Danmarks Tekniske Universitet

Computer Systems 02132

Assignment 1

Bjarke Søderhamn Petersen
s235240

William Pii Jæger
s235245

Hugo Bede Johnson
s242071

February 9, 2025

Work distribution

We split the work evenly between group members and everyone had a input in all parts of our project both for the code and the report.

1 Design

The design process for this project involved first implementing the erosion algorithm described in the lectures and then looking for improvements that could be made to detect more cells. Our group first prioritised maximising the accuracy of our algorithm, and then focused on optimising the code by minimising memory usage and execution time.

The main challenge was isolating individual cells, especially where multiple were overlapping each other. We needed a solution which could separate the cells which were overlapping/clustered together. Since the cells typically have round shapes with a gradual increase in brightness toward their center, we decided to use a feature-based solution to define and detect them.

By constructing a filter kernel which mimics the appearance of a cell, we could apply convolution on the picture with it. To construct this kernel, we needed an algorithm that could generate a 2D representation of a typical cell. This was done with a Gaussian kernel. It has a smooth intensity gradient the decreases from the center towards the edges. This matches the features of the cells that appear in the image set provided.

Our Gaussian kernel was designed with a negative background value. Any pixels that are in the overlapping part of two cells, will get a lower value because they don't fit into a cell. On the other hand a cluster of white pixels will also get lower values because the negative background of the kernel will factor into the sum of the pixels value. That ensures the separation of the cells. (Figure 1) shows a kernel which can be utilized in our program.

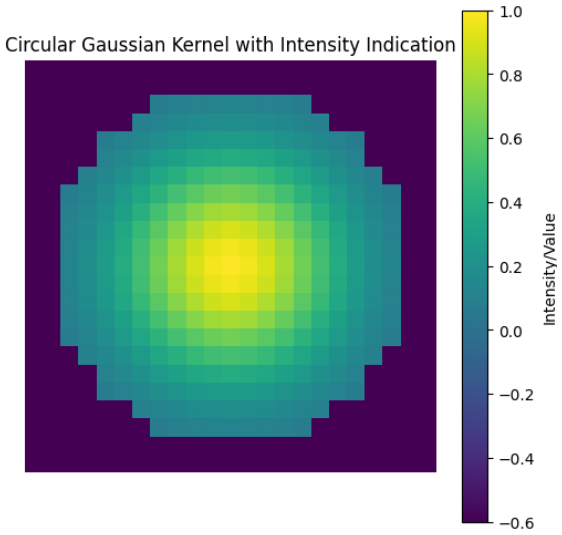

Figure 1: An example of a Gaussian kernel used in the convolution.

The convoluted image was much more effective at identifying cells, however using the erosion algorithm was still ineffective at detecting large 'regions' of cells on the hard and impossible levels. We thus developed an 'region detection' algorithm to detect cells on the hard and impossible levels.

Lastly, the parameters used in the Gaussian kernel were fine-tuned using an evolutionary algorithm to optimise the performance of the algorithm designed.

We focused on decomposing our code into independent functions and stored these in different folders. Where possible, we used integers in place of floats to maximise the efficiency of the algorithms.

To optimise memory, instead of instantiating an array each time to erode the image, we instead passed in a pointer to an existing array, and switched the images at every stage of the erosion algorithm.

2 Implementation

2.1 Convolution

Preprocessing the image with a 2-dimensional Gaussian convolution allowed us to identify the centers of the cells much more easily than with the raw image. The kernel was set up according to the equation

$$G(x, y) = \frac{1}{\sqrt{2\pi\sigma^2}}\ e^{\frac{-d^2}{2\sigma^2}}$$

Where d is the distance to the center of the kernel, i.e. $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$, where x and y are the coordinates of each pixel in the kernel.

The result of applying this convolution to the image is that pixels that best conform to the shape of the cell - i.e., those at the center of each cell become brighter, whereas those further away from the center become darker. This is visible in Figure 2.

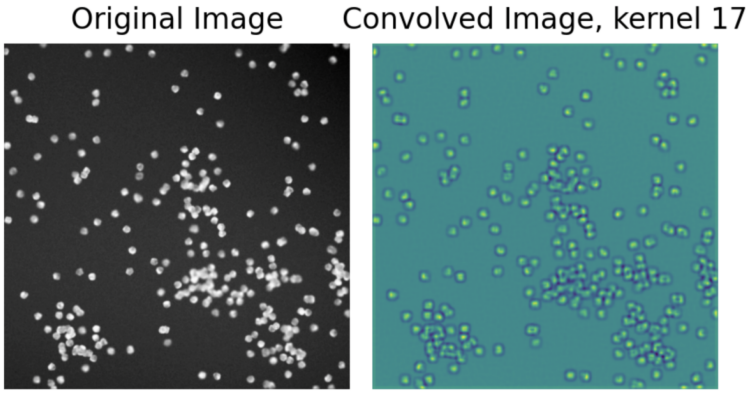

Figure 2: Left: the original image. Right: the same image after a Gaussian convolution.

2.2 Cell Detection

To detect cells we have deviated from the original proposed algorithm given to us. Instead of looking of at a square sample of our picture we find connected regions using the Depth first search algorithm.

2.2.1 Depth-first search

We have implemented depth-first search as a recursive function that counts the total amount of white pixels connected to a given pixel, and the coordinates of said pixels. To make sure

we don't end up in a recursion loop and that our algorithm is efficient we maintain an array containing the coordinates of all prior visited pixels, we this way assure we only look at a given pixel once. When we have added the new pixel to our array of pixels in the region we check the surrounding pixels for a pixel over our binary threshold that we have not visited and run our depth first search function on that pixel. This way we find all the pixels connected to each other and the total amount in the region.

2.2.2 Count region

For each pixel in our picture we check if it has not been visited already, and if it is over our binary threshold. If the pixel we are looking at is valid we perform depth-first search on it and make sure that the region size is within the expected size and area of a cell we mark the center of it and turn all the pixels black.

3 Optimizations and enhancements

3.1 Convolution

Convolution is applied over an image with a kernel. Convolution is used after we have applied a clamp threshold on the image, which removes the background. This means what the image we are left with only has a background consisting of black pixels, but keep the original values for the cells. We are only interested in calculating the values of the cells, to separate them. That is why is have introduced an optimization where we skip any pixel in an image, if said pixel has a value of 0.

In our testing we have seen that the convoluted images values never exceed values over 60.000. They are typically around 22.000. Instead of using a float array to keep this, we have instead chosen to store the values in a uint16. That halves the memory usage. The sum is calculated in float values, but is then cast to $uint_16$ as the rounding down has virtually no effect on the result.

3.2 Region detection

To make full use of our convoluted image we opted to get entirely rid of erosion. The solution we instead decided to implement was increasing the binary threshold on our convoluted image. The idea behind this process was that we this way could isolate all of our cells into individual regions since the pixels with the highest values from our convoluted image would best represent the approximate location of cells.

It was because of this change that it was necessary to change how we detected what was and what was not a cell. When increasing the binary threshold on convolution the cells don't decrease consistently across the entire image. If we therefore stuck with the original detection algorithm a cell could persists in the corner of the detection square until the cell we wanted to detect completely vanished. This problem was solved by implementing our new detection algorithm using depth-first search, described in the earlier segment.

3.3 Evolution

Depending on the kernel utilized, widely different outcomes are achieved. For our purposes, we need a kernel capable of separating cells. As mentioned prior, we utilize a Gaussian kernel. To create this the following parameters are required: (sigma, kernel size, Gaussian kernel size, negative). Additionally we also need to adjust the region detection parameters (binary threshold, area). Adjust any of these values impact the effect of the others. You could "perfect" a sigma

for a single negative value, but if that negative value changes, a new "perfect" sigma value has to be found.

We wanted to optimise the parameters of the custom kernel used in the convolution, and so we decided to create an algorithm to programmatically find these values. The steps involved were:

1. Manually mark the positions of the cells to have the 'correct' placements of the cells
2. Design an algorithm which takes the set of correct placements and evaluate them by calculating an accuracy score
3. Iterate over the desired parameters and compare the evaluations of the predictions against the 'solutions'

The implementation of the evolution was successful, but we did not have enough time to run it to find perfect values. We did find promising values, however these did not perfect perfectly on easy images. This is a result of the scoring not being strict enough. Figure 3 shows the log of evolution, which has been running for a while trying to find good kernel parameter values.

```
1 0.000000 23 7 11.490000 -0.270000 65 25
2 42.455418 23 9 11.490000 -0.270000 65 25
3 79.126717 25 7 11.490000 -0.710000 64 21
5 80.297997 25 7 11.490000 -0.710000 60 17
8 81.737686 23 7 12.219999 -0.710000 60 19
9 82.624649 23 7 11.329999 -0.710000 61 15
10 82.781517 23 7 11.359999 -0.710000 63 15
11 82.781578 23 7 11.359999 -0.710000 63 15
12 82.924866 23 7 11.359999 -0.710000 63 13
15 83.383774 23 7 11.359999 -0.710000 67 13
17 83.535271 23 7 11.359999 -0.710000 68 13
20 83.543861 23 7 11.219998 -0.710000 68 13
55 83.687431 23 7 11.339998 -0.710000 68 13
63 83.687592 23 7 11.339998 -0.710000 69 13
94 84.049675 23 6 11.059999 -0.770000 63 13
97 84.206024 23 6 11.059999 -0.770000 67 13
99 84.223564 23 6 10.249998 -0.770000 67 13
100 84.661880 23 6 10.249998 -0.770000 67 12
104 84.808022 23 6 10.249998 -0.770000 66 12
111 84.945435 23 6 11.219998 -0.770000 62 12
113 84.945442 23 6 11.219998 -0.770000 62 12
114 84.947716 23 6 11.219998 -0.770000 61 12
115 84.947769 23 6 11.219998 -0.770000 61 12
139 85.089668 23 6 11.219998 -0.770000 63 12
140 85.089676 23 6 11.219998 -0.770000 65 12
141 85.089684 23 6 11.219998 -0.770000 65 12
150 85.089828 23 6 11.219998 -0.770000 64 12
154 85.089844 23 6 11.219998 -0.770000 64 12
198 85.092690 23 6 11.219998 -0.770000 68 12
203 85.092712 23 6 11.219998 -0.770000 68 12
209 85.093094 23 6 11.219998 -0.770000 69 12
210 85.403419 23 6 11.219998 -0.770000 73 12
214 85.547325 23 6 11.219998 -0.770000 76 12
228 85.855179 23 6 11.219998 -0.770000 80 12
251 86.157303 23 6 11.099998 -0.770000 81 12
293 86.593063 23 6 11.119999 -0.770000 81 12
```

Figure 3: Evolution log. Generation, score, kernel size, kernel offset, sigma, negative, binary thresh hold, area

3.3.1 Marking the cells

We designed a small program in Python that displays one of the images and allows us to marks the cells on it by clicking. The program then logs the outputs to a file for later use. We read the files in our integration tests. We have utilized [Sta] code and modified it to read the lines from files.

3.3.2 Evaluation algorithm

The algorithm takes a set of predicted coordinates $P = (x_p, y_p)$, and a set of 'ground truth' coordinates $T = (x_t, y_t)$. It then does the following:
- Calculate the distance between all cells in P vs T.

- Select each pair $\{(p,t) \mid p \in P, t \in T\}$ such that the distance between p and t is within a threshold distance (so as to cut off predictions which are too far away), and sort these pairs in increasing order of distance.
- Match the pairs p and t until the maximum number of pairs is found where any p or t appear at most once.

This algorithm returns the correct matches between P and T in order to give a score representing how accurate the set of predictions were.

3.3.3 Iterate over parameters

Initially, a brute force algorithm was used to iterate over every combination of parameters. However, eventually there were so many parameters that this approach became computationally expensive and as such, an evolutionary algorithm was used to find optimal parameters. Each generation contained 30 sets of parameters and the top 10 were chosen for the next generation.

4 Test and analysis

4.1 Unit tests

Unit tests were implemented for the convolution, erosion and region detection algorithms by testing them on a smaller subset of numbers which were calculated by hand. The output of the algorithm is compared to the manual calculation when the tests are run. Unit tests were written for each of the three algorithms implemented - erosion, convolution, and island detection.

4.2 Integration tests

The chosen algorithm (erosion, convolution or regions) are run on each of the images in easy, medium and hard, and the number of cells counted is compared to the number of cells counted previously. The integration tests also optionally output images with crosses drawn above the predicted coordinates for visual confirmation and evaluation of the algorithm.

For more detailed integration testing, the function ‘write_frame‘ was created to automatically convert a 2d array of integers to bmp format and write it to the output folder, as well as update the output path for the next image (e.g. from output_00 to output_01). This allowed us to easily debug our program as a whole by letting us see the output of each iteration of the erosion or region detection algorithms.

4.3 Memory usage

The memory footprint of individual variables used for calculations are very small in comparison to the array's. Therefore we have only calculated the sizes of the arrays. We have optimized for memory by for example specifically using a

We work with images of 950 by 950. All the images, 3, are "input image", "image to analyze" and "convoluted image". The input image is: `(char)*(BMP_WIDTH)*(BMP_HEIGHT)*(BMP_CHANELS) = 1*950*950*3 = 2,707,500`

Image to analyze: `(char)*(BMP_WIDTH)*(BMP_HEIGHT) = 1*950*950 = 902,500`

`902,500+2,707,500 = 3,610,000`

Convoluted image: `(uint_16)*(BMP_WIDTH)*(BMP_HEIGHT) = 2*950*950 = 1,805,000`

Kernel: `(float)*(kernel_width)(kernel_height) = 4*23*23 = 2,116`

Visited array is `(char)*(BMP_WIDTH)*(BMP_HEIGHT) = 1*950*950 = 902,500`

Cell centers: `MAX_CELLS_DETECT * int * 2 = 350*4*2 = 2800`

Pixel size: `MAX_REGION_PIXELS * int * 2 = 2000*4*4 = 32000`

In total this comes out to 6.05 MB. Factoring in the additional variables used for calculations we could expect it to go up to 6.1 MB total.

If we assume that we stored our "image to analyze" in an int array, and treated each bit as black or white, we could use:

`950*950 = 902,500` bits needed `902,500/8 = 112,812.5` bytes

This could also be used for the visited array.

If we were to free up the input picture while handling the cell detection we could decrease the amount of memory used. In reality, memory fragmentation occurs as soon as we release our initial input image. If the memory was allocated sequentially, when we release our input image, since other parts of memory will consume that space, we can no longer allocate that input image there again. If we allocate all the consistently needed items and those that consume minimal memory, we hope that when the input image is loaded into memory, it will occupy one of the front elements of the allocated memory. That way, when it's freed, we can hopefully reclaim space for other things.

4.4 Results

When we run our integration tests we get the following results:

```
CONVOLUTION EROSION
EASY           300/300 300/300 300/300 300/300 300/300 300/300 300/300 299/300 300/300 300/300 300/300 300/300 |  9/10 | Cell Avg: 299 | FAIL | Avg time: 362.00 ms
MEDIUM         282/292 286/292 282/293 277/289 274/289 284/290 280/293 263/286 272/295 278/295 |  0/10 | Cell Avg: 277 | FAIL | Avg time: 306.50 ms
HARD           282/292 268/286 265/279 272/288 278/292 283/291 278/296 283/298 281/295 276/290 |  0/10 | Cell Avg: 276 | FAIL | Avg time: 296.00 ms
IMPOSSIBLE     253/276 248/283 257/276 256/278 269/281 |  0/5  | Cell Avg: 256 | FAIL | Avg time: 276.00 ms
CONVOLUTION REGION DETECTION
EASY           300/300 300/300 300/300 300/300 300/300 300/300 300/300 300/300 300/300 300/300 300/300 300/300 | 10/10 | Cell Avg: 300 | PASS | Avg time: 385.90 ms
MEDIUM         290/292 292/292 291/293 287/289 284/289 289/290 290/293 277/286 288/295 292/295 |  1/10 | Cell Avg: 288 | FAIL | Avg time: 399.20 ms
HARD           288/292 282/286 280/279 283/288 287/292 287/291 295/296 291/298 292/295 285/290 |  0/10 | Cell Avg: 287 | FAIL | Avg time: 359.30 ms
IMPOSSIBLE     268/276 258/283 267/276 271/278 275/281 |  0/5  | Cell Avg: 267 | FAIL | Avg time: 358.40 ms
EROSION DETECTION
EASY           300/300 298/300 300/300 299/300 300/300 300/300 300/300 299/300 300/300 299/300 300/300 |  6/10 | Cell Avg: 299 | FAIL | Avg time: 279.40 ms
MEDIUM         261/292 265/292 260/293 266/289 246/289 269/290 251/293 243/286 256/295 262/295 |  0/10 | Cell Avg: 257 | FAIL | Avg time: 239.80 ms
HARD           259/292 228/286 237/279 251/288 251/292 262/291 258/296 259/298 255/295 245/290 |  0/10 | Cell Avg: 250 | FAIL | Avg time: 223.20 ms
IMPOSSIBLE     214/276 211/283 216/276 216/278 227/281 |  0/5  | Cell Avg: 216 | FAIL | Avg time: 225.00 ms
```

We here denote the amount of cells found for each picture compared to the total amount of cells in the picture. (It should here be noted that the total amount is hand counted by us for each picture). We count it as a success if we find all the cells, and a failure if we don't. We run the tests for 3 different algorithms. The top one is for the convoluted image where we run erosion and the cell detection algorithm as it was given to us. The second one is our final algorithm. This is where we run convolution and use our region detection algorithm to detect where cells are located. The final one is erosion on a non-convoluted image with basic cell detection exactly as it was given to us in the assignment description.

If we look at the cell averages across the 3 different algorithms we can see a clear difference. The best performing one is Convolution region detection, the second best is Convolution erosion and the worst one is Erosion detection. This shows that our enhancements have had a direct correlation with the detection rate of cells. The drawback to the optimisations we made is that since we run a more complicated algorithm our execution time has also increased. If we compare the relative execution time between the different algorithms compared to Erosion detection we get the following:

Set	Convolution erosion(%)	Convolution region detection(%)
Easy	29.56	38.12
Medium	27.81	66.47
Hard	32.62	60.98
Impossible	22.67	59.29

Table 1: Relative increases in execution time compared to Erosion detection

4.5 Failure points and future improvements

Despite the convolution improving the algorithm's ability to detect clumps of cells, our algorithm still struggled with tightly packed cells, especially where the outline of each cell is not clearly visible. A suggested improvement for this could be to implement a bottleneck algorithm, as used by other groups in our cohort. Combining these techniques would allow us to effectively analyze tightly packed cells in two distinct ways.

Secondly, during testing we realised that some values of the kernel are prone to falsely detecting cells instead of noise - this is due to having a binary threshold that is too low. Because of this, we were forced to find a balance between keeping as much detail as possible from the convoluted image, and making sure noise is not included in our image. One possible solution is to remove background noise from the image initially with a low binary threshold, and then perform convolution on the de-noised image.

References

[Sta] gsamaras (Stackoverleaf User). *C read file line by line*. URL: `https://stackoverflow.com/a/64406356`. (accessed: 06.10.2024).

ChatGPT was also partly used in the writing of the make file and for fixing certain errors between Mac and Windows