Christopher Erwin
9 November 2016
CSC 872

Fast Prototyping 2: Mean Shift

The provided paper introducing the Mean Shift algorithm provided several examples and applications with which an implementation can be compared. This report will explore the similarities and differences of a limited implementation of Mean Shift through a visual comparison of the implementation's output and the output demonstrated in the paper.

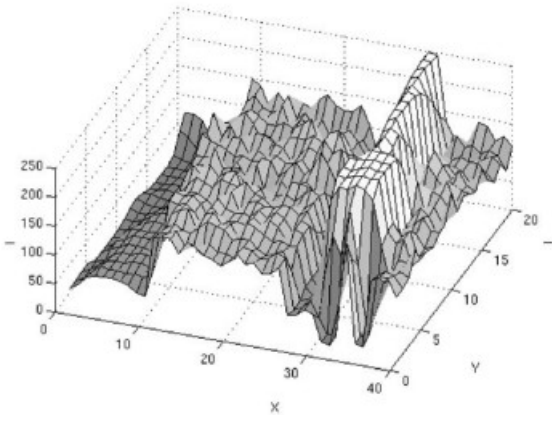
Although the Mean Shift algorithm technically has only one tuning parameter (the bandwidth h), there are a number of other decisions to be made. The kernel function selected and the neighborhood size have a considerable influence over the output and utility of the algorithm. In this implementation, useful and correct results were obtained using the Gaussian kernel function, a neighborhood radius of 15 pixels, a bandwidth value of 30, and a stopping parameter of 20 iterations. These values were initially chosen arbitrarily, but tuned to provide reasonable results in the second and third fast prototyping sessions. Also, while the Gaussian kernel results in a smoother image than the alternative flat Epanechnikov Kernel, the application of a threshold function to the output will produce a similarly segmented image.

Lets begin exploring this implementation through the medium of the 'Cameraman' image. The figure below shows (a) the original image, (b) the “official” MS output from the paper, and (c) the output from this implementation of MS.

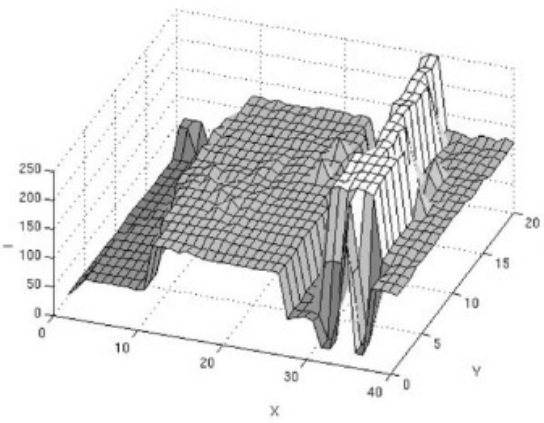
(a) Original Image (b) Paper's output (c) Implementation Output

The rather high bandwidth parameter produced sharper edges than the paper's image, but the effect is the same. The implementation's output image has a lower complexity, with intensity values varying less from pixel to pixel in similar regions of the image. Some detail is lost in the process, but the technique lends itself well to segmentation problems.

The claim that pixel intensities are converging to stable clusters corresponding to the image's inherent regions can be proven by comparing 3-dimensional surfaces of pixel intensities throughout the image. Below is a comparison of (a) the original image's pixel intensities, (b) the paper's reported pixel intensities after MS. The Mean Shift algorithm clearly smooths the surface.

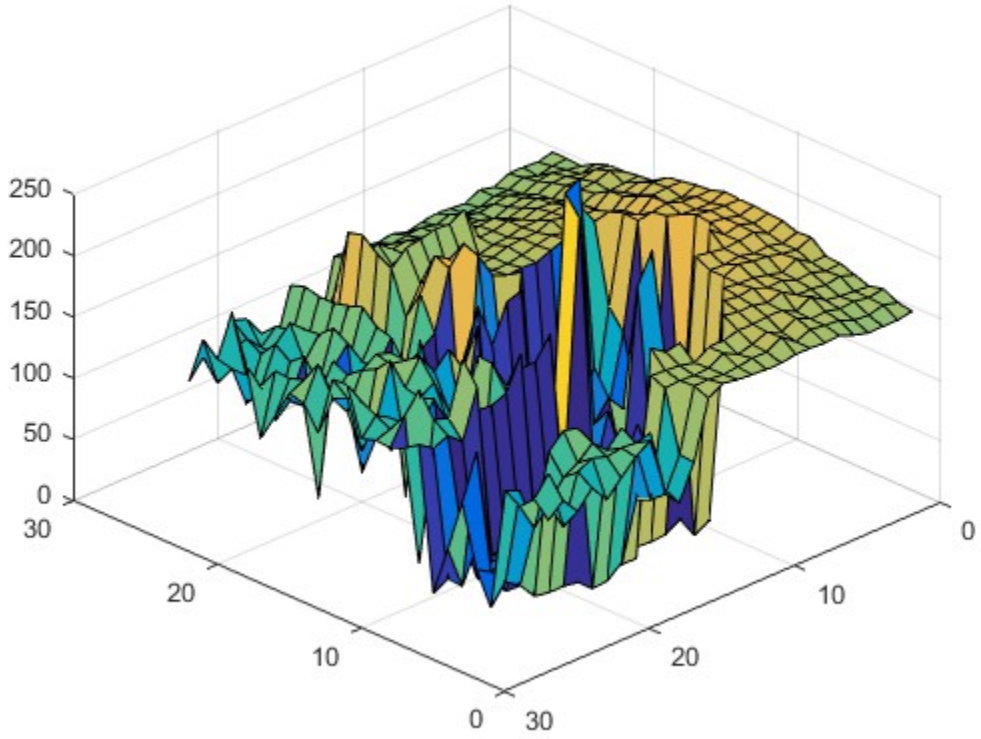


(a) Original intensity data from paper

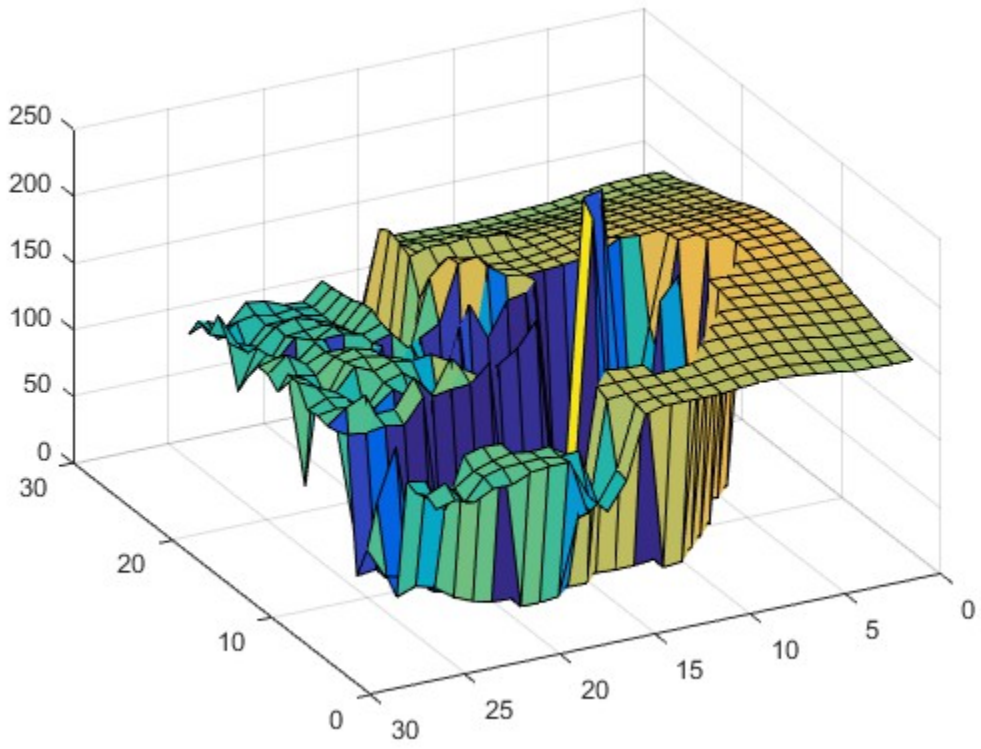


(b) Paper's resulting intensity data

Below is a comparison between the surfaces representing the intensities of (a) the original Cameraman image and (b) the image produced by my Mean Shift implementation.



(a) The surface depicting the original Cameraman image's pixel intensities. Note the roughness and many sharp peaks, including the roughness of the plateau in the back of the graph



(b) The Cameraman image's surface intensities after applying MS. Note the much smoother surfaces. The aforementioned plateau in the back of the graph is significantly smoother.

This implementation of the Mean Shift algorithm is clearly effective at performing the requisite data clustering on grayscale images. The use of the Gaussian kernel function causes minor differences in the results given in the paper, but the desired effect of the algorithm is achieved in this implementation. This code can be further refined by improving performance, extending its operation to the realm of multi-dimensional RGB images, and adding the capability for complete segmentation of the input image. Despite the low scope of this project, this report demonstrates the operation and correctness of my implementation of the Mean Shift algorithm. For reference, the final source code is listed on the next page.

Final Source Code:

```
function [ out ] = MS()
    % Define constants
    h = 30;

    % Load image
    filePath = 'Cameraman.bmp';
    n = 256;
    i = uint8(zeros(n, n));
    i(:,:) = imread(filePath);
    oldImage = double(i);
    newImage = oldImage;

    % Initalize iterator y
    % Loop over each pixel on X axis
    nn = 50;
    for i = 1:n
        for j = 1:n
            % Loop over each pixel in the image
            % y is the current pixel
            y = oldImage(i, j);

            for k = 1:20
                % Do MS on y k times
                % Compute next y i.e. y=m(y,h) + yOld
                num = 0;
                denom = 0;

                % Loop over each pixel in the neighborhood
                for oi = i-nn:i+nn
                    for oj = j-nn:j+nn
                        if (oi > 1 && oi < n && oj > 1 && oj < n)
                            % Calculate numerator
                            %
                            % Norm of X - Xn
                            dist = sqrt((newImage(i, j) - oldImage(oi, oj))^2);

                            % Compute Kernel of the distance
                            g = exp(-(dist^2)/(h^2));

                            % Offset for (i, j) (summed)
                            num = num + (oldImage(oi, oj) * g);

                            % Calculate denominator (summed)
                            denom = denom + g;
                        end
                    end
                end
                % Set the adjusted point
                y = num / denom;
            end
            % Set intensity of y in output image
            newImage(i, j) = y;
        end
    end
    imshow(uint8(newImage))
    out = newImage;
end
```