02132 Assignment 1: Software implementation of a cell detection and counting algorithm in C

Luca Pezzarossa and Jan Madsen

September 3, 2025

Preface

In this assignment, you are required to implement an algorithm for the detection and the counting of living cells in a microscopic image. The algorithm should be implemented in software using the C programming language. The C language is largely used in the development of embedded systems since it offers a sufficiently high programming abstraction level combined with the possibility to control and manage low-level features of the computing device. This assignment aims to let you practice with the C language and to give you a detailed understanding of the challenges involved in programming embedded systems.

1 General information

This assignment should be carried out in **groups of two or three people**. You are free to select your group members. Further information regarding the group registration will follow as DTU-Learn announcements. In addition to the implementation algorithm, you are also required to prepare a short report describing the approach used in its implementation (further details are provided in Section 6). All the material related to this assignment can be found on the DTU-Learn course page at the following location:

`DTU-Learn/Content/Assignments/Assignment 1`

The deadline for this assignment is **Sunday 5th October 2025 at 11:59**. By this date, you have to hand in an electronic version of the short report in PDF format and the source files as ZIP archive using the assignment utility in the DTU-Learn course page at the following location:

`DTU-Learn/Assignments/Assignment 1`

Before the deadline, during the last laboratory session dedicated to this assignment held on **Wednesday 1st October 2025** , you are asked to demonstrate the functionality of your implementation to the teacher or to the TAs. More information regarding the DEMO session will follow as DTU-Learn announcements.

This document is divided into 6 sections:

- Section 2 provides the general background needed for the assignment, as well as the specifications of the C program (i.e., the format of the input/output files, information on the code we provide to facilitate the implementation, etc.).
- Section 3 describes the algorithm to be used for the detection and the counting of cells.
- Section 4 presents the tests needed to prove the correct functionality of the implemented algorithm and to analyse its performance.
- Section 5 discusses possible optimizations and enhancements you can apply to your implementation.
- Section 6 lists the assignment tasks and the report requirements.
- Section 7 contains notes and hints from previous iterations of the course.

There are many ways to implement the cell detection and count algorithm. We are not enforcing a particular approach in the implementation (we give you the algorithm and the main function interfaces, but you are free in the implementation), rather we would like you to explore. For this reason, it may be difficult for the teachers to understand your implementation and thus, help us read/debug your code by taking into account the following hints:

- Make sure you understand the algorithm.
- Structure your code such that it is clear what the different parts are doing.
- Think about testing your code while planning and make sure that you test and debug the individual parts.
- Use a structural approach in the implementation.
- Comment your code.
- Deliver all the source code needed to test the implementation and, if necessary, a README file with instructions on how to run it.

Feel free to ask or send an e-mail to the course teacher if you have questions regarding practical matters and the assignment in general.

2 Background and specifications

You work as an engineer at the fictive company Bioware System, which develops and produces advanced equipment for biochemical laboratories. One of the products that the company is developing is a digital microfluidic biochip to perform clinical and diagnostic tests.

A Digital Microfluidic Biochip (DMB) is a small device that allows for complete lab processes to be carried out. In a traditional wet lab, reactions of milliliter-sized volumes take place in test tubes and reagents are moved between test tubes by manual (or robot-automated) pipetting. Processes like heating and shaking or measurements of temperature and screening of molecules or DNA, are carried out by placing the test tubes in dedicated instruments. A DMB allows for precise handling of microliter-sized droplets that act as reaction chambers, i.e. working on 1/1000 of the volumes in traditional wet-lab test tubes. A DMB may integrate a range of sensors and actuators in order to automate and miniaturize the traditional wet lab processes. Droplets can be moved using the electrowetting effect and serve the function of fluidic vehicles and reaction chambers, and they can be mixed, merged, split, dispensed, disposed, heated, etc. without the use of any pipetting or manual interventions. Figure 1 shows the full DMB platform under development at DTU and an enlargement of the active area (electrodes and droplets).

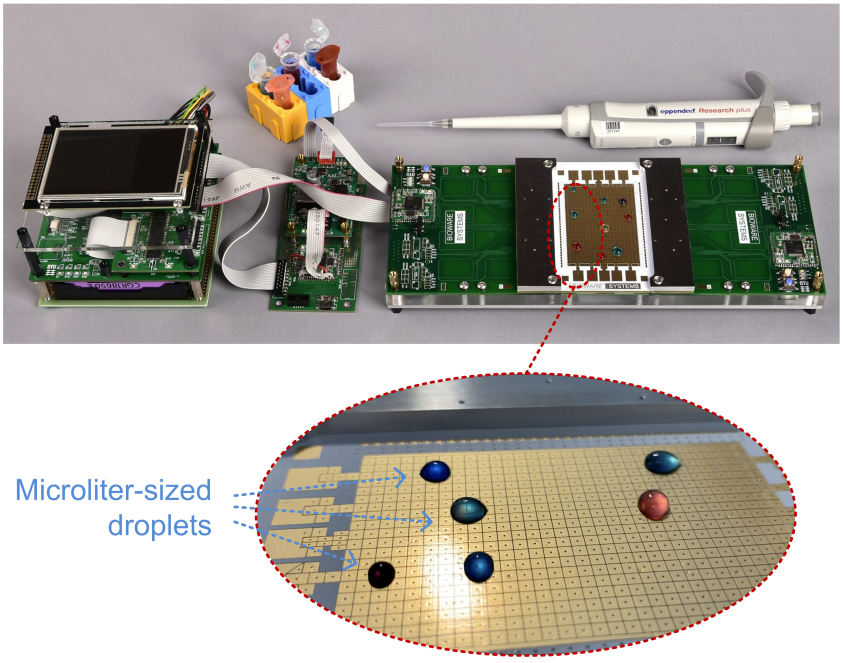

Figure 1: the full DMB platform under development at DTU and an enlargement of the active area (electrodes and droplets).

The clinical test that the Bioware System company wants to develop requires the detection and the counting of cells in a resulting droplet and you are given

the task of investigating and implementing a computer vision-based algorithm to perform this task. More specifically, your main task is to implement the algorithm in C and determine whether the algorithm is suitable to be implemented in an embedded system as part of their new product.

Figure 2 shows the full system and the role of the detection and count algorithm. The left side of the figure shows the bio-chip with a sample droplet. A microscopic picture of the sample is taken by a camera and processed by the algorithm running on the embedded computing system. The algorithm produces in output the total number of detected cells and a list of coordinates (x_pixel, y_pixel) for each detected cell. In addition, an output image is created by overlaying symbols (e.g., a cross) to indicate the detected cells. Figure 3 shows the picture received in input by the algorithm and the produced output image.

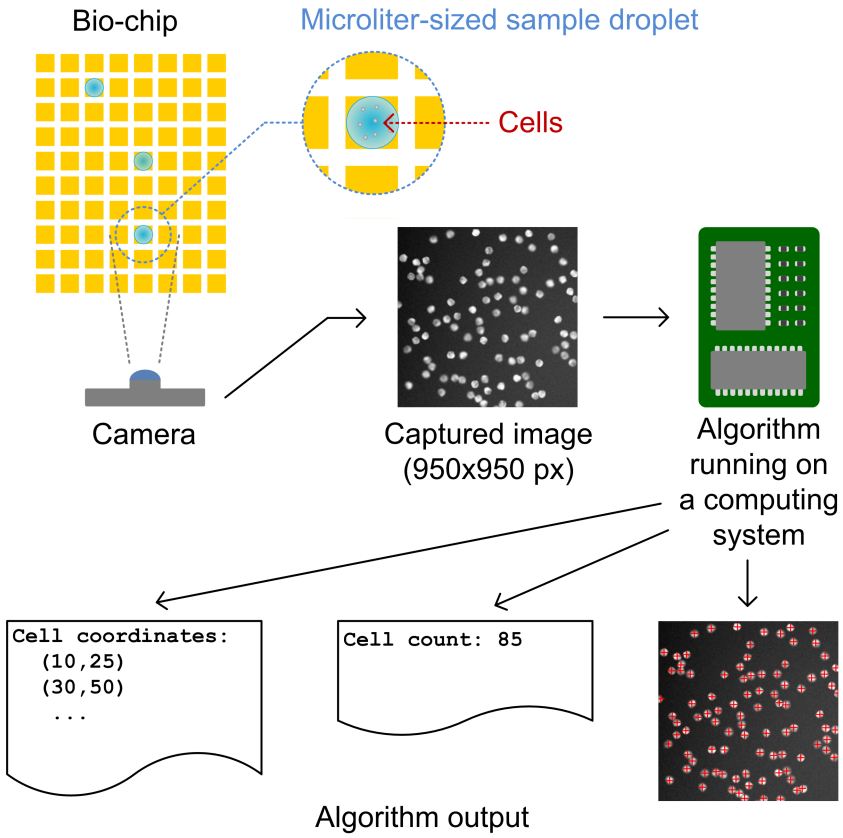

Figure 2: An overview of the full system and the role of the detection and count algorithm.

The size of the picture taken by the microscopic camera is 950 pixels by 950 pixels. The picture is a bitmap characterized by 3 colour channels: red, green, blue. Each colour channel is represented by a byte (8-bits). Thus, 24-bits per pixel. Figure 4 shows the mapping of the pixels in the image. Please, note that the pixels are zero-indexed. From a programming point-of-view, the image is represented as a 3-dimensional array of type unsigned char (8 bits), for example:

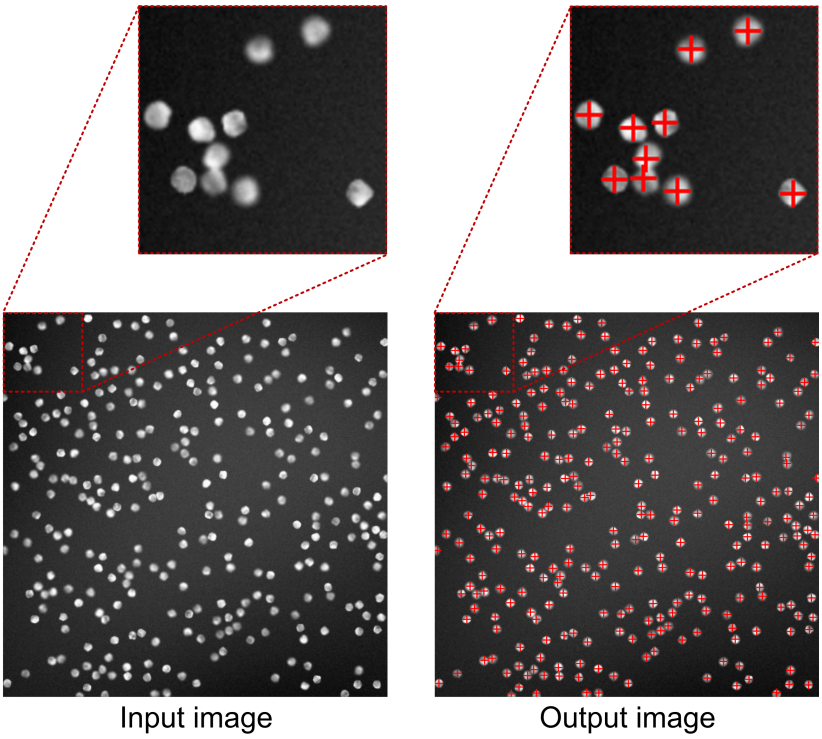

Figure 3: The algorithm input and output images.

```
unsigned char image[950][950][3]
```

Where the first index is the x-pixel coordinate, the second index is the y-pixel coordinate, and the third index is the colour channel (0 = red, 1 = green, 2 = blue).

In summary, the main specifications for the algorithm development are as follows:

- The algorithm is written in the C language.
- The size of the input image is 950 pixel by 950 pixels.
- The input image has 3 colour channels: red, green, blue. Each colour channel is represented by a byte (8-bits). Thus, 24-bits per pixels.
- The algorithm should produce in output: (1) the total number of detected cells, (2) a list of coordinates (x_pixel, y_pixel) for each detected cell, and (3) an output image created by overlaying symbols (e.g., a cross) to indicate the detected cells.
- The program should take in input the path of the input image file to be processed and the path of the output image file to be saved.

To help you get started, we provide you with the library `cbmp.h` to load and save images to files. The description of the functions offered by this library is

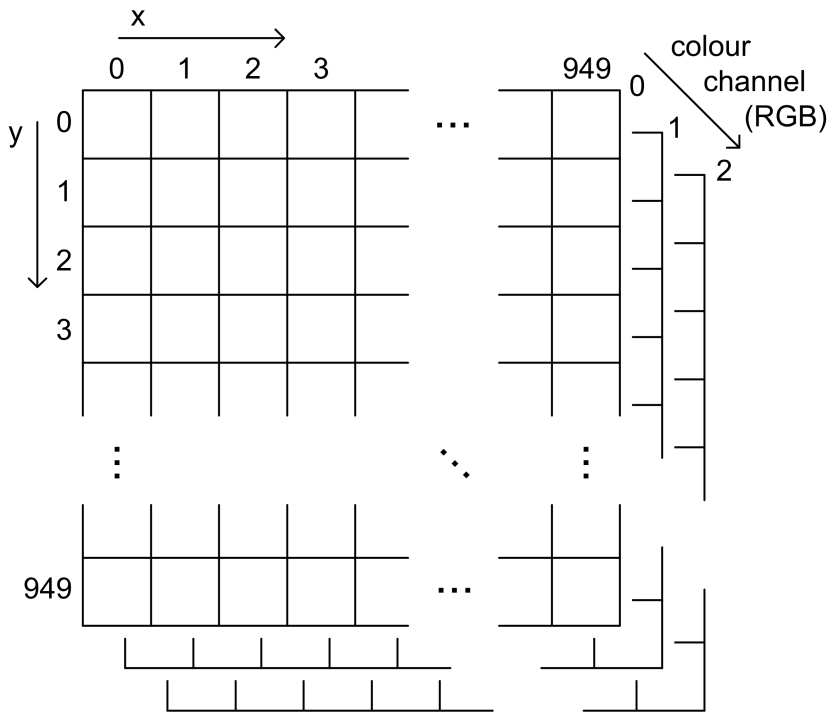

Figure 4: The pixel mapping of the 950 pixel by 950 pixels, 3 color channels, 24-bits per pixels image format used in this project.

provided in Section 3 (in the Step 1 and Step 7 descriptions). In addition, we also provide an example code that loads an image from a file, inverts the value of the colour channels (i.e. creates the negative of the image), and saves it to a file. As a starting point, make sure you can run this code without error. Then start implementing the algorithm.

3 Algorithm overview

Many computer vision algorithms are available for the detection and counting of cells in a microscopic picture. These algorithms cover a large variety of cell types (e.g., shapes, colour, etc.) and, thus, are quite complex and difficult to understand and implement. For this assignment, we have developed a simplified algorithm that allows for the detection of cells and works well for the set of test pictures we provide. Figure 5 shows the full flow-chart of the algorithm. In the following subsections, we provide a description of all the algorithm steps used in the flow chart.

Please take into account that the description of the algorithm steps is general and aims to give you an idea of the operations performed by the algorithm. Use the provided information as a starting point for your implementation of the

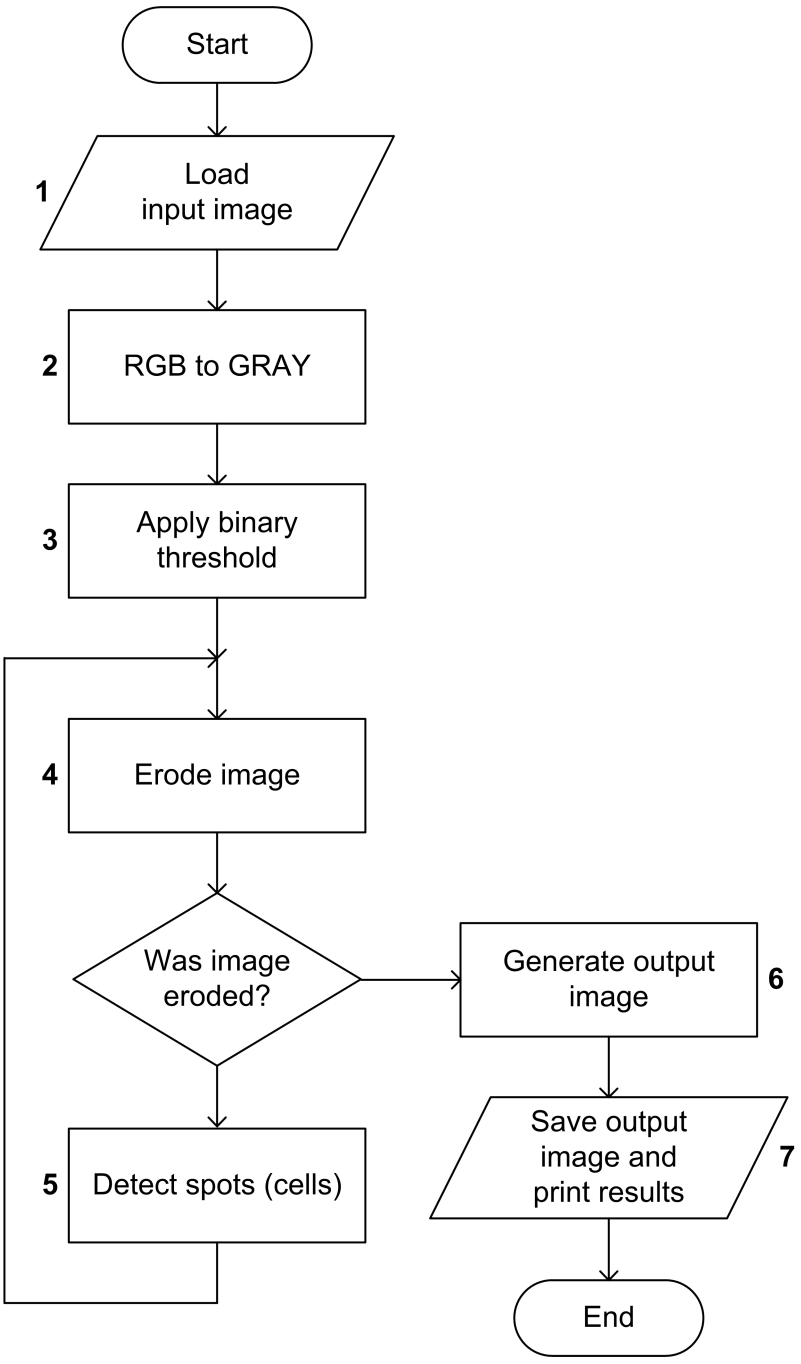

Figure 5: The flow chart of the algorithm.

algorithm. You are **expected to adapt and enhance** the provided algorithm as described in Section 5.

For each of the following steps, we recommend you implement a dedicated function. Doing this can simplify unit testing of the single function and the execution time analysis of the program. However, if you think that merging/mixing some of the steps might increase performance or deliver a better result, you are welcome to experiment with it.

Step 1: Load input image

In this step, the image provided as a bitmap file (.bmp) is loaded to memory. In order to simplify this operation, we provide the already implemented function `read_bitmap` in the library `cbmp.h`. The function loads a bitmap file into a 3-dimensional array according to the pixel mapping presented in Section 2. The function prototype is:

```
void read_bitmap(char * input_file_path,
                 unsigned char output_image_array[BMP_WIDTH]
                                                 [BMP_HEIGTH][BMP_CHANNELS]
                );
```

The argument `input_file_path` is a string (characters array) that specifies the path of the bitmap image file to be loaded. The argument `output_image_array` is a 3-dimensional array that the function fills with the values of the image pixels. Both arguments are passed by reference.

In order to use this function you must include the library in your code using:

```
#include "cbmp.h"
```

Please not that the constants `BMP_WIDTH, BMP_HEIGTH, BMP_CHANNELS` are also defined in the `cbmp.h` library as `BMP_WIDTH = 950, BMP_HEIGTH = 950, and BMP_CHANNELS = 3`. These values **should not be changed** and can be used in your code instead of explicitly writing the sizes.

Once the image is loaded in the 3-dimensional array, the following algorithm steps work on the array, without referencing the bitmap file again.

Step 2: Convert to gray-scale

In this step, the image is converted into grey-scale. A grey-scale image uses only a single byte per pixel. The purpose of this step is to reduces the information contained in the color channels and generate a single-channel image, which is easier to analyse/manipulate. The value of each pixel is the average of the values of the three color channels (red, green, blue) of the corresponding pixel in the

original image, as shown in the following equation:

$$\begin{aligned} gray\_px_{x,y} =& (rgb\_px_{x,y,red} + rgb\_px_{x,y,green} + rgb\_px_{x,y,blue})/3, \\ & \text{for } x \text{ in } [0,\ BMP\_WIDTH), \\ & \text{for } y \text{ in } [0,\ BMP\_HEIGTH) \end{aligned}$$

Thus, this step converts the 3-dimensional array of the colour image into a 2-dimensional array. Please note that this step should not modify the original colour image, since it is needed in step 7 to produce the output image.

Step 3: Apply binary threshold

In this step, the grey-scale image is converted into a binary image. A binary image is characterized by pixels that can only have 2 colours: black and white. The value of each pixel depends on the value of the corresponding pixel in the grey-scale image. If the gray-scale value is lower or equal to the foreground/background threshold $Th_{f/b}$ the binary pixel value is set to black; otherwise, it is set to white, as shown by the following equation:

$$\begin{aligned} binary\_px_{x,y} = & \begin{cases} 0 & \text{if } gray\_px_{x,y} \leq Th_{f/b} \\ 255 & \text{if } gray\_px_{x,y} > Th_{f/b} \end{cases} \\ & \text{for } x \text{ in } [0,\ BMP\_WIDTH), \\ & \text{for } y \text{ in } [0,\ BMP\_HEIGTH) \end{aligned}$$

Figure 6 shows the effect of this step on the image shown in Figure 3.

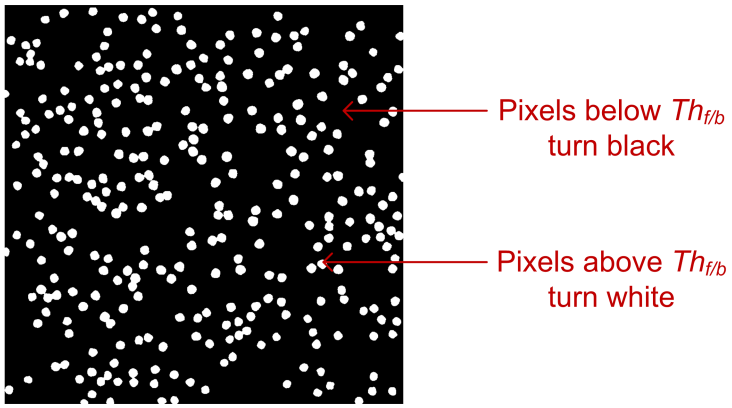

Figure 6: Binary threshold of the image shown in Figure 3.

We recommend you first implement the algorithm using a fixed threshold $Th_{f/b}$ of around 90. You can then experiment with different threshold $Th_{f/b}$ values. An optional task can be to try to calculate this threshold dynamically for each image. A good starting point for this is the Otsu's method [1,2]

[1] See article: Nobuyuki Otsu (1979). "A threshold selection method from gray-level histograms". IEEE Trans. Sys. Man. Cyber. 9 (1): 62-66, at link.

[2] See Wikipedia page at link.

Step 4: Erode image

This step performs a morphological operation called binary erosion. The basic idea of a morphological operation is to probe an image with a small and simple pre-defined shape and make decisions based on how this shape fits or misses the shapes in the image. The predefined shape is a matrix (N-by-N elements) and takes the name of 'structuring element'.

The overall effect of the binary erosion is shown in Figure 7, where the erosion

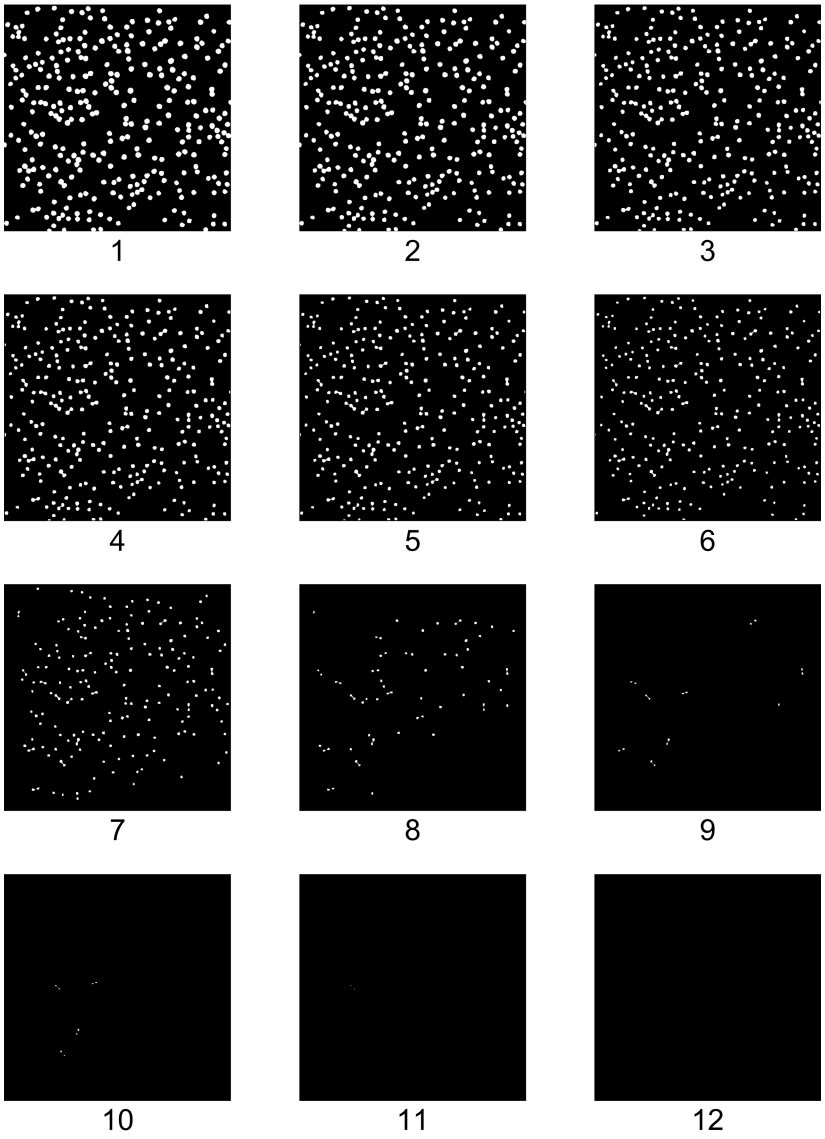

Figure 7: The effect of the binary erosion operation applied multiple times on the binary image shown in Figure 6.

operation is applied multiple times on the binary image shown in Figure 6. From the image, it is possible to observe that the white shapes (the cells) are actually eroded of 1 pixel every erosion passage (hence the name of the operation).

The erosion operation analyses all the pixels of the image one by one and generates an output image. If a pixel is black, the corresponding pixel in the output image is also black. If a pixel is white, the neighbouring pixels are taking into consideration according to the structuring element values. The structuring element used for this algorithm is the following 3-by-3 matrix.

$$structuring\_element = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

For each pixel to be analysed, the center of the structuring element is aligned with the pixel itself. If all the pixels corresponding to a '1' in the structuring element are white, then the corresponding pixel in the output image is set as

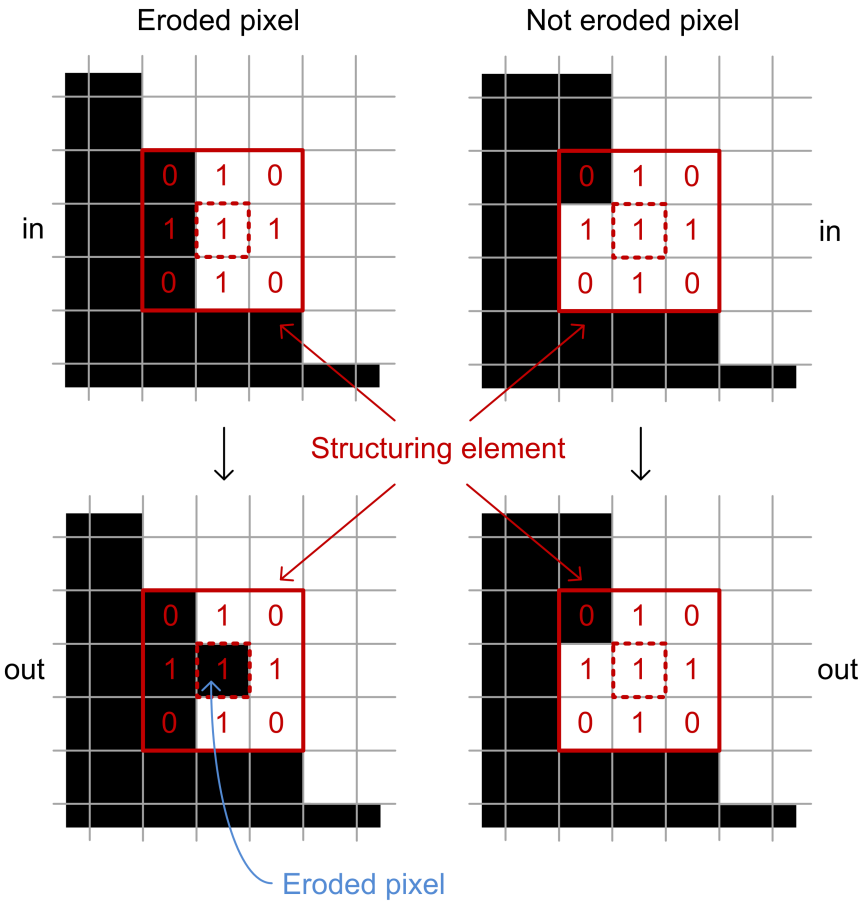

Figure 8: Erosion process for a pixel that gets eroded (turns to black) and for a pixel that survives the erosion (remains white)

.

white, otherwise, it is set as black. In other words, each white pixel 'survives' the erosion only if enough neighbouring pixels are white. The ones in the structuring element define which neighbouring pixel should be taken into consideration. Figure 8 shows this process for a pixel that gets eroded (turns to black) and for a pixel that survives the erosion (remains white).

We suggest starting developing using the given structuring element. Other sizes and placements of ones of the structuring element can also be explored to increase performance and/or detection quality.

In this step, you should also keep track if any pixel was eroded since this is the condition to exit the loop shown in the flow-chart of the algorithm in Figure 5. Once the image is completely eroded (all pixels are black), there is nothing more to detect and the loop is interrupted.

Step 5: Detect spots (cells)

In this step, the detection of cells is performed. The idea is that each white spot corresponding to a cell gets smaller with each erosion passage. If a spot is small enough it gets collected by this step and it counts as a cell.

This step analyses all the pixels of the image one by one and uses a capturing area around that pixel. The capturing area is a 12-by-12 pixel square surrounded by a 1-pixel exclusion frame. The center of the capturing area is first aligned with the pixel to be analysed. If at least one pixel is white in the capturing area and all the pixels in the exclusion frame are back, a cell detection is registered. In this case, all the pixels inside the capturing area are set to black (in order not to detect the same cell twice).

Figure 9 shows this process for a cell white spot being captured (fits in the capturing window) and for a cell white spot not being captured (white pixels are in the exclusion frame).

Other sizes and shapes of the capturing window can be explored to increase performance and/or detection quality.

Step 6: Generate output image

This step generates the output image by overlaying symbols (e.g., a cross or a circle) to the original colour input image to indicate the detected cells, as shown in Figure 3.

To generate the output image, draw the marker symbol for each set of coordinates of the detected cells by overwriting a set of pixels of the original colour image with a solid colour of your choice (e.g., red = (255,0,0)) according to your marker symbol shape.

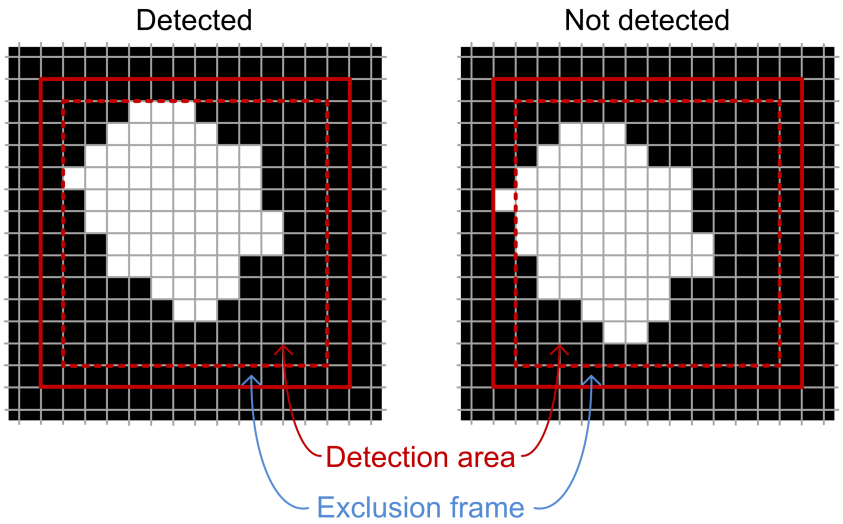

Figure 9: Detection of a cell white spot. On the left, it is captured (fits in the capturing window). On the right, it is not captured (white pixels are in the exclusion frame).

.

Step 7: Save output image and print results

In this step, the output image is saved into a bitmap file (.bmp). For this, we provide the already implemented function `write_bitmap` in the library `cbmp.h`. The function saves a 3-dimensional array representation of the image into a bitmap file. The function prototype is:

```
void write_bitmap(unsigned char input_image_array[BMP_WIDTH]
                                                 [BMP_HEIGTH][BMP_CHANNELS],
                  char * output_file_path
                 );
```

The argument `input_image_array` is a 3-dimension array of the image to be saved, and the argument `output_file_path` is a string that specifies the path of the bitmap image file to be saved. Both arguments are passed by reference.

Please note that the `read_bitmap` function used in step 1 must be called at least once before the `write_bitmap` function. This is due to some initialization procedures happening in the `cbmp.h` library.

In addition to saving the output image, the program should print to the console the total count and the list of coordinates of the detected cells.

4 Program test and analysis

This section presents the test and program analysis needed to prove the correct functionality of the implementation and determine whether the algorithm is suitable to be implemented in an embedded system,

4.1 Functionality tests

For testing the functionality of the algorithm, we provide 3 sets of cell images characterized by different level of detection difficulty: easy, medium, and hard. The detection difficulty depends on the level of overlapping of cells in the images.

You should be able to identify fairly well the cells in the easy set. However, to identify cells in the medium and hard sets you may need to enhance the provided algorithm. Please note that a certain level of error in the detection is expected and you should characterize the error and describe the reasons for it in the report.

4.2 Execution time analysis

For the execution time analysis, you will have to use code instrumentation. Source code instrumentation consists of specific instructions added to the original source code, which are used to measure the execution time of the code.

In order to access the CPU clock cycles and the execution time, you can include the time library (`#include <time.h>`) and use the clock functions as shown in the following:

```
clock_t start, end;
double cpu_time_used;

start = clock();
   /* The code that has to be measured. */
end = clock();

cpu_time_used = end - start;
printf("Total time: %f ms\n", cpu_time_used * 1000.0 /
                                                CLOCKS_PER_SEC);
```

Consider using macros to annotate parts of the code that you want to measure. The macros can be implemented such that they are easy to turn on/off using conditional compilation.

If some parts of the code stand out due to long execution time, you can try to improve it.

4.3 Memory use analysis

The algorithm uses large arrays to store images. Some of these arrays can be reused between steps, while the content of others needs to persist for the entire duration of the algorithm.

For the memory use analysis, you should perform a theoretical analysis of the lifetime of these large arrays and calculate the maximum required memory. Also, you should aim towards maximum reuse of the arrays, which correspond to the minimum memory requirements.

5 Optimizations and enhancements

As previously mentioned, you are expected to perform optimizations and enhancements to your implementation in order to improve cells detection rate, execution time, memory use, and/or other algorithm characteristics you consider relevant. Very often, when trying to improve one characteristic, another gets worse. This is expected and you should state in the report what is you aim to improve and how this affects other characteristics. You are free to adapt the algorithm, however, you should not use a completely different algorithm.

Possible optimizations and enhancements include:

- Dynamic calculation of the threshold for the generation of the binary image. As previously mentioned, understanding and investigating the Otsu's method is a good starting point.
- Efficient and optimized erosion and spot detection implementation.
- Using different size and placements of ones of the structuring element for the erosion.
- Using different size and shape for the detection area and exclusion frame for the spot detection.
- Adapting the algorithm to improve performance or detection ratio.
- **Minor optimizations:** More efficient encoding of the binary image, optimized conversion to gray-scale to avoid division, re-use of the arrays where the images are stored.

The aim of the list above is to inspire you. However, you are free to come up with your own optimizations. You do not need to implement all the optimization above. **We expect at least a couple. Please note that the 3 minor optimizations listed above count as one. If you want to implement an optimization not listed above, please discuss it with the teacher or the TAs to make sure you are on the right track.** For each optimization/enhancement you implement, you need to perform tests to prove its validity. If you have implemented an optimization/enhancement that does not give the expected benefits, describe and test why it does not work.

6 Tasks summary and report requirements

In the following, we list the tasks that you should perform in this assignment and the requirements for the short report.

These are the tasks you should perform in this assignment:

T1 Read carefully this *entire* document in order to acquire a clear and complete understanding of the algorithm to be implemented.

T2 Design and structure your code (e.g., divide functionality into functions, decide the functions prototypes, decide what buffers to use, etc.).

T3 Implement your design in C and perform unit tests to make sure new code sections are working as expected.

T4 Test the functionality and perform execution time and memory use analysis.

T5 Perform optimizations and enhancements to your implementation in order to improve cells detection rate, execution time, and memory use.

T6 Test the functionality and perform execution time and memory use analysis again to show the effect of the implemented optimizations and enhancements.

T7 Prepare a short report according to the instruction provided in the following.

The short report should not be longer than 8 pages (everything included) and should provide an overall description of your ideas and decisions for the algorithm implementation. The report should be "directly-to-the-point", without re-explaining and presenting the basic ideas already presented in this document. A template for the report is available in DTU-Learn together with the assignment files and its use is mandatory. Do not include the full code in the report. You can include some code snippets if these are relevant to explain certain aspects of the implementation.

The recent years we have experienced that students have had a hard time determining whether they are behind or on track, time-wise. Therefore, the following plan should guide you into checking that you are on track.

Date	**Expected task completion**
10^{th} September	Completed: T1 - Work on: T2
17^{th} September	Completed: T2 - Work on: T3, T4
24^{th} September	Completed: T3, T4 - Work on: T5, T6
1^{st} October	Completed: T5, T6 - Work on: T7
5^{th} October	Completed: T7

Please note that we **also** expect that you work on this as part of your preparation for the course. You will probably not be able to finish the assignment if you only use laboratory sessions to work on it.

Your work will be evaluated according to the following criteria on a base of 100 points.

Achievement	**Points**
Convert to gray-scale	5
Apply binary threshold	5
Erode image	10
Detect spots (cells)	10
Generate output image	10
Print results	5
Functional test	15
Execution time analysis	5
Memory use analysis	5
Optimizations and enhancements (including testing)	25
Report	5
Total	100

7 Notes and hints

In the following, we report some notes and hints from previous iterations of the course.

7.1 Pointer to bi-dimensional array in C

Below you can find how you declare a pointer to the image bi-dimensional array. The syntax is not intuitive.

```
//Array declaration (reserve memory space)
unsigned char binary_image_0[BMP_WIDTH][BMP_HEIGTH];
unsigned char binary_image_1[BMP_WIDTH][BMP_HEIGTH];

//Pointers declaration and
//initialization to point to the two arrays above
unsigned char (*in_image_buffer)[BMP_HEIGTH] = binary_image_0;
unsigned char (*out_image_buffer)[BMP_HEIGTH] = binary_image_1;
```