimes known as 'interest points'. Edges occur when there is an abrupt variation in the image gradient in a single direction. Corners occur when there is an abrupt variation in image intensity in *both* directions. The simplest algorithm is the Harris Corner Detector, which uses convolutions to approximate an image gradient. A more recent method is the Shi-Tomasi Corner Detector, which uses a slightly different scoring function.

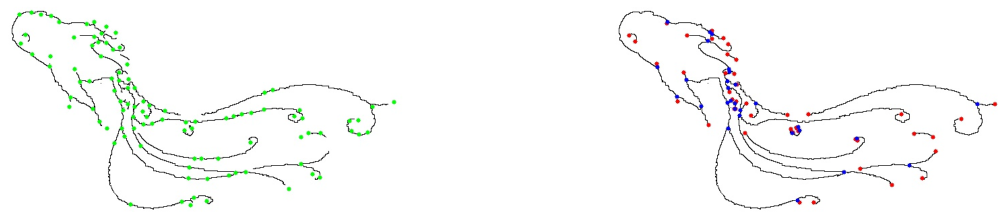

(a) Image after MST pruning, overlayed with detected corners.

(b) Image after MST pruning, overlayed with endpoints and junctions

Figure 5.4: **Swan.** Comparison of interest points.

In several areas, the corners are more accurate (and thus more desirable) than the junctions and endpoints obtained by pruning. Specifically, the top hat, the wing, and the bowtie are more accurately portrayed. This are all areas with more complex, subtle curves. In the case of the top hat and the wing, important points are missing or skewed when only the MST is used. In the case of the bowtie, the location of the corners better reflects the topology of the image (whereas the endpoints / junctions are somewhat of a jumbled mess).

However, in other respects, the endpoints are more accurate, since they are closer to the topology of the MST. This also means that simply combining corners, endpoints, and junctions together for the baseline extraction process would involve a lot of duplicates. For example, consider:



(a) Image after MST pruning, overlayed with detected corners.

(b) Image after MST pruning, overlayed with endpoints and junctions

Figure 5.5: **Koi Fish.** Comparison of interest points.

The corners tend excessively separate the Koi Fish. However, they are more accurate in the centre portion of the picture, where the detailed, swirl-shaped artwork is done.

This opens the door to several options:

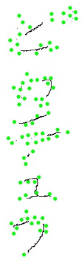
1. **Identify areas where the junctions and endpoints are lacking and add corners.** This would involve setting a minimum radius. If no endpoint or junction was detected within that radius, the corners would be added. The radius would likely be based on the local stroke thickness.
2. **Identify areas where the junctions and endpoints are 'messy' and replace them with corners.** This is tricky, because it is difficult to define what the term 'messy' means from a mathematical standpoint. One idea is to identify local areas of endpoints-junctions and calculate the distance between all the different points, then calculate the distance between corners in the analogous local area. If the distance had gotten smaller, this would indicate that the endpoints and junctions had been skewed during the pruning process, and that the corners would be a better fit.
3. **Conduct two different baseline center extractions; one with endpoints and junctions, the other with corners.** Then compare the accuracy of the resulting curves.

Regardless, there would have to be a mechanism to associate particular corners with pixel objects. Note that because the corners are calculated on the original image, the associated pixel will have shifted slightly during the clustering and pruning processes.

Note that this improvement was not implemented in this particular project.

Also note that corner detection using the original image is not appropriate for images with thick lines. (For images with thin lines, using the 'clustered pixel image' is less accurate than the original).

(a) Original Image.


(b) Image after MST pruning, overlayed with detected corners.

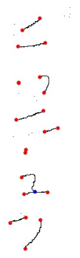

(c) Image after MST pruning, overlayed with endpoints and junctions

Figure 5.6: **Kanji.** Comparison of interest points. Image by Julie Fiala

Since the maximum stroke thickness is detected during pixel clustering, however, it is possible to use this information to decide which image should be used for corner detection.

5.3 Base Centerline Extraction

Next, the point set is separated into distinct curves by calculating the shortest path (on the Complete Cluster Graph) between each connected pair of junctions and endpoints. First, however, it is necessary to figure out which pairs of junctions and endpoints to use. For example, a path may exist between two particular endpoints, but if that path passes through a junction point, then this pair is not suitable for extracting a curve. (Note that the original paper does not address this issue.)

ALGORITHM 4: BASE CENTERLINE EXTRACTION

1. **Group the junctions and endpoints into a single set, and then generate the cartesian product of this set with itself.** This creates all possible pairs of endpoints and junctions.
2. **Remove all junctions and endpoints from the minimum spanning tree.** If the tree has been correctly pruned, then removing a junction disconnects the graph into two separate components. Essentially, each connected component is analogous to a distinct curve.
3. **For each connected component, iterate through the list of 'point pairs' and find the pair that best matches.** This is done by calculating the distance between the new endpoints of the component and the point pair in question. The pair with the minimal distance is selected.
4. **For each pair of selected points, calculate the shortest path on the full cluster graph.** The NetworkX implementation of *Dijkstra's Algorithm* is used.

Unfortunately, it is not an algorithm that scales well. If there are n endpoints and junctions, then the cartesian product is n^2. The number of connected components is also proportional to the number of endpoints and junctions, say cn. All in all, this leads to an algorithm with a runtime of $O(n^3)$.

While optimizations to the algorithm are not implemented, they are certainly possible. For example, each curve currently iterates through all possible pairs of points in order to find the ideal match. Practically speaking, it would be simpler if curve only tested pairs that were close to either of its endpoints. This could be arranged using a variation of the hashed grid that is used to locate neighbours in the Pixel Clustering phase. While creating this grid would be a bit tricky - since points must now be linked to a bin as well as their partner - it could definitely speed up the algorithm.

Code Snippet (Base Centerline Extraction)

```
    components_subgraphs = nx.connected_component_subgraphs(mst)
    point_pairs = []
    counter = 0
    cc_start_time = time.time()
    for comp in components_subgraphs:
        counter += 1
        endpoints = [x for x in comp.nodes_iter() if comp.degree(x) == 1]
        if len(endpoints) != 2: continue

        min_dist = 1000 # arbitrarily large number to start
        pair = []
        for p0, p1 in all_pairs:
            dist1 = util.pointDist(p0, endpoints[0]) + util.pointDist(p1, endpoints[1])
            dist2 = util.pointDist(p0, endpoints[1]) + util.pointDist(p1, endpoints[0])
            if dist1 < min_dist:
                min_dist = dist1
                pair = (p0, p1)
            if dist2 < min_dist:
                min_dist = dist2
                pair = (p1, p0)
        point_pairs.append(pair)
```

Results:

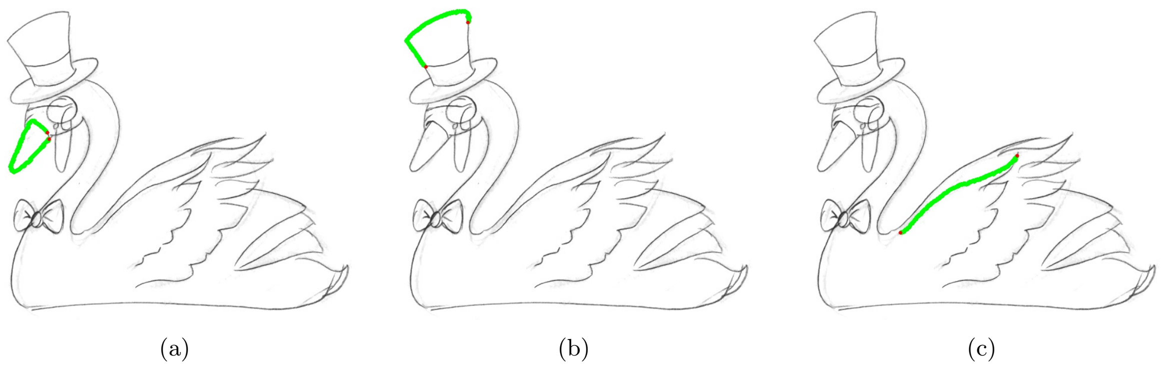

(a) (b) (c)

Figure 5.7: **Swan.** Examples of curves detected using *Algorithm 3*. All the points of the connected component are highlighted in green, and the selected endpoint-junction pair is depicted in red.

Subfigure (a) is an excellent example of another problem encountered using the base centerline extraction. When calculating the shortest path on the complete cluster graph, the result bypasses the beak entirely. In order to correct this, Noris et al. (2013) add an extra step to their algorithm. For each pair of points, they check to see if their are two unique paths between these points that don't pass through any other endpoints or junctions. If so, then an edge is removed from one path, forcing the algorithm to choose the other. This process is then repeated on the second path. This ensures that all paths and pixels are accounted for in the final result. Note that this correction phase was not implemented in this particular project.

Vectorization

6.1 Smoothing

In order to generate aesthetically shaped curves (and correct for any wobbliness caused during clustering or pruning), a *Gaussian Smoothing Operator* is applied to each curve. For each point p_i in a particular curve:

$$\bar{p_i} = \frac{\sum_j w_j p_j}{\sum_j w_j} \tag{6.1}$$

$$w_j = \exp - \frac{D(p_i, p_j)}{2\sigma^2} \tag{6.2}$$

$\forall p_j \in N_i, p_i \neq p_j$. Once again, the hashed grid is used to locate the neighbours of a particular point. Note that the above equations are a simplification of the equations listed in Noris et al. (2013).

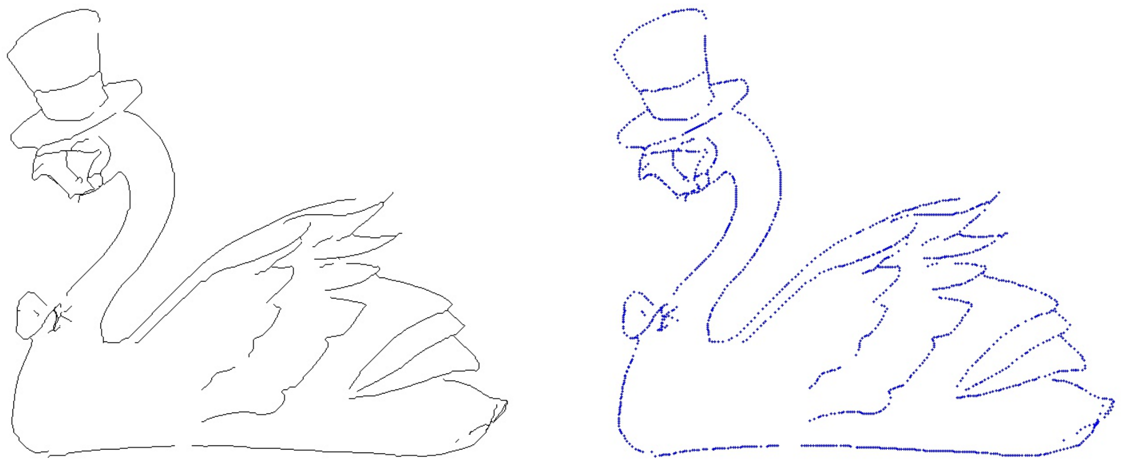
Since only points in a single curve are considered, the smoothing algorithm will not corrupt corners. Effectiveness at this stage is therefore determined by the accuracy of the topology extraction.

In order to avoid re-calculating hashed grids at every iteration, each curve is smoothed individually. The number of smoothing iterations is preset in advance:

Code Snippet: Smoothing Functions

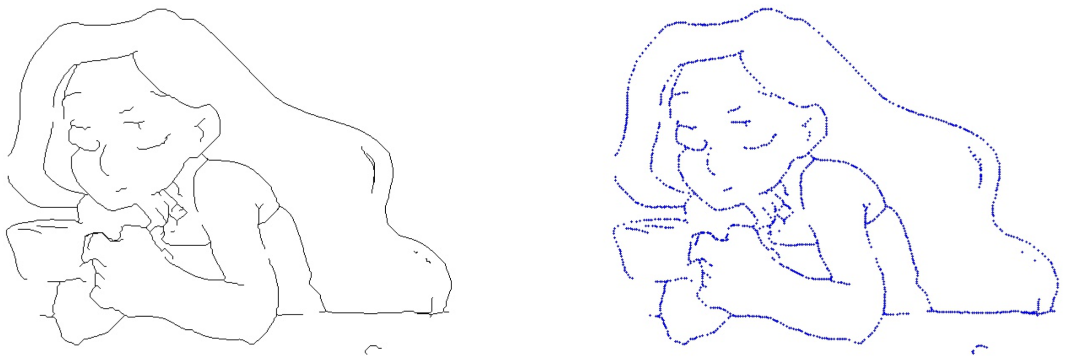
```
""" Iteratively smooth each curve """
def smooth_image(f, curves, img, artist, img_name):
    for curve in curves:
        point_map = create_point_map(curve, img.shape)
        for i in xrange(SMOOTHING_ROUNDS):
            for point in curve:
                gaussian_smoothing_function(f, point_map, point, img.shape)

""" Smoothing function for a single point """
def gaussian_smoothing_function(f, point_map, p1, shape):

    neighbors = util.get_neighbours_x(point_map, p1, shape, NBDH_SEARCH_SIZE)
    idx = (util.R(p1.x), util.R(p1.y))

    if len(neighbors) < 2: return

    sum1 = [0, 0]
    sum2 = 0
    for n in neighbors:
        if p1 == n: continue
        w = math.exp(-(util.pointDist(p1, n)/2*p1.lsw**2))
        sum1[0] += w * n.x
        sum1[1] += w * n.y
        sum2 += w
    if sum2 == 0: return
    p1.x = sum1[0]/sum2
    p1.y = sum1[1]/sum2

    # update point map
    idx2 = (util.R(p1.x), util.R(p1.y))

    # update space in bin
    if idx != idx2:
        point_map[idx].remove(p1)
        if idx2 in point_map:
            point_map[idx2].append(p1)
```

Results:



(a) Before (b) After

Figure 6.1: **Swan**. Comparison before and after Smoothing.



(a) Before (b) After

Figure 6.2: **Coffee**. Comparison before and after Smoothing.

From far away, there is little difference between the two pictures. Essentially, the smoothing algorithm is ineffective (or less effective) if the points in a curve are too 'close'. In other words, if the point are very dense given the length of the curve, then the algorithm is unable to overcome the bias of individual points to create a smooth whole.

While there is some improvement from the smoothing, it is not sufficient to correct the inaccuracies created in the first two phases of the vectorization algorithm.

However, the algorithm is more robust if the points in a curve are scarcer. This can best be seen through close-up examination of the pictures before and after smoothing:

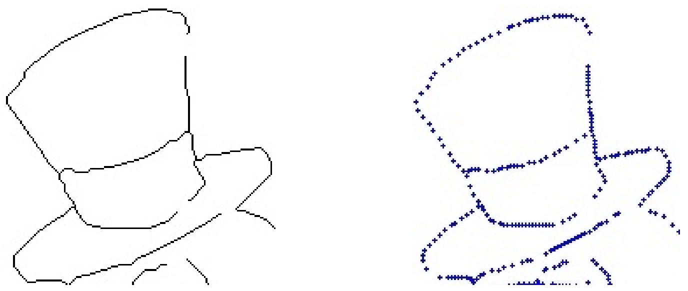

(a) Before Smoothing (b) After Smoothing

Figure 6.3: Closeup of the tophat in **Swan**.

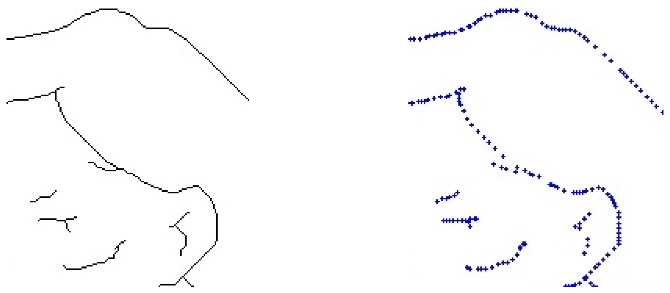

(a) Before Smoothing (b) After Smoothing

Figure 6.4: Closeup bangs, ear, and eye in **Coffee**.

Notice how the lines shift towards more mathematically pleasing curves; some are more successful than others. One potential solution is to only keep a subset of points from each curve. They could be selected randomly, or a more orderly approach (such as choosing every third) could be tried. Ultimately, by reducing the number of points, the smoothing algorithm will be less prone to 'bumps' and 'noise' that do not contribute to the overall flow of the curve.

The danger here is that some curves may, in fact, depend on all their points, and that thinning them down will destroy import topological information. A compromise may have to be struck; perhaps only curves will a sufficient number of points are thinned. Or, perhaps the total number of points is calculated and compared against the vector distance between endpoints.

Otherwise, the smoothing algorithm contains two constants that contribute to accuracy: the number of smoothing iterations, and the size of the neighbourhood to search. When using a small neighbourhood (say 3×3), the position of individual pixels contributes more to the resulting shape of the curve. This is ideal for small, detailed work, but not for large sinuous curves.

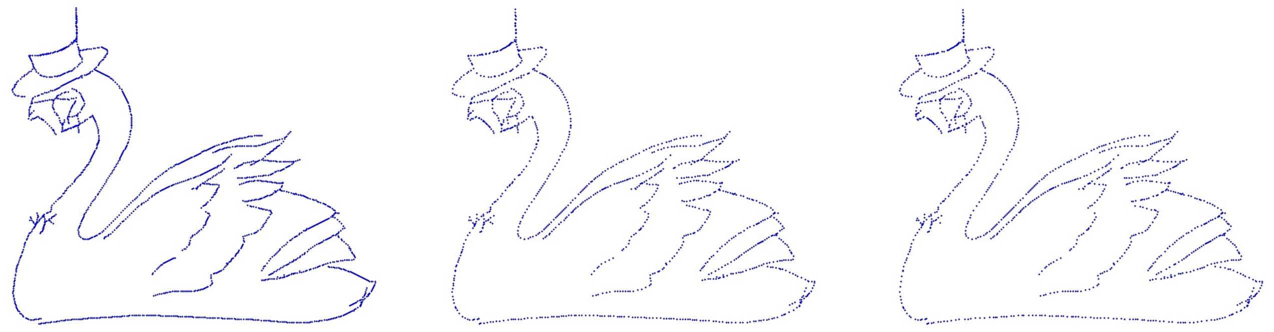

(a) 3 × 3 Neighbourhood. (b) 5 × 5 Neighbourhood. (c) 7 × 7 Neighbourhood.

Figure 6.5: **Swan.** Impact of neighbourhood size in Smoothing Algorithm.

More rounds of smoothing make the lines 'rounder' but to not otherwise impact the drawing. The optimal number of rounds depend upon the intent of the images. 'Cartoon' like images, which rely on exaggerated lines, would benefit from an increased number of smoothing rounds. More 'realistic' images however, should not to be subjected to exaggeration. Ultimately, it a subjective, aesthetic decision, and depends upon the preference of the artist. For the purpose of this project, a 'compromise value' has been selected: 50. Ideally, in practical applications, there would be a way to manually control the number of smoothing rounds.

6.2 Curve Fitting

In the paper by Noris et al. (2013), the program immediately goes from smoothing to reverse-drawing, which is a process made to correct details such as ambiguous junction regions and spikes (sharp, angular regions). There is no curve fitting step.

However, during the smoothing process, gaps appear in between the points as they converge to new spots. For this particular implementation, therefore curve fitting is necessary, and the extra step has been added.

The most popular curve fitting methods in computer graphics are *splines*. A spline is a piece-wise polynomial curve constructed to pass through a series of particular points (known as *knots*). They are popular in computer graphics since they are versatile, easy to construct, and can be combined to approximate very complex shapes. The most common type are cubic splines.

Although the SciPy package contains a few curve fitting algorithms, the MATLAB Curve Fitting Toolbox was chosen to run the initial tests. The MATLAB algorithms are well-implemented, easy to use, and diverse. For this step, it was necessary to write a program that generated MATLAB scripts automatically from the series of curves provided. From there, the scripts were easily run in the MATLAB environment.

Ideally, once the desired curve fitting algorithm was found, it can be implemented in Python to automate the entire vectorization process.

The simplest MATLAB spline-fitting function (csapi: cubic spline interpolation) created some interesting results:

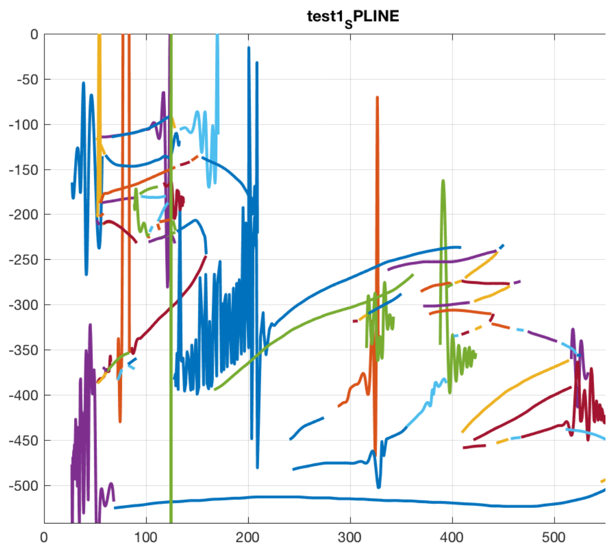


Figure 6.6: **Swan**. Application of **csapi** function to smoothed data.

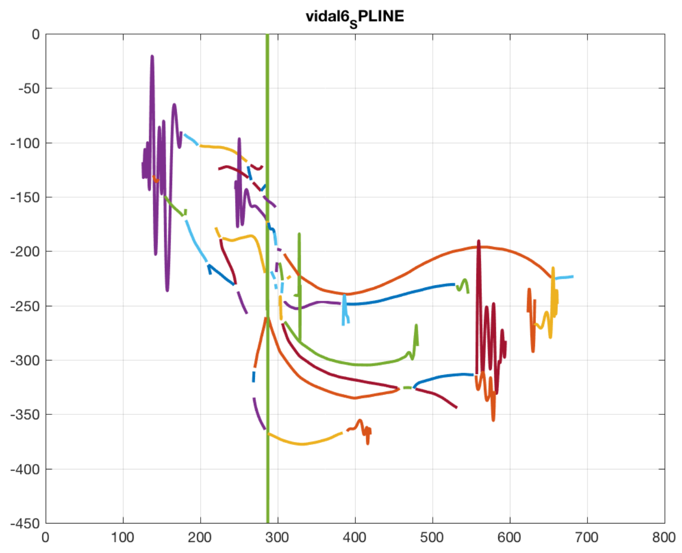
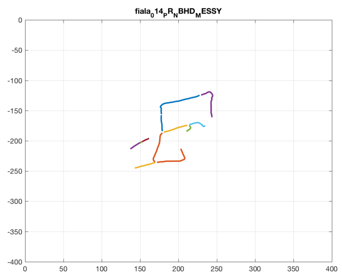
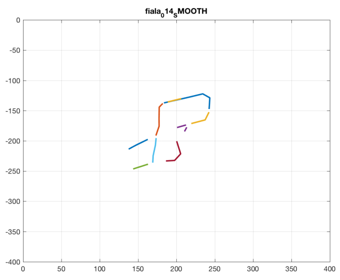


Figure 6.7: **Koi Fish**. Application of **csapi** function to smoothed data.

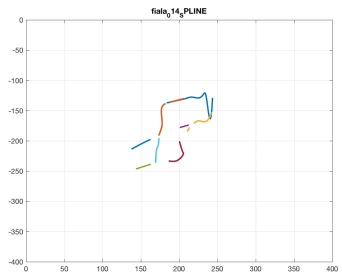
The two above figures demonstrate the limitations of this specific spline-fitting algorithm. Other functions were experimented with:



(a) **cscvn** (”Natural” or periodic interpolating cubic spline curve) over original data.



(b) **csaps** (cubic smoothing spline) over smoothed data.



(c) csapi (cubic spline interpolation) over smoothed data.

Figure 6.8: **Dinosaur.** Vectorization using different curve fitting algorithms.

Using the original, non-smoothed data has the advantage of keeping the curve contained within reasonable distance of the original shape. However, it meant the curves were less ’clean’ and more prone to variation based on individual pixel positions. Furthermore, the natural and smoothed splines created more accurate curves. The regular spline was prone to over-stylize (and in certain cases, was very obviously an out-of-place sinusoidal curve).

It is also worth noting that the smoothed data has problems at ’junctions’, were the fitted curves fail to connect. This is a byproduct of the smoothing algorithm, which was displaced certain curve endpoints. Most likely, it could be fixed be re-adding the endpoints to each curve.

In short, at this stage the original image has been transformed from a raster sketch into a list of cubic splines, and in thus vectorized. There is no doubt that the program requires several adjustments in order to correctly deal with details. However, the overall intent of the program was successful, in terms of absolute basic functionality.

Evaluation

7.1 Time Constraints

Image Name	Initial Number of Moving Pixels	**Total Time (s)**
Dinosaur	6,023	42.6
Kitty	10,305	37.9
Kanji	11,227	40.2
Koi Fish	23,990	133.1
Lotus	37,984	181.6
Princess	38,102	176.4
Wave	41,933	195.0
Coffee	44,282	300.9
Swan	44,804	260.4
Fawn	53,463	285.5
Attitude	70,939	370.6
Eye	74,384	395.3
Octopus Hair	95,923	986.2
2014	109,308	1014.0
Natt	124,535	1256.1
Autumn Vignette	182,874	2757.2

Table 7.1: Speed of Vectorization Algorithm on Various Inputs

Overall, the runtime of the algorithm is satisfactory for its current application. It is worth noting that smoothing is the most time-consuming step, accounting for 60-80% of the listed runtime. This is because the smoothing phase is currently the most underdeveloped of the steps. In order to compensate for the relative ineffectiveness of the algorithm in the face of overly dense curves, 200 smoothing rounds are being used in the hopes of correcting some of the inaccuracies caused during topology extraction. Ultimately, this is quite excessive. With proper line-thinning the number of smoothing rounds could be brought down to 100 or even 30, which would slash the runtime of each input.

It is also worth noting that the runtime contains many operations that aren't directly related to vectorization, namely file I/O operations (to track program statistics) and periodically saving images. The latter frequently requires mapping a point set to a 2D image, which takes $O(n)$ time.

However, the trend of the program's runtime seems clear: exponential based on the number of moving points:

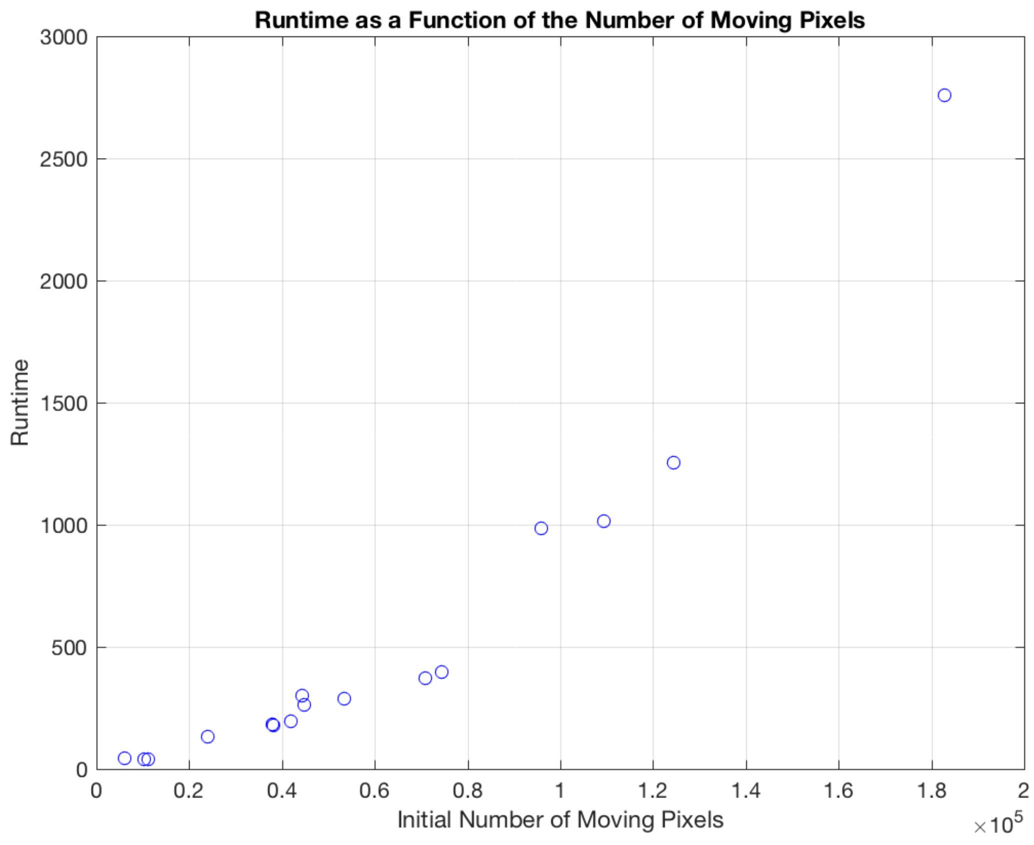


Figure 7.1: Graph of Program Runtime (s) vs. the Number of Moving Points.

While the number of curves (i.e. the level of detail) and the thickness of the stroke may impact the runtime, the sheer number of coloured pixels is the defining factor.

7.2 Space Constraints

Space complexity is tricky to compute because the program is constantly shifting between different inputs: 2D matrices, lists or sets of points, graphs. Most of these are $O(nm)$, where n is the width of the picture and m is the height of the picture; even the most detailed of images will have only a fraction of nm points.

The most space-consuming structure is the Complete Cluster Graph from the Topology Extraction Phase. Not only does it contain $O(nm)$ vertices, but it also has a significant number of edges. A complete graph with n nodes has:

$$\frac{n(n-1)}{2}$$

edges, otherwise known as $O(n^2)$. However, each vertex in the cluster graph is only connected to pixels within its local stroke width; barely a fraction of the total number of nodes.

In conclusion, the space complexity of the graph should still be much closer to $O(nm)$ than $O(n^2m^2)$, but the constant is likely quite high.

Conclusion

In conclusion, this report presents a detailed vectorization algorithm based off of the work of Noris et al. (2013). It includes original improvements, suggestions for further work, and detailed assessments of the algorithm's strengths and weaknesses.

Unquestionably, the program leaves much to be desired in terms of accuracy. While the overall 'feel' of an image can nicely be captured and vectorized, small details are often clobbered. The Pixel Clustering phase does not deal well with filled areas. Topology extraction causes 'wobbliness' that didn't exist beforehand in certain lines, creates gaps in loops, and destroys some lines entirely. Shortcomings in the vectorization phase prevent the extracted curves from becoming perfectly smoothed. Finally, small gaps between adjacent lines are present in the final image.

However, the report provides concrete steps to fix these problems, and perhaps eventually create robust vectorization software. Furthermore, in terms of time and space algorithms, the program is entirely satisfactory for its intended purpose.

Bibliography

A. Bartolo, K. P. Camilleri, S. G. Fabri, and P. Farrugia. Scribbles to Vectors: Preparation of Scribble Drawings for CAD Interpretation. *ACM*, 2007.

H.-H. Chang and H. Yan. Vectorization of hand-drawn images using piecewise cubic bezier curve fitting. *Pattern Recognition 31*, 1998.

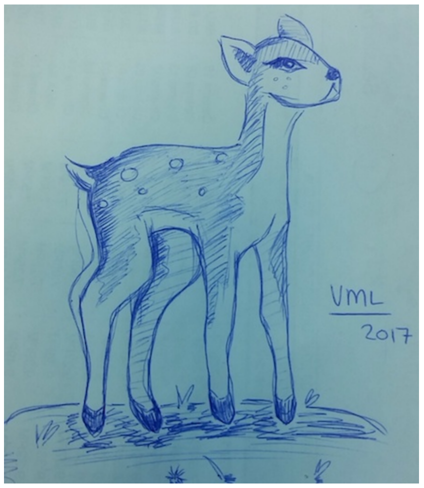
X. Hilaire and K. Tombre. Robust and accurate vectorization of line drawings. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 2006.

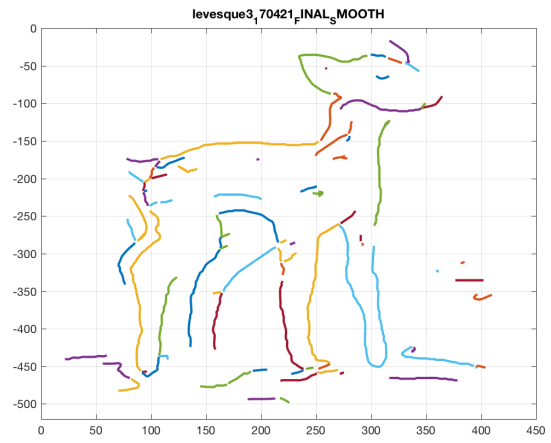
G. Noris, A. Hornung, R. W. Sumner, M. Simmons, and M. Gross. Topology-Driven Vectorization of Clean Line Drawings. *ACM Transactions on Graphics*, 2013.

T. Sezgin and R. Davis. Handling Overtraced Strokes in Hand-Drawn Sketches. 2004. URL http://rationale.csail.mit.edu/publications/Sezgin2004Handling.pdf.

E. Simo-Serra, S. Iizuka, K. Sasaki, and H. Ishikawa. Learning to Simplify: Fully Convolutional Networks for Rough Sketch Cleanup. *ACM Transactions on Graphics (SIGGRAPH)*, 2016.

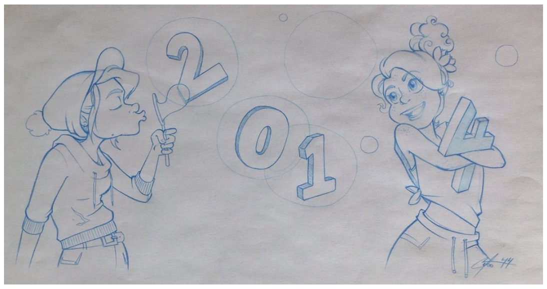
Appendix 1: Additional Outputs

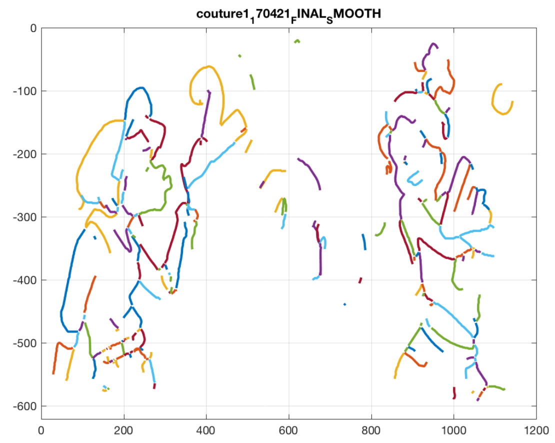

(a) Original Image



(b) Vectorized Image

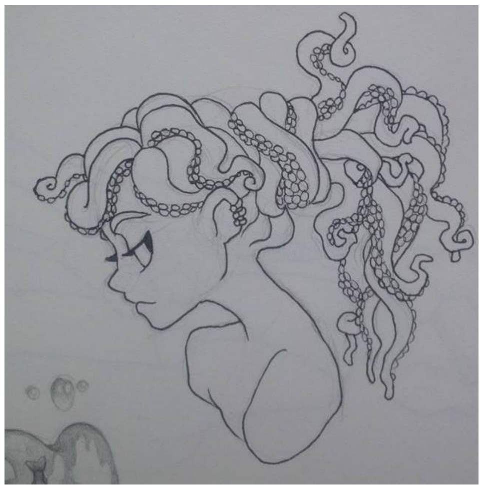
Figure 9.1: **Fawn**. Image by Victoria Levesque.


(a) Original Image

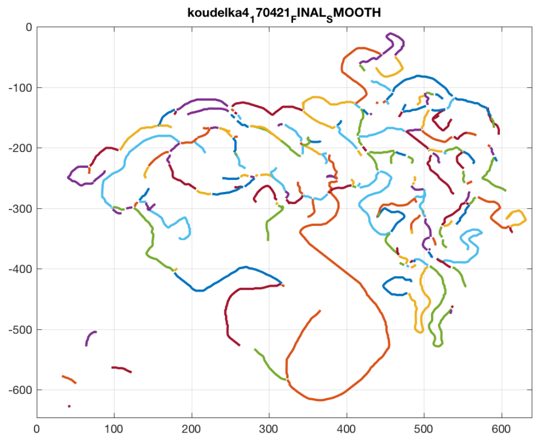


(b) Vectorized Image

Figure 9.2: **2014**. Image by Chloe Couture.

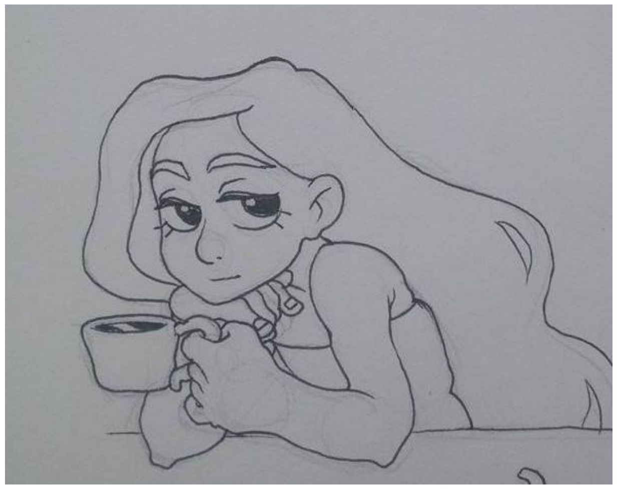


(a) Original Image

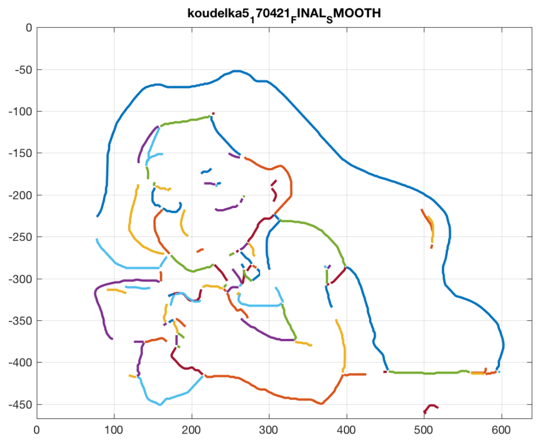


(b) Vectorized Image

Figure 9.3: **Octopus Hair**. Image by Sarah Koudelka.

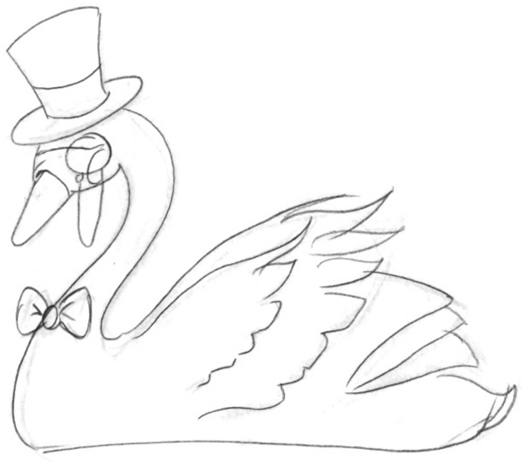


(a) Original Image

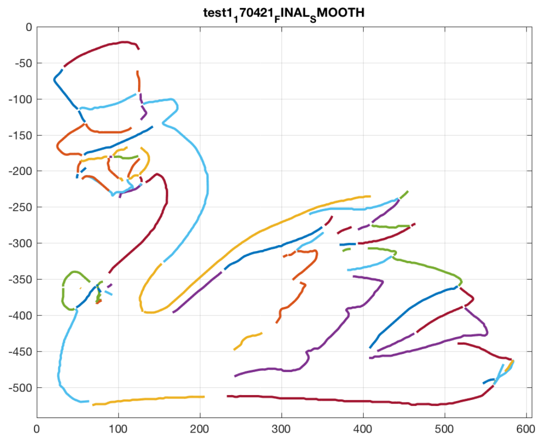


(b) Vectorized Image

Figure 9.4: **Coffee**. Image by Sarah Koudelka.

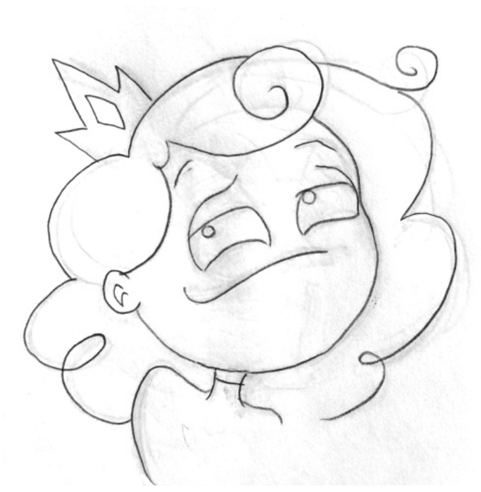


(a) Original Image

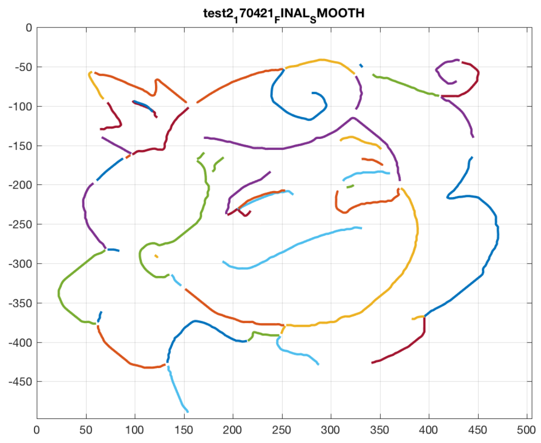


(b) Vectorized Image

Figure 9.5: **Swan**. Image by Patricia Foster.

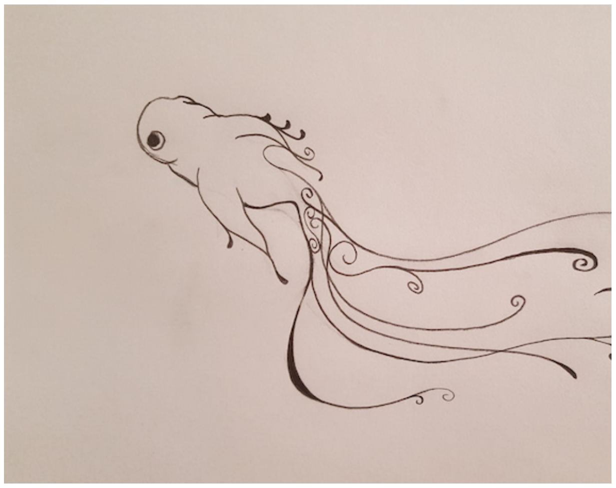


(a) Original Image

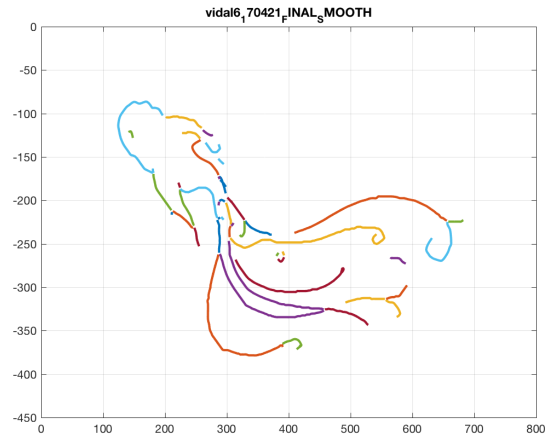


(b) Vectorized Image

Figure 9.6: **Princess**. Image by Patricia Foster.



(a) Original Image



(b) Vectorized Image

Figure 9.7: **Koi Fish**. Image by Alexei Vidal.