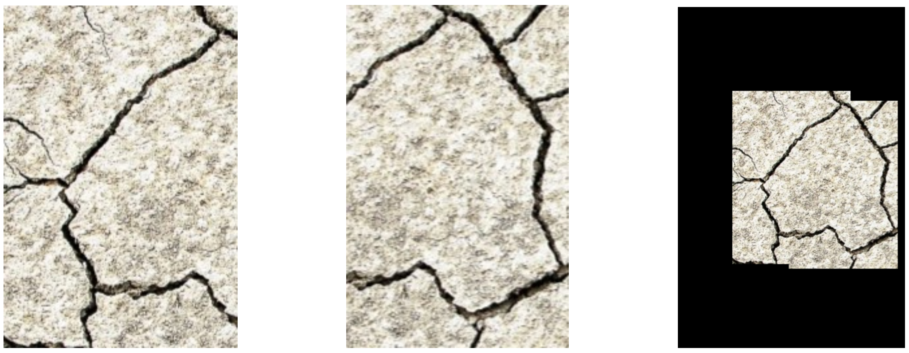


(a) Input image 1. (b) Input image 2. (c) Merged image.

Figure 1: The pipeline applied to a crack input-image-pair. Subfigure (a) and (b) show the two input images, subfigure (c) the merged view within an extended black matching frame.

CV/task1 — Crack Matching

1 Overview

This task serves as an introduction to working with the popular OpenCV library. The goal is to merge two images of different regions of a cracked surface, which requires several steps of image pre-processing. Fig. 1 shows a stripped-down example of the pipeline in fig. 2.

To obtain the desired outcome, we firstly apply different filtering techniques, including Gaussian blur, logarithmic transformation and bilateral filtering, to the grayscaled input image. After extracting the edges of the image, using the Canny edge detector, we apply a morphological closing, so that the cracks are represented in a clean and binary way. Finally, the images will be merged following a bruteforce approach by finding the minimum of the L2 distances between the two pre-processed crack views.

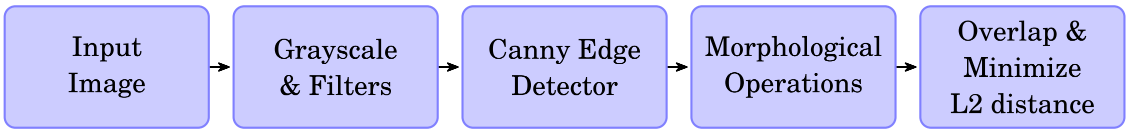


Figure 2: Overview of the individual steps of the pipeline.

2 Tasks

The provided framework contains a `main.cpp` (which must not be edited as it is used for automatic execution) and an `algorithms.cpp` where the tasks should be implemented in the respective functions (`compute_grayscale`, `gaussian_blur`, `compute_log_transform`, `apply_bilateral_filter`, `canny`, `apply_morph_operation`, `L2_distance_transform`, `match_cracks`, `blend_originals`, as well as for the bonus task `compute_gradient`, `non_maxima_suppression` and `hysteresis`). The required parameters for each test case (e.g., `crack1`) are stored in separate JSON files (e.g. `crack1.json`) located in the folder `tests` in your repository. The file `crack1.json` has to be passed to the application to execute the test case `crack1`. Then, the input data is fetched automatically from the JSON file and can be used directly for the calculations in the respective functions. Do not modify these parameters. They are also passed to the respective subroutines for you. The content of this task is limited solely to the 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. Note that the tests build upon each other: Diffs in one image will (likely) result in diffs for subsequent images.

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

```
Task 1 [16 + 3 pts]
├─Grayscale [1pt]
├─Gaussian Blur [1pt]
├─Logarithmic Transform [1pt]
├─Bilateral Filter [2pts]
├─Canny Edge Detector [1pt]
├─Morphological Operations [3pts]
├─Crack Matching [5pts]
│ ├─Distance Transform [2 of 5pts]
│ └─Distance Minimization and Mask Creation [3 of 5pts]
├─Blending Originals [2pts]
└─Bonus - Custom Canny Edge Detector [3pts]
  ├─Gradient Calculation [1 of 3pts]
  ├─Non-Maxima Suppression [1 of 3pts]
  └─Hysteresis Thresholding [1 of 3pts]
```

This task has to be implemented using OpenCV[1] 4.5.4. 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 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 accessing 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.

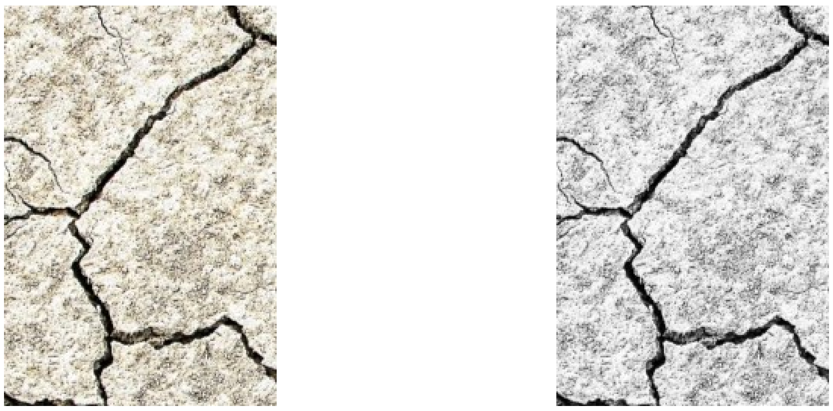
There are two things to keep in mind when fetching color information from `cv::Mat` objects in OpenCV:

- 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

$$I = 0.2989 \cdot R + 0.5870 \cdot G + 0.1140 \cdot B \tag{1}$$

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 can directly be written to the passed single-channel `cv::Mat` reference `grayscale_image`. Fig. 3 shows an example of the expected output.



(a) RGB input image.

(b) Grayscale output image.

Figure 3: Expected output for `compute_grayscale(...)`.

Forbidden Functions:

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

2.2 Gaussian Blur (1 Point)

In the function `gaussian_blur(...)`, you have to blur the image according to the parameters σ and k_s. This is done by convolving the image with a Gaussian kernel $K \in \mathbb{R}^{k_s \times k_s}$, where k_s denotes the kernel size. To retrieve a 1D Gaussian kernel use the OpenCV function `cv::getGaussianKernel(...)`. This returns a filter $\mathbf{k} \in \mathbb{R}^{k_s \times 1}$. Next, compute K using an outer product (eq. 2) and use `cv::filter2D(...)` to convolve the image with the kernel K. Use -1 for the `ddepth` parameter.

$$K = \mathbf{k}\mathbf{k}^T. \tag{2}$$

Useful Functions:

- `cv::getGaussianKernel(...)`
- `cv::filter2D(...)`
- `cv::mulTransposed(...)`

Forbidden Functions:

- `cv::GaussianBlur(...)`

2.3 Logarithmic Transform (1 Point)

In the function `compute_log_transform(...)`, you will adjust the intensities of the input image's pixels by applying a logarithmic transformation. By looking at the log-shaped graph in fig. 4, you can see that a narrow range of low input intensities will be mapped to a wider range of output levels. On the contrary, a wide range of high input intensities will be compressed and yields a narrow range of output levels. These coherences positively contribute to our need of being able to clearly identify the dark crack areas.

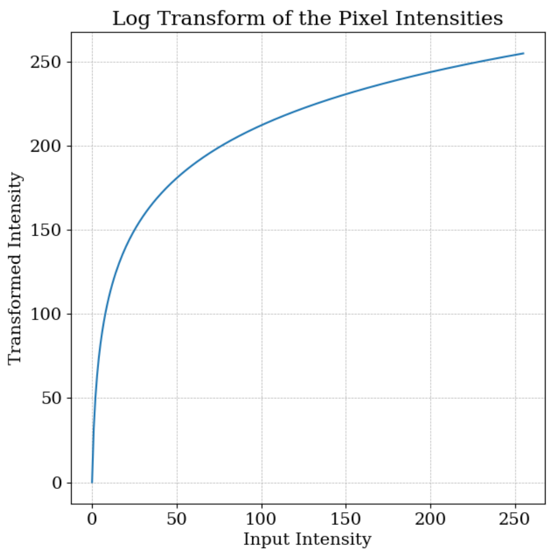


Figure 4: Logarithmic transformation curve according to equation 3.

To scale the output image to the available grayscale spectrum, you firstly need to normalize the logarithmic transform according to the maximum value in the input image. Additionally, multiply the outcome with the maximum value of the spectrum, i.e., 255. Keep in mind that log(0) is not defined. Therefore, add 1 to the according arguments.

The logarithmic transform can be summarized with eq. 3

$$\mathbf{y}(\mathbf{x}) = \frac{\log(\mathbf{x}+1)}{\log(1+\max(\mathbf{x}))} \cdot 255 \tag{3}$$

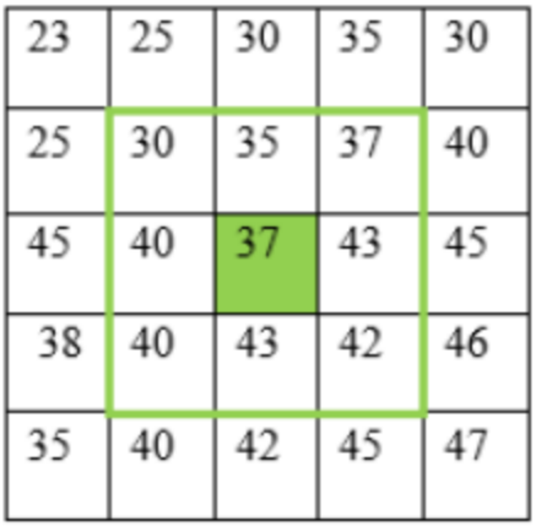
where $\mathbf{x}$ denotes the input image and $\mathbf{y}$ the transformed image.

Useful Functions:

- `cv::log(...)`
- `cv::minMaxIdx(...)`

2.4 Bilateral Filter (2 Points)

Before applying edge detection, an essential preprocessing step is noise removal, as noise can interfere with accurate edge detection in subsequent algorithms. However, simple Gaussian blurring provides isotropic denoising, which considers weighted neighboring pixel locations but also smooths over edges, making them blurry. A more effective approach is to use a bilateral filter, which, in addition to the spatial component, accounts for pixel intensity variations. This filter performs an averaging operation that considers both spatial and intensity differences among neighboring pixels. You will implement the bilateral filtering process in the function `apply_bilateral_filter(...)`. This involves iterating over each pixel in the log-transformed output and treating it as the center within a given window size of $d = 5$ (5x5 neighborhood).



23	25	30	35	30
25	30	35	37	40
45	40	37	43	45
38	40	43	42	46
35	40	42	45	47

Figure 5: This figure illustrates the pixels within a 3×3 local neighborhood where the filter mask will be applied.

The filtering process computes a weighted sum of neighboring pixel intensities, where the weights are determined based on both spatial distances and intensity similarities (eq. 4).

$$h[m,n] = \frac{1}{W_{mn}} \sum_{k,l} g_{mn}[k,l]\, r_{mn}[k,l]\, f[k,l] \tag{4}$$

Here, h is the output image, m and n are the center-pixel coordinates and f refers to the input image. k and l are the kernel coordinates, i.e., the absolute coordinates of the pixels within the considered local neighborhood.

Normalization factor $\mathbf{W_{mn}}$:

$$W_{mn} = \sum_{k,l} g_{mn}[k,l] r_{mn}[k,l] \tag{5}$$

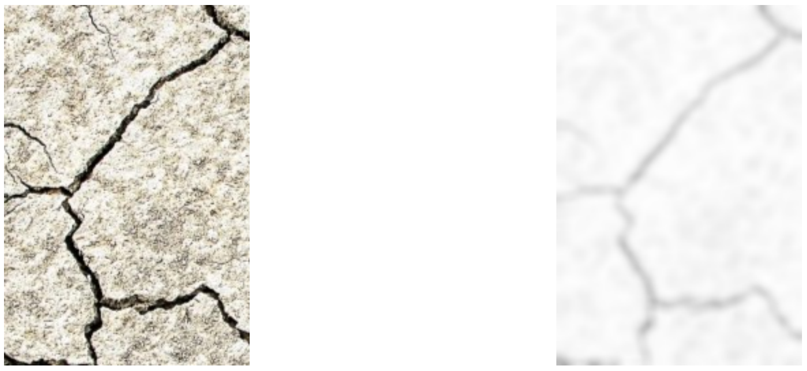
Spatial weighing $\mathbf{g_{mn}}$:

$$g_{mn}[k,l] = \exp\left(-\frac{(m-k)^2 + (n-l)^2}{2\sigma_s^2}\right) \tag{6}$$

Intensity range weighing $\mathbf{r_{mn}}$:

$$r_{mn}[k,l] = \exp\left(-\frac{(f[m,n]-f[k,l])^2}{2\sigma_r^2}\right) \tag{7}$$

The image will get smoother on parts that are not near edges (smoothing) and will not get smoother on parts that are near the edge (edge preserving).



(a) RGB original image. (b) Bilateral filtered image.

Figure 6: Expected output for `apply_bilateral_filter(...)`.

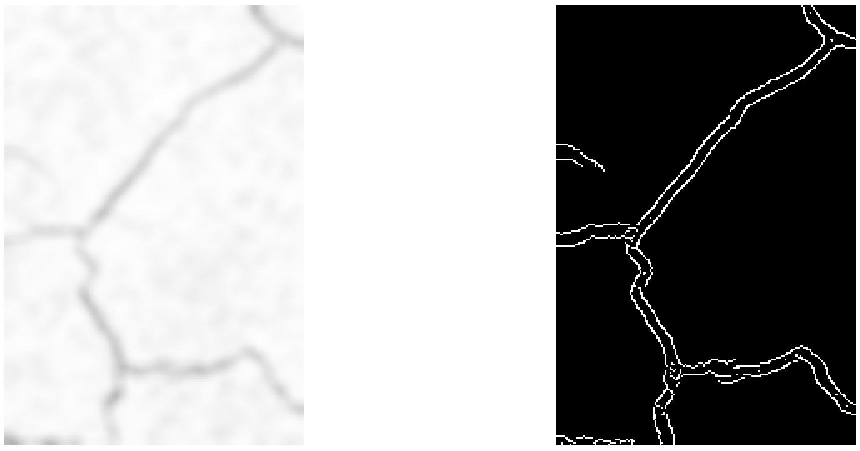
Forbidden Functions:

- `cv::bilateralFilter(...)`

2.5 Canny Edge Detector (1 Point)

In order to determine edge points within the pre-processed image, you will use the function `canny(...)` to apply the famous Canny edge detection algorithm [1]. The algorithm can be broken down into several steps, including gradient determination, non-maxima suppression and double thresholding following a hysteresis approach. These parts will be discussed in the bonus section of this assignment in greater detail. For the standard task, just make sure to use the two thresholds accordingly. The lower threshold describes the minimum gradient value to classify a pixel as a weak edge, the higher threshold describes the minimum gradient value to classify a pixel as a strong edge. It requires further logic to understand what this differentiation achieves. Please refer to the description in the bonus section for additional information on that matter.

An example of the Canny edge detection can be found in fig. 7.



(a) Source image containing (smooth) edges.

(b) Detected edges using Canny's algorithm.

Figure 7: Canny edge detection in use.

Useful Functions:

- `cv::Canny(...)`

2.6 Morphological Operations (3 Points)

The function `apply_morph_operation(...)` is used to apply different morphological operations (erosion, dilation or combinations) to a given image. The reason for that is to create a possibility to remove unwanted components within the image by enlarging/shrinking certain regions depending on their surroundings.

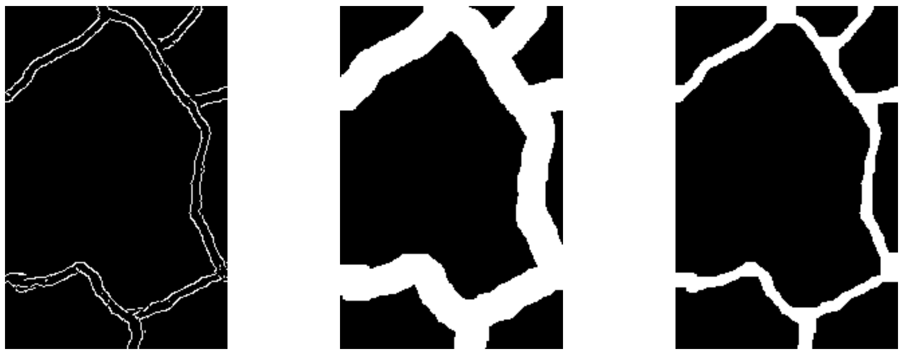
In other words, erosion causes each pixel to be set to the darkest value in its surroundings, which leads to a size increase of dark areas and shrinks the brighter ones. Dilation works vice versa. Each pixel will be set to the brightest value in its surroundings, which leads to larger bright areas and smaller dark areas. Therefore, small gaps between bright components of an image can be eliminated by applying morphological dilation. In order to prevent the output image from being a filled, bright, though bloated version of the input image, it is necessary to apply erosion after dilation, which shrinks the main component again without destroying the achieved continuity. This process is called morphological **closing**.

The surrounding of each pixel that will be considered during erosion/dilation is described by the **structuring element** or **kernel**. In our case, the kernel should be a NxN-sized square, where N is given by the parameter `kernel_size`.

Your task is to implement the two basic morphological operations erosion and dilation - without the use of existing OpenCV functions - in order to be able to receive continuous cracks for further computations.

Please consider following requirements for `apply_morph_operation(...)`:

- Perform an erosion when the parameter `mode` is equal to `cv::MORPH_ERODE` and a dilation when `mode` equals `cv::MORPH_DILATE`.
- Loop through the pixels of the image and fixate the center of the kernel. It is not necessary to set the anchor of the kernel to every single pixel. Only consider black pixels for dilation and non-black pixels for erosion.
- Consider every pixel within the kernel around the defined center point for the min/max value comparison - except any pixels that do not lie within the dimensions of the image.
- The value of the center pixel is likely to be overwritten. Assign the maximum value of the surrounding region for dilation, and the minimum value for erosion.
- You can assume that the kernel size is odd.



(a) Detected edges. (b) Dilation applied. (c) Closing applied.

Figure 8: Expected output for `apply_morph_operation(...)`. The gaps between the edges are eliminated by applying dilation. The original size is restored by a subsequent erosion (closing).

Hint: Dilation and erosion are linked to a closing event in the background. Your task is only to implement dilation and erosion separately and return the resulting image by storing it in the parameter `morphed_image`!

Forbidden Functions:

- **`cv::dilate(...)`**
- **`cv::erode(...)`**
- **`cv::morphologyEx(...)`**
- **`cv::getStructuringElement(...)`**

2.7 Crack Matching (5 Points)

In the scope of this assignment, the actual crack matching relies on a simple bruteforce approach. The two crack-views are overlapped at different positions for each of which a distance transform is computed. Determining the alignment that yields the minimal distances, results in the best possible match, represented in the form of two masks (fig. 10), one for each image. The masks are consequently used by `blend_originals(...)` to blend the original pictures for a final crack-matched output image.

2.7.1 Distance Transform (2 of 5 Points)

In the function `L2_distance_transform(...)` you will compute the transform of a given image based on the two-dimensional Euclidean distance metric (eq. 8). For that to achieve, iterate through the entire binary input image. For every white (= 255) pixel p examine its 5x5 neighborhood. Calculate the distance d between each neighboring black (= 0) pixel p_0 and p according to eq. 8 and set the minimal value to the parameter `transformed_image` at the position of p. In case you find no neighboring pixel to be black, assign the transformed pixel with the numeric limit `FLT_MAX`.

$$d = \sqrt{(p_x - p_{x,0})^2 + (p_y - p_{y,0})^2} \tag{8}$$

Important:

While working with floats during the computation of the minimal distance, explicitly set the distance value as `uint8_t` in the transformed image.

Forbidden Functions:

- `cv::distanceTransform(...)`

2.7.2 Distance Minimization and Mask Creation (3 of 5 Points)

In the function `match_cracks(...)` your goal is to find the relative position of the two morphologically closed images to one another that yields the minimal overall distance-sum. You can assume the two images to overlap by at least 25 % of the second image's size. Therefore, please consider a possible matching-frame as provided with the parameter `matching_frame` (fig. 9).

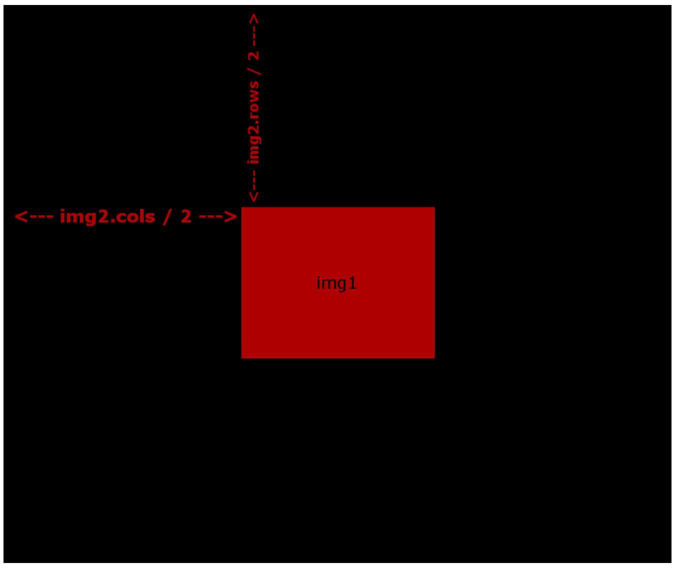


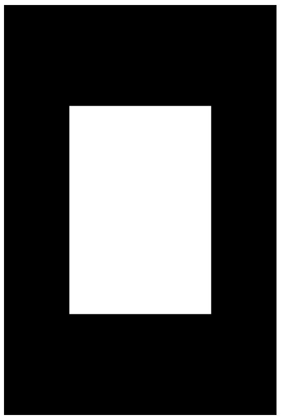
Figure 9: Dimensions of the matching-frame.

To find the minimal distances iterate over the (all-black) matching-frame. Place the first image in the center of the matching-frame and initially add the second image in the top left corner. Now that both images overlap, compute the distance of each pixel using your own function `L2_distance_transform(...)` and keep track of the minimal distances in an according matrix - update it when a new minimal distance-sum was found. With each iteration, renew the position of the second image in the matching-frame according to the current iteration's row and column and recalculate the distances. The first image always stays in the center.

Due to the fact, that the here practiced bruteforce approach is quite costly, you will not iterate over each pixel of the matching-frame. Instead, use the given `step` parameter as an iteration step size.

Creating the mask:

Once the optimal position of the second image is found, you can create the masks (fig. 10). The white masks are sized and positioned exactly like the images used for the distance minimization, i.e. the mask of the first image is centered, the mask of the second image is set according to the position that minimizes the previously calculated sum of distances.


(a) Mask of the first image. Always centered.

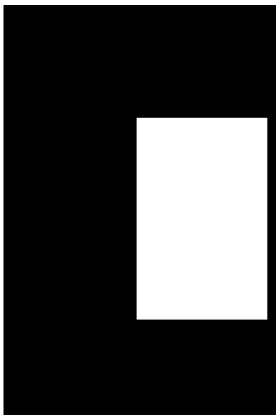

(b) Mask of the second image.

Figure 10: White masks representing the optimal positions of the original images after the crack matching.

Return the two masks in the parameters `mask_img1` and `mask_img2`. The size of these initially black matrices is equal to the size of the matching frame. Color the masked pixels white according to your computations.

Additionally, return the matrix containing the minimum distances in the parameter `minimal_distances`.

Useful Functions:

- `clone(...)`
- `copyTo(...)`
- `cv::Rect(...)`
- `cv::add(...)`
- `cv::sum(...)`
- `cv::Mat::ones(...)`
- `std::numeric_limits<long>::max()`

2.8 Blending Originals (2 Points)

In the `blend_originals(...)` function, your goal is to merge two images using their binary masks, which are sized to match the entire shape of each original image. Since the white rectangle in each mask corresponds exactly to the full original image, simply replace the white pixels (255) in each mask with the corresponding pixels from its original image. You can view each mask as a function $M(x,y)$ that selects the original image pixels only where $M(x,y) = 255$. After populating these white regions with the full content of both original images, blend them together by first taking valid (non-black) pixels from the first modified image and then filling any remaining pixels with those from the second modified image. The result is a single output where both originals appear in their correct positions and complement each other visually.

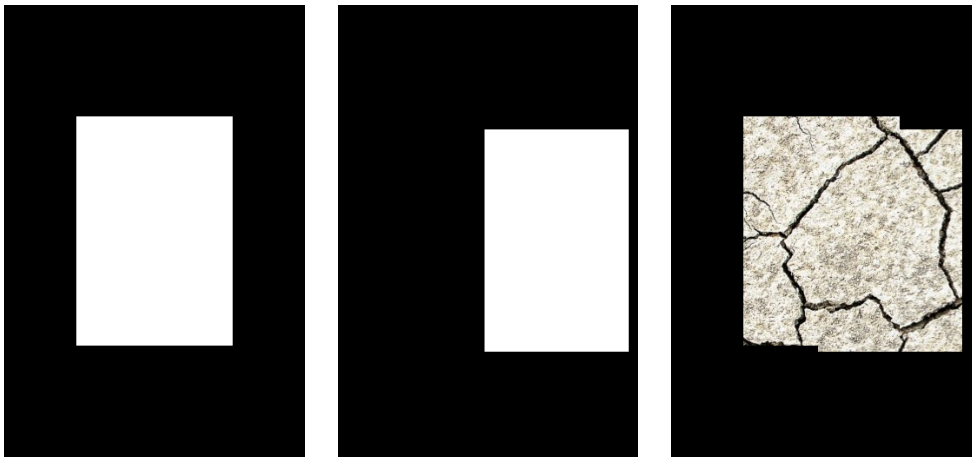

(a) Original image 1 mask. (b) Original image 2 mask. (c) Blended images.

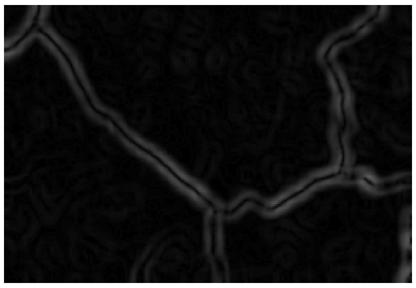
Figure 11: Blending two images using binary masks.

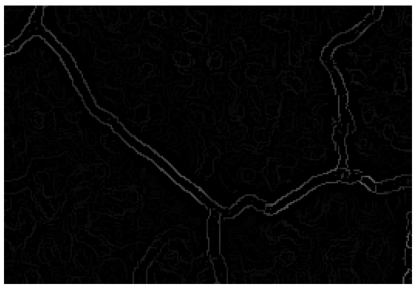
2.9 Bonus - Custom Canny Edge Detector (3 Points)

The Canny edge detector is a popular technique used in image processing to identify edges in an image. Implementing a custom version involves several steps: applying a Sobel operator to find gradient magnitude and direction, performing non-maximum suppression to thin out the edges, and using double thresholding to detect strong and weak edges, followed by edge tracking by hysteresis to finalize the detection (fig. 12).

2.9.1 Gradient Calculation (1 of 3 Points)

(a) Bilateral filtered input image.


(b) Gradient magnitude.


(c) Non-maxima suppression.

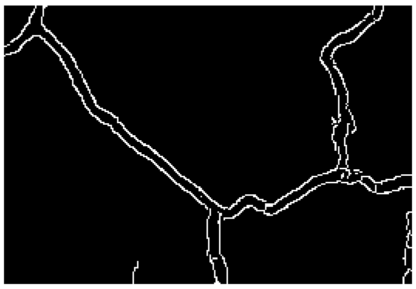

(d) Detected edges.

Figure 12: Pipeline of the custom Canny edge detector.

This part of the task has to be implemented in the function `compute_gradient(...)`. In Canny edge detection, this step involves calculating the gradient magnitude to identify potential edges in an image. This is achieved by applying derivative filters to measure changes in intensity across the image. For this purpose, the Sobel operator is often utilized due to its effectiveness in emphasizing edges.

The gradient magnitude, $\|\nabla f\|$, can be calculated using:

$$\|\nabla f\| = \sqrt{\left(\frac{\partial f}{\partial x}\right)^2 + \left(\frac{\partial f}{\partial y}\right)^2}$$

where $\frac{\partial f}{\partial x}$ and $\frac{\partial f}{\partial y}$ represent the first derivatives of the image in the x and y directions, respectively. These derivatives are typically computed using the Sobel kernels:

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}, \quad G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

The calculation of the gradient magnitude (fig. 12b) highlights the areas of highest intensity change, which correspond to edge boundaries. In the context of Canny edge detection, high values of $\|\nabla f\|$ indicate strong edge candidates. The method relies solely on the magnitude of gradients for initial edge detection, disregarding orientation, which simplifies the process and focuses on detecting the most significant edges.

Useful Functions:

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

2.9.2 Non-Maxima Suppression (1 of 3 Points)

This part of the task has to be implemented in the function `non_maxima_suppression(...)`. Non-Maxima Suppression (NMS) (fig. 12c) is a crucial step in the Canny edge detection process, designed to thin out the edges by retaining only the local maxima in the gradient magnitude of the image. This process ensures that the resulting edges are sharp and well-defined.

After calculating the gradient magnitudes using a Sobel operator, each pixel's gradient direction is determined by:

$$\alpha = \arctan\left(\frac{G_y}{G_x}\right)$$

where G_x and G_y are the gradients in the x and y directions, respectively. Angles are then normalized:

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

Based on the calculated angle β, a pixel p is classified into one of four categories to determine its neighboring pixels q and r for comparison:

Class	**Angle Range**	**Neighbor Relation**
1 | horizontal	$\beta \leq 22.5°$ or $\beta > 157.5°$	$q = (p_x - 1, p_y), r = (p_x + 1, p_y)$
2 | diagonal-1	$22.5° < \beta \leq 67.5°$	$q = (p_x - 1, p_y - 1), r = (p_x + 1, p_y + 1)$
3 | vertical	$67.5° < \beta \leq 112.5°$	$q = (p_x, p_y - 1), r = (p_x, p_y + 1)$
4 | diagonal-2	$112.5° < \beta \leq 157.5°$	$q = (p_x - 1, p_y + 1), r = (p_x + 1, p_y - 1)$

Table 1: Classification table of the angles and relationship to the neighboring pixels.

The suppression rule is applied as follows:

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

where $\hat{G}(p)$ is the result of the non-maxima suppression, and $G(p)$, $G(q)$, and $G(r)$ are the gradient magnitudes of the current pixel and its neighbors, respectively.

This process is applied across the entire image, ensuring that only the strongest edges — those that are local maxima in their direction — are retained. As a result, the edges become thinner, avoiding the problem of wide, blurry edges and achieving a single-pixel thickness wherever possible.

Useful Functions:

- `fmod(...)`
- `std::atan2(...)`

2.9.3 Hysteresis Thresholding (1 of 3 Points)

Hysteresis thresholding is implemented in the function `hysteresis(...)`. The already found and refined edges should be compared with the thresholds τ_{min} and τ_{max}. Each pixel p is classified, whether it belongs to a weak, a strong, or no edge. If the value of pixel p is greater than the upper threshold τ_{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 τ_{min} then this pixel is certainly not part of an edge and therefore not part of the final result. If the value of pixel 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:

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

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). If this is the case, the weak edge becomes a strong edge. 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. Furthermore, search 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, $H(p) \in \{0, 255\}\ \forall p$ (fig. 12d).

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, PAMI-8(6):679–698, 1986.