Vis Comput (2015) 31:717–732
DOI 10.1007/s00371-014-0997-3

ORIGINAL ARTICLE

A vectorization framework for constant and linear gradient filled regions

Ruchin Kansal · **Subodh Kumar**

Published online: 5 July 2014

Abstract Linear gradients are commonly applied in non-photographic artwork for shading and other artistic effects. It is sometimes necessary to generate a vector graphics form of raster images comprising such artwork with the expectation to obtain a simple output and plug it into a traditional workflow, to be further edited and arranged. Many such workflows support only linear gradients and our goal is to generate a standard vector form of the image that can fit such workflow. This vectorization process should be automatic with minimal user intervention. We present a simple image vectorization algorithm that detects regions of linear gradient in potentially noisy images and reconstructs the vector definition on the basis of that information. It uses a novel interval gradient optimization scheme to derive large regions of uniform gradient. We also demonstrate the technique on noisy and hand-drawn portraits.

Keywords Image vectorization · Gradient reconstruction · Region detection

1 Introduction

This paper presents a framework to identify linear gradient regions in digital images and then reproduce their possible original definition for vector graphics that may be inserted into a typical art workflow and further processed. Vector graphics consists of mathematical primitives for the objects present in an image in contrast to a per-pixel bitmap representation. For this reason, vector graphics is considered to be suitable for editing, animation and rendering at varying resolutions.

The process of converting a bitmap to vector graphics is called *vectorization* [2,13–15,21,25,28,29]. It may be viewed as a reverse process of rasterization where a vector image is converted to raster. Significant research exists in the field of vectorization. Many commercial software are also available to perform automatic vectorization. However, in practice this problem tends to be deceptively hard. Standard techniques work well on clean images, for example, those directly rasterized from a vector form. However, when the image is from the wild, i.e., it is noisy with processed edges and shading, many of these methods fail to accurately vectorize it satisfactorily. For example, as shown in Fig. 1, commercial software like Adobe Illustrator [1] and Inkscape [11] approximate the linear gradient definition with solid colored regions while ARDECO [15] creates multiple patches for the same gradient region. Because of this inappropriate gradient construction, any further editing of the vectorized asset is challenging. It is hardly surprising that much recent work in this area has focussed on the need to use higher order gradients for accurate vectorization. However, such gradients are not always supported in actual workflows, which often require a series of software and only the gradients supported by all the stages of the pipeline can be used. This paper proposes an approach for vectorization of linear gradient regions containing accumulated noise such that the output consists of a small number of vector parts.

Vector graphics may be represented using an open vector format such as *EPS*, *PDF*, or *SVG* [20] or it could be a proprietary format (such as *Adobe Illustrator* or *Corel*). Among open formats, SVG is possibly the most widely used vector format for web and digital media, which we have chosen as our output. Nonetheless, the definition of linear gradient is

R. Kansal (✉) · S. Kumar
IIT, Delhi, Delhi, India
e-mail: rkansal@adobe.com

S. Kumar
e-mail: subodh@cse.iitd.ac.in

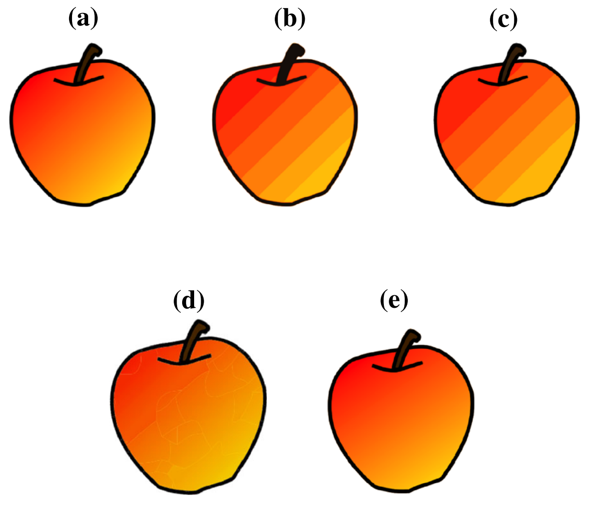


Fig. 1 Comparison of different outputs. **a** Original image, **b** Abode Livetrace output, **c** Inkscape output, **d** ARDECO output, **e** our output

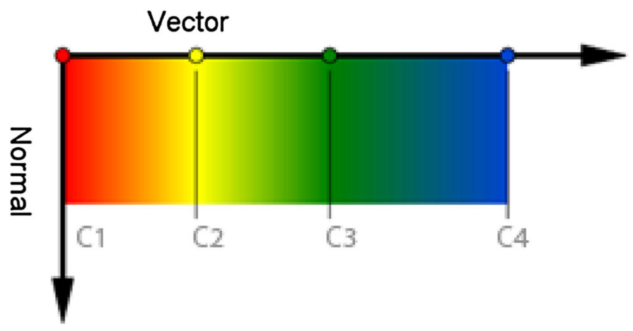


Fig. 2 Linear Gradient defined by four gradient stops (*C1*, *C2*, *C3* and *C4*). Notice the color along gradient vector is defined by linear interpolation from one stop to another, but the color of a pixel along gradient normal remains same

largely the same in prevalent vector standards. SVG defines *Linear Gradient* as continuous smooth color transition along a 2D direction from one given color at a known position to another. This direction is called the *Gradient Vector*. The value of each pixel along the gradient vector may be calculated by linearly interpolating the two end colors. The *Gradient Normal* is the vector perpendicular to the gradient vector. The color of each pixel on the gradient normal is the same. The SVG standard also allows fixing of more than two colors along the gradient vector, to form a smooth multi-color transition. These specific points on the gradient vector with pre-defined color values are called *Gradient Stops* or *Color Stops* (see Fig. 2). Unless otherwise stated, the gradient in this paper refers to the linear gradient at a pixel which is represented by the slope of line in the direction of the maximum change of color in its neighborhood.

We have developed an approach that can detect linear gradient filled regions as well as the gradient values. While the contributions of this paper are primarily in effective recovery of regions with uniform gradient, for completeness we do also produce boundary curves and regions with uniform fill color where necessary. We do not target vectorization of photographic quality images, but rather art-design by artists. The distinguishing feature of such images is that they contain relatively large areas of uniform fill and gradients, but suffer from noise and other smoothing and post-processing artifacts. The main contributions of this paper are:

1. A novel interval gradient optimization scheme to derive large regions of uniform gradient.
2. The ability to reconstruct the linear gradient definition by estimating gradient stops and direction.
3. The ability to generate a small number of regions to keep the output vectors editable.
4. The ability to refine the segments further in multiple passes based on an error analysis.

2 Previous work

Much research has been performed in the field of vectorization. A good discussion of vector primitives related to color gradients is provided by Barla et al. [2]. They describe various available techniques for construction and rendering of such vector primitives. They mention the current methods of vector creation by hand as well as through automated vectorization. Some practical challenges and limitations of these approaches are also explained. The literature relevant to vectorization may be divided into following four broad categories.

2.1 Edge detection-based approaches

Early work in vectorization was focused on only line drawings and bi-tonal images [6,7]. These approaches are mainly based on edge detection [5], thresholding [23,24], thinning [26] and contour tracing [10]. The extracted line, image contour or region is represented by vector graphics primitives, e.g., curves and paths.

2.2 Triangular patches

Delaunay triangulation is a popular technique for image segmentation. Many approaches have been suggested using this segmentation basis for vectorization. One such popular vectorization technique is *ARDECO* [15]. It tries to decompose the input image into triangular patches. Each patch is approximated by a constant color, linear or higher order gradient in order to minimize the overall energy. The energy function in their approach is determined by a boundary length function and a curve fitting term as per the Mumford and Shah model [18]. The boundary length functional is computed by using LLoyd's algorithm [16] to decompose the set of points into compact cells. The curve fitting function is defined as a linear combination of a constant color and first or higher order gradient functions.

The weights of terms is controlled by user input. Since the energy functional is quite generic, it can handle images with fine details. At the same time it often produces a large number of patches and consequently it is not possible to edit the final vector graphics easily for post-editing. Further, for large gradient fill regions, it often fails to converge to a result. Also, due to linear constraints the segment boundaries produced by them is often not smooth. Finally, the user needs to adjust several parameters by experimentation. Our algorithm is simpler to use than ARDECO and it discovers only first-order gradients. Further, our algorithm produces fewer regions so that the user can edit the image easily.

Xia et al. [28] propose a vector-based representation in which the image is decomposed into non-overlapping triangular patches with curved boundaries. The color variation over each triangular patch is approximated with a thin-plate spline for every color channel. This allows them to approximate raster images with both smooth variations and curvilinear features. Also, this vector representation is more accurate and compact in comparison to triangulation-based vectorization. In the same paper, they also propose a GPU-based rasterization algorithm to render their patch-based vector format. Although, the representation is powerful and compact, again editability and portability is a concern.

2.3 Parametric mesh representation

Mesh-based techniques claim for a more editable and flexible representation. Price and Barett [3] propose an approach for interactive image editing using object-based vectorization. They allow the user to select an object and then graph cut is used recursively to form a hierarchical object tree. For each object they define a mesh by locating the corner points and doing recursive subdivision. The resulting mesh can be edited by various tools. However, the approach is designed to be driven by user manually. Also, the algorithm does not handle gradient re-construction, it only provides a better means of object construction.

Sun et al. [25] introduce a vectorization approach using *Gradient Mesh*. There, a gradient mesh is defined by a grid using topologically planar rectangular Ferguson patches with mesh-lines. Control points of the mesh have three attributes: position, derivative and color. Because of these attributes, a rich color object may be represented by the gradient mesh by simply varying one or more of these attributes.

While this earlier approach relies on user assistance for mesh initialization and placement, Lai et al. [14] improve that algorithm by generating the gradient mesh automatically. The output of gradient mesh is quite impressive and it can even be applied to photographic images. However, the size of their representation is too unwieldy for further editing and manipulation. Moreover, gradient mesh is not a standard primitive and hence is less portable; gradient meshes cannot be rendered or edited by standard tools.

2.4 Curvilinear feature detection

Diffusion curves [19] form a different approach to represent smooth shaded images. A diffusion curve partitions the space through which it is drawn, defining different colors on either side. These colors may vary smoothly along the curve. In addition, the sharpness of the color transition from one side of the curve to the other can be controlled. Given a set of diffusion curves, the final image is constructed by solving a Poisson equation whose constraints are specified by the set of gradients across all diffusion curves. An automatic diffusion curve coloring algorithm is discussed in [12] such that the resulting diffused image looks very similar to the original source image. They also suggest a way to estimate and store the texture details with diffusion curves on the basis of Gabor Noise, and then generalizing it to store any number of attributes. However, due to the limitations with Poisson equations, the color variations in all raster images may not be represented by this system, especially when the image has sparse features in some areas. Like gradient mesh, it is also not a standard primitive yet, although it has been considered for inclusion in the SVG standard. As noted previously, for common workflows users expect good, editable and automatic vector output generation. ARDECO [15] may generate gradient patches and it improves the appearance, but the output is hardly editable because of too many patches. For examples, please see Table 1. Other approaches like gradient mesh ([14,25], Xia et al. [28], Price and Barett [3]) and diffusion curves [19] also yield good results, but their output is not in the traditional standard form. So, further processing becomes difficult.

3 Our approach

Figure 3 depicts our pipeline. We start with a raster image. In our experiments all input images are 8-bit per-channel RGB images, but the technique is independent of the color format. Like most vectorization approaches, we first segment a filtered version of the image. Color discontinuity imposes segment boundaries. Next, for each segmented region, we determine if it can be represented by a single linear gradient.

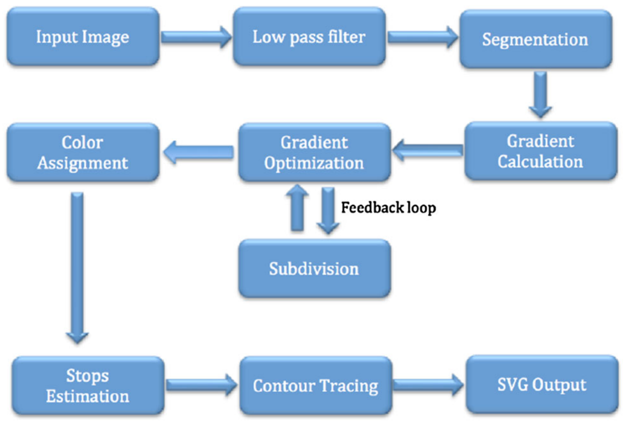


Fig. 3 The complete pipeline

We determine this by searching for a gradient value that is supported by all the pixels of the region. In particular, we compute the range of gradient values supported by each pixel. A global optimization across pixels of the region determines the most plausible gradient for the region.

We perform an error analysis with this optimized gradient. If the error is too high for a segment, we divide it further and recompute the gradients for smaller regions. Finally, using this optimized gradient direction slope for each accepted region, we find its gradient stops. We then trace the contours for each segment and assign the vector information. If the region is identified with a gradient fill, we store the gradient information with the contour, otherwise we perform color averaging over the region and use this average color for region fill approximation. Finally, SVG is output with these details.

As shown in Fig. 3, the pipeline consists of various stages and each stage is dedicated to a specific function. We now discuss the functions and design of each stage.

3.1 Smoothing

To reduce the effect of noise in the image, it is advisable to use a low-pass filter. We have used a Gaussian blur of radius 3 for this purpose. It reduces the sharpness near edges and produces a relatively smooth image.

3.2 Segmentation

We can perform initial image segmentation using a standard scheme. We have performed image segmentation using seed-fill [9] from openCV [4], which generates reasonable segments. A more expensive segmentation algorithm can also be employed, slightly improving the performance, but it is not critical for the purposes of this paper.

3.3 Gradient calculation

From the pixel values, we estimate the unknown gradient direction slope m for each segment. In general, due to independent filters and masks the noise in the images is incoherent and discontinuous. Hence, local gradient estimation followed by standard best-fit regression applied to the entire segment is less effective. We instead apply the best-fit method only locally and then search for the most consistent gradient globally in a segment.

Note that the input color values become especially imprecise due to smoothing, rasterization and rounding. In particular, as the pixels go through a series of image processing steps, computational error adds up and sampled values deviate from the vector values. Rasterization, in particular in areas of low gradient, results in a number of pixels snap to the same color value, before a change is encountered. We conjecture, however, that the pixel color values are largely within a small interval of the precise interpolated color. This range may be larger in case the image undergoes successive rasterizations or filtering. Hence, if the color at pixel p input to this stage is c, we allow that the actual color lies in the interval $[c_- : c_+]$, where $c \in [c_- : c_+]$. For example, if c only has rounding error, $c_- = c - 0.5$ and $c_+ = c + 0.5$. We use ± 0.5 in our experiments.

If the gradient at pixel p is m, we expect the color along the gradient normal, $p + k\frac{1}{m}$, to be in the interval $[c_- : c_+]$ for all values of k within a segment, if the input color at p is c. However, due to the discrete nature of rasterization in the input image, that entire line may miss all pixel centers and hence colors along the line must be reconstructed from the sample values at the pixels.

For this we consider a neighborhood around pixel p and locate the normal line passing through the center of p. We choose a rectangular neighborhood (only the part that lies within the segment). We search for candidates in this neighborhood through which the normal line may pass. Instead of this search through the entire region, it is sufficient to find the range $[c_- : c_+]$ on the boundary of the region, call it R (see Fig. 4). Assuming linear interpolation along R, we find two samples p_1 and p_2 on R such that the color c_1 at p_1 and color c_2 at p_2 contain the range $[c_- : c_+]$, where p_1 and p_2 are the closest such pixels on the contour. In other words, $[c_- : c_+] \subseteq [c_1 : c_2]$. We deduce that the normal line through p intersects R somewhere between p_1 and p_2. As an aside, if one were to search for the exact value c reconstructed from samples near p_1 and p_2, it would produce unreliable estimates for m, which are often inconsistent with the estimates of p's neighbors. The location of the normal line is its estimate at p. This may be inaccurate. A range of normal slopes would be more reliable. We derive this range. Figure 4 and Algorithm 1 describe how.

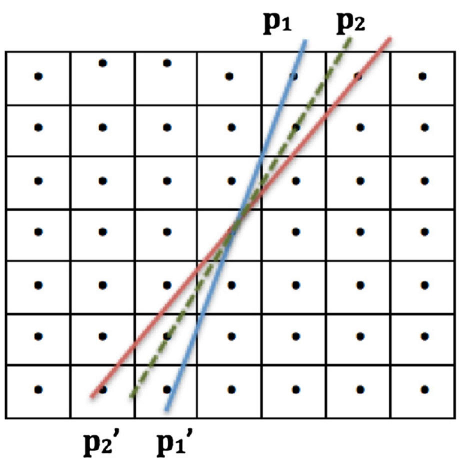


Fig. 4 The setup: a rectangular grid of pixels around a pixel p is considered. The color interval of pixel p, $[c_- : c_+]$, lies in the colors at pixels p_1 and p_2. This implies the normal direction through p, passes between p_1 and p_2. Consider *lines* joining p with p_1 and p_2, respectively. These *lines* intersect at the opposite side of the grid on p'_1 and p'_2. If $[c_- : c_+]$ is spanned by the colors at p'_1 and p'_2, the normal directions is assumed to lie between the two *solid lines*. Additionally, the *green dotted line* is formed by fitting a line among all pixels whose colors are similar to that of pixel p. This estimated slope is also stored for each pixel p

Algorithm 1 Computation of Slope Range for pixel p

```
1: procedure FINDRANGE($p$, $R$) ▷ Calculate range of slopes for pixel $p$ over region $R$
2:     $p.range \leftarrow \emptyset$          ▷ Initialize the range of $p$ as empty
3:     $S \leftarrow$ Boundary pixels of $R$          ▷ Initialize vector $S$ with boundary pixels of $R$ in such an order that all consecutive pixels in $S$ are neighbors in $R$
4:     $i \leftarrow 0$
5:     $c \leftarrow p.color$
6:     for $i < S.size - 1$ do
7:         $p_1 \leftarrow S[i]$
8:         $p_2 \leftarrow S[i+1]$  ▷ Get two neighbor pixels from $S$ in $p_1$ and $p_2$
9:         if $[c_- : c_+] \bigcap [p_1.color, p_2.color] \neq \emptyset$ AND $p.region = p_1.region = p_2.region$ then
10:            $L_1 \leftarrow line(p_1, p)$     ▷ Find line $L_1$ passing through $p_1$ and $p$
11:            $L_2 \leftarrow line(p_2, p)$
12:            $p'_1 \leftarrow intersection(L_1, R)$     ▷ Find intersection of $L_1$ with $R$
13:            $p'_2 \leftarrow intersection(L_2, R)$
14:            if $[c_- : c_+] \subseteq [p'_1.color, p'_2.color]$ then
15:                $p.range \leftarrow [slope(L_1), slope(L_2)]$
16:                break;
17:            end if
18:        end if
19:        $i \leftarrow i+1$
20:    end for
21: end procedure
```

Algorithm 2 Computation of Favored Gradient

```
1: procedure FINDGRADSLOPE($p$, $R$)     ▷ Calculate favored gradient slope of pixel $p$ over region $R$
2:     $p.slope \leftarrow nil$
3:     $Q \leftarrow$ all pixels of $R$     ▷ Initialize vector $Q$ with all pixels of $R$
4:     $i \leftarrow 0$
5:     $S \leftarrow \emptyset$
6:     for $i \neq Q.size$ do                    ▷ Loop on all pixels in $Q$
7:         $q \leftarrow Q[i]$
8:         if $p.color \in [q.color_- : q.color_+]$ AND $p.region = q.region$ then
9:             $S \leftarrow S \bigcup \{q\}$
10:        end if
11:        $i \leftarrow i+1$
12:    end for
13:    if $S \neq \emptyset$ then
14:        $L \leftarrow FitStraightLine(S)$
15:        $p.slope \leftarrow slope(L)$
16:    end if
17: end procedure
```

Recall that the color values are expected to be constant along the entire normal line passing through a pixel. Hence, if p is not on the boundary of its region, a pair (p_1, p_2) implies the existence of another pair (p'_1, p'_2) on the opposite side of the region (see Fig. 4). For pair (p_1, p_2), we extend the lines joining p_1 and p, and respectively, p_2 and p (Fig. 4 explains this construction, see the blue and red lines passing through p). The intersections of these lines with the opposite boundary of contour R provides the conjugate pair (p'_1, p'_2). Again, the image is unlikely to include samples taken precisely at p'_1 and p'_2. We reconstruct the color, respectively, c'_1 and c'_2 at positions p'_1 and p'_2 from the neighboring samples. If again $[c_- : c_+] \subseteq [c'_1 : c'_2]$, it is evidence of the normal line passing between p'_1 and p'_2. If the slopes of lines $p_1 p'_1$ and $p_2 p'_2$ are $\frac{1}{m_1}$ and $\frac{1}{m_2}$, respectively, we say that pixel p supports color gradient in the range $[m_1 : m_2]$ subject to the condition that pairs (p_1, p_2) and (p'_1, p'_2) lie in the same image region. Please note that if the range $[p'_1 : p'_2]$ is not tight and its subset contains the color range $[c_- : c_+]$, that subset is used instead to provide a tighter gradient range.

Not the entire range of gradients $[m_1 : m_2]$ are equally probable, even locally. We assume the likelihood to have a single maxima and a truncated Normal-like distribution. We compute the *favored gradient*, i.e., the most likely gradient m' and attenuate the likelihood for gradients $m \in [m_1 : m_2]$, increasingly for m further away from m'. To find m', we compute the best-fit line to the color values nearest c within R. In particular, we consider the set of pixels S lying within R with color within $[c_- : c_+]$. We compute the least-squares straight line for point in S. The slope of this line is $\frac{1}{m'}$. If m' does not lie in the previously estimated range (m_1, m_2), the pixel is marked as excessively noisy and its estimates are discarded from the optimization step. The process is detailed in Algorithm 2.

Please note that, for some pixels, no value of gradient is favored if any of the following holds:

1. The range of normal lines is not entirely contained within the same region as p.
2. There is no pair of pixels (p_1, p_2) within the region that satisfies the above conditions.
3. All pixels on R all have the same color. This means p possibly lies in a solid fill region.
4. $m' \notin [m_1 : m_2]$.

If a pixel does not produce a gradient range, either it is not a part of a gradient filled region, or it cannot provide candidate gradients due to noise.

Every pixel p_i of a presumed gradient fill region similarly produces its favored gradient m'_i and supported slope range (m_{i_1}, m_{i_2}). We explain next how we choose a single gradient value for the entire region that best satisfies all ranges.

3.4 Gradient optimization

After computing the local gradient for each pixel p_i, favored m'_i and range $[m_{i_1} : m_{i_2}]$, the final gradient m_r for the region should ideally lie in this range and as close to m'_i as possible. We compute m_r by optimizing across all pixels of the region.

We use a function that maximizes its potential if the selected gradient $m_r = m'_i$. This potential monotonically decreases as m_r grows apart from m'_i. With this observation, we can select the optimization function in several ways. For our experiments, we have used a dot product-based optimization function.

Given two vectors aligned with the slopes m' and m'', the dot product of the vectors gives a projection of one on the other. We define our objective function to maximize the sum of such dot products. In particular, the value of objective function $f(x)$ is computed by finding all pixels p_i which have the range $[m_{i_1} : m_{i_2}]$ containing x and performing a summation over the dot products with their favored slope m'_i, i.e.,

$$f(x) = \sum_{i=0}^{n} |g(x, i)|,$$

where $g(x, i) = \hat{x}.\hat{m'_i} | m_{i_1} \leq x \leq m_{i_2}$ and $\hat{x}$ and $\hat{m'_i}$ are unit vectors in the directions of x and m'_i, respectively. This global optimization can be performed using any standard technique. For our experiments, we have used the dynamic system-based global optimization [17,22] from GANSO library [27], since it requires few configuration parameters and converges fast. Moreover, an implementation was already available. This technique starts with a box domain, samples the objective function within this search domain and chooses a number of these values to define the system evolution rules. The algorithm performs sampling in this domain until it converges to a stationary point which is then returned as the solution.

3.5 Color assignment

Every region in our final output is represented by either a linear gradient or a solid color. This solid color value is calculated by the average color of pixels in that segment. The decision for color assignment is presented in Algorithm 5. In short, the algorithm checks whether a sufficient number of pixels favor the optimized gradient. If not, the algorithm checks whether this region may be approximated with its average color. If neither of these holds, it means that the sufficient information is not available for color assignment and the region is subdivided and the approach is applied on all sub-regions. This subdivision is explained in the next section.

3.6 Gradient analysis and re-segmentation

The objective of this step is to refine segmentation if the computed gradients are not satisfactory. To do so, we compute $m_r \cdot m'_i$ for each pixel p_i in the given region. This dot product provides an estimate of the error in gradient for each pixel.

Algorithm 3 Gradient evaluation

```
1: procedure ANALYZEGRADIENT($m_r$, $R$, $R'$)    ▷ Analyze whether
       gradient $m_r$ approximates the complete region $R$. The results of
       re-segmentation are stored in $R'$
2:     $S \leftarrow \emptyset$                     ▷ Initialize set $S$ as empty
3:     $R' \leftarrow \emptyset$                    ▷ Initialize set $R'$ as empty
4:     for every pixel $i$ in $R$ do
5:         $S[i] \leftarrow m_r.m'_i$    ▷ Compute the dot product of $m_r$ with $m'_i$
6:     end for
7:     $R' \leftarrow Segment(S, \epsilon)$ ▷ Re-segment set $S$ with some threshold $\epsilon$
8:     for every region $X$ in $R'$ do     ▷ Eliminate regions which are
           smaller than a threshold $\delta$
9:         if $X.size \leq \delta$ then
10:            Dissolve $X$ in to nearby region since it is too small.
11:        end if
12:    end for
13: end procedure
```

Algorithm 4 Gradient re-computation Feedback Loop

```
1: procedure RECOMPUTEGRADIENT($m_r$, $R$)   ▷ Analyze the gradient
       $m_r$ for region $R$ and subdivides the region $R$, if necessary
2:     $R' \leftarrow \emptyset$              ▷ Initialize region $R'$ as empty
3:     $AnalyzeGradient(m_r, R, R')$    ▷ Call Algorithm 3 to analyze
           gradient for region $R$
4:     while $R'.size \geq \delta$ do   ▷ If region $R'$ is sufficiently small, then
           stop
5:         for every region $X$ in $R'$ do
6:             $m_x \leftarrow OptimizedGradient(X)$           ▷ Compute
                   new optimized gradient $m_x$ for region $X$ as described
                   in Sect. 3.4
7:             $c_x \leftarrow AverageColor(X)$   ▷ Calculate the average color
                   of region $X$ in $c_x$
8:             $AssignColor(X, m_x, c_x)$ ▷ Try color assignment on this
                   sub region now
9:         end for
10:    end while
11: end procedure
```

Algorithm 5 Color Assignment

```
1: procedure AssignColor($R, m_r, c_r$)  ▷ Analyze whether gradient $m_r$ or average color $c_r$ is appropriate for complete region $R$
2:     $count_g \leftarrow 0$  ▷ Initialize $count_g$ as 0
3:     $error_c \leftarrow 0$  ▷ Initialize $error_c$ as 0
4:     for every pixel $i$ in $R$ do
5:         if $m_r \in [m_{i_-} : m_{i_+}]$ then
6:             $count_g \leftarrow count_g + 1$ ▷ Increment $count_g$ if $m_i$ is in the range of slopes which this pixel supports
7:         end if
8:         $error_c \leftarrow error_c + (c_r - c_i) * (c_r - c_i)$  ▷ $error_c$ adds the squared error for this pixel with average color $c_r$
9:     end for
10:    if $count_g \geq \epsilon_g$ then
11:        $R.gradient \leftarrow m_r$
12:    else
13:        if $error_c \leq \epsilon_c$ then
14:            $R.color \leftarrow c_r$
15:        else
16:            $RecomputeGradient(R, m_r)$
17:        end if
18:    end if
19: end procedure
```

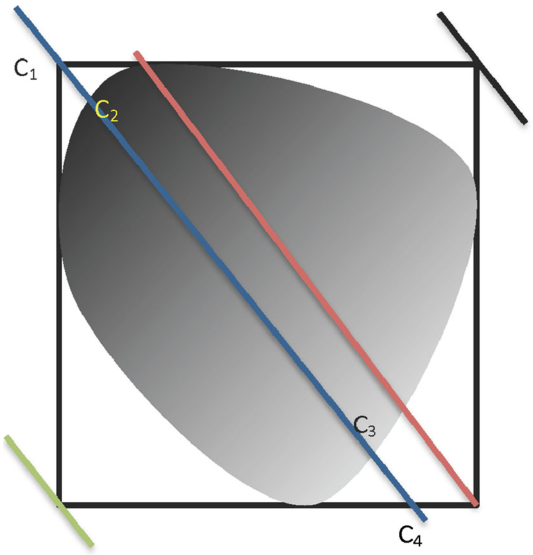


Fig. 5 Gradient Stops Estimation: the shaded area is a gradient filled region while its bounding box is marked as *black rectangle*. We draw *lines* from four corners of the bounding box parallel to gradient axis (shown in *different colors*), since the *line* passing from top-left corner (marked in *blue*) overlaps the maximum pixels of the region, it is used for gradient stops estimation. Two stops are generated where line hits the bounding box(C_1 and C_4) while two stops are generated where *line* intersects the regions (C_2 and C_3). Also, note that value of C_2 and C_3 is determined by using the pixel color at respective location of image, while value of C_1 and C_4 is computed by extrapolation of C_2 and C_3 along gradient axis

We iteratively perform a re-segmentation on the basis of these dot products as described in Algorithm 3. Formation of dot-product segments indicate non-homogeneity of gradients. If this re-segmentation produces non-trivial sub-regions, i.e., the sub-regions are larger than a threshold, we deduce that

Algorithm 6 Adjust Stop

```
1: procedure AdjustStop($P, R, g_r, P'$)▷ Adjusts stop $P$ along $g_r$ for region $R$
2:     $P' \leftarrow P$  ▷ Initialize $P'$ with $P$
3:     if $m_r \notin [m_{p_-} : m_{p_+}]$ then ▷ Check if point $P$ supports optimal gradient $m_r$
4:     else
5:         for every pixel $i$ in $R$ along normal to $g_r$ on $P$ do
6:             if $m_r \in [m_{i_-} : m_{i_+}]$ then  ▷ Check if point $P_i$ supports optimal gradient $m_r$
7:                 $P'.position \leftarrow P.position$  ▷ Copy the stop position from $P$
8:                 $P'.color \leftarrow P_i.color$  ▷ Copy the color from $P_i$
9:                 break;
10:            end if
11:        end for
12:    end if
13: end procedure
```

Algorithm 7 Stops Estimation

```
1: procedure AssignStops($R, m_r, g_r, S$)  ▷ Compute set of gradient stops $S$ along $g_r$ for region $R$
2:     $S \leftarrow \emptyset$  ▷ Initialize $S$ as empty
3:     $\{P_1, P_2\} \leftarrow Intersection(g_r, R)$▷ Find intersection of $g_r$ with $R$
4:     $C_2 \leftarrow ADJUSTSTOP(P_1)$▷ Adjust stop $P_1$ to the best color value in neighborhood
5:     $C_3 \leftarrow ADJUSTSTOP(P_2)$
6:     $C_1 \leftarrow Extrapolate(C_2)$  ▷ Extrapolate $C_2$ to place it on bounding box
7:     $C_4 \leftarrow Extrapolate(C_3)$
8:     $S \leftarrow \{S \bigcup \{C_1\}\}$  ▷ Include $C_1$ in $S$
9:     $S \leftarrow \{S \bigcup \{C_2\}\}$
10:    $S \leftarrow \{S \bigcup \{C_3\}\}$
11:    $S \leftarrow \{S \bigcup \{C_4\}\}$
12:    for every pixel $P_i$ in $R$ along $g_r$ between $P_1$ and $P_2$ do▷ check for all pixels on the selected line
13:        $c'_i \leftarrow Interpolate(S)$  ▷ Compute color $c'_i$ using interpolation from color stops in $S$
14:        if $|color_i - c_i| > \delta$ then  ▷ Check the deviation of computed color $c'_i$ with actual color $color_i$
15:            $S \leftarrow \{S \bigcup \{ADJUSTSTOP(P_i)\}\}$ ▷ Generate a color stop at $P_i$ if the color difference is above a threshold
16:        end if
17:    end for
18: end procedure
```

the original gradient estimate m_r is not appropriate for the entire region. Therefore, we optimize the gradient again for these smaller regions (see Algorithm 4). We employ this step iteratively until the color of each segment is assigned.

3.7 Gradient stops estimation

Once we finalize the gradient vector for a region, m_r, it is important to find the gradient stops in order to reconstruct the original colors. To find these stops accurately, we need a vector parallel to m_r that overlaps a sufficiently large number of pixels in the region.

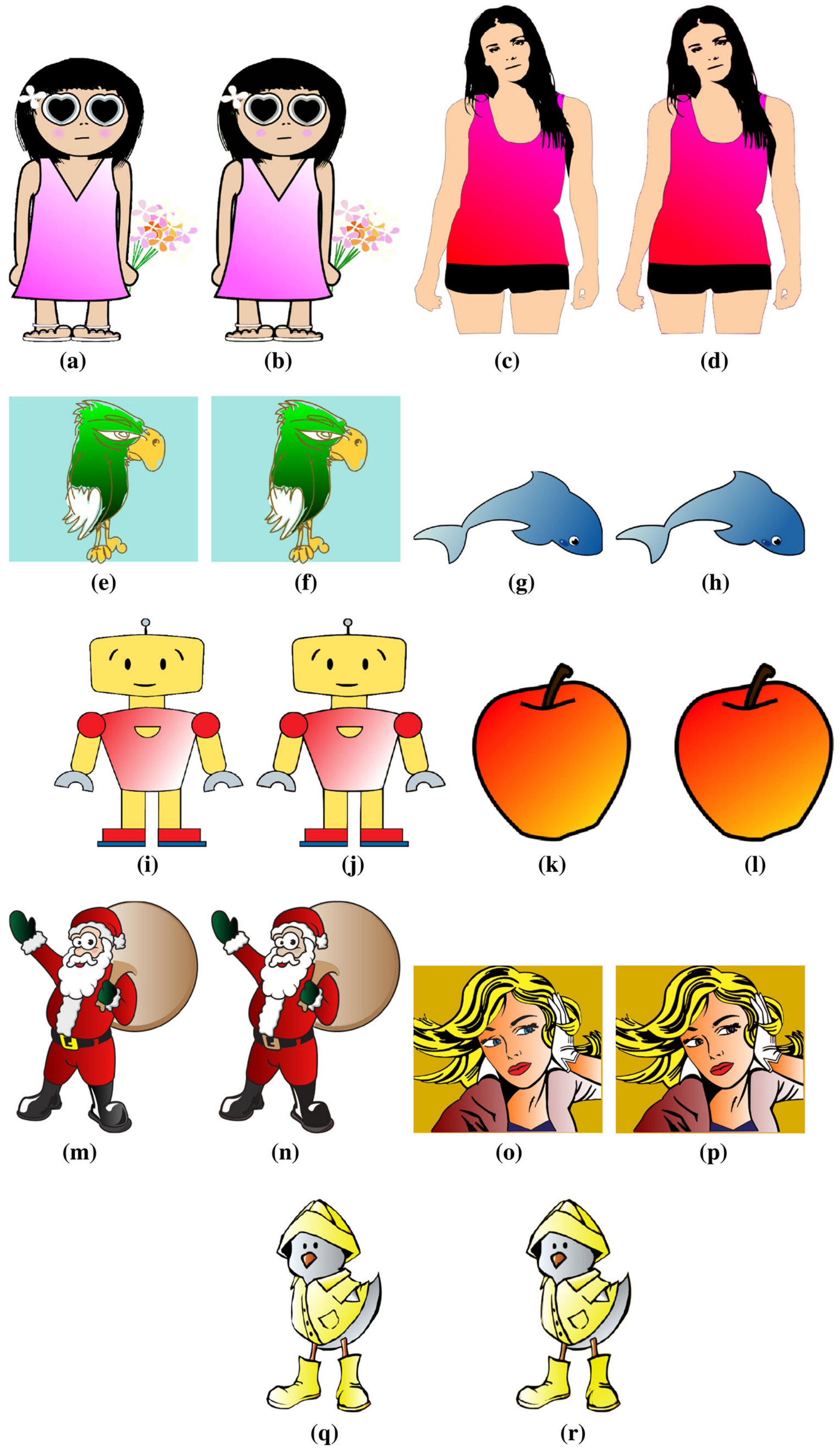


Fig. 6 The results with our approach. Original image is on the *left* and the final vector image is shown on *right* (contd. on next page)

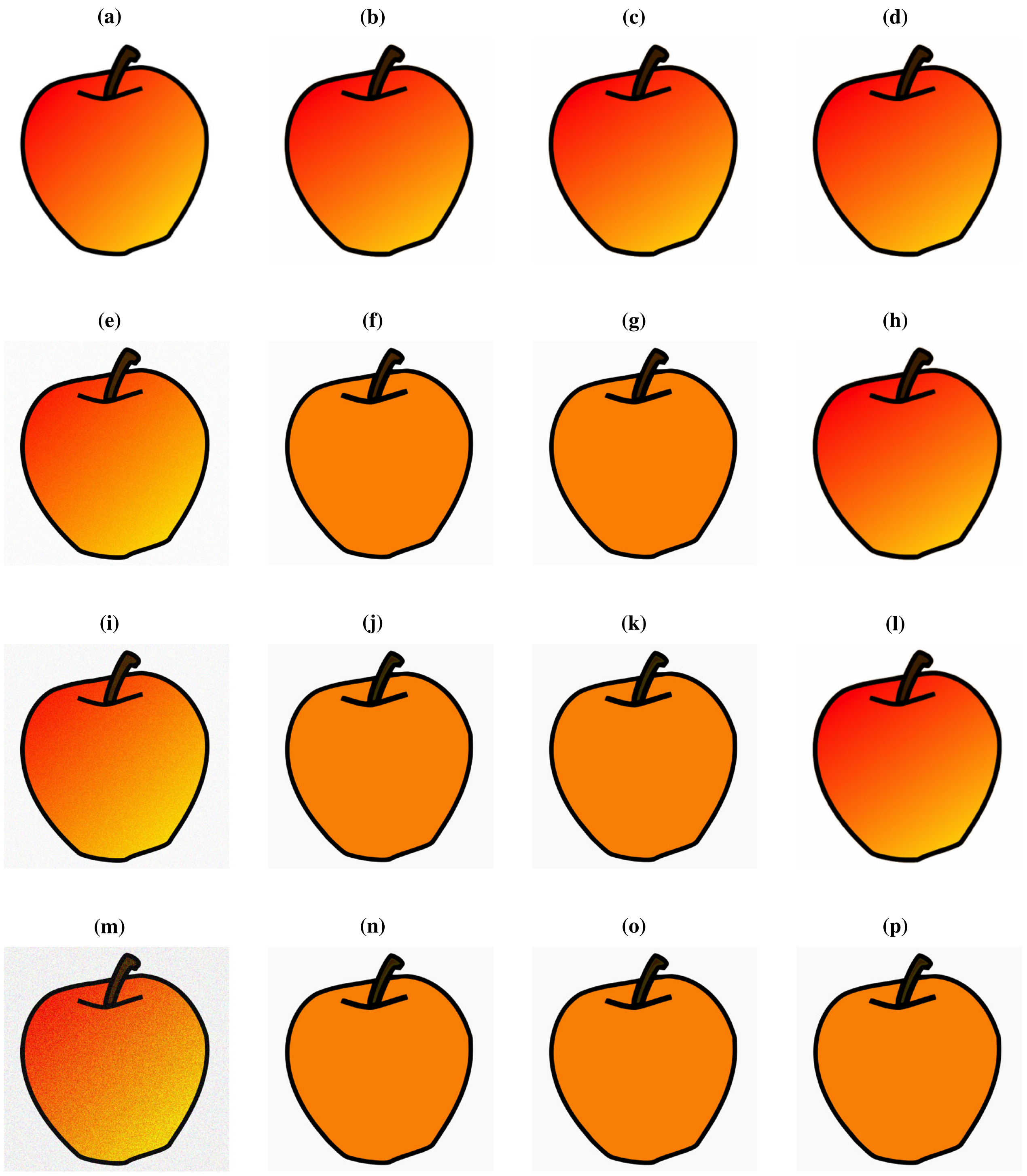


Fig. 7 Comparison of Sobel filter-based approach on noisy images. Kernel size of sobel filter was 5 for this example. **a** Input image without artificial noise, **b** output by taking average gradient, **c** output by using RANSAC, **d** output using our method, **e** input image having low noise, **f** output by taking average gradient, **g** output by using RANSAC, **h** output using our method, **i** input image having more noise than Fig. 7e, **j** output by taking average gradient, **k** output by using RANSAC, **l** output using our method, **m** very noisy input image, **n** output by taking average gradient, **o** output by using RANSAC, **p** output using our method

We use a heuristic to find two approximate intermediate color stops and then generate the end colors by extrapolation. In particular, we draw lines parallel to the computed gradient vector m_r from each corner of the bounding box of the region as shown in Fig. 5. Among the four vectors, the one with the largest overlap with the region is selected for stops' estimation. Note that at least one vector, call it g_r, always has an overlap with the bounded region.

We generate multiple color stops on the gradient vector, two of which lie on the bounding box and two on the region boundary (see Fig. 5). As per the SVG specifications, the stops have to span the bounding box. Therefore, if the gradient stops at C_1 and C_4 do not lie in the region, their colors are estimated using extrapolation from C_2 and C_3 as shown. For a smooth color transition, we generate multiple color stops between C_2 and C_3 by taking multiple color samples along the selected direction (see Algorithm 7). An important aspect to note here is that simply using the color of a point on the gradient line does not produce a satisfactory stop color due to the noise. A local neighborhood search must be performed to compute the most reliable colors supporting the computed gradient as described in Algorithm 6.

3.8 Contour tracing

Tracing is the process of fitting curves that bound each image region. After tracing, we obtain a set of curves that represent the image geometry. We employ the potrace engine developed by Schilinger [21] for this outline tracing. The same engine is also used by open source vector drawing package Inskscape [11].

3.9 SVG output

Once we obtain the curves outlining each image segment, we apply fills to these curves and generate the final vector representation (in the SVG format). The region color, as noted before, may be either a solid fill color obtained through color assignment (Sect. 3.5) or a linear gradient produced by the optimization algorithm (Sect. 3.4).

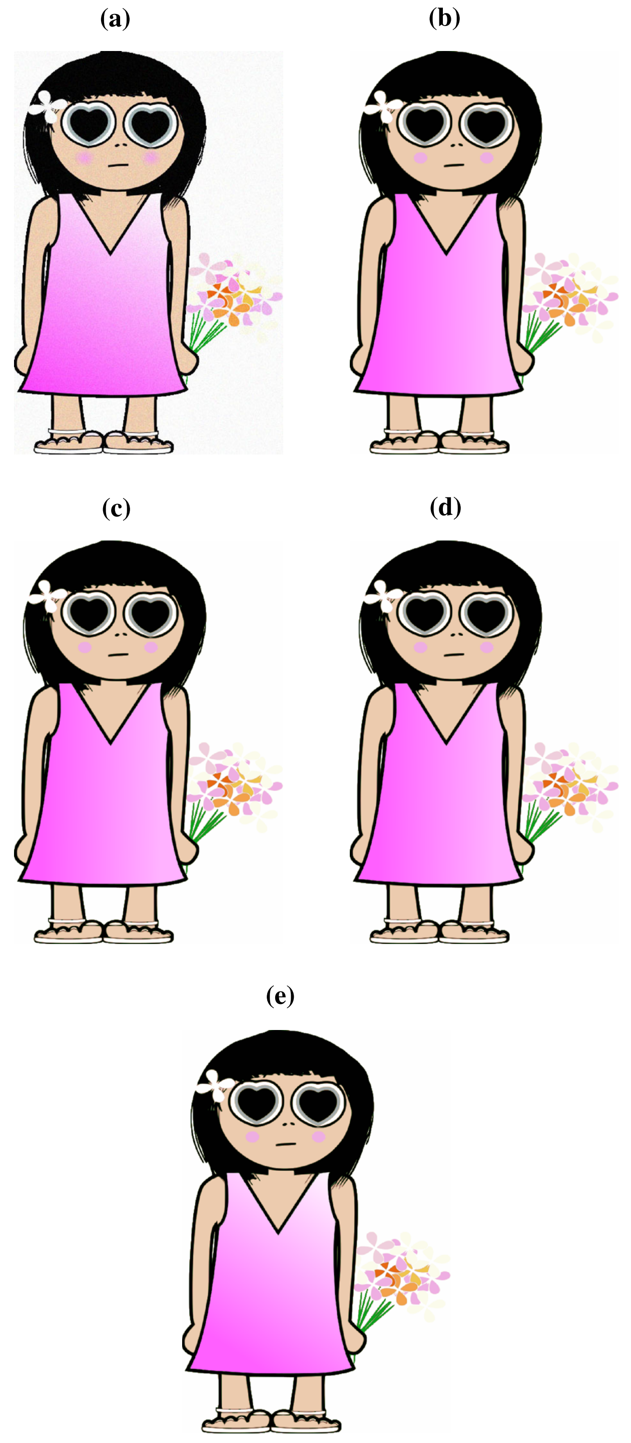


Fig. 8 Results by changing Sobel filter kernel size. **a** Input image, **b** output using mean Sobel gradient of kernel size 3, **c** output using mean Sobel gradient of kernel size 5, **d** output using mean Sobel gradient of kernel size 7, **e** output using our method

4 Results and validations

4.1 Results

Figure 6 shows our results on a few images. The regions of input images that contain linear gradient are identified and then reconstructed successfully. The generated SVG when rasterized resembles the input image closely.

4.2 Comparison with naive Sobel filter-based approach

The computation of 'average' gradients appears to be a simple problem. We evaluate the performance of naive

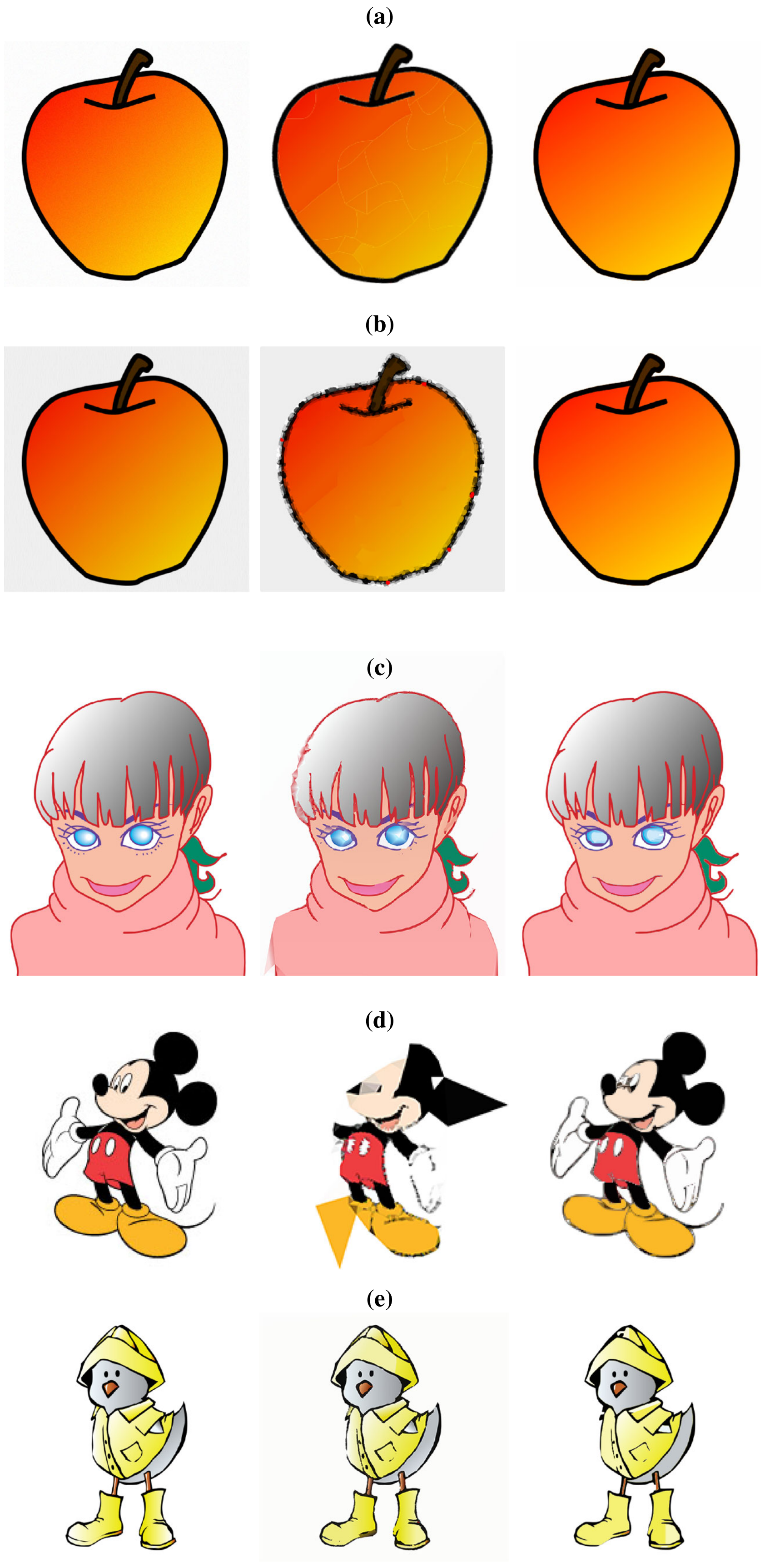


Fig. 9 Noisy images and comparison with ARDECO. Input image is on *left*, ARDECO result is in *middle* and result with our approach is shown on *right*. **a** A noisy image. Random RGB noise was added to the input image, **b** a noisy image. Noise was added using a filter in Adobe Photoshop, **c** an image with a big gradient patch, **d** an image with *solid colors* only, **e** an Image having many gradient patches

Table 1 Comparison with ARDECO on the basis of number of patches generated and root mean squared error per pixel

Input	Our results		ARDECO results	
	Patches	Error	Patches	Error
Fig. 9a	4	15.6	1,200	111.48
Fig. 9b	4	15.6	1,200	55.1
Fig. 9c	78	17.17	601	51.70
Fig. 9d	202	37.8	601	94.44
Fig. 9e	90	35.4	601	24.0

approaches. We compute the per-pixel gradients using a Sobel filter kernel of size 5. Then to find the optimized gradient for whole region, we employ two simple techniques.

In the first, we simply calculated the mean of the gradient values for each region.The second technique used a least squares formulation augmented with RANSAC-based outlier rejection [8]. For each region we provided all the gradient values to the RANSAC algorithm and calculated the RANSAC distance based on the dot product between two gradients. This algorithm removes outliers based on this distance and a threshold, and finally outputs an optimized value.

Both of these techniques work satisfactorily for simple images as shown in Fig. 7a and the results are comparable with that of our approach. However, even with a medium degree of noise, these fail and the output is quite different from the original, as shown in Fig. 7f, g. We show the effect of increasing noise in Fig. 7j, k, n, o, p. Please note that with extreme noise, even our technique fails to reproduce the gradient with the default parameters because too many pixels vote for the wrong gradient. Varying the Sobel kernel size does not help in most cases. For example, Fig. 8 uses a kernel size of 7.

4.3 Noisy images and comparison with ARDECO

Our approach works on noisy images also. We added a random RGB noise in the input image (see Fig. 9) and then used it as an input to our approach. The output is reasonably close to the actual gradient. We have also compared it to the ARDECO output. The results with ARDECO were generated using default values for all parameters. The number of patches in ARDECO is much higher than with our approach. Table 1 summarizes the results of comparison with ARDECO.

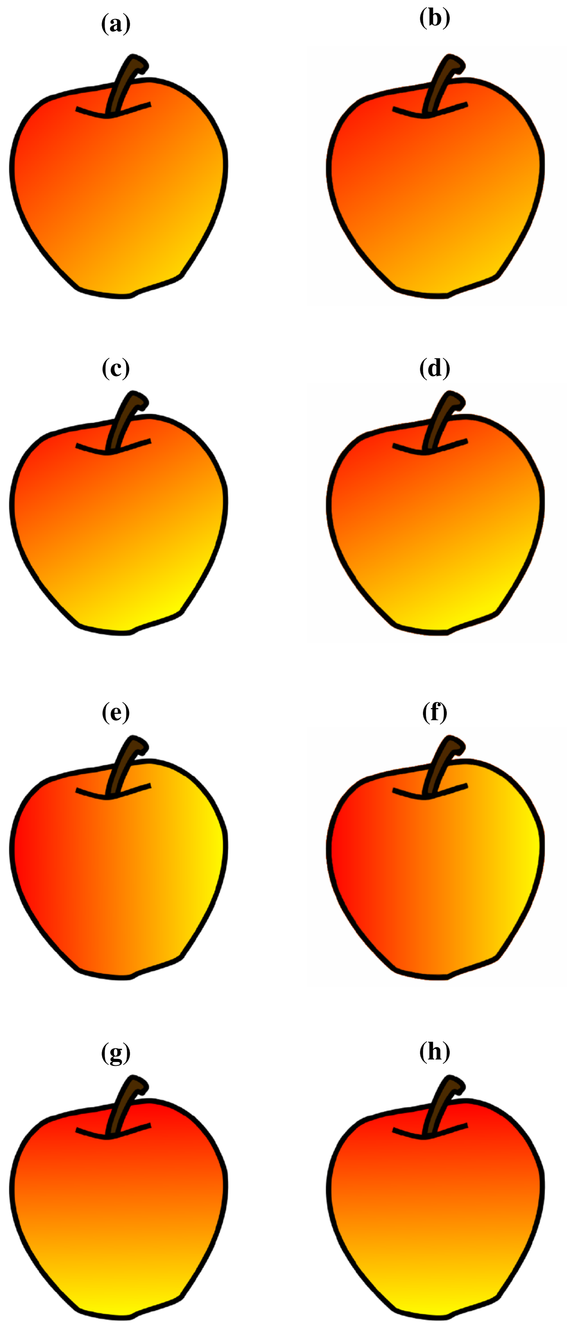


Fig. 10 Comparison of computed gradient with original known gradient in image. **a** Original gradient direction: 1.0, color varying from (255, 0, 0) to (255, 255, 0), **b** computed gradient direction: 1.2, color varying from (255, 4, 0) to (255, 248, 0), **c** original gradient direction:1.75 , color varying from (255, 0, 0) to (255, 255, 0), **d** computed gradient direction:1.85 , color varying from (255, 8, 0) to (255, 242, 0), **e** original gradient direction: 0, color varying from (255, 0, 0) to (255, 255, 0), **f** computed gradient direction: 0, color varying from (255, 2, 0) to (255, 252, 0), **g** original gradient direction: ∞, color varying from (255, 0, 0) to (255, 255, 0), **h** computed gradient direction: ∞, color varying from (255, 2, 0) to (255, 251, 0)

Table 2 We calculated the per pixel error for gradient and solid colored regions separately in our output and then compared with the corresponding region error in Inkscape

Input	Our error		Inkscape error		Patches	
	Gradient	Solid	Gradient	Solid	Ours	Inkscape
Fig. 6e	7.7	16.25	17.49	24.40	18	16
Fig. 6k	11.4	13.2	18.64	21.31	4	16
Fig. 6c	11.63	18.67	14.76	26.55	20	16
Fig. 6i	15.64	25.14	26.20	44.8	25	16
Fig. 6a	25.51	34.26	46.92	37.40	36	16

The number of patches are also compared

4.4 Comparison with vectors

To measure the accuracy, we applied our algorithm to images whose gradient direction and magnitude were already known. The results are shown in Fig. 10.

4.5 Comparison with commercial software

To analyze the per-pixel error in our output, we rasterized our vector output and then compared it with the original image using root mean squared error. Table 2 compares the error in our gradient and solid colored regions with the corresponding regions in Inkscape output. We used a quantization palette size of 16 colors for Inkscape results. Since, Inkscape uses color quantization for patches, the number of output patches are always equal to the quantization palette size, as the results show below.

The per-pixel error in the output can be attributed to several factors:

1. Vector and raster spaces are not congruent. The pixel at location (x, y) in the input image may not be present at the same exact location in the vector space.
2. The input image may contain small pixel-level features, which are merged in larger regions during vectorization. Further, the noise is smoothed over by design.
3. Vector output is optimized to be represented with fewer colors using color minimization.

4.6 Results on hand-drawn portraits

As noted before, our algorithm is designed for non-photorealistic images. The approach works well when the gradient in image rasterized with a software. However, even on hand-drawn portraits our approach generates reasonable results which we also compare with ARDECO (see Fig. 11).

4.7 Performance

Since, we use global optimization as discussed in Sect. 3.4, the output generation takes time. The time taken depends on the number of gradient patches in the image, and the size of these patches. The system currently takes around 5–6 min to produce vectorizations. Vectorization with ARDECO also took almost the same time on these images. Being a one-time initial step, this is generally sufficient. However, with a more tuned optimization using multi-threading and GPU computing, it may be possible to reduce it to seconds.

4.8 Editability

The output SVG can be easily edited using any standard vector graphics tool like Inkscape. Examples are shown in Fig. 12.

4.9 Conclusions and future work

We have presented an approach to find the linear gradient in images that optimizes the gradient values across noisy pixels. We have demonstrated its robustness on a variety of images. It mainly targets reconstruction of simple non-photorealistic art drawings that can then be further edited or stylized. Its application on noisy images is also demonstrated. The output is compared to other gradient-based vectorization approaches like ARDECO.

Our approach does not produce good results when the linear gradient is applied on small width regions, like linear gradient on a single pixel wide curve, for it needs to find segments with a few neighbors around its pixels.

In our approach, we have focussed on detection and reconstruction of linear gradients only, but we believe the algorithm can be adapted to handle non-linear gradients like a radial or circular gradient as well.

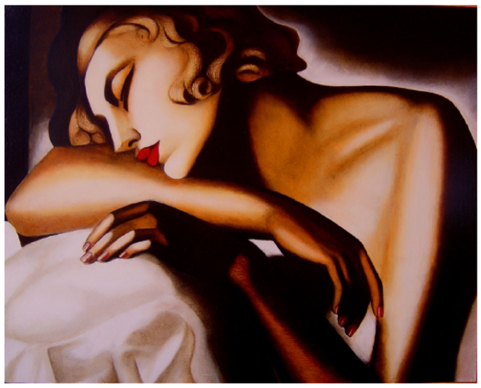
Figure 11 shows that ARDECO performs better on the hand-drawn portraits, but our method performs well on images that have software-generated gradients as shown in Fig. 9. Segmentation should be improved to generate better output with hand-drawn portraits.

Our algorithm is designed to operate on each pixel independently, therefore it can parallelize well. Future work can leverage this aspect and focus on parallelization. It is possible to use both GPU and CPU-based parallelism with this approach.

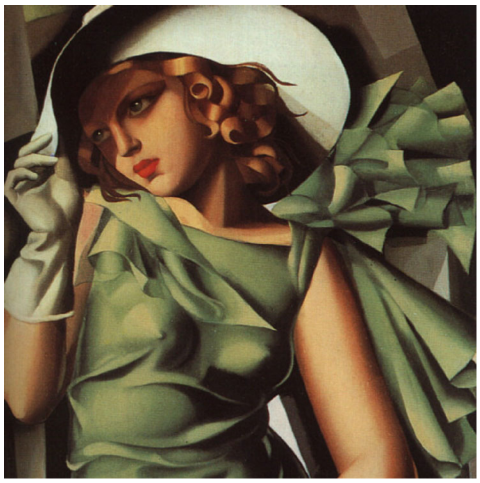
Future work should also include deriving vector graphics for videos and using the level of optimization in a feedback loop to refine the segmentation, potentially producing even fewer patches.

Fig. 11 Results on a hand-drawn portrait. **a** Original portrait, **b** original portrait, **c** our output, **d** our output, **e** ARDECO output, **f** ARDECO output

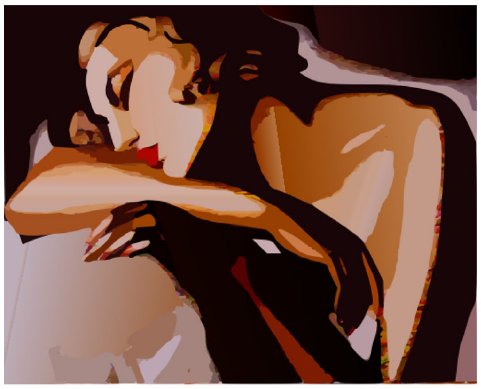
(a)



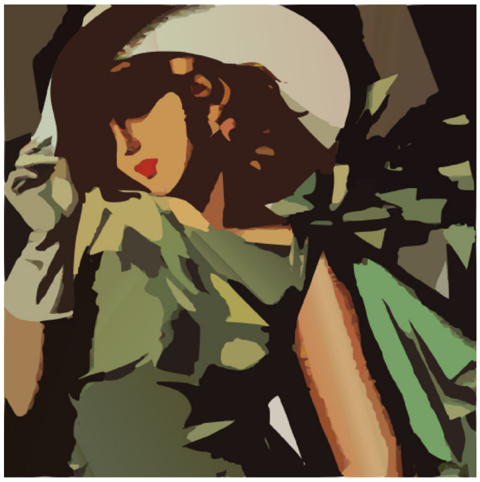
(b)



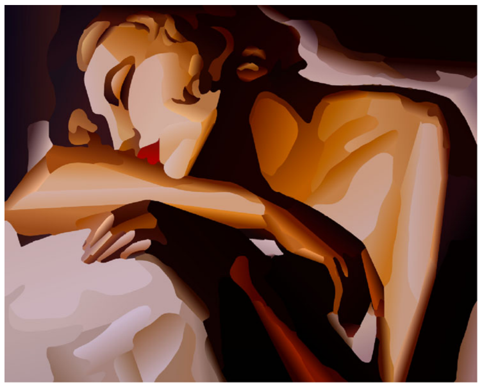
(c)



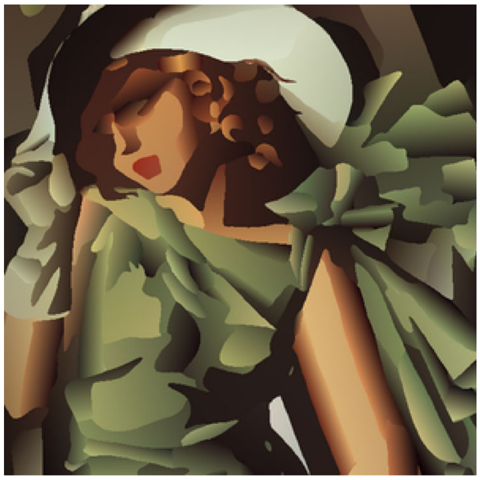
(d)



(e)



(f)



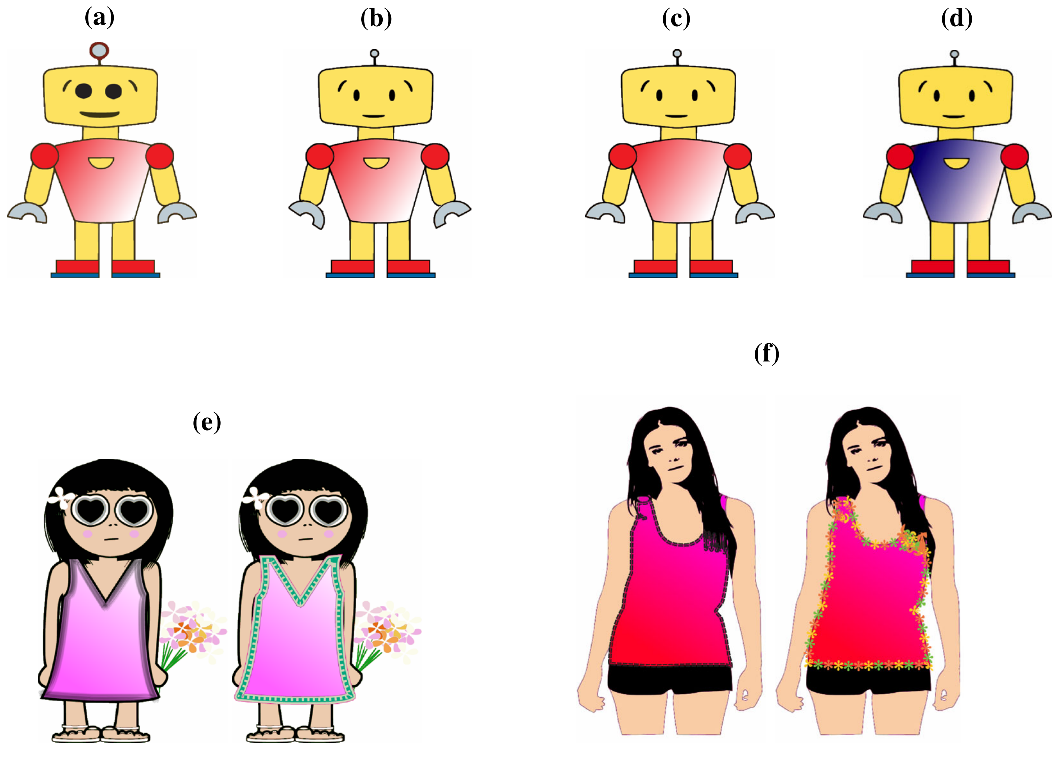


Fig. 12 Editing the final output. **a** Editing the output vector: scaled the body parts, **b** editing the output vector: rotated the arm levers, **c** editing the output vector: removed a path, **d** editing the output vector: the original linear gradient color stops (as shown in Fig. 6 were towards *red to white*. Using Inkscape, we edited the output so that the gradient stops are changed to *blue and white*, **e** editing the output vector: changed the outline style, **f** editing the output vector: changed the outline style

Acknowledgments The images used in Sect. 4.6 have been taken from ARDECO [15] website. Other images have been taken from various open content provider websites. This research was supported in part by the department of Science and Technology, government of India.

References

1. Adobe Systems Inc., Adobe illustrator CS5 (2010)
2. Barla, P., Bousseau, A.: Gradient art: creation and vectorization. In: Rosin, P., Collomosse, J. (eds.) Image and Video Based Artistic Stylization. Springer, New York, Nov 2012
3. Barrett, W.A., Cheney, A.S.: Object-based image editing. In: Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques, ACM, SIGGRAPH '02, pp. 777–784, New York, NY, USA (2002)
4. Bradski, G.: The OpenCV Library. Dr. Dobb's Journal of Software Tools (2000)
5. Canny, J.: A computational approach to edge detection. IEEE Trans. Pattern Anal. Mach. Intell. **8**(6), 679–698 (1986)
6. Dori, D., Liu, W.: Sparse pixel vectorization: an algorithm and its performance evaluation. IEEE Trans. Pattern Anal. Mach. Intell. **21**, 202–215 (1999)
7. Fan, K.-C., Chen, D.-F., Wen, M.-G.: A new vectorization-based approach to the skeletonization of binary images. In: Proceedings of ICDAR, pp. 627–630. IEEE Computer Society Washington, DC, USA (1995)
8. Fischler, M.A., Bolles, R.C.: Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. Commun ACM **24**(6), 381–395 (1981)
9. Gonzalez, R.C., Woods, R.E.: Digital Image Processing, 2nd edn. Addison-Wesley Longman Publishing Co. Inc, Boston (1992)
10. Hori, O., Tanigawa, S.: Raster-to-vector conversion by line fitting based on contours and skeletons. In: Proceedings of the Second International Conference on Document Analysis and Recognition 1993, pp. 353–358, Oct 1993
11. Inkscape. An open source linux/windows scalable vector graphics editor (2010)
12. Jeschke, S., Cline, D., Wonka, P.: Estimating color and texture parameters for vector graphics. Comput. Gr. Forum **30**(2), 523–532 (2011). This paper won the 2nd best paper award at Eurographics 2011
13. Kansal, R., Kumar, S.: A framework for detection of linear gradient filled regions and their reconstruction for vector graphics. In: Proceedings of WSCG'2013, communication papers, pp. 220–229, June 2013
14. Lai, Y.-K., Hu, S.-M., Martin, R.R.: Automatic and topology-preserving gradient mesh generation for image vectorization. ACM Trans. Gr. **28**(3), 85:1–85:8 (2009)
15. Lecot, G., Levy, B.: Ardeco: automatic region detection and conversion. In: Proceedings of Eurographics Symposium on Rendering (2006)
16. Lloyd, S.: Least squares quantization in PCM. IEEE Trans. Inf. Theor. **28**(2), 129–137 (2006)
17. Mammadov, M.A.: A new global optimization algorithm based on dynamical systems approach. In: Proceedings of the 6th International Conference on Optimization: Techniques and Applications (ICOTA' 04), Ballarat, Australia (2004)
18. Mumford, D., Shah, J.: Boundary detection by minimizing functionals. In: Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (1985)

19. Orzan, A., Bousseau, A., Winnemöller, H., Barla, P., Thollot, J., Salesin, D.: Diffusion curves: a vector representation for smooth-shaded images. In: Proceedings of ACM SIGGRAPH 2008 papers, SIGGRAPH '08, ACM, pp. 92:1–92:8. New York, NY, USA (2008)
20. SVG working group. SVG format for vector graphics
21. Selinger, P.: Potrace: a polygon-based tracing algorithm (2003)
22. Sertl, S., Dellnitz, M.: Global optimization using a dynamical systems approach. J. Glob. Optim. **34**(4), 569–587 (2006)
23. Sezgin, M., Sankur, B.: Survey over image thresholding techniques and quantitative performance evaluation. J. Electron. Imaging **13**(1), 146–168 (2004)
24. Stockman, G., Shapiro, L.G.: Computer Vision, 1st edn. Prentice Hall PTR, Upper Saddle River (2001)
25. Sun, J., Liang, L., Wen, F., Shum, H.-Y.: Image vectorization using optimized gradient meshes. In: Proceedings of ACM SIGGRAPH 2007 papers, SIGGRAPH '07, ACM, New York, NY, USA (2007)
26. Tamura, H.: A comparison of line thinning algorithms from digital geometry viewpoint. In: Proceedings of the Fourth International Joint Conference Pattern Recognition, Kyoto, Japan (1978)
27. University of Ballarat. GANSO library for optimization functions
28. Xia, T., Liao, B., Yu, Y.: Patch-based image vectorization with automatic curvilinear feature alignment. In: Proceedings of ACM SIGGRAPH Asia 2009 papers, SIGGRAPH Asia '09, ACM, pp. 115:1–115:10. New York, NY, USA (2009)
29. Zhang, S.-H., Chen, T., Zhang, Y.-F., Martin, R.R.: Vectorizing cartoon animations. IEEE Trans. Vis. Comput. Gr. **15**(4), 618–629 (July 2009)

Ruchin Kansal received his Bachelor of Science from C.C.S, Meerut in 2001 and Masters in Computer Science from M.D. University in 2003. From July 2009 to July 2013, he worked towards his M.S (Research) in the Department of Computer Science and Engineering, Indian Institute of Technology, Delhi. His current research interests include three-dimensional computer graphics, GPGPUs and parallel processing. Currently, he is working as a computer scientist with Adobe Systems Pvt Ltd., Noida (India).

Subodh Kumar is a faculty member in the Department of Computer Science at IIT Delhi. Before this he was a faculty member at Johns Hopkins University. He also spent 2 years at nVIDIA. Prof. Kumar received his bachelor's degree in computer science and engineering from IIT Delhi, and an MS and a Ph.D. in computer science from The University of North Carolina at Chapel Hill. His primary areas of interest include interactive three-dimensional computer graphics, geometry processing, virtual worlds, scientific and medical visualization and parallel programming.