General Program for Edge Detection

Como Adrien and Demeersseman Daniel

Department of Computer Science
University Claude Bernard Lyon 1
Villeurbanne, 69100, France

{adrien.como & daniel.demeersseman}@etu.univ-lyon1.fr

Abstract : Edge detection systems are one of the most used computer analysis algorithms. They are implemented inside self-driving cars but also in medical systems and other applications like detection of default in products on a manufacturing line. Edge detection includes a variety of mathematical methods that aim at identifying edges, curves in a digital image at which the image brightness changes sharply or, more formally, has discontinuities.

Key words : Edge detection, convolution, kernel, Canny , Sobel.

I. Technical Introduction

This project uses the openCV library for image pixel access, image opening and writing. It also uses state of the art STL template libraries in C++. A makefile is provided for simple software execution and to obtain quick results. This software can also be used with CMake for ease of convenience. EVerything else has been developed by hand from scratch.

A. Aim of this project

This project has always kept in mind that large images take a lot of time to process. We have therefore always kept the code as simple and clean as possible to maximise the speed at which the edge detection kernels are applied to the image. This allows us to provide a low cost, immediate results, edge detection solution. Basic parameters are already provided so that anyone including non-initiated people can still use the edge detection system. Several outputs are provided at the end of the execution of the software. They correspond to each step of the edge detection process.

B. Organisation

Our software is divided in 2 separate sections. A main.cpp that recuperates all necessary parameters from the user. And the Image class where most of the magic happens. The image class is broken into two pieces. The first part is used to initialise all the parameters (arrays, cv::mat…etc). The second part is where all the logic is executed behind the scene to produce an output that informs the user of all the edges that might be present in the image.

All the code is stored in the src/ folder. The images in the img/ folder. The output will be stored where the user wants when using the commands parameters.

C. Recommendations

It is recommended to use the provided blurring functions. The edge detection works best when the Gaussian blur kernels of size 3 or 5 are used. Results are significantly better and edges can be distinguished from the background more easily. This edge detection algorithm is strong enough for any type of image as long as the user has a basic knowledge of what the image looks like so that he can tune the parameters of the threshold to produce the best results possible.

D. How to run

```
make clean && make && clear && ./bin/main a b c d e f g
```

a : input path to image (img/…) (works with png, jpg, jpeg)
b : 0 < x < 5 : Gaussian blur kernel size (0, 3 , 5)
c : kernel choice
- 0 : Prewitt
- 1 : Sobel
- 2 : Kirsch

d : time to display openCV windows when executing
e : 0 : bi-directionnal, 1 : multi-directional
f : high threshold
g : low threshold

A good example is : *make clean && make && ./bin/main img/bugatti.jpg 5 2 100 1 40 90*

II. Description of the algorithm

A. Basic knowledge

An image if made of pixels. Pixels each having 3 values (r,g,b). But for this algorithm we will only work on a grayscale image. The formula to transform each pixel is **[1]** :

- rgb to grayscale = 0.299.r + 0.587.g + 0.114.b

This is the first part of the edge detection algo. We immediately slightly blur the image after the grayscale formula has been applied.

To find edges, convolution kernels are needed. They are generally composed of a 3*3 matrix. This matrix is then "swept" across the whole image. If the values in the matrix are normalised and respect certain properties, edges will slowly form. Usually the edges in an image are at many different angles. A rotation of the kernel is needed to find all edges. Simply rotating by steps of 45° or 90° angles the kernel, several times can suffice depending on the precision needed.

-1	0	+1
-2	0	+2
-1	0	+1

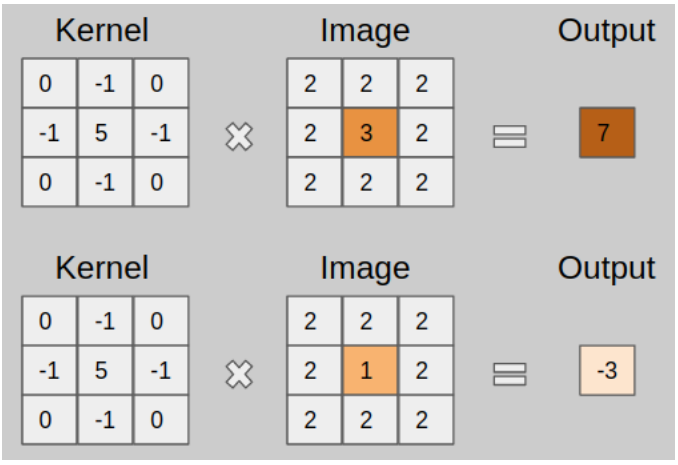
Gx

+1	+2	+1
0	0	0
-1	-2	-1

Gy

As shown here (above) the Sobel kernel is once applied in the horizontal (Gx) manner and then rotated 90° anticlockwise to find edges in the vertical axis (Gy). If the kernel had been rotated by 45° we would have done so 3 more times to obtain 0°, 45°, 90°, 135° kernels. This is the multidirectional method.

As shown below are two examples on how to apply the kernel to the image. We can see different outputs for different values of the image.


Kernel

0	-1	0
-1	5	-1
0	-1	0

Image

2	2	2
2	3	2
2	2	2

Output

7

Kernel

0	-1	0
-1	5	-1
0	-1	0

Image

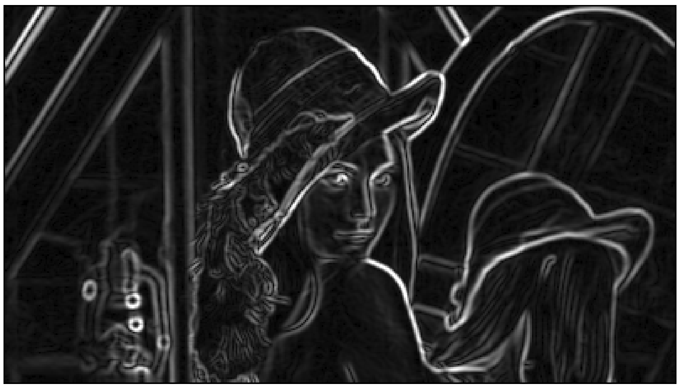
2	2	2
2	1	2
2	2	2

Output

-3

B. Edge detection in detail

This is an example of contour that can be found using a multi-directional method.

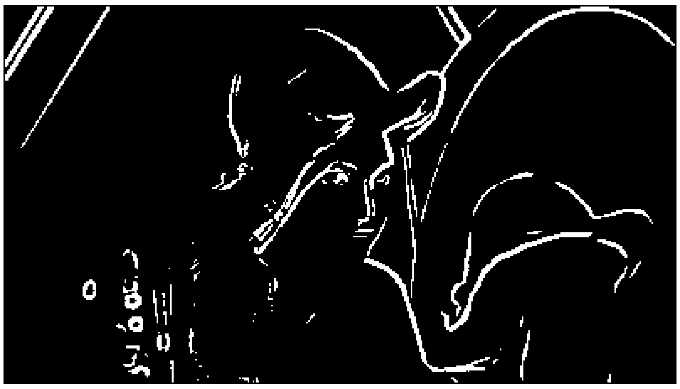


We can see that contours are sharp and vibrant which is a guarantee of a good quality. We can then apply threshold algorithms to only keep the most prominent contours. Two methods we developed in our project.

A basic one consisting of a value chosen by the user, that sets to 0 all pixels under this threshold and lightens up all other pixels with higher values than this threshold. This works well to keep only the most significance of contours.

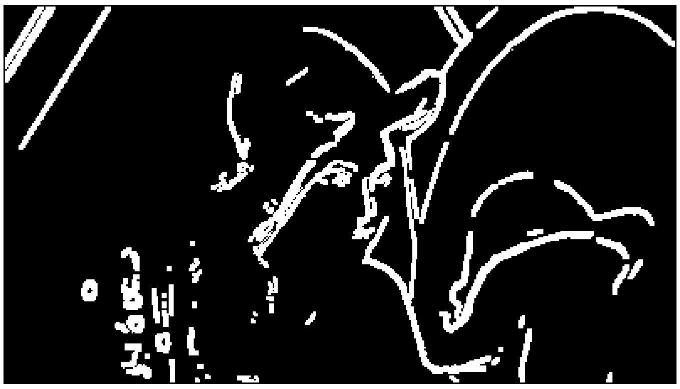
Another method which is slightly more complex, uses two thresholds. A low cut-off threshold where pixels under this threshold, will be set to 0. A high cut-off threshold value, where all pixels over this value will be set to 255. All pixels, between these two values, will have to be further analysed. If their neighbours have at least one pixel that belongs to a contour then this pixel is set to 255. If no neighbours belong to a contour, pixels are set to 0.

An example for the first method can be found here.



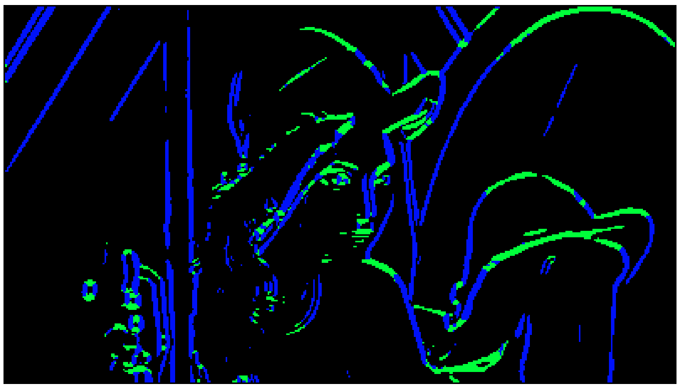
We can clearly see all the edges. Of course the higher the threshold is, the fewer edges will be kept. Only the most remarkable will still stand out. If the value is set to low, most edges will be kept. It is up to the user to choose a threshold that corresponds to his own needs. Several tries can be done in quick succession as our algorithm is fast.

The second method produces an output of this sort. This method is called Hysteresis threshold.

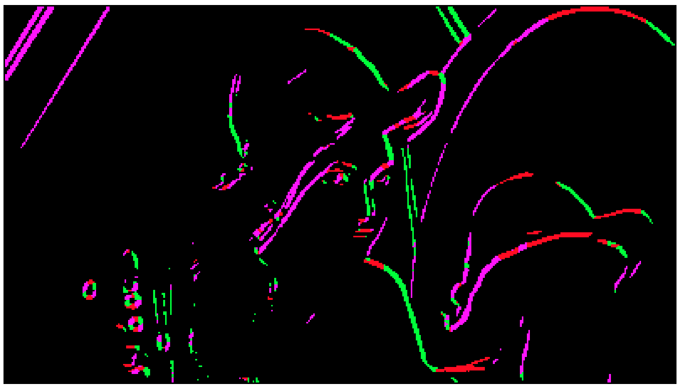


Our algorithm also proposes a quick way to preview angles at which edges were found in the image. We drew all the edges found at different angles in different colours.

As shown below, we use 2 colours because edges were found here in bi-diretionnal mode. X and Y axis. All contours found on the X axis are drawn in blue. The contours at the Y axis are green.

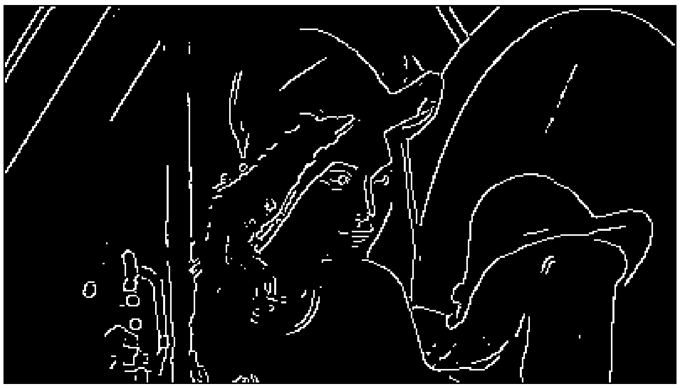


In this photo, we can see 4 colours. This is because we applied 4 different kernels to the image and chose for each contour the maximum of the 4 kernels. We then applied the colours accordingly. This is the multidirectional method of the rotation of the same kernel 4 times, each at 45° each.



C. *Refining*

The images produced are also refined to produce more consistent and better looking images. This is done by looking for each direction of the contour (0, pi/4, pi/2, 3pi/4 for multi-directional), each adjacent pixel. If the center pixel has a higher value than both, then it is set at 255. If not at 0. Here is an example of refining.



IV. Speed and Power

A. *Makefile optimization*

Several optimizations are presented in the image edge detection project. In the makefile and CMake we can see the use of "-Ofast" **[2]** which astoundingly speeds up the process by applying many compiler optimizations.

C. *Large array initialization optimization*

Array flag initialization was done with std::memset() which is significantly faster than traditional methods. **[3].** All arrays use pointers for fast access to the internal values.

D. *Out of bound testing optimization*

We never have to check whether we are inside the image as we don't work on the image borders. This is for two reasons. We cannot apply the convolution kernel on border pixels. And removing these pixels saves quite some time, especially when the image is of a large size. Up to two percent is gained this way.

H. *Function and procedure calls*

Each function or procedure call bears a certain cost depending on how many parameters are given. That's why we have limited the amount of calls to the minimum.

V. Conclusion

The algorithm works like intended. Results are correct and show great performance. This algorithm can definitely be used with most images and produce good enough results to be used as the input in some other software.

VI. Acknowledgment

We would like to thank the teacher for giving us tips on how to better our algorithm during the classes and explaining in great detail the intricacies of edge detection, convolution, kernels and stuff.

VII. References

[1] http://support.ptc.com/help/mathcad/en/index.html#page/PTC_Mathcad_Help/example_grayscale_and_color_in_images.html

[2] https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

[3] https://cplusplus.com/reference/cstring/memset/