
Project 2: Edge Detection

The deadline for this project is **Monday, 27th of May**, end of day, anywhere on earth (AoE)¹. You can find more information about the project submission in appendix A.

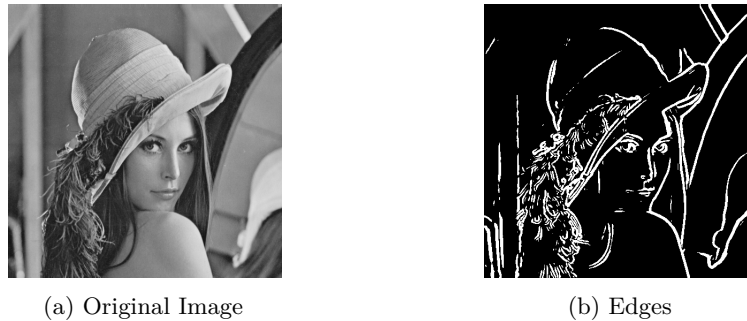


Figure 1: Edge Detection Example

1 Overview

In this project, you will implement a C program that can detect edges within images. The goal is to find the borders between homogeneous areas of similar hues and grey-scale values. Detecting edges finds numerous applications in object detection, such as in self-driving cars, artistic image filters, or image compression algorithms.

In section 2, you can find a high-level description of the edge detection algorithm. Section 3 and the code comments in your source code files document the individual functions you need to implement. The appendix gives more technical information and answers several frequently asked questions.

2 Edge Detection

The foundation of the edge detection algorithm lies in computing the discrete derivatives of the image. Although you may only know derivatives from analysis, the same idea applies here: the absolute value of the derivate measures how different nearby values are.

In the context of image processing, we are concerned with the difference of the grey-scale values of neighboring pixels of the image. A difference of two neighboring pixels' grey-scale values that exceeds some threshold indicates an edge in the content of the image. In order to ignore less important edges of the image, we will initially blur the image. Both computing the discrete derivative as well as blurring the image can be represented using the mathematical operation *convolution*.

The Algorithm Briefly, your edge detection algorithm will consist of the following steps:

- (a) reading the original image
- (b) blurring the image through convolution with a Gaussian kernel
- (c) computing the discrete derivative in x and y direction through convolution with Sobel kernels
- (d) computing the magnitude of the gradient vector at each pixel
- (e) determining edge pixels by comparing the grey-scale values against the threshold
- (f) outputting the image as black and white image

The individual steps of this procedure are illustrated in fig. 2.

¹https://en.wikipedia.org/wiki/Anywhere_on_Earth

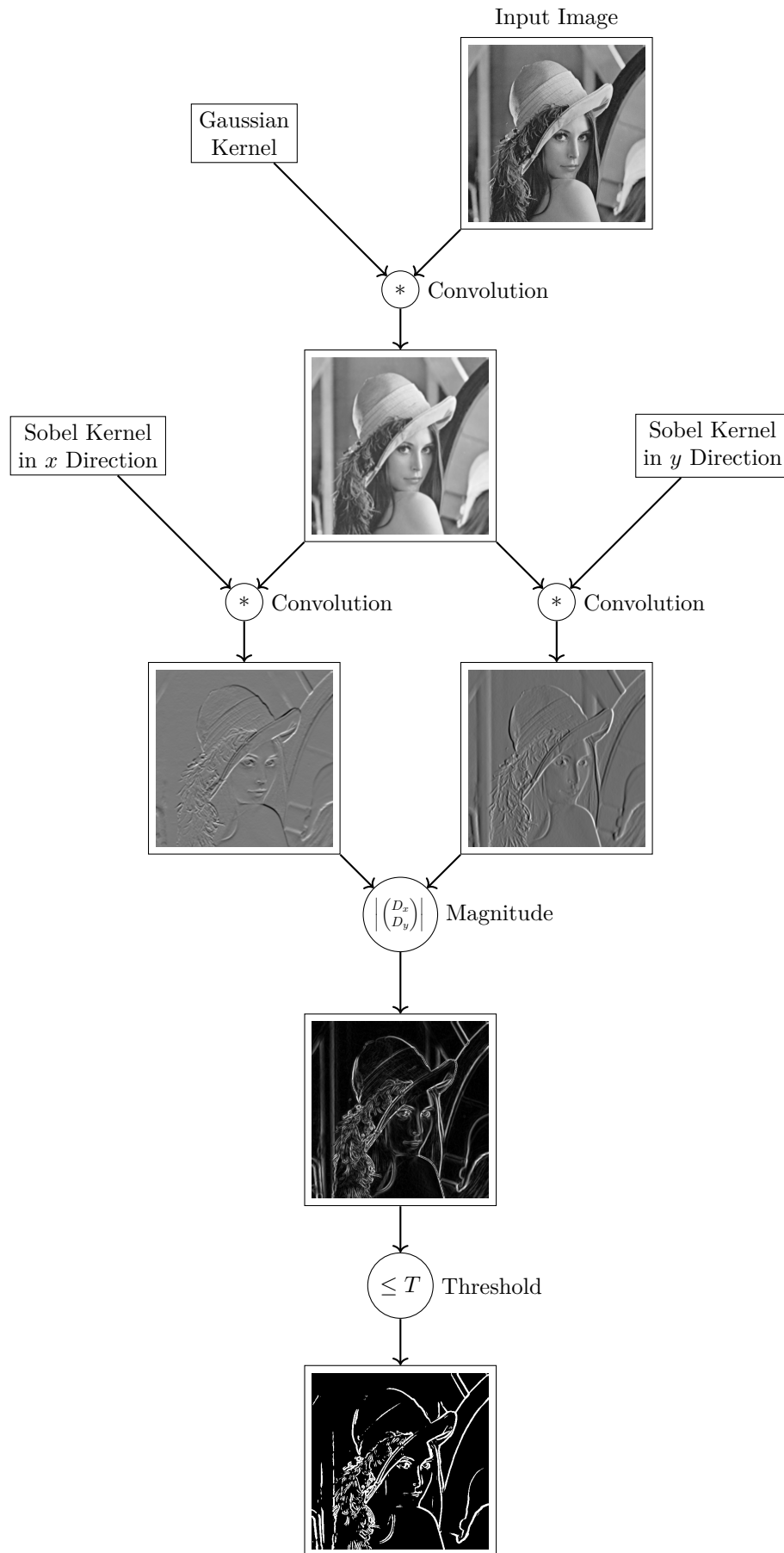



Figure 2: Overview of the Edge Detection Algorithm

3 Assignments

The assignments are mostly sorted by their difficulty and along the contents of the lecture, not by their order in the algorithm. We recommend solving them in the given order. All functions to be implemented are documented in the respective `.h` files.

Assignment 1: Threshold (1 Point)


In `apply_threshold`, you will recolor every pixel whose gray-scale value exceeds the threshold `T` white (255). If it does not exceed the threshold ($\leq T$), color it black (0). 

Implement the function `apply_threshold` in the file `image.c`.

```
1 void apply_threshold(float *img, int w, int h, int T)
```

The parameter `float *img` contains the image in which you should recolor the pixel values. The parameters `int w` and `int h` contain the width and height of the image. `T` is the threshold value to compare the pixels against.

Assignment 2: Magnitude of Gradient (1 Point)

After computing the discrete derivatives $D_x(x, y)$ and $D_y(x, y)$ at some point (x, y) , you must compute the absolute value of the gradient. The gradient is a vector, whose each component is a partial derivative. 

Use the following formula to compute the magnitude of the gradient $\text{grad}(x, y)$ at pixel (x, y) :

$$|\text{grad}(x, y)| = \sqrt{D_x(x, y)^2 + D_y(x, y)^2}$$


Implement the function `gradient_magnitude` in the file `derivation.c`.

```
1 void gradient_magnitude(float *result, const float *d_x, const float *d_y, int w, int h)
```

The parameter `float *result` contains the output image, into which you should enter the gradients' magnitudes. The parameters `float *d_x` and `float *d_y` contain the arrays which contain the derivatives in x and y direction. Lastly, `int w` and `int h` contain the width and height of the image. Note that the lengths of the arrays `d_x` and `d_y` are both $w \cdot h$.

Note: The standard library function `sqrt` will probably be useful to you.

Assignment 3: Scaling of Gray-Scale Values (1 Point)

Computing the derivatives and the magnitude of the gradient introduces values outside of the interval from 0 to 255. In order to output the image in the `pgm` format, you will have to scale these values back to the interval from 0 to 255. Let max and min be the maximum and minimum pixel value of the image, and $P(x, y)$ be the value of pixel (x, y) . We define the scaled pixel value $P_S(x, y)$ as follows: 

$$P_S(x, y) = \frac{P(x, y) - min}{max - min} \times 255$$

In the edge case where $max = min$, a black image should be output.

Implement the function `scale_image` in the file `image.c`.

```
1 void scale_image(float *result, const float *img, int w, int h)
```

The parameter `float *result` contains the output image, into which you should enter the scaled gray-scale values. The parameter `float *img` contains the input image to be scaled. Lastly, `int w` and `int h` contain the width and height of the image.

Assignment 4: Convolution (3 Points)



The convolution of an image and a matrix \mathcal{M} , the so-called *convolution kernel*, is a process that computes a new image, where every new pixel is computed as the weighted sum of its neighboring pixels. For example, if you compute the new pixel values as the average of the original pixel's neighbours values, your result is a blurred version of the original image.

Formally, the two dimensional convolution P_C between an $w_P \times h_P$ image with pixels $P(x, y)$ and the $w_M \times h_M$ convolution matrix \mathcal{M} with entries $\mathcal{M}(i, j)$ is defined for each pixel (x, y) as

$$P_C(x, y) = \sum_{j=0}^{h_M-1} \sum_{i=0}^{w_M-1} \mathcal{M}(i, j) \cdot P(x - a + i, y - b + j),$$

where we denote the central x coordinate of the matrix \mathcal{M} by a and the central y coordinate by b . So, (a, b) is the center of the image. Throughout the project, you may assume that the dimensions w_M and h_M are both odd, so the center coordinates are uniquely defined.

The convolution between two matrices is commonly denoted using an asterisk, e.g. $P_C = P * \mathcal{M}$.

Edge Handling The formula for $P_C(x, y)$ is well defined, if the coordinate $P(x - a + i, y - b + j)$ lies within the image. There are several reasonable strategies how one can define the value of $P(x - a + i, y - b + j)$, in the case it lies outside of the image.

In the project, you will use the *mirror* edge handling strategy: If you must access a pixel that is outside of the image, pretend that the image extends past its boundaries as a mirrored copy of itself and use the corresponding pixel on the other side of the “mirror”. In the following example, we will refer to the image extended beyond its boundaries as P_m .

Example Figure 3 illustrates an example in which a 5×5 image P is being convolved with a 3×3 matrix \mathcal{M} . For visualization, P has been extended to P_m using the mirror edge handling strategy, indicated by the yellow cells of P_m . All cells depicted in yellow have coordinates outside of the original image's coordinates, including negative coordinates. To obtain the value of a yellow cell, it may be necessary to mirror the original pixel twice. In order to compute $P_C(3, 0)$, the matrix \mathcal{M} is overlaid over P_m , centered at pixel $(3, 0)$. According to the formula, the entry $P_C(3, 0)$ is computed as the sum of the products between the values $P_m(i, j)$ and $\mathcal{M}(i, j)$, for all i and j within the overlapping region. You can find more great visualizations on Wikipedia.

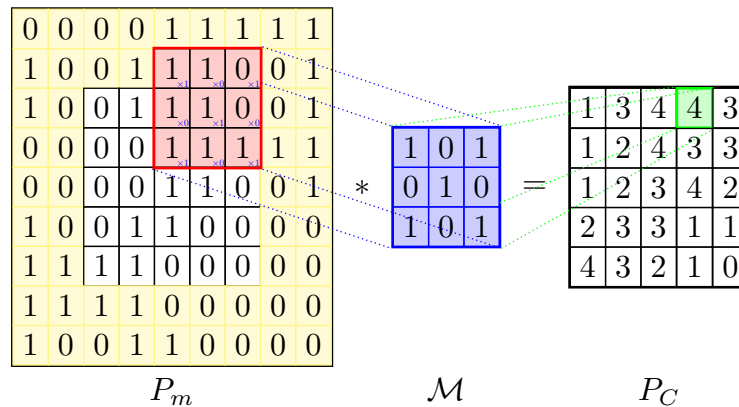


Figure 3: Convolution Example using the *Mirror* Edge Handling Strategy

Firstly, think about how to mirror a coordinate, if it is outside of the image. Implement the function

```
1 float get_pixel_value(const float *img, int w, int h, int x, int y)
```

in `image.c`, which for a given image `float *img`, width `w`, height `h` and coordinate `(x, y)` should compute the corresponding pixel value of the extended image $P_m(x, y)$. Note that, if `(x, y)` lies within the original image, the result should be the same as $P(x, y)$. Use `get_pixel_value` to implement

```
1 void convolve(float *result, const float *img, int w, int h, ...)
```

in `convolution.c`.

Assignment 5: Reading and Writing the Image (4 Points)

In this assignment, you should implement functions to read/write an image from/to a file. To this end, we will use the .pgm image format, documented in appendix C.



Robust Reading from a File Implement your reading function robustly, that is, in such a way that it can detect potential errors in the input file, and return a `NULL` pointer in that case. Make sure to cover the following cases:

- (a) the file does not exist, or `if fp == NULL , prin error`
- (b) invalid header data, particularly non-positive image dimensions
- (c) too few or too many pixels
- (d) gray-scale values outside of the range 0 to 255

Note: Your function to write an image to a file need not be robust.

Utility Functions Since the dimensions of the image will only be known during runtime of the program, you must dynamically allocate memory to store the image (see appendix C). Additionally, this means that you must free that memory before termination of your program. To achieve this, implement the following two auxillary functions

```
1 float* array_init(int size)
```

and

```
1 void array_destroy(float *m)
```

in `image.c`. The function `array_init` should dynamically allocate an array of floats, representing the grey-scale values of the image. Complementary, the function `array_destroy` should free such an allocated array. Then, use your auxillary functions to implement

```
1 float* read_image_from_file(const char *filename, int *w, int *h);  
2 void write_image_to_file(const float *img, int w, int h, const char *filename);
```

in `image.c`. The function `read_image_from_file` expects the path to the input image in the parameter `filename`. Parameters `int *w` and `int *h` are pointers to where the dimensions of the read image should be stored. The function should return a pointer to the dynamically allocated array that contains the image data. Remember to free dynamically allocated data, even in the case of an error, and close all opened files before returning from the function.

The function `write_image_to_file` expects an image in the `img` parameter. Parameters `w` and `h` specify the image's width and height. `filename` contains the file name to which the image should be written. Round pixel grey-scale values down to the nearest integer before writing them to the file. Remember to close opened files before returning from the function!

Assignment 6: The main function (2 Points)

Your final assignment is to combine the previous functions to the algorithm presented in 2. Complete the indicated sections of the `main` function



```
1 int main(int const argc, char **const argv)
```

in `main.c`. As described in appendix B, you do not have to implement command-line argument parsing yourself. The comments inside the function document what you need to implement. Remember to free dynamically allocated memory after use.

Note: Intermediate values must be scaled to the interval from 0 to 255 before writing them to an image file. The algorithm itself should do intermediate computations with the unscaled values.

A Infrastructure, How to pass, Assorted Hints

Using dGit

Use git to clone your personal project repository from dGit, for example by entering

```
1 git clone ssh://git@dgit.cs.uni-saarland.de:2222/prog2/2024/students/project-2-{your matriculation number}
```

Submit your project using `git push` until Monday, 27th of May, end of day, anywhere on earth (AoE). If you need a refresher on how to use git, consult the materials from the Programming 2 Pre-Course or additional material online².

Grading

Similar to the way you obtain your project, you will submit the project through git. The last commit pushed to our server before the deadline will be considered for grading. Please ensure that the last commit actually represents the version of your project that you wish to hand in.

Once you `git pushed` your project to our server, we will automatically grade your implementation on our server and give you feedback about its correctness. You can view your latest test results by following the link to the Prog2 Leaderboard in dCMS Personal Status page. We use three types of tests: **public** tests, **regular** tests and **eval** tests. The public tests are available to you from the beginning of the project in your project repository. You should use them to test your implementation locally on your system. Once you *pass a sufficient number of public tests*, we will run the regular tests for your implementation, disclosing to you the names of the regular tests and whether or not you successfully pass them.

Eval tests are similar to regular tests, but run only once after the project deadline. They directly determine the number of points your implementation scored.

Notes

- (a) Attempts at tampering with our grading system, trying to deny service to, purposefully crash, or extract grading related data from it will (if detected) result in your immediate expulsion from the lecture and a notification to the examination board. In the past, similar attempts have led to us filing criminal complaints with the police.
- (b) Do not create new files or new functions, and neither change existing functions' signatures.
- (c) We will not only test the correctness of your entire project, but also that of individual functions. After implementing a feature, make sure that you pass the respective public tests.
- (d) The `void` casts, e.g. `(void) result;` in `convolution.c` are solely used to suppress compiler warnings and should be removed by you.
- (e) You may use functions of the C standard library, `math.h`, in particular `double sqrt(double arg);`.
- (f) Carefully read the documentations of the functions in the respective `.h` files.
- (g) Ensure that your program frees dynamically allocated memory after use.

²Such as <https://git-scm.com/book/en/v2>

B Building, Running and Testing

Compiling the Project

You may build the project by executing the provided **Makefile**, e.g. by running **make** in your WSL, or terminal. Upon successful build, a binary executable **edgedetection** will be created in the directory **bin/**. Alternatively, you may compile and link the files in the **src/** directory manually.

Command-Line Arguments

The **edgedetection** application implements the following command-line syntax:

```
1 edgedetection -T <threshold> <image file>
```

where **-T <threshold>** sets the threshold value relevant for step (e) of the algorithm described in section 2 and **<image file>** is the path to the input image. The file **main.c** documents where the (intermediate) output images will be stored.

To launch the application after successfully **makeing** the project from the root directory of your project with a threshold value of 100 and the test image **img_P.pgm**, you should execute

```
1 ./bin/edgedetection -T 100 test/data/input/img_P.pgm
```

Parsing the command-line arguments is already implemented in **argparse.c** of your project template and may be ignored. After parsing the command-line arguments, the respective values are stored in the global variables **image_file_name** and **threshold**, see **argparser.h**.

Testing

With the command **make tests**, you can run the **public** tests from the root directory of your project.

As long as large parts of your implementation are still missing, it may be desirable to only run individual tests. In order to run an individual test, pass the name of the test to the command-line flag **-f** of the **run-tests.py** script. To only run the test **public.read_image_from_file.imgbroken1**, you should execute

```
1 python3 test/run-tests.py -f public.read_image_from_file.imgbroken1
```

You can obtain a complete list of available tests by executing

```
1 python3 test/run-tests.py -l
```

Note: **make tests** will compile your project, if necessary, before testing. When running tests using **run-tests.py**, you must manually recompile your project before testing.

C Image Representation

We use the *portable graymap format* as input and output file format for the images used in this project. Portable graymap files typically end in `.pgm` and solely contain human-readable text for its pixels.

Image Header The first few bytes of a `pgm` file are always structured in the same way. We call this structure the *header* of the file type.

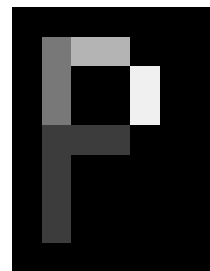
The header of a `pgm` file consists of the following ASCII text – in order:

- (a) the two characters “P2”, indicating the file type
- (b) some whitespace
- (c) the width of the image
- (d) some whitespace
- (e) the height of the image
- (f) some whitespace
- (g) the maximum ‘brightness’ value, which for us will always be 255

Image Data After the header, a `pgm` file contains the integer gray-scale values of the image’s pixels. All pixel values are integers that lie in the range from 0 to 255 (inclusive). Individual pixel values are separated by at least one character of whitespace (e.g. spaces ‘`_`’, tabulators ‘`\t`’, newline characters ‘`\n`’, or carriage returns ‘`\r`’). Figure 4 illustrates an example. For better visualization, we break the line after every seventh pixel; generally however, this need not be the case.

```
P2
7 9
255
0 0 0 0 0 0 0
0 120 180 180 0 0 0
0 120 0 0 240 0 0
0 120 0 0 240 0 0
0 60 60 60 0 0 0
0 60 0 0 0 0 0
0 60 0 0 0 0 0
0 60 0 0 0 0 0
0 0 0 0 0 0 0
```

(a) Textual Representation



(b) Graphical Representation

Figure 4: PGM Format Example

Representing an Image in a C Program Images are represented as arrays of `floats`. The elements of the array correspond to the gray-scale values of the pixels of the image. To represent a two-dimensional image of height h and width w , we use a one-dimensional array of length $w \cdot h$. The pixels are stored row-wise inside the array: a pixel with coordinate (x, y) is stored at position $y \cdot w + x$ of the array.

Note: Since there is no way (in C) of finding the length of an array in memory, all functions working with the image’s data must receive w and h as parameters.

D Using Visual Studio Code

We recommend you use Visual Studio Code (VSCode) to work on the project. Of course, you may use any other editor of your choice. It is important, that you manage to compile your project according to the steps described in appendix B, so you can replicate the testing infrastructure of the server locally. You can use the integrated terminal of VSCode to execute commands in the commandline.

Moreover, we are providing you with several configuration files (which VSCode recognises inside the `.vscode` directory) to facilitate your work in the Editor. For example, you can compile the project with the keyboard shortcut `Ctrl + Shift + B`. Pre-defined *tasks* allow you to execute common compilation and testing commands via *Terminal* → *Run Tasks*.

To interactively debug your project in VSCode, you need to

- install the *GNU Debugger*, `gdb`, on your system, as well as
- the VSCode extension `C/C++` from the VSCode extensions marketplace.

After successful installation, you can debug your project's `main` function by using the keyboard shortcut `Ctrl + Shift + D` and executing the pre-selected start configuration "Debug Main".