

An Accurate System for Fashion Hand-drawn Sketches Vectorization

Luca Donati*

Simone Cesano†

Andrea Prati*

* IMP Lab - Department of Engineering and Architecture - University of Parma, Italy

† Adidas AG - Herzogenaurach, Germany

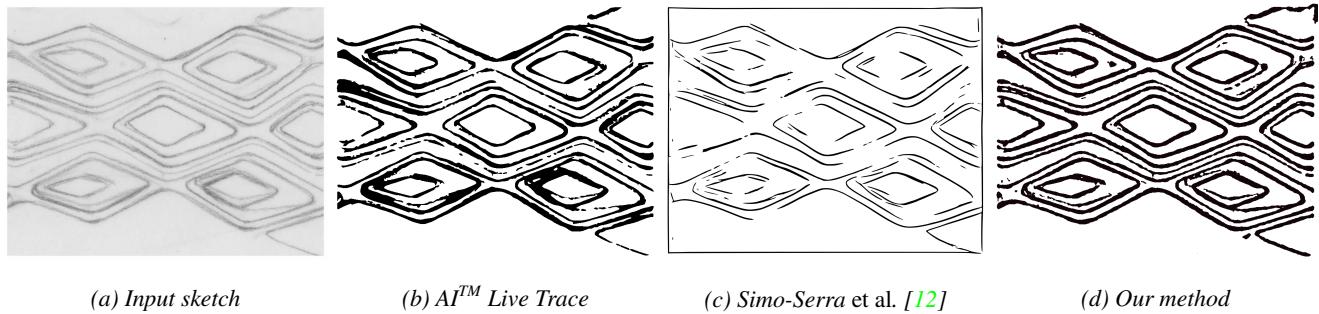


Figure 1: Visual comparison of three line extraction algorithms tested over a portion of a hand-drawn sketch (a). Our proposed method (d) grants optimal “recall” performance compared to state of the art (Simo-Serra et al. [12] (c)), without bargaining its “precision”. Traditional approaches like Adobe Illustrator™ Live Trace (b) fail greatly in both “precision” and “recall” performance, and need to be manually fine tuned.

Abstract

Automatic vectorization of fashion hand-drawn sketches is a crucial task performed by fashion industries to speed up their workflows. Performing vectorization on hand-drawn sketches is not an easy task, and it requires a first crucial step that consists in extracting precise and thin lines from sketches that are potentially very diverse (depending on the tool used and on the designer capabilities and preferences). This paper proposes a system for automatic vectorization of fashion hand-drawn sketches based on Pearson’s Correlation Coefficient with multiple Gaussian kernels in order to enhance and extract curvilinear structures in a sketch. The use of correlation grants invariance to image contrast and lighting, making the extracted lines more reliable for vectorization. Moreover, the proposed algorithm has been designed to equally extract both thin and wide lines with changing stroke hardness, which are common in fashion hand-drawn sketches. It also works for crossing lines, adjacent parallel lines and needs very few parameters (if any) to run.

The efficacy of the proposal has been demonstrated on both hand-drawn sketches and images with added artificial noise, showing in both cases excellent performance w.r.t.

the state of the art.

1. Introduction

Hand-drawn sketches on raw paper are often the starting point of many creative and fashion workflows. Later, the prototype idea from the sketch needs to be converted in a real world product, and at this point, converting the sketch in a full-fledged vectorial image is a requisite. The first crucial step of the vectorization process is to extract line positions and shapes, trying to be as much as possible precise and invariant to image noise and contrast. Algorithms for finding lines provided in the literature are either based on a “straight-lines” assumption, that does not always hold in the case of hand-drawn sketches, or they work for curvilinear lines but are fragile against noisy, real-world images. After performing an accurate line extraction, vectorization can be straightforwardly done using Bezier curves, resulting in a vectorized version of the sketch, suitable for further steps (post-processing, fabric cutting, rendering for marketing, etc.).

Line extraction is an universal task that can be found in a plethora of applications. It must often be both robust and precise in extracting lines from 2D data. Unfortunately,

in many real-world applications the 2D data available are noisy and line extraction can be quite challenging. For this reason, several methods in the literature have addressed this problem. The most immediate approach to line extraction is to work under a straight-lines assumption. Lines are expected to be straight, and models using that assumption are created. They typically work in a parameter space, in which they try to isolate and extract the most probable line locations in an image (e.g., the Hough transform [2]).

If the problem is more general, with curvilinear lines to be found, creating parametric models may become harder. Some proposals on this approach use derivative-based estimations (Steger [13]). Other algorithms focus on obtaining perfect results from clean images, using metrics like sub-pixel Centerline Error (Noris *et al.* [6]). While these methods are very accurate with clear lines, such as those generated from a PC drawing table, or a very neat pen/pencil, they cannot consistently operate with noisier data. Examples of noisy data are rough/dirty hand-drawn sketches or ancient book's text and drawings, corrupted by the time and humidity and having rough background and faint lines. Notably, noisy, corrupted and missing data are common in real-world applications. Other approaches have been proposed to tackle this challenging problem. Most of the less recent approaches proposed ad-hoc solutions, based, for instance, on high-frequency filtering with computer vision operators, adaptive thresholding, and many others. More recent approaches suggested solutions like Gabor filtering (Bartolo *et al.* [1]), and other more sophisticated approaches based on theories in the field of missing information and uncertainty, such as [3, 12].

This paper presents a novel system for automatic vectorization of hand-drawn sketches based on multiple correlation filters which has the distinguishing following aspects:

- the combination of multiple correlation filters allows us to handle variable-width lines, which are common in hand-drawn sketches;
- the use of Pearson's Correlation Coefficient allows us to handle variable-strength strokes in the image;
- even though the proposed algorithm has been conceived for hand-drawn sketches of shoes or dresses, it exhibits good generalization properties, showing to be capable of achieving good accuracy also with other types of images.

The rest of the paper is structured as follows. The next section will briefly introduce the background about cross correlation used in our system for accurate line extraction. Section 3 will describe the overall system for vectorization, with emphasis on the different steps of the system. Section 4 will present the experimental results, performed on both fashion hand-drawn sketches and a specifically created new

dataset for comparison purposes. The last section will draw the conclusions about our work.

2. Background about Cross Correlation

The key idea behind the proposed algorithm is to exploit Pearson's Correlation Coefficient (PCC, hereinafter) and its properties to identify the parts of the image which resemble a "line", no matter the line width or strength. This section will briefly introduce the background about PCC.

The naïve Cross Correlation is known for expressing the *similarity* between two signals (or *images* in the discrete 2D space), but it suffers of several problems, i.e. dependency on the sample average, the scale and the vector's sizes. In order to address all these limitations of Cross Correlation, Pearson's Correlation Coefficient (PCC) [8] can be used:

$$pcc(a, b) = \frac{cov(a, b)}{\sigma_a \sigma_b} \quad (1)$$

where $cov(a, b)$ is the covariance between a and b , and σ_a and σ_b are their standard deviations. From the definition of covariance and standard deviation, this can be re-written as follows:

$$pcc(a, b) = pcc(m_0 a + q_0, m_1 b + q_1) \quad (2)$$

$\forall q_{0,1} \wedge \forall m_{0,1} : m_0 m_1 > 0$. Eq. 2 implies invariance to most affine transformations. Another strong point in favor of PCC is that its output value is of immediate interpretation. In fact, $-1 \leq pcc(a, b) \leq 1$. $pcc \approx 1$ means that a and b are very correlated, whereas $pcc \approx 0$ means that they are not correlated at all. On the other hand, $pcc \approx -1$ means that a and b are strongly inversely correlated (i.e., raising a will decrease b accordingly).

PCC has been used in image processing literature and in some commercial machine vision applications, but mainly as an algorithm for object detection and tracking. Its robustness derives from the properties of illumination and reflectance, that apply to many real-case scenarios involving cameras. Since the main lighting contribution from objects is linear, PCC will give very consistent results for varying light conditions, because of its affine transformations invariance (eq. 2), showing independence from several real-world lighting issues.

Stepping back to our application domain, at the best of our knowledge, this is the first paper proposing to use PCC for accurate line extraction from hand-drawn sketches. Indeed, PCC can grant us the robustness in detecting lines also under severe changes in the illumination conditions, for instance when images can potentially be taken from very diverse devices, such as a smartphone, a satellite, a scanner, an x-ray machine, etc.. Additionally, the "source" of the lines can be very diverse: from hand-drawn sketches, to fingerprints, to paintings, to corrupted textbook characters,

etc.. In other words, the use of PCC makes our algorithm generalized and applicable to many different scenarios.

3. Description of the System

3.1. Pearson's Correlation Coefficient applied to images

In order to obtain the punctual PCC between an image I and a (usually smaller) template T , for a given point $p = (x, y)$, the following equation can be used:

$$pcc(I, T, x, y) = \frac{\sum_{j,k} (I_{xy}(j, k) - u_{I_{xy}})(T(j, k) - u_T)}{\sqrt{\sum_{j,k} (I_{xy}(j, k) - u_{I_{xy}})^2 \sum_{j,k} (T(j, k) - u_T)^2}} \quad (3)$$

$\forall j \in [-T_w/2; T_w/2]$ and $\forall k \in [-T_h/2; T_h/2]$, and where T_w and T_h are the width and the height of the template, respectively. I_{xy} is a portion of image I with the same size of T and centered around $p = (x, y)$. $u_{I_{xy}}$ and u_T are the average values of I_{xy} and T , respectively. $T(j, k)$ (and, therefore, $I_{xy}(j, k)$) is the pixel value of that image at the coordinates j, k computed from the center of that image.

It is possible to apply the punctual PCC from eq. 3 to all the pixels of the input image I (except for border pixels). This process will produce a new image which represents how well each pixel of image I resembles the template T . In the remainder of the paper, we will call it PCC . It is worth remembering that $PCC(x, y) \in [-1, 1], \forall x, y$. To perform just this computation, the input grayscale image has been inverted; in sketches usually lines are darker than white background, so inverting the colors gives us a more “natural” representation to be matched with a positive template/kernel.

3.2. Template/Kernel for extracting lines

Our purpose is to extract lines from the input image. To achieve this, we apply PCC with a suitable template, or *kernel*. Intuitively, the best kernel to be used to find lines would be a sample approximating a “generic” line. A good generalization of a line might be a 1D Gaussian kernel replicated over the y coordinate, i.e. $KLine(x, y, \sigma) = gauss(x, \sigma)$.

This kernel achieves good detection results for simple lines, which are composed of clear (i.e., well separable from the background) and separated (from other lines) points. Unfortunately, this approach can give poor results in the case of multiple overlapping or perpendicularly-crossing lines. In particular, when lines are crossing, just the “stronger” would be detected around the intersection point. If both lines have about the same intensity, both lines would be detected, but with an incorrect width (extracted thinner than they should be).

Considering these limitations, a full symmetric 2D Gaussian kernel might be more appropriate, also

considering the additional benefit of being isotropic: $KDot(x, y, \sigma) = gauss(x, \sigma) \cdot gauss(y, \sigma)$.

This kernel has proven to solve the concerns raised with *KLine*. In fact, it resembles a dot, and considering a line as a continuous stroke of dots, it will approximate our problem just as well as the previous kernel. Moreover, it behaves better in line intersections, where intersecting lines become (locally) *T-like* or *plus-like* junctions, rather than simple straight lines. Unfortunately, this kernel will also be more sensitive to noise (e.g. paper roughness, “dirt”, erased traits), so the system will need a better post-processing phase to filter out unwanted background.

3.3. Achieving size invariance

One of the major objectives of this method is to detect lines without requiring finely-tuned parameters or custom “image-dependent” techniques. We also aim at detecting both small and large lines that might be mixed together as happens in many real drawings. In order to achieve invariance to variable line widths, we will be using kernels of different size.

We will generate N Gaussian kernels, each with its σ_i . In order to find lines of width w , a sigma of $\sigma_i = w/3$ would work, since a Gaussian kernel gives a contribution of about 84% of samples at $3 \cdot \sigma$.

We follow an approach similar to the scale-space pyramid used in SIFT detector [5]. Given w_{min} and w_{max} as, respectively, the minimum and maximum line width we want to detect, we can set $\sigma_0 = w_{min}/3$ and $\sigma_i = C \cdot \sigma_{i-1} = C^i \cdot \sigma_0, \forall i \in [1, N-1]$, where $N = \log_C(w_{max}/w_{min})$, and C is a constant factor or base (e.g., $C = 2$). Choosing a different base C (smaller than 2) for the exponential and the logarithm will give a finer granularity.

The numerical formulation for the kernel will then be:

$$KDots_i(x, y) = gauss(x - S_i/2, \sigma_i) \cdot gauss(y - S_i/2, \sigma_i) \quad (4)$$

where S_i is the kernel size and can be set as $S_i = \text{next_odd}(7 \cdot \sigma_i)$.

This generates a set of kernels that we will call *KDots*. We can compute the correlation image *PCC* for each of these kernels, obtaining a set of images *PCCdots*, where $PCCdots_i = pcc(\text{Image}, KDots_i)$ with *pcc* computed using eq. 3.

3.4. Merging results

Once the set of images *PCCdots* is obtained, we need to merge the results in a single image that can uniquely express the probability of line presence for a given pixel of the input image. This merging is obtained as follows:

$$MPCC(x, y) = \begin{cases} \max PCC_{xy}, & \text{if } |\max PCC_{xy}| > |\min PCC_{xy}| \\ \min PCC_{xy}, & \text{otherwise} \end{cases} \quad (5)$$

where

$$\begin{aligned} \min PCC_{xy} &= \min_{\forall i \in [0, N-1]} PCCdots_i(x, y), \\ \max PCC_{xy} &= \max_{\forall i \in [0, N-1]} PCCdots_i(x, y). \end{aligned}$$

Given that $-1 \leq pcc \leq 1$ for each pixel, where ≈ 1 means strong correlation and ≈ -1 means strong inverse correlation, eq. 5 tries to retain the most confident decision: “it is definitely a line” or “it is definitely NOT a line”.

By thresholding MPCC of eq. 5, we obtain a binary image called *LinesRegion*. The threshold has been set to 0.1 in all our experiments.

3.5. Post-processing Filtering

The binary image *LinesRegion* will unfortunately still contain incorrect lines due to the random image noise. Some post-processing filtering techniques can be used, for instance, to remove too small connected components, or to delete those components for which the input image is too “white” (no strokes present, just background noise).

For post-processing hand-drawn sketches, we first apply a high-pass filter to the original image, computing the median filter with window size $s > 2 \cdot w_{max}$ and subtracting the result from the original image value. Then, by using the well-known Otsu method [7], the threshold that minimizes black-white intra class variance can be estimated and then used to keep only the connected components for which the corresponding gray values are lower (darker stroke color) than this threshold.

3.6. Thinning

The filtered binary image *LinesRegion* now contains only accurate and cleaned line shapes. In order to proceed towards the final vectorization step, an intermediate phase is necessary: thinning. Thinning is an algorithm that transforms a generic binary image (shape) in a set of one-pixel-wide lines. This is highly desirable for the final vectorization step. We decided to stick with the rather standard thinning implementation from Zhang and Suen [14]. Their algorithm is fast enough for our purpose, and it produces thinned lines that can be 1 pixel or 2 pixels wide. An erosion-like mask is then applied to obtain the desired single pixel wide skeleton, called *LinesSkeleton*.

Zhang-Suen’s thinning algorithm (like many other thinning algorithms) produces somewhat biased skeletons when reducing steep angles. If the application needs high precision over strong angles an unbiased thinning algorithm (see Saeed *et al.* review of thinning methods [9]) could improve the extraction. Further enhancements to the skeleton (such as pruning and edge linking) are possible but not treated here.

3.7. Vectorization

The final step is vectorization, that involves transforming the obtained *LinesSkeleton* in a full-fledged vectorial image representation. The simplest and most standard vectorial representation exploits Bezier curves. A cubic Bezier curve can be described by four points (P_0, P_1, P_2, P_3) as follows:

$$B(t) = (1-t)^3 P_0 + 3(1-t)^2 t P_1 + 3(1-t)t^2 P_2 + t^3 P_3 \quad (6)$$

where the curve coordinates can be obtained varying t from 0 to 1. If the curve to be approximated is simple, a single Bezier curve may be enough, otherwise it is possible to concatenate many Bezier curves to represent any curvilinear line with arbitrary precision.

To approximate our *LinesSkeleton* we decided to use an adapted version of the fitting algorithm from Schneider [11]. Schneider’s algorithm will accept a curve at a time, not the whole sketch, therefore *LineSkeleton* must be split in single curve segments, called *paths*. A path is simply defined as a consecutive array of 2D coordinates:

$$path : (x_0, y_0), (x_1, y_1), \dots (x_n, y_n) \quad (7)$$

These coordinates will be obtained taking each “white” pixel coordinates from the input “thinned” image *LineSkeleton*. A path can be further grouped in two classes: closed paths (e.g., circles) and open paths (paths for which the first and last point do not coincide). Paths can be stand-alone paths, or connected with other paths via a “junction”. In this case, each one is considered separately. Schneider’s algorithm works only for open paths, but it is easy to extend and use it for closed paths.

The algorithm will approximate an input path, producing as its output one or more Bezier curves. Each Bezier curve will be described by the four points P_0, P_1, P_2, P_3 . This algorithm uses an iterative approximation method, so it can be parametrized to obtain more precision or more smoothing (and different time complexity).

By doing this Bezier approximation for each path in *LinesSkeleton* we obtain the final complete image vectorization. Some example outputs can be seen in Fig. 2.

4. Experimental Results

In order to assess the accuracy of the proposed method, we performed an extensive evaluation on different types of images. First of all, we have used a large dataset of hand-drawn shoe sketches (courtesy of Adidas AG™). These sketches have been drawn from expert designers using different pens/pencils/tools, different styles and different backgrounds (thin paper, rough paper, poster board, etc.). Each image has its peculiar size and resolution, and has been taken from scanners or phone cameras.

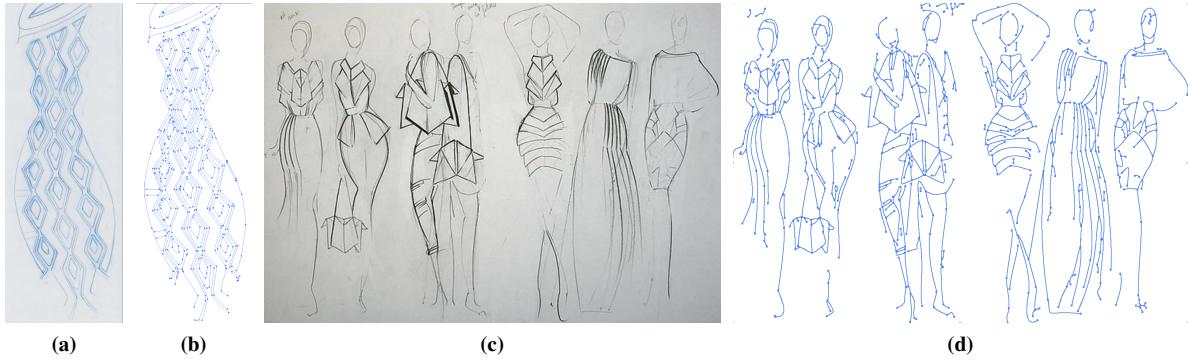


Figure 2: Examples of full vectorizations performed by our system. An Adidas AG™ final shoe sketch (a), (b), and a much dirtier, low resolution, preparatory fashion sketch (c), (d).

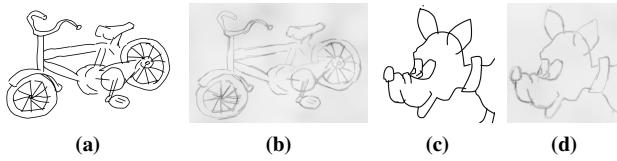


Figure 3: Examples of the “inverse dataset” sketches created from SHREC13 [4].

In addition to that dataset, we also created our own dataset. The motivation relies in the need for a quantitative (together with a qualitative or visual) evaluation of the results. Manually segmenting complex hand-drawn images such as that reported in Fig. 4, last row, to obtain the ground truth to compare with, is not only tedious, but also very prone to subjectivity. With these premises, we searched for large and public datasets (possibly with an available ground truth) to be used in this evaluation. One possible solution is the use of the SHREC13 - “Testing Sketches” dataset [4], alternatives are Google Quick Draw and Sketchy dataset [10]. SHREC13 contains very clean, single-stroke “hand-drawn” sketches (created using a touch pad or mouse), such as those reported in Figs. 3a and 3c. It is a big, representative database: it contains 2700 sketches divided in 90 classes and drawn by several different authors. Unfortunately, these images are too clean to really challenge our algorithm, resulting in almost perfect results. The same can be said for Quick Draw and Sketchy datasets. To fairly evaluate the ability of our algorithm to extract lines in more realistic situations, we have created a “simulated” dataset, called *inverse dataset*. We used the original SHREC13 images as ground truth and processed them with a specifically-created algorithm (not described here) with the aim of “corrupting” them and recreating as closely as possible different draw-

ing styles, pencils and paper sheets. More specifically, this algorithm randomly selects portions of each ground truth image, and moves/alters them to generate simulated strokes of different strength, width, length, orientation, as well as multiple superimposed strokes, crossing and broken lines, background and stroke noise. The resulting database has the same size as the original SHREC13 (2700 images), and each picture has been enlarged to 1 MegaPixel to better simulate real world pencil sketches. Example results (used as inputs for the experiments) are reported in Figs. 3b and 3d.

At this point we performed visual/qualitative comparisons of our system with the state-of-the-art algorithm [12] using various input images. Results are reported in Figs. 1 and 4. It is rather evident that our method performs better than the above method in complex and corrupted areas.

To obtain quantitative evaluations we used the “inverse dataset” and compared our extraction results on it with two other algorithms: the one presented by Simo-Serra *et al.* in [12] and the Adobe Illustrator™’s tool “Live Trace”. As performance metrics, we used precision and recall in line extraction and the *mean Centerline Distance*, similar to the notion of Centerline Error used in [6]. The results are shown in Table 1. Our method outperforms Live Trace and strongly beats Simo-Serra *et al.* [12] in terms of recall, while nearly matching its precision. This can be explained by the focus we put in designing a true image color/contrast invariant algorithm, also designed to work at multiple resolutions and stroke widths. Simo-Serra *et al.* bad “recall” performance is probably influenced by the selection of training data (somewhat specific) and the data augmentation they performed with Adobe Illustrator™ (less general than ours). This results in a global F-measure of **97.8%** for our method wrt **87.0%** of [12]. It is worth saying that Simo-Serra *et al.* algorithm provides better results when applied to clean images; unfortunately, it shows sub-optimal results applied to real shoes sketches (as shown in Fig. 4, last row).

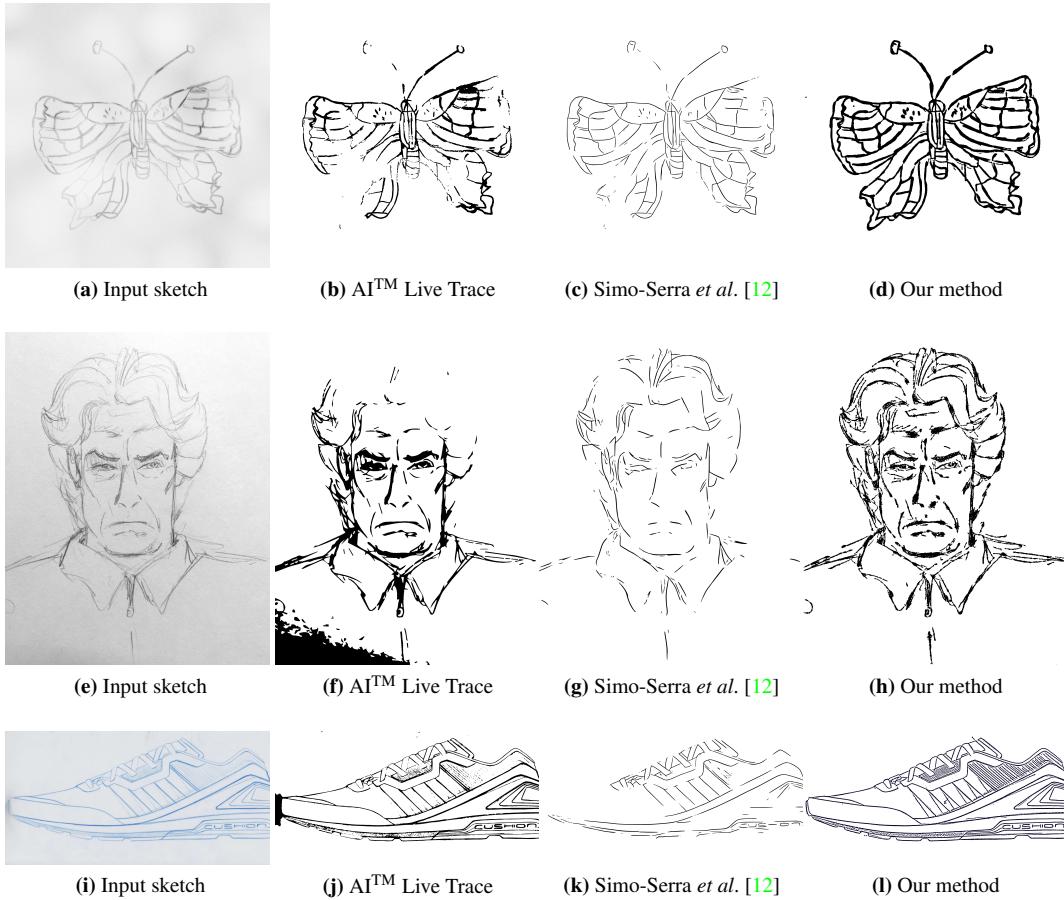


Figure 4: Examples of line extractions from a commercial tool (b), (f), (j), the state of the art algorithm (c), (g), (k), and our method (d), (h), (l). In detail: an image from our inverse dataset (a); a man’s portrait art (e), author: Michael Bencik - Creative Commons 4; a real Adidas AG™ hand drawn design (i)

Finally, we compared the running times of these algorithms. We run our algorithm and “Live Trace” using an Intel Core i7-6700 @ 3.40Ghz, while the performance from [12] has been extracted from that paper. The proposed algorithm is much faster than Simo-Serra *et al.* method (0.64 sec on single thread instead of 19.46 sec on Intel Core i7-5960X @ 3.00Ghz using 8 cores), and offers performance within the same order of magnitude as Adobe Illustrator™ Live Trace’s one (which on average took 0.5 sec).

| | Precision | Recall | Center. Dist. |
|-------------------------------|-----------|--------|---------------|
| Our method | 97.3% | 98.4% | 3.58 px |
| Simo-Serra <i>et al.</i> [12] | 98.6% | 77.9% | 4.23 px |
| Live Trace | 85.0% | 83.8% | 3.73 px |

Table 1: Accuracy of the three implementations over the inverse dataset generated from SHREC13 [4] (2000+ images).

5. Conclusions

Within the complete system for automatic sketch vectorization, the proposed algorithm for line extraction has proven its high accuracy by an extensive evaluation on both “real-world” sketches and a challenging, generated large dataset. It showed the nice property to reconstruct noisy and missing data from images with very different stroke styles, crossing lines, different line strengths and widths, and various noise. In the quantitative comparison, the proposed approach shows the highest recall, confirming its ability to handle complex, missing or corrupted data. Finally, the proposed approach showed to be much faster than other approaches. Possible future directions include to further speed up the approach (exploiting the highly parallelizable algorithms) and to extend and improve it by changing the kernel and/or the merging strategy. This research has been fully funded by Adidas AG™ to which we are very grateful.

References

- [1] A. Bartolo, K. P. Camilleri, S. G. Fabri, J. C. Borg, and P. J. Farrugia. Scribbles to vectors: preparation of scribble drawings for cad interpretation. In *Proceedings of the 4th Eurographics workshop on Sketch-based interfaces and modeling*, pages 123–130. ACM, 2007. 2
- [2] R. O. Duda and P. E. Hart. Use of the hough transformation to detect lines and curves in pictures. *Communications of the ACM*, 15(1):11–15, 1972. 2
- [3] M. Ghorai and B. Chanda. A robust faint line detection and enhancement algorithm for mural images. In *Computer Vision, Pattern Recognition, Image Processing and Graphics (NCVPRIPG), 2013 Fourth National Conference on*, pages 1–4. IEEE, 2013. 2
- [4] B. Li, Y. Lu, A. Godil, T. Schreck, M. Aono, H. Johan, J. M. Saavedra, and S. Tashiro. *SHREC’13 track: large scale sketch-based 3D shape retrieval*. 2013. 5, 6
- [5] D. G. Lowe. Object recognition from local scale-invariant features. In *International Conference on Computer Vision*, 1999, pages 1150–1157. IEEE, 1999. 3
- [6] G. Noris, A. Hornung, R. W. Sumner, M. Simmons, and M. Gross. Topology-driven vectorization of clean line drawings. *ACM Transactions on Graphics (TOG)*, 32(1):4, 2013. 2, 5
- [7] N. Otsu. A threshold selection method from gray-level histograms. *Automatica*, 11(285-296):23–27, 1975. 4
- [8] K. Pearson. Note on regression and inheritance in the case of two parents. *Proceedings of the Royal Society of London*, 58:240–242, 1895. 2
- [9] K. Saeed, M. Tabędzki, M. Rybnik, and M. Adamski. K3m: a universal algorithm for image skeletonization and a review of thinning techniques. *International Journal of Applied Mathematics and Computer Science*, 20(2):317–335, 2010. 4
- [10] P. Sangkloy, N. Burnell, C. Ham, and J. Hays. The sketchy database: Learning to retrieve badly drawn bunnies. *ACM Trans. Graph.*, 35(4):119:1–119:12, July 2016. 5
- [11] P. J. Schneider. Graphics gems. chapter An Algorithm for Automatically Fitting Digitized Curves, pages 612–626. Academic Press Professional, Inc., San Diego, CA, USA, 1990. 4
- [12] 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)*, 35(4), 2016. 1, 2, 5, 6
- [13] C. Steger. An unbiased detector of curvilinear structures. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(2):113–125, 1998. 2
- [14] T. Zhang and C. Y. Suen. A fast parallel algorithm for thinning digital patterns. *Communications of the ACM*, 27(3):236–239, 1984. 4