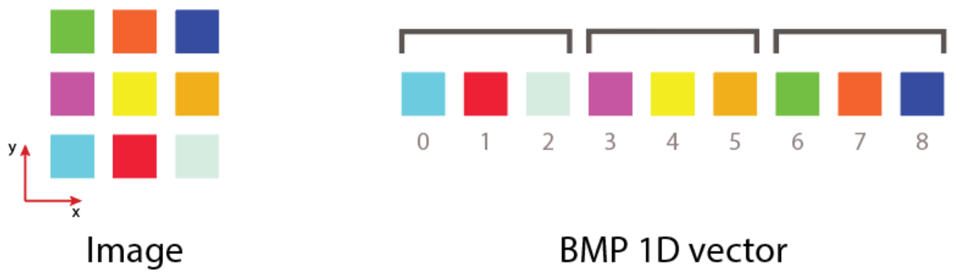
1. Background

The C++ program takes a .bmp image as input and allows the user to perform a Canny edge detection operation on it. This gives the user a binary (black and white) map of where the edges in the image are found.

How bmp data is stored and accessed



The above image shows a simplified 3 by 3 image where each pixel is a coloured square. For bitmaps, the origin of the image is bottom left—contrary to the conventional top left for images.
The *pixel data* is stored in a 1D array or vector, however, which is shown on the right side.

To find a pixel in the 1D vector, we need to know the width and height of the image. The index of a pixel in the vector can be found as follows:

$$index\ of\ pixel\ with\ coordinates\ (x,y) = (y\ *\ im\_width\ +\ x)$$

Pixel $x = 0,\ y = 1$ of a 3 by 3 image would have the following array index:

$$index\ of\ pixel\ with\ coordinates\ (0,1) = (1\ *\ 3\ +\ 0) = 3$$

Keep in mind, each square (pixel) represents three values: the RGB data. In bitmaps, this data is ordered in the blue-green-red (BGR) order, instead of the conventional red-green-blue (RGB).

To calculate the array index of a pixel that is stored in the 1D array, *with* these BGR channels, the following formulas can be used:

$$index\ of\ B\ value\ of\ pixel\ with\ coordinates\ (x,y) = channels * (y\ *\ im\_width +\ x) + 0$$

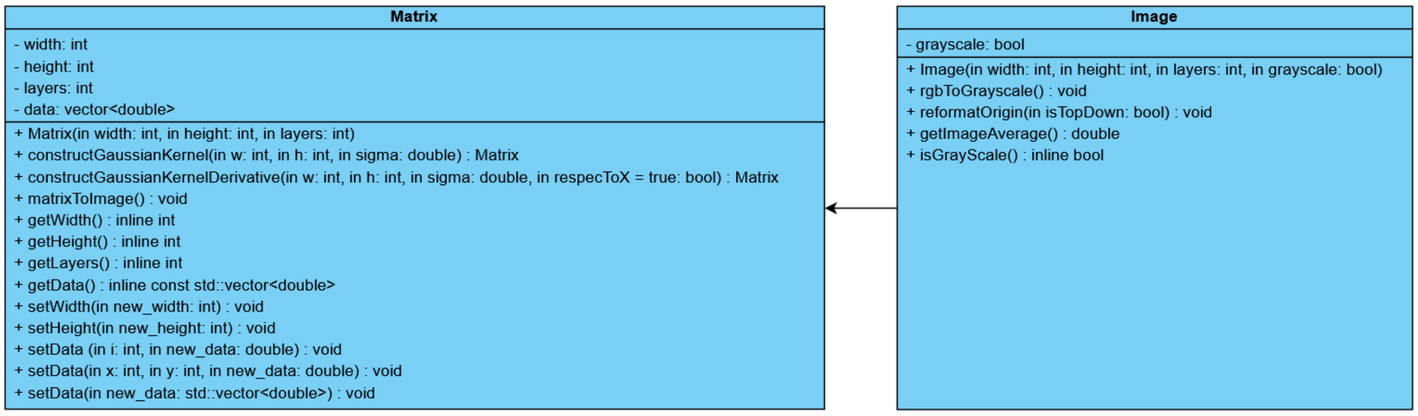
$$index\ of\ G\ value\ of\ pixel\ with\ coordinates\ (x,y) = channels * (y\ *\ im\_width +\ x) + 1$$

$$index\ of\ R\ value\ of\ pixel\ with\ coordinates\ (x,y) = channels * (y\ *\ im\_width +\ x) + 2$$

Note the use of the BGR format instead of RGB (since bitmaps are used in the program). The array index of the green channel value of pixel $x = 0,\ y = 0$ (assuming three-channel BGR) is then calculated as follows:

$$index\ of\ G\ value\ of\ pixel\ with\ coordinates\ (0,0) = 3 * (0\ *\ 240 +\ 0) + 1 = 1$$

The index of the blue channel value would then be 0 (because blue comes before green in BGR) and the index of the red channel value would be 2. **Converting $(\mathbf{x}, \mathbf{y})$ image coordinates to the array index as shown above is used extensively in the program and therefore it is useful to become familiar with the conversion.**


Matrix
- width: int
- height: int
- layers: int
- data: vector<double>
+ Matrix(in width: int, in height: int, in layers: int)
+ constructGaussianKernel(in w: int, in h: int, in sigma: double) : Matrix
+ constructGaussianKernelDerivative(in w: int, in h: int, in sigma: double, in respecToX = true: bool) : Matrix
+ matrixToImage() : void
+ getWidth() : inline int
+ getHeight() : inline int
+ getLayers() : inline int
+ getData() : inline const std::vector<double>
+ setWidth(in new_width: int) : void
+ setHeight(in new_height: int) : void
+ setData (in i: int, in new_data: double) : void
+ setData(in x: int, in y: int, in new_data: double) : void
+ setData(in new_data: std::vector<double>) : void
Image
- grayscale: bool
+ Image(in width: int, in height: int, in layers: int, in grayscale: bool)
+ rgbToGrayscale() : void
+ reformatOrigin(in isTopDown: bool) : void
+ getImageAverage() : double
+ isGrayScale() : inline bool
Class diagram for the Matrix and Image classes

The Matrix class

The Matrix class is built with a similar structure in mind. The contents of a matrix are stored in a 1D array together with a variable for the matrix width and matrix height. Accessing an element of the matrix is similar to accessing an element of a BMP image. The difference between bitmaps and the matrix class is that the origin is intended to be in the upper-left corner for the Matrix class, instead of the bottom-left corner.

The Image class

The Image class inherits all members of the Matrix class. In the C++ program, the .bmp images that are read are converted to Image class objects. This includes reformatting their origin to be top left. This is required for some functions that loop through the image pixels.

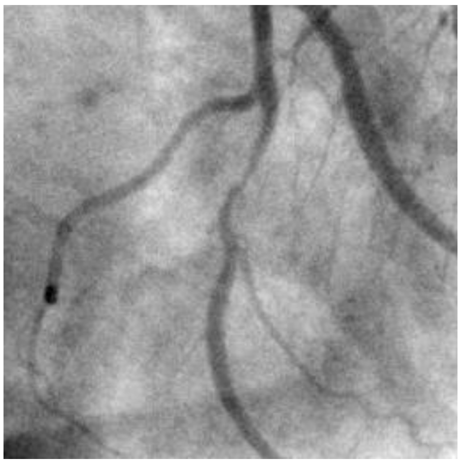
2. Canny edge detection method

The Canny edge detection method consists of performing the following steps:

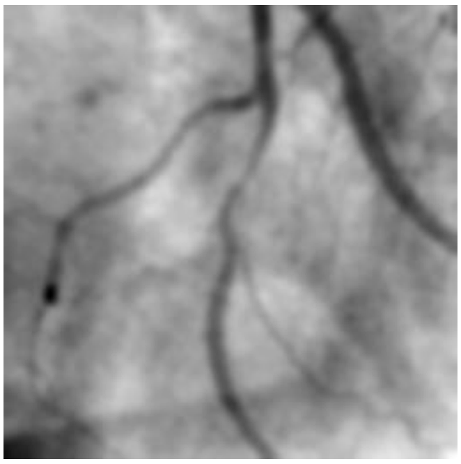
1. Blur the grayscale image (Gaussian blur)
2. Compute the derivatives
    a. Compute derivative in X direction (sometimes called 'fX' or 'deriv_x')
    b. Compute derivative in Y direction (sometimes called 'fY' or 'deriv_y')
3. Calculate gradient magnitude using those derivatives
4. 'Non-maximum' suppression
5. Hysteresis thresholding

To visualise this process, the program writes images for the result of each step.

Read the image (and convert to grayscale)
Consider the following image:

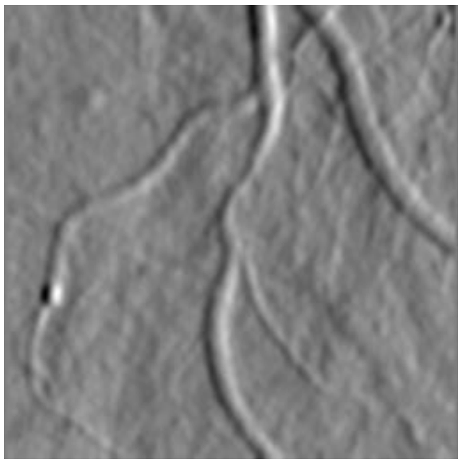


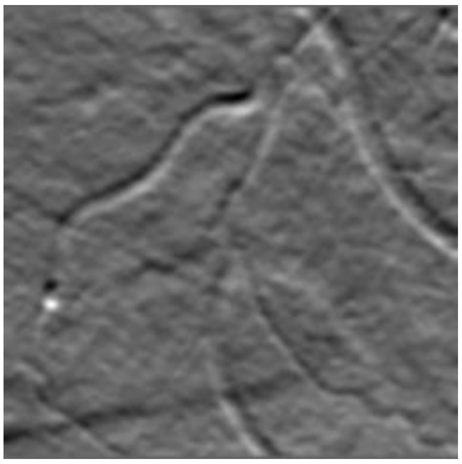
1. Blur the image (Gaussian blur)
The purpose of blurring the image is to reduce noise in the image which can result in false edges.
A sigma of 2 for the Gaussian function was chosen:



2. Compute the derivatives
The derivatives show the areas where the image pixels sharply increase or decrease in value, which means there is a high contrast and probably an edge. Though step 1 and 2 are listed as separate steps here (to help visualise the process), in reality these are done at the same time. Instead of blurring the image with Gaussian blur (using a kernel matrix that was constructed with a Gaussian function), the image is blurred with the derivative of a Gaussian.

The result looks as follows:

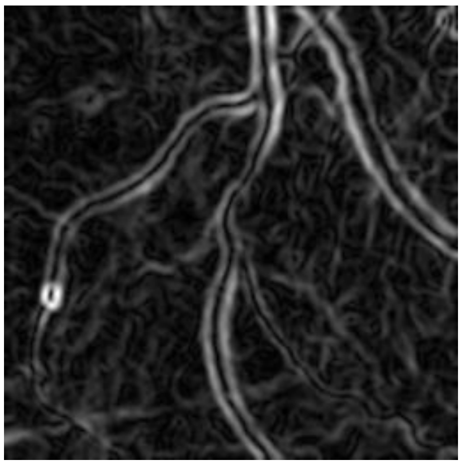

2a Derivative of X


2b Derivative of Y

3. Calculate gradient magnitude
The two derivatives are used to calculate the gradient magnitude, using the following formula:

$$Magnitude = \sqrt{{F_x}^2 + {F_y}^2}$$

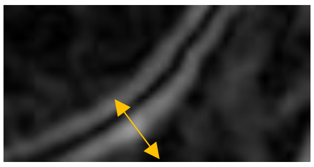
Note: F_x and F_y here are the two derivative images (which are in fact just matrices). The resulting gradient magnitude image looks as follows:



The white areas correspond to places where edges are found. The lighter the area (grayscale value 255 corresponds with the colour white), the clearer the edge. Though we have found the areas where edges are, we still do not know their location *exactly*.

Besides, there is still noise in the image.

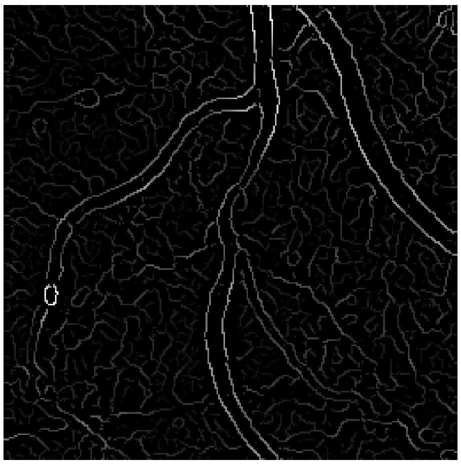
4. 'Non-maximum' suppression
To pinpoint the location of an edge exactly, we simply have to look at the largest (=lightest) value perpendicular to the direction of an edge:



To find this direction, the gradient magnitude *direction* needs to be calculated for every pixel of the image. The direction is calculated by using the following formula:

$$direction = \theta = \arctan\left(\frac{F_y}{F_x}\right)$$

After suppressing the non-maximum values, the image looks as follows:

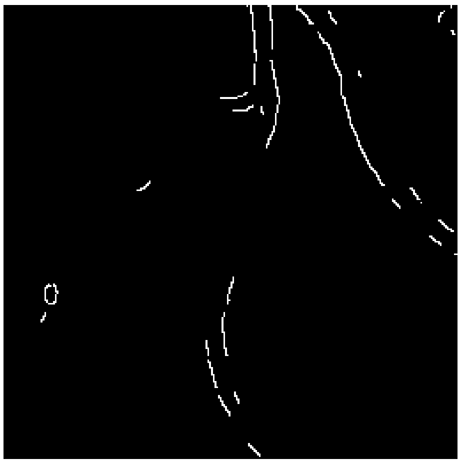


As you can see, the brightness (in grayscale max 255) of a pixel determines how strong the edge is. The noise (but also some parts of the true edges) has lower pixel values. The only step left is to filter this noise.

5. Hysteresis thresholding
Hysteresis thresholding is done by determining two thresholds: a **high** threshold and a **low** threshold. The following logic is then applied:

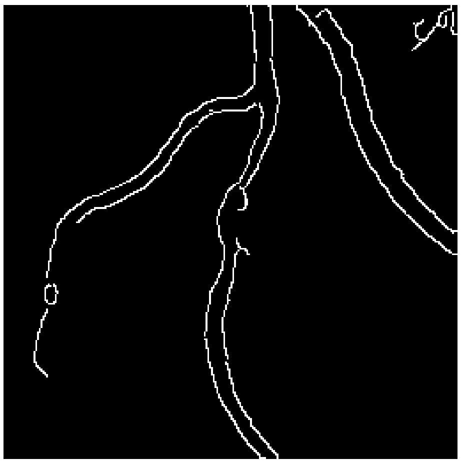
- If the gradient at a pixel is…
  - …above **high**, mark it as an edge pixel
  - …below **low**, discard it
  - …between **low** and **high**, consider its neighbours:
    - If the neighbour is an edge pixel, mark it as an edge pixel
    - If the neighbour is connected to an edge pixel via other pixels that are between **low** and **high**, mark it as an edge pixel
    - If it is not connected to a true edge in any way, discard it

One way of determining the right threshold is to first set only the high threshold (the low threshold should then be equal to the high threshold) and try values until all noise is gone. The threshold chosen for this was 10:



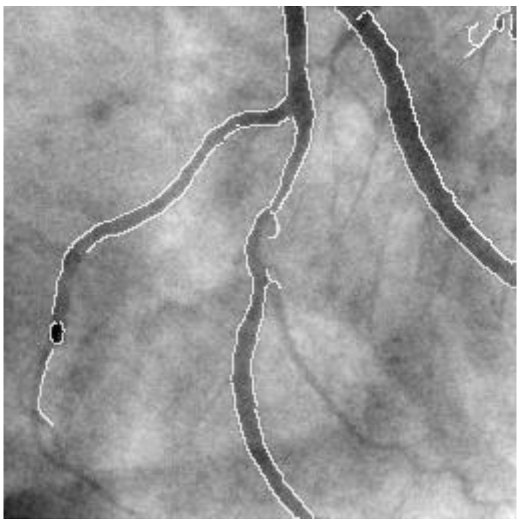
As you can see, only the edge pixels (and some noise in the upper right corner) are left. Large parts of the edges are gone though, but those will reappear with the right low threshold.

The value for the low-threshold was determined to be 0.5, which gave the following edge map:



A better result might be obtainable by further tweaking the sigma, high threshold and low threshold values, but this result was deemed acceptable for the sake of explaining the method.

In the following image, the edge map is overlayed on the original image:

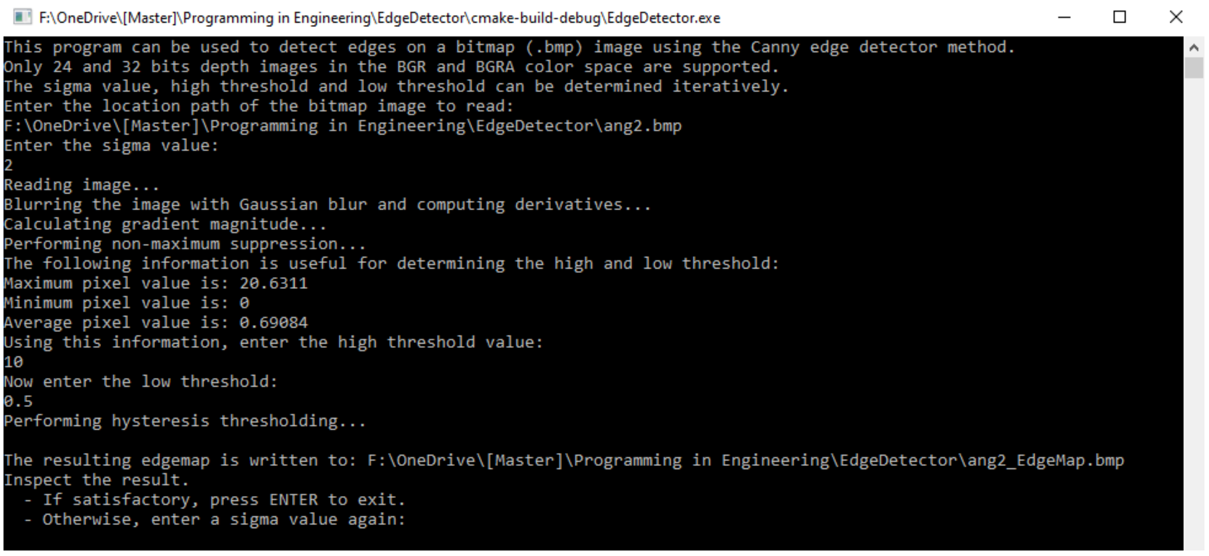


…which shows the locations of the found edges are very accurate.

3. The program

The program consists of three parts: the main file, the Matrix class and the Image class. The BMP struct code is not considered here, because (1) it does not change how the Canny edge detector works and (2) it is not code I wrote myself. Support for any other image file type can be added without a lot of effort, because the program uses the 'neutral' Image class for the edge detection.

Console interface


F:\OneDrive\[Master]\Programming in Engineering\EdgeDetector\cmake-build-debug\EdgeDetector.exe

```
This program can be used to detect edges on a bitmap (.bmp) image using the Canny edge detector method.
Only 24 and 32 bits depth images in the BGR and BGRA color space are supported.
The sigma value, high threshold and low threshold can be determined iteratively.
Enter the location path of the bitmap image to read:
F:\OneDrive\[Master]\Programming in Engineering\EdgeDetector\ang2.bmp
Enter the sigma value:
2
Reading image...
Blurring the image with Gaussian blur and computing derivatives...
Calculating gradient magnitude...
Performing non-maximum suppression...
The following information is useful for determining the high and low threshold:
Maximum pixel value is: 20.6311
Minimum pixel value is: 0
Average pixel value is: 0.69084
Using this information, enter the high threshold value:
10
Now enter the low threshold:
0.5
Performing hysteresis thresholding...

The resulting edgemap is written to: F:\OneDrive\[Master]\Programming in Engineering\EdgeDetector\ang2_EdgeMap.bmp
Inspect the result.
  - If satisfactory, press ENTER to exit.
  - Otherwise, enter a sigma value again:
```

As seen above, the user interface in the console starts with asking the user to input the location of the bitmap image to read. The program first asks the user to input the *filename*/pathname of the image and the *sigma* for the Gaussian blur. The image is then opened and blurred with derivatives of Gaussians, the gradient magnitude is calculated and non-maximum suppression is performed.

After that, the maximum, minimum and average pixel value in the image is found/calculated. These give an indication for what range the threshold values should be in. The user is asked to input *high threshold* and *low threshold*, respectively. The resulting edge map is written to an image (alongside images that show the results of intermediate steps). If the result is adequate, the user can press 'Enter' to exit the program. If the user wishes to use different values, the user can enter a new sigma (and then press 'Enter') to start the whole process over. All previously created images will be overwritten in the new process.

Blurring the image (convolution)

The first step of the edge detection method is to blur the image with a Gaussian blur. This is done by *convolving* the image with a matrix (called the *kernel*) that has a Gaussian distribution. Convolution is an important process in image processing. The Wikipedia definition[1] is as follows:

*Convolution is the process of adding each element of the image to its local neighbours, weighted by the kernel. This is related to a form of mathematical convolution. The matrix operation being performed—convolution—is not traditional matrix multiplication, despite being similarly denoted by *.*

[1] https://en.wikipedia.org/wiki/Kernel_(image_processing)#Convolution

Constructing the kernel

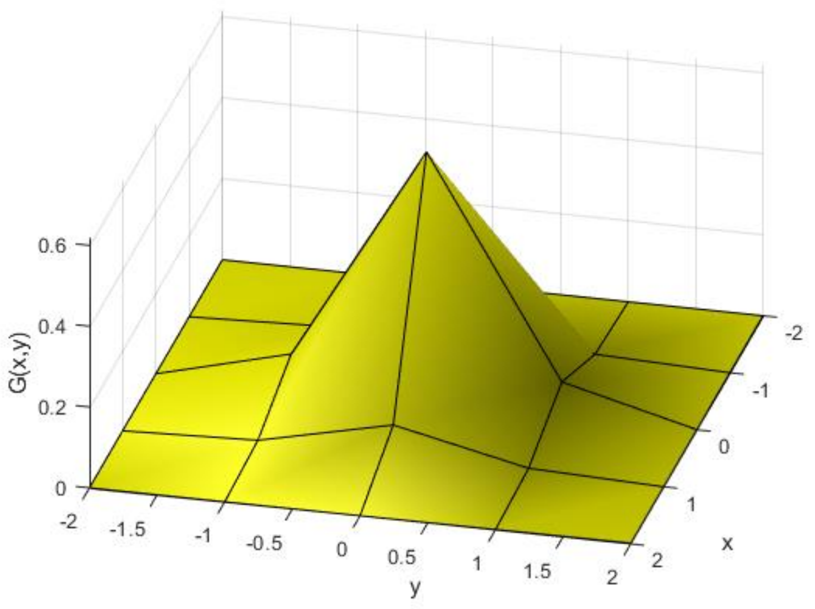
First, we consider the kernel (matrix). One of the functions of the Matrix class is `constructGaussianKernel(int w, int h, double sigma)`. This creates a Matrix with width `w`, height `h` and a Gaussian distribution with the given `sigma` (σ). The function can be used to create a 1D Matrix (with either height or width being equal to 1) or a 2D Matrix. Because of the nature of convolution, the width and height can only be odd numbers, so there is always an element of the matrix that lies in the centre. For now we assume we create a 2D Matrix. The following formula is used for that purpose:

$$G(x, y) = \frac{1}{2\pi\sigma^2} * e^{\frac{x^2+y^2}{2\sigma^2}}$$

The x and y values are picked so that $x = 0$ and $y = 0$ lie in the centre of the matrix. For a size of 5×5 with $\sigma = 0.5$, we would get the following matrix:

	-2	-1	0	1	2	$\rightarrow x$
-2	6.96e-8	2.80e-5	0.0002	2.80e-5	6.96e-8	
-1	2.80e-5	0.0113	0.0837	0.0113	2.80e-5	
0	0.0002	0.0837	0.6187	0.0837	0.0002	
1	2.80e-5	0.0113	0.0837	0.0113	2.80e-5	
2	6.96e-8	2.80e-5	0.0002	2.80e-5	6.96e-8	
$\downarrow$ y						

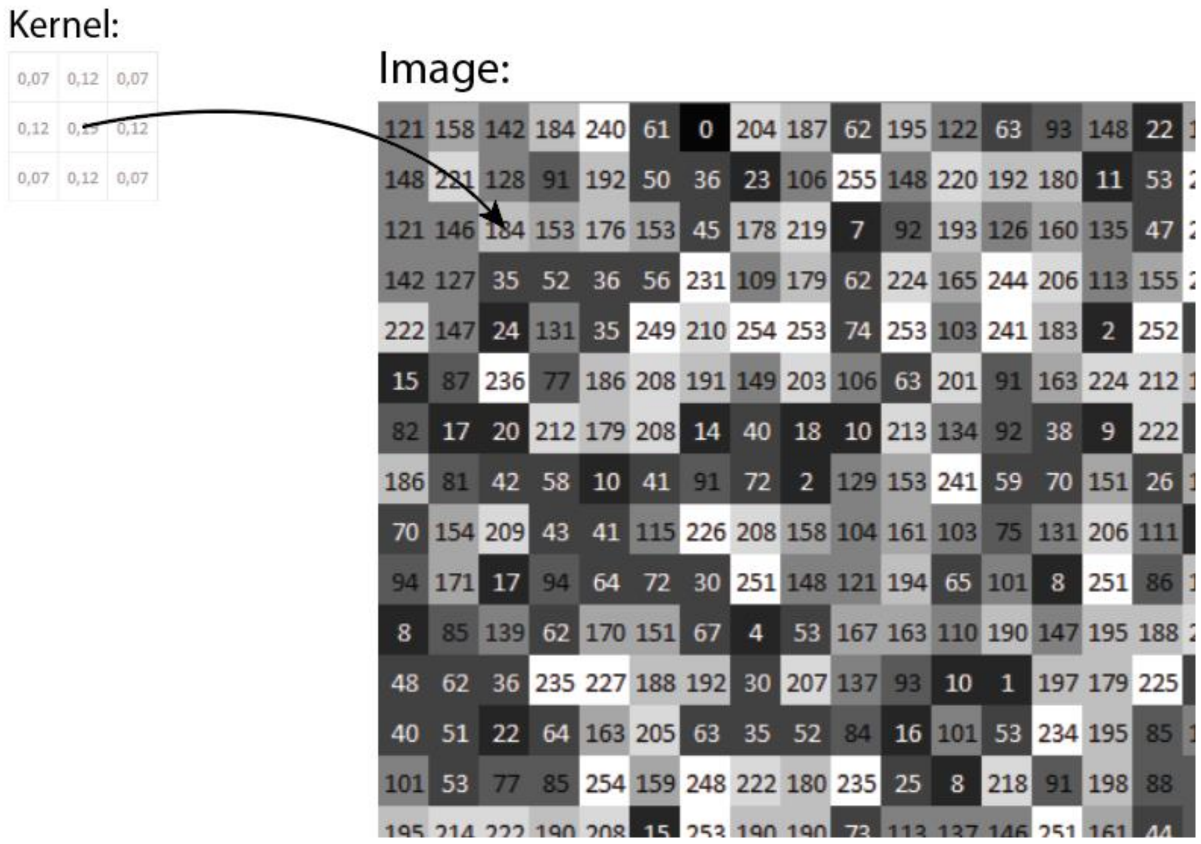
The best way to visualize this, is as a surface plot:



A critical reader might notice that the y-axis in the plot is mirrored compared to the y-axis in the matrix. For a Gaussian function this does not matter, since the distribution is symmetrical. For the derivative of a Gaussian function, this behaviour is accounted for in the formula.

Convolution on the image with the kernel

The next step would be to convolve the image with the kernel. Imagine a grayscale image as a matrix of grayscale values (0 to 255). For each pixel of the image, place the centre point of the kernel on that pixel.



Kernel:

0,07	0,12	0,07
0,12	0,15	0,12
0,07	0,12	0,07

Image:

121	158	142	184	240	61	0	204	187	62	195	122	63	93	148	22
148	221	128	91	192	50	36	23	106	255	148	220	192	180	11	53
121	146	184	153	176	153	45	178	219	7	92	193	126	160	135	47
142	127	35	52	36	56	231	109	179	62	224	165	244	206	113	155
222	147	24	131	35	249	210	254	253	74	253	103	241	183	2	252
15	87	236	77	186	208	191	149	203	106	63	201	91	163	224	212
82	17	20	212	179	208	14	40	18	10	213	134	92	38	9	222
186	81	42	58	10	41	91	72	2	129	153	241	59	70	151	26
70	154	209	43	41	115	226	208	158	104	161	103	75	131	206	111
94	171	17	94	64	72	30	251	148	121	194	65	101	8	251	86
8	85	139	62	170	151	67	4	53	167	163	110	190	147	195	188
48	62	36	235	227	188	192	30	207	137	93	10	1	197	179	225
40	51	22	64	163	205	63	35	52	84	16	101	53	234	195	85
101	53	77	85	254	159	248	222	180	235	25	8	218	91	198	88
195	214	222	190	208	15	253	190	190	73	113	137	146	251	161	44

In the image above, the centre point of the kernel (value 0.15) is placed on the pixel with value 184. Afterwards, each kernel value is multiplied with the overlapping image pixel value. Resulting values are added to each other, this will be the new value for that pixel. Starting from the top-left of the kernel and moving from left to right, top to bottom:

$$(0.07 \times 221) + (0.12 \times 128) + (0.07 \times 91) +$$

$$(0.12 \times 146) + (0.15 \times 184) + (0.12 \times 153) +$$

$$(0.07 \times 127) + (0.12 \times 35) + (0.07 \times 52) =$$

$$= 117.41$$

The new value for that pixel becomes 117.41. This process is repeated for every pixel of the image (which can take considerable time on large images/with large kernels). Because each new pixel value is a weighted sum of their neighbours, the image will look blurred.

One problem occurs around the borders of the image, where some parts of the kernel will have no corresponding pixel value. The edge handling solution[2] used in the program is to *mirror* the pixels inside the image to the area outside the image.

The convolution is handled by the `convolve(Image& image, const Matrix& kernel)` function in main.cpp:221.

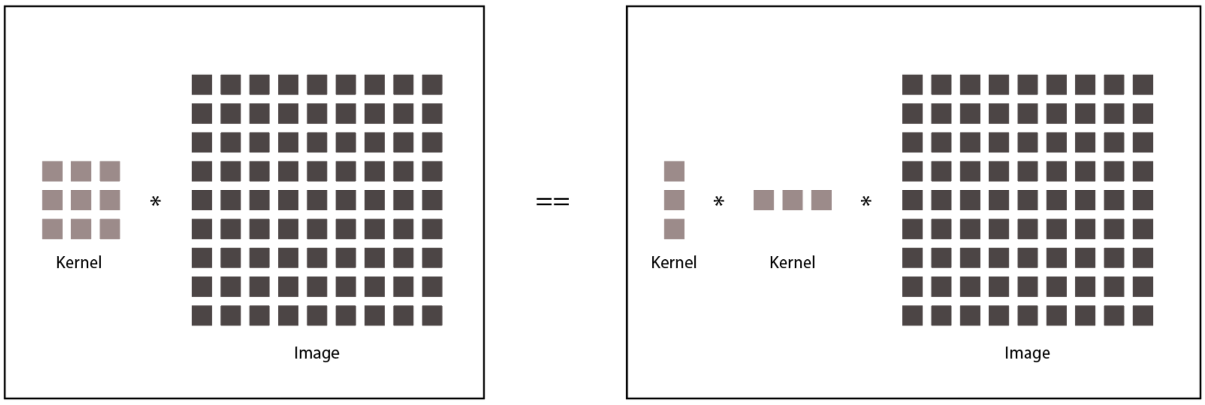
Constructing the Gaussian kernel and convolving the image happens in the main.cpp:319 function `blurGaussian(Image& image, double sigma, int kernel_type)`, where `image` is the input image, `sigma` is the sigma for the Gaussian function and `kernel_type` determines whether the derivative is used and what derivative is used.

Optimizations

An interesting property of convolution is its separability. This means that convolution with a 2D Gaussian kernel provides the same result as convolving with two 1D Gaussian kernels (horizontal and vertical):

$$G(x,y) * image(x,y) = G(x) * G(y) * image(x,y)$$

Or visualized:



This reduces the amount of operations needed, which is why the code in the program only uses 1D kernels.

Because the edge detection method requires the derivative of the image (with respect to x and y), instead of blurring with a Gaussian kernel, the derivative Gaussian function is used for the kernel. This gives the same result as blurring the image with a normal Gaussian kernel and *then* taking the derivative, but is easier since the derivative of a Gaussian function is known.

Gradient magnitude calculation

To calculate the gradient magnitude, we need two matrices: one blurred image with the derivative with respect to x and one blurred image with the derivative with respect to y, we call those F_x and F_y. The formula for the gradient magnitude was:

$$Magnitude = \sqrt{{F_x}^2 + {F_y}^2}$$

[2] https://en.wikipedia.org/wiki/Kernel_(image_processing)#Edge_Handling

This means that every pixel in the gradient magnitude image is the square root of the sum of the corresponding pixel in F_x and F_y squared. This is done in the main.cpp:362 function called `gradientMagnitude(const Image& im_derivX, const Image& im_derivY)`.

Non-maximum suppression

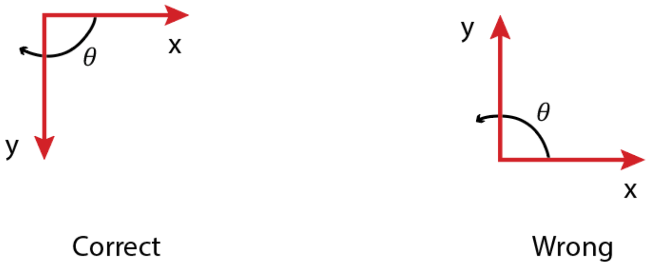
In the main.cpp:407 function `nonMaximumSuppression(const Image& gradMag, const Image& im_derivX, const Image& im_derivY)`, the non-maximum values around the edges are suppressed. For each pixel in the gradient magnitude image (`gradMag`), the gradient direction is calculated using F_x (`im_derivX`) and F_y (`im_derivY`):

$$direction = \theta = \arctan\left(\frac{F_y}{F_x}\right)$$

This is converted into degrees and then rounded to either 0, 45, 90, 135, 180, 225, 270 or 315, i.e.; one of the neighbouring 8 pixels:

225 °	270 °	315 °
180 °	pixel	0 °
135 °	90 °	45 °

Remember: the image is orientation is top-down (the origin is in the top left of the image, y increases as you move down), which means rotations are clockwise:



Then, both directly neighbouring pixels alongside that direction are checked: if they are larger, the current pixel will be suppressed (value becomes 0 and therefore black).

Hysteresis thresholding

The hysteresis thresholding is done in the main.cpp:549 function `hysteresisThresholding(Image& gradMag, double high_threshold, double low_threshold)`.

The function loops through the pixels of the gradient magnitude image (`gradMag`), from left to right, top to bottom. If the pixel value is above the `high_threshold`, it will be changed to 255 (white) and the eight neighbouring pixels are considered:

1	2	3
4	pixel	5
6	7	8

If one of these eight neighbours is between the `low_threshold` and `high_threshold`, its value will also be changed to 255. If pixel 1, 2, 3 or 4 are changed to 255, the loop will go back to that pixel first before continuing. This allows the algorithm to trace weak edges backwards as well.

In a second loop all remaining pixels that are between the `low_threshold` and `high_threshold`, but are not connected to true edge pixels, are cleared.

The result of this is the edge map, with edge pixels being white (value 255) and all other pixels being black (value 0).