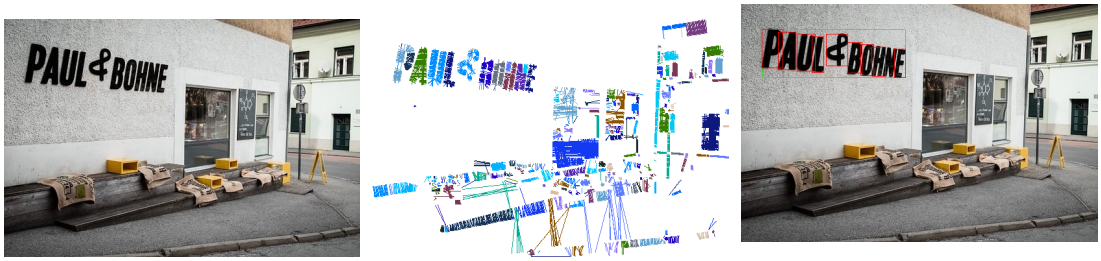


(a) Input image (b) Stroke Width Transform (SWT) (c) Detected letters (red) and grouped text (green)

Figure 1: The Stroke Width Transform (SWT) algorithm applied to a colored input image. On the left the original image, in the middle the highlighted components of the SWT algorithm and on the right the detected text marked by bounding boxes.

CV/task1 — Stroke Width Transform

1 Overview

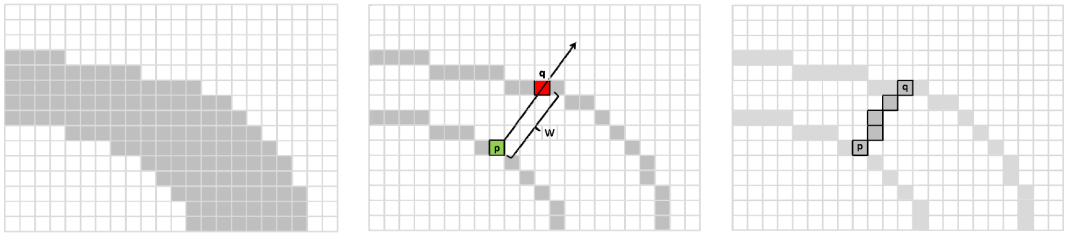
This task aims to become familiar with basic functionalities and a part of the extended range of functions of the OpenCV library. For this purpose, the Stroke Width Transform (SWT) [2] should be implemented. This algorithm enables to localize text in images taken in a natural environment. Fig. 1 shows an example of the SWT.

Automatically detecting written text in natural environments is a challenging problem in computer vision. With SWT, Epshtein *et al.* [2] propose an algorithm that can reliably accomplish this task. At its heart lies a transformation that assigns to each pixel in the original image the width of the stroke to which it most likely belongs. In this case, a stroke is defined as a contiguous image region of constant width (see Figure 3). In the following, the method's steps are described briefly to provide a first overview of the individual parts (see Figure 2).

The basic idea behind the algorithm is that written text usually consists of strokes that have a similar width. This assumption is valid for most scenarios regardless of the used language and font, making the algorithm versatile and applicable in different situations.

In order to find the strokes, we first extract the edges of the input image using the Canny edge detection algorithm [1]. This method returns an image where for each pixel, it is

Figure 2: Overview of the individual steps of the SWT algorithm.



(a) Typical stroke. (b) Boundary pixels p and q. (c) Ray between p and q.

Figure 3: Stroke width definition according to the Stroke Width Transform (SWT) [2] algorithm.

indicated whether it lies on an edge or not. We can now use this information to emit a ray $\mathbf{r}$ starting from the edge pixel $\mathbf{p}$ in the direction $\mathbf{d_p}$:

$$\mathbf{r} = \mathbf{p} + n\mathbf{d_p}, \quad n > 0, \tag{1}$$

where the direction $\mathbf{d_p}$ is defined as the normalized gradient in $\mathbf{p}$. We have found the opposite side of the stroke if we get to a pixel $\mathbf{q}$, which is also located on an edge and the gradient in $\mathbf{q}$ points approximately in the opposite direction of $\mathbf{d_p}$. All pixels on the ray from $\mathbf{p}$ to $\mathbf{q}$ are then assigned the same stroke width value $\|\mathbf{p} - \mathbf{q}\|$.

After some post-processing, the attained results can now be utilized to find letter candidates by comparing various properties of the found connected components. Finally, these results can be grouped into words marked by a bounding box at the end.

2 Tasks

The provided framework contains a `main.cpp` and `helper.cpp` (both should not be edited, they are used for automatic execution) and an `algorithms.cpp` where the tasks should be implemented in the respective functions (`compute_grayscale`, `compute_gradient`, `compute_directions`, `swt_compute_stroke_width`, `swt_postprocessing`, `get_connected_components`, `compute_bounding_boxes`, `discard_non_text` and for the bonus task `non_maxima_suppression`, `hysteresis` and `canny_own`).

The required parameters for each test case (e.g. `graz`) are stored in separate JSON files (e.g. `graz.json`) located in the folder `tests` in your repository. The file `graz.json` has to be passed to the application to execute the test case `graz`. Then the input data is fetched

automatically from the JSON files and can be used immediately for the calculations in the respective functions. You do not need to modify the parameters. They are also passed to the respective subroutines for you. The content of this task is limited solely to the SWT-related functions in `algorithms.cpp`. You do not have to modify any other provided file. The framework is built such that only the previously mentioned functions have to be implemented to achieve the desired output.

The example is divided into several sub-tasks. The process consists of the following steps:

- Compute Grayscale
- Compute Gradient
- Compute Directions
- Stroke Width Transform
  - Stroke Width Estimate
  - Post Processing
- Compute Connected Components
- Compute Bounding Boxes
- Discard Non-Text

For the Bonus task:

- Canny Edge Detector
  - Non Maxima Suppression
  - Hysteresis
  - Canny Edges

This task has to be implemented using OpenCV[1] 4.2.0. Use the functions provided by OpenCV and pay attention to the different parameters and image types. OpenCV uses the BGR color format instead of the RGB color format for historical reasons, i.e., the reversed order of the channels. This has to be considered in the implementation.

[1] `http://opencv.org/`

2.1 Compute Grayscale (1 Point)

In the function `compute_grayscale(...)`, you have to generate a grayscale image from a 3-channel RGB input image. For this purpose, the information of the three color channels must be extracted for each pixel of the input image. We recommend the access to the pixel at location `(row, col)` via `input_image.at<cv::Vec3b>(row, col)`. This command returns a vector of 3 byte values containing the channel intensities.

There are two things to keep in mind when fetching color information from `cv::Mat` objects in OpenCV:

- OpenCV uses a row-major order, which means that the first index always refers to the row that should be accessed and the second index to the column of interest, respectively.
- OpenCV uses the BGR channel order. As a consequence, the first element of an extracted `cv::Vec3b` object is related to blue, the second to green, and the third one to red.

After the color information is obtained, it has to be combined according to

$$I = 0.2989 \cdot R + 0.5870 \cdot G + 0.1140 \cdot B \tag{2}$$

to retrieve the intensity value for the corresponding pixel in the output image. This formula considers that the human eye is more sensitive to green components in the spectrum of light. That is the reason for giving the green color channel the highest weight. After the calculation, the result has to be cast to an 8-bit unsigned integer. It can directly be written to the passed single-channel `cv::Mat` reference `grayscale_image` after the conversion. Figure 4 shows an example for the expected output.

Forbidden Functions:

- `cv::cvtColor(...)`

2.2 Compute Gradient (1 Point)

This task is to be solved in the function `compute_gradient(...)`. The gradient calculation is done by applying the Scharr operator. This should be done by utilizing the OpenCV function `cv::Scharr` which detects directed changes in the input image with the help of derivatives. A large gradient value in certain image regions indicates strongly pronounced intensity changes caused by edges, color transitions, exposure differences, and so forth. For the derivative in x-direction G_x, the image I is convolved with the kernel S_x. Analogously,

(a) RGB input image.

(b) Grayscale output image.

Figure 4: Example input and corresponding output image.

(a) The gradient in x-direction.

(b) The gradient in y-direction.

(c) The absolute value of the gradient.

Figure 5: Expected output of function `compute_gradient(...)`.

for the derivative in y-direction G_y, the image I is convolved with the kernel S_y. This can be formally written as

$$G_x = S_x * I = \begin{bmatrix} -3 & 0 & 3 \\ -10 & 0 & 10 \\ -3 & 0 & 3 \end{bmatrix} * I \tag{3}$$

and

$$G_y = S_y * I = \begin{bmatrix} -3 & -10 & -3 \\ 0 & 0 & 0 \\ 3 & 10 & 3 \end{bmatrix} * I. \tag{4}$$

The convolution operation is denoted with $*$.

First, you have to determine the respective first order derivative in the directions x and y. When calling the function `cv::Scharr` set the parameter describing the bit depth of the output image to the type `CV_32F`, which correspondes to a 32-bit floating point value. After that, combine the two gradient images according to

$$G = \sqrt{G_x^2 + G_y^2}, \tag{5}$$

where the resulting image G stores the per-pixel l_2-norm of the gradient. The three generated images (x-derivative, y-derivative, and gradient norm) are supposed to be returned via the provided `cv::Mat` references. In Figure 5 you can see an example of the expected output.

Useful Functions:

- `cv::Scharr(...)`
- `cv::pow(...)`
- `cv::add(...)`
- `cv::sqrt(...)`

2.3 Compute Directions (1 Point)

In function `compute_directions(...)`, the previously created derivatives in the x- and y-direction should now be normalized to unit length. To do this, iterate over each pixel of the input image and check whether the norm of the gradient at that location is greater than 0. If this is the case, the derivatives have to be divided by the magnitude of the gradient. However, if the norm is zero, the values of the x and y directions should also be set to zero. Again, store the normalized derivatives in the provided `cv::Mat` references.

Forbidden Functions:

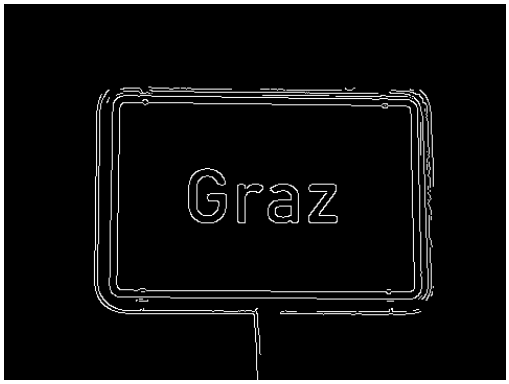


Figure 6: Example of an edge image.

- `cv::normalize(...)`

2.4 Stroke Width Transform (5 Points)

Now it is time to implement the SWT. The overall transformation is divided into two parts. The first part has to be implemented in the function `swt_compute_stroke_width(...)` and the second part in the function `swt_postprocessing(...)`. In addition to the normalized directions, an edge image created with the Canny edge detector [1] is required, as described in the introduction. You do not need to perform this step as the edge image is already provided as function parameter in `swt_compute_stroke_width(...)`, but you are encouraged to implement it by yourself in the bonus task.

2.4.1 Stroke Width Estimate (3 of 5 Points)

As described above, SWT assigns to each pixel in the input image the width of the stroke to which the pixel most likely belongs. Since strokes are bounded on two sides by an edge, the edge information is an integral part of the algorithm to find stroke candidates. The `cv::Mat` reference `edges` provides this information. Each pixel belonging to an edge is assigned a non-zero value in this image, whereas all other pixels are set to zero (see Fig. 6). Since the stroke width must be assigned to each pixel, a so-called ray has to be emitted from each edge pixel (that is, the edge image is not equal to zero at this pixel) in or against the direction of the gradient belonging to this point (see gradient direction explanation in next paragraph). The direction is provided via the previously calculated direction images `direction_x` and `direction_y`.

Here, it is noted that the gradient of a function assigns the direction of the steepest slope to each point. Considering images as 2D functions, the gradient always points from dark

spots (smaller values) to lighter ones (higher values). In the case of bright font on a dark background, this is the desired direction, since the goal is to get from one side of the stroke to the edge on the opposite side of it (see Fig. 3). However, in the opposing case, the direction must be reversed by multiplying it with (-1). Therefore, the Boolean parameter `black_on_white` is provided to distinguish these two cases (see Fig. 7).

Your task is now to iterate over all edge pixels. For every pixel $\mathbf{p}$ of these, move in the corresponding direction $\mathbf{d_p}$ until either the valid image area is left or another edge pixel $\mathbf{q}$ is reached. To calculate the current row position of the ray the following formular should be used:

$$current\_row = \left\lfloor \mathbf{p}_{row} + direction * \mathbf{d}_{\mathbf{p}_{row}} * step\_size \right\rfloor, \tag{6}$$

where $direction$ is -1 when the parameter `black_on_white` is `true` and 1 when `black_on_white` is `false`. The variable $step\_size$ should increase by one for every step taken. The current column position of the ray is calculated similarly (use column specific values). The ray is valid if the direction belonging to the other pixel $\mathbf{d_q}$ is approximately the opposite of the current direction. The angle between $-\mathbf{d_p}$ and $\mathbf{d_q}$ must not be greater than $\pm\frac{\pi}{6}$ to form a valid ray. Since the direction vectors are normalized, this can be checked by applying the dot product:

$$-\mathbf{d}_\mathbf{p}^T \mathbf{d_q} \geq \cos\frac{\pi}{6}. \tag{7}$$

If this condition is not met, or if the edge of the image is reached earlier, the ray can be discarded. Each valid ray must be stored as `std::vector<cv::Point2i>`. This is a list of the coordinates of those pixels that belong to the ray. All these vectors should be appended to the vector reference `rays`.

After a whole ray has been captured and it is valid, the stroke width is given by the length of the ray which is equivalent to the distance between the two edge pixels:

$$w = \|\mathbf{q} - \mathbf{p}\|_2. \tag{8}$$

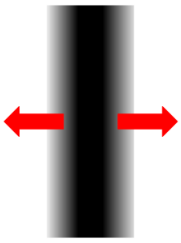
The newly calculated stroke width w is assigned to each pixel on the ray that previously had a larger value stored in the SWT image. This operation is also the reason to initialize the SWT image with `FLT_MAX` (already done for you in the code). For any pixel $\tilde{\mathbf{p}}$ on the ray between $\mathbf{p}$ and $\mathbf{q}$ the following applies:

$$\mathrm{SWT}(\tilde{\mathbf{p}}) = \min\left\{w, \mathrm{SWT}(\tilde{\mathbf{p}})\right\}. \tag{9}$$

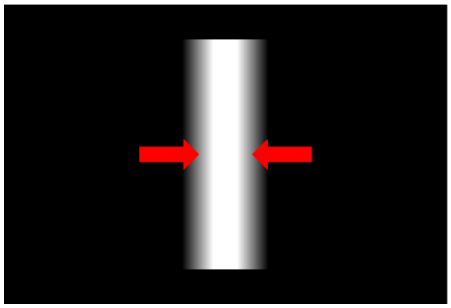
After all edge pixels have been processed, this part of the SWT is complete.

Useful Functions:

- `cv::Point2i(...)`
- `std::floor(...)`



(a) Gradient direction of a dark-colored letter on bright background.

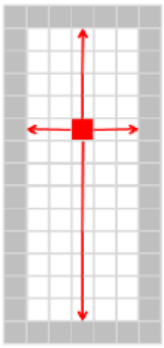


(b) Gradient direction of a light-colored letter on dark background.

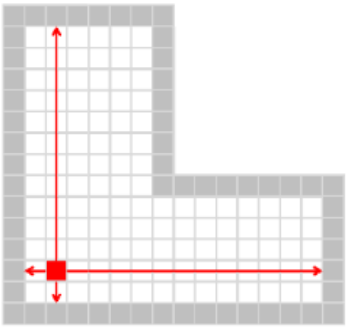
Figure 7: Example of the gradient directions with different letter colors and backgrounds.

- `std::sqrt(...)`
- `std::pow(...)`
- `std::min(...)`

2.4.2 Post Processing (2 of 5 Points)



(a) Red pixel is filled with minimum between the lengths of vertical and horizontal rays passing through it.



(b) Red pixel stores the minimum between the two rays lengths. This example should also show the necessity for the postprocessing step.

Figure 8: Filling pixels with SWT values [2].

Since there are situations (see Figure 8) where there are still erroneous values after an initial estimation of the stroke width, the previously obtained result must be post-processed. This task has to be implemented in function `swt_postprocessing(...)`.

The goal is to iterate over all previously found rays and calculate the median m of the stroke width values for each ray. Then the stroke width of all pixels that belong to the ray and have an SWT value greater than the obtained median is clamped to it.

Assume that the ray consists of n pixels with corresponding SWT values s_i. If these stroke widths are sorted, then

$$m = \begin{cases} s_{k+1} & \text{if } n = 2k+1 \text{ odd}, \\ \frac{1}{2}(s_k + s_{k+1}) & \text{if } n = 2k \text{ even} \end{cases} \tag{10}$$

is the corresponding median. After that, all pixels whose SWT value remained at `FLT_MAX` must be set to 0. Save the resulting image in the designated parameter `swt_final_image`. The SWT is now completed.

Useful Functions:

- `std::sort(...)`
- `std::min(...)`
- `cv::Mat().setTo(...)`

2.5 Compute Connected Components (4 Points)

Now that the SWT is complete, pixels with similar widths should be clustered into connected components. This has to be done in the function `get_connected_components(...)`. A connected component (see Figure 10) is characterized by all associated pixels having the same label number. Pixels that do not belong to any component and pixels with a stroke width of zero receive the label 0.

Your task is to find all connected components in the SWT image and assign them an ascending label number. That means all pixels belonging to the first component get a 1, all pixels belonging to the second component get a 2, and so forth. The final labeled image should be returned at the end of the function via the `cv::Mat` reference `labels`.

If the stroke width of pixel $\mathbf{p}$ is denoted by $s_\mathbf{p}$, and the stroke width of a neighboring pixel $\mathbf{q}$ is denoted by $s_\mathbf{q}$, then

$$\frac{s_\mathbf{p}}{s_\mathbf{q}} \in (\frac{1}{t}, t) \tag{11}$$

has to be true to cluster the pixels into the same component. The stroke width ratio parameter t is passed to the function as `stroke_width_ratio_threshold`.

(r-1, c-1)	(r-1, c)	(r-1, c+1)
(r, c-1)	(r, c)	(r, c+1)
(r+1, c-1)	(r+1, c)	(r+1, c+1)

(a) With a parameter `neighbor_offset` of 1.

(r-2, c-2)	(r-2, c-1)	(r-2, c)	(r-2, c+1)	(r-2, c+2)
(r-1, c-2)	(r-1, c-1)	(r-1, c)	(r-1, c+1)	(r-1, c+2)
(r, c-2)	(r, c-1)	(r, c)	(r, c+1)	(r, c+2)
(r+1, c-2)	(r+1, c-1)	(r+1, c)	(r+1, c+1)	(r+1, c+2)
(r+2, c-2)	(r+2, c-1)	(r+2, c)	(r+2, c+1)	(r+2, c+2)

(b) With a parameter `neighbor_offset` of 2.

Figure 9: Examples for pixel neighborhoods, where r represents the row and c the column. Be aware that the zero point (0,0) starts in the left upper corner.

Pixels are considered neighboring if they are in close proximity to each other. This proximity is defined by the `neighbor_offset` parameter. An example of a pixel neighborhood can be seen in Figure 9.

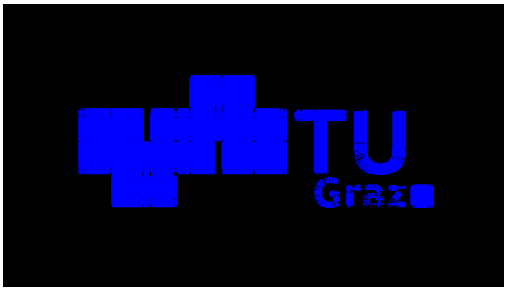
Now, to perform the labeling, iterate over all pixels of the SWT image. All pixels which have a stroke width of zero or already received a label can be skipped. A new label should be created by incrementing a label counter for each pixel that has not yet been processed.

Since the neighborhood of each unprocessed pixel should be considered, it makes sense to create a list of neighboring pixels that have to be processed. A struct named `PosStrokeWidth` is provided for this task, which contains the pixel position and the corresponding stroke width.

For each point in this neighborhood list, check whether it is a member of the current connected component. In addition to the stroke width ratio, it must also be verified whether the pixel coordinates lie in the valid image area. A neighbor pixel should be withdrawn if

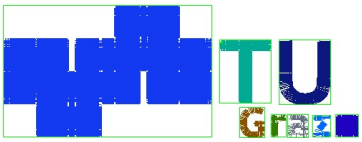
- it is not in the valid image area
- it has an associated stroke width of zero
- it has already received another label
- it does not fulfill the stroke width ratio constraint

(a) Input image.



(b) Detected rays.

(c) Connected components.



(d) Connected components with bounding boxes.

Figure 10: Connected components with their respective bounding boxes. Different colors denote different label numbers. Every component has one label.

However, if these reasons for exclusion do not occur, the pixel receives the current label. Furthermore, the neighborhood is extended to those pixels that fulfill the neighborhood criterion for the current one. A component is expanded until the neighborhood list is completely processed. Store the pixel coordinates for each component in a `std::vector<cv::Point2i>` and append them to the parameter `components`.

2.6 Compute Bounding Boxes (1 Point)

In this task, place bounding boxes around the found connected components. For each component, the extreme coordinates (maximum row, maximum column, minimum row, minimum column) should be found to create a `cv::Rect2i` object defined by the upper left point and a height and width. Afterward, the found bounding box shall be appended to the `std::vector<cv::Rect2i>& bounding_boxes`. Figure 10 shows an image of connected components and their associated bounding boxes.

Useful Functions:

- `std::min(...)`
- `std::max(...)`

- `cv::Rect2i(...)`

Forbidden Functions:

- `cv::boundingRect(...)`

2.7 Discard Non-Text (3 Points)

Now that all connected components have been found, the next step is to find letter candidates. Implement this task in function `discard_non_text(...)`. The statistical properties of the connected components and their bounding boxes must be checked to find promising text locations. Those components that violate one of the properties can be discarded as letter candidates.

The concrete characteristics that have to be verified are:

- `variance_ratio`: The variance σ^2 of a set of data points indicates how much the individual measured values vary on average around the sample mean. One can interpret the connected components found and the SWT values of the associated pixels as such a data set. Therefore, it is possible to calculate the variance of a component by first calculating the mean

$$\mu = \frac{1}{n}\sum_{i=1}^{n} s_i \tag{12}$$

  and then use it to compute

$$\sigma^2 = \frac{1}{n}\sum_{i=1}^{n} (s_i - \mu)^2. \tag{13}$$

  Here, s_i denotes the stroke width of the i-th pixel of one particular component. A connected component can be considered a possible letter if the relation

$$\sigma^2 \leq \texttt{variance\_ratio} \cdot m, \tag{14}$$

  where m denotes the median as defined in Eq. 10, holds.

- `aspect_ratio_threshold`: The ratio between height and width of the bounding box of a connected component should also be considered for deciding whether it is a letter or not. This constraint is important because rectangles too lengthy or wide indicate that the connected component is not a letter. The aspect ratio a is defined as

$$a = \frac{w_b}{h_b}, \tag{15}$$

where w_b denotes the width of the bounding box and h_b the height, respectively. The condition

$$\frac{1}{\texttt{aspect\_ratio\_threshold}} \leq a \leq \texttt{aspect\_ratio\_threshold} \tag{16}$$

must hold for a valid character candidate.

- `diameter_ratio_threshold`: The ratio between the bounding box diameter

$$d_b = \sqrt{w_b^2 + h_b^2} \tag{17}$$

and the median stroke width m in the connected component is within a specific interval for letters, which the candidates must also fulfill. The condition

$$\frac{d_b}{m} <= \texttt{diameter\_ratio\_threshold} \tag{18}$$

has to be fulfilled for letter candidates.

- `min_height` and `max_height`: In addition, the absolute height of the bounding boxes in pixels is also relevant. Letter candidates must be in the interval

$$\texttt{min\_height} \leq h_b \leq \texttt{max\_height}. \tag{19}$$

The needed values for the thresholds are passed to the function as parameters. After all valid letter candidates have been found, the corresponding components must be returned in `text_components`, the bounding boxes in `text_bounding_boxes` and the correct labels via `text_labels`. Note that the already assigned label numbers should not be changed. This means that the third connected component will keep label number 3, even though the first and second may have been discarded.

The whole algorithm for detecting text components is now completed. The result for one of the input images can be seen in Figure 1. The last part of marking the individual text components and their corresponding letter groups is done automatically by the framework.

Useful Functions:

- `std::sort(...)`
- `std::pow(...)`
- `std::sqrt(...)`

(a) Grayscale input image.

(b) Gradient magnitude (Sobel filter).

(c) Non-maxima suppression (Sobel grad.).

(d) Detected Edges.

Figure 11: Pipeline of Canny Edge Detector

2.8 Bonus: Canny Edge Detector (3 Points)

As an additional task, you have the possibility to implement the famous Canny edge detection algorithm [1]. The method is an essential part of the SWT and many other computer vision algorithms. An example of this pipeline can be seen in Figure 11. You are not allowed to use the function `cv::Canny(...)` in the bonus task.

2.8.1 Non-Maxima Suppression (1 of 3 Points)

This part of the task has to be implemented in the function `non_maxima_surpression(...)`. Here local maxima should be found. This leads to the fact that in the resulting image only the detected edges remain. For each pixel $\mathbf{p}$, the already calculated gradient from Section 2.2 is used. Calculate the angle

$$\alpha = \angle G = \arctan \frac{G_y}{G_x} \tag{20}$$

and transform all angles smaller than 0° to the interval $[0, 180]$:

$$\beta = \begin{cases} (\alpha + 360) \mod 180 & \text{if } \alpha < 0, \\ \alpha & \text{otherwise.} \end{cases} \tag{21}$$

Table 1: Classification table of the angles and relationship to the neighboring pixels $\mathbf{q}$ and $\mathbf{r}$. The position of $\mathbf{p}$ is (r,c), where r and c denote the corresponding row and column, respectively.

Class	Angle	Neighbor Relation $(\mathbf{p}=(r,c))$
1	$\beta \leq 22.5$ or $\beta > 157.5$	$\mathbf{q}=(r,c-1)$, $\mathbf{r}=(r,c+1)$
2	$22.5 < \beta \leq 67.5$	$\mathbf{q}=(r+1,c+1)$, $\mathbf{r}=(r-1,c-1)$
3	$67.5 < \beta \leq 112.5$	$\mathbf{q}=(r-1,c)$, $\mathbf{r}=(r+1,c)$
4	$112.5 < \beta \leq 157.5$	$\mathbf{q}=(r+1,c-1)$, $\mathbf{r}=(r-1,c+1)$

The calculated angle β is now assigned to one of the four classes (1: horizontal, 2: diagonal-1, 3: vertical, 4: diagonal-2) according to Tab. 1. Depending on the class of the angle β, the focus lies on neighboring pixels in different directions. If either the value of pixel $\mathbf{q}$ or $\mathbf{r}$ is larger than the value of pixel $\mathbf{p}$, it is set to zero, because it is not the largest value in the gradient direction:

$$\hat{G}(\mathbf{p}) = \begin{cases} 0 & \text{if } G(\mathbf{p}) < G(\mathbf{q}) \text{ or } G(\mathbf{p}) < G(\mathbf{r}), \\ G(\mathbf{p}) & \text{otherwise.} \end{cases} \tag{22}$$

The non-maxima suppression result $\hat{G}(\mathbf{p})$ should be written to `non_max_sup`.

Useful Functions:

- `cv::copyTo(...)`
- `std::atan2(...)`

2.8.2 Hysteresis Thresholding (1 of 3 Points)

This part of the task must be implemented in the function `hysteresis(...)`. The already found and refined lines should be compared with the thresholds $\tau_{\min}$ and $\tau_{\max}$. Each pixel $\mathbf{p}$ is classified, whether it belongs to a weak, a strong, or no edge. If the value of pixel $\mathbf{p}$ is greater than the upper threshold $\tau_{\max}$ then it is part of a strong edge. Strong edges are certainly part of the final result. If the value is smaller than the lower threshold $\tau_{\min}$ then this pixel is certainly not part of an edge and therefore not part of the final result. If the value of pixel $\mathbf{p}$ is between the two thresholds, it is part of a weak edge. The value assignment in the hysteresis thresholded image H is as follows:

$$H(\mathbf{p}) = \begin{cases} 255 \text{ (strong edge)} & \text{if } \hat{G}(\mathbf{p}) \geq \tau_{\max}, \\ \text{weak edge} & \text{if } \tau_{\min} < \hat{G}(\mathbf{p}) < \tau_{\max}, \\ 0 \text{ (no edge)} & \text{if } \hat{G}(\mathbf{p}) < \tau_{\min}. \end{cases} \tag{23}$$

After all non-edges and strong edges in the image have been found and marked, weak edges are classified iteratively or recursively depending on their neighborhood. If in the 8-neighborhood of a pixel **p**, which belongs to a weak edge, at least one pixel is part of a strong edge, then the pixel **p** also becomes a strong edge (=255). It is recommended to save all pixels that have already been classified as strong edges and then examine their 8-neighborhood to search for a weak edge in this area. If this is the case, the weak edge becomes a strong edge. Furthermore, it has to be searched in the 8-neighborhood of the freshly found strong edge as well. Using a recursive approach makes sure to not miss any weak edges. At the end of this function, the result H consists only of either 0 (no edge) or 255 (strong edge) pixels.

2.8.3 Canny Edges (1 of 3 Points)

In the `canny_own(...)` function, all previously implemented parts of the algorithm should now be applied to create an edge image. The function gets a grayscale image and the two thresholds τ_{min} and τ_{max} as input. The Sobel operator (a generalized form of the previously used Scharr filter) has to be used to calculate the derivatives in x- and y-direction. Also, the absolute value of the gradient should be calculated. All this is done analogous to the calculation in task 2.2. Then call the `non_maxima_suppression(...)` and `hysteresis(...)` functions with according parameters. The result is the finished edge image which should be returned.

Useful Functions:

- `cv::Sobel(...)`
- `cv::pow(...)`
- `cv::add(...)`
- `cv::sqrt(...)`
- `cv::Mat().copyTo(...)`

3 Framework

The following functionality is already implemented in the program framework provided by the ICG:

- Processing of the passed JSON configuration file.

- Reading of the corresponding input image.
- Iterative execution of the functions in `algorithms.cpp`.
- Writing of the generated output images to the corresponding folder.

4 Submission

The tasks consist of several steps, which build on each other but are evaluated independently. On the one hand, this ensures objective assessment and, on the other, guarantees that points can be scored even if the tasks are not entirely solved.

We explicitly point out that the exercise tasks must be solved independently by each participant. If source code is made available to other participants (deliberately or by neglecting a certain minimum level of data security), the corresponding part of the assignment will be awarded 0 points for all participants, regardless of who originally created the code. Similarly, it is not permitted to use code from the web, books, or any other source. Both automatic and manual checks for plagiarism will be performed.

The submission of the exercise examples and the scheduling of the assignment interviews takes place via a web portal. The submission takes place exclusively via the submission system. Submission by other means (e.g. by email) will not be accepted. The exact submission process is described in the `Readme.md` in your repository.

The tests are executed automatically. Additionally, the test system has a timeout after seven minutes. If the program is not completed within this time, it will be aborted by the test system. Therefore, be sure to check the runtime of the program when submitting it.

Since the delivered programs are tested semi-automatically, the parameters must be passed using appropriate configuration files exactly as specified for the individual examples. In particular, interactive input (e.g. via keyboard) of parameters is not permitted. If the execution of the submitted files with the test data fails due to changes in the configuration, the example will be awarded 0 points.

The provided program framework is directly derived from our reference implementation, by removing only those parts that correspond to the content of the exercise. Please do not modify the provided framework and do not change the call signatures of the functions.

References

[1] John Canny. A computational approach to edge detection. IEEE Transactions on Pattern Analysis and Machine Intelligence, 8(6):679–698, 1986.

[2] Boris Epshtein, Eyal Ofek, and Yonatan Wexler. Detecting text in natural scenes with stroke width transform. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2010.