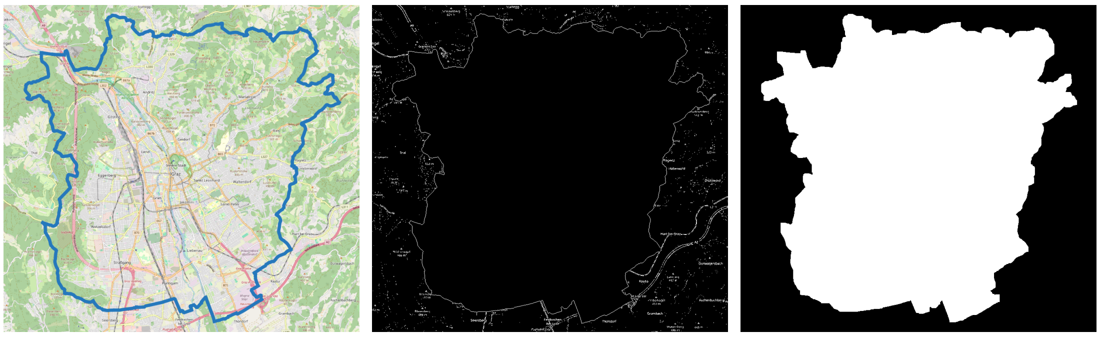

(a) Input image. (b) Detected contours. (c) Extracted area.

Figure 1: The pipeline applied to an input image. On the left the original image, in the middle the detected contours and on the right the filled area of the given region after a morphological opening.

CV/task1 — Geographic Computations

1 Overview

This task serves as an introduction to working with the popular OpenCV library. The goal is to determine the area, perimeter, and center of mass of a given geographic region, which requires several steps of image pre-processing. Fig. 1 shows a stripped-down example of the pipeline in Fig. 2.

To find the desired values, we firstly convert the grayscaled input image to a binary image. After extracting the outer contours of the region, the geographic area needs to be filled and cleaned using a morphological opening. Finally, the unknown variables can be calculated by analyzing the white pixels and applying certain conversion factors.

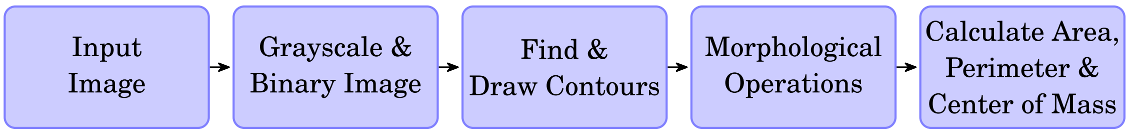


Figure 2: Overview of the individual steps of the pipeline.

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, compute_histogram, compute_threshold, compute_binary, draw_contours, apply_morph_operation, calc_area, calc_perimeter, calc_center_of_mass, crop_center_of_mass, and for the bonus task bonus_find_contours).

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 should 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
 |__Compute Grayscale  [1pt]
 |__Compute Binary  [5pts]
 |   |__Compute Histogram [1pt]
 |   |__Compute Optimal Threshold [3pts]
 |   |__Apply Threshold [1pt]
 |__Find & Draw Contours [1pt]
 |__Apply Morphological Operations [3pts]
 |__Calculate Area [1pt]
 |__Calculate Perimeter [2pts]
 |__Calculate Center of Mass [1pt]
 |__Crop Center of Mass [2pts]
 |__Bonus - Find Contours [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 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 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.

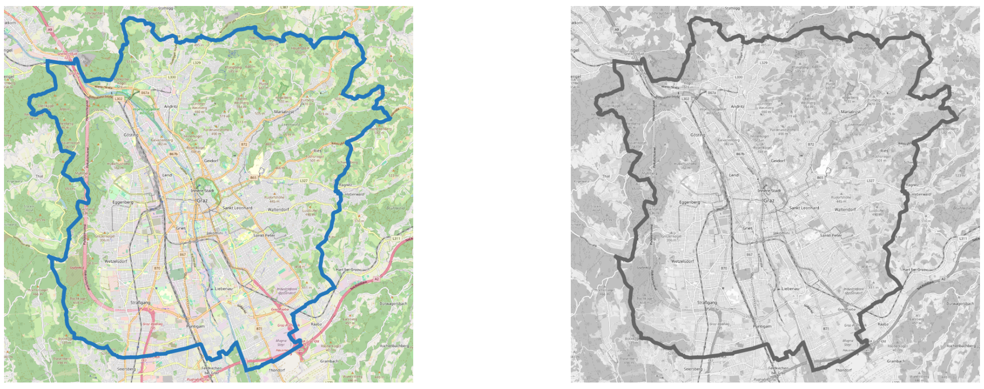
There are two things to keep in mind when fetching color information from `cv::Mat` objects in OpenCV:

- 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` after the conversion. 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 Compute Binary (5 Points)

The next task in our pipeline requires you to convert the previously calculated grayscale image into a binary image, a fundamental concept in most image-processing pipelines that enables easier processing. Unlike grayscale images, binary images contain only two types of values, typically 0 (black) and 255 (white).

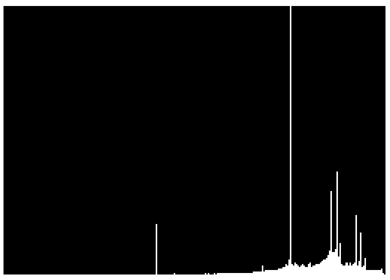
Converting grayscale to binary images involves three crucial steps. For this task, each of these steps must be implemented in the respective functions:

- Calculate the histogram of pixel intensities
- Find an optimal threshold for (binary) image segmentation
- Apply the threshold to convert the grayscale image into a binary image

2.2.1 Compute Histogram of Pixel Intensities (1 of 5 Points)

The first step is the computation of the histogram, which you should implement in the function `compute_histogram(...)`. A histogram represents the distribution of pixel intensities within the grayscale image, in our case value in the range 0 (black) ... 255 (white). The parameter `normalized_histogram` is an array of size 256 (number of bins), which has been initialized for you. Thus, every index, i.e., bin, represents a different pixel intensity.

Examples of such histogram images can be seen in Figure 4.


(a) Histogram of `/graz/01_grayscale.png`.

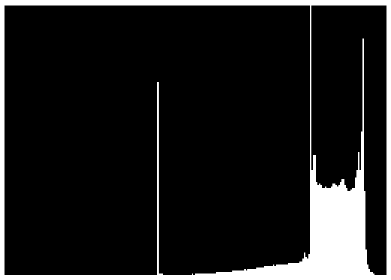

(b) Histogram of `/austria/01_grayscale.png`.

Figure 4: Example histogram images.

Your task in the function `compute_histogram(...)` is to:

- Iterate through all pixels in the grayscale image
- Sum up and update the bins for the current pixel intensity
- Normalize each bin of the histogram at the end

Forbidden Functions:

- `cv::calcHist(...)`

2.2.2 Find an Optimal Threshold Value (3 of 5 Points)

To calculate the binary image, we classify each grayscale pixel value as either 0 (black) or 255 (white). We also refer to this as (binary) image thresholding, or segmentation into two classes (typically called foreground and background). The threshold value serves as our reference for this classification, as it allows us to classify pixel values below the threshold into one of the classes, e.g., foreground, and vice versa.

This task should be implemented in the function `compute_threshold(...)`. You should compute an optimal threshold value for binary image segmentation by hand using a simplified version of OTSU's method [1]. It is a widely used method to compute an optimal threshold by maximizing inter-class variance between the background and foreground pixel intensity values. For just two classes, the inter-class variance σ^2 can be calculated using the following formula:

$$\sigma^2(t) = w_0(t) \cdot w_1(t) \cdot \left[\mu_0(t) - \mu_1(t)\right]^2 \tag{2}$$

The weights, or probabilities, $w_{0,1}(t)$ of both foreground and background at the threshold t, calculated as

$$w_0(t) = \sum_{i=0}^{t-1} p(i), \tag{3}$$

and

$$w_1(t) = \sum_{i=t}^{L-1} p(i), \tag{4}$$

respectively, where $p(i)$ denotes the normalized pixel intensity and L is the number of bins.

Since we are working with normalized histogram data, and thus weights in the range [0...1], we can use the following relation to calculate $w_1(t)$ more efficiently:

$$w_0(t) + w_1(t) = 1. \tag{5}$$

The class means $\mu_{0,1}(t)$ represent the weighted mean pixel intensities of both foreground and background at the threshold t, calculated as

$$\mu_0(t) = \frac{\sum_{i=0}^{t-1} i \cdot p(i)}{w_0(t)}, \tag{6}$$

and

$$\mu_1(t) = \frac{\sum_{i=t}^{L-1} i \cdot p(i)}{w_1(t)}, \tag{7}$$

respectively, where $p(i)$ denotes the normalized pixel intensity, weighted by the potential threshold i and the probability $w_{0,1}(t)$.

To summarize, you must implement the following tasks to compute the optimal threshold value in the function `compute_threshold(...)`:

- Iterate through all possible threshold values t in the range [1...255].
- For each threshold t, calculate the weights, i.e., class probabilities, $w_0(t)$ and $w_1(t)$ according to Eq. 3 and Eq. 4.
- Furthermore, calculate the weighted mean pixel intensities $\mu_0(t)$ and $\mu_1(t)$ according to Eq. 6 and Eq. 7.
- Find the optimal threshold value that maximizes the inter-class variance according to Eq. 2
- Return the optimal threshold value by saving it to the parameter `threshold`.

Forbidden Functions:

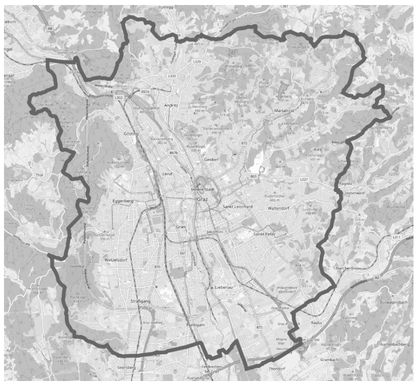
- **`cv::threshold(...)`**

2.2.3 Binary Thresholding (1 of 5 Points)

Finally, the function `compute_binary(...)` is where we apply the optimal threshold value to convert the grayscale input image into a binary image. To this extent, we compare each pixel value of the grayscale image against the threshold t to determine its new color.

For our task, we want to convert the grayscale image into an inverted binary image. Thus, for pixel intensities above the threshold, the corresponding pixel in the binary image should be set to black (0), otherwise white (255).

$$binary(x,y) = \begin{cases} 0, & \text{if } grayscale(x,y) > t \\ 255, & \text{otherwise} \end{cases} \tag{8}$$


(a) Grayscale input image.

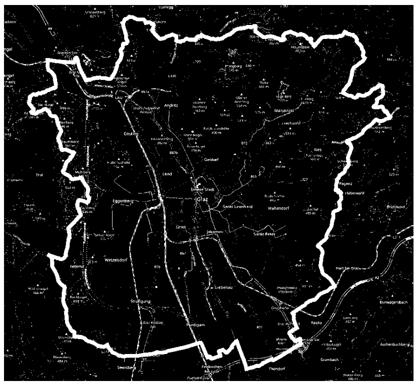

(b) Binary output image.

Figure 5: Expected output for `compute_binary(...)`.

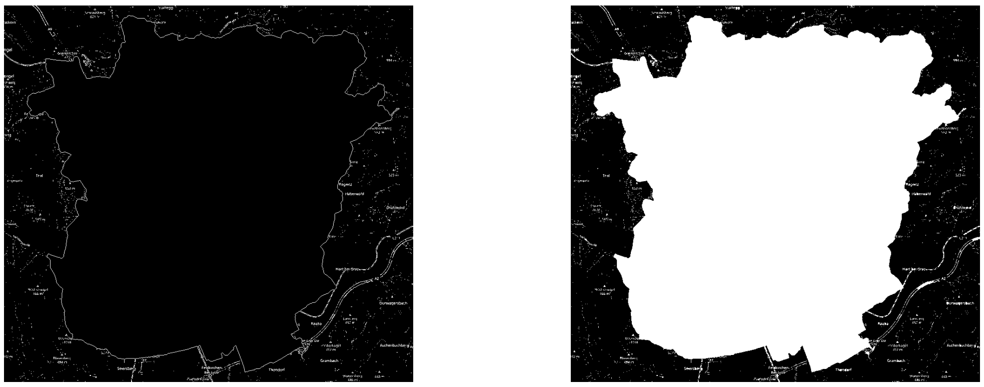
2.3 Find & Draw Contours (1 Point)

This task should be implemented in the function `draw_contours(...)`. Here, we aim to find and extract contours from the previously calculated binary image. For the standard task, we will use OpenCV functions to achieve this goal. For the bonus task, you will have to implement the algorithm to find contours yourself (see Section 2.9).

Contours represent the outline, i.e., boundaries, of an image object and in our case geographical region. Use `cv::findContours(...)` to find contours in the binary image. Pass the appropriate parameter `cv::RETR_EXTERNAL` to detect outer contours only. For approximation use `cv::CHAIN_APPROX_SIMPLE`. Be aware that this will return a 2-dimensional `std::vector`, populated with all pixels (given as `cv::Point`) for each detected contour.

Furthermore, we want to fill the enclosed inner area of our detected contours. Use `cv::drawContours(...)` and pass appropriate parameters to draw all identified contours. Use the `cv::FILLED` parameter to fill enclosed areas, and choose the fill color such that the enclosed area is white (255).

Remember to store both the detected contours and the binary image with filled contours in `contours` and `filled_image`, respectively.



(a) Detected contours. (b) Filled contours.

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

Useful Functions:

- `cv::findContours(...)`
- `cv::drawContours(...)`

2.4 Apply 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, dilation causes each pixel to be set to the brightest value in its surrounding, which leads to a size increase of bright areas and shrinks the darker ones. Erosion works vice versa. Each pixel will be set to the darkest value in its surrounding, which leads to larger dark areas and smaller bright areas. Therefore, small, bright and discontiguous regions can be fully eliminated by applying morphological erosion. In order to prevent the output image from being a cleaned, though shrunk version of the input image, it is necessary to apply dilation after erosion, which enlarges the main component again without reintroducing the noise. This process is called morphological **opening**.

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 cleaned images for further (area) calculations.

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

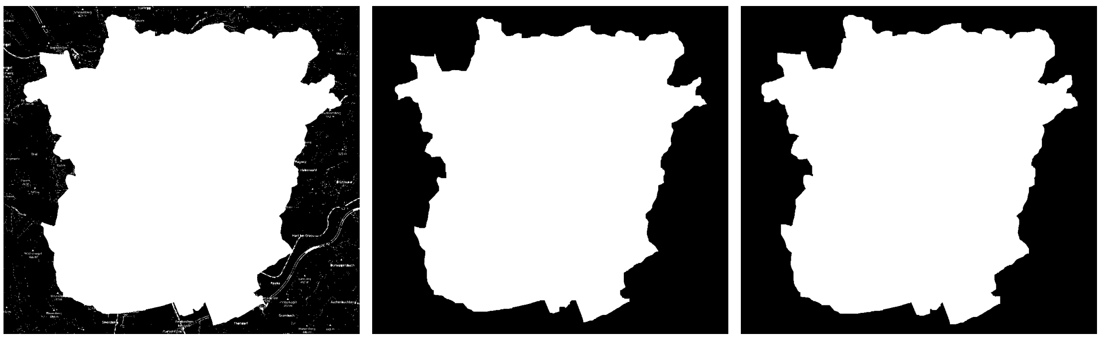
- Perform an erosion when the parameter `mode` is equal to `cv::MORPH_ERODE` and a dilation when `mode` equals `cv::MORPH_DILATE`.

- 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) Noisy input image. (b) Erosion applied. (c) Opening applied.

Figure 7: Expected output for `apply_morph_operation(...)`. The noise is eliminated by applying erosion. The original size is restored by a subsequent dilation (opening).

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

Forbidden Functions:

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

2.5 Calculate Area (1 Point)

After extensive image pre-processing it is time to compute relevant geometrical parameters. The function `calc_area(...)` should calculate the area of the given geographical region by analyzing the pixels of the pre-processed image.

The parameter `normalized_image` holds a normalized version of the morphologically opened image. This means, all pixels within the regional border (i.e. all white pixels) were set to 1, and all black pixels kept the value 0.

Your task is to determine the occupied area both in pixels as well as in square kilometers. Please make sure to use the given `conversion_factor_m2` which expresses how many square meters are represented by one pixel in the given scale of the input image and keep in mind to convert the result accordingly to $\mathbf{km^2}$.

Hint: The results will be printed to the console automatically and will also be stored in a corresponding .txt file in the output folder.

2.6 Calculate Perimeter (2 Points)

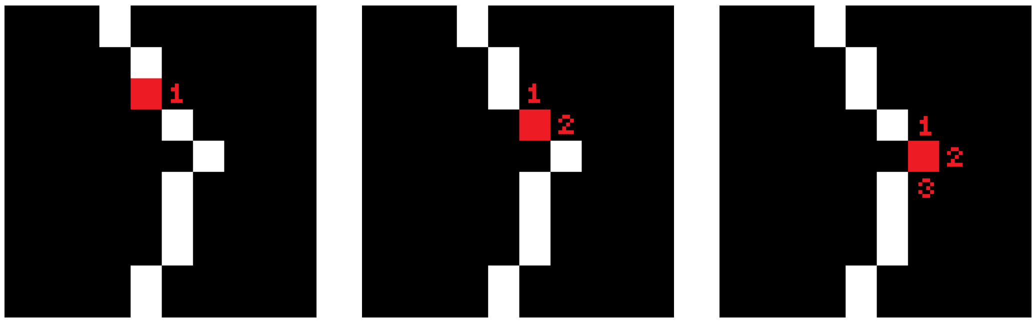
The function `calc_perimeter(...)` should calculate the total perimeter of the given outline of the geographical region, both in pixels and kilometers. Again, we achieve this by analyzing the pixels of the pre-processed image.

You will be given a 2-dimensional `std::vector`, populated with all pixels (given as `cv::Point`) for each detected contour. The given contour for this task has a maximum width of exactly 1 pixel. We approximate the total outer perimeter by counting the outer edges for all outline pixels and convert the result to real-world units.

Since the image might still contain some pixel artifacts, which may be detected as contours, your first task is to find the largest contour in the 2D array of contours. You can assume that this will be the correct outline of the geographical region. For the largest detected contour, you should implement the following logic:

- The total perimeter is calculated as the sum of all outer edges.
- For each pixel in the contour, count the outer edges by checking its 4-neighborhood.
- For each neighboring pixel, utilize `cv::pointPolygonTest(...)` to check if it, i.e., the edge of the pixel, lies outside the contour.
- Add to the perimeter for each edge outside the contour (`cv::pointPolygonTest(...)` returns -1 in this case).

A visual representation of this approach is given in Figure 8.



(a) Pixel with 1 outer edge. (b) Pixel with 2 outer edges. (c) Pixel with 3 outer edges.

Figure 8: Example perimeter contour (white). Here, **the outer area is to the right**.

Finally, make sure to use the given `conversion_factor_m` which expresses how many meters are represented by one pixel in the given scale of the input image and keep in mind to convert the result accordingly to **km**.

Hint: The results will be printed to the console automatically and will also be stored in a corresponding .txt file in the output folder.

Useful Functions:

- `cv::pointPolygonTest(...)`

2.7 Calculate Center of Mass (1 Point)

In `calc_center_of_mass(...)` you will have to calculate the center of mass and store it as a `cv::Point` object in the parameter `center_of_mass`. For that, take a look at the given morphologically opened and normalized image and apply the following formulas concerning the weighted average:

$$x_{center} = \frac{\sum_{i=0}^{N-1} \sum_{j=0}^{M-1} j \cdot I_{i,j}}{A} \tag{9}$$

$$y_{center} = \frac{\sum_{i=0}^{N-1} \sum_{j=0}^{M-1} i \cdot I_{i,j}}{A} \tag{10}$$

Be aware that **NxM** are the dimensions of the given image, $\mathbf{I_{i,j}}$ represents the pixel value at position (i|j) and **A** is the area of the identified region in pixels.

Hint: The results will be printed to the console automatically and will also be stored in a corresponding .txt file in the output folder.

2.8 Crop Center of Mass (2 Points)

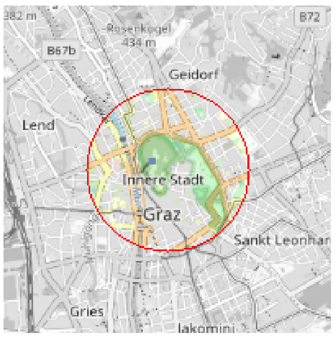
The last missing piece of the pipeline involves creating the final output image. For this, we want to crop and highlight the calculated center of mass in the input image. You should implement this task in the function `crop_center_of_mass(...)`.

To be precise, we want a final grayscale output image with a highlighted colored region (around the center of mass). Additionally, the output image size should be cropped to 200x200 pixels. All relevant size parameters are given to you as constant variables in the code.

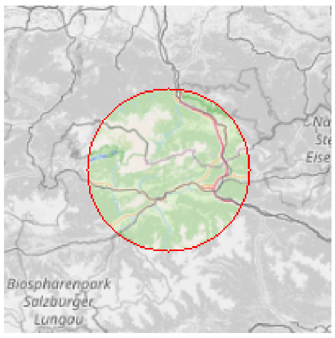
The general outline for this task is:

- Convert the single-channel grayscale image to a 3-channel image. You may utilize `cv::cvtColor(...)` for this.
- Create a circular mask of size `crop_radius`, centered at the calculated center of mass.
- Copy or replace the masked circular area in the grayscale image with the colored input image.

- Highlight the masked circular region by drawing a red circle with radius `crop_radius` around the center.
- Finally, crop the final image to the correct size. The expected size is given to you as `crop_radius`.
- Save the final image to the parameter `cropped_center`.



(a) Cropped center for input image `graz.png`.



(b) Cropped center for input image `austria.png`.

Figure 9: Expected output for `crop_center_of_mass(...)`.

Useful Functions / Data Types:

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

2.9 Bonus - Find Contours (3 Points)

In the third task, you used the OpenCV function `cv::findContours(...)` to extract contours from a binary image. The algorithm that this function implements was introduced by Suzuki and Abe [2] and published in 1985. You can earn up to 3 bonus points by implementing a simplified version of the algorithm yourself.

On a high level, the algorithm proceeds in the following way: The binary image is scanned row by row until a non-zero pixel is reached. For this task, the values of the binary image will be either 0 or 1. Then, we walk along the contour counterclockwise and store and mark each pixel that belongs to it until we eventually return to the pixel from which we started. We continue to scan the image, adding contours as we find them until we reach the bottom right corner of the image. Below you will find a more detailed description of what you should implement. From now on (i, j) will be the pixel in the i-th row and j-th column, and I_{ij} the value at that pixel.

Init. The version of the contour tracing algorithm we implement only finds the outermost contour of a given structure present in the image. To achieve this create an integer variable called LNBD and set it to 0. It will later be used to tell the algorithm to ignore a contour if a surrounding contour has already been found.

1. The input image is scanned in a row-wise manner until the bottom right corner is reached. Whenever you reach a new row, set the variable $LNBD$ to 0. Proceed to step 2, if the considered pixel at position (i, j) has a non-zero value. Otherwise, continue scanning.

2. If for the current pixel $I_{ij} = 1$ and for its left neighbor $I_{ij-1} = 0$ (i.e., we are at an outer border) and $LNBD \leq 0$ continue to step 3. Otherwise, go back to step 1 (i.e., continue the scan). In either case set $LNBD = I_{ij}$ before moving to the next step.

3. Next, the algorithm will discover and follow the contour the original pixel (i, j) belongs to. For this purpose, we create a new vector in which we will store the points belonging to the current contour. We also define 4 helper variables of the type `cv::Point` which we will call $(i_1, j_1), (i_2, j_2), (i_3, j_3)$ and (i_4, j_4), initialized with the original pixel position (i, j). Next let $(i_2, j_2) = (i, j-1)$ and examine the 4-neighborhood of (i, j) clockwise starting from (i_2, j_2) until a non-zero pixel is found. If so, store its position in (i_1, j_1). Afterwards set $(i_2, j_2) = (i_1, j_1)$ and $(i_3, j_3) = (i, j)$. If no such pixel is found write the value -2 to I_{ij}, push (i, j) to the contour vector, and go to step 5. Otherwise set $(i_2, j_2) = (i_1, j_1)$ and $(i_3, j_3) = (i, j)$ and go to step 4.

4. Now traverse the 4-neighborhood of (i_3, j_3) in counterclockwise order. Start from the neighbor that is next in order after (i_2, j_2). For example: if (i_2, j_2) were the neighbor to the right of (i_3, j_3), then you would start the search from the pixel above, i.e., $(i_3 - 1, j_3)$. If any non-zero pixels are found write the first one to (i_4, j_4). Now we do one of the following:

- if $(i_3, j_3 + 1)$ has the value 0 set $I_{i_3 j_3} = -2$ and add (i_3, j_3) to the current contour.
- else if $I_{i_3 j_3} = 1$ set $I_{i_3 j_3} = 2$ and also add (i_3, j_3) to the current contour.

Once we return to the starting point (i.e., $(i_4, j_4) = (i, j)$ and $(i_3, j_3) = (i_1, j_1)$) go to step 5. If this is not the case, set $(i_2, j_2) = (i_3, j_3)$ and $(i_3, j_3) = (i_4, j_4)$ before you continue to trace the contour by going back to the start of step 4.

5 Add the new contour to the provided vector `contours` and continue scanning the image from $(i, j+1)$ (go back to step 1).

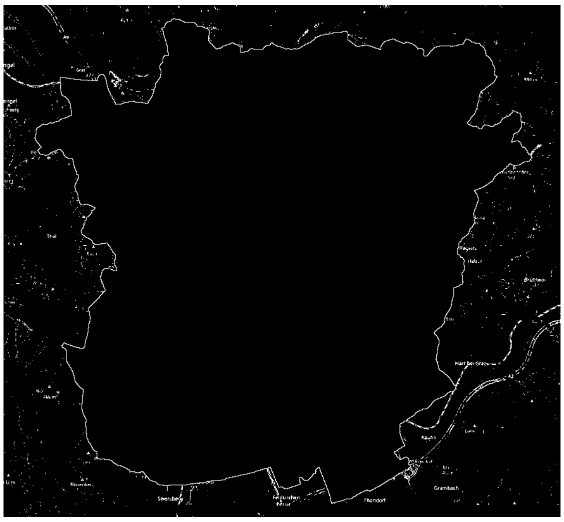


Figure 10: Expected output for `bonus_find_contours(...)`. Visual representation of detected contours for `graz.png`.

Forbidden Functions:

- `cv::findContours(...)`

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] Nobuyuki Otsu. A threshold selection method from gray-level histograms. IEEE Transactions on Systems, Man, and Cybernetics, 9(1):62–66, 1979.

[2] Satoshi Suzuki and KeiichiA be. Topological structural analysis of digitized binary images by border following. Computer Vision, Graphics, and Image Processing, 30(1):32–46, 1985.