Marc Jerrone R. Castro
Activity 10 Written Report

2015-07420

Activity 10 – Blob Analysis

Blobs?

What are blobs?

"Blobs are groups of connected pixels which share a common property (e.g. grayscale value)" (Mallick,2015)

How do we detect these blobs?

*There are a variety of methods in detecting such blobs. One method is through OpenCV's **SimpleBlobDetector** .*

Why do we need to detect them?

In some cases in image processing, there is a large need for analysis of several regions of interest within an image. Such examples are:

- Cell counting
- Particle Analysis
- Granulometry.

As it is increasingly difficult to manually crop out individual entities and perform analysis on them one by one, blob analysis provides a more preferable method for such data as it is capable of doing there repetitive tasks in one go.

SAMPLE BLOB DETECTION

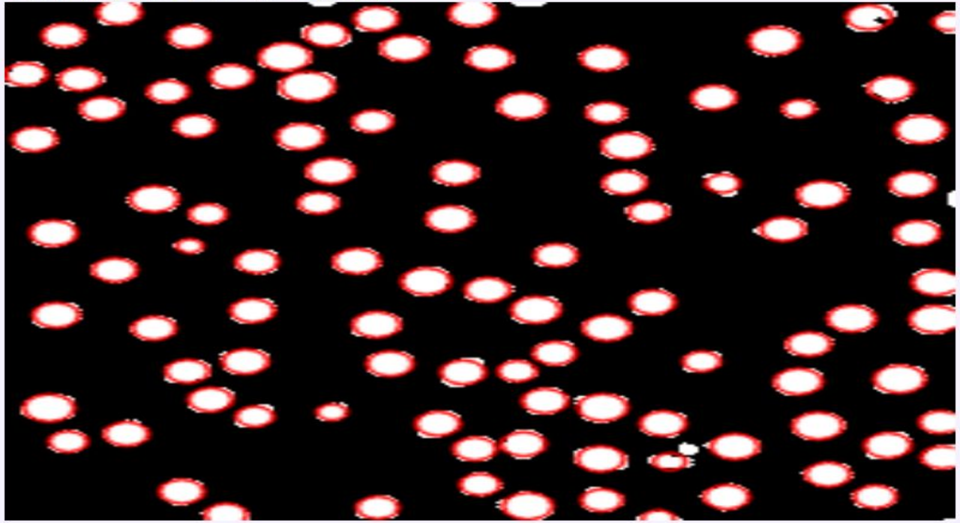


Figure 1.0 Detected blobs given an unprocessed image of blood cells within a 250 x 250 image. The blobs were detected using OpenCV's SimpleBlobDetector.

Input Image

Red Blood Cells

Detecting blobs from an image of blood cells

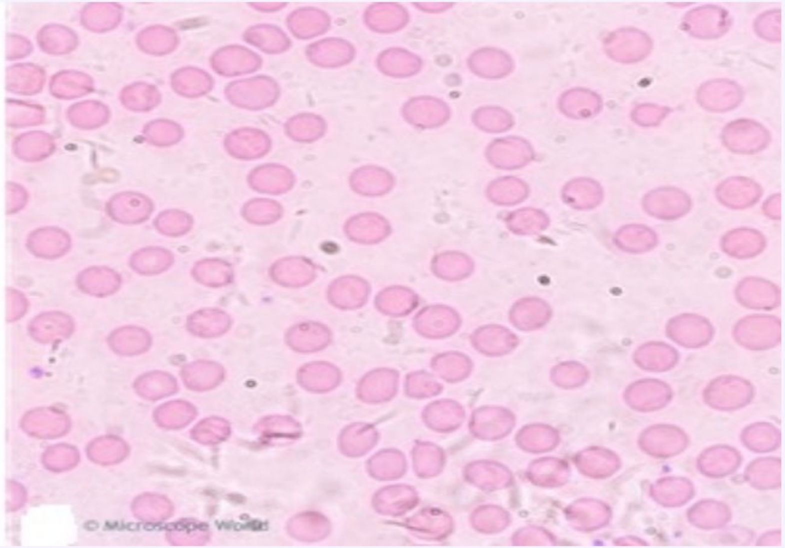


Figure 2 Input image of a red blood cell glass slide. Noticeable artifacts are observed in the image which poses a challenge for blob analysis. This is in addition to the non-uniformity of the pink hue contained in each cell.

Blood Cell Analysis

From our basic biology,we know that red blood cells (RBCs) play a key role in most, if not all, biological processes. As such, performing keen analysis on these cells is particularly important when tracking the general health of one's body.

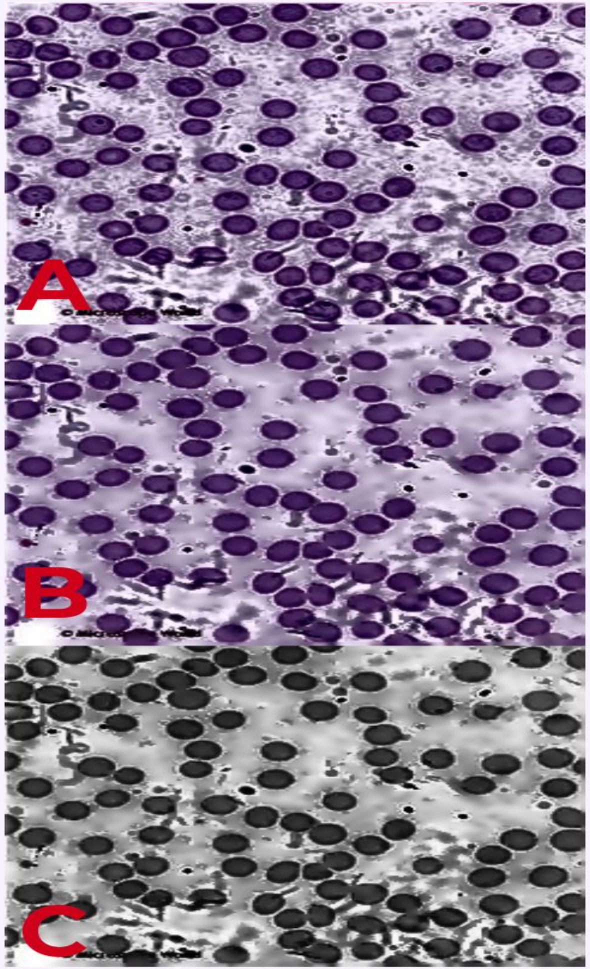
The Problem

These cells come in hundreds, if not by the thousands, for each sample. Thus it is increasingly tedious to quantify and perform qualitative analysis on each.

Random Facts

- There is roughly **25-30 trillion** red blood cells in the human adult body.
- There is a 600:1 ratio between RBCs and White Blood Cells

Pre-processing



Pre-Processing

There are 3 general steps for pre-processing the image:

Why pre-process?

Pre-processing is an essential step for all image processing problems. Through this step, we are able to isolate only relevant information from the input image and generally save precious computing power and time.

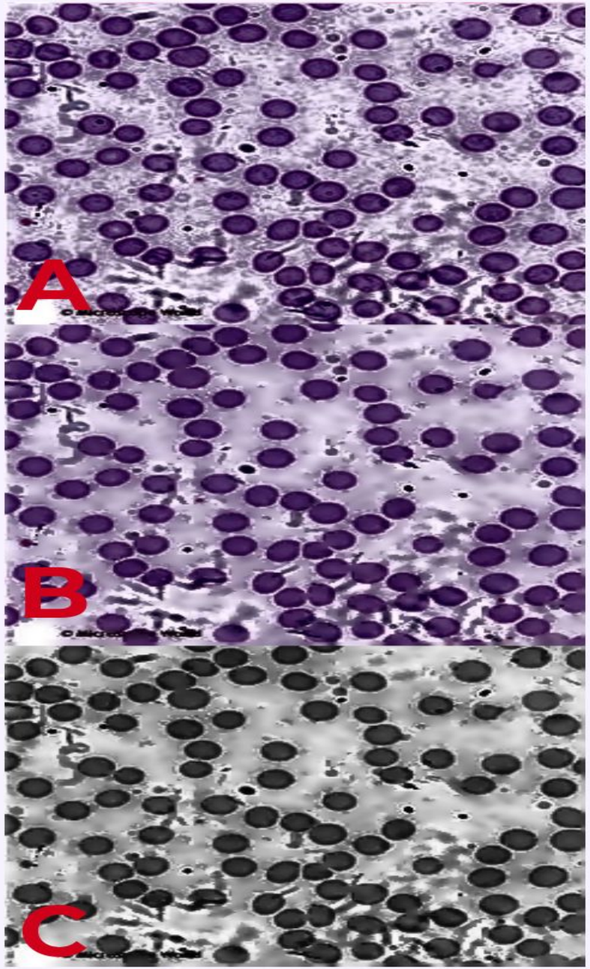
Step 1
Histogram Equalization

This step, exhibited by **image A**, is performed on a YUV colormap version of the input image. This step was performed to increase the contrast between the red blood cells and the background. Using what we learned from previous activities, I used histogram equalization to produce a higher contrast image.

Code Snippet

```
image_yuv = cv2.cvtColor(image_bgr, cv2.COLOR_BGR2YUV)
image_yuv[:, :, 0] = cv2.equalizeHist(image_yuv[:, :, 0])
image_rgb = cv2.cvtColor(image_yuv, cv2.COLOR_YUV2RGB)
cv2_imshow(image_rgb)
```

Pre-processing



Pre-Processing

There are 3 general steps for pre-processing the image:

Why pre-process?

Pre-processing is an essential step for all image processing problems. Through this step, we are able to isolate only relevant information from the input image and generally save precious computing power and time.

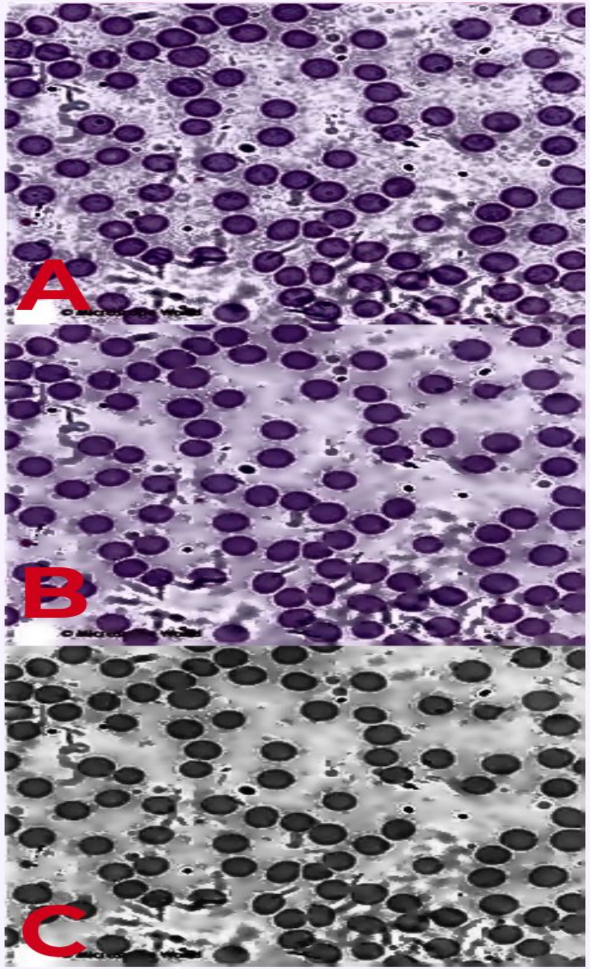
Step 2
Image Blurring

The second step, exhibited by **image B**, utilizes pyramid mean shift filtering to somewhat blur the image such that only a general outline and color distribution remains for each cell. As one may observe, the resulting image reduced the amount of noise that resulted from the previous step and somewhat bled the background such that it is easier to parse out. However, an accuracy trade-off occured for those cells that overlapped and those with non-ROI entities superimposing their boundaries.

Code Snippet

```
shifted = cv2.pyrMeanShiftFiltering(image_rgb, 0, 80)
cv2.imshow(shifted)
```

Pre-processing



Pre-Processing

There are 3 general steps for pre-processing the image:

Why pre-process?

Pre-processing is an essential step for all image processing problems. Through this step, we are able to isolate only relevant information from the input image and generally save precious computing power and time.

Step 3
YUV to Grayscale Conversion

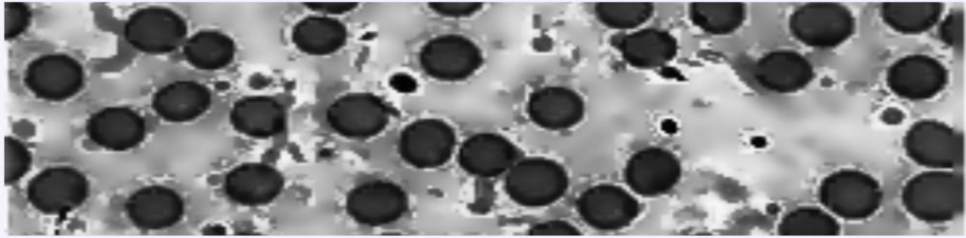
The last step, exhibited by **image C**, converts the blurred YUV image to a Grayscale image. Basically it flattens out the three channel YUV image into a single channel composed of pixel values ranging from 0 to 255 depending on the intensity of gray at each pixel. This makes it easier to segment the image, in addition to significantly reducing processing time.

Code Snippet

```
im2 = cv2.cvtColor(shifted, cv2.COLOR_BGR2GRAY)
cv2_imshow(im2)
```

Image Thresholding

From Grayscale to Binary Images



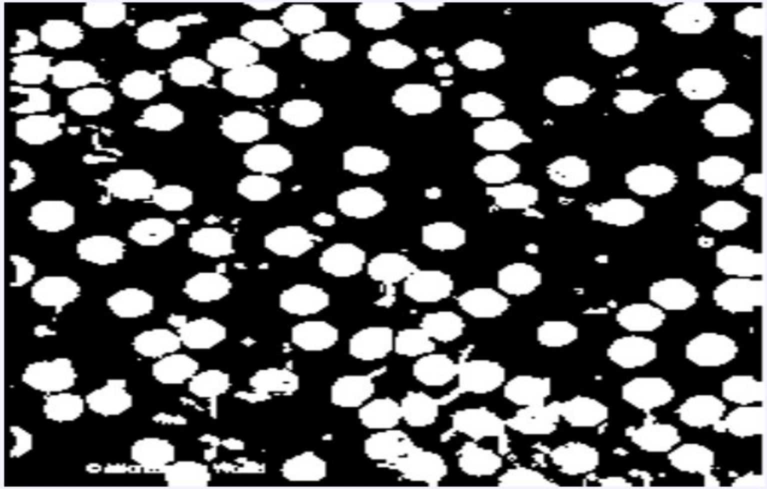


Figure 4 Resulting binary image from image thresholding. The threshold value was determined through trial and error whereas the value that resulted into the most parsed ROIs was selected.

Why do we need to translate it to binary?

To simplify our input image, we perform image thresholding to simplify the values from values ranging from 0 to 255 to only False (below threshold) and True (equal to or above threshold).

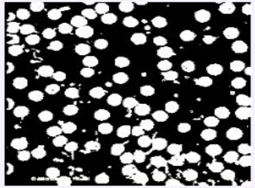
Observations

Although majority of the cells were identified, some artifacts (non-RBCs) were also parsed into our image. Thus we must perform morphological operations to remove such entities and to separate the cells into individual blobs

Code Snippet

```
ret,thresh1 = cv2.threshold(im2,80,255,cv2.THRESH_BINARY_INV)
cv2_imshow(im2)
cv2_imshow(thresh1)
```

Morphological Operations



INPUT BINARY IMAGE

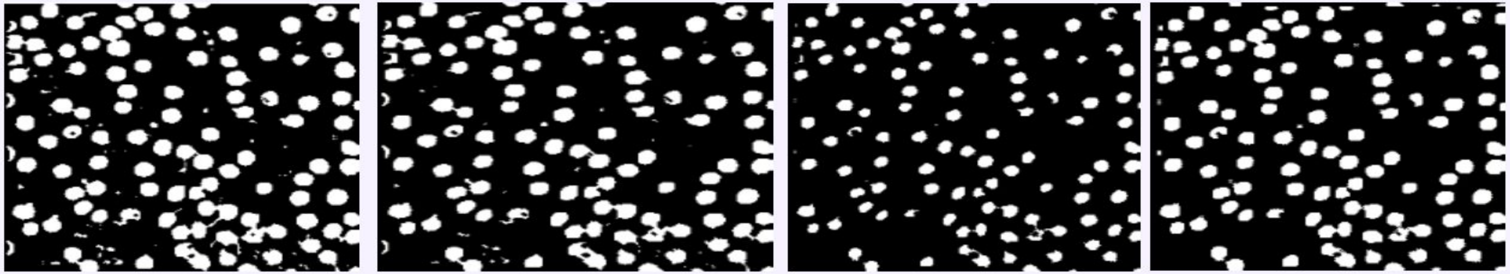


Figure 5 As to reduce the amount of unncessary pixels within the image, I perform morphological operations using a combination or erosion and dilation techniques with a variety of structuring elements such as ellipses, crosses, and diagonals. At each step the amount of white pixels either get reduced (for erosions) or increased (for dilations). The final image resulted to fewer ROIs in comparison to the input image, it also resulted to the separation of blobs which were formerly connected by a minute amount of pixels. As one may notice, a reduced area is observed for the cells which may factor as a source of error when calculating for the area and perimeter. The order of morphological processes is from left to right.

Code Snippets

Structuring Elements I got from Ysabella Ong

```
kernel = np.ones((3,3),np.uint8)
kernel1 = np.ones((4,4),np.uint8)
kernel2 = np.ones((2,2),np.uint8)
ellipse_kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE,(1,1))
ellipse_kernel1 = cv2.getStructuringElement(cv2.MORPH_ELLIPSE,(2,2))
ellipse_kernel2 = cv2.getStructuringElement(cv2.MORPH_ELLIPSE,(3,3))
ellipse_kernel3 = cv2.getStructuringElement(cv2.MORPH_ELLIPSE,(3,3))
ellipse_kernel4 = cv2.getStructuringElement(cv2.MORPH_ELLIPSE,(5,5))
diag = np.zeros([4,4],np.uint8)
diag[2,1] = 1
diag[1,2] = 1
diag2 = np.zeros([4,4],np.uint8)
diag2[1,1] = 1
diag2[2,2] = 1
cross = np.array([[0, 0, 0, 0, 0],
                  [0, 1, 0, 1, 0],
                  [0, 0, 1, 0, 0],
                  [0, 1, 0, 1, 0],
                  [0, 0, 0, 0, 0]], np.uint8)
```

Morphological Operations

```
im_erode = cv2.erode(thresh1, ellipse_kernel3, iterations = 1)
im_erode1 = cv2.erode(im_erode, diag2, iterations = 1)
im_erode2 = cv2.erode(im_erode1, cross, iterations = 1)
im_dilate = cv2.dilate(im_erode2, cross, iterations = 1)
```

Watershed

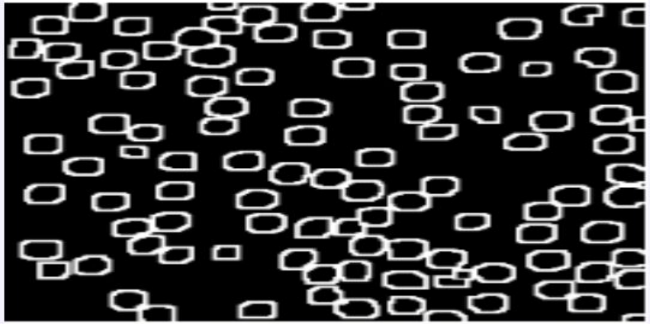


Figure 6 Range of pixels marked as "*unknown*" which may or may not be a part of their corresponding ROIs. Also considered as the difference between known foreground and known background pixels.

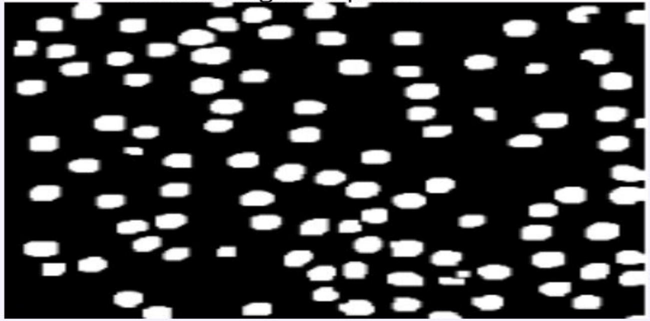


Figure 7 Range of pixels marked as known pixels that are believed to be parts of ROI/cells.

Watershed Operations (Rosebrock , 2015)

What is the watershed operation?

The watershed algorithm is a valuable step when extracting *touching* or *overlapping* objects. It utilizes the concept of extracting know portions of an ROI to determine individual entities within an object.

How does it work?

It basically imposes the use of morphological operation such as *OPEN* and *dilation* to extract the foreground and background of ROIs. From there it determines a known portion of the ROI and takes it as an input for isolating grouped components through *cv2.connectedComponents* which parses all similar pixels which are neighbors to each other and groups them into a cluster.

Code Snippet

```
# noise removal
kernels = np.ones((3,3),np.uint8)
opening = cv2.morphologyEx(im_dilate,cv2.MORPH_OPEN,kernels, iterations = 2)

# sure background area
sure_bg = cv2.dilate(opening,kernels,iterations=2)

# Finding sure foreground area
dist_transform = cv2.distanceTransform(opening,cv2.DIST_L2,3)
ret, sure_fg = cv2.threshold(dist_transform,0.1*dist_transform.max(),255,0)
# Finding unknown region

sure_fg = np.uint8(sure_fg)
unknown = cv2.subtract(sure_bg,sure_fg)
```

Watershed

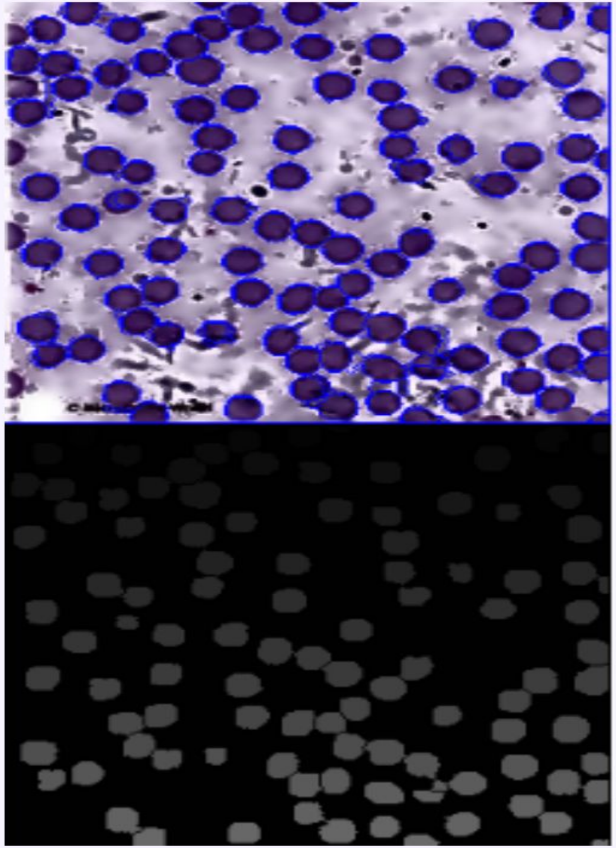


Figure 8 The top image is an overlay of the input blurred YUV image and the contours of each detected ROI. The bottom image is representation of all detected ROIs using the watershed algorithm.

Watershed Operations (Rosebrock , 2015)

What is the watershed operation?

The watershed algorithm is a valuable step when extracting *touching* or *overlapping* objects. It utilizes the concept of extracting know portions of an ROI to determine individual entities within an object.

How does it work?

It basically imposes the use of morphological operation such as *OPEN* and *dilation to* extract the foreground and background of ROIs. From there it determines a known portion of the ROI and takes it as an input for isolating grouped components through *cv2.connectedComponents* which parses all similar pixels which are neighbors to each other and groups them into a cluster.

Code Snippet

```
# Marker labelling
ret, markers = cv2.connectedComponents(sure_fg)
# Add one to all labels so that sure background is not 0, but 1
markers = markers+1
# Now, mark the region of unknown with zero
markers[unknown==255] = 0
```

```
markers = cv2.watershed(shifted,markers)
shifted[markers == -1] = [255,0,0]
markers
```

Simplifying

From Gradient to Binary
From Binary to Morphing

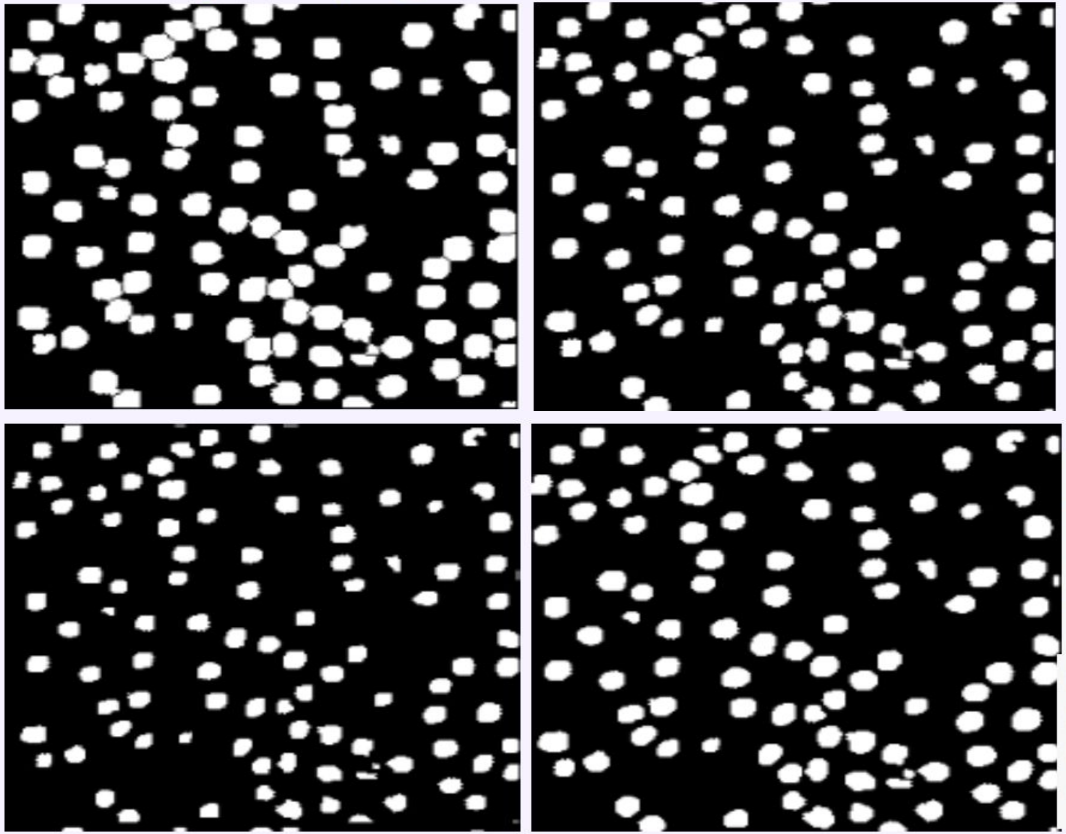


Figure 9

(Top-Left) Converted watershed ROIs to binary using thresholding. (Top-Right) Intersection of watershed ROI pixels and final image from morphological processes which further reduced overlap between cells.(Bottom-Left) Erosion of image to reduce overlapping pixels. (Bottom-Right) Dilation of image to restore some of the reduced areas on the image.

Code Snippets

```
node = markers
node[markers ==-1] = 0
node[markers == 1] = 0
node[markers != 0] = 255
segments = im_dilate.astype('uint8')
segments[im_dilate != node] = 0
```

```
erode1 = cv2.erode(segments,ellipse_kernel2)
cv2_imshow(erode1)
dilate1 = cv2.dilate(erode1,ellipse_kernel2)
cv2_imshow(dilate1)
```

Blob Detections

```
# Setup SimpleBlobDetector parameters.
params = cv2.SimpleBlobDetector_Params()

# Filter by Color
params.filterByColor = True
params.blobColor = 255

# Filter by Area.
params.filterByArea = True
params.minArea = 18
params.maxArea = 250

# Filter by Circularity
params.filterByCircularity = True
params.minCircularity = 0

# Filter by Convexity
params.filterByConvexity = True
params.minConvexity = 0

# Filter by Inertia
params.filterByInertia = True
params.minInertiaRatio = 0
```

Blob Detection Parameters

To isolate only specific cells within the image, we instilled the following parameters.

255
Color

This is to specify that the blobs of interest are those pixels with a value of 255.

18-250
Area

Range of area of pixel clusters to be considered as blobs.

0
Circularity
Convexity
Inertia

Since most noise have been parsed out, we are free to set minimal dependence on these three parameters as only ROIs are left within the image

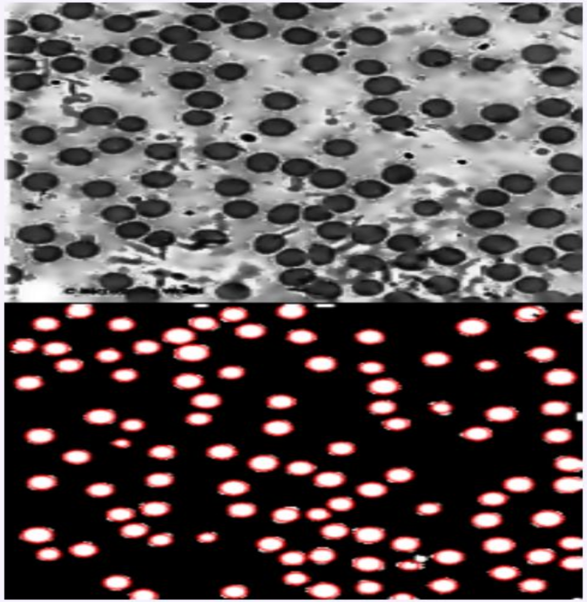
Blobs Detected

Detecting the RBC blobs

With all pre-processing and cleaning, we can know perform blob detection using a ***cv2.SimpleBlobDetector*** . The resulting blob detection is shown on the right whereas its parameters where determined the total number of blobs detected is

100/105

manually counted blobs



The remaining blobs may have been parsed out as they might have been positioned to the borders of the image and through various morphological processes - parsed out. Some blobs may have also been loss through the parameters we set from thresholds, area range, and by simple lack of pink pigment in the original image.

Code Snippets

```
detector = cv2.SimpleBlobDetector_create(params)
keypoints = detector.detect(dilate1)
im_with_keypoints = cv2.drawKeypoints(dilate1, keypoints, np.array([]), (0,0,255), cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
im_with_keypoints2 = cv2.drawKeypoints(dilate1, keypoints, np.array([]), (0,0,255))
cv2_imshow(im2)
cv2_imshow(im_with_keypoints)
cv2_imshow(im_with_keypoints2)
```

Detected Properties

Code Snippets

```
label, N = sm.label(dilate1, background=0, return_num=True)
reg = sm.regionprops(label,dilate1)

area = []
eccen = []
mal = []
per= []
for i in range(N):
  if reg[i].area > 18:
    area.append(reg[i].area)
    eccen.append(reg[i].eccentricity)
    mal.append(reg[i].major_axis_length)
    per.append(reg[i].perimeter)
  else:
    continue

print('AREA = ',np.mean(area),'+/-',np.std(area))
print('Eccentricity = ',np.mean(eccen),'+/-',np.std(eccen))
print('Major Axis Length = ',np.mean(mal),'+/-',np.std(mal))
print('Perimeter = ',np.mean(per),'+/-',np.std(per))
```

Blob Properties determined through scikit-image

Using scikit-image's regionprops, the following properties were determined

MEAN VALUES

Area	113.16 +/- 27.91
Eccentricity	0.51 +/- 0.13
Major Axis Length	12.98 +/- 1.57
Perimeter	37.32 +/- 5.55

References

REFERENCES

- https://www.pyimagesearch.com/2015/11/02/watershed-opencv/
- https://www.learnopencv.com/blob-detection-using-opencv-python-c/
- https://web.mit.edu/scicom/www/blood.html

Self Evaluation

QOP : 5/5

TC: 5/5

Initiative :2/2