Cell Counter

A CIS project

Student:	Leon Brandes
Instructor :	Roman Sandler (Lab. Head: Michael Lindenbaum)
Initiator:	Dr. Nahum Rozenberg
Time:	April 2010 (end) 2008 (Beginning)

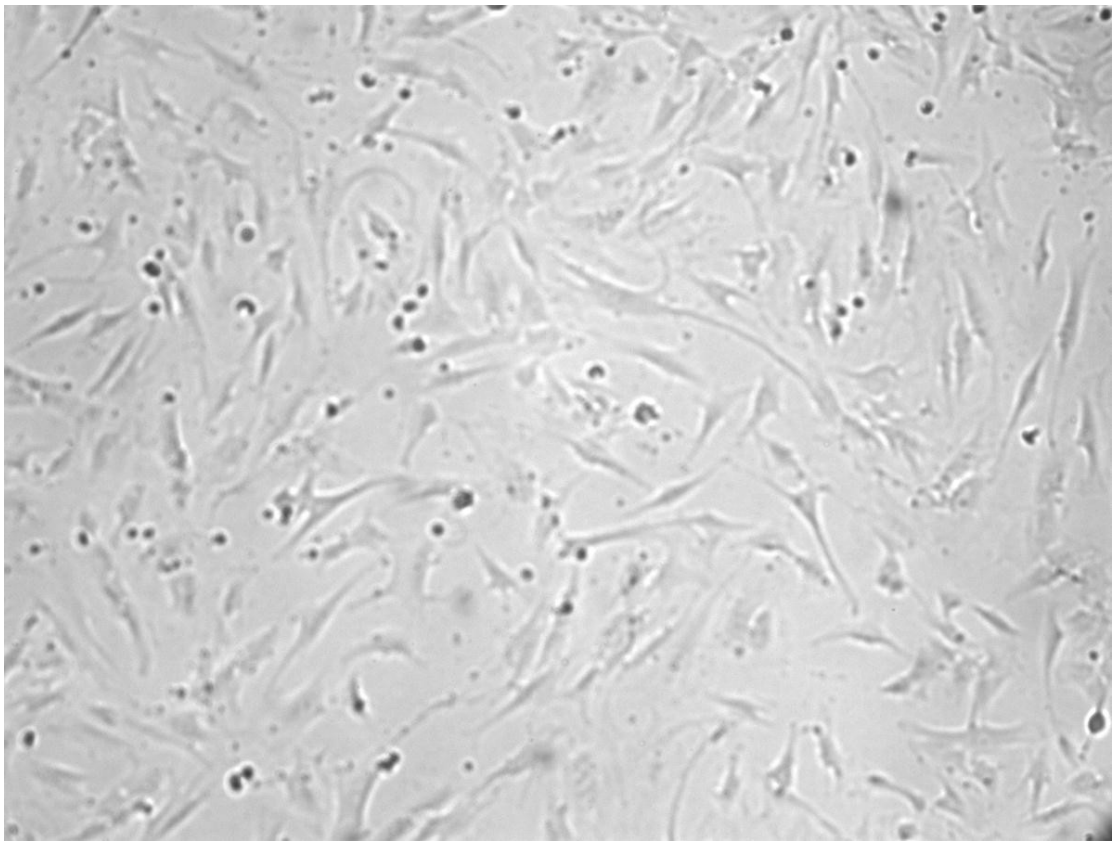
Cell counter project

1. Introduction

a. Problem description

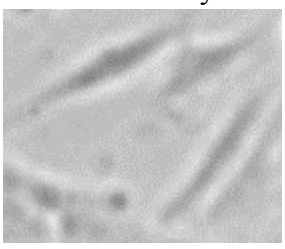
Some routine medical tasks may be automated to increase their efficiency and diminish their dependency on the human factor. One of such tasks is cell counting. In medical research hundreds of gray-level cell images are taken in order to count them manually. In a typical image, e.g. *image 1*, there are tens of cells. Nowadays a researcher has to count these cells manually, which is a monotonous time consuming procedure. A computerized automatic cell counter is essential to simplify and standardize this task.

Image 1

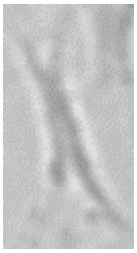


An example of input image

At the first sight the task is challenging. As one may see in *image 1*, the cells are scattered in different sizes, shapes, angles and gray-level intensities. They also may partially overlay each other.

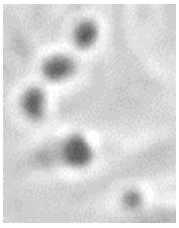


3 cells

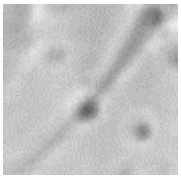


2 overlaying cells

The images contain noises which make the task even more complex. The most noticeable noise phenomenon is a dark, mostly circular, area surrounded by a bright border. We denominate them *'noise blobs'*.



5 noise blobs



2 noise blobs overlay a cell

A noise blob may be somewhat similar to a cell and it can overlay a cell. That makes the detection and distinction process very difficult.

b. Goal of the project

We aim to introduce a cell detection algorithm which counts the cells in a given image "accurately enough". The algorithm should receive a gray-level image and output the estimated number of cells. Ideally, the error should be less than 5% from the number counted by a human operator.

c. Brief summary of this work

This work introduces a fast and simple method for cell counting. The obtained accuracy is around 8%, which is yet to achieve the declared goal. However, this method allows a fast assessment of the cell numbers and if needed, may be succeeded with more precise cell detection algorithm based on the obtained results for specific locations.

Shape detection methods could also be considered by studying the shapes dissimilarities of noise blobs and cells.

2. The proposed approach

a. Brief description of the approach principles.

Looking at example images, e.g. *image 1*, one can see that the noise blobs and the cells are darker than the background. That tempts us to base our initial detection process on intensity levels. But how can we distinguish between the noise blobs and the cells?

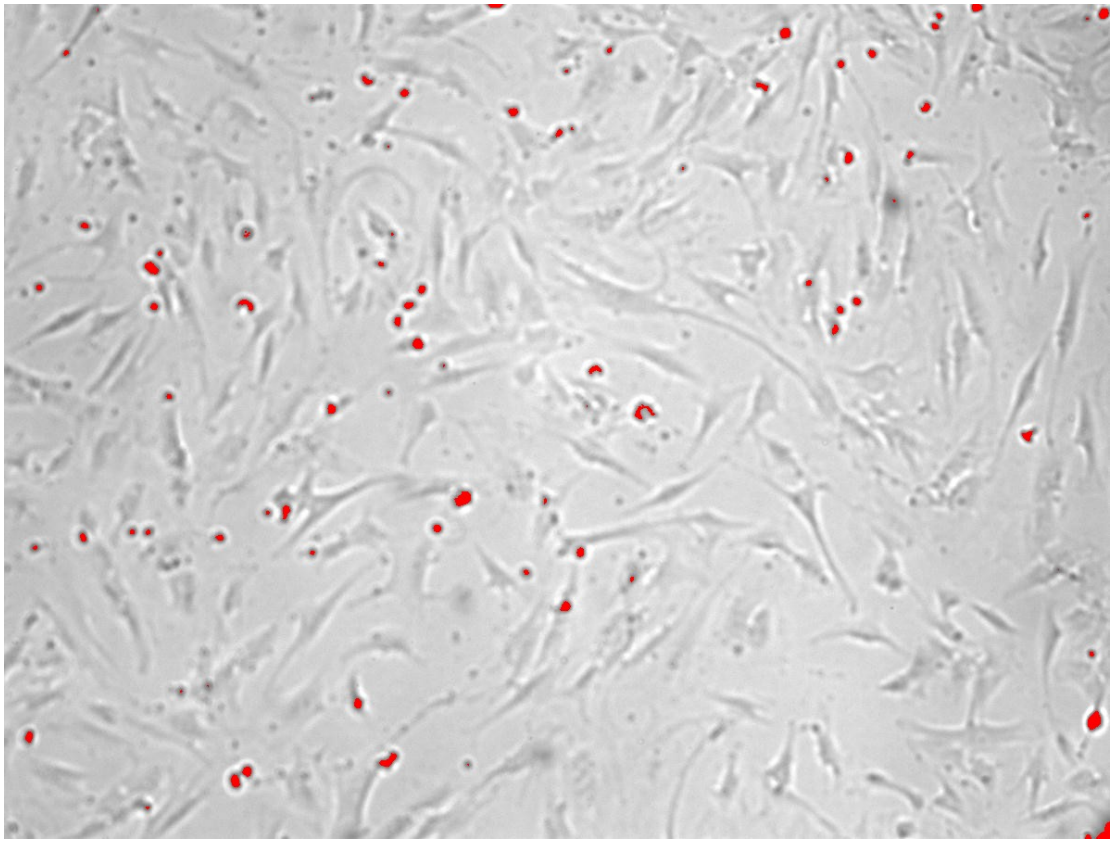
Looking at the images one can notice that noise blobs and cells differ in 2 properties – intensity level and shape. The noise blobs tend to be darker and while their shape is circular, the cells tend to be a very curved elliptic shaped, even lined.

Computing the shape of a dark object is very difficult, all the more considering the dark objects are overlaying each other and have different angles and shapes. It's much easier to exploit the intensity difference between the noise blobs and the cells.

The noise blobs are the darkest objects in the image. That means our detection process is able to filter the noise blobs out and leave us with a "clean" image so that our algorithm can count the cells in an "ideal environment".

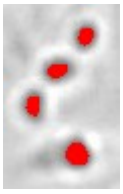
By experiments and looking at images, we have noticed that the level of general brightness is not constant. For example, in *image 1*, the center is brighter than the sides and corners. Our noise blob sifting is based on intensity level filtering, thus we must take the local general brightness level into account. For example, the grey level threshold of noise blobs in the center of *image 1* is higher than in its corners.

Image 2



Pixels which are darker than the locally-adapted threshold are marked in red.

Image 2 shows that the simple filtering which is based on intensity level recognizes the very most of the noise blobs. However, it doesn't cover them well.



Simple grey level filtering result

We see in the image that around the red pixels there are still dark pixels. They are part of the noise blob but they are not dark enough to be marked by the intensity level based filter. These are dangerous pixels because their intensity level is similar to the intensity levels of the cells. We want to cover these pixels as well because we want to have a "clean" image after wiping out, ideally, all the pixels which belong to noise blobs.

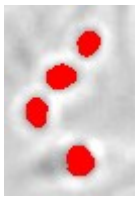
We want the marker to go at least as far as the border between the dark part of the noise blob and the bright area which surrounds it.

We need an edge detector to mark the borders for us. We chose to use Canny's edge detection algorithm[1].

[1] Canny, J., *A Computational Approach To Edge Detection*, IEEE Trans. Pattern Analysis and Machine Intelligence, 8:679–714, 1986.

The blue pixels are Canny detector marks

Our marker picks a pixel in the red area, which is the result of the simple threshold-based filtering marker, and flood-fills[2] from there until it reaches Canny's marked pixels. It yields better noise blobs coverage.

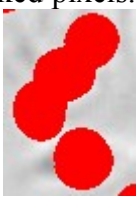


Flood-fill marking results in better coverage

One still sees the bright ring which surrounds the marked dark area. Although these are bright pixels which cannot confuse the detection process with the dark cells, we want our image to be "clean" of pixels which belong to noise blobs.

Empiricism showed us that whatever the surface of the noise blob is, the width of the white ring is similar. Utilizing it we apply a simple dilation operator on the marked images.

We received a satisfying-enough noise blob coverage. The next phase of the detection process can assume a clean image, because it takes into consideration only unmarked pixels.

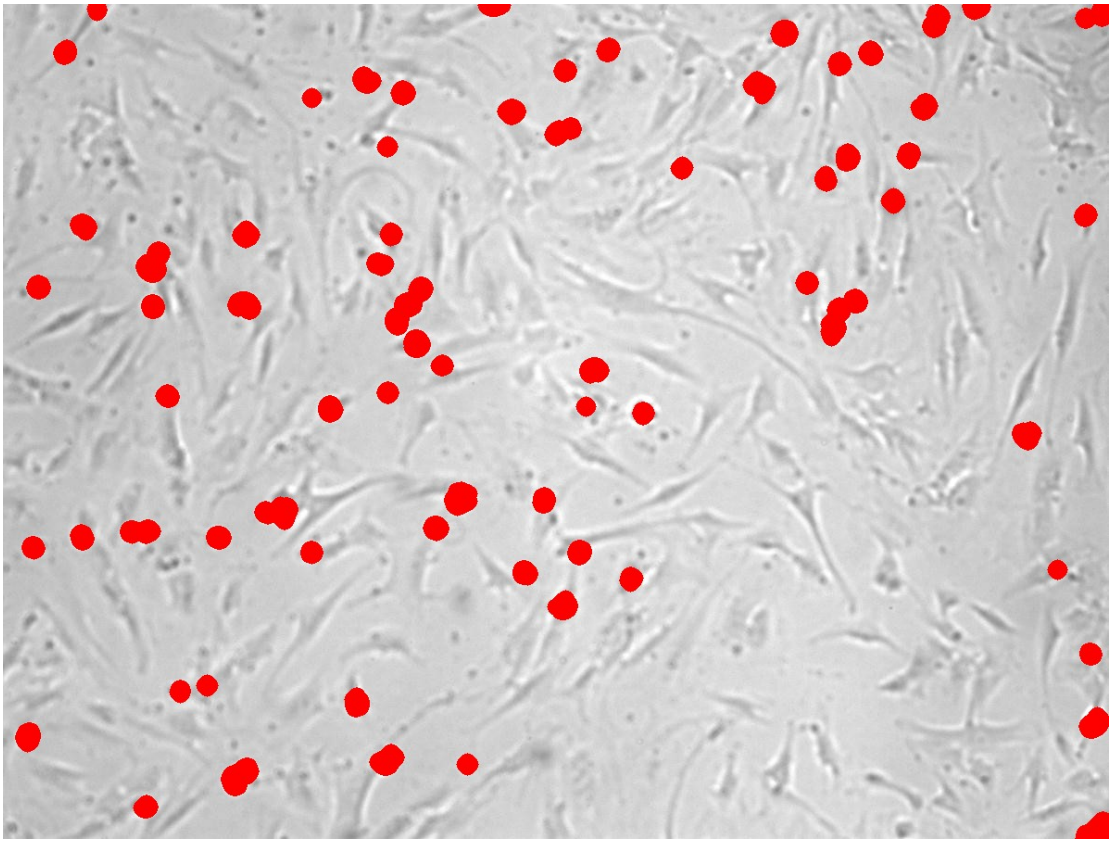


Marking result after dilation

That was the 1st phase of our detection process algorithm. Its purpose is to wipe the noise blobs out and provide the next phase with a "perfect" working environment.

Image 3

[2] http://en.wikipedia.org/wiki/Flood_fill



Marking result of the 1st phase

After sifting the noise blobs, most of the dark objects of the image are the cells. Does it remind something? Before the 1st phase the noise blobs were the darkest objects. Before the 2nd phase, these are the cells we need to detect. Thus, shall we apply the same mechanism yet again, this time ignoring the marked pixels?

We can do that, but we cannot ignore the difference between noise blobs and cells. Cells are more complex.

An example of a cell with a broken open boundary

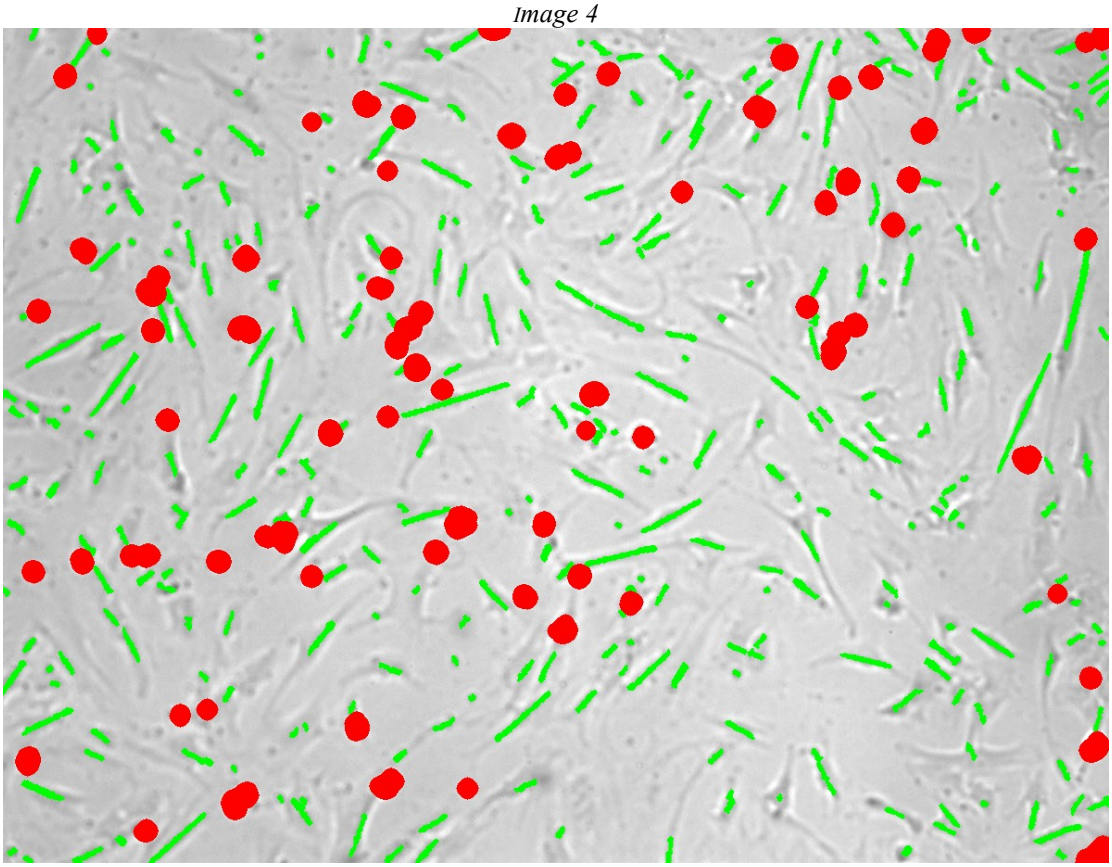
Since the grey level difference between the cells and the background is smaller than between the noise blobs and the background, Canny's edges are broken open. That causes the flood-fill algorithm to "slip out" of the cell which results in a very poor cell coverage.

How can we avoid these "slip outs"? We want the flood-fill algorithm to continue spreading **along the cell** even if there are some holes in its sides. That means we want a prioritized flood-fill spreading, i.e. we want to prioritize the pixels along the cell over the pixels which are perpendicular to the cell. By saying "along the cell" we imply the cell has a direction, a slope. But how do we compute its slope? We don't even know this cell, because we're only about to detect it!

We can only estimate its slope. We can do it using the simple initial information we have about the cell – the threshold-based marked

pixels. With these pixels we compute the "center of mass" pixel, i.e. the $average(x,y)$, then we take into consideration the other marked pixels relatively to the "center of mass", to compute the estimated slope.
Prioritizing the flood-fill algorithm doesn't prevent the flood-fill from spreading all over the image through the boundary holes. We have to take care of it. The broken open boundary issue occurs also in noise blobs, though to a much smaller extent. We want the flood-fill marking to reach Canny's edges, but since we probably have holes, we don't demand all of the flood-filling pixels to touch the boundary. We demand that at least certain percentage out of all the flood-filling pixels sits on the boundary, and then we stop the flood-fill spreading.

Image 4



Red – phase 1 result. Green – phase 2 result.

The marking hit ration of the cells is obviously far from being perfect. A discussion of remaining problems will be held in *2f*.

b. Reasons to choose the approach

The algorithm flow we offer is modular; it allows us to tune or improve each of the 2 phases independently, even though 1st phase result effects 2nd phase as the latter avoids area marked by the former. The separation to phases simplifies the whole process, since the 2nd phase assumes the dark areas may belong to cells only because the 1st phase already marked the noise blobs them and the 2nd phase avoid these marked areas. The 1st phase allows the 2nd phase to work in a much more convenience environment.
The threshold test used to identify pixels suspected to belong to a noise

blob or a cell is a simple comparison operation which focuses the workload on the dark areas.
It was also the simplest way to identify noise blobs quickly because they tend to be very dark compared to their surroundings.
The shapes of the cells is very varied, they overlay and are overlaid and the length-width proportion also varies, this it was impractical for us to base the detection process on shape.

c. The proposed cell counting method.

Our cell counting algorithm clearly fails to achieve the accuracy needed, as it reports on a number of cells which is several times the real number of cells. However, our RoC curves, as can be seen in sub-chapter *4e*, suggest the ratios between the number of False Negative marks and False Positive marks to the number of cells counted by lour algorithm tend to be similar. We utilized to implement an estimation method which produces much better results. The method is described in sub-chapter *4f*.

d. Problems

We encountered several problems developing the cells counter.

The broken open Canny's edges occurred in both phases, i.e. in noise blobs and in cells. The flood-fill algorithm "slips out" of the dark element (which can be either a noise blob or a cell) and stops only after covering the whole image. The grey level gap between the cell and the surrounding may be very blurred so the Canny boundary often tends to be broken open.
Then noise blobs usually have 2 borders; one between the dark part and the bright ring which surrounds it, and another one between the ring and the background. We initially wanted to perform a 2nd flood-filling in the 1st phase, so the flood-fill keeps spreading until the 2nd border, thus covering and marking the noise blob accurately. Unfortunately, the 2nd border is considerably more broken and open so we opted for another solution.

Our detector is based on threshold filtering. Looking at the images one can see that the grey level of the background and the dark elements vary. It varies inside an image and it also varies between images; some images are brighter than others.

In the 2nd phase the initial threshold-based marker marks a lot of very small areas, some of them consist of just a few pixels. Areas like these are very unlikely to form a cell and we didn't want our detector to count them as one.

e. Solutions found

The constant values in this chapter were chosen empirically; We tried different values and analyzed the results to get the best values.

The broken open Canny's edges are solved by enforcing 2 flood-filling stopping criteria's – ratio and size. The ratio is the number of flood-filling pixels sitting on the boundary divided by the overall number of flood-filling pixels. Ratio of 0.65 in 1st phase and 0.73 in 2nd phase are good values. The size is the surface of current flood-fill, i.e. the number of pixels the current flood-fill run already marked. Looking at images it's obvious that there is a connection between the size of the dark element and the number of pixels marked by the initial threshold-based marker. In 1st phase the area size upper threshold is 6 times the number of marked pixels (by the threshold-based marker). In 2nd phase it 2 times. It makes sense when looking at cells and noise blobs; in cells the dark area takes most of the cell's surface, while the noise blobs tend to have a wide bright ring around the dark area.
When the flood-fill algorithm reaches either of the stopping criteria's, it stops spreading.
In the 1st phase we wanted to cover the noise blobs well by flood-filling further until the 2nd border, but the border is too fragile, so we opted to apply the dilation operator. The ring's width proved to be very similar regardless the surface of the noise blob. The usage of dilation operator proved to be fast and accurate enough.

We couldn't use a constant threshold in the threshold-based marker all over the image, so we decided to divide the image to blocks. Each block is assigned a constant threshold value. The size of each block is 50X50 and the formula to compute its value is described in *2c*. 50X50 are small enough to provide us with a good brightness level adaptation. The blocks in the center of the image usually have higher threshold values than the corner blocks.

We used simple filtering to eliminate tiny marked areas in the 2nd phase, because they are unlikely to be cells. We decided the algorithm ignores all areas whose surface is not bigger than 45 pixels. It spared the algorithm from many local dark phenomena's which are not cells.

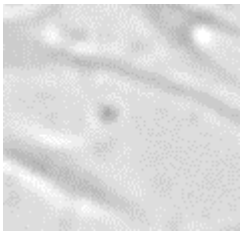
f. Remaining problems

Unfortunately we still have major problems.

The image consists of cells which are almost as bright as the background.

An example of a bright cell.

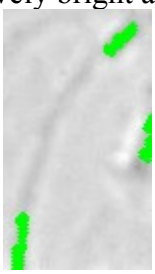
Our detector misses such cells because it is based on grey level differences.
For similar reasons our algorithm misses noise blobs or confuses them with cells. A noise blob which is too bright for the 1st phase but dark enough for the 2nd phase will be considered as a cell (unless it's too small).



An example of a noise blob which is considered to be a cell in our detector.

Obviously, the opposite case is also possible – a relatively dark cell may be marked as a noise blob in the 1st phase.

Our detector also may mark 2 distinct areas which belong to the same cell. It can happen in long cells, especially if they consist of multiple dark areas separated by relatively bright areas.



A cell which is marked twice

Our detector counts such cells as multiple cells, depends on the number of marks.

Most of the problems occur because our detector starts its work with a grey-level based filtering.

3. Implementation

a. Adaptive threshold

Algorithm 1:

Input: Image I, Threshold T, holes matrix H.
H is a matrix of the same dimension as I. $H(x,y)==255$ if (x,y) is a marked pixels which shouldn't be taken into consideration.

i. Divide the image into 50X50 blocks $B_{i,j}$.

ii. Find the average gray-level value of each block $M_{i,j} = <B_{i,j}>$. Take into consideration only pixels (x,y) for which $H(x,y)! =255$.

iii. Mark all the pixels which gray-level is under the adaptive threshold $I(x,y) < M_{i,j}-+T$.

The threshold is locally adapted; the image is divided into blocks, each block has its own threshold. The threshold is basically a pixel average of the block minus a constant value given by the user.

b. Flood fill algorithm,

Algorithm 2:

Input: Image *I*, surface threshold *ST*, *(row,col)*.

i. Create *(x,y)*'s container and put there *(row,col)*.
ii. Let *n* be the size of the container, *m* be 0 and *s* be 0.
iii. For 1 to *n*
    1. Pop the 1st pixel from the container to be *(x,y)*.
    2. If *(x,y)* sits on Canny edge or *I*'s boundary, increment *m*.
    3. If *(x,y)* sits on Canny edge go to next iteration.
    4. for each 4-neighbour of *(x,y)*, *(nx,ny)*
        a. If *(nx,ny)* inside *I* and unmarked, add it to the container and mark it.
iv. Let *n* be the size of the container**.**
v. If $m/(m + n) > 0.73$ then stop.
vi. $s = s + n$.
vii. If $s>ST$ then stop.
viii. Go to (iii).

In steps (v) and (vii) we have the stopping criteria's we mentioned in the previous chapter.

c. Noise detection algorithm.

Input: Image *I*, Noise threshold *NT*, Cell threshold CT, Cell surface threshold *S*.

i. Call **Algorithm 1** with image *I*, threshold *NT* and no holes matrix.
ii. Label 8-neighbourhood connected components in the resulting marked matrix of (i).
iii. For each connected component do:
    1. Find the mass center *(rc,cc)* which is (mean row, mean column).
    2. Call **Algorithm 2** with 6 times the surface of the component as a surface threshold, and *(rc,cc)*.
iv. Apply the dilation operator on the result of (iii).

This is the 1st phase of our detector.

d. Flood fill algorithm, directional flow modification.

It's similar to algorithm 3, but cell-slope-oriented. The container here keeps complex elements; each element contains pixel coordinate *(x,y)* and a weight value.

Algorithm 3:

Input: Image *I*, surface threshold *ST*, cell angle *CA*, *(row,col)*.

i. Create a container and put there *(row,col)* and any value as weight.
ii. Let *n* be the size of the container, *m* be 0 and *s* be 0.
iii. For 1 to *n*
    1. Pop the 1st pixel from the container to be *(x,y)*.
    2. If *(x,y)* sits on Canny edge or *I*'s boundary, increment *m*.
    3. If *(x,y)* sits on Canny edge and it's not the 1st iteration of (iii), go to next iteration.
    4. If it's not the 1st iteration of (iii), let *a* be the angle of the line stretched from *(row,col)* to *(x,y)*.
    5. for each 4-neighbour of *(x,y)*, *(nx,ny)*
        a. If *(nx,ny)* inside *I* and unmarked, add it to the container and and $|CA - a|$ as the weight value.
        b. Mark it.
iv. Sort the container according to the weights, in ascending order.
v. Let *n* be the size of the container**.**
vi. If $m/(m + n) > 0.65$ then stop.
vii. $s = s + n$.
viii. If $s>ST$ then stop.
ix. If $n>8$ then $n = n/2$.
x. Go to (iii).

We use the condition in (ix) because we don't want the flood-fill spread to be slope-oriented if we have too few pixels in the container. If we have only few pixels, we want to spread them all.
In each spreading iteration, having a set of open pixels (i.e. spreading pixels); we only spread the half of them which are closer to the slope of the cell.

e. Cell detection algorithm

Input: Same as noise detection algorithm from (C).
i. Call **Algorithm 1** with image *I*, threshold *CT* and the markings of Noise detection algorithm(ie 1st phase) as the holes matrix.
ii. Filter out marked areas which are smaller than *S*.
iii. Label 8-neighbourhood connected components in the resulting marked matrix of (ii).
iv. For each connected component do:
    1. Find the mass center *(rc,cc)* which is (mean row, mean column).
    2. Compute *(str,stc)* = (rows standard deviation, columns standard deviation).
    3. Compute the cell slope as *atang(stc/str)*.
    4. Call **Algorithm 3** with 2 times the surface of the component as a surface threshold, the above cell slope and *(rc,cc)*.

The resulting marked areas are the detected cells.

This is what we referred to as a 2nd phase.
Here we use the pixels which are marked by the simple threshold-based filter in order to approximate the cell's slope.

4. Experiments

a. Available images

Here we show 3 sample images on which we experimented our algorithm most. The 1st image is presented in this document as *Image 1*. The other two are presented here.

Image 5

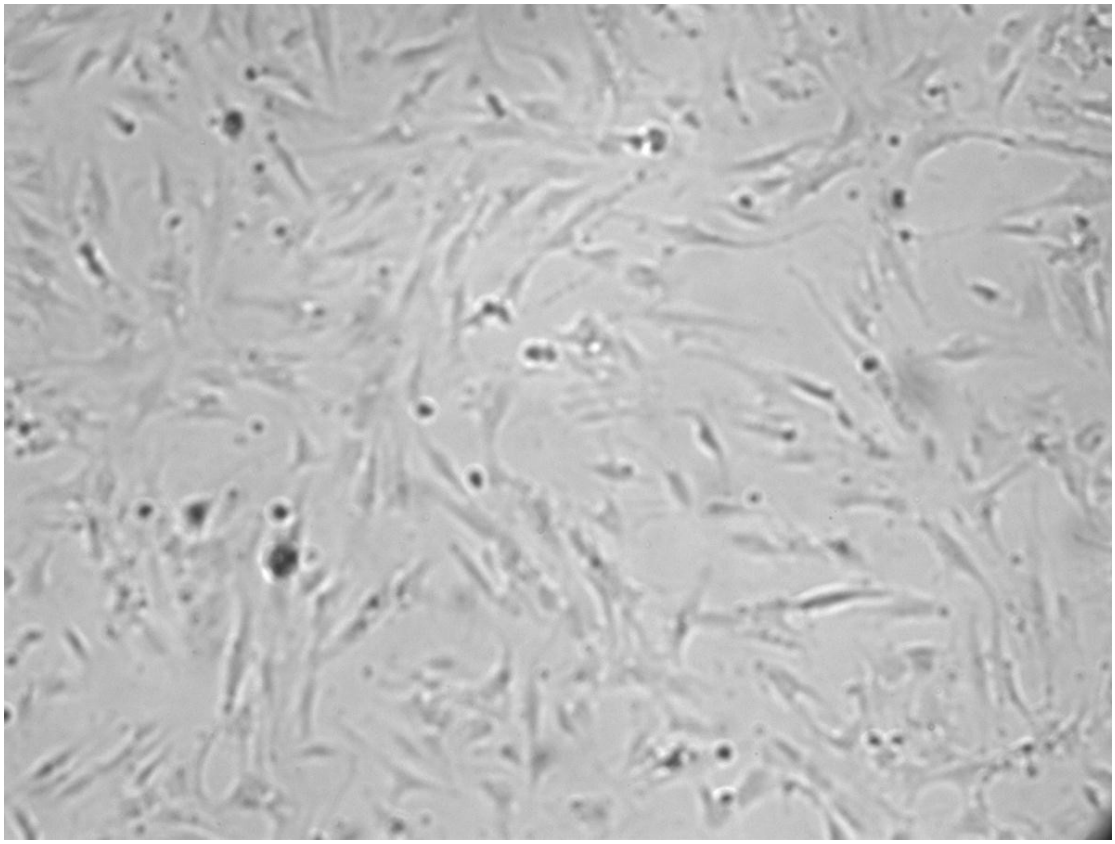
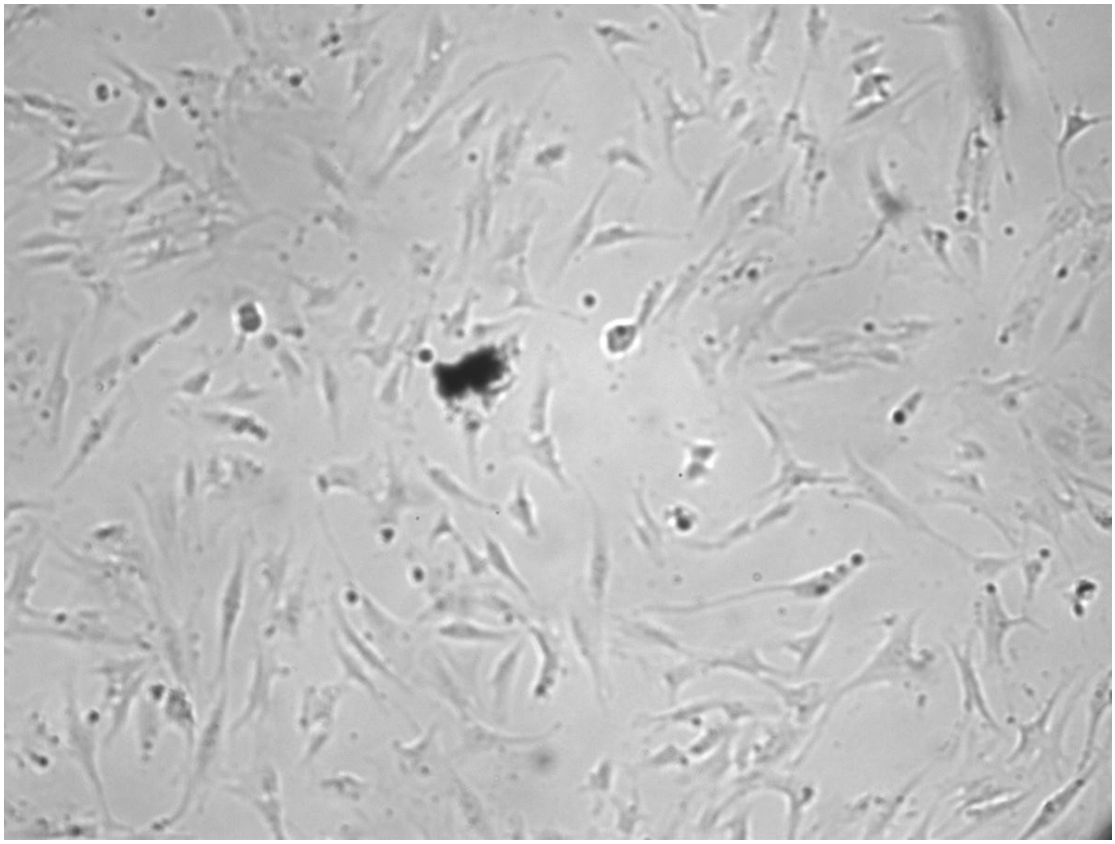


Image 6

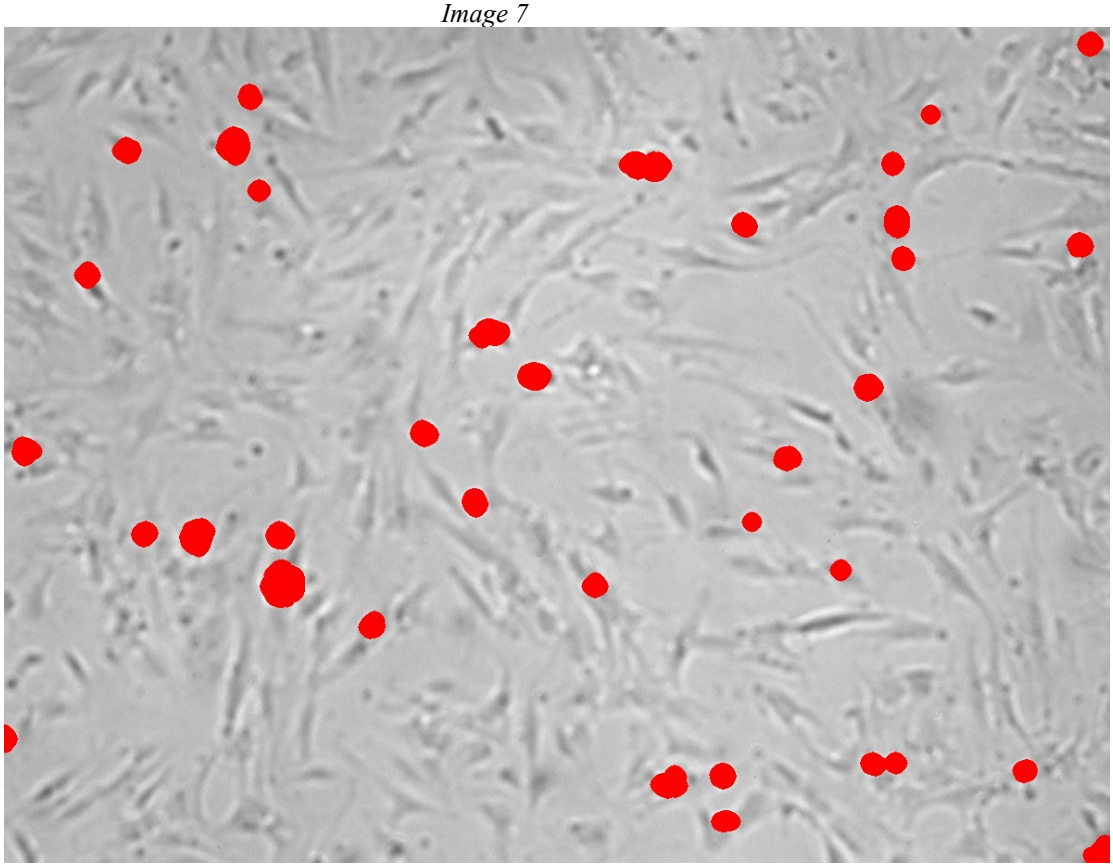


b. Noise markings

Here we demonstrate noise removal on our 3 test images. Noise removal of *Image 1* is marked in red in *Image 4*.

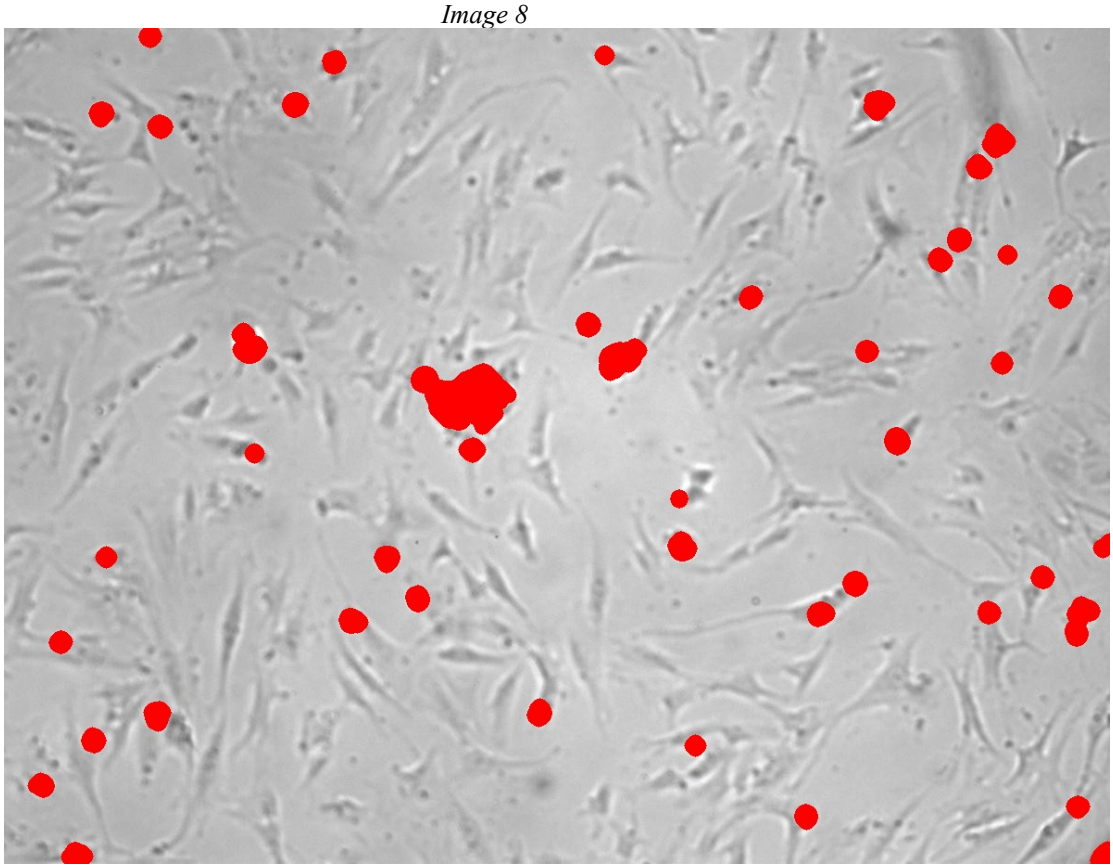
Image 5 markings –

Image 7



And *Image 6* markings –

Image 8

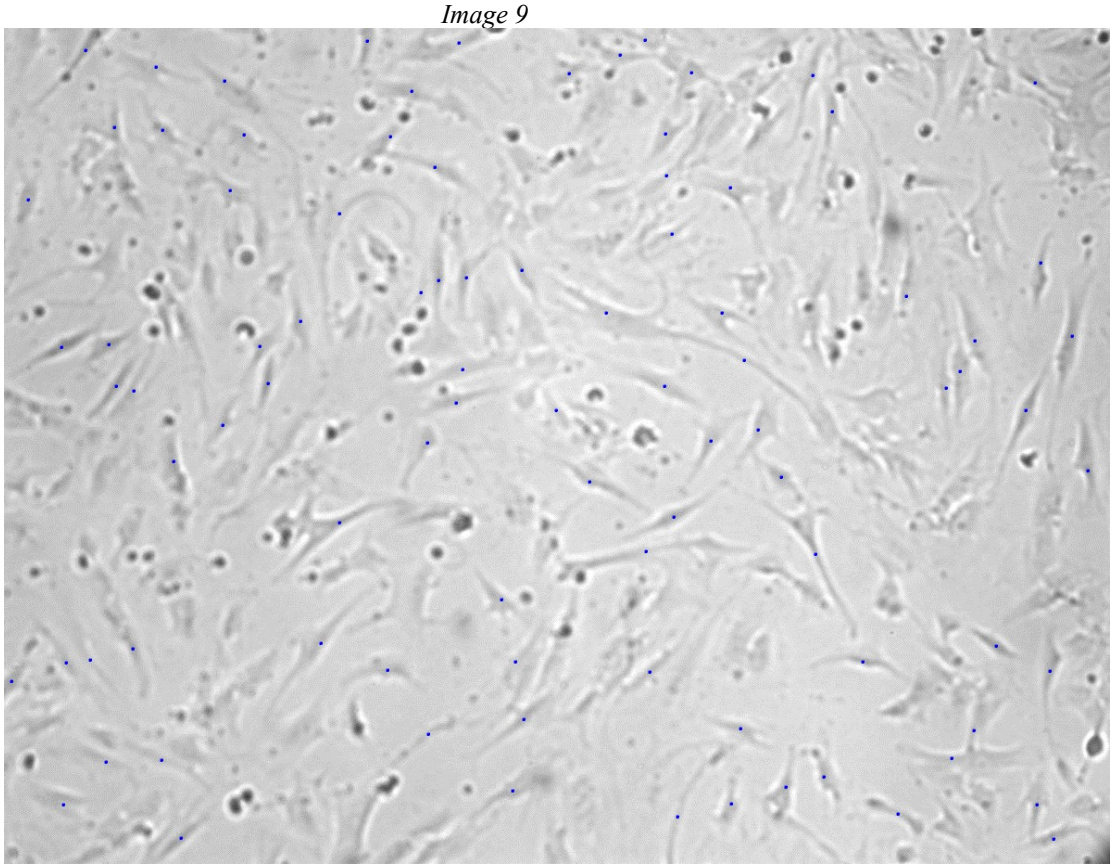


c. Manual markings

We received our sample images without additional information regarding the number or identity of the cells. We had to identify the cells ourselves by looking at the images. In order to make some statistics and compare results numerically, we manually-marked the cells in our 3 test images so that we can estimate how well our algorithm performed to detect those cells.

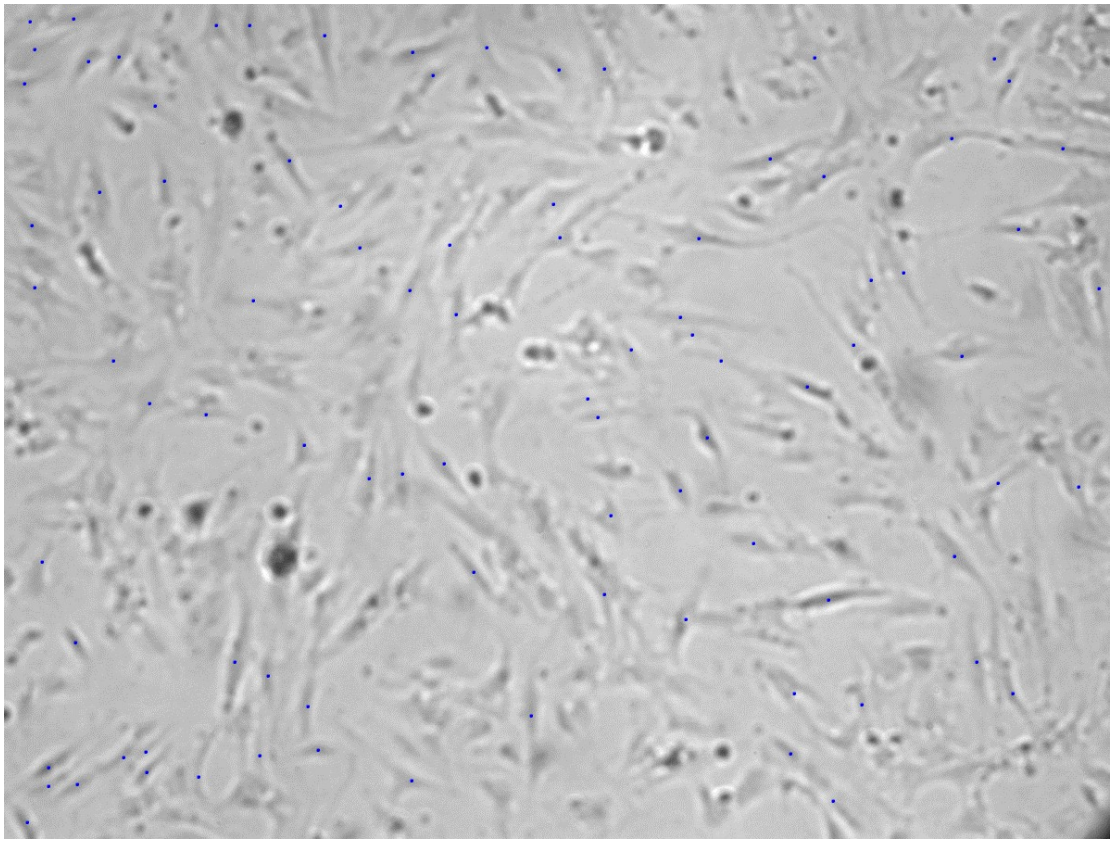
The elements which we considered to be cells are marked with blue dots. Each cell is marked with a single dot. Here are the manual markings, in this order, of the image *Image 1*, *Image 5* and *Image 6*.

Image 9



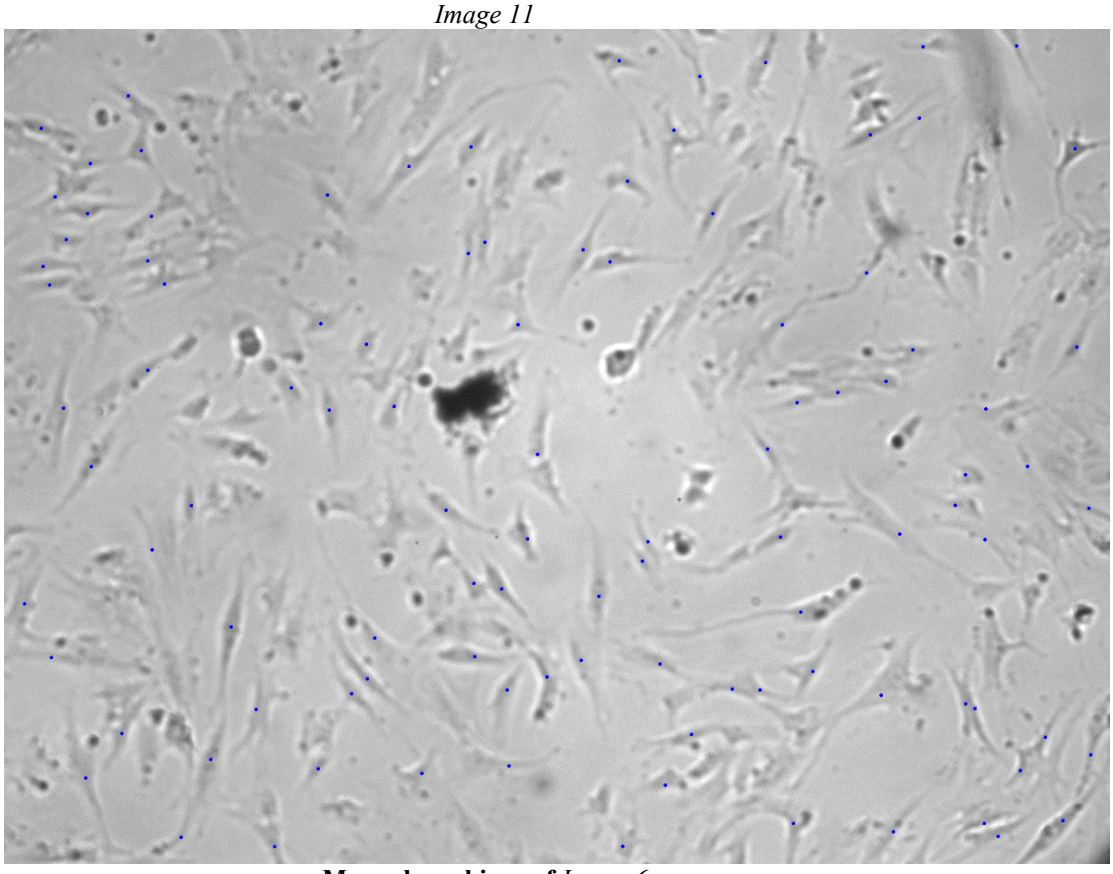
Manual markings of *Image 1*

Image 10



Manual markings of *Image 5*

Image 11



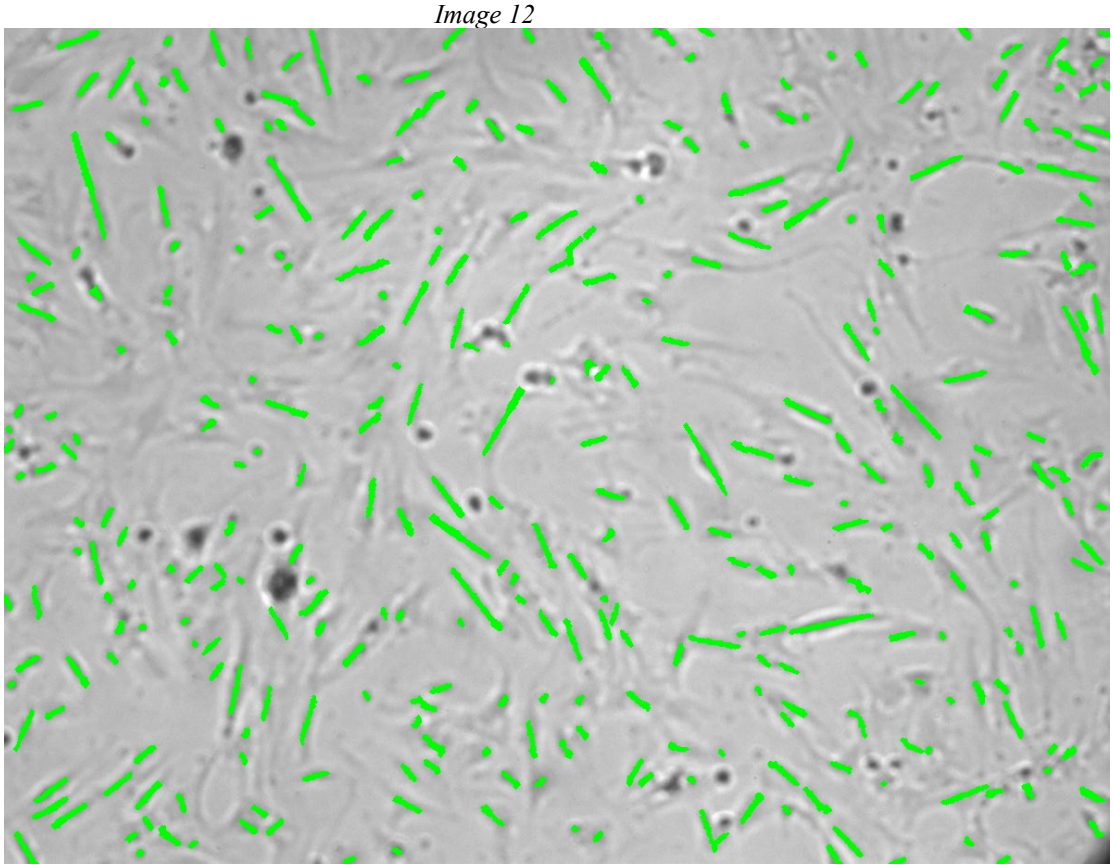
Manual markings of *Image 6*

d. Automatic markings

Here we show the cell markings by our algorithm. We recall that running our algorithm with a cell threshold *CT* means the simple threshold based filter uses a threshold whose value is the local average gray-level minus *CT*.

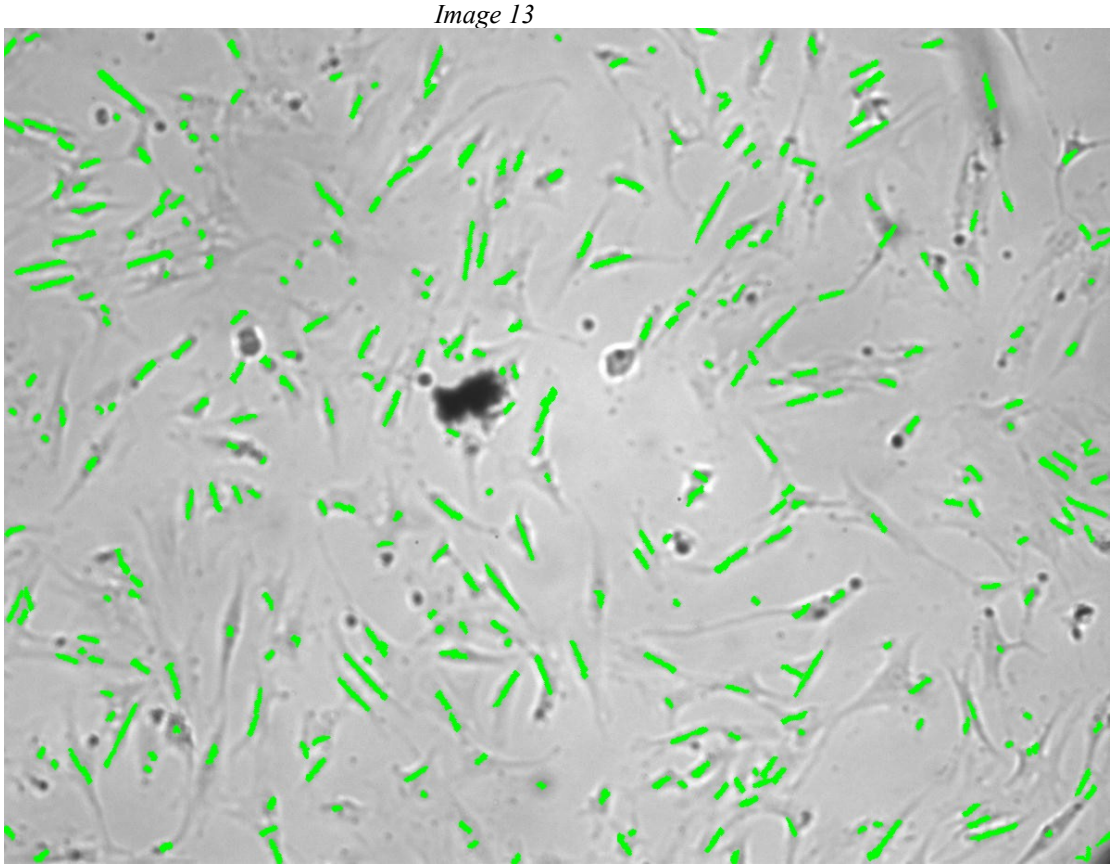
The cell marking of *Image 1* can be seen in green in *Image 4*. It was run with *CT==18*.
Cell markings of *Image 5* and *Image 6*, respectively, with the same *CT* value are shown down here.

Image 12



Cells markings of *Image 5*

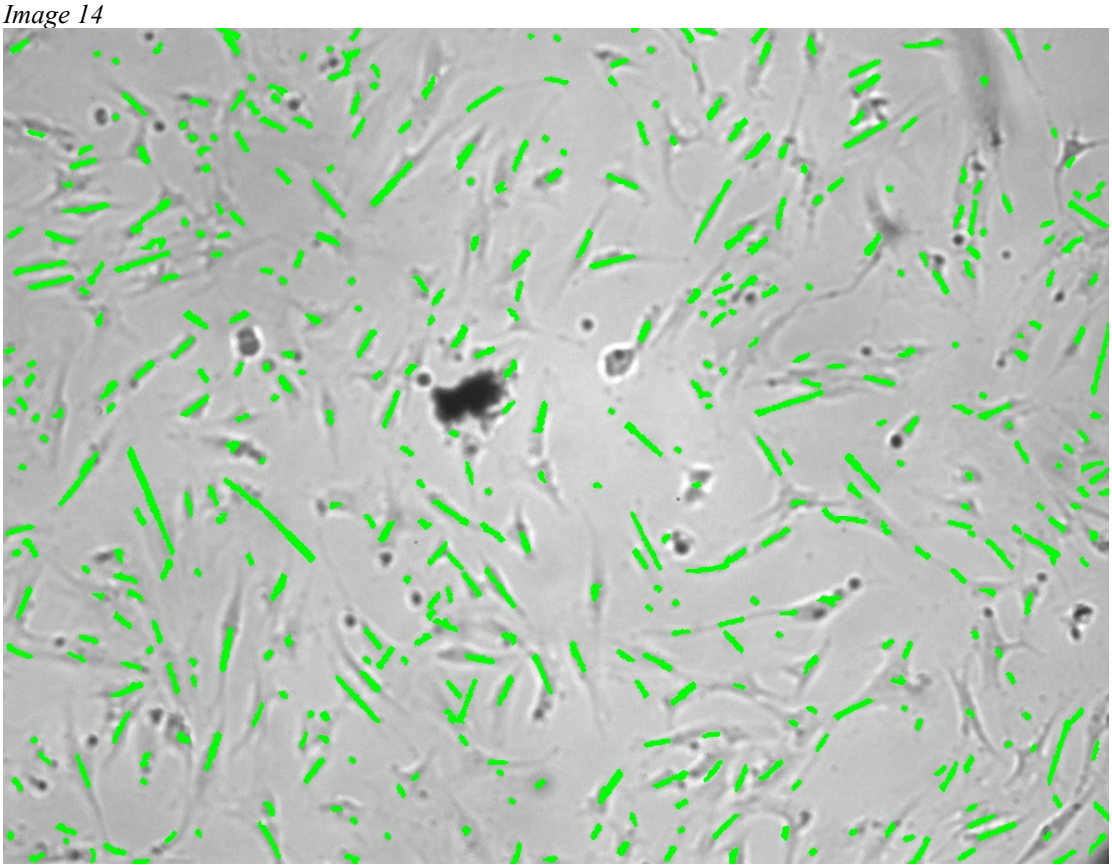
Image 13



Cells markings of *Image 6*

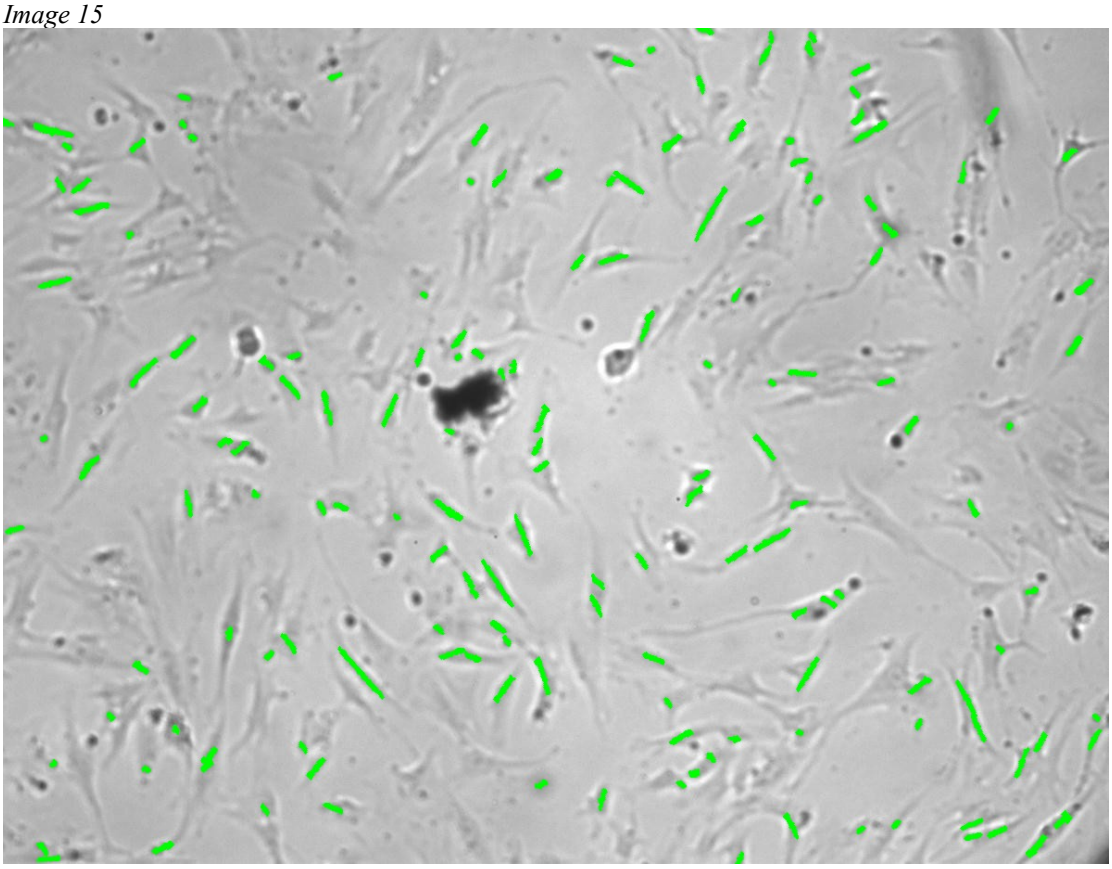
We want to present a couple more cells marking of *Image 6* with different cell threshold.

Image 14



Cell marking of *Image 6* with cell threshold value of 8

Image 15



Cell marking of *Image 6* with cell threshold value of 28

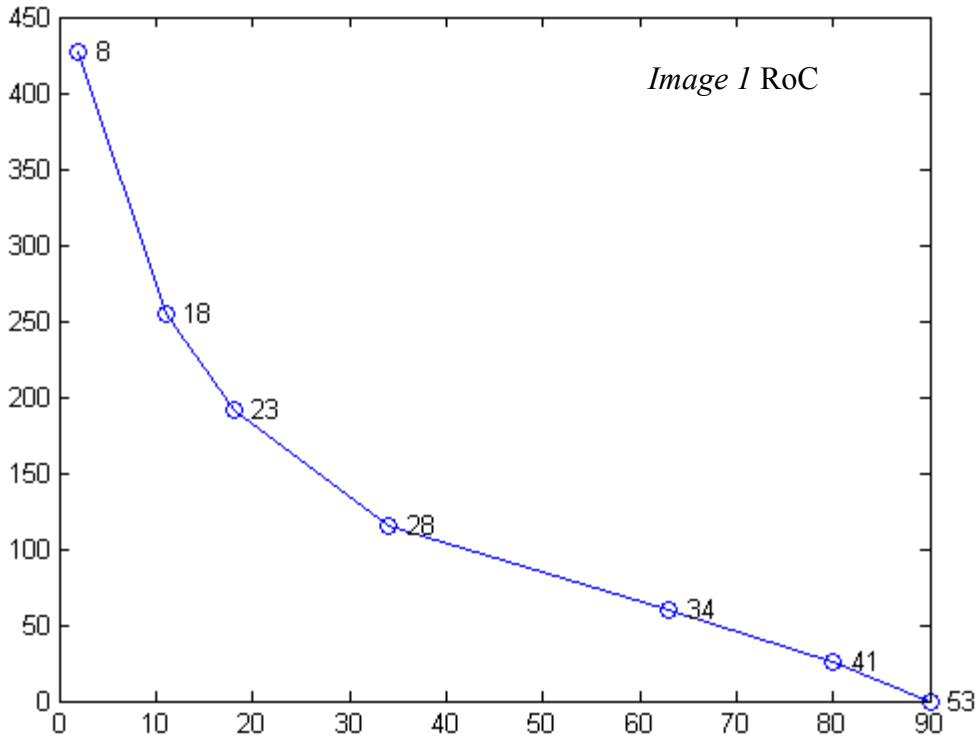
As expected, the higher the threshold value is, the more cell markings we have, but it also means the algorithm incorrectly marks dark elements as cells. If the cell threshold value is low, the algorithm misses more true cells.

So what are the best cell threshold values? Let's see some RoC curves.

e. RoC curves for different images.

Different cell threshold yield different results. We created RoC figures for our 3 test images.

The horizontal axis represents false negatives.
The vertical axis represents false positives.
The numbers right to the circles represent the cell threshold values.



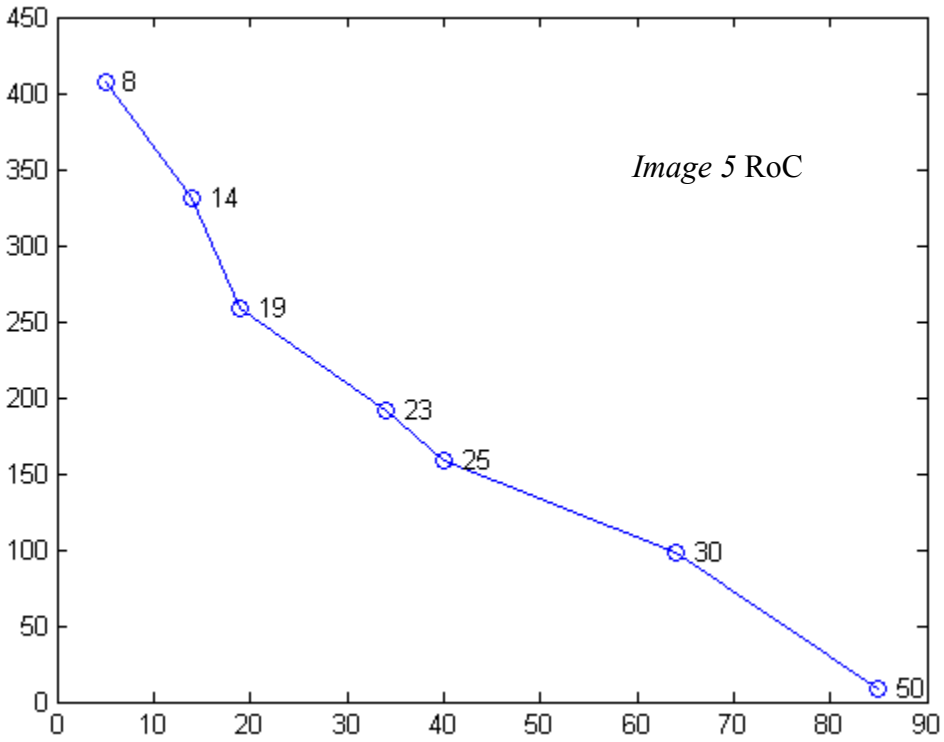


Image 5 RoC

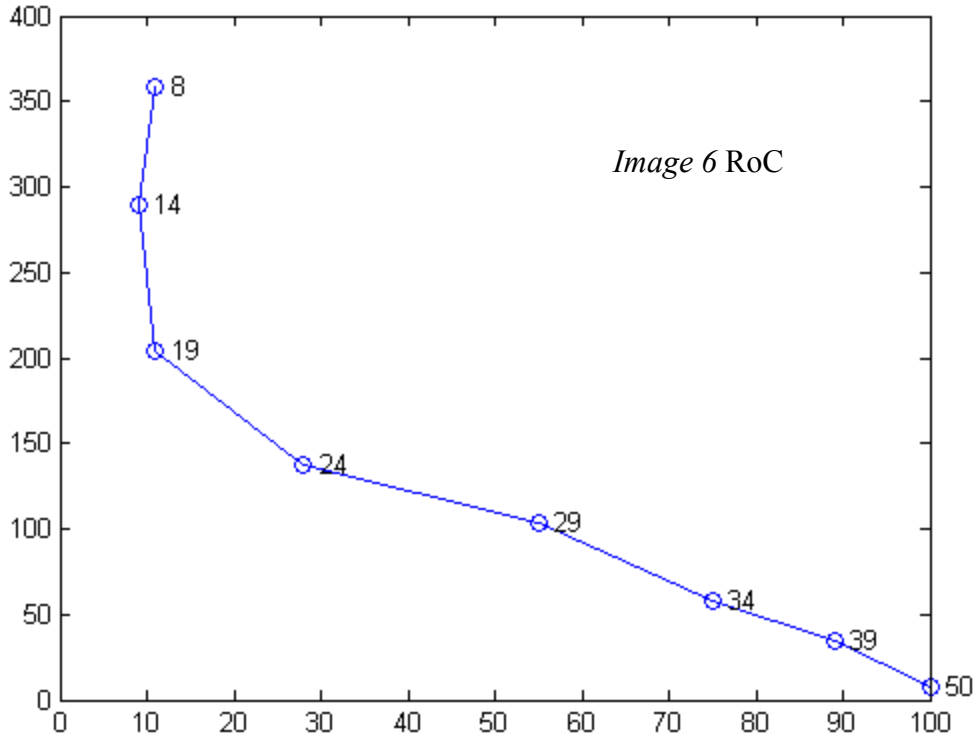


Image 6 RoC

f. Estimation of the true number of cells.

Here we try to correct the estimated number of the cells in the image. Consider RoC curves of other images. For each threshold value T_i we know the ratio of the detected cells to the false positive rFP_i and false negative rFN_i estimation. Then, we can use the following algorithm to correct our estimation:

i. For i=1,2,,N

1. Find number of cells E_i using cell detection algorithm 3.e.
2. Correct the number of cells to be $EC_i = E_i - rFP_i \bullet E_i + rFN_i \bullet E_i$

ii. Output: $EC = \Sigma EC_i / N$.

We took 5 images to demonstrate the above estimation algorithm, 3 of which are our test images. For each of them we'll use the other 4 images' RoCs to estimate the image's number of cells.
Considering the RoC values we have, we chose to use the following 4 cell threshold values – 8, 19, 23 and 29.

The following table shows the estimated and the true (manually marked) number of cells in each of the images:

	Manually marked	Estimation
Image 1	91	102.7381
Image 5	90	99.0235
Image 6	103	89.6815
4th image	98	95.0410
5th image	93	90.4069

The average error is 8%. The differences between our estimation and the manually counting are small and seem to be unbiased. We did not find a simple way to improve our estimation further.
Having 5 images' RoC results, we computed the ratios which can be used for other images.

$$rFN_8 = 0.0120,\ rFN_{19} = 0.0357,\ rFN_{23} = 0.0905,\ rFN_{29} = 0.3158$$

$$rFP_8 = 0.8112,\ rFP_{19} = 0.7293,\ rFP_{23} = 0.6978,\ rFP_{29} = 0.6644$$

5. Conclusions

We developed a cell counting algorithm which is based on gray-level filtering and flood-filling marking. Our 2-phased algorithm is simple and fast.
However, our cell counting algorithm implementation results in numbers which are several times the true number of cells in the image. Noticing the mistakes are similar over several test images we offered a method which roughly estimates the true number of cells. The accuracy we achieved may be sufficient for many applications.

Further methods could be used to improve the accuracy. Studying the statistical relations between the estimated values and the true values over many test images may suggest an addition to our estimation method, which reduces the mistake expectancy.
More complex detection tools may be used; the shapes of the noise blobs and the cells may be better studied in order to improve the distinction between them by using shape detection methods.
More local detection may be adopted. Collecting local characteristics such as brightness, contrast between the dark elements and the environment and even the severity of background noise may all be taken into consideration to produce a complex local detection.
These suggestions are beyond the scope of our work.

Appendix 1.
Software documentation

Our algorithm is implemented in Matlab[3]. Here we describe the functions of which our cell detector consists.

```
function [markedImageOut] = MarkAdaptedAreas(image, threshold, holes)
```
This function corresponds exactly to the adaptive threshold algorithm described in *3a*.

```
function [MarkedImageOut] = FillDirtArea(MarkedImageIn, edges, area_size_threshold, row, col)
```
This function does exactly what is described as the noise blobs flood-fill algorithm from sub-chapter *3b*. The parameter `edges` is a 2D matrix which holds the Canny edges marks. It's size is exactly the size of the image matrix.

```
function [MarkedImageOut] = MarkCells(MarkedImageIn, edges, areaThreshold, origAngle, orig_row, orig_col)
```
This function is cells flood-fill algorithm. It differs to function `FillDirtArea` because the flood-fill spreading is cell-slope-oriented. The pseudo-code implementation is written in sub-chapter *3d*.

```
function [filteredImOut] = Filter(labeledImage, sizeThreshold)
```
This function receives a labeled matrix(same size as images' matrix) and filters out the labeled areas which are smaller than `sizeThreshold`. It is used to filter out small dark elements which otherwise are going to be regarded as cells.

```
function [imOut, n] = CellDetection(I, blackDirtThreshold, cellThreshold, cellAreaSizeThreshold)
```

[3] Numerical computing environment and fourth-generation programming language

This is our main function. It receives the image matrix, noise blob and cell gray-level thresholds and cells' area threshold which is used as a parameter to function `Filter`. This function implements the 2 phases described in chapter 2. It uses all the above functions.

```
function [n] = EstimatedCellDetection(I)
```

This is the estimation function which implements the proposed cell estimation algorithm described in sub-chapter *4f*. This function receives the image and runs `CellDetection` function 4 times with predefined parameters values and returns the estimated number of cells.

Appendix 2

Algorithm running instructions

In order to run our cell counter algorithm, one should invoke `CellDetection`.
Suggested parameter values may be `CellDetection(I, 80, 23, 45)`.
In order to run the estimated cell counter method described in sub-chapter *4f* one just needs to call `EstimatedCellDetection`.