CMSC 630 Project Part 1 Report

Purpose

The aim of this part of the project is to implement basic image processing algorithms based on what was discussed in lecture. 500 images of cell slides were provided as the source material to be operated on. However, one of the source images (super45.bmp) was omitted due to it being corrupt. Several image operations are implemented in this application including RGB to grayscale conversion, salt-and-pepper noise (also known as impulse noise) addition, gaussian noise addition, histogram calculation, histogram equalized image generation, uniform quantization, linear filtering, and median filtering techniques. Each image processing operation will be described, furnished with results, and some discussion on implementation in code.

Application Design

The application for this project was designed to be user friendly. It is designed in WinForms framework, relying on C# for the implementation of code. It offers both single image and batch image processing, with the ability to set both input and output destinations to process images. For single image processing, the application will display results of the image processing within the application itself, to allow for users to determine if the selected settings are desirable for batch processing. The only dependency this application requires is .NET 6.0 runtime installed. Originally, the application was written to execute on a single thread. However, it was later revised, and a parallel processing method was added to take advantage of 6 cores/12 threads mobile CPU. The parallelized process significantly reduces the processing time, on average about 4.5x faster than the single threaded process (at least on the tested laptop).

Github Repo

https://github.com/talejl/CMSC-630-Image-Processing

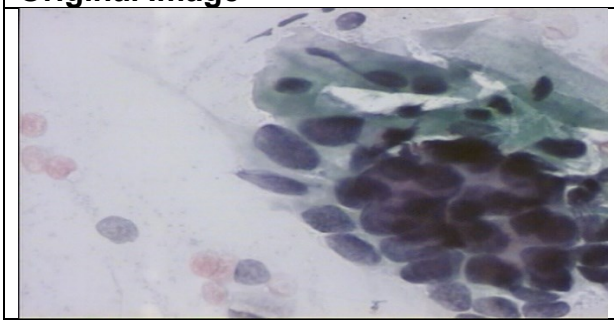
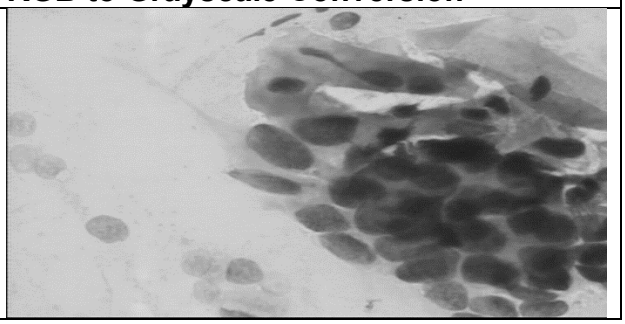
To run, please download Release.zip, unzip, and then launch the executable. The release page also contains a zip file with all processed images for convenience.

https://github.com/talejl/CMSC-630-Image-Processing/releases/tag/ProcessorP1

RGB to Grayscale Conversion Process

The source images were provided in 24BPP (bit-per-pixel) BMP format. A RGB image is a photo composed of red, green, and blue intensities representing a single pixel for each pixel in the image. Each color channel has a byte value assigned to it between 0 and 255, denoting the intensity of the color in its respective channel of that pixel. To convert to grayscale, three methods were considered. The first was to apply widely accepted coefficients to each channel byte representation that would result in a grayscale image. The accepted coefficients for red, green, and blue are 0.299, 0.587, and 0.114

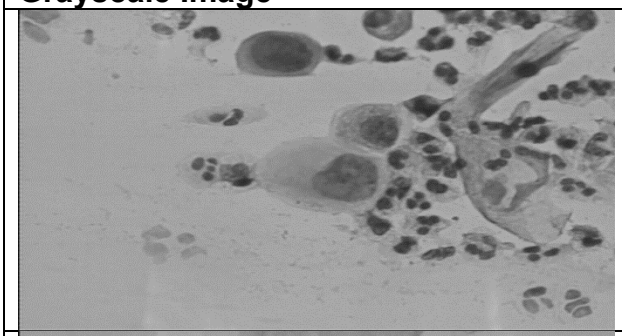
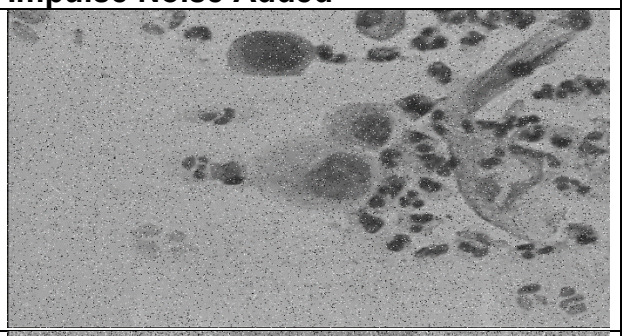
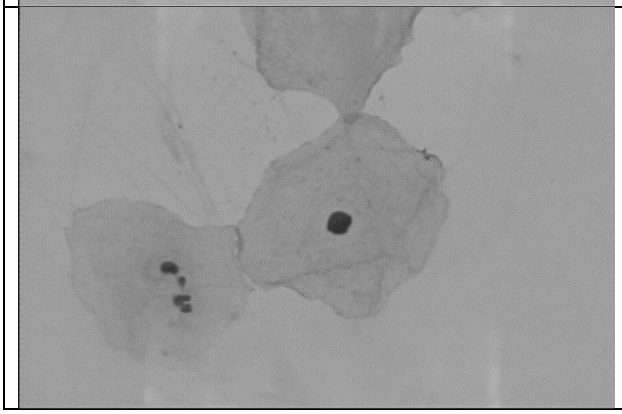
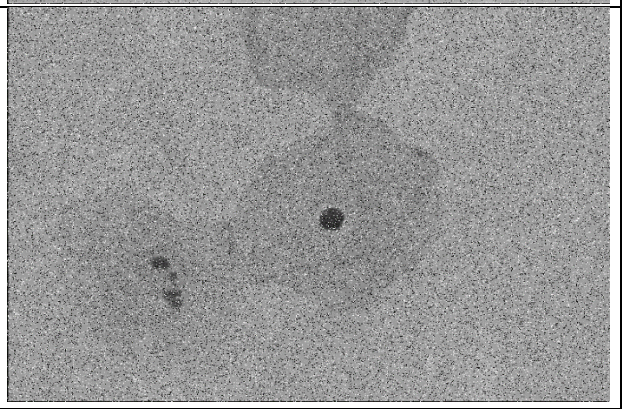
respectively. The second method considered was to average the byte values of the red, green, and blue channels per pixel. The first and second methods consider the values of all 3 color channels. However, the project required the ability to select a conversion to grayscale with respect to a single-color channel. This third method was achieved by assigning a coefficient of 1 to the selected color channel, and 0 to the remaining channels. Another stumbling block that was discovered along the way was that BMP data is read in BGR format, not RGB, so coefficients were not being assigned correctly at first. Furthermore, the image data was originally being manipulated by the C# method of GetPixel and SetPixel, which loads the image into memory, gets the x,y coordinate of the pixel, and then reads out the RGB information, which than can be set on a per pixel basis, then unloads the image from memory. Unfortunately, this method is extremely inefficient and slow. The process was changed to manipulate raw byte data in a byte array for each image using the Lockbits/Unlockbits methods in C#, which is far more efficient and grants performance closer to that of C++. The determined byte value is then written to all 3 color channels to preserve the 24BPP BMP image format. For sample images, please refer to Appendix A. Choosing different color channels to convert to grayscale yielded differing results, highlighting aspects of some cells, while muting others. A sample of an original image and its grayscale counterpart are displayed below.

Original Image	RGB to Grayscale Conversion
	

Salt-and-Pepper/Impulse Noise Addition

Salt and pepper noise is the addition of white and black pixels throughout an image. In the case of grayscale images, the pepper pixel would contain a value of 0, while the salt pixel would contain a value of 255. To meet the criteria of user specified strength, a slider was implemented with values of 0 through 100 as an input to determine the probability of corrupting the pixels of an image. The higher the value, the more corruption will be introduced in the image. A random number generator between 0 and 1 was used to determine whether a salt or pepper corruption would occur, and only if the probability of corruption was met. An alternate method involving passing in a byte array of the grayscale image instead of the whole image was implemented to improve performance, but then scrapped when it resulted in longer processing time in comparison to processing the image. This method is commented out in the code. For sample images, please refer to Appendix B. Impulse noise may be removed with the

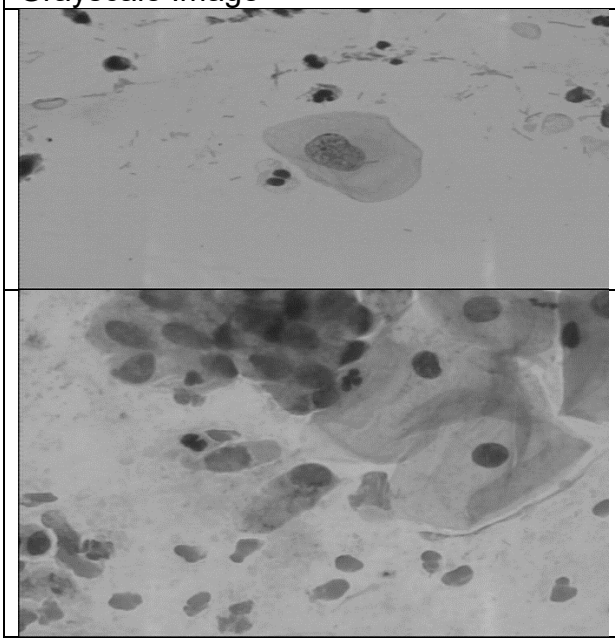
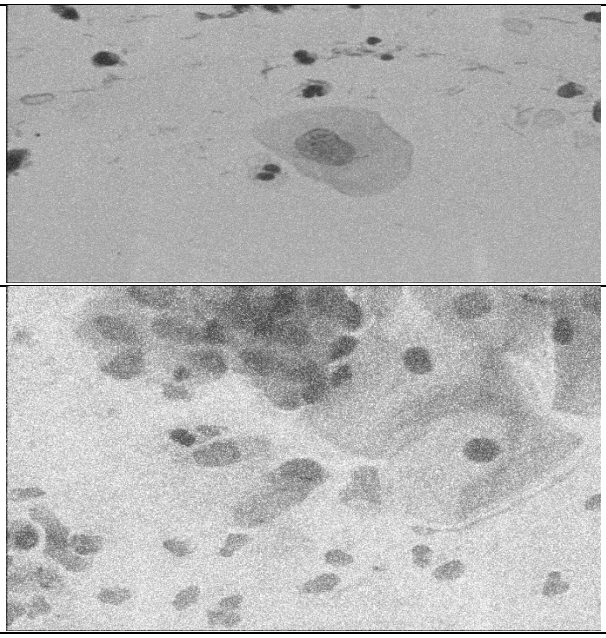
median filter, if the amount of noise is not extreme. A sample grayscale image and an image with impulse noise added are displayed below.

Grayscale Image	Impulse Noise Added
	
	

Gaussian Noise Addition

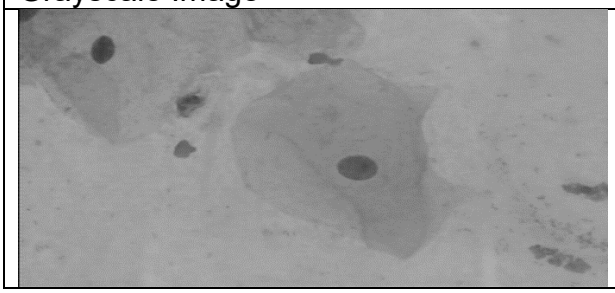
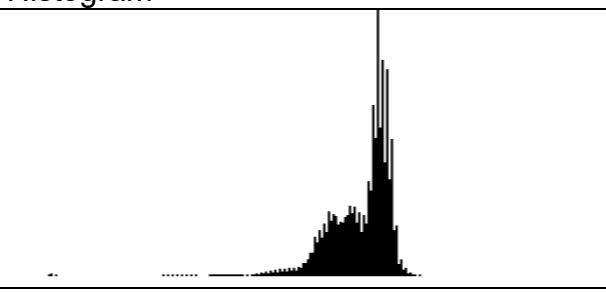
For the purposes of this project, a zero-mean Gaussian distribution equation was used to generate Gaussian noise. The user specified input for gaussian noise was defined in the GUI as a slider, with values between 0 and 100. This variable was directly referenced in the Gaussian equation as the variance. As before, the grayscale image is read into memory and then converted into a byte array. To generate Gaussian noise, a new float array for values between 0 and 255 was declared, then iterated through the Gaussian equation to generate the distribution. A running total was kept. In the next phase, the respective values generated by the equation in the array were divided by the sum of all the values in the array, then multiplied by the byte size of the image to determine how many bytes this noise would occupy. The resulting value is then stored in another byte array of the same size as the image, with the value being written out for as long as the representation was determined for that intensity. The remaining space in the noise array was zeroed out, and then that noise array was randomized. The original image buffer is looped through, adding noise to the original byte in the same position in the original image buffer, then copying that result value to the modified image buffer. The greater the value of the variance, the more noise is generated in the resulting image. Parallelization of some of the process was attempted, but no discernable performance gain was found. These operations are found commented out within the gaussian noise generation method. For

sample images, please refer to Appendix C. A sample grayscale image and respective gaussian noisy image are displayed below.

Grayscale Image	Gaussian Noise Added
	

Histogram Generation

The process of histogram generation in a grayscale image is not an overly complicated one. In the case of this application, a new integer array is declared with a length of 256, to count all intensities between 0 and 255. The grayscale image is loaded into memory, with the raw byte information copied into a byte array, and then looped over in increments of 3, since all 3 byte values will contain the same intensity value as per the grayscale conversion within 24BPP context. While counting the intensity value of each pixel, a temporary integer keeps track of the highest encountered intensity. That placeholder value is then used to determine the roof of the graph. A function then draws the out a line per intensity along the y axis, with length of the line representing the how frequently that intensity occurs in the image. For sample images, please refer to Appendix D. A sample of an image and histogram are displayed below.

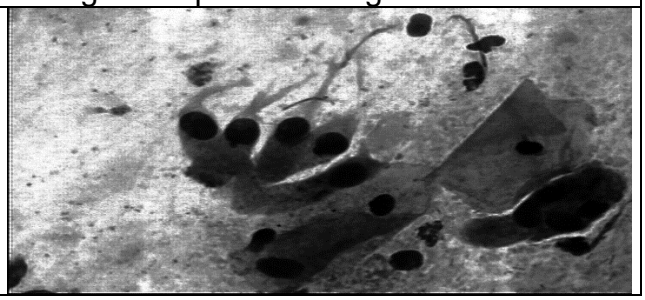
Grayscale Image	Histogram
	

Averaged Histograms Per Cell Class

As per project specifications, an averaged histogram for each class of cells was to be calculated. The process was relatively simple, in which the filenames were used to determine the members of each class of cell. The histogram calculation function was called for each member of the cell class and stored in its own array. The arrays for each member of their respective class were added together, then divided by the number of members in each class, creating the average for that class. The drawing function was then used on the averaged histogram array. For sample images, please refer to Appendix E.

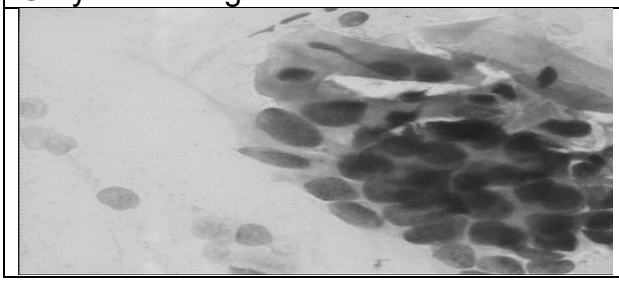
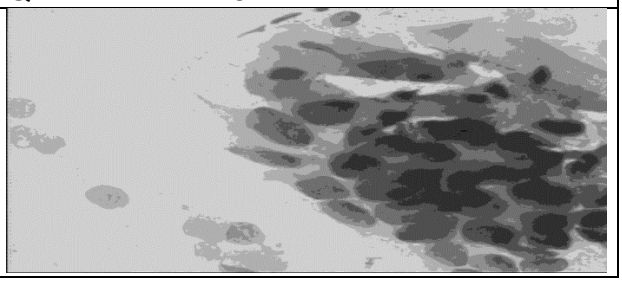
Histogram Equalized Image

Histogram equalization is an image processing technique used to attempt and extract more information from an image by boosting the contrast. The methodology to create a histogram equalized image involved first calculating the histogram of the image. The index of the array represents the intensity, and the value assigned to that index represents the number of pixels containing that intensity. Each index value was divided by the total number of pixels to get the probability of that intensity occurring within the image. Then, the original image is looped through, to get the intensity of each byte. Using the byte value at the position, the probabilities for that intensity are summed, then multiplied by the maximum intensity of 255, since this is a grayscale image. The resulting value is written in the same position as the original byte value, in a separate image buffer, written to all 3 bytes representing the pixel. While contrast is indeed boosted for the images, it can also lead to a loss of detail, some of which may be critical in other parts of image analysis. Histogram Equalization also introduced/amplified noise in images where it was previously hidden. Therefore, it may be prudent to pass histogram equalized images through a denoising filter before proceeding to more complex operations. A sample histogram equalized image is displayed below. For more sample images, For sample histogram equalized images, please refer to Appendix F.

Grayscale Image	Histogram Equalized Image
	

Uniform Image Quantization

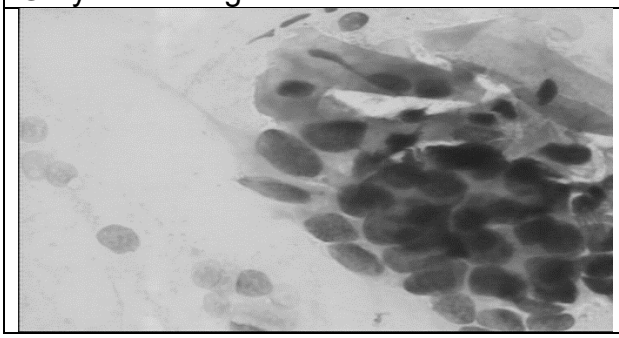
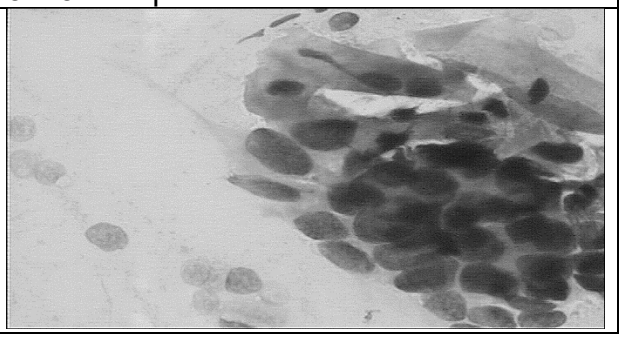
Uniform Image Quantization is the discretization of a set of values to a specified amount. In essence, it is compression of information within the image. The project specification required to have image quantization to user specified levels. The slider in the application allows the user to decide the number of colors it wants the new image to contain. With the default setting set to 8, the image will be reduced from 256 intensities down to 8, containing a step size of 32. The average between two steps becomes the new intensity, and the new image uses the intensity values of the original image as the indices to determine the quantized intensity to use from the quantization calculation. The more compression is performed, the larger the quantization error is since there is more information being lost compared to the original image. Quantization error between images at the same quantization level depends on how varied the intensities are in the source grayscale image. The quantization error of the image was determined by taking the difference of the quantized value and the original intensity value, then squaring the result. The results for each intensity were then summed together to the total quantization error. If the range is not extremely variable, the image is more easily compressed, with less error in quantization. In images with a large variety of intensities, the quantization error will be higher, reflecting the loss of information because of the compression. A sample quantization is demonstrated below. For more sample 8-level uniform quantized images, please refer to Appendix G.

Grayscale Image	Quantization to 8 Levels
	

Linear Filtering

As per project specifications, linear filters of user specified strength and customizable weights were to be included in the application. In the application, a dropdown menu allows for a linear filter with a mask size of 3x3 or 5x5, along with factor and bias textbox inputs for the user to define. Preset values were provided for the purposes of demonstration. IN the case of the 3x3 linear filter, the preset is to sharpen the image with a 3x3 mask. The hot pixel carries a weight of 5, while the surrounding cardinal pixels contain values of -1, while the ordinal pixels contain values of 0. The sum of the sharpening filter pixel weights is 1, with a factor of 1.0 and a bias of 0. The 5x5 filter contains a preset for a Gaussian Blur filter. Since the byte information of the images is flattened into a 1D array, the matrix of weights is applied by calculating the byte positions using x/y coordinates, image stride, and 3 bytes per pixel consideration. The linear filtering masks both need to start an offset

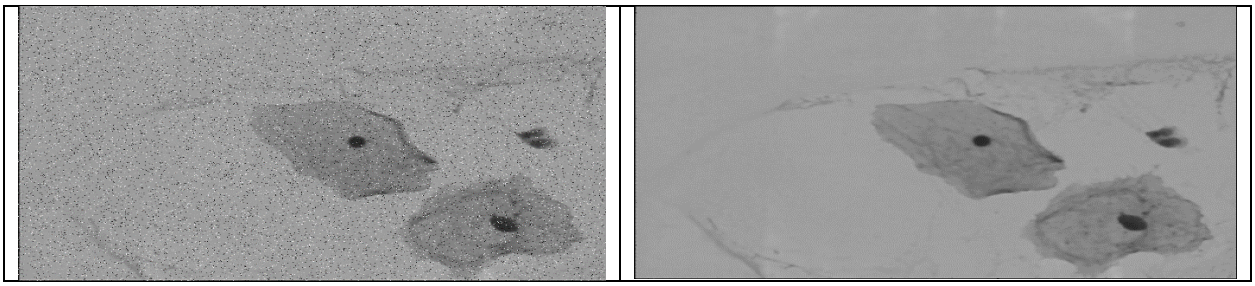
in the image, or else it would begin in an area that is out of bounds of the image. In the case of sharpening, the definition of the edges is enhanced with this filter, at the expense of amplifying artifacts and noise in the image. For sample sharpened images, please refer to Appendix H. A sample of the original grayscale image and its linear filtered counterpart are displayed below.

Grayscale Image	3 x 3 Sharpen Kernel
	

Median Filter

Median filter exceeds at denoising salt and pepper noise, while preserving edges of objects. The process of applying the median filter to the actual byte information is identical to that of the linear filter as described previously, but with an added sort step before applying the change to the hot pixel. It is a non-linear filtering process with a time complexity of O(n log n), which can be attributed to the sort function For this application, median filter with mask sizes of 3x3 and 5x5 are defined, with base weights of 1 on every pixel. Changing a weight will replicate the specified pixel in the mask as many times as specified in the weight matrix, making it more likely for that pixel to become the median value to be assigned to the hot pixel. With default values of 1 for all pixels, each pixel value appears only once in the list of values to be sorted. For sample median filtered images, please refer to Appendix I. A simple test may be conducted by using the application to generate the impulse corrupted image, then feeding that image to be processed by the median filter and checking to see if it has been successfully denoised. Like the linear filter, the median filters also start at an offset as to not consider pixels that are out of bounds. As a result, there is a small loss of information around the borders of the image, evidenced as black pixels around the entire image. A sample impulse noise corrupted image, and post median filtering of the image are displayed below.

Impulse Corrupted Image	Median Filtered Image



Performance Metrics

To evaluate the performance of each image processing operation, a stopwatch for each process was declared. The stopwatch was started immediately before the method is called and stopped immediately after the method returns the processed image. Single image operations are performed on the order of milliseconds, with the Gaussian Noise generation consuming the most time. As stated earlier, the initial process was single threaded. Single thread performance metrics are described below

Single-Threaded Batch Processing Performance Metrics		
Operation	**Total Execution Time in milliseconds**	**Average Execution Time in milliseconds**
Grayscale Conversion	11530	23
Impulse Noise Addition	46248	92
Gaussian Noise Addition	216792	434
Histogram Generation	2487	4
Histogram Equalized Image	45864	91
Class Average Histogram Generation	6513	930
Uniform Quantization	29435	58
Linear Filter	16475	33
Median Filter	37020	74
Total Execution Time	412364	
Separate Total Timer Time	415207	
Difference in Timers	2843	
Execution Time in Minutes	6.92	

The process was then converted to run in parallel to improve performance. While the single threaded process completes in just under 7 minutes, the multithreaded process completes in under 2 minutes. However, the timers set within the parallel process are completely broken and do not measure correctly. Only the total timer, which is located outside of the parallelization loop, measures the completion time correction. On a 6 core / 12 thread processor, the timers in the parallel loop had to be divided by 3 to give an approximate result close to that of the total timer. This number will likely have to change

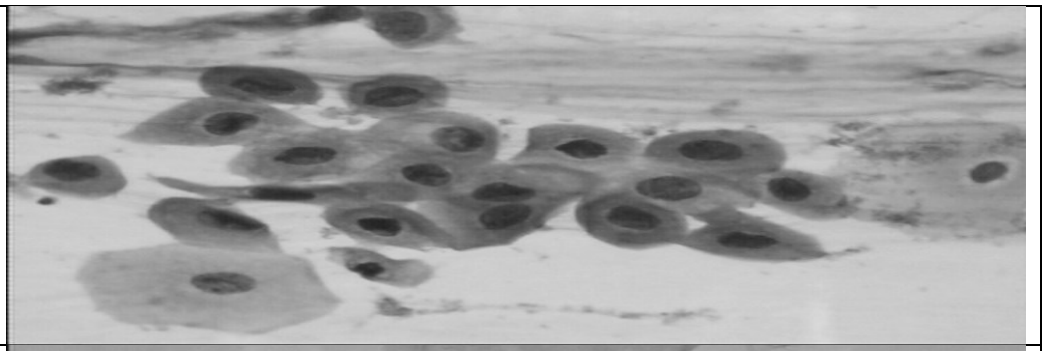
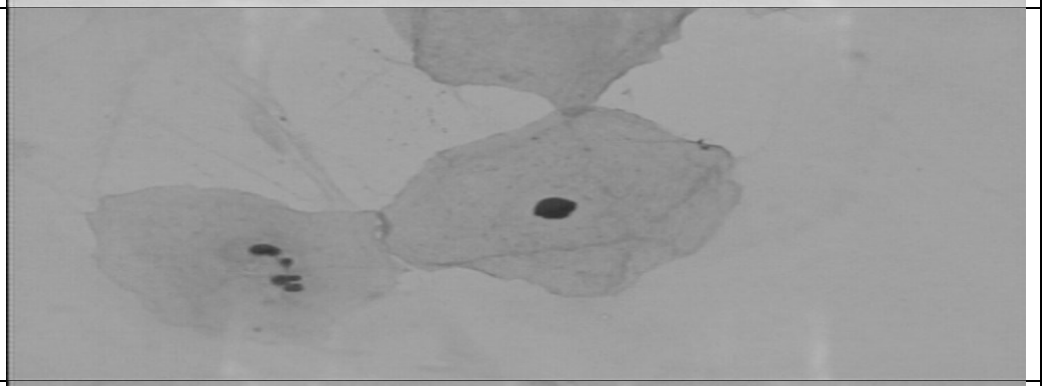
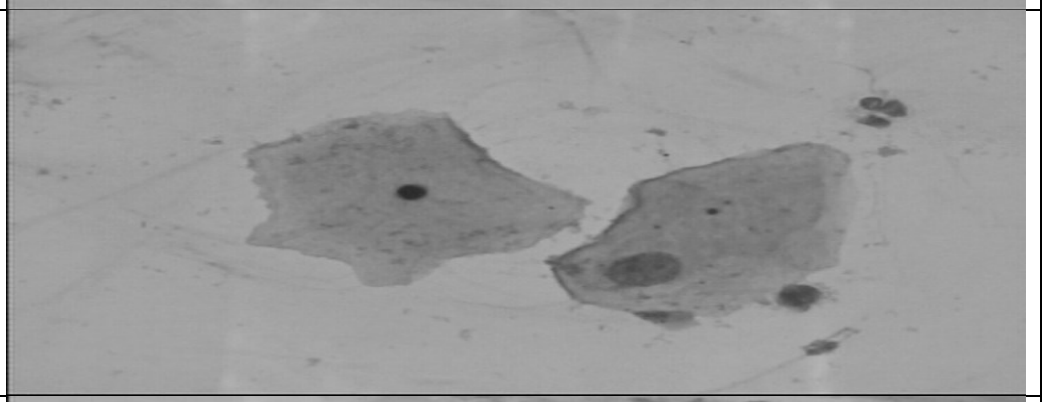
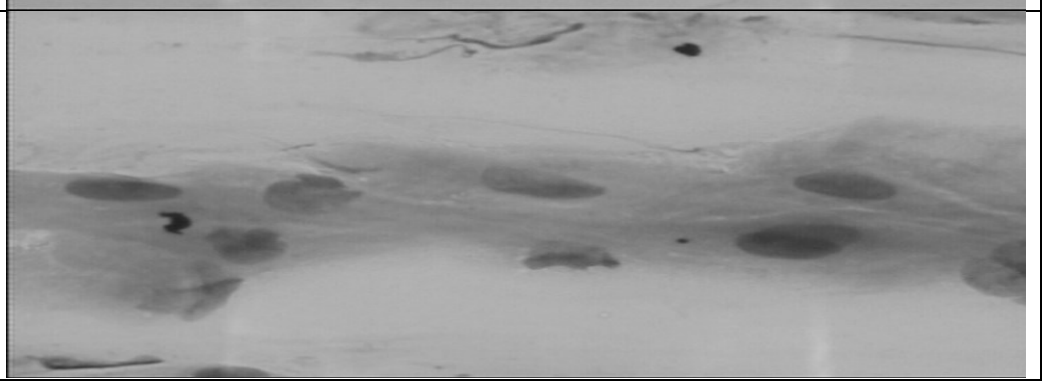
with the number of available threads if performed on other systems. When comparing the single-threaded total timer to the multi-threaded total timer, the multithreaded process is nearly 4.5x faster.

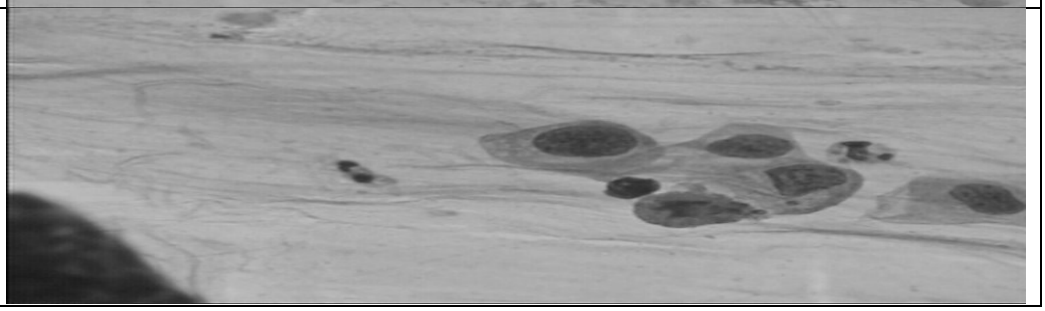
Multi-Threaded Batch Processing Performance Metrics		
Operation	**Execution Time in milliseconds**	**Average Execution Time in milliseconds**
Grayscale Conversion	8673	17
Impulse Noise Addition	15823	31
Gaussian Noise Addition	21098	42
Histogram Generation	2876	5
Histogram Equalized Image	13411	26
Class Average Histogram Generation	2433	347
Uniform Quantization	13116	26
Linear Filter	9595	19
Median Filter	12008	24
Total Execution Time	99033	
Separate Total Timer Time	96223	
Difference in Timers	2810	
Execution Time in Minutes	1.60	

Appendix

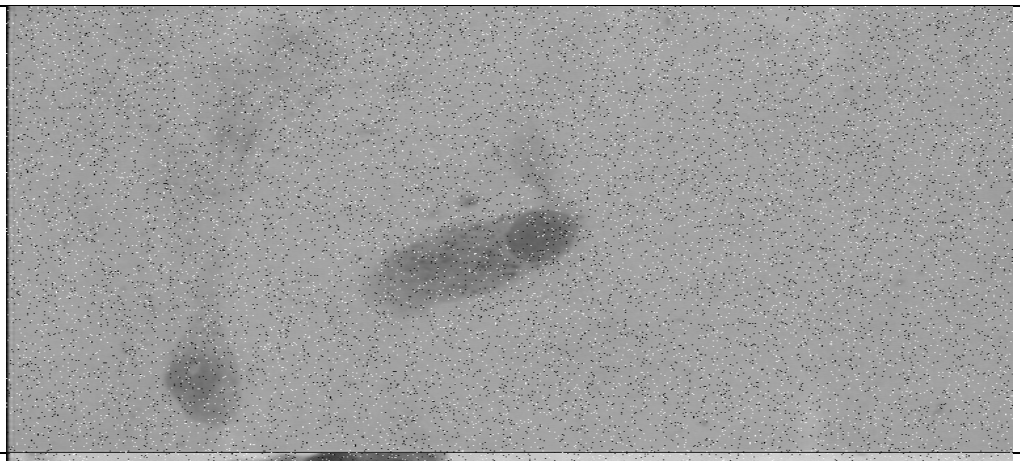
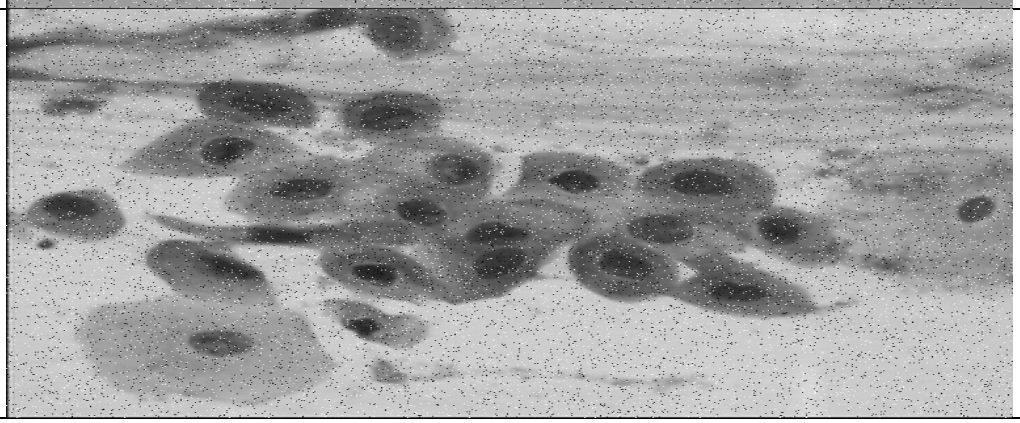
A. RGB to Grayscale Conversion

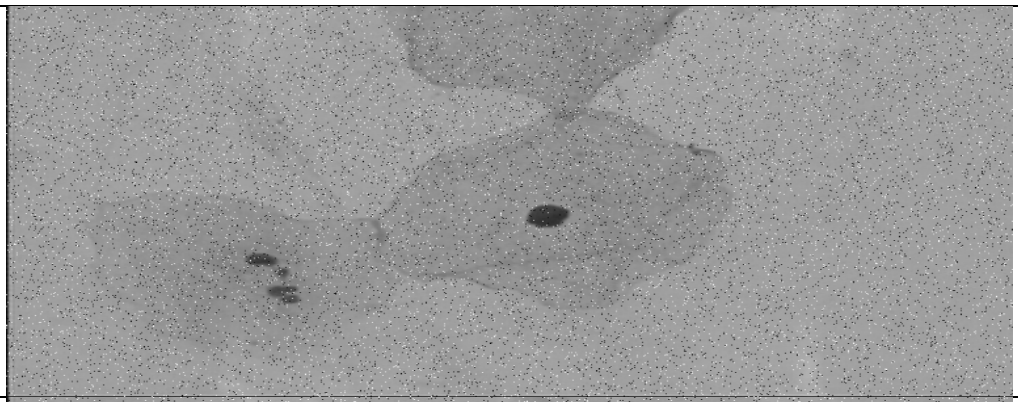
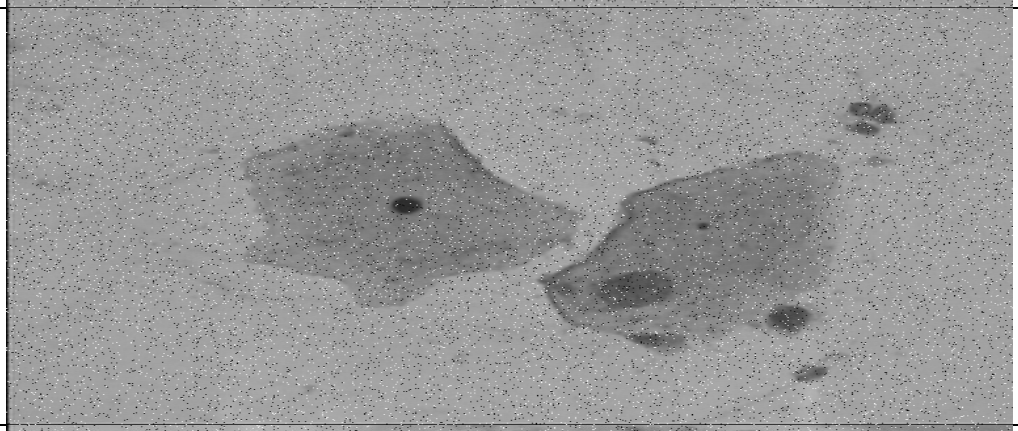
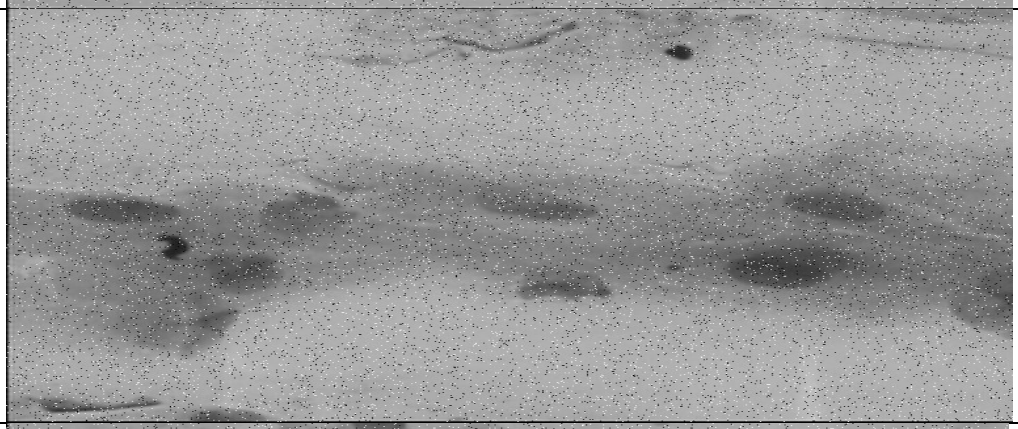
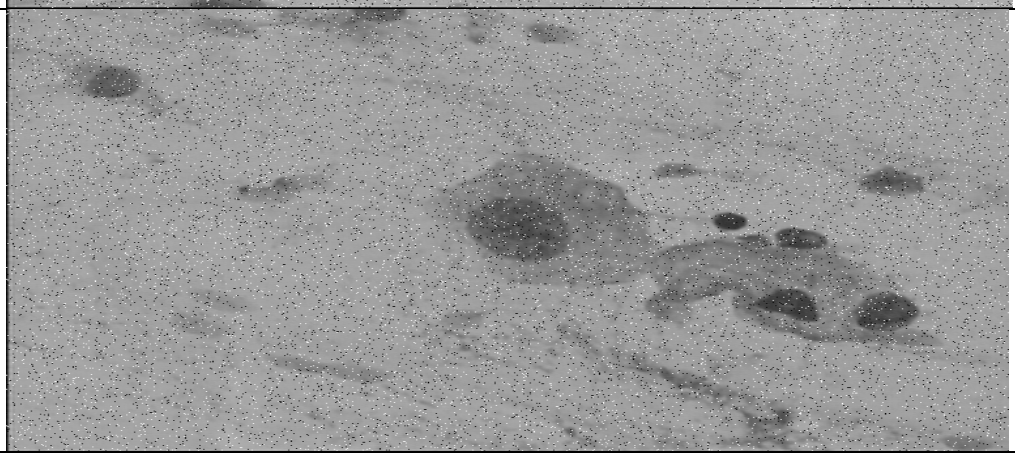
Cell Class	RGB to Grayscale
Columnar E	

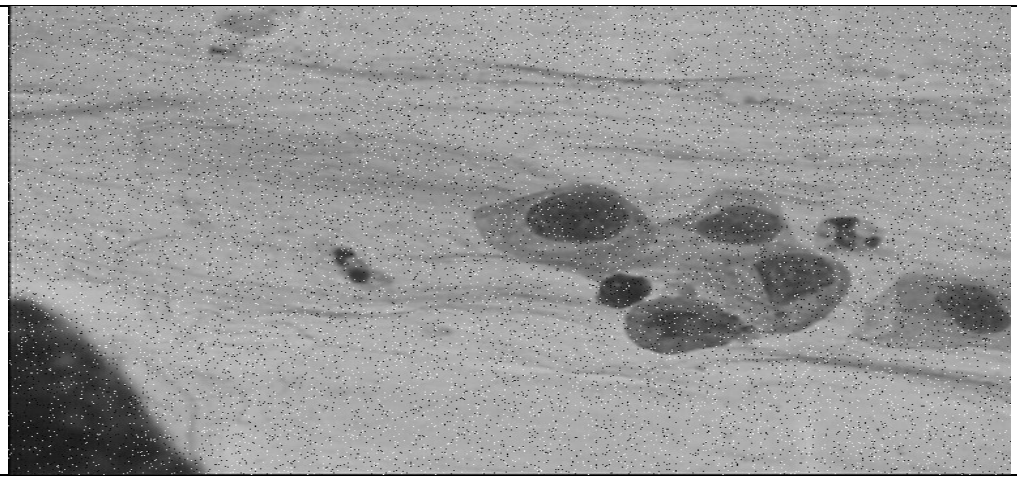
Parabasal SQ E	
Intermediate SQ E	
Superficial SQ E	
Mild NKDP	

Moderate NKDP	
Severe NKDP	

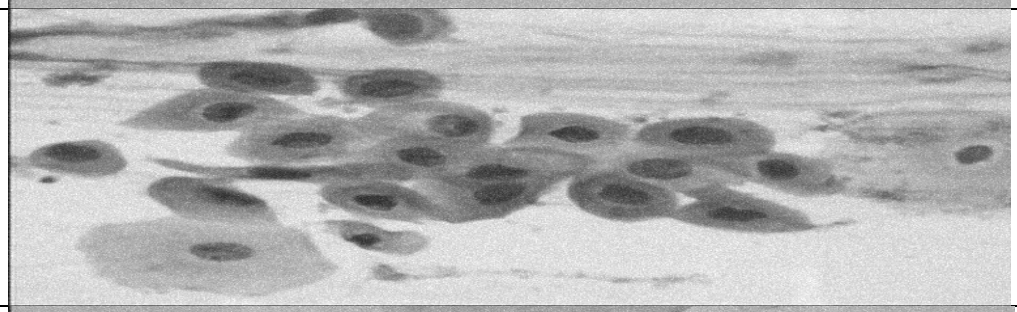
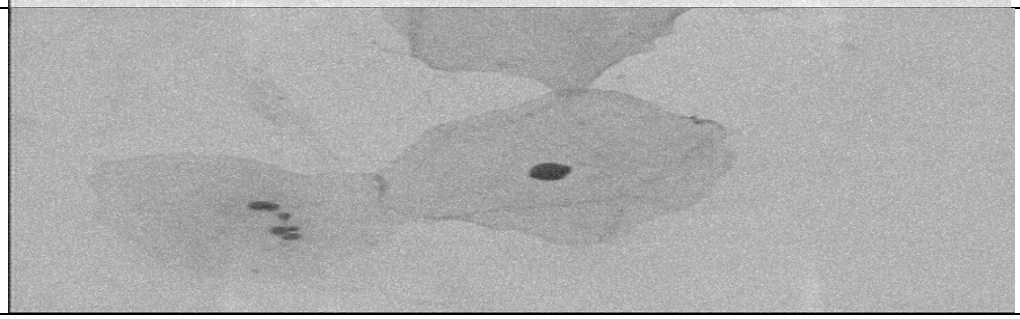
B. Impulse Noise Corruption

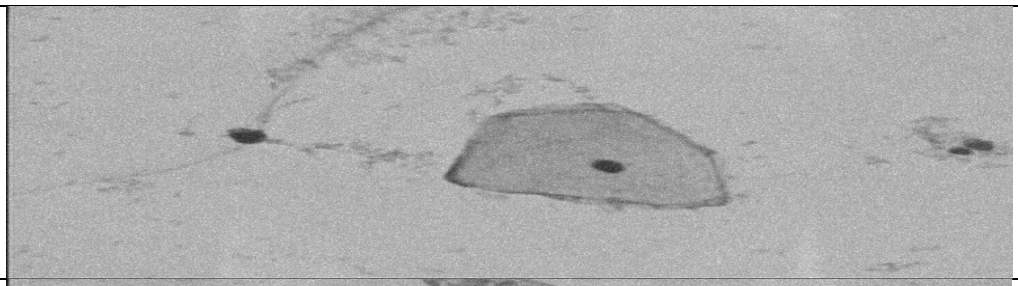
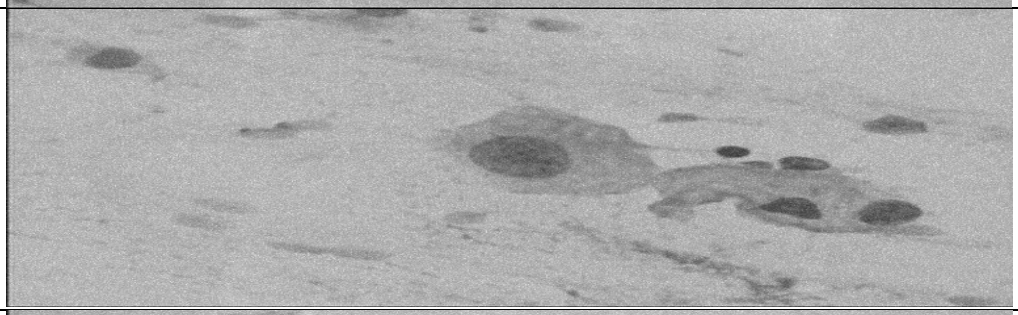
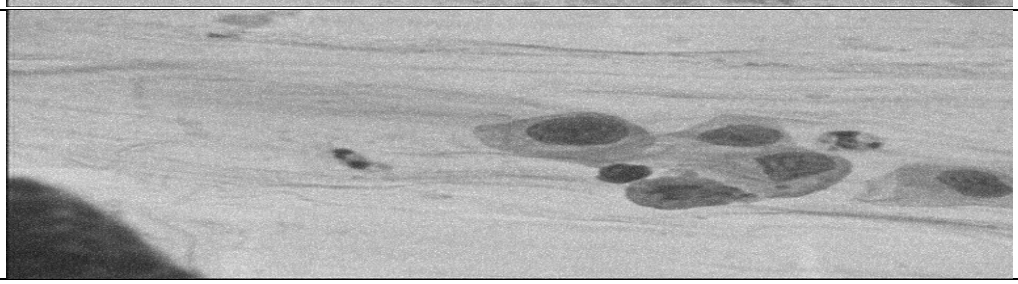
Cell Class	Impulse Corrupted Image
Columnar E	
Parabasal SQ E	

Intermediate SQ E	
Superficial SQ E	
Mild NKDP	
Moderate NKDP	

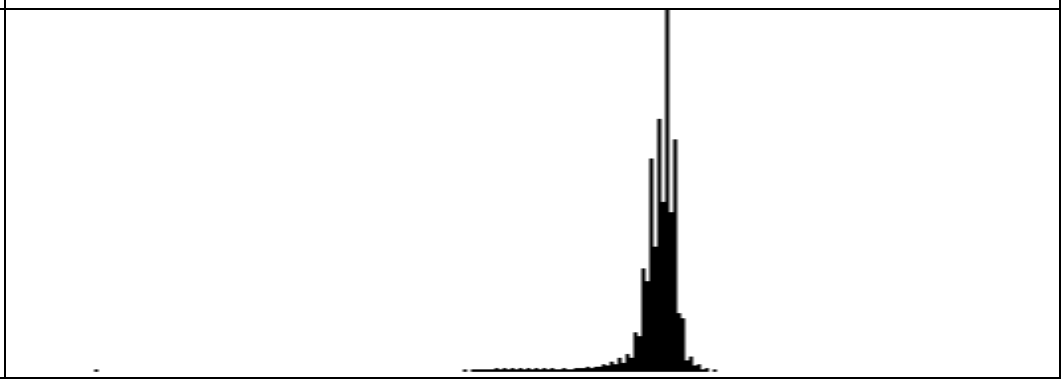
Severe NKDP	

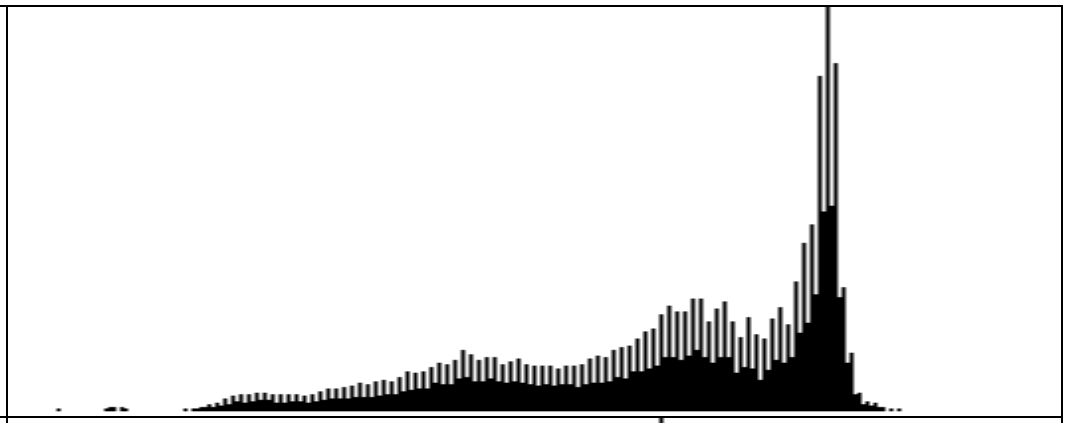
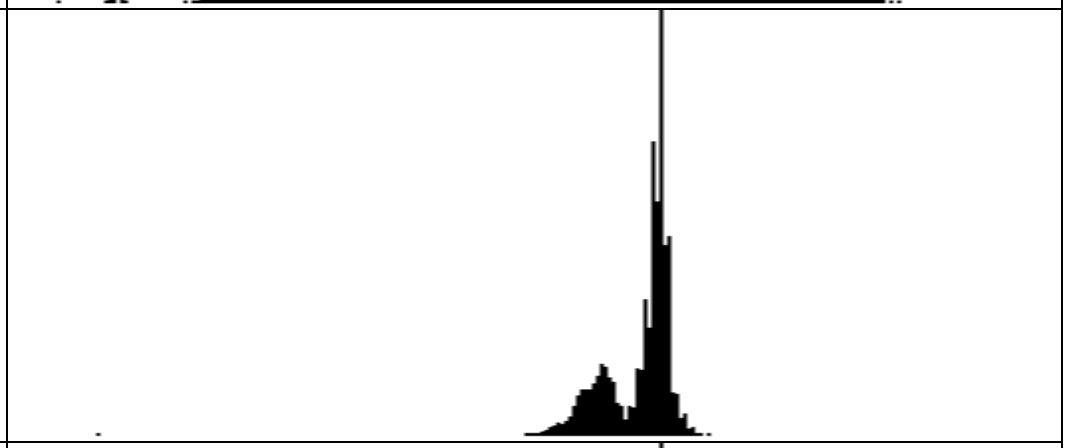
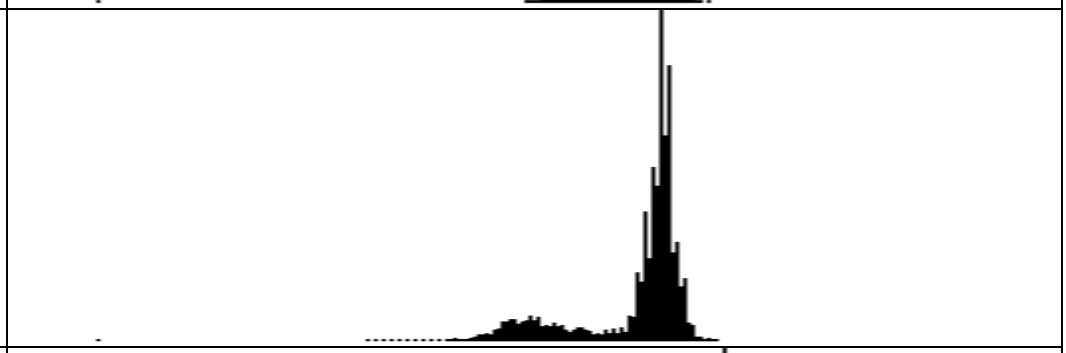
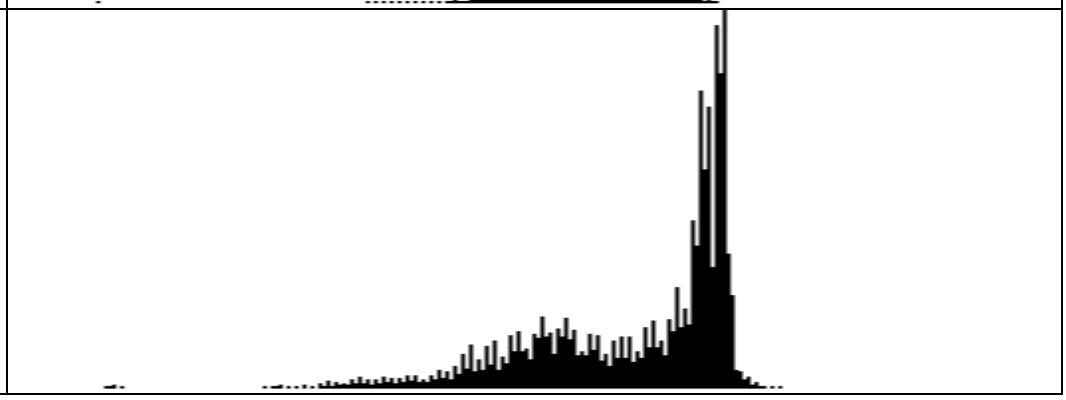
C. Gaussian Noise Corruption

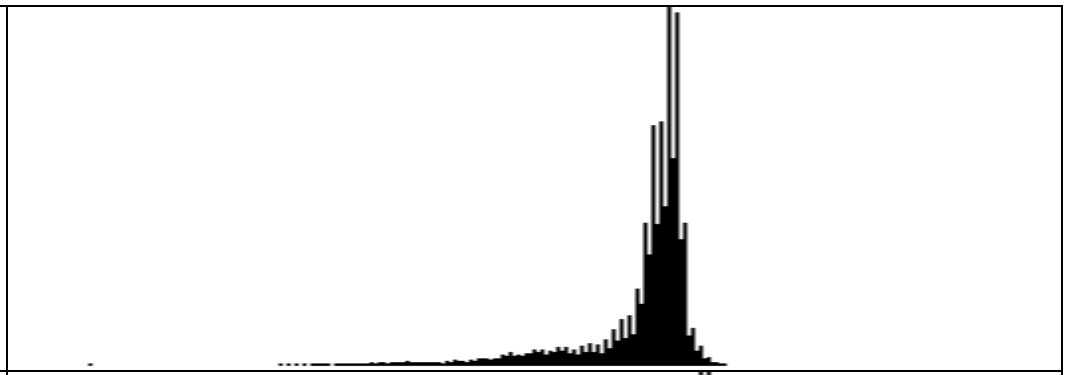
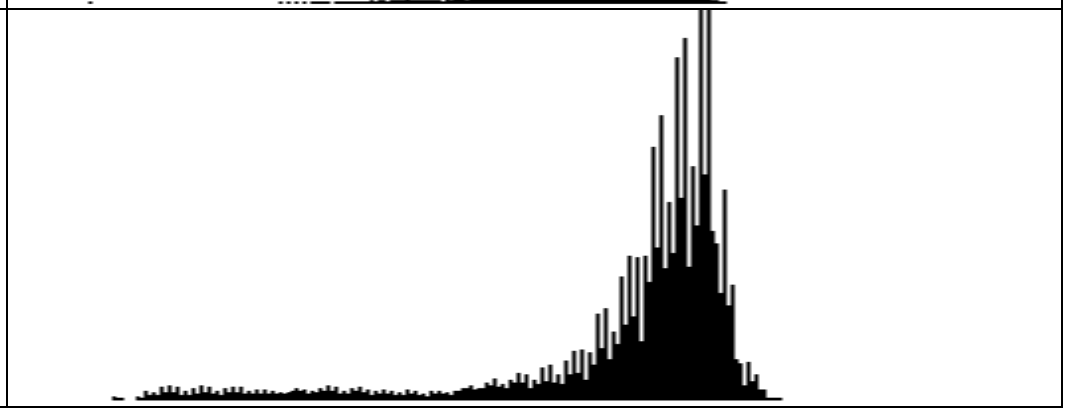
Cell Class	Gaussian Noise Corrupted Image
Columnar E	
Parabasal SQ E	
Intermediate SQ E	

Superficial SQ E	
Mild NKDP	
Moderate NKDP	
Severe NKDP	

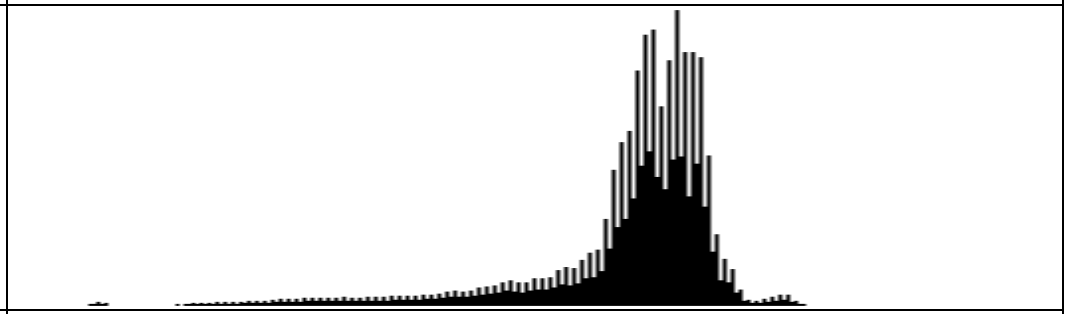
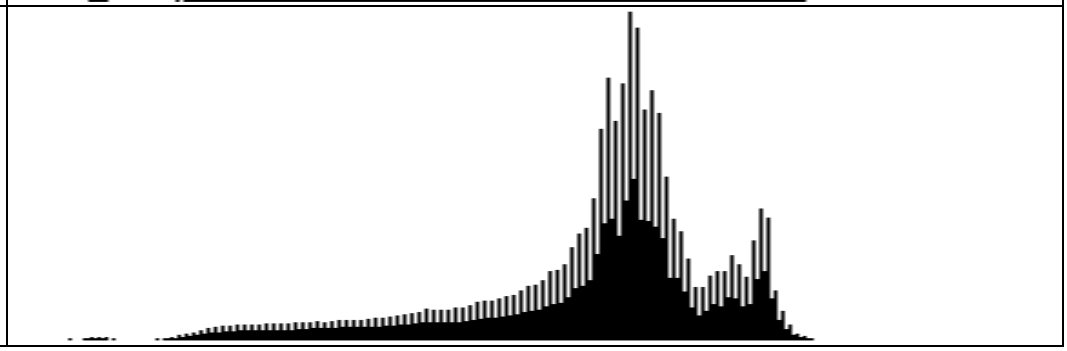
D. Histograms

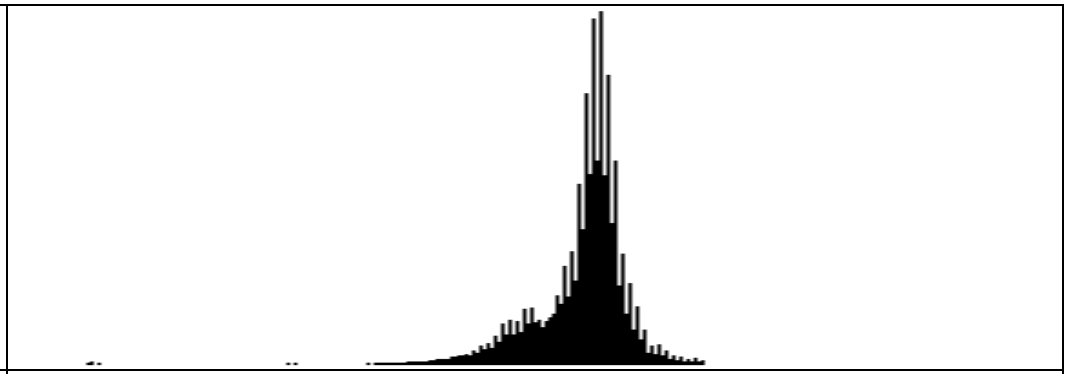
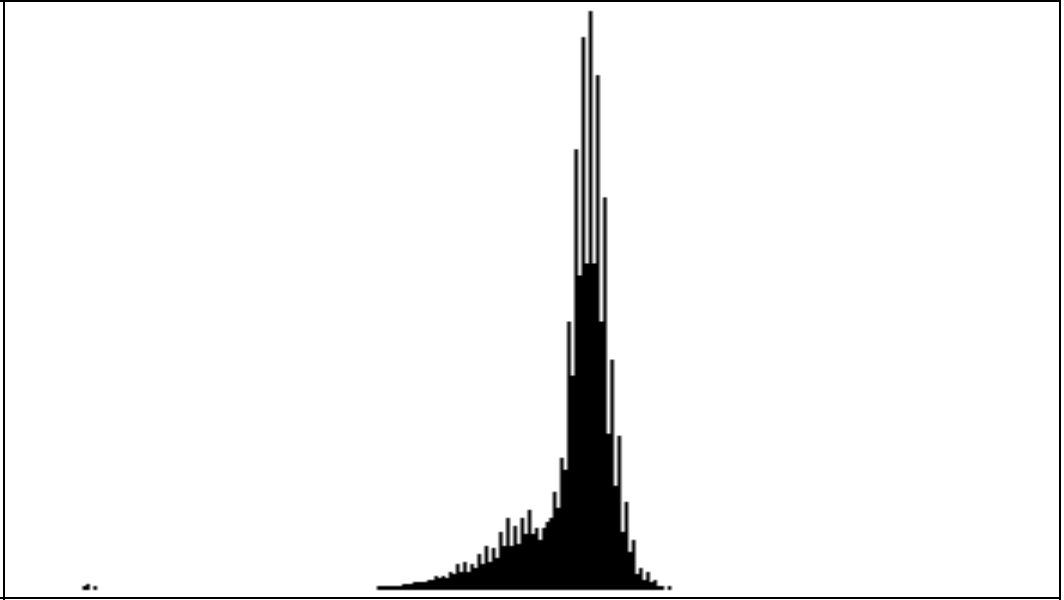
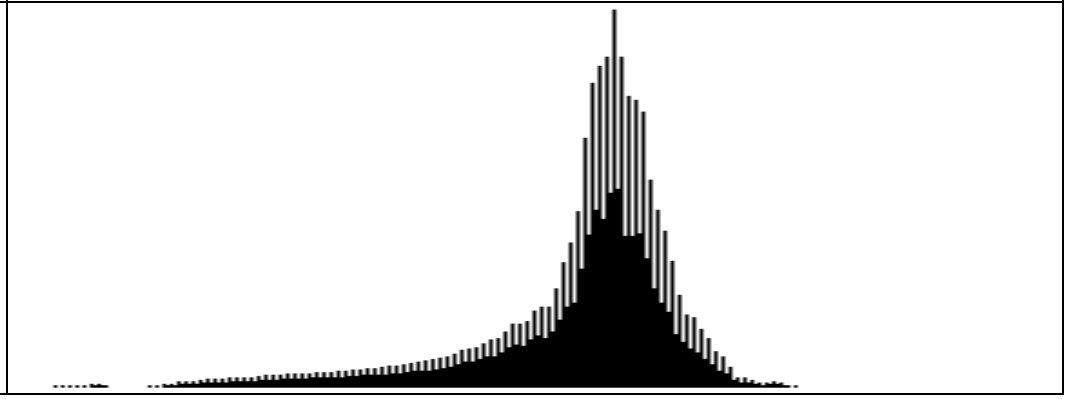
Cell Class	Histogram
Columnar E	

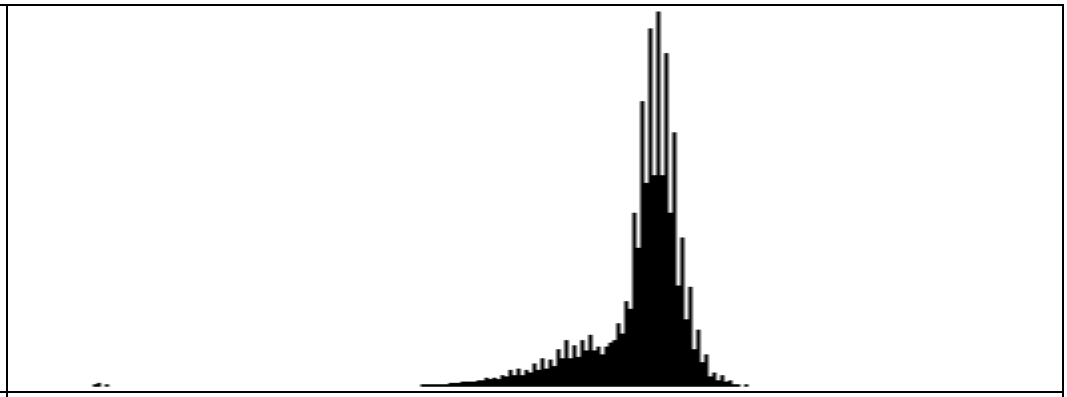
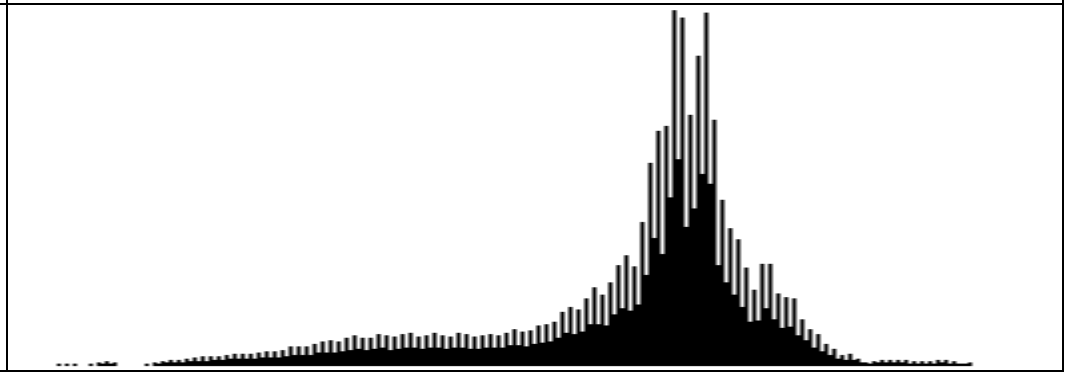
Parabasal SQ E	
Intermediate SQ E	
Superficial SQ E	
Mild NKDP	

Moderate NKDP	
Severe NKDP	

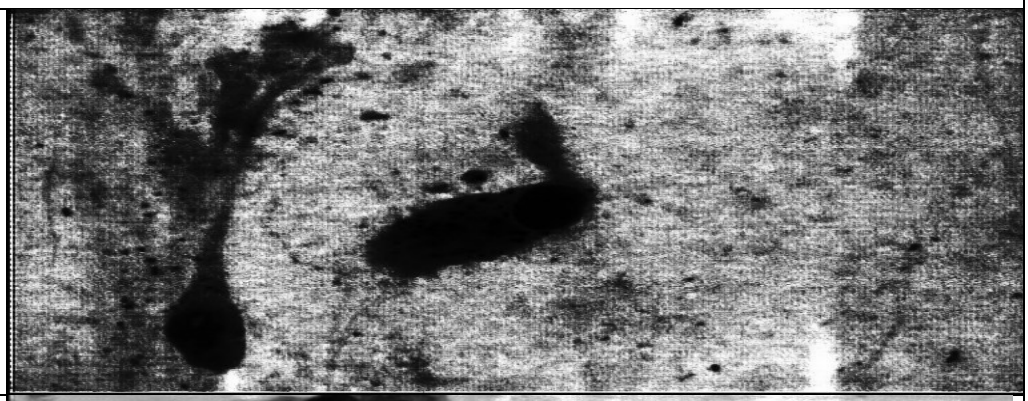
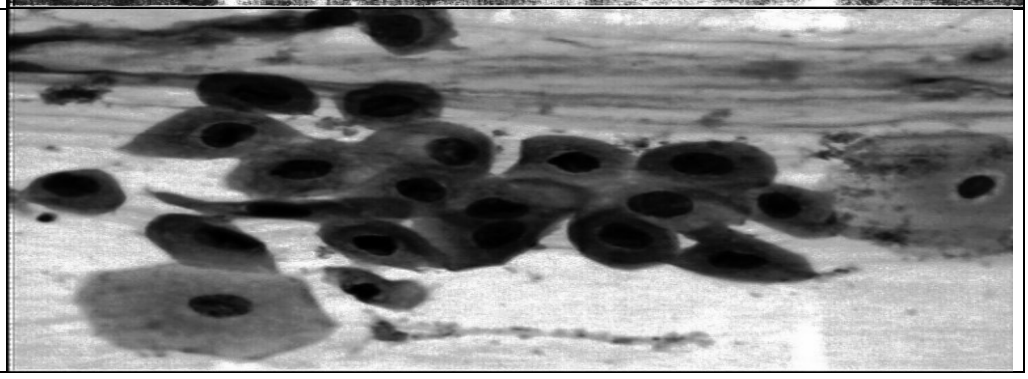
E. Class Average Histogram

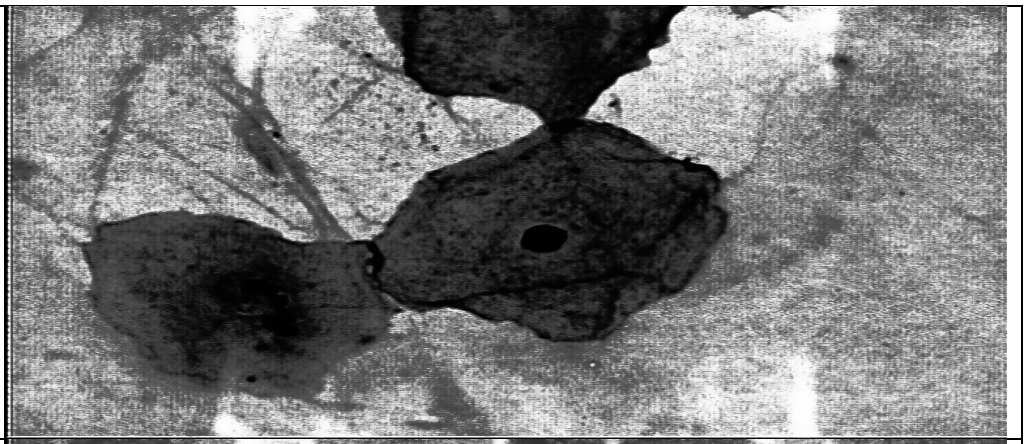
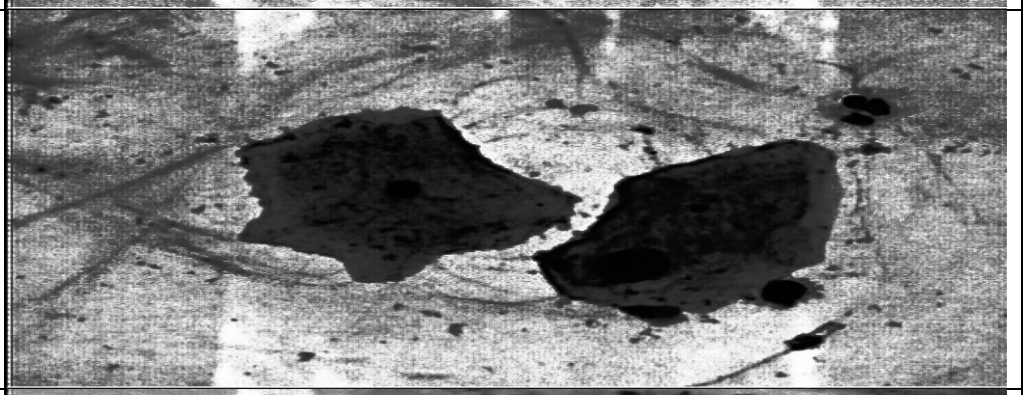
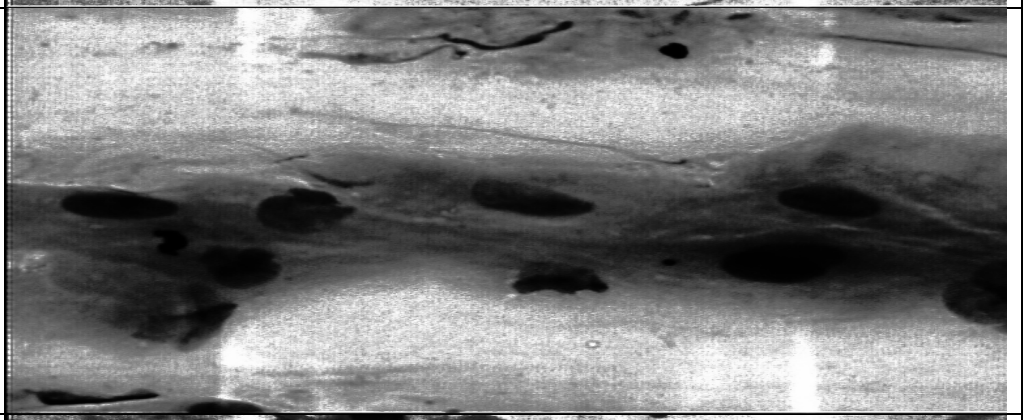
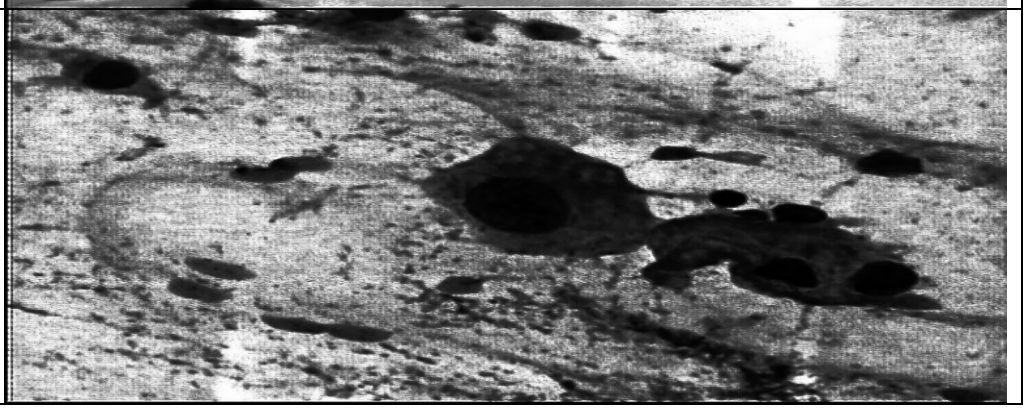
Cell Class	Averaged Histogram
Columnar E	
Parabasal SQ E	

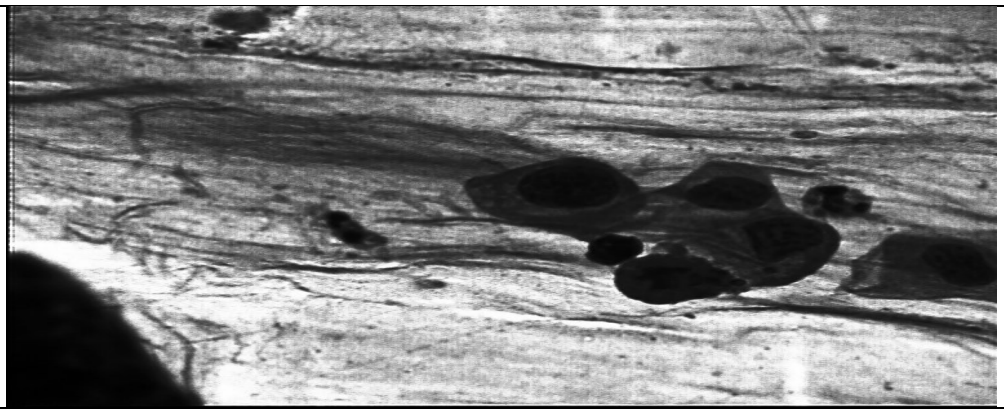
Intermediate SQ E	
Superficial SQ E	
Mild NKDP	

Moderate NKDP	
Severe NKDP	

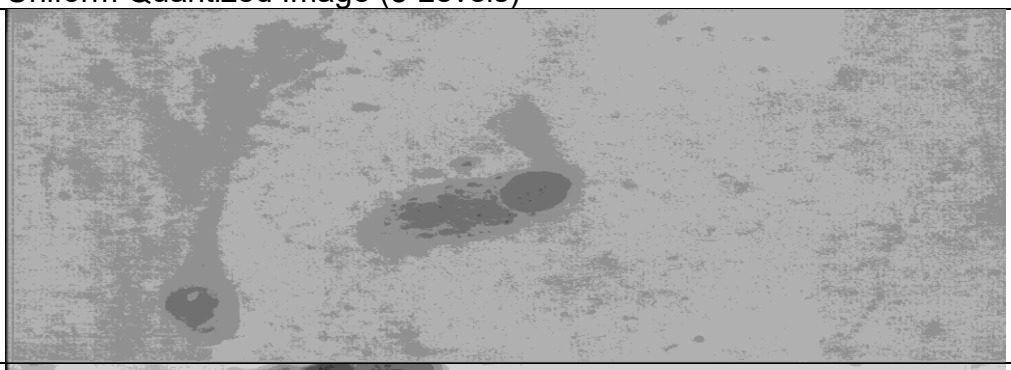
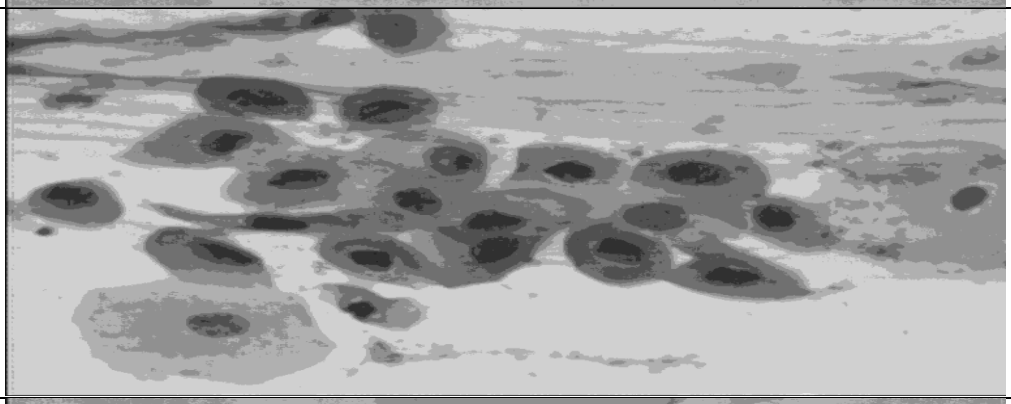
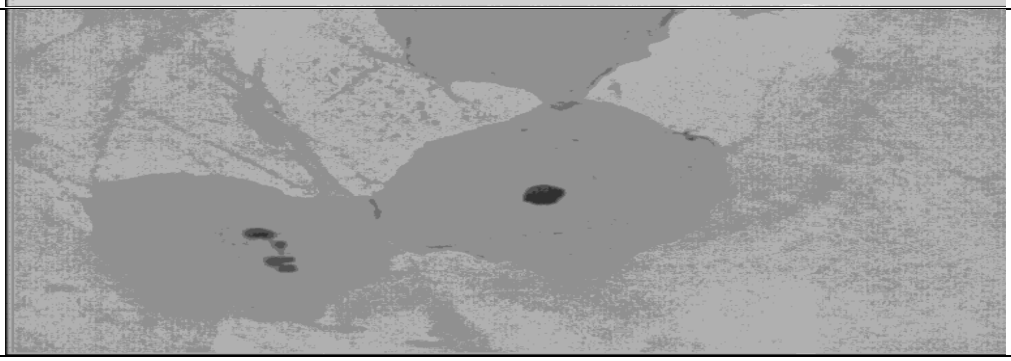
F. Histogram Equalized Images

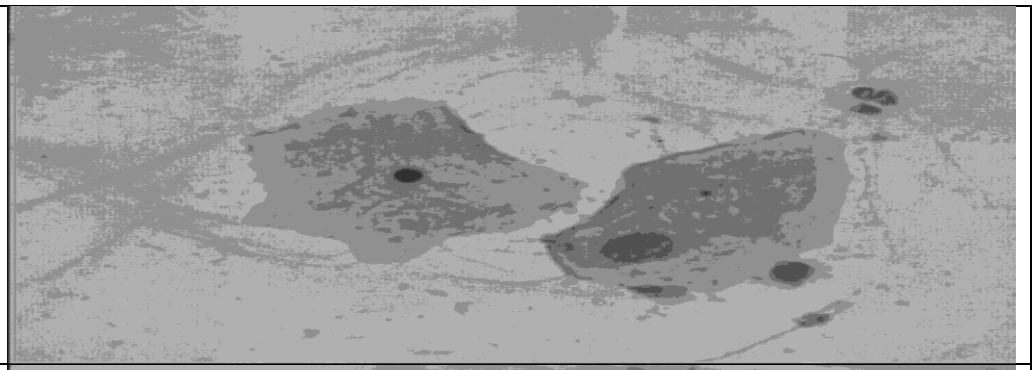
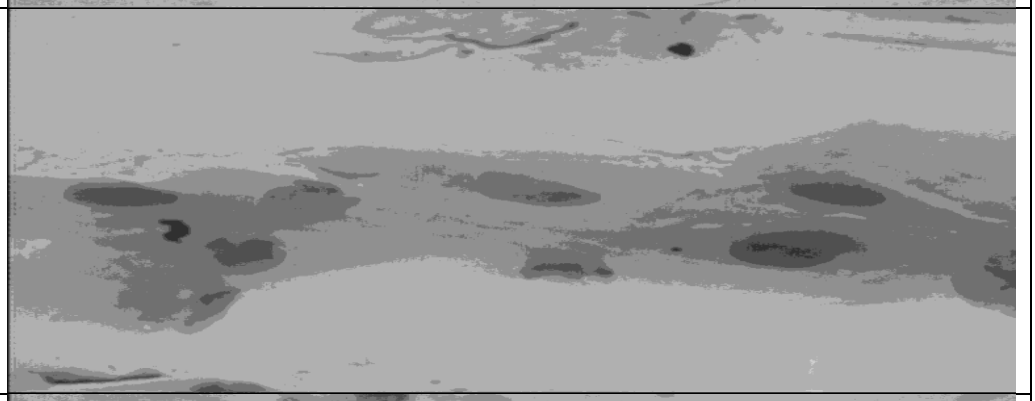
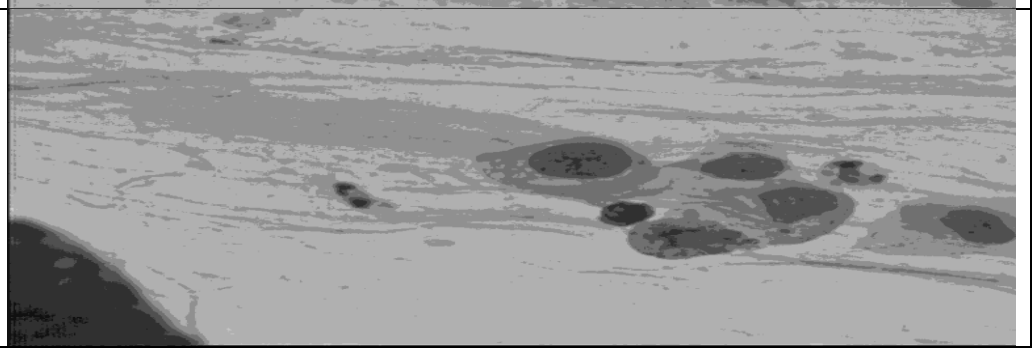
Cell Class	Histogram Equalized Image
Columnar E	
Parabasal SQ E	

Intermediate SQ E	
Superficial SQ E	
Mild NKDP	
Moderate NKDP	

Severe NKDP	

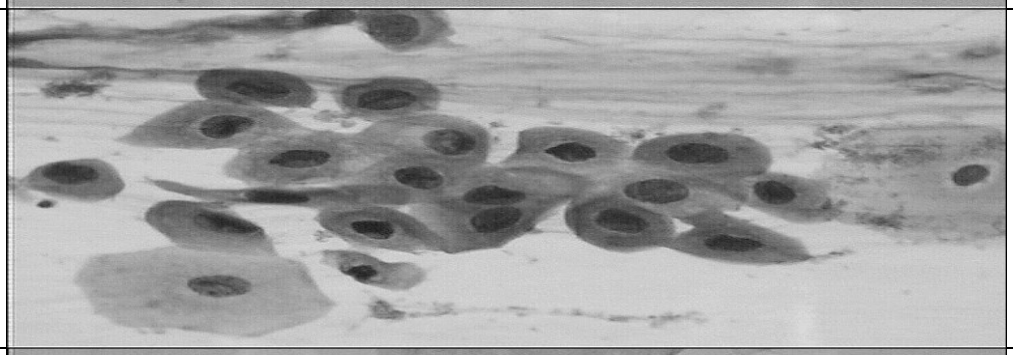
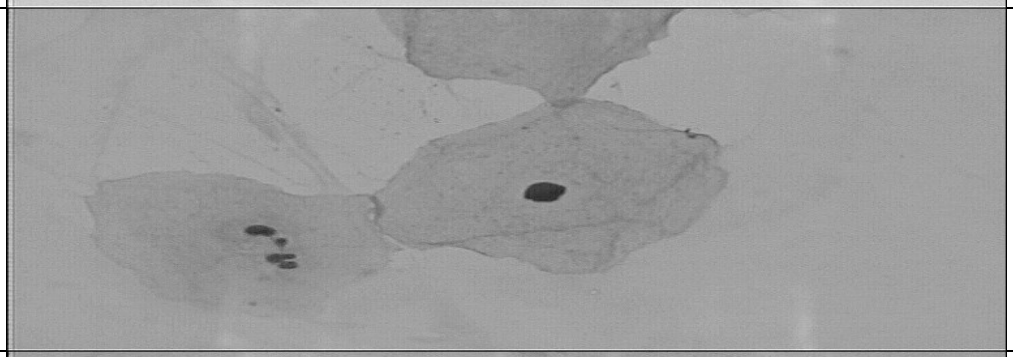
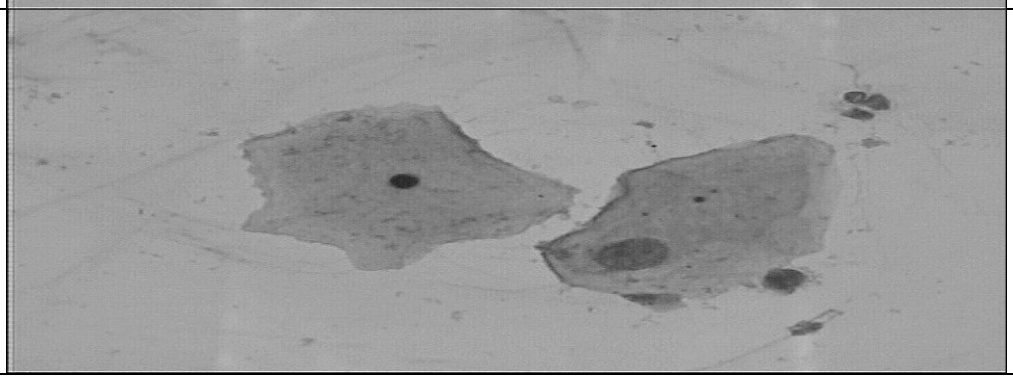
G. Uniform Quantized Images

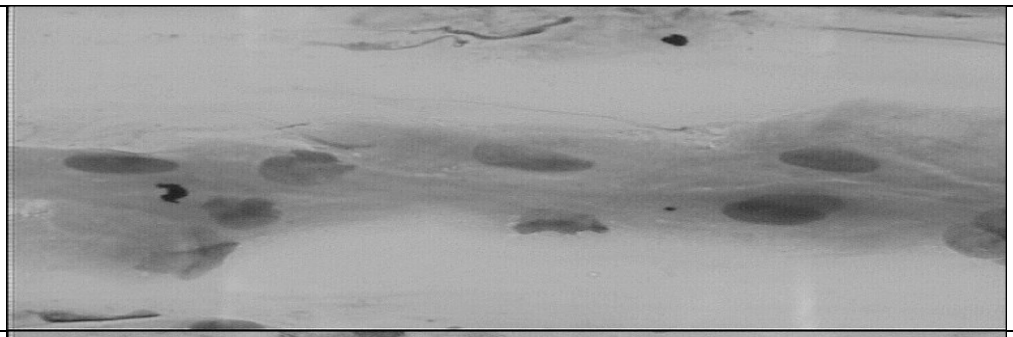
Cell Class	Uniform Quantized Image (8 Levels)
Columnar E	
Parabasal SQ E	
Intermediate SQ E	

Superficial SQ E	
Mild NKDP	
Moderate NKDP	
Severe NKDP	

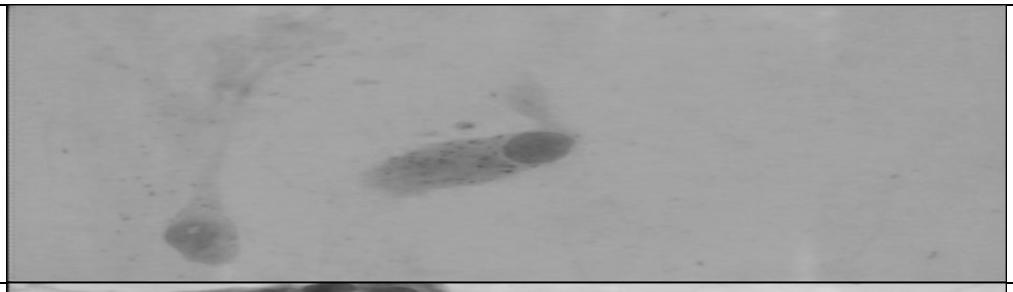
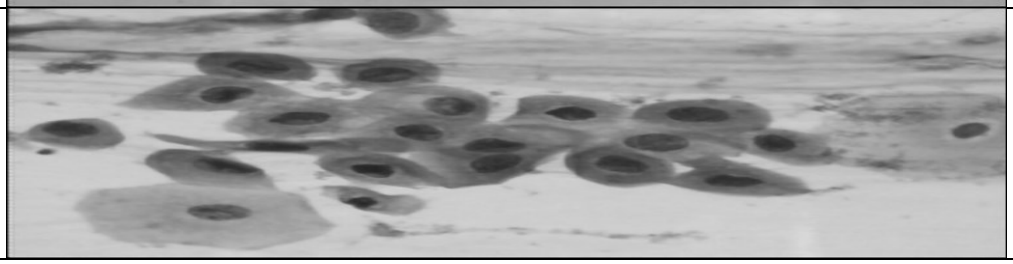
H. Linear Filtered Images

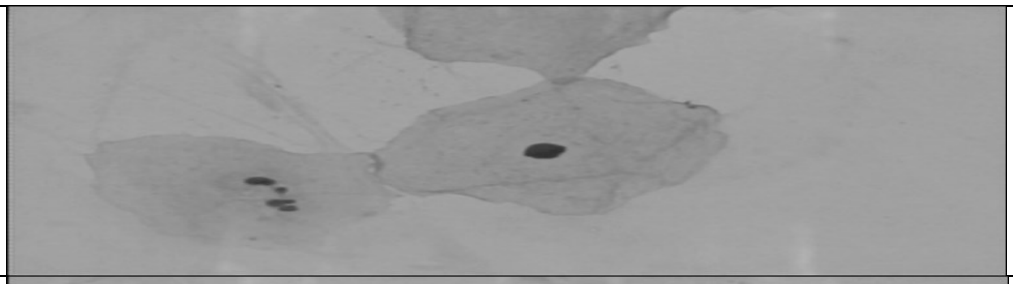
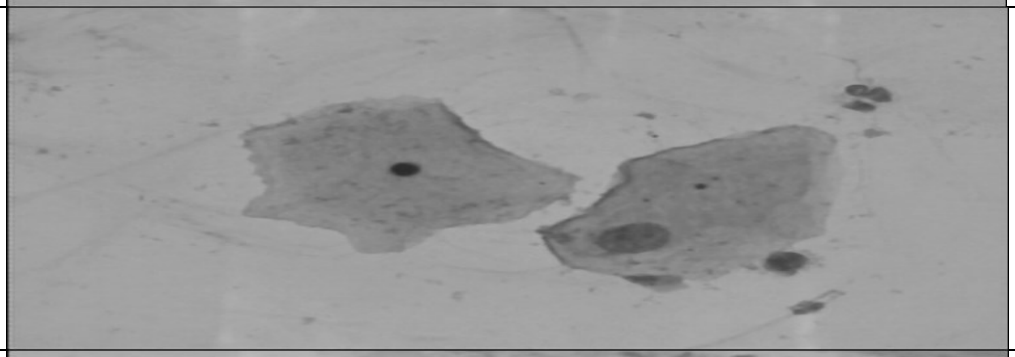
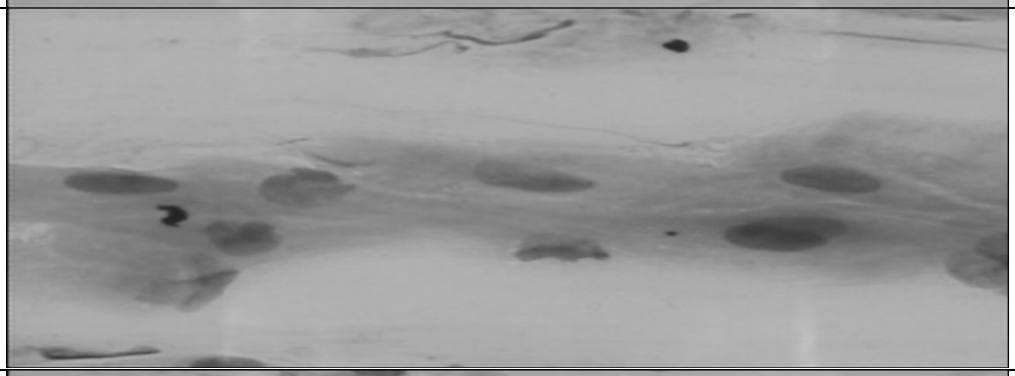
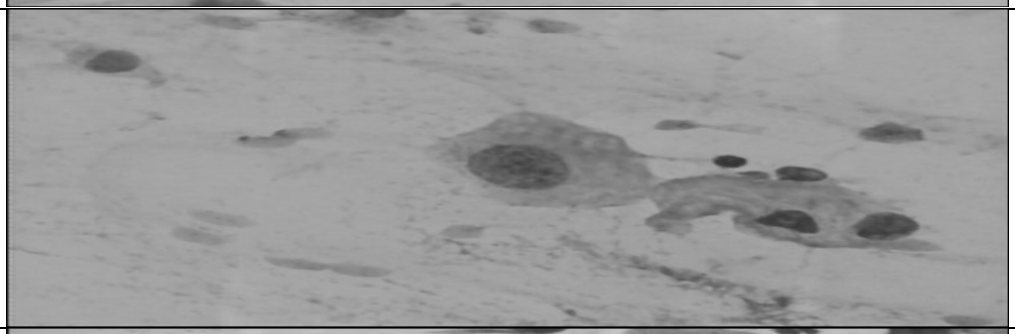
Cell Class	Linear Filtered Image (Sharpen Preset)

Columnar E	
Parabasal SQ E	
Intermediate SQ E	
Superficial SQ E	

Mild NKDP	
Moderate NKDP	
Severe NKDP	

I. Median Filtered Images

Cell Class	Median Filtered Image
Columnar E	
Parabasal SQ E	

Intermediate SQ E	
Superficial SQ E	
Mild NKDP	
Moderate NKDP	
Severe NKDP