Санкт-Петербургский политехнический университет Петра Великого
Институт компьютерных наук и технологий

Отчёт

по дисциплине «Проектная деятельность»

Выполнили
студенты гр.23534/1

Стойкоски Н.С.
Ракитин О.А.
Амирасанов Э.Г
Нгуен Хай Йен
Нгуен Чунг Киен
Филиппе Мулонде
Станислав Полевич

Руководитель

И. А. Селин

«_29_» __мая_ 2018 г.

Санкт-Петербург
2018

Оглавление

Введение

OpenCV- библиотека компьютерного зрения с открытым исходным кодом(Open Source Computer Vision Library), содержащая более 500 функций, заточенных под выполнение в реальном времени. Библиотека содержит алгоритмы для обработки, реконструкции и очистки изображений, распознания образов, захвата видео, слежения за объектами, калибровки камер и др.

В компьютерном зрении, **сегментация** — это процесс разделения цифрового изображения на несколько сегментов (множество пикселей, также называемых суперпикселями). Цель сегментации заключается в упрощении и/или изменении представления изображения, чтобы его было проще и легче анализировать.Сегментация изображений обычно используется для того, чтобы выделить объекты и границы (линии, кривые, и т. д.) на изображениях. Более точно, сегментация изображений — это процесс присвоения таких меток каждому пикселю изображения, что пиксели с одинаковыми метками имеют общие визуальные характеристики.

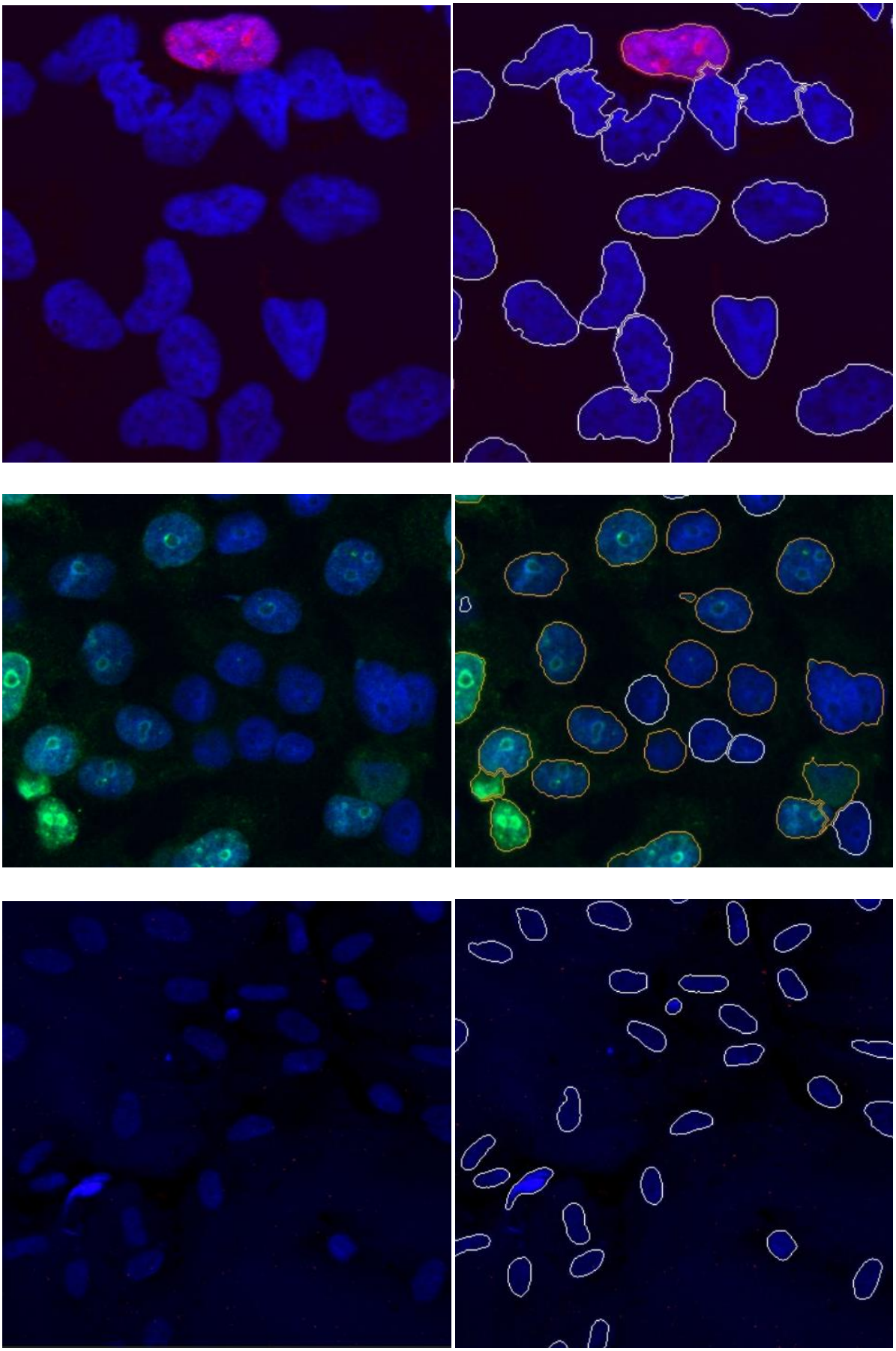
Результатом сегментации изображения является множество сегментов, которые вместе покрывают всё изображение, или множество контуров, выделенных из изображения (см. Выделение границ). Все пиксели в сегменте похожи по некоторой характеристике или вычисленному свойству, например, по цвету, яркости или текстуре. Соседние сегменты значительно отличаются по этой характеристике.[1]

Задание

Разработка системы автоматизированного анализа и классификации медицинских изображений с целью выявления аномалий.

Задан набор медицинских изображении содержащие клеток разных типов. Нужно посчитать их количество, найти их в картину, т.е. определить их место нахождение, размеры, цвет итд.

Примеры работы прогаммы:



Описание работы

Исходное изображение сначало приводится к оттенки серого и бинаризируется используя адаптивные threshold алгоритмы.

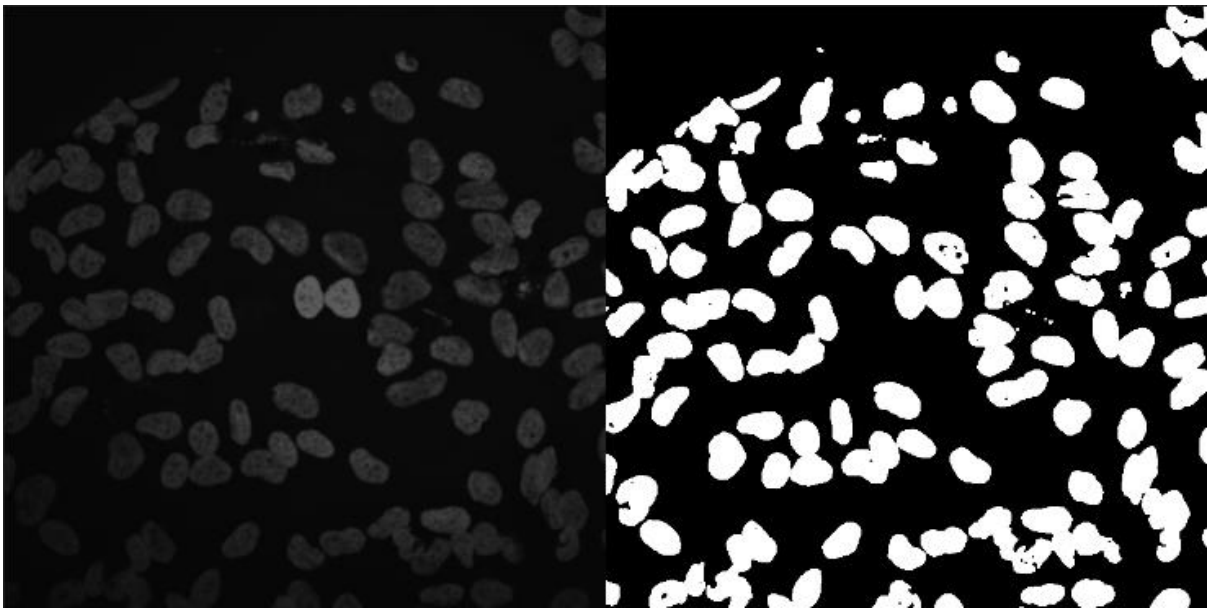


Рис 1. Оттенки серого и результат бинаризации исохдного изображения.

Так же отдеьно бинаризируется красный/зеленый канал, для далнейшее определение цвета клеток. В бинаризированную картину идет поиск по групп соединенных белых пикселов (connectedcomponents). Проблема в том что в бинармон изображений несколько клеток могут быть соединенны в одна и та же група, их нужно соответственно разделить.

Проводится статистическое определение медианной клеткой. Отстраняются маленькие группы соединенных пикселов (которые могут быть вызвани из за шум в изображений, или чувствительности в бинаризации).

Аналогично тому как происходил поиск групп связанных белых пикселов, делается тоже самое, но только для черные. Это нам позволяет определить фон изображений (в качестве которого берется самая большая группа черных пикселов). Остальные группы в зависимости от их площадь по сравнению с медианной клеткой, могут остатся на месте, либо их можно заполнить белым пикселам (могут быть дырки внутри клеток из за шума, недостаточная чувствительность бинаризации). Исправление этих имперфекции нам позвольяет качественно выполнить следующий шаг – разделение группы пикселов на найденные (детектированные) клетки. Нам это позволяет сделать distanceTransform и watershed алгоритм.

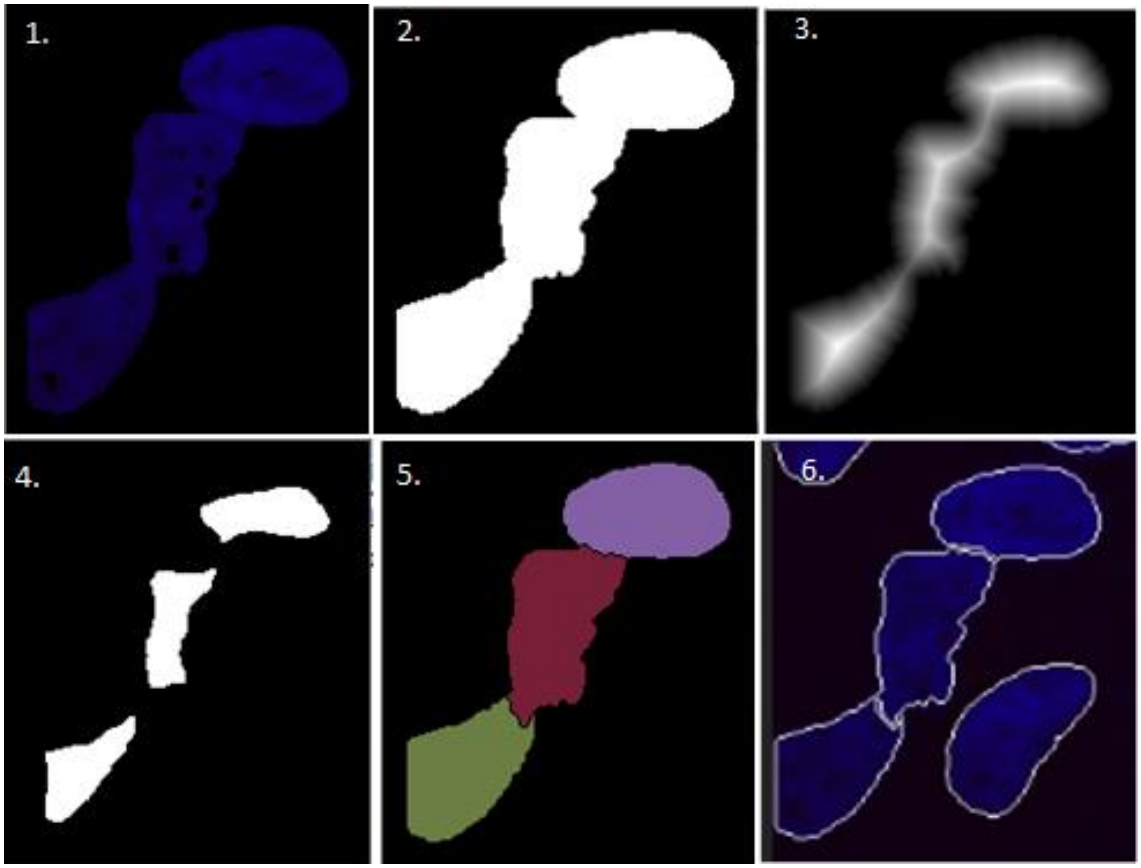


Рис 2. Способ работы watershed алгоритма.

Берется одна из найденных групп белых пикселов, и строится отдельная бинарная картина на черном фоне, которая их содержит. Реализуется distanceTransform (3) который каждому пикселю ставит в соответствие определенная яркость в зависимости от его растояние до ближайшего черного пикселя. После этого реализуется адаптивный threshold (4) где получены маркеры для процедуру watershed. Видно что группа разделилась на 3 клеток. Остается расширить границы клеток с помощи процедуры watershed, результат которой можно увидеть в (5). Видно что сегментация прошла успешно и в (6) показаны эти клетки в конечном результате работы программы.

Для качественная сегментация найболее важно выбрать хорошие параметры для adaptiveThreshold. Для этого была создана отдельная программа которая в активном режиме позволяет посмотреть на результаты сегментации и позволяет управлять параметрам.

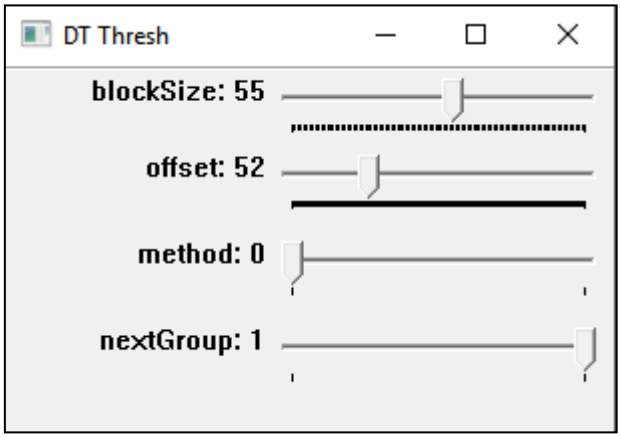


Рис. 3 часть программы управления параметрам adaptiveThreshold

Были рассмотрены несколько примеров из заданного набора, и выбрани следующие параметры:

$$blockSize = \sqrt{\frac{medianCellArea}{\pi}}$$

Offset выбирается в зависимости от blockSize по функцию, полученную через интерполяции по точкам (blockSize, offset) :
(20, 40), (48, 80), (27, 60), (60, 81)

$$f(x) = \frac{809x^3}{776160} - \frac{25931x^2}{155232} + \frac{82871x}{9240} - \frac{43602}{539}$$

$$offset = f(blockSize)$$

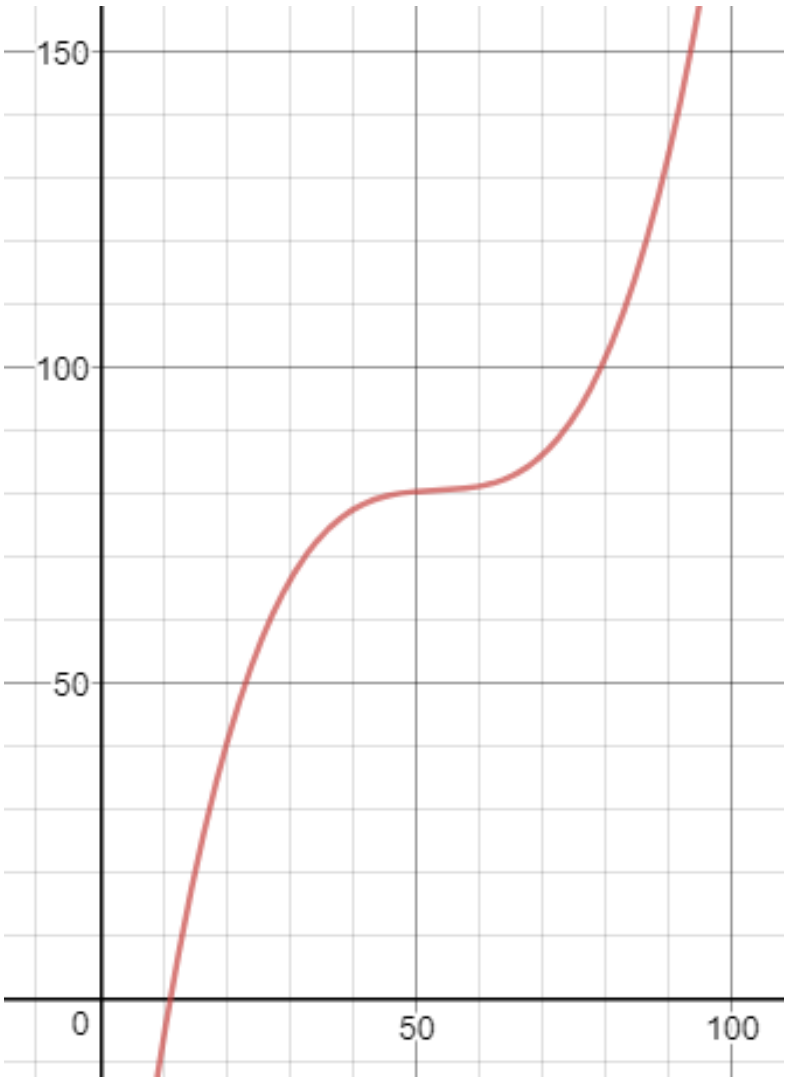


Рис 4. График функции f(x)

Для оценки правильности подсчета программой количества клеток была создана таблица погрешности. (Мы смотрели сколько ошибок допустила программа в подсчете).
Таблица приложена.

Вывод

В результате выполнения работы была реализована система автоматизированного анализа и классификации медицинских изображений с целью выявления аномалий. Также можно сказать, что нам удалось достичь довольно таки высокой точности подсчета, средняя погрешность равна 6,74%.

Таблица погрешностей

имя файла	синие клетки	красные/зеленые клетки	всего клеток	ошибки	погрешность,% = ошибки / всего клеток
detectionsContours1	173	3	176	18	10.23%
detectionsContours2	6	16	22	0	0.00%
detectionsContours3	1072	222	1294	93	7.19%
detectionsContours4	324	82	406	6	1.48%
detectionsContours5	336	73	409	19	4.65%
detectionsContours6	251	32	283	17	6.01%
detectionsContours7	116	4	120	5	4.17%
detectionsContours8	135	9	144	5	3.47%
detectionsContours9	140	8	148	9	6.08%
detectionsContours21	43	162	205	6	2.93%
detectionsContours22	149	14	163	7	4.29%
detectionsContours23	50	24	74	9	12.16%
detectionsContours24	48	31	79	1	1.27%
detectionsContours25	45	30	75	3	4.00%
detectionsContours26	149	14	163	5	3.07%
detectionsContours27	130	82	212	7	3.30%
detectionsContours28	113	22	135	3	2.22%
detectionsContours29	34	42	76	3	3.95%
detectionsContours30	148	26	174	6	3.45%
detectionsContours31	143	28	171	7	4.09%
detectionsContours32	149	50	199	2	1.01%
detectionsContours33	87	27	114	4	3.51%
detectionsContours34	125	20	145	9	6.21%
detectionsContours35	172	26	198	0	0.00%
detectionsContours36	49	37	86	12	13.95%
detectionsContours37	153	6	159	30	18.87%
detectionsContours38	105	23	128	6	4.69%
detectionsContours39	99	20	119	7	5.88%
detectionsContours40	118	8	126	4	3.17%
detectionsContours41	78	18	96	11	11.46%
detectionsContours42	83	14	97	5	5.15%
detectionsContours43	93	14	107	3	2.80%
detectionsContours44	100	10	110	9	8.18%
detectionsContours45	95	17	112	16	14.29%
detectionsContours46	250	100	350	10	2.86%
detectionsContours47	204	143	347	7	2.02%
detectionsContours48	134	194	328	16	4,88%
detectionsContours49	159	146	305	18	16,95%
detectionsContours50	11	17	28	3	10,7%
detectionsContours51	88	60	148	16	10,8%
detectionsContours52	36	1	37	0	0.00%
detectionsContours53	40	0	40	2	5.00%
detectionsContours54	0	128	128	7	5,5%
detectionsContours55	7	12	19	0	0.00%
detectionsContours56	1	18	19	0	0.00%
detectionsContours57	86	25	111	4	3,6%
detectionsContours58	122	86	208	37	17.79%
detectionsContours59	7	111	118	21	17.80%

detectionsContours60	211	0	211	26	12.32%
detectionsContours61	36	27	63	5	7.94%
detectionsContours62	38	26	64	6	9.38%
detectionsContours63	19	10	29	4	13.79%
detectionsContours64	19	13	32	3	9.38%
detectionsContours65	33	12	45	4	8.89%
detectionsContours66	108	10	118	8	6.78%
detectionsContours67	76	10	86	6	6.98%
detectionsContours68	135	10	145	12	8.28%
detectionsContours69	100	12	112	9	8.04%
detectionsContours70	116	11	127	6	4.72%
detectionsContours71	130	14	144	6	4.17%
detectionsContours72	128	10	138	7	5.07%
detectionsContours73	69	4	73	3	4.11%
detectionsContours74	111	14	125	5	4.00%
detectionsContours75	65	14	79	5	6.33%
detectionsContours76	60	14	74	3	4.05%
detectionsContours77	52	12	64	1	1.56%
detectionsContours78	45	12	57	1	1.75%
detectionsContours79	94	21	115	6	5.22%
detectionsContours80	37	3	40	3	7.50%
detectionsContours81	33	7	40	5	12.50%
detectionsContours82	64	4	68	20	29.41%
detectionsContours83	137	0	137	12	8.76%
detectionsContours84	93	0	93	10	10.75%
detectionsContours85	9	8	17	0	0.00%
detectionsContours86	38	8	46	15	32.61%
detectionsContours87	81	31	112	7	6.25%
detectionsContours88	125	36	161	10	6.21%
detectionsContours89	56	4	60	2	3.33%
detectionsContours90	55	4	59	12	20.34%
detectionsContours91	34	0	34	4	11.76%
detectionsContours92	52	0	52	7	13.46%
detectionsContours10	863	67	930	28	3.01%
detectionsContours11	751	89	840	27	3.21%
detectionsContours12	933	58	991	19	1.92%
detectionsContours13	766	110	876	50	5,7%
detectionsContours14	839	76	915	38	4.15%
detectionsContours15	857	81	938	25	5.07%
detectionsContours16	889	55	944	77	8.16%
detectionsContours17	892	122	1014	59	5.82%
detectionsContours18	916	93	1009	46	4.56%
detectionsContours19	843	53	896	27	3.01%
detectionsContours20	650	95	745	45	6.00%
				Avg:	6.74%

Таблица использованных функций OpenCV

Функция	Назначение
imread	Loads an image from a file.
Imwrite	Saves an image to a specified file.
imshow	Displays an image in the specified window.
namedWindow	Creates a window.
createTrackbar	Creates a trackbar and attaches it to the specified window.
split	Divides a multi-channel array into several single-channel arrays.
merge	Creates one multichannel array out of several single-channel ones.
cvtColor	Converts an image from one color space to another.
resize	Resizes an image.
GaussianBlur	Blurs an image using a Gaussian filter.
medianBlur	Blurs an image using the median filter.
adaptiveThreshold	Applies an adaptive threshold to an array.
erode	Erodes an image by using a specific structuring element.
dilate	Dilates an image by using a specific structuring element.
distanceTransform	Calculates the distance to the closest zero pixel for each pixel of the source image.
normalize	Normalizes the norm or value range of an array.
findContours	Finds contours in a binary image.
drawContours	Draws contours outlines or filled contours.
watershed	Performs a marker-based image segmentation using the watershed algorithm.
putText	Draws a text string.
circle	Draws a circle.

Текст программы

CellDetector.hpp

```
#ifndef CELLDETECTOR_HPP
#defineCELLDETECTOR_HPP

#include<opencv2/core/core.hpp>
#include<opencv2/highgui/highgui.hpp>
#include<opencv2/imgproc/imgproc.hpp>
#include<vector>
#include<list>
#include<iostream>

usingnamespace cv;

structpoint_t
{
int row, col;
};

enumColor_t{RED, BLUE};

structCellGroup
{
double cells, cellsRed, cellsBlue;
int minRow, maxRow, minCol, maxCol;
size_t size;
unsignedint sizeRed;
double redToBlueRatio;
point_t center;
  std::list<point_t> *pixels;
bool detectedWithWatershed = false;
};

classCellDetector
{
public:
	structParameters
	{
		double resizeAmount = 0.2; //deprecated
		double binaryThreshold = 16.0; //deprecated
		int minimumCellPixels = 20;
	};

	CellDetector() = default;
	CellDetector(constCellDetector::Parameters&params);
	virtual ~CellDetector() = default;

	void setParameters(constCellDetector::Parameters&params);

void detect(constMat&image, std::ostream&log, const std::string&imgFileName,
std::ostream *userLog);


private:
	CellDetector::Parameters params_;
  cv::Mat binaryRGB_, binaryRED_, rgb_resized_, rgb_gray_resized_,
rgb_resized_blackBackground_;
  cv::Mat groupLabels_, rgb_resized_contours_;
unsignedint medianCellArea_;
```

```
int __blockSize, __offset;
  std::string _imgFileName;
  std::ostream *userLog_;

       void processImage(constMat&src_rgb, std::ostream&log);
Mat getBinaryImage(constMat&gray);
  std::list<CellGroup> *findPixelGroups();
void filterDetections(std::list<CellGroup> *groups, std::ostream&log);
void calculateCellCenters(std::list<CellGroup> *groups);
void displayDetections(const std::list<CellGroup> * groups);
void watershedDetection(std::list<CellGroup> *groups);

void labelDetections(const std::list<CellGroup> *groups, std::ostream&log);

  std::list<point_t> *findBackground();
  std::list<point_t> *backgroundPixels_;
Mat isFromWatershed_;

};

#endif
```

CellDetector.cpp

```
#include"CellDetector.hpp"
#include<iostream>
#include<algorithm>
#include<queue>
#include<vector>
#include<fstream>
#include<list>
#include<numeric>
#include<iomanip>
#include<set>

//#define IMG_T1
//#define SHOW_DEBUG

voidCellDetector::processImage(const cv::Mat&src_rgb, std::ostream&log)
{
usingnamespace cv;

//Get Binary Image from Red and Green Channel
//note: red will mean red and green
Mat BGRChannels[3], src_red;
  split(src_rgb, BGRChannels);
  BGRChannels[0] =Mat::zeros(src_rgb.rows, src_rgb.cols, CV_8UC1);  //remove blue
channel
  merge(BGRChannels, 3, src_red);

//Convert both images to grayscale
Mat gray_rgb_big, gray_red_big;
  cvtColor(src_rgb, gray_rgb_big, CV_RGB2GRAY);
  cvtColor(src_red, gray_red_big, CV_RGB2GRAY);

//Resize both images
//double resizeAmount = 1 / ((double)gray_rgb_big.rows / 640.0);
double resizeAmount = 1 / ((double)gray_rgb_big.rows / 1024.0);

  std::cout <<"Resize amount = "<< resizeAmount << std::endl;
log<<"Resize amount = "<< resizeAmount << std::endl;

Mat gray_rgb, gray_red;
  resize(gray_rgb_big, gray_rgb, Size(), resizeAmount, resizeAmount);
```

```
  resize(gray_red_big, gray_red, Size(), resizeAmount, resizeAmount);

//Save image for further use (showing results)
  rgb_gray_resized_ = gray_rgb;
  resize(src_rgb, rgb_resized_, Size(), resizeAmount, resizeAmount);

//Get binaries
  binaryRGB_ = getBinaryImage(gray_rgb);
  binaryRED_ = getBinaryImage(gray_red);

//imshow("SourceRGB", rgb_resized_);
//imshow("binaryRGB", binaryRGB_);
//imshow("binaryRED", binaryRED_);

}
MatCellDetector::getBinaryImage(constMat&gray)
{
Mat bin, blured;
#ifdef IMG_T1 //for normal images
  GaussianBlur(gray, blured, Size(3, 3), 0);
  adaptiveThreshold(blured, bin, 255, ADAPTIVE_THRESH_MEAN_C, THRESH_BINARY, 131, -1);
  Mat tmp;
  erode(bin, tmp, Mat());
  dilate(tmp, bin, Mat());
#else// for noisy images
  medianBlur(gray, blured, 7);
  adaptiveThreshold(blured, bin, 255, ADAPTIVE_THRESH_GAUSSIAN_C, THRESH_BINARY, 131,
-5);
#endif

//static int i = 1;
//imwrite("binary" + std::to_string(i++) + ".jpg", bin);
return bin;
}


voidCellDetector::detect(constMat&image, std::ostream&log, const
std::string&imgFileName,
  std::ostream *userLog)
{
  userLog_ = userLog;
log<<" - - - - - - - - - - - - - - - -  - - -"<< std::endl;
  _imgFileName =imgFileName;

//Binarize the source image
  processImage(image, log);

//Find groups of connected pixels
  std::cout <<"FPG1"<< std::endl;
  std::list<CellGroup> *groups = findPixelGroups();

//Analyze groups
  std::cout <<"FILTER"<< std::endl;
  filterDetections(groups, log);

//Find background pixel group
//todo: grupite na pixeli sho se pomali od mozebi 0.2*medianArea popolni gi so bela
boja
//findBackground (which colors the inside of cells)
//must be after filterDetections because it depends on medianCellArea
  std::cout <<"FINDBACKGROUND"<< std::endl;
  backgroundPixels_ = findBackground();
```

```
//cells now dont contain any more holes, groups need to be updated
//for correct usage in watershed
//easy way (slow):
//delete groups;
  std::cout <<"FPG2"<< std::endl;
  groups = findPixelGroups();
  std::cout <<"NEW GROUPS SIZE: "<< groups->size() << std::endl;
  filterDetections(groups, log);
//hard way:....

  std::cout <<"WATERSHED"<< std::endl;
  watershedDetection(groups);

//Calculate cell centers
  std::cout <<"CENTERS"<< std::endl;
  calculateCellCenters(groups);

//Draw cell contours and display result
  std::cout <<"LABELDETECTIONS"<< std::endl;
  labelDetections(groups, log);

//Show centers on image
//std::cout << "DISPLAY" << std::endl;
//displayDetections(groups);

}
std::list<CellGroup> *CellDetector::findPixelGroups()
{
Mat bin = binaryRGB_;
Mat binRed = binaryRED_;

//visited pixel matrix
  std::vector<std::vector<bool>> visited(bin.rows);
for (size_t i = 0; i < visited.size(); i++)
    visited[i].resize(bin.cols, false);

//directions for BFS
int rdir[4] = { 0, 0,  1, -1 };
int cdir[4] = { -1, 1, 0,  0 };

//check if coordinate is inside img
auto inRange = [&](introw, intcol)
  {
if (row< 0 || row>= bin.rows || col< 0 || col>= bin.cols)
returnfalse;
returntrue;
  };

  std::list<CellGroup> *groups = new std::list<CellGroup>();

for (int i = 0; i < bin.rows; i++)
  {
    std::cout <<".";
for (int j = 0; j < bin.cols; j++)
    {
if (visited[i][j] || bin.at<uchar>(i, j) == 0) //pixel is black (background)
continue;

      groups->emplace_back();
      groups->back().pixels = new std::list<point_t>();

      groups->back().minRow = i;
      groups->back().maxRow = i;
      groups->back().minCol = j;
```

```
      groups->back().maxCol = j;


      std::queue<point_t> q;
      q.push(point_t{ i, j });

while (!q.empty())
      {
point_t cur = q.front();
        q.pop();

if (!inRange(cur.row, cur.col))
continue;
if (visited[cur.row][cur.col])
continue;
if (bin.at<uchar>(cur.row, cur.col) == 0)
continue;

        visited[cur.row][cur.col]=true;
        groups->back().pixels->push_back(cur);

        groups->back().minRow = min(groups->back().minRow, cur.row);
        groups->back().maxRow = max(groups->back().maxRow, cur.row);
        groups->back().minCol = min(groups->back().minCol, cur.col);
        groups->back().maxCol = max(groups->back().maxCol, cur.col);

if (binRed.at<uchar>(cur.row, cur.col) != 0) //If pixel is red
        {
          groups->back().sizeRed++;
        }

for (size_t k = 0; k < 4; k++)
        {
          q.push(point_t{ cur.row + rdir[k], cur.col + cdir[k] });
        }
      }
      groups->back().size = groups->back().pixels->size();
    }
  }

  std::cout <<"finished bfs"<< std::endl;

return groups;
}
voidCellDetector::filterDetections(std::list<CellGroup> *groups, std::ostream&log)
{
  std::cout <<"Getting Statistics"<< std::endl;
  std::cout <<"minimumCellPixels: "<< params_.minimumCellPixels << std::endl;
log<<"Getting Statistics"<< std::endl;
log<<"minimumCellPixels: "<< params_.minimumCellPixels << std::endl;

//Remove cells below minimum
auto it = groups->begin();
while (it !=groups->end())
  {
//if (it->size < params_.minimumCellPixels)
if(it->size < 50)
    {
      it =groups->erase(it);
continue;
    }
    it++;
  }
  std::cout <<"Erased groups below minimum"<< std::endl;
```

```
//Sort the cellGroups in order to find median
size_t numGroups = groups->size();
groups->sort([](constCellGroup&a, constCellGroup&b) { returna.size <b.size; });
//TEST:
groups->reverse();

unsignedint medianGroupSize = std::next(groups->begin(), numGroups / 2)->size;
  std::cout <<"Median Cell Pixels: "<< medianGroupSize << std::endl;
log<<"Median Cell Pixels: "<< medianGroupSize << std::endl;
  medianCellArea_ = medianGroupSize;

//Calculate cell color
for (auto it = groups->begin(); it !=groups->end(); it++)
  {
    it->cells = (double)it->size / (double)medianGroupSize;
//it->cellsRed = it->sizeRed / ((double)medianGroupSize * 0.8);
    it->cellsRed = it->sizeRed / (double)it->size;
//it->cellsRed = std::min(it->cellsRed, it->cells);
    it->cellsBlue = it->cells - it->cellsRed;
    it->redToBlueRatio = (double)it->sizeRed / (double)it->size;
  }

}
voidCellDetector::calculateCellCenters(std::list<CellGroup> *groups)
{
  std::cout <<"Calculating cell centers"<< std::endl;
for(auto it=groups->begin(); it !=groups->end(); it++)
  {
    it->center.row = std::accumulate(it->pixels->begin(), it->pixels->end(), 0,
      [](unsignedinttotal, constpoint_t&pt) { returntotal + pt.row; }) / it->size;

    it->center.col = std::accumulate(it->pixels->begin(), it->pixels->end(), 0,
      [](unsignedinttotal, constpoint_t&pt) { returntotal + pt.col; }) / it->size;
  }
}
voidCellDetector::displayDetections(const std::list<CellGroup> *groups)
{
  std::cout <<"Displaying Detections"<< std::endl;
  cv::Mat img = rgb_resized_;

for(auto it = groups->begin(); it !=groups->end(); it++)
  {
    std::string information ="+";

CvScalar color;

if (it->cellsRed > 0.5)
      color = cvScalar(0, 163, 244);
else
      color = cvScalar(255, 255, 255);


//todo: remove (for debugging)
if (it->detectedWithWatershed)
    {
      color = cvScalar(0, 255, 0);
    }

    cv::putText(img, information,
      cvPoint(it->center.col, it->center.row),
      cv::FONT_HERSHEY_COMPLEX, 0.3,
      color, //cvScalar(255,255,255),
      1, CV_AA);
```

```
  }


staticint idx = 0;
  idx++;
  std::string filename = "img_detections_"+ std::to_string(idx) +".jpg";
  cv::imwrite(filename, img);

//cv::namedWindow("Detected Cells", cv::WINDOW_NORMAL);
//cv::imshow("Detected Cells", img);

}
void non_maxima_suppression(const cv::Mat&image, cv::Mat&mask, boolremove_plateaus) {
// find pixels that are equal to the local neighborhood not maximum (including
'plateaus')
  cv::dilate(image, mask, cv::Mat());
  cv::dilate(image, mask, cv::Mat());
  cv::compare(image, mask, mask, cv::CMP_GE);

// optionally filter out pixels that are equal to the local minimum ('plateaus')
if (remove_plateaus) {
    cv::Mat non_plateau_mask;
    cv::erode(image, non_plateau_mask, cv::Mat());
    cv::compare(image, non_plateau_mask, non_plateau_mask, cv::CMP_GT);
    cv::bitwise_and(mask, non_plateau_mask, mask);
  }
//imshow("mask1", mask);
}
voidCellDetector::watershedDetection(std::list<CellGroup> *groups)
{
// create matrix for labeling if pixel is part of cell found with watershed
//todo: mozebi nema potreba ovie raboti da se prat tuka voobspto, proveri pak!
Mat tmp(rgb_resized_.rows, rgb_resized_.cols, CV_16UC1, cv::Scalar(0));
  isFromWatershed_ = tmp;

  std::list<CellGroup> newCellGroups;
//make the background of resized src_rgb image black
  rgb_resized_blackBackground_;
  rgb_resized_.copyTo(rgb_resized_blackBackground_);
for (int i = 0; i < binaryRGB_.rows; i++)
  {
for (int j = 0; j < binaryRGB_.cols; j++)
    {
if (binaryRGB_.at<uchar>(i, j) == 0)
      {
        rgb_resized_blackBackground_.at<Vec3b>(Point(j, i)) =Vec3b(0, 0, 0);
      }
    }
  }

for (auto it = groups->begin(); it !=groups->end(); )
  {
if(it->cells > 1.0)
    {

//Create binary image from group pixels with black border
int originalWidth = it->maxCol - it->minCol + 1;
int originalHeight = it->maxRow - it->minRow + 1;

int width = originalWidth + 2*10; // +20 for black frame around img
int height = originalHeight + 2*10;

Size size(width, height);
Mat bin = Mat::zeros(size, CV_8U);
```

```
int offsetRow = it->minRow - 10;
int offsetCol = it->minCol - 10;
for (auto it2 = it->pixels->begin(); it2 != it->pixels->end(); it2++)
      {
        bin.at<uchar>(Point(it2->col - offsetCol, it2->row - offsetRow)) = 255;
      }

//Get image block for current group from src_rgb with black border
Mat img = Mat::zeros(size, rgb_resized_blackBackground_.type());
for (auto it2 = it->pixels->begin(); it2 != it->pixels->end(); it2++)
      {
        img.at<Vec3b>(Point(it2->col - offsetCol, it2->row - offsetRow))
= rgb_resized_blackBackground_.at<Vec3b>(Point(it2->col, it2->row));
      }

//imshow("binary filled", bin);
//imshow("added border", img);

/* * * Watershed algorithm * * */

//Perform the distance transform algorithm
Mat dist;
      distanceTransform(bin, dist, CV_DIST_L2, 5);
      normalize(dist, dist, 0.0, 1.0, NORM_MINMAX);

Mat tmp1;
      dist.convertTo(tmp1, CV_8UC1, 255.0);
      dist = tmp1;

#ifdef SHOW_DEBUG
      imshow("distance transform cellgroup", dist);
#endif

Mat tmp;
//TEST3: calculated blocksize, interpolated offset
constint minBlockSize = 21, maxBlockSize = 79;
int additionalOffset = 0;
int blockSize = std::min(maxBlockSize, 2 * (int)std::sqrt((double)medianCellArea_ /
3.14));
if (blockSize < minBlockSize)
      {
        additionalOffset = minBlockSize - blockSize;
        blockSize = minBlockSize;
      }
if (blockSize % 2 == 0)
        blockSize++;

//lagrange interpolation by points (blockSize, offset): (20,40) (27,60) (48,80) (60,
81)
auto fOffset = [](doublex){
return ((809.0 * x*x*x) / 776160.0) - ((25931.0 * x*x) / (155232.0))
          + ((82871.0 * x) / 9240.0) - (43602.0 / 539.0);
      };
int offset = (int)fOffset(blockSize);

      __blockSize = blockSize;
      __offset = offset;
//std::cout << "blockSize, offset : " << blockSize << " " << offset << std::endl;

      adaptiveThreshold(dist, tmp, 255, ADAPTIVE_THRESH_MEAN_C, THRESH_BINARY,
blockSize, -offset);
      dist = tmp;

      erode(dist, dist, Mat());
```

```
      dilate(dist, dist, Mat());

#ifdef SHOW_DEBUG
      imshow("threshold", dist);
#endif

//imshow("thresh", dist);
// Create the CV_8U version of the distance image
// It is needed for findContours()
Mat dist_8u;
      dist.convertTo(dist_8u, CV_8U);

// Find total markers
      std::vector<std::vector<Point>> contours;
      findContours(dist_8u, contours, CV_RETR_EXTERNAL, CV_CHAIN_APPROX_SIMPLE);

// Create the marker image for the watershed algorithm
Mat markers = Mat::zeros(dist.size(), CV_32SC1);

// Draw the foreground markers
for (size_t i = 0; i < contours.size(); i++)
        drawContours(markers, contours, static_cast<int>(i),
Scalar::all(static_cast<int>(i) + 1), -1);
      circle(markers, Point(2, 2), 1, CV_RGB(255,255,255), -1);

Mat mark1 = Mat::zeros(markers.size(), CV_8UC1); //
      markers.convertTo(mark1, CV_8UC1);               //
//imshow("Before watershed", mark1*10000);

// Perform the watershed algorithm
      watershed(img, markers);

//todo:remove, just for testing
Mat mark = Mat::zeros(markers.size(), CV_8UC1); //
      markers.convertTo(mark, CV_8UC1);              //
//imshow("mark", mark*10000);
//imshow("img", img);

      std::vector<Vec3b> colors;
for (size_t i = 0; i < contours.size()+10; i++)
      {
int b = theRNG().uniform(0, 255);
int g = theRNG().uniform(0, 255);
int r = theRNG().uniform(0, 255);
        colors.push_back(Vec3b((uchar)b, (uchar)g, (uchar)r));
      }



      std::vector<CellGroup> newGroups(contours.size());
for (int i = 0; i < newGroups.size(); i++)
      {
        newGroups[i].pixels = new std::list<point_t>();
      }
// Create the result image
Mat dst = Mat::zeros(markers.size(), CV_8UC3);
// Fill labeled objects with random colors
for (int i = 0; i < markers.rows; i++)
      {
for (int j = 0; j < markers.cols; j++)
        {
int index = markers.at<int>(i, j);
//std::cout << index << " ";
//if (index > 0 && index <= static_cast<int>(contours.size()))
```

```
if (index > 0 && index <= contours.size())
          {
            dst.at<Vec3b>(i, j) = colors[index - 1];
            newGroups[index - 1].pixels->push_back(point_t{ i + offsetRow, j +
offsetCol});

if (binaryRED_.at<uchar>(i + offsetRow, j + offsetCol) != 0) //If pixel is red
            {
              newGroups[index - 1].sizeRed++;
            }
          }
else
            dst.at<Vec3b>(i, j) =Vec3b(0, 0, 0);

        }
      }

#ifdef SHOW_DEBUG
      imshow("binary filled", bin);
      imshow("added border", img);
// Visualize the final image
      imshow("Final Result", dst);
      waitKey();
#endif
//waitKey();

for (int i = 0; i < newGroups.size(); i++)
      {
        newGroups[i].size = newGroups[i].pixels->size();
        newGroups[i].detectedWithWatershed = true;
        newCellGroups.push_back(newGroups[i]);
      }
if (newGroups.size() > 0)
      {
//label group pixels as found with watershed
//todo: mozebi nema potreba od ova ovde?, proveri pak!
        std::for_each(it->pixels->begin(), it->pixels->end(), [&](constpoint_t&pt) {
          isFromWatershed_.at<ushort>(pt.row, pt.col) = 1;
        });

//erase group from list
        it =groups->erase(it);
continue;
      }
    }
    it++;
  }

  filterDetections(&newCellGroups, std::cout);

for (auto it = newCellGroups.begin(); it != newCellGroups.end(); it++)
  {
groups->push_back(*it);
  }

}
voidCellDetector::labelDetections(const std::list<CellGroup> *groups,
std::ostream&log)
{
Mat labels(rgb_resized_.rows, rgb_resized_.cols, CV_16UC1, cv::Scalar(0));
  groupLabels_ = labels;

Mat cellColors;
  groupLabels_.copyTo(cellColors);
```

```
Mat isFromWatershed = isFromWatershed_;
//groupLabels_.copyTo(isFromWatershed);

//Label full groups
//TODO: also take cell colors
ushort groupNum = 1;

int totalRedCells = 0, totalBlueCells = 0, totalCells = 0;

for (auto it = groups->begin(); it !=groups->end(); it++)
  {
    std::for_each(it->pixels->begin(), it->pixels->end(), [&](constpoint_t&pt) {
      groupLabels_.at<ushort>(pt.row, pt.col) = groupNum;

if (it->cellsRed > 0.5)
        cellColors.at<ushort>(pt.row, pt.col) = 2; //red
else
        cellColors.at<ushort>(pt.row, pt.col) = 1; //blue
    });
    groupNum++;

//Get statistics
    totalCells++;
if (it->cellsRed > 0.5)
      totalRedCells++;
else
      totalBlueCells++;
  }


//Extract contours
//todo: optimize by not going through whole image, but only group pixels
  rgb_resized_.copyTo(rgb_resized_contours_);
for (int row = 1; row < groupLabels_.rows - 1; row++)
  {
for (int col = 1; col < groupLabels_.cols - 1; col++)
    {
if (groupLabels_.at<ushort>(row, col) != 0 && isFromWatershed.at<ushort>(row, col) ==
1
&& (groupLabels_.at<ushort>(row - 1, col) == 0 ||
          groupLabels_.at<ushort>(row + 1, col) == 0 ||
          groupLabels_.at<ushort>(row, col - 1) == 0 ||
          groupLabels_.at<ushort>(row, col + 1) == 0))
      {
if (cellColors.at<ushort>(row, col) == 1)
          rgb_resized_contours_.at<Vec3b>(row, col) =Vec3b(255, 255, 255);
else
          rgb_resized_contours_.at<Vec3b>(row, col) =Vec3b(0, 152, 234);
      }
    }
  }
for (auto it = backgroundPixels_->begin(); it != backgroundPixels_->end(); it++)
  {
int row = it->row;
int col = it->col;

staticint dirs_row[4] = { 1, -1, 0, 0 };
staticint dirs_col[4] = { 0, 0, 1, -1 };

for (int i = 0; i < 4; i++)
    {
if (groupLabels_.at<ushort>(row + dirs_row[i], col + dirs_col[i]) > 0 &&
        isFromWatershed.at<ushort>(row + dirs_row[i], col + dirs_col[i]) != 1)
```

```
      {
if (cellColors.at<ushort>(row + dirs_row[i], col + dirs_col[i]) == 1)
          rgb_resized_contours_.at<Vec3b>(row, col) =Vec3b(255, 255, 255);
else
          rgb_resized_contours_.at<Vec3b>(row, col) =Vec3b(0, 152, 234);
break;
      }
    }
  }

//cv::namedWindow("With contours", cv::WINDOW_NORMAL);
//imshow("With contours", rgb_resized_contours_);
staticint i = 52;
  imwrite("detectionsContours"+ std::to_string(i++) +".jpg", rgb_resized_contours_);
log<<"outputFileName: detectionsContours"+ std::to_string(i - 1) +".jpg"<< std::endl;
log<<"blockSize, offset: "<< __blockSize <<", "<< __offset << std::endl;
log<<"imgFileName: "<< _imgFileName << std::endl;

  (*userLog_) <<" - + - + - + - + - + - + - + - + - + - + - + - + - + - + - + - + - + - + - + -
"<< std::endl;
  (*userLog_) <<"Filename: detectionsContours"+ std::to_string(i - 1) +".jpg"<<
std::endl;
  (*userLog_) <<"Blue, Red/Green, Total: "<< totalBlueCells <<", "<< totalRedCells
<<", "<< totalCells << std::endl;
}
CellDetector::CellDetector(constCellDetector::Parameters&params)
{
  params_ =params;
}
voidCellDetector::setParameters(constCellDetector::Parameters&params)
{
  params_ =params;
}

std::list<point_t> *CellDetector::findBackground()
{
  std::list< std::list<point_t>* > blackPixelGroups;

//visited pixel matrix
  std::vector<std::vector<bool>> visited(binaryRGB_.rows);
for (size_t i = 0; i < visited.size(); i++)
    visited[i].resize(binaryRGB_.cols, false);

//directions for BFS
int rdir[4] = { 0, 0,  1, -1 };
int cdir[4] = { -1, 1, 0,  0 };

//check if coordinate is inside img
auto inRange = [&](introw, intcol)
  {
if (row< 0 || row>= binaryRGB_.rows || col< 0 || col>= binaryRGB_.cols)
returnfalse;
returntrue;
  };

  std::cout <<"Searching for background pixel group"<< std::endl;

for (int row = 0; row < binaryRGB_.rows; row++)
  {
    std::cout <<".";
for (int col = 0; col < binaryRGB_.cols; col++)
    {
if (!visited[row][col]&& binaryRGB_.at<uchar>(row, col) == 0) //pixel is black
      {
```

```
        std::list<point_t> *group = new std::list<point_t>();

        std::queue<point_t> q;
        q.push({ row, col });

while (!q.empty())
        {
point_t cur = q.front();
          q.pop();

if (!inRange(cur.row, cur.col))
continue;
if (visited[cur.row][cur.col] || binaryRGB_.at<uchar>(cur.row, cur.col) != 0)
continue;

          visited[cur.row][cur.col]=true;

//skipping frame pixels for optimization
if (cur.row > 0 && cur.row < binaryRGB_.rows - 1 &&
            cur.col > 0 && cur.col < binaryRGB_.cols - 1)
          {
            group->push_back(cur);
          }


for (size_t k = 0; k < 4; k++)
          {
            q.push(point_t{ cur.row + rdir[k], cur.col + cdir[k] });
          }
        }

        blackPixelGroups.push_back(group);
      }
    }
  }
  std::cout << std::endl;

    blackPixelGroups.sort([](const std::list<point_t> *g_a, const std::list<point_t>
*g_b) {
returng_a->size() <g_b->size();
  });

//fill holes inside cell
for (auto it = blackPixelGroups.begin(); it != blackPixelGroups.end(); it++)
  {
if ((*it)->size() < 0.1 * (double)medianCellArea_)
    {
      std::for_each((*it)->begin(), (*it)->end(), [&](constpoint_t&pt) {
        binaryRGB_.at<uchar>(pt.row, pt.col) = 255;
      });
    }
else
    {
break;
    }
  }

staticint i = 1;
  imwrite("binaryFilled"+ std::to_string(i++) +".jpg", binaryRGB_);

//return largest black pixel group (background)
return blackPixelGroups.back();

}
```

Main.cpp

```
#include<opencv2/core/core.hpp>
#include<opencv2/highgui/highgui.hpp>
#include<opencv2/imgproc/imgproc.hpp>
#include<opencv2/features2d.hpp>
#include<iostream>
#include<algorithm>
#include<queue>
#include<vector>
#include<fstream>
#include<list>
#include<iomanip>
#include"CellDetector.hpp"
#include"WatershedCellDetector.h"


int main(intargc, char **argv)
{
usingnamespace cv;

CellDetector detector;

  std::ifstream config("config.txt");

if (config.is_open())
  {
    std::cout <<"Reading from config file"<< std::endl;

    std::string tmp;
    std::getline(config, tmp);

CellDetector::Parameters params;
    config >> params.minimumCellPixels;
    config >> params.resizeAmount;
    config >> params.binaryThreshold;
    detector.setParameters(params);
  }
else
  {
    std::cout <<"No config file found, using default configuration"<< std::endl;
  }

  std::ofstream log("log.txt");
  std::ofstream userLog("log_user.txt");

for (int i = 1; i <= 92; i++)
  {
    std::string filename = "1 ("+ std::to_string(i) +").jpg";
    std::ifstream imgFile(filename);

if (imgFile.is_open())
    {
      imgFile.close();
Mat src_img = imread(filename);
      detector.detect(src_img, log, filename, &userLog);
    }
  }

       return 0;
}
```