Map Filter
Algorithm Documentation
Request Code Sample - Brain Corporation

Jose Pablo Sanchez Rodriguez

March 19, 2019

Abstract

This is the documentation for the requested code sample for the Autonomous Navigation Software Engineer position at Brain Corporation.

There are situations on which maps have phantom pixels, product of false-positives. These need to be filtered. In Section 1 I explain how it occured to me to address this problem keeping in mind that this should run in the most efficient way as the robot has limited resources. In Section 2 I explain how the algorithm works. In Section 3 I show and explain the testing results and I explain how the user should use and run this code.

If you just want to run the code and samples, please refer directly to section 3.

1 Introduction

The task is to write a filter that takes as its input a 256x256 map image, where each pixel has a value of 0, 127 or 255 indicating CLEAR, UNEXPLORED and OBSTACLE respectively. The filter will scan the image and change the value of any pixel that is marked as OBSTACLE to CLEAR if, and only if, that pixel is located at the center of some 41x41 pixel subset of the map, and all of the other pixels in that subset are marked as CLEAR.

At a first glance I thought of two options. The first one was to fully check each pixel of the 256x256 with a 41x41 subset of pixels. This solution would be complete and easy to write but slow to compute because the 41x41 subset of pixels would need to be checked if a pixel of the image is candidate to be cleared. The second one came after the phrase "divide and conquer". This one, divides the image into a grid. THe 41x41 subset of pixels is also divided. The intention of doing this is to find either candidate or non-candidate pixels to be clared faster without cehcking every single pixel of both the image and the subset of pixels.

2 The filter algorithm

For this particular case (i.e a 256x256 input image and a 41x41 subset of pixels) the divided image is a 6x6 grid. The remaining pixels at the left and bottom are also considered in the last column and last row check.

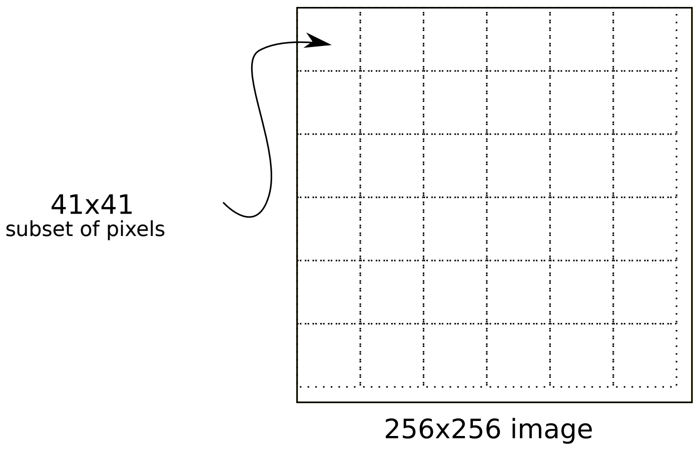


Figure 1: Image divided into 6x6 grid.

The subset of pixels was divided in four quadrants (i.e Q1, Q2, Q3, and Q4).

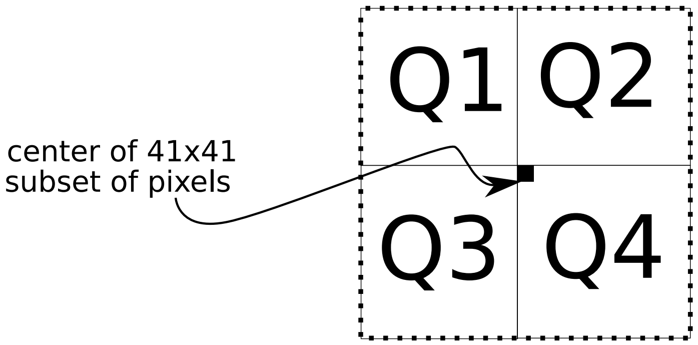


Figure 2: 41x41 subset of pixels divided into quadrants.

2.1 void filter(int img[])

If you want to use this filter, the only thing you need to do is to call the filter method.

The principal method, called "filter", checks the 6x6 grid stablishing the origin of the 41x41 subset of pixels at the upper-left corner. The algorithm checks with a left to right riority first and then from up to down. The first origin point is the (0,0) which corresponds to the first element of the 6x6 grid, located at the upper-left part of the grid. The second origin point is the (0,1*41) is right at the left side of the first one.

2.2 void check6()

The next method, called "check6", checks if the center of the 41x41 subset of pixels is a pixel marked as an "OBSTACLE". If true, then the algorithm verifies if all the other pixels are marked as "CLEAR". If false, the algorithm searches for another "OBSTACLE" pixel inside the subset of pixels.

2.3 bool check1(), bool Q1(), Q2(), Q3() and Q4()

This next method verifies if all the other pixels of the 41x41 subset of pixels are marked as "CLEAR". It returns true only if all the members of each quadrant are "CLEAR". It returns false when not

”CLEAR” in some of the wuadrants. Each Q1() corresponds to the first quadrant and so on. Each of those methods returns false when finding a pixel different from zero or ”CLEAR”. This prevents from having to check all of the pixels inside the subset of pixels.

2.4 void obs_check()

This method searches for ”OBSTACLE” pixels, candidates to be cleared if the center of the 41x41 subset of pixels is not a candidate. This method searches for ”OBSTACLE” phantom pixels inside the subset of pixels. Here, the algorithm will check the quadrants of the subset of pixels.There is a method called obsQ1() that searches for those pixels inside Q1 and so on for the other quadrants. If the algorithm finds only one pixel marked as ”OBSTACLE” inside the quadrants, it is a potential candidate to be verified by check1() method and marked as ”CLEAR”.

There are cases on which is not necessary to check all the quadrants because a 41x41 wouldn’t fit. That situation happens next to the image borders. Therefore, to avoid the algorithm to waste time, I defined 9 cases for this problem. The section coloured in yellow is also reached from subsets of pixels whose center is in the last column or row.

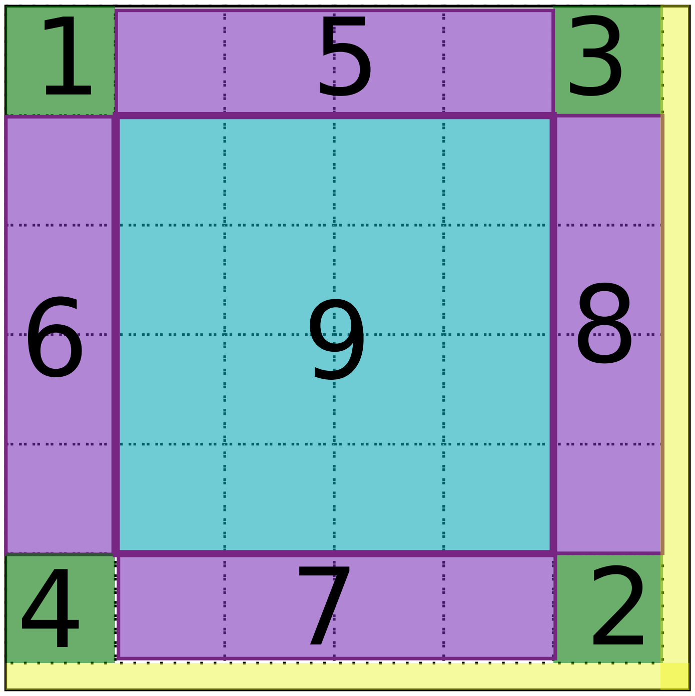


Figure 3: Nine cases location

Case 1: The first element of the 6x6 grid.
In this case, the algorithm only searches for one pixel on Q4.

Case 2: The last element of the 6x6 grid.
In this case, the algorithm searches for one pixel on ALL the quadrants.

Case 3: The last element of the first row of the 6x6 grid.
In this case, the algorithm only searches for one pixel on Q3 and Q4.

Case 4: The first element of the last row of the 6x6 grid.
In this case, the algorithm only searches for one pixel on Q2 and Q4.

Case 5: All the elements in the first row, except the first and the last.
In this case, the algorithm only searches for one pixel on Q3 and Q4.

Case 6: All the elements in the first column, except the first and the last.
In this case, the algorithm only searches for one pixel on Q2 and Q4.

Case 7: All the elements in the last row, except the first and the last.
In this case, the algorithm searches for one pixel on ALL the quadrants.

Case 8: All the elements in the las column, except the first and the last.
In this case, the algorithm searches for one pixel on ALL the quadrants.

Case 9: Al the other elements of the grid.
In this case, the algorithm searches for one pixel on ALL the quadrants.

In some of the cases, specially those which are next to the right and bottom borders, the algorithm verifies that the 41x41 fits correctly.

Everytime a candidate is found, the method check1() is called to verify if it's actually a phantom pixel and therefore cleared.

3 Testing

To test the correctness of my solution, I prepared a set of 8 images. The main.cpp file is used to test the algorithm. First, I wrote a code to read an image and convert it to a 2D array (modified from [1]). Then, an object is created and the filter method is called. The input to the filter method is the image represented as a 2D array, and the output is the same 2D array but filtered. To verify the solution, I wrote another code to convert the 2D array to an image and show it to the user.

3.1 Compiling

To compile the code, you need to enter the root directory:

```
$ cd Sanchez\ Rodriguez-Jose\ Pablo/
```

Then, being in such directory, compile using the following code:

```
$ g++ -std=c++11 src/main.cpp -o filter
```

3.2 Running some tests

Once it's compiled, the tests are run as follows and a window showing the result should prompt.

```
$ ./filter test1.pgm
$ ./filter test2.pgm
$ ./filter test3.pgm
$ ./filter test4.pgm
$ ./filter test5.pgm
$ ./filter test6.pgm
$ ./filter test7.pgm
$ ./filter test8.pgm
```

Everytime a test is run, a file called $filtered\_ < name - of - image > .pgm$ is created.

4 Results

Here there are the results of the 8 tests. I present 3 images for each test. The first one is the test image. The second one show some boxes and circles. Green box means that the pixel must be cleared. Red boxes means that the pixel is not inside a "CLEAR" subset. Yellow box means that the pixel is inside an "UNEXPLORED" subset or that there is an "UNEXPLORED" pixel inside a "CLEAR" subset. Yellow circle means that such pixel is not even considered as candidate to become "CLEAR" because the 41x41 subset does not fit inside the image. The third image is the filtered image.

4.1 Example 1

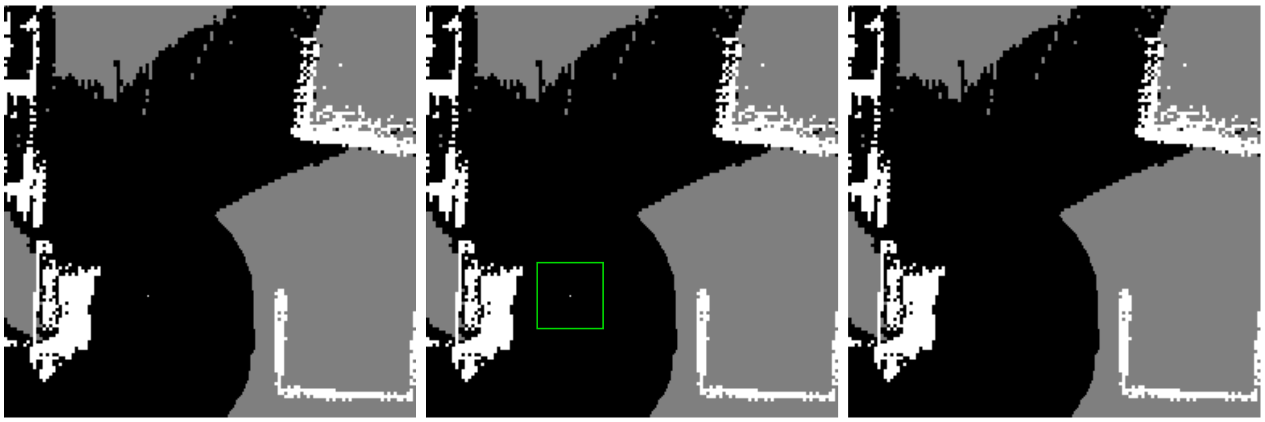

(a) (b) (c)

Figure 4: (a) Testing image. (b) Testing image with boxes. (c) Filtered image.

4.2 Example 2

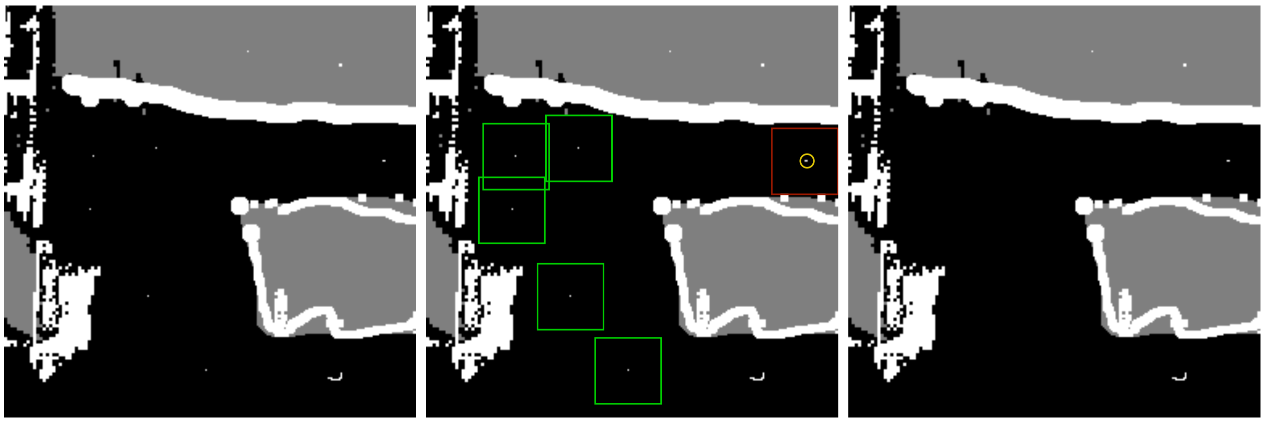


Figure 5: (a) Testing image. (b) Testing image with boxes. (c) Filtered image.

4.3 Example 3

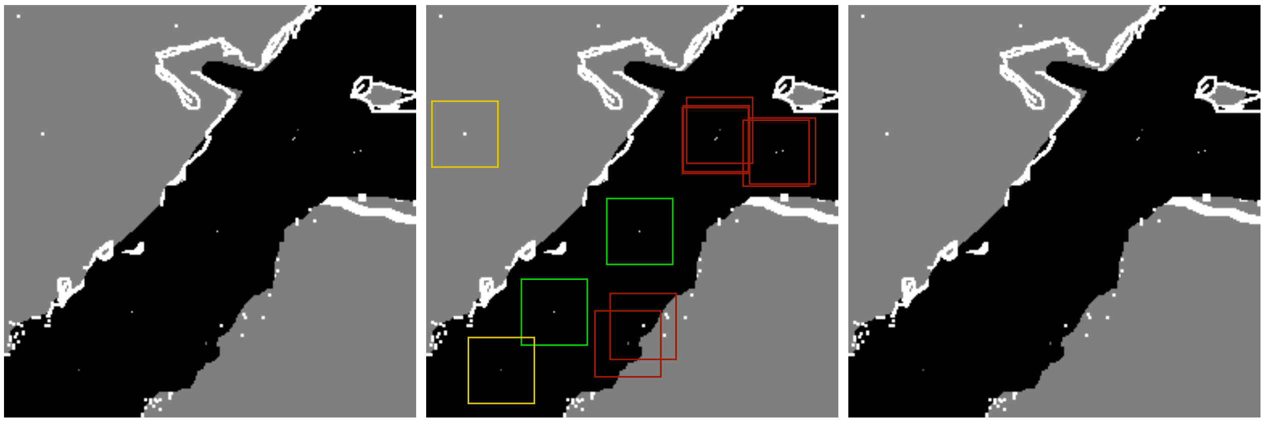


Figure 6: (a) Testing image. (b) Testing image with boxes. (c) Filtered image.

4.4 Example 4

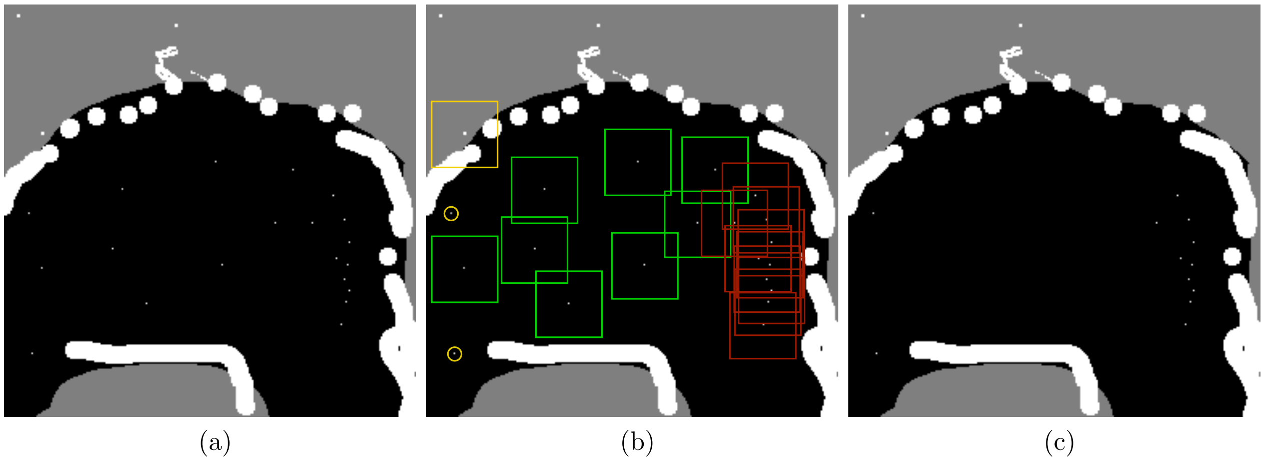


Figure 7: (a) Testing image. (b) Testing image with boxes. (c) Filtered image.

4.5 Example 5

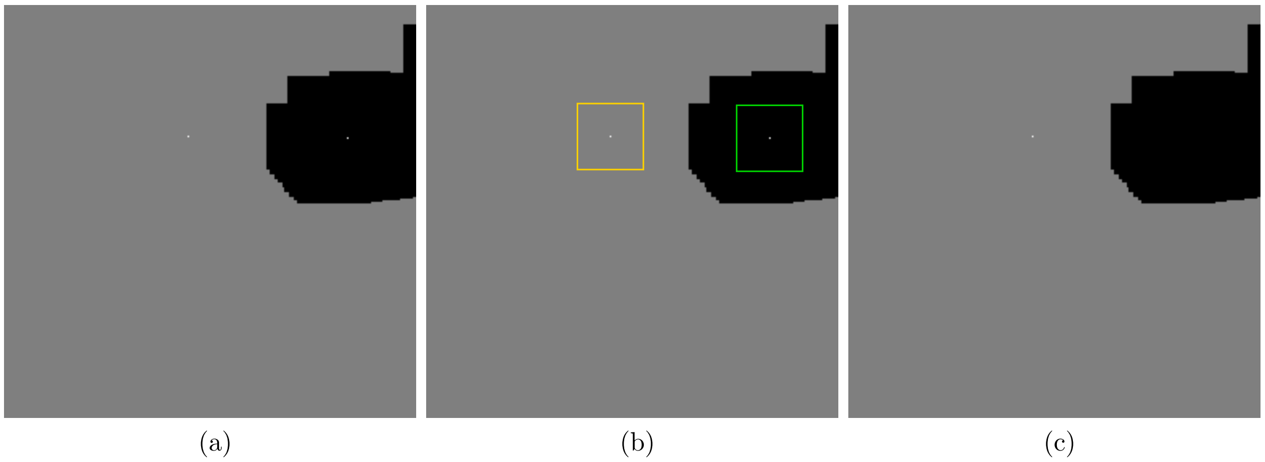
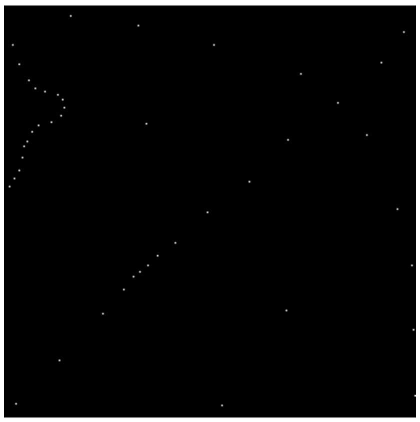
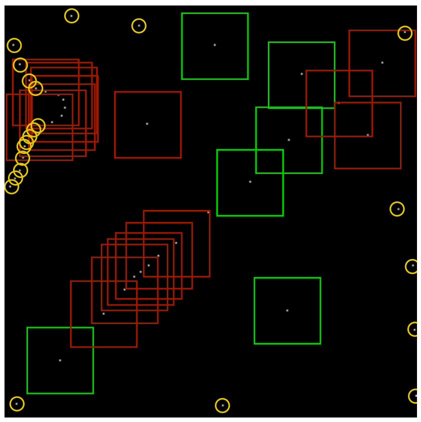


Figure 8: (a) Testing image. (b) Testing image with boxes. (c) Filtered image.

4.6 Example 6


(a)


(b)

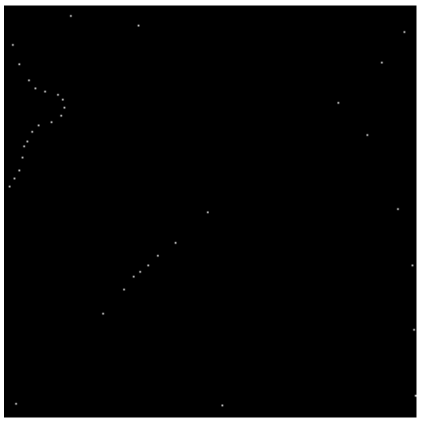
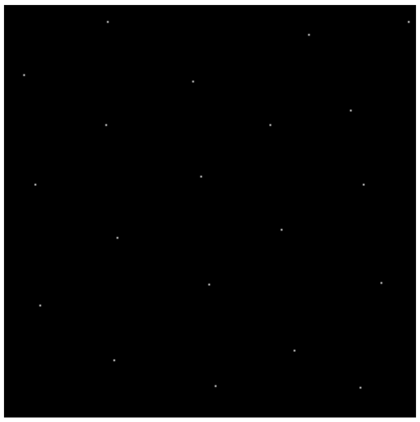
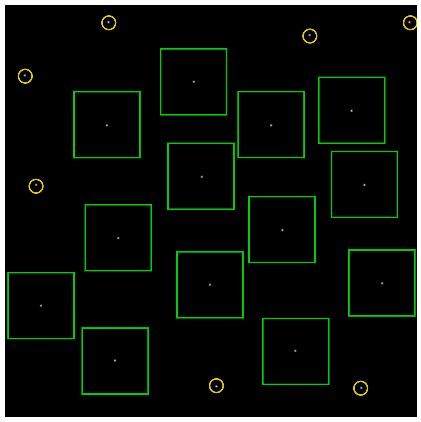

(c)

Figure 9: (a) Testing image. (b) Testing image with boxes. (c) Filtered image.

4.7 Example 7


(a)


(b)

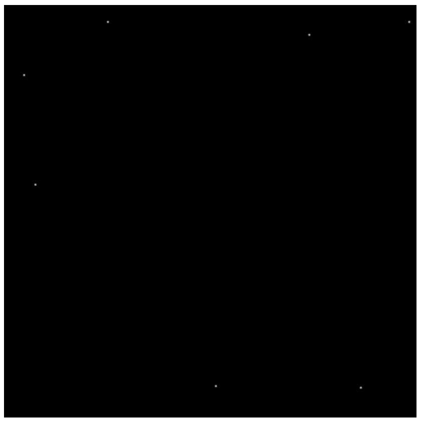

(c)

Figure 10: (a) Testing image. (b) Testing image with boxes. (c) Filtered image.

4.8 Example 8

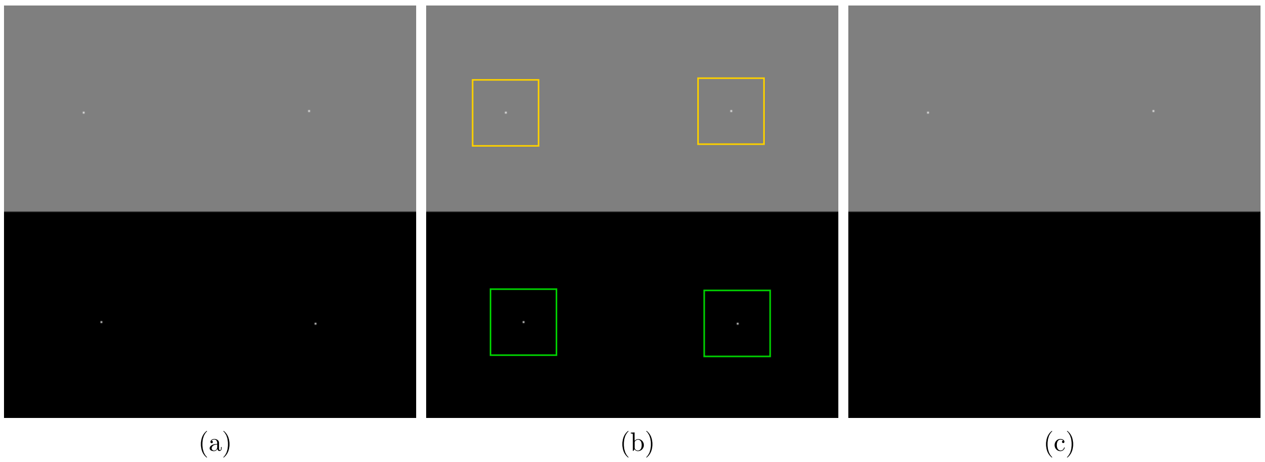


Figure 11: (a) Testing image. (b) Testing image with boxes. (c) Filtered image.

References

[1] Benc. Read pgm file to array https://stackoverflow.com/questions/8126815/how-to-read-in-data-from-a-pgm-file-in-c.