SketchArt: pencil drawing effect for natural photos

Xinxin Zhang
University of Washington
Seattle, WA 98105, USA
zhangx43@uw.edu

ABSTRACT
I built a customizable SketchArt rendering tool to produce pencil drawing effect images from natural images. To simulate pencil drawing and color pencil drawing effects, I produced a line drawing image from natural image and modeled a tone map image based on the features of tone distribution. They are combined together to make the eventual output image visually alike an artist' sketch.

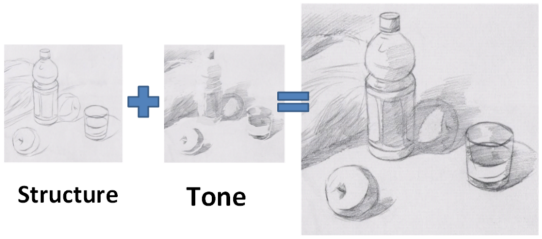


Figure 1: pencil sketch combines tone and structure

Author Keywords
Digital image processing, Pencil sketch, Non-Photo-Realistic Rendering, Tonal drawing, Texture

INTRODUCTION
My SketchArt tool takes a natural image as input and outputs a sketch that exhibits the pencil or color pencil drawing style.

Since the sketch is defined as a rapidly executed freehand drawing to depict the global structure and main contours, the lines are across and not continuous. Inspired by the idea of generating pencil stroke in a "scribbled" way [4] and iterative directional difference-of-Gaussian [1], I applied convolution approach to stimulate sketchy lines, which appears very often in artists' work.

Beside of the line drawing effect simulation, another major problem for producing artistic sketch is to express the luminance, shadow and shading properly. The method of representing the tone features of sketch with natural image tones by [2] could lead to hatching patterns and not consistent with the sketch. Thus, by analyzing the features of tone distribution among artists' work, I finally implemented a tone map by modeling three major layers of tone with different statistical distribution [3].

Generating suitable pencil textures for images is complex. The patterns of pencil texture can be seen as the pencil sketch patterns without obvious direction. In human drawing, by repeatedly drawing at the same place, we can output a texture alike image and darken the tone of it. Therefore, to produce an appropriate texture image, I applied gamma correction to an downloaded pencil texture image repeatedly to find the best one.

The SketchArt tool is an interactive system, which provides a lot of customizable setup to enable users manually generate and enhance the artistic effect.

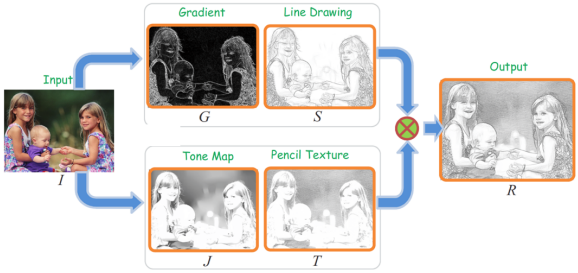


Figure 2: Overview of pencil drawing process

METHODOLOGY
The overview of the process is illustrated in Fig. 2.

To get the gradient image (G). I first used median filter to smooth the image and then used Sobel operator to detect the edges of the image and compute the gradient image following formula (1).

$$G = ((\partial_x I)^2 + (\partial_y I)^2)^{\frac{1}{2}} \quad (1)$$

$$C_i(p) = \begin{cases} G(p) & \text{if } \operatorname{argmax}_i\{\psi_i \otimes G\}(p) = i \\ 0 & \text{otherwise} \end{cases} \quad (2)$$

$$S = \sum_{i=1}^{8} \psi_i \otimes C_i \quad (3)$$

After obtaining the gradient image (G), I began to find an optimal line direction at each pixel. I first generated 8 direction line segments at 45 degree apart, each of them is denoted as a convolution kernel. Then, I convoluted kernel of each direction with each pixel of G and found the maximum response to obtain the magnitude map (C) of the image according to the formula (2). Finally, I convoluted 8 directions line segments with corresponding magnitude map at each direction and summed up the responses to compute the line drawing image (S) following formula (3). Fig.3 shows the process of this step.

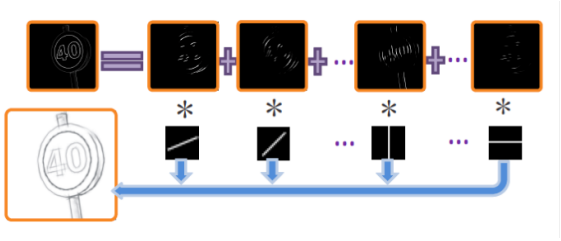


Figure 3: Line drawing with convolution

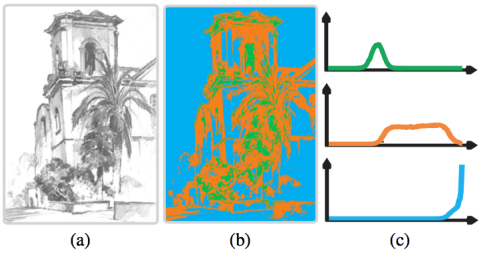


Figure 4: Tone distribution of different layers of sketch

To make sure rendering image is more like the real pencil sketch, I modeled tone distribution with Laplacian distribution for the bright layer, uniform distribution for mid-tone layer and normal distribution for the darker layer. Formula (4), (5), (6) shows the formulas of distribution. In Fig. 3 the green curve represents the distribution of darker layer, the orange curve represents the distribution of mid-tone layer and blue curve represents the bright layer. Fig. 4 shows the distribution from a human pencil sketch [3]. The parameters in the formulas of distribution have the closed-form representation [3] written in formula (7).

$$p_1(v) = \begin{cases} \frac{1}{\sigma_b} e^{-\frac{1-v}{\sigma_b}} & \text{if } v \le 1 \\ 0 & \text{otherwise} \end{cases} \quad (4)$$

$$p_2(v) = \begin{cases} \frac{1}{u_b - u_a} & \text{if } u_a \le v \le u_b \\ 0 & \text{otherwise} \end{cases} \quad (5)$$

$$p_3(v) = \frac{1}{\sqrt{2\pi}\sigma_d} e^{-\frac{(v-\mu_d)^2}{2\sigma_d^2}}$$

$$\sigma_b = \frac{1}{N}\sum_{i=1}^{N} |x_i - 1|, \quad u_a = m_m - \sqrt{3}s_m, \quad u_b = m_m + \sqrt{3}s_m, \quad (6)$$
$$\mu_d = m_d, \quad \sigma_d = s_d,$$

ω_1	ω_2	ω_3	σ_b	u_a	u_b	μ_d	σ_d
11	37	52	9	105	225	90	11

Figure 5: learned parameters

To have a proper pencil texture image (T), I applied gamma correction to a downloaded pencil texture image (Fig. 6) repeatedly to find the best one. Fig. 7 shows several pencil texture images after different times of gamma correction. To figure out the specific number of doing gamma correction, I compute β based on formula (8). T can be computed with formula (9). Larger value of gamma results in a darker output image.

Figure 6: Pencil texture image

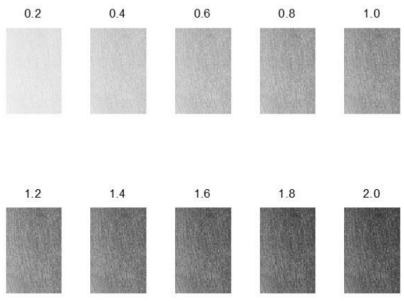


Figure 7: Pencil texture image after gamma correction

$$\beta^* = \arg\min_{\beta} \|\beta \ln H - \ln J\|_2^2 + \lambda \|\nabla \beta\|_2^2, \quad (8)$$

$$T = H^{\beta^*}. \quad (9)$$

$$R = S \cdot T. \quad (10)$$

The last step of the drawing process is to simply multiply S and T as formula (10) to compute the rendering image.

Figure 10: origial image of bicycle kick

RESULTS

Artistic effect images

Here are some original and rendering images after applying the pencil effect drawing technique.

(a) coffee

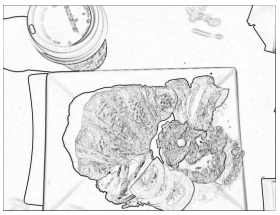
(b) breakfast

Figure 8: original images

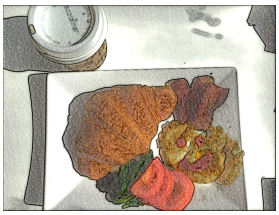
(a) pencil effect

(b) color pencil effect

Figure 11: artistic images of coffee



(a) pencil effect



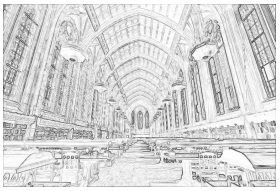
(b) color pencil effect

Figure 12: artistic images of breakfast

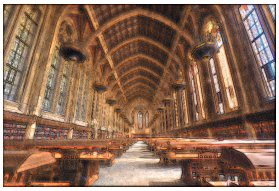
(a) suzzallo

(b) The Beatles

Figure 9: original images

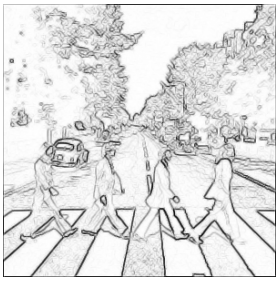


(a) pencil effect

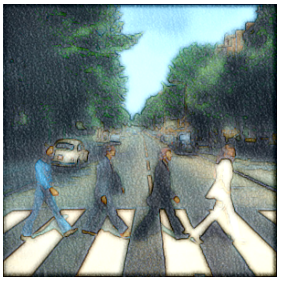


(b) color pencil effect

Figure 13: artistic images of Suzzallo library

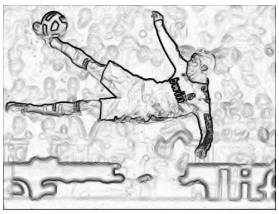


(a) pencil effect



(b) color pencil effect

Figure 14: artistic images of The Beatles



(a) pencil effect

(b) color pencil effect

Figure 15: artistic images of bicycle kick

CONCLUSION AND FUTURE WORK

In this project, the artistic effect images above validate the approach that is used to simulate and optimize the pencil drawing effect is robust. Decomposing a sketch into three major parts: line structures, tone distribution and pencil texture, capturing the features of each part, teasing out the proper method to best reproduce the pencil drawing effect and enhancing the output by adjusting the luminance and pencil texture image.

To evaluate the technique of generating eight equal-angle apart line segments, I also customized the degree of the angle to be different and changed the number of kernels to see the consequent performance. The results showed that reducing number of line segments negatively affect the pencil drawing effect since the curves of the output are smoother. But setting disparate degrees between angles had no big influence on the rendering performance, which is understandable that pencil stroke varies for different artists.

Although this system renders images with impressive sketching effect, it still has some limitations on dealing with those images with ambiguous background. For images like that, this system is not able to portrait with proper shadowing and sketching as an artist does. I hope to address this problem in the future work.

Despite that, I think some new features can be added based on what I learn from this class and what I research on image processing area. It could be more fun if this technique can be applied in a real-time system, such as producing a real-time rendering telescope to transform the natural views to sketch. The rendering images generated with this system is realistic, I think it is also possible to generate some impressionistic or cartoon-like images. These tasks are challenging but intriguing for me to explore in the future.

APPENDIX

Customizable GUI and Tutorial

1. Run main.c in Matlab then the following GUI will pop up.

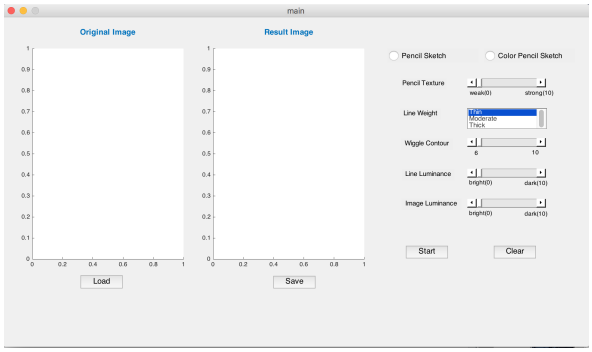


Figure 16: main GUI of SketchArt project

2. Click **Load** button to choose an image in computer to load and display on both **Original Image** and **Result Image** Panels.

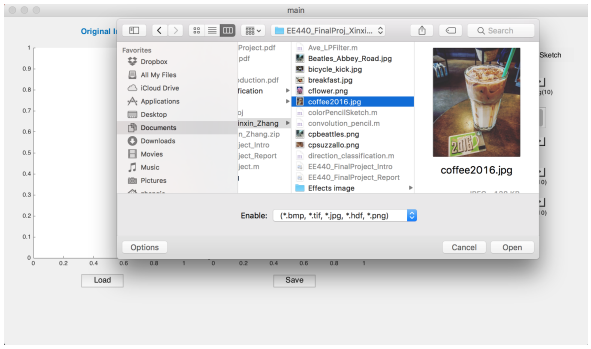


Figure 17: click **Load** and choose image

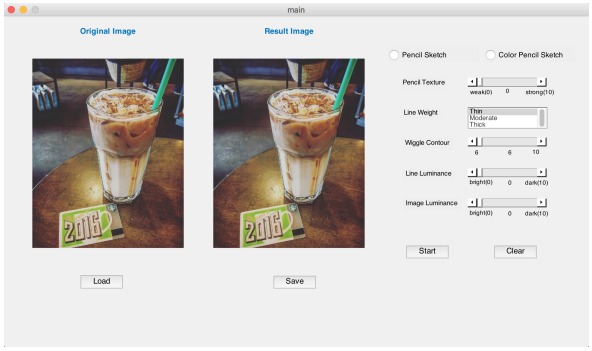


Figure 18: GUI before rendering

3. Adjust the settings of parameters on the right side of the window and then press **Start** button to obtain the rendering image on the **Result Image** Panel.

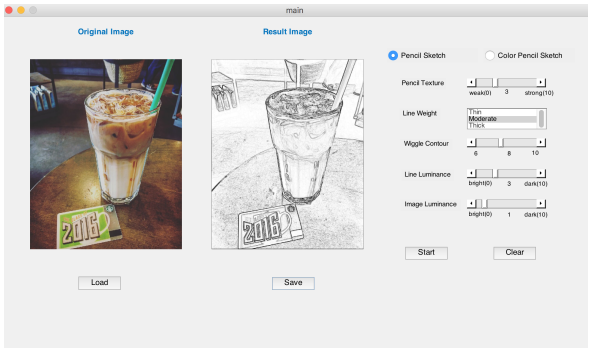


Figure 19: pencil effect

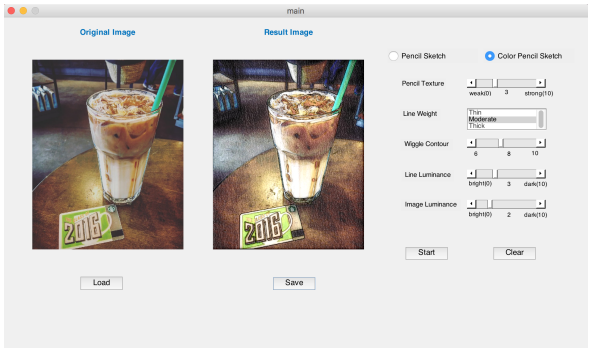


Figure 20: color pencil effect

4. Press **Save** button to save the image in computer. The saving name, format and location of the image are all customized.

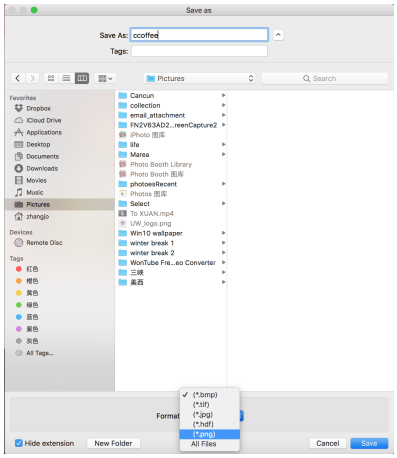


Figure 21: save the rendering image

5. Press **Clear** button to reset all parameters and to convert the rendering image on **Result Image Panel** to the original image.

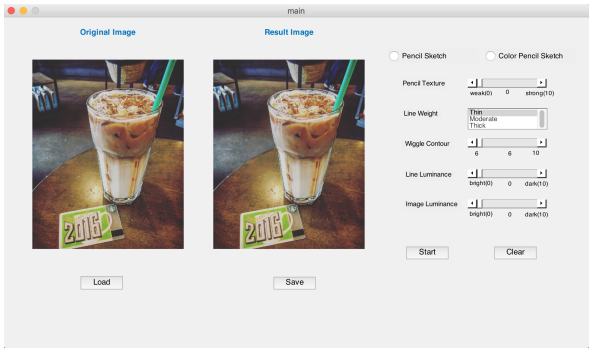


Figure 22: GUI after clicking clear

Codes

Original: *sobel_operator.m*

```
% Function SOBEL_OPERATOR is used to output the edge of
% given image.
% ------------------------------------------------------
% Parameter:
%      Input:
%           im: data of input image
%
%      Output:
%           Y : edge iamge
% ------------------------------------------------------
% Usage:
%      e.g.
%      Y = sobel_operator(imread('imagename'))
% ------------------------------------------------------
% Author: Xinxin Zhang
% Date: 12/04/2016
% ====================================================== %

function Y = sobel_operator(im)

    img = im2double(im);

    % Horizontal and vertical Sobel operators
    hor_oper = [-1 0 1; -2 0 2; -1 0 1];
    ver_oper = [-1 -2 -1; 0 0 0; 1 2 1];

    Gx=abs(conv2(img, ver_oper, 'same'));
    Gy=abs(conv2(img, hor_oper, 'same'));
    Y = sqrt(Gx.^2 + Gy.^2);

end
```

Original: *medianFilter.m*

```
% Function MEDIANFILTER smooths and denoises the image
% ------------------------------------------------------
% Parameter:
%       input:
%             I: input image
%             W: size of the sliding window(filter size)
%                usually a square
%       output:
%             Y: output image after smoothing
% ------------------------------------------------------
% Usage:
%      img = imread('imagename');
%      Y = medianFilter (img, 3);
% ------------------------------------------------------
% Note:
```

```
%
% ------------------------------------------------------
% Author: Xinxin Zhang
% Date: 10/23/2016
% ====================================================== %

function [Y] = medianFilter(I, W)

    % get a dimension of pixel values
    I1 = I(:, :, 1);

    % repeat four boundaries
    topRow = I1(1:((W-1)/2), :);
    bottomRow = I1(end-(((W-1)/2))+1:end, :);

    I_noise = [topRow; I1];
    I_noise = [I_noise; bottomRow];

    topColumn = I_noise(:,1:((W-1)/2));
    bottomColumn = I_noise(:, end-(((W-1)/2))+1:end);

    I_noise = [topColumn, I_noise];
    I_noise = [I_noise, bottomColumn];

    I_Denoise = zeros(size(I_noise));

    % replace the value in each pixel with the median
         value within in sliding window
    for i = 1:size(I1, 1)
        for j = 1:size(I1, 2)
            I_Denoise(i+1, j+1) =
                 median(median(I_noise(i:(i+(W-1)),
                 j:(j+(W-1)))));
        end
    end

    % denoise the image
    I_Denoise = I_Denoise(2:(end-1), 2:(end-1));
    Y(:, :, 1) = I_Denoise;

end
```

Original: *Ave_LPFilter.m*

```
% Function AVE_LPFILTER smooths and denoises the image
% ------------------------------------------------------
% Parameter:
%       input:
%            I: input image
%            W: size of the LPF, usually a square
%       output:
%            Y: output image
% ------------------------------------------------------
% Usage:
%      e.g.
%      Y = Ave_LPFilter(img, 10)
% ------------------------------------------------------
% Note:
%
% ------------------------------------------------------
% Author: Xinxin Zhang
% Date: 10/24/2016
% ====================================================== %

function [Y] = Ave_LPFilter(I, W)

    % generate a LPF
    LPF = (1 / W^2) * ones(W);

    I1 = conv2(double(I(:, :, 1)), LPF, 'same');
    Y(:, :, 1) = I1;
    Y(:, :, 2) = I1;
    Y(:, :, 3) = I1;

end
```

Original: *direction_classification.m*

```
% Function DIRECTION_CLASSIFICATION is used to compute
     the index and angle representation of the line
     segment.
% ------------------------------------------------------
% Parameter:
%      Input:
%           img    : input image [1D, 'double' type]
%           ks     : convolutional kernal size [1, 2,
     3...]
%           dirNum : numbers of direction segments [8]
%      Output:
%           Dirs   : direction angle of pixels
%           cindex : index of segment direction
% ------------------------------------------------------
% Usage:
%      e.g.
%      [Dirs, cindex] = direction_classification(img, 1,
     8)
% ------------------------------------------------------
% Note:
%      e.g. Define 8 line segement direction:
%      cindex[1 2 3 4 5 6 7 8] --> Dirs[0 22.5 45 67.5 90
     112.5 135 157.5]
%      Thus, Dirs[22.5 22.5 22.5] --> cindex[2 2 2]
% ------------------------------------------------------
% Author: Xinxin Zhang
% Date: 12/04/2016
% ====================================================== %

function [Dirs, cindex] = direction_classification(img,
     ks, dirNum)

    imgEdge = sobel_operator(img); % smooth the image
    kerRef = zeros(ks*2+1);
    kerRef(ks+1, :) = 1;

    % convolute direction line-segement with the input
         edge image
    response = response_img(img, imgEdge, kerRef, dirNum);

    % compute the direction angle
    [~, index] = sort(response, 3);
    Dirs = (index(:, :, end)-1) * 180 / dirNum;

    % returns the largest response along 3 dimensions
    [~, cindex] = max(response, [], 3);

end
```

Original: *response_img.m*

```
% Function RESPONSE\_IMG is used to find the response
     gradient magnitude image on each segment direction.
% ------------------------------------------------------
% Parameter:
%      Input:
%           img    : input image [1D, 'double' type]
%           map    : edge image
%           kerRef : kernel reference
%           dirNum : numbers of direction segments [8]
%      Output:
%           outMap : ouput image after convolution
```

```
% ------------------------------------------------------
% Usage:
%      e.g. outMap = response_img(img, edgeImage, kerRef,
     8)
% ------------------------------------------------------
% Note:
%      line segments can be customizedly defined [2...8],
%      the larger number leads to the better output
% ------------------------------------------------------
% Author: Xinxin Zhang
% Date: 12/04/2016
% ====================================================== %

function outMap = response_img(img, map, kerRef, dirNum)

    % img = imread(X);

    % classification
    [H, W, sc] = size(img);
    outMap = zeros(H, W, dirNum);

    for li = 1 : dirNum

        % generate line segments with certain direction
             from 0 to 180 degree
        % the angle is uniformly divided into dirNum times
        ker = imrotate(kerRef, (li-1)*180/dirNum,
             'bilinear', 'crop');

        % convolve the kernel with the each pixel of
             gradient map
        outMap(:, :, li) = conv2(map, ker, 'same');

    end
end
```

Original: *sobel_operator.m*

```
% Function LINEDRAWING is used to generate line-drawing
     image.
% ------------------------------------------------------
% Parameter:
%      Input:
%           img   : input image [1D, 'double' type]
%           ks    : convolutional kernal size [1, 2, 3...]
%           w     : width of the stroke
%           dirNum : numbers of direction segments [8]
%      Output:
%           S     : output image
% ------------------------------------------------------
% Usage:
%      e.g.
%      S = lineDrawing(img, 1, 8);
% ------------------------------------------------------
% Note:
%      For better performance:
%      ks should be small, 1, 2, and 3 are smart choices
%      w should be no larger than 3 for better smoothing
     effect
% ------------------------------------------------------
% Author: Xinxin Zhang
% Date: 12/04/2016
% ====================================================== %

function S = lineDrawing(img, ks, w, dirNum)

    % close all; clear all; clc

    % X = 'suzzallo.png'; % name of the image
    % im = imread(X);     % load image
    %
    % [ly, lx, sc] = size(im);
    %
    % if sc == 3
    %     img = rgb2gray(im); % convert rgb to grayscale
    %     img = img(:,:,1);
    % else
    %     img = im;
    % end

    wsize = 3;              % lowpass filter window size
    img = medianFilter(img, wsize);  % smooth image
    imgEdge = sobel_operator(img); % find the edge image
         of original image

   % get the length, width and color dimension of image
    [ly, lx, sc] = size(img);

    % find the optimal direction of sketch at each pixel
         and form a direction map
    [Dirs, cindex] = direction_classification(img, ks, 8);

    % create the magnitude map of direction i
    cMap = zeros(ly, lx, dirNum);
    for li = 1 : dirNum
        cMap(:, :, li) = imgEdge .* (cindex == li);
    end

    kerRef = zeros(ks*2+1);
    kerRef(ks+1, :) = 1;

    for i = 1 : w
        if (ks+1-i) > 0
            kerRef(ks+1-i, :) = 1;
        end

        if (ks+1+i) < (ks*2+1)
            kerRef(ks+1+i, :) = 1;
        end
    end

    % generate lines at each pixels
    S_prime_i = zeros(ly, lx, dirNum);

    for li = 1 : dirNum
        ker = imrotate(kerRef, (li-1)*180/dirNum,
             'bilinear', 'crop');
        S_prime_i(:, :, li) = conv2(cMap(:, :, li), ker,
             'same');
    end

    S_prime = sum(S_prime_i, 3);

    % map to [0, 1]
    S_prime = (S_prime - min(S_prime(:))) /
         (max(S_prime(:)) - min(S_prime(:)));

    % invert pixle values to get final pencil stroke map S
    S = 1 - S_prime;

    % show line drawing of image
    % imshow(S);

end
```

Original: *toneDrawing.m*

```
% Function TONEDRAWING is used to generate tone drawing
     image.
% ------------------------------------------------------
% Parameter:
%      Input:
```

```
%           img   : input image [1D, 'double' type]
%           weightRat : weight ratio of three tone layers
     [1, 2, 3]
%      Output:
%           J     : output tone image
% -------------------------------------------------------
% Usage:
%      e.g.
%      J = toneDrawing(img, 1);
% -------------------------------------------------------
% Note:
%
% -------------------------------------------------------
% Author: Xinxin Zhang
% Date: 12/04/2016
% ======================================================= %

function J = toneDrawing(img, weightRat)

    % close all; clear all; clc
    % X = 'flower.png';
    img = im2double(img);

    % parameter
    ua = 105;
    ub = 225;

    sigmab = 9;
    sigmad=11;

    mud = 90;

    % choose the weights
    if weightRat==1
        % ratio 11:37:52
        omega1 = 11; omega2 = 37; omega3 = 52;
    elseif weightRat==2
        % ratio 42:29:29
        omega1 = 42; omega2 = 29; omega3 = 29;
    else
        % ratio 2:76:22
        omega1 = 2; omega2 = 22; omega3 = 76;
    end

    % generate tone map
    tmap = zeros(1, 256);
    tot = 0;
    for v = 0:255

        % l1 bright layer: Laplacian distribution
        l1 = (1/sigmab)*exp(-(255-v)/sigmab);

        % l2 mid tone layer: Uniform distribution
        if v>=ua && v<=ub
            l2 = 1/(ub-ua);
        else
            l2 = 0;
        end

        % l3 dark tone layer: Normal distribution
        l3 = (1/sqrt(2*pi*sigmad)) *
             exp(-(v-mud)^2/(2*sigmad^2))*0.01;

        % l sum up three layers
        tmap(v+1) = omega1*l1 + omega2*l2 + omega3*l3;
        tot = tot + tmap(v+1);
    end

    % normalization
    tmap = tmap/tot;

    % let the image histogram matches the intended
         histogram
    J = histeq(img, tmap);
    % smooth the image
    J = Ave_LPFilter(J, 10);

    % imshow(J);

end
```

Original: *colorPencilSketch.m*

```
% Function COLORPENCILSKETCH is used to simulate the
     color pencil sketch effect on image.
% -------------------------------------------------------
% Parameter:
%      Input:
%           inImg    : input image [any type]
%           ks       : convolutional kernal size [1, 2,
     3...]
%           theta    : parameter to adjust the pencil
     texture intensity
%                      (larger theta means larger
     intensity) [0.1~]
%           weightRat : weight ratio of three tone layers
     [1, 2, 3]
%           w        : width of the strocke
%           dirNum   : numbers of direction segments [8]
%           betaS    : the darkness of the stroke [1, 2,
     3...]
%           betaI    : the darkness of the resulted image
     [1, 2, 3...]
%      Output:
%           outImg   : output image
% -------------------------------------------------------
% Usage:
%      e.g.
%      im = imread('nameofimage');
%      I = colorPencilSketch(im, 1, 0.2, 8, 3, 2, 2)
% -------------------------------------------------------
% Note: For better performance:
%      ks should be small, 1, 2, and 3 are smart choice
%      w should be no larger than 3 for better smoothing
     effect
% -------------------------------------------------------
% Author: Xinxin Zhang
% Date: 12/04/2016
% ======================================================= %

function outImg = colorPencilSketch(inImg, ks, theta, w,
     dirNum, weightRat, betaS, betaI)

    % inImg = imread('suzzallo.png');
    inImg = im2double(inImg);  % convert image type to
         'double'
    [len, wid, sc] = size(inImg); % get image length,
         width, and dimension

    % get the luminance information of image convert
         color image to grayscale if necessary
    % Y = Y
    % U = 0.872021 Cb
    % V = 1.229951 Cr

    % Because I only care the Y(lumina), so instead to
         convert RGB to YUV, I simply convert RGB to YCbCr
    if (sc == 3)
        imgLumina = rgb2ycbcr(inImg);
        img = imgLumina(:,:,1);
    else
        img = inImg;
```

```
    end

    %% generate line-drawing image (stroke map)
    S = lineDrawing(img, ks, w, dirNum) .^betaS; % darken
        the result by gammaS
    % figure, imshow(S);

    %% generate tone map
    J = toneDrawing(img, weightRat).^betaI ; % darken the
        result by gammaI
    % figure, imshow(J);

    %% generte pencil texture image
    P = im2double(rgb2gray(imread('tonalTexture.png')));
        % load pencil texture image
    T = pencilTexture(img, P, J, theta).^ betaI;
    % figure, imshow(T);

    %% compute output image
    img = S .* T;

    if (sc == 3)
        imgLumina(:,:,1) = img;
        I = ycbcr2rgb(imgLumina);
    else
        I = img;
    end

    outImg = I;

    imshow(outImg);

end
```

Original: *pencilSketch.m*

```
% Function PENCILSKETCH is used to simulate the pencil
    sketch effect on image.
% ------------------------------------------------------
% Parameter:
%      Input:
%           inImg    : input image [any type]
%           ks       : convolutional kernal size [1, 2,
    3...]
%           dirNum   : numbers of direction segments [8]
%           w        : width of the strocke
%           betaS    : the darkness of the stroke [1, 2,
    3...]
%      Output:
%           outImg   : output image
% ------------------------------------------------------
% Usage:
%      e.g.
%      im = imread('nameofimage');
%      I = pencilSketch(im, 1, 8, 2);
% ------------------------------------------------------
% Note:
%      For better performance:
%      ks should be small, so 1, 2, and 3 are smart
    choices
%      w should be no larger than 3 for better smoothing
    effect
% ------------------------------------------------------
% Author: Xinxin Zhang
% Date: 12/04/2016
% Reference: "Combining Sketch and Tone for Pencil
    Drawing Production"
% Cewu Lu, Li Xu, Jiaya Jia (NPAR 2012), June, 2012

% ====================================================== %

function outImg = pencilSketch(inImg, ks, w, dirNum,
    betaS)

    % inImg = imread('suzzallo.png');
    inImg = im2double(inImg); % convert image type to
        'double'
    [len, wid, sc] = size(inImg); % get image length,
        width, and dimension

    % get the luminance information of image, convert
        color image to grayscale if necessary
    % Y = Y
    % U = 0.872021 Cb
    % V = 1.229951 Cr

    % Because I only care the Y(lumina), so instead to
        convert RGB to YUV, I simply convert RGB to YCbCr

    if (sc == 3)
        imgLumina = rgb2ycbcr(inImg);
        img = imgLumina(:,:,1);
    else
        img = inImg;
    end

    %% generate line-drawing image (stroke map)

    outImg = lineDrawing(img, ks, w, dirNum).^betaS; %
        darken the result by gammaS
    % figure, imshow(S);

end
```

Original: *pencilTexture.m*

```
% Function PENCILTEXTURE is used to generate pencil
    texture map and apply it on the tone map.
% ------------------------------------------------------
% Parameter:
%         Input:
%               P: tonnal texture image
%               J: tone map of original image
%               theta: parameter to adjust the pencil
    texture intensity
%                    (larger theta means larger
    intensity)
%         Output:
%               T: pencil-texture image
% ------------------------------------------------------
% Usage:
%      e.g.
%      J = pencilTexture(P, J, 0.2);
% ------------------------------------------------------
% Note:
%     A larger theta will result in a larger intensity.
% ------------------------------------------------------
% Author: Xinxin Zhang
% Date: 12/04/2016
% ====================================================== %

function T = pencilTexture(img, P, J, theta)

    % im = imread('flower.png');
    % P = im2double(rgb2gray(imread('tonalTexture.png')));
    % P = imresize(P, [len wid]);

    J = toneDrawing(img, 1);
    J = rgb2gray(J);
    [len, wid, ~] = size(img);

    % Initialization
```

```
    P = imresize(P, [len, wid]);
    P = reshape(P, len*wid, 1);
    logP = log(P);
    logP = spdiags(logP, 0, len*wid, len*wid);

    J = imresize(J, [len, wid]);
    J = reshape(J, len*wid, 1);
    logJ = log(J);

    e = ones(len*wid, 1);
    Dx = spdiags([-e, e], [0, len], len*wid, len*wid);
    Dy = spdiags([-e, e], [0, 1], len*wid, len*wid);

    % Compute matrix A and b
    A = theta * (Dx * Dx' + Dy * Dy') + (logP)' * logP;
    b = (logP)' * logJ;

    % Conjugate gradient
    beta = pcg(A, b, 1e-6, 60);

    % Compute the result
    beta = reshape(beta, len, wid);

    P = reshape(P, len, wid);

    T = P.^beta;

    % figure;
    % imshow(T);

end
```

Original: *main.m*

```
% main GUI of Sketch Art project
% Author: Xinxin Zhang
% Date: 12/07/2016

function varargout = main(varargin)
% MAIN MATLAB code for main.fig
%      MAIN, by itself, creates a new MAIN or raises the
     existing
%      singleton*.
%
%      H = MAIN returns the handle to a new MAIN or the
     handle to
%      the existing singleton*.
%
%      MAIN('CALLBACK',hObject,eventData,handles,...)
     calls the local
%      function named CALLBACK in MAIN.M with the given
     input arguments.
%
%      MAIN('Property','Value',...) creates a new MAIN or
     raises the
%      existing singleton*. Starting from the left,
     property value pairs are
%      applied to the GUI before main_OpeningFcn gets
     called. An
%      unrecognized property name or invalid value makes
     property application
%      stop. All inputs are passed to main_OpeningFcn via
     varargin.
%
%      *See GUI Options on GUIDE's Tools menu. Choose "GUI
     allows only one
%      instance to run (singleton)".
%
% See also: GUIDE, GUIDATA, GUIHANDLES

% Edit the above text to modify the response to help main

% Last Modified by GUIDE v2.5 07-Dec-2016 09:26:04

% Begin initialization code - DO NOT EDIT
gui_Singleton = 1;
gui_State = struct('gui_Name',  mfilename, ...
                   'gui_Singleton', gui_Singleton, ...
                   'gui_OpeningFcn', @main_OpeningFcn, ...
                   'gui_OutputFcn', @main_OutputFcn, ...
                   'gui_LayoutFcn', [] , ...
                   'gui_Callback', []);
if nargin && ischar(varargin{1})
    gui_State.gui_Callback = str2func(varargin{1});
end

if nargout
    [varargout{1:nargout}] = gui_mainfcn(gui_State,
        varargin{:});
else
    gui_mainfcn(gui_State, varargin{:});
end
% End initialization code - DO NOT EDIT


% --- Executes just before main is made visible.
function main_OpeningFcn(hObject, eventdata, handles,
     varargin)
% This function has no output args, see OutputFcn.
% hObject  handle to figure
% eventdata reserved - to be defined in a future version
     of MATLAB
% handles  structure with handles and user data (see
     GUIDATA)
% varargin command line arguments to main (see VARARGIN)

% Choose default command line output for main
handles.output = hObject;
clc;
clear global;

% Update handles structure
guidata(hObject, handles);

% UIWAIT makes main wait for user response (see UIRESUME)
% uiwait(handles.figure1);

% --- Outputs from this function are returned to the
     command line.
function varargout = main_OutputFcn(hObject, eventdata,
     handles)
% varargout cell array for returning output args (see
     VARARGOUT);
% hObject  handle to figure
% eventdata reserved - to be defined in a future version
     of MATLAB
% handles  structure with handles and user data (see
     GUIDATA)

% Get default command line output from handles structure
varargout{1} = handles.output;

% --- Executes on button press in Load.
function Load_Callback(hObject, eventdata, handles)
% hObject  handle to Load (see GCBO)
% eventdata reserved - to be defined in a future version
     of MATLAB
% handles  structure with handles and user data (see
     GUIDATA)

global X;
global Y;
global RP;
```

```
global FullPathName;

% open an image and show it on both original and result
    panel
[FileName,PathName] =
    uigetfile('*.bmp;*.tif;*.jpg;*.hdf;*.png','Select
    the image file');
FullPathName = [PathName,FileName];
X = imread(FullPathName);
imshow(X, 'Parent', handles.originalPanel);

RP = findobj(gcf,'Tag','resultPanel'); % resultPanel is
    the name of Tag
set(gcf, 'CurrentAxes', RP); % set current working panel
    as resultPanel
imshow(X); % show inedited image on resultPanel

% --- Executes on button press in Save.
function Save_Callback(hObject, eventdata, handles)
% hObject  handle to Save (see GCBO)
% eventdata reserved - to be defined in a future version
    of MATLAB
% handles  structure with handles and user data (see
    GUIDATA)

global Y;

[FileName,PathName] =
    uiputfile({'*.bmp';'*.tif';'*.jpg';'*.hdf';'*.png'},'Save
    as');
FullPathName = [PathName,FileName];
imwrite(Y,FullPathName);

% check the existence of the file and displays the result
    of the file selection operation.
if isequal(FileName,0) || isequal(PathName,0)
   disp('User selected Cancel')
else
   disp(['User selected ',fullfile(PathName,FileName)])
end

% --- Executes on button press in pSketch.
function pSketch_Callback(hObject, eventdata, handles)
% hObject  handle to pSketch (see GCBO)
% eventdata reserved - to be defined in a future version
    of MATLAB
% handles  structure with handles and user data (see
    GUIDATA)

global cp;
global p;

if (get(hObject,'Value') == get(hObject,'Max'))
%    str = 'Yes';
    p = 1; cp = 0;
    display('Pencil Sketch Selected');
    set(handles.cpSketch,'value',0);
else
    p = 0;
    display('Not selected');
%    str = 'No';
end

% --- Executes on button press in cpSketch.
function cpSketch_Callback(hObject, eventdata, handles)
% hObject  handle to cpSketch (see GCBO)
% eventdata reserved - to be defined in a future version
    of MATLAB
% handles  structure with handles and user data (see
    GUIDATA)

global cp;
global p;

if (get(hObject,'Value') == get(hObject,'Max'))
%    str = 'Yes';
    cp = 1; p = 0;
    display('Color Pencil Sketch Selected');
    set(handles.pSketch,'value',0);
else
    display('Not selected');
    cp = 0;
%    str = 'No';
end

% --- Executes on slider movement.
function slider1_Callback(hObject, eventdata, handles)
% hObject  handle to slider1 (see GCBO)
% eventdata reserved - to be defined in a future version
    of MATLAB
% handles  structure with handles and user data (see
    GUIDATA)

% Hints: get(hObject,'Value') returns position of slider
%       get(hObject,'Min') and get(hObject,'Max') to
    determine range of slider

% global originalPanel;
% global resultPanel;
% global Y;
% global FullPathName;

global theta;

theta = get(hObject, 'Value');
assignin('base', 'sliderVal', theta);
set(handles.texture1, 'String', theta);

% --- Executes during object creation, after setting all
    properties.
function slider1_CreateFcn(hObject, eventdata, handles)
% hObject  handle to slider1 (see GCBO)
% eventdata reserved - to be defined in a future version
    of MATLAB
% handles  empty - handles not created until after all
    CreateFcns called

% Hint: slider controls usually have a light gray
    background.
if isequal(get(hObject,'BackgroundColor'),
    get(0,'defaultUicontrolBackgroundColor'))
   set(hObject,'BackgroundColor',[.9 .9 .9]);
end

% --- Executes on slider movement.
function slider2_Callback(hObject, eventdata, handles)
% hObject  handle to slider2 (see GCBO)
% eventdata reserved - to be defined in a future version
    of MATLAB
% handles  structure with handles and user data (see
    GUIDATA)

% Hints: get(hObject,'Value') returns position of slider
%       get(hObject,'Min') and get(hObject,'Max') to
    determine range of slider

global dirNum;

currentValue = get(hObject, 'Value');
dirNum = round(currentValue);
assignin('base', 'sliderVal', dirNum);
set(handles.textWigCon, 'String', dirNum);
```

```
% --- Executes during object creation, after setting all
    properties.
function slider2_CreateFcn(hObject, eventdata, handles)
% hObject  handle to slider2 (see GCBO)
% eventdata reserved - to be defined in a future version
    of MATLAB
% handles  empty - handles not created until after all
    CreateFcns called

% Hint: slider controls usually have a light gray
    background.
if isequal(get(hObject,'BackgroundColor'),
    get(0,'defaultUicontrolBackgroundColor'))
   set(hObject,'BackgroundColor',[.9 .9 .9]);
end

% --- Executes on selection change in listbox1.
function listbox1_Callback(hObject, eventdata, handles)
% hObject  handle to listbox1 (see GCBO)
% eventdata reserved - to be defined in a future version
    of MATLAB
% handles  structure with handles and user data (see
    GUIDATA)

% Hints: contents = cellstr(get(hObject,'String'))
    returns listbox1 contents as cell array
%       contents{get(hObject,'Value')} returns selected
    item from listbox1

global w;
% set(listbox_handle, 'Value', [] );
items = get(hObject, 'String');
index_selected = get(hObject,'Value');
w = index_selected-1;
item_selected = items{index_selected};
display(item_selected);
display(index_selected);

% --- Executes during object creation, after setting all
    properties.
function listbox1_CreateFcn(hObject, eventdata, handles)
% hObject  handle to listbox1 (see GCBO)
% eventdata reserved - to be defined in a future version
    of MATLAB
% handles  empty - handles not created until after all
    CreateFcns called

% Hint: listbox controls usually have a white background
    on Windows.
%      See ISPC and COMPUTER.

if ispc && isequal(get(hObject,'BackgroundColor'),
    get(0,'defaultUicontrolBackgroundColor'))
   set(hObject,'BackgroundColor','white');
end

set(hObject,'String',{'Thin';'Moderate';'Thick'});

% --- Executes on slider movement.
function slider3_Callback(hObject, eventdata, handles)
% hObject  handle to slider3 (see GCBO)
% eventdata reserved - to be defined in a future version
    of MATLAB
% handles  structure with handles and user data (see
    GUIDATA)

% Hints: get(hObject,'Value') returns position of slider
%       get(hObject,'Min') and get(hObject,'Max') to
    determine range of slider

global betaS;

currentValue = get(hObject, 'Value');
betaS = currentValue;
assignin('base', 'sliderVal', betaS);
set(handles.textLineLum, 'String', betaS);

% --- Executes during object creation, after setting all
    properties.
function slider3_CreateFcn(hObject, eventdata, handles)
% hObject  handle to slider3 (see GCBO)
% eventdata reserved - to be defined in a future version
    of MATLAB
% handles  empty - handles not created until after all
    CreateFcns called

% Hint: slider controls usually have a light gray
    background.
if isequal(get(hObject,'BackgroundColor'),
    get(0,'defaultUicontrolBackgroundColor'))
   set(hObject,'BackgroundColor',[.9 .9 .9]);
end

% --- Executes on slider movement.
function slider4_Callback(hObject, eventdata, handles)
% hObject  handle to slider4 (see GCBO)
% eventdata reserved - to be defined in a future version
    of MATLAB
% handles  structure with handles and user data (see
    GUIDATA)

% Hints: get(hObject,'Value') returns position of slider
%       get(hObject,'Min') and get(hObject,'Max') to
    determine range of slider

global betaI;

currentValue = get(hObject, 'Value');
betaI = currentValue;
assignin('base', 'sliderVal', betaI);
set(handles.textImgLum, 'String', betaI);

% --- Executes during object creation, after setting all
    properties.
function slider4_CreateFcn(hObject, eventdata, handles)
% hObject  handle to slider4 (see GCBO)
% eventdata reserved - to be defined in a future version
    of MATLAB
% handles  empty - handles not created until after all
    CreateFcns called

% Hint: slider controls usually have a light gray
    background.
if isequal(get(hObject,'BackgroundColor'),
    get(0,'defaultUicontrolBackgroundColor'))
   set(hObject,'BackgroundColor',[.9 .9 .9]);
end

% --- Executes on button press in start.
function start_Callback(hObject, eventdata, handles)
% hObject  handle to start (see GCBO)
% eventdata reserved - to be defined in a future version
    of MATLAB
% handles  structure with handles and user data (see
    GUIDATA)

global RP;
global X;
global Y;
global cp;
global p;

% Controlling parameters
global theta;
```

```
global w;
global dirNum;
global betaS;
global betaI;
weightRat = 3;
ks = 1;

if cp == 1
%    set(handles.pSketch,'value',0);
    set(gcf, 'CurrentAxes', RP); % set current working
        panel as resultPanel
    Y = colorPencilSketch(X, ks, theta, w, dirNum,
        weightRat, betaS, betaI);
end

if p == 1
%    set(handles.cpSketch,'value',0);
    set(gcf, 'CurrentAxes', RP); % set current working
        panel as resultPanel
    Y = pencilSketch(X, ks, w, dirNum, betaS);
end
    imshow(Y);

% --- Executes on button press in clearButt.
function clearButt_Callback(hObject, eventdata, handles)
% hObject  handle to clearButt (see GCBO)
% eventdata reserved - to be defined in a future version
     of MATLAB
% handles  structure with handles and user data (see
     GUIDATA)

global X;
global RP;

set(handles.pSketch,'value',0);
set(handles.cpSketch, 'value', 0);
set(handles.textImgLum, 'String', 0);
set(handles.textLineLum, 'String', 0);
set(handles.textWigCon, 'String', 6);
clc;

set(gcf, 'CurrentAxes', RP);
imshow(X);
```


ACKNOWLEDGMENTS

Working on this project is a great source of immense knowledge to me. I would like to sincerely thank Professor Ming-Ting Sun and teaching assistant Maolong Tang for their guidance, advice and encouragement in carrying out this project work. I also want to say thank you to Baihan Lin for enlightened discussion. I wouldn't be able to surpass many difficult challenges without their consistent supports.

REFERENCES

1. Henry Kang, Seungyong Lee, and Charles K Chui. 2007. Coherent line drawing. In *Proceedings of the 5th international symposium on Non-photorealistic animation and rendering*. ACM, 43–50.
2. Nan Li and Zhiong Huang. 2003. A feature-based pencil drawing method. In *Proceedings of the 1st international conference on Computer graphics and interactive techniques in Australasia and South East Asia*. ACM, 135–ff.
3. Cewu Lu, Li Xu, and Jiaya Jia. 2012. Combining sketch and tone for pencil drawing production. In *Proceedings of the Symposium on Non-Photorealistic Animation and Rendering*. Eurographics Association, 65–73.
4. Mario Costa Sousa and John W Buchanan. 1999. Computer-Generated Graphite Pencil Rendering of 3D Polygonal Models. In *Computer Graphics Forum*, Vol. 18. Wiley Online Library, 195–208.