Figure 1: Marked features (left) and the calculated homography with matched features (right).

Figure 2: Transformation of the rotated image to its rectified state (left-middle). The detected differences in the original image (right).

CV/task2 — Spot the Difference

The goal of this assignment is to implement a basic feature matching pipeline that can find differences in similar pictures. First, we have to detect feature points and calculate feature descriptors. Since we have to be able to process tilted and/or moved pictures the first and the second picture have to be rectified with a homography. After that, we compute and mark the differences (by circling them).

For the task of feature detection we use the Harris detector. As Harris is not scale invariant, we use the Laplacian detector for scale estimation (i. e. Harris-Laplacian detector). As bonus task, we implement the FAST algorithm for feature detection.

Feature Points represent the most interesting points in an image. In our assignment we have two images of the same scene and want to find feature point correspondences in the first and second image (see Fig. 1-left). To find such matching points we describe

each feature point with a **Feature Descriptor**. After the matching procedure we estimate a **Homography** (Fig. 1-right) from the best matches. This homography transforms one image to overlap the other one (Fig. 2-left). In the last step we compute the differences in these overlapped images (Fig. 2-right).

1 Introduction

The example is split into multiple subtasks, which build up on the previous one. The task is supposed to be solved in the following order:

- Harris-Laplace Feature Detection (5 Points)
  - Building the Scale Space Pyramid - `buildOctavesPyramid` (2 Points)
  - Harris Corner Measure - `calculateHarrisCornerMeasure` (1.0 Points)
  - Harris Non-Maxima Suppression - `nonMaximaSuppressionHarris` (0.5 Points)
  - Feature Detection - `detectFeatures` (1.5 Points)
- BinBoost Feature Descriptors (6 Points)
  - Gradient Orientation and Magnitude of the Grayscale Image - `computeGradientOrientationsAndMagnitudes` (0.5 Points)
  - Oriented Gradients of the Grayscale Image - `computeGradients` (2 Points)
  - Integral Images - `computeIntegrals` (0.5 Points)
  - Orientation of Feature-Points - `computeOrientation` (2 Points)
  - Patch Rectification - `rectifyPatch` (0.5 Points)
  - Weak Learners Response - `computeWLResponse` (0.5 Points)
- Image Matching (2 Points)
  - Feature Matching - `matchFeatures` (1 Point)
  - Calculate Homography - `calculateHomography` (0.5 Points)

  - Transformation of the Image - `transformImage` (0.5 Points)

- Compute Differences (3 Points)
  - Compute Difference Points - `computeDiffs` (1 Point)
  - Mark the Difference Points - `markDiffPoints` (2 Points)

- Bonus FAST feature descriptor (3 Points)

The task runs on OpenCV 2.4.9 [1]. You can make use of the available functions. Be aware of the different image-types (i. e. floating point vs unsigned) and parameters. Additional some helpful functions are cited after each section.

[1] http://opencv.org/

2 Harris-Laplace Feature Detection (5 Points)

For feature detection we use the Harris-Laplace [2] detector. This task consists of building a scale space pyramid (function `buildOctavesPyramid`), calculating the Harris corner measure (function `calculateHarrisCornerMeasure`), non-maximum suppression (function `nonMaximumSuppressionHarris`) and choosing the scale that maximizes the Difference of Gaussians (DoG) response (function `detectFeatures`).

The main idea is to combine the Harris corner measure and the Gaussian scale-space representation. Consequently, Harris corners can be detected regardless of the image or object scale. The scale space representation is an image pyramid consisting of several octaves. An octave is a differently sized version of the input image (i. e. factor two). Each octave consists of so-called layers, which are Gaussian filtered versions of the input image. Layers of higher scale have a larger blur, i. e. they are blurred with a Gaussian of larger standard deviation. Then, the Harris Corner measure is applied to these layers to find feature points.

2.1 Building the Scale Space Pyramid - `buildOctavesPyramid` (2 Points)

In this step, we build a scale space pyramid (see Fig. 3). The pyramid consists of several so-called "octaves". Octaves are differently resized versions of the image (i. e. factor two). Each octave consists of multiple layers (scales), which are filtered by Gaussian filters with different standard deviation parameters. For scale space detection we need the second derivative of these layers, i. e. Laplacian of Gaussians (LoG). However, as LoG is computationally expensive, we use a cheaper approximation. We compute the absolute difference between two successive layers, which is called DoG, which achieves similar accuracy compared to LoG in practice.

The Gaussian pyramid starts with images of the same size as the input image I. The pyramid has a depth of $numOctaves = 6$ octaves. The images in each octave o have a size of 0.5 the size of those in the previous octave. So images in last octave have a size of $2^{-numOctaves+1} = 2^{-5}$ of the original size. Each octave itself contains $numLayers + 3 = 7$ images B_l which are convolutions of the resized grayscale image with a Gaussian kernel. The standard deviation σ_l (scale) of the Gaussian kernels for all layers l except the first one(!) is defined as:

$$\sigma_l = \sigma_0 \cdot k^l, \tag{1}$$

where $\sigma_0 = 1$ and $k = 2^{\frac{1}{numLayers}}$. To speed up computation, we incrementally blur the images from the previous layer of the Gaussian image pyramid. To compute the correct incremental $\widehat{\sigma_l}$, recall that if we convolve two Gaussians with σ_1 and σ_2, the resulting

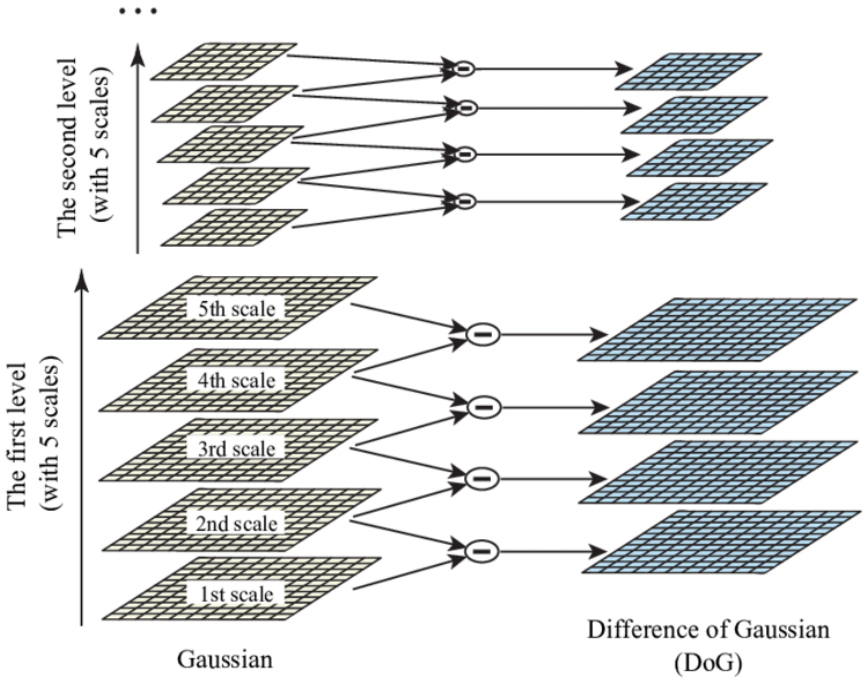

Figure 3: Schematic drawing of the Gaussian and the DoG pyramid

Gaussian kernel has a **variance** of $\sigma_{1+2}^2 = \sigma_1^2 + \sigma_2^2$. Therefore, $\widehat{\sigma_l} = \sqrt{\sigma_l^2 - \sigma_{l-1}^2}$.

The kernel size for a convolution with a Gaussian kernel (the Gaussian normal distribution is defined on whole $\mathbb{R}$), as a rule of thumb, is chosen as $3 \times \sigma$ in all directions from the center of the kernel:

$$\widehat{size_l} = \lceil \widehat{\sigma_l} \cdot 3 \rceil \cdot 2 + 1. \tag{2}$$

To compute the response B_l for layer l, we blur the previous layer B_{l-1} of the corresponding octave o with the Gaussian kernel $G(\cdot)$ by convolution, i. e. $B_l = B_{l-1} * G(\widehat{\sigma_l})$, $1 \leq l \leq numLayers + 2$. To compute the first layer ($l = 0$) of an octave, we resize the $numLayer$-th image of the previous octave (this layer should have had a σ of 2, as $\sigma_{numLayers} = 2$). To compute the first layer of the first octave, we blur the image with a Gaussian of σ_0, i. e. $B_0 = I * G(\sigma_0)$. With these blurred images, we can compute the DoG responses D_l by $D_l = |B_{l+1} - B_l|$.

Therefore, the DoG pyramid contains the same amount of octaves, but one layer less (see also Fig. 3). This is due to the fact that we need two neighboring layers of the Gaussian pyramid to form a layer of the DoG pyramid by computing the absolute difference between these layers.

The implementation of building the pyramids has to be done in the function `buildOctavesPyramid`. We go through all the octaves. First, we compute the first layer of the octave in the Gaussian pyramid (parameter `pyramid`) by either just resizing the $numLayer$-th layer from the previous octave or blurring the input image I with the specified Gaussian (`l=0`, `octave=0...numOctaves-1`). Next, we push the computed Gaussian layer to the `layers` vector. Then, we compute all remaining layers of Gaussians ($1 \leq l \leq numLayers + 2$). Also, we compute the absolute differences of the previous calculated layer and the new layer and push them to the `DOG_layers` vector. Finally, we push the vectors of Gaussian and DoG layers onto the pyramids.

Helpful OpenCV functions:

- `cv::GaussianBlur(...)`
- `cv::absdiff(...)`
- `cv::resize(...)` with interpolation method $INTER_AREA$

2.2 Harris Corner Measure - `calculateHarrisCornerMeasure` (1.0 Points)

The Harris corner measure (also Harris response) defines the "cornerness" of a point in an image. The neighborhood of a corner point changes strongly into two different directions (whereas edge points only require strong change in one direction). The Harris response correlates with the presence of such corners. To compute the response we use a so-called autocorrelation matrix. For each image location (x, y) the autocorrelation matrix is defined as

$$M(x,y) = \sum_{p,q} w(p,q) \cdot \begin{bmatrix} I_x^2(p,q) & I_x I_y(p,q) \\ I_x I_y(p,q) & I_y^2(p,q) \end{bmatrix}. \tag{3}$$

To compute M, we first need to compute the spatial derivatives I_x and I_y of the image I and multiply them with a scaling factor σ_D. To compute the derivatives we use a Sobel filter with kernel size 1. Therefore, $I_x = (I * [-1,0,1]) \cdot \sigma_D$ and $I_y = (I * [-1,0,1]^\top) \cdot \sigma_D$. With I_x and I_y we can simply compute I_x^2, I_y^2 and $I_x I_y$ by elementwise multiplication. The positions (p, q) describe a neighborhood around the point (x, y) and

$$w(p,q) = \frac{1}{2\pi\sigma_I^2} \cdot \exp - \frac{p^2 + q^2}{2\sigma_I^2} \tag{4}$$

is the weighting function. This weighting function is the Gaussian distribution. So, the multiplication of the matrix with the weighting function corresponds to convolving the

scaled multiplication of derivatives, i. e. I_x^2, I_y^2, $I_x I_y$, with a Gaussian kernel with standard deviation σ_I and kernel size calculated as in Equation (2).

The Harris corner measure at each image location can now be calculated as follows, using the matrix M:

$$R(x,y) = \det M - 0.04 \cdot \text{trace}^2(M), \tag{5}$$

where

$$\det M = M_{1,1}(x,y) \cdot M_{2,2}(x,y) - M_{1,2}(x,y) \cdot M_{2,1}(x,y) \tag{6}$$

$$\text{trace}(M) = M_{1,1}(x,y) + M_{2,2}(x,y) \tag{7}$$

The constant 0.04 is an empirically determined value.

The task is to compute the Harris corner measure at each image location using the described relations in the function `calculateHarrisCornerMeasure`. For calculating the derivatives of the image, we use the OpenCV function `Sobel(...)` with border type `BORDER_REPLICATE`, kernel size 1 and destination data type CV_32F. For smoothing the image with a Gaussian kernel we use the function `GaussianBlur(...)` with $BORDER_REPLICATE$ as well.

2.3 Harris Non-Maxima Suppression - `nonMaximaSuppressionHarris` (0.5 Points)

The Harris response has typically a large number of near duplicate detections. To get stable detection and matching results, we want to remove these near duplicates. To this end, we apply non-maxima suppression on the Harris cornerness measure. First, only "good" corner responses, i. e. responses above a given threshold t, are considered as feature points. t is given as:

$$t = t_c \cdot \max(R) \tag{8}$$

where $t_c = 0.01$ is a threshold, and $\max(R)$ is the maximum value in the response matrix R (`cornerness`). We set all values lower than this threshold t to zero. The resulting thresholded matrix is called R_t.

To perform non-maxima suppression, we dilate the matrix R_t with a 3×3 rectangular structure element. Dilating a grayscale image results sets each pixel to the maximum value of its neighborhood, as defined by the structure element. The dilated marix is called R_d.

After this dilation we can check, if the value of a pixel in the thresholded matrix R_t equals the value in the dilated matrix R_d. Only these points are local maxima and are therefore considered as feature points. We set these local maxima to > 0 in the `local_maxima` matrix and all other values to 0.

Helpful OpenCV functions:

- `cv::minMaxLoc(...)`
- `cv::threshold(...)`
- `cv::dilate(...)`
- `cv::compare(...)`
- `cv::bitwise_and(...)`

2.4 Feature Detection - `detectFeatures` (1.5 Points)

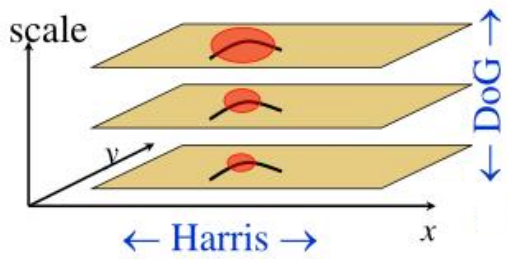

Figure 4: Overview of our Harris-Laplacian detector.

In this step we create a scale invariant feature detector (i. e. Harris Laplacian) by combining the Harris corner detection with the DoG pyramid to compute features on multiple scales of the image (see Fig. 4). To this end, we rely on the `buildOctavesPyramid`, `calculateHarrisCornerMeasure` and `nonMaximaSuppressionHarris` functions. The main idea is to find Harris corners on multiple layers of the Gaussian scale space pyramid. To achieve scale invariance, we also check if the corner is a local maxima in the DoG pyramid by checking if the DoG response of the corner of the current scale is larger than the response of the next and previous scale.

More specifically, we loop through all our octaves o and for each octave we loop through all layers l which have a previous and follower layer in the DoG pyramid (i. e. $1 \leq l \leq numLayers$). For the current octave o and layer l we have to find the layer on which we want to find Harris corners. By definition this layer is two scales below the current layer l. To get this layer, we distinguish three cases:

- If $l \geq 2$, we can simply get this layer by accessing the Gaussian pyramid `pyramid` in the current octave o and layer $l-2$.

- If $l = 1$ and $o \geq 1$, we can compute the layer by resizing a layer from the previous octave, i. e. the `numLayers` $-$ 1-th layer of the $o-1$-th octave. We resize this layer by a factor of 0.5 to match the size of the current octave.

- If $l = 1$ and $o = 0$, we skip this case.

On this layer we compute the cornerness measure (`calculateHarrisCornerMeasure`) and perform spatial non-maxima suppression (`nonMaximaSuppressionHarris`). For corner detection, we set $\sigma_i = 2^{\frac{l}{numLayers}}$ and $\sigma_d = \sigma_i \cdot 0.7$.

Finally, we have to check which of our corners are also local maxima in the scale space. To this end, we use the previous DoG layer $l-1$, the current DoG layer l and the next DoG layer $l+1$ of the current octave o. We loop through all non-maxima (x, y) of our Harris cornerness matrix and check if the corner on this position is a local maxima in the DoG pyramid, i. e. $D_l(x,y) > D_{l-1}(x,y) \wedge D_l(x,y) > D_{l+1}(x,y)$ and if the DoG response is larger than a threshold, i. e. $D_l(x,y) \geq$`DOG_thresh`, where $D_l(x,y)$ denotes the DoG response on layer l on position (x, y).

To find the coordinates of the keypoint, we translate the coordinates of the current pixel in the cornerness matrix into coordinates in the original sized image (each octave contains images half the size of the previous octave!). This can be done by

$$x_i = x \cdot 2^o, y_i = y \cdot 2^o, \tag{9}$$

where x_i, y_i are the coordinates in the original image, x, y are the coordinate in the smaller version of the image in the current octave with number o. The size s of the potential keypoint at this position can be calculated with

$$s = 3 \cdot 2^o \cdot \sigma_i \cdot 2. \tag{10}$$

Further, we set the angle of the keypoint to 0, the response to the Harris cornerness response and the octave to $o \cdot (numLayers + 1) + l$.

The current point is defined as a keypoint of the image, if these three conditions are fulfilled:

- the whole keypoint ($\pm \frac{s}{2}$ in all directions) is within the image

- the value in the current DoG layer exceeds the values in the previous and following layer (DoG non-max suppr.)

- the DoG response exceeds the DoG threshold `DOG_thresh` defined as a constant at the beginning of the function.

For all keypoints, we create a `cv::Keypoint` object with the corresponding parameters (i. e. x_i, y_i, s, the cornerness measure of the point and the current octave/layer-idx $o \cdot (numLayers + 1) + l$). Further, we set the orientation of the keypoint to 0. We add all keypoints to the features vector.

3 BinBoost Feature Descriptors (6 Points)

In this section our goal is to describe the previously found keypoints in a way that is invariant regarding scaling, rotation, distortion and other simple operations. We can find a function that calculates such a description for a keypoint with a Machine Learning algorithm called Boosting, which learns multiple Weak Learners that represent this function. This is the basic idea of BinBoost feature descriptors [4]. For a full description of how BinBoost works, check out the paper "Learning Image Descriptors with Boosting" by Tomasz Trzcinski, Mario Christoudias and Vincent Lepetit from 2015 [4].

The file $binboost_128.bin$ contains the learned parameters of these weak learners. These parameters are loaded by the function $readWLsBinForBinboost$ (which you do not have to change).

We divide this part of the exercise into the following functions:

- `computeGradientOrientationsAndMagnitudes`: 0.5 Points
- `computeGradients`: 2 Points
- `computeIntegrals`: 0.5 Points
- `computeOrientation`: 2 Points
- `rectifyPatch`: 0.5 Points
- `computeWLResponse`: 0.5 Points

3.1 Gradient Orientation and Magnitude of the Grayscale Image - `computeGradientOrientationsAndMagnitudes` (0.5 Points)

The first step of our feature description pipeline is to compute the Gradient Magnitudes and Orientations of a grayscale image. We can interpret a gradient of a pixel as a vector which points into the direction of steepest ascent (see Fig. 5).

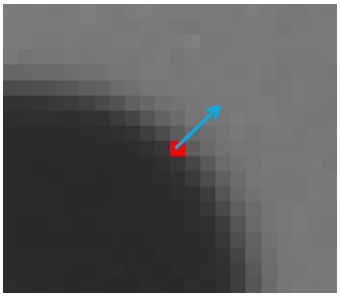

Figure 5: Gradients of a pixel are vectors pointing to the direction of steepest ascent.

We can compute the x and y components of these vectors by convolution with `Sobel` filters in x and y direction:

$$S_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}, \quad S_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} \tag{11}$$

I_x and I_y are computed by convolving one of the kernels with the image I:

$$I_x = I * S_x, \qquad I_y = I * S_y\,. \tag{12}$$

To compute this convolution, we actually use the OpenCV function `cv::Sobel(...)`. Further, we use CV_32FC1 as destination data type to compute the gradients as floating points.

The gradient vectors on each position of an image are now represented by the I_x and I_y matrices. To generate the gradient magnitude and gradient orientation matrices we need the magnitude and the angles for each pixel.

The magnitude on position x, y is defined as

$$M(x,y) = \sqrt{I_x(x,y)^2 + I_y(x,y)^2}. \tag{13}$$

The angle θ on position x, y is defined as:

$$\theta(x,y) = \text{atan2}(I_y(x,y), I_x(x,y)). \tag{14}$$

Further, the angle has to be in the range $(0,\ 2\cdot\pi)$. Therefore, if the angle is below 0 we shift it by $2\cdot\pi$ to get a valid result.

Helpful OpenCV functions

- `std::atan2(...)`
- `std::sqrt(...)`
- `cv::Sobel(...)`

3.2 Oriented Gradients of the Grayscale Image - `computeGradients` (2 Points)

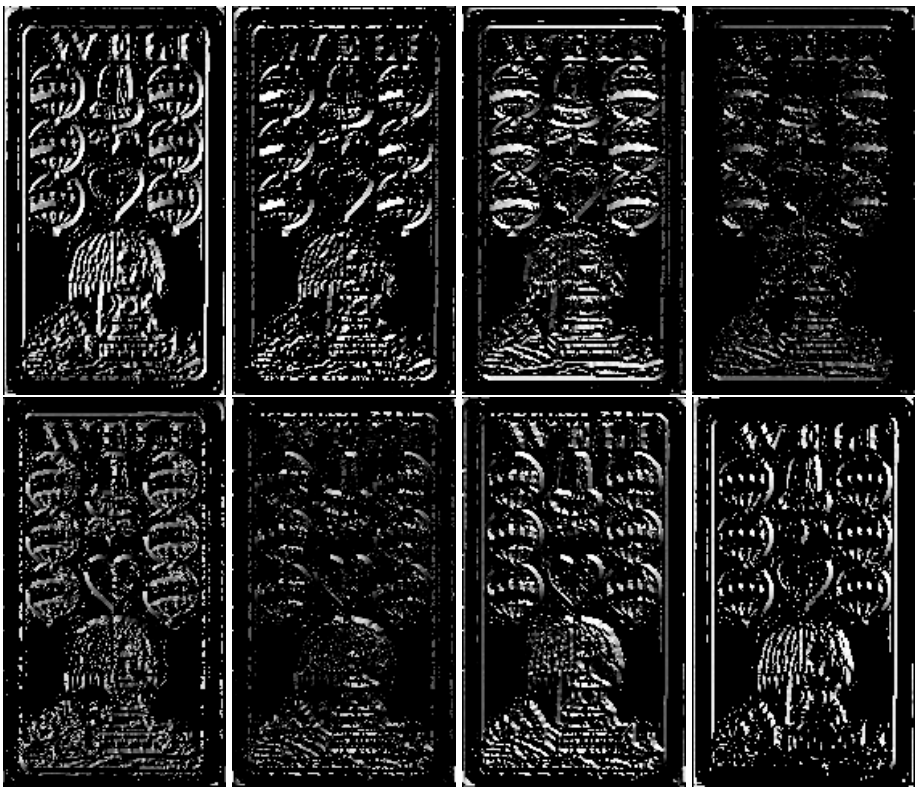

Figure 6: Gradients in all 8 directions.

The main objective of this function is to quantize the gradient into 8 gradient orientation images (see Figure 6). To this end, we rely on the previously implemented function `computeGradientOrientationsAndMagnitudes(...)`. We iterate through all pixels (x, y) of the image and consider only pixels with a high enough gradient magnitude for further processing. Specifically, if the magnitude for the pixel exceeds the threshold `magnitudeThreshold`, we need to compute the assignment to the quantized orientation gradient images. Otherwise, we just set all orientation gradient images on this position to 0.

The quantized orientation bin boundaries are defined as $k \cdot \frac{2 \cdot \pi}{orientQuant}$ for $0 < k < orientQuant$. In our case $orientQuant$ is 8, i.e. there are 8 orientation bins. Therefore, to compute the gradient bin for the current pixel (x, y), we divide the orientation (in radiants) by the bin size:

$$index(x, y) = \left\lfloor \frac{\theta(x, y) \cdot orientQuant}{2 \cdot \pi} \right\rfloor \tag{15}$$

Note that $index(x, y)$ is a integer variable, i. e. we use a cast to int to convert from floating points to integer at the end of this operation. Further, we ensure that $index(x, y)$ is between 0 and $orientQuant$. Specifically, if $index(x, y) = orientQuant$ we set it to 0.

Rather than computing a hard assignment to the gradient bin, we do a soft assignment to several gradient bins. Every gradient affects 2 adjacent gradient bins b in each direction, i. e. $b \in \{index(x, y) - 2, \ldots, index(x, y) + 2\}$. Therefore, one gradient affects 5 bins in the gradients matrix. To avoid negative indices for b, we normalize it to be between 0 and $orientQuant$ by $\widehat{b} = \mod(b + index(x, y) + orientQuant, orientQuant)$.

For every of these 5 bins we calculate a weight w, which is the cosine of the angle between the bin center and the angle $\theta(x, y)$. The angle of the bin center of bin $\widehat{b}$ is defined as $bc = (\widehat{b} + 0.5) \cdot \frac{2 \cdot \pi}{orientQuant}$. We also avoid negative values of w by defining it as

$$w = \max(0, \cos(\theta - bc)). \tag{16}$$

Finally, we assign $\mathrm{round}(w \cdot 255)$ to the `gradients` matrix on the current position (x, y) and the current neighbor bin $\widehat{b}$.

3.3 Integral Images - `computeIntegrals` (0.5 Points)

Integral images are useful for calculating sums of pixels within rectangular regions of an image. Such sums can then be calculated in constant time by 4 add operations. We need this for calculating our feature descriptors. The function `computeImages` gets a vector of gradient images of the grayscale image, with $orientQuant$ gradients in different directions, as input. We want to compute the integral images of these gradient images. The definition of an integral image is:

$$I(x, y) = \sum_{i=0}^{i<x} \sum_{j=0}^{j<y} H(i, j), \tag{17}$$

where I denotes the integral image, H denotes the grayscale image and (x, y) an integral image location. As a consequence of the above equation, the first row and column of I are equal 0. The integral image has the size $(rows + 1) \times (cols + 1)$, where $rows$ is the number of rows and $cols$ the number of columns of H. Naïvely computing the integral image from the above equation is computationally expensive. However, we can see that the following

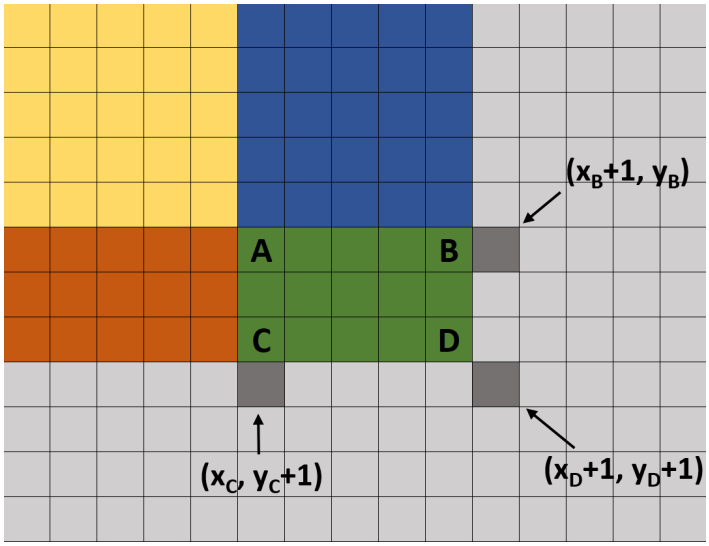

Figure 7: Illustration of calculating the sum of a region using the integral image

holds:

$$I(x,y) = I(x,y-1) + \sum_{i=0}^{i<x} H(i,y-1). \tag{18}$$

We can further reduce computational load by computing the second term iteratively. Let $R(x,y)$ be the cumulative sum of the pixels in the row y:

$$R(x,y) = \begin{cases} R(x-1,y) + H(x-1,y-1) & if\, x > 0, \\ 0 & otherwise. \end{cases} \tag{19}$$

Then, $I(x,y) = I(x,y-1) + R(x,y)$ and therefore the integral image can be calculated by going through all the pixels only once.

Using the integral image I of the image H, the sum S of the pixels from point (x_A, y_A) to point (x_D, y_D) in the image H can be computed very fast independently of the rectangle size:

$$S = I(x_A, y_A) + I(x_D+1, y_D+1) - I(x_D+1, y_A) - I(x_A, y_D+1). \tag{20}$$

In the function `computeIntegrals` we iterate through all gradient images in the `gradients` vector. We compute the integral image of the i-th gradient image within the `Mat integrals[`i`]`, for $0 \leq i < orientQuant$. Furthermore, the last image in `integrals` (number `orientQuant`) is the sum of all prior integral images.

`Hint`: The data types for our integral images and row-sums are integers. The gradient images H are uint8.

3.4 Orientation of Feature-Points - `computeOrientation` (2 Points)

In this step, to make our features rotation invariant, we estimate the orientation of a Keypoint as the dominant gradient orientation of that point. To this end, we use a simplified version of the SIFT [1] orientation estimation (see Fig. 8). We compute the gradient magnitudes and orientations for a region around the keypoint. Then, we create a histogram of gradient orientations. Each pixel votes with its gradient magnitude into a bin of the histogram. As pixels in the center should have stronger votes compared to pixel on the boundary, we weight the gradient magnitude by a Gaussian window. The keypoint orientation is the orientation with most votes in the histogram.

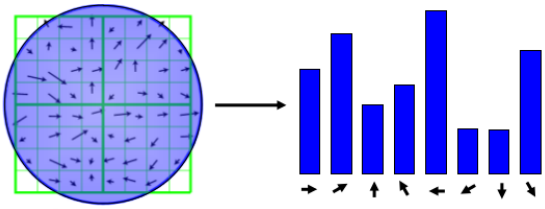

Figure 8: Orientation estimation by computing a histogram of gradient orientations.

The basis of the orientation estimation are the gradient magnitudes and orientations, which we get as input to our function. We iterate through all keypoints kp and transform the keypoint coordinates into the coordinates of the current keypoint octave kp_o and layer kp_l. The octave and layer can be calculated from the $octave$ field of the keypoint object by division and modulo of $numLayers + 1$ (this is already done for you). The coordinates have then to be scaled by $\frac{1}{2^{kp_o}}$ (this is also already done for you).

Next, we consider a local neighborhood around the keypoint position (x, y) of $patchsize \times patchsize$, i. e. $(x - patchsize/2, y - patchsize/2)$ to $(x + patchsize/2, y + patchsize/2)$ (we also make sure that we are not out of bounds for the current image!). In this rectangle, we want to estimate the dominant gradient orientation (see Fig. 8). For every pixel (i, j) in this neighborhood we now compute the histogram entry. The histogram bin b is defined as

$$b = \text{round}\left(\theta_{kp_o,kp_l}(i,j) \cdot \frac{180}{\pi} \cdot \frac{nAngleBins}{360}\right), \tag{21}$$

where $\theta_{kp_o,kp_l}(i,j)$ denotes the i,j-th entry of the orientation image with octave kp_o and kp_l. Further, we make sure that b is not out of bounds. Specifically, if $b < 0$ we shift it by $nAngleBins$. If $b \geq nAngleBins$ we subtract it by $nAngleBins$.

We now cast a vote into the histogram H by

$$H[b] := H[b] + M_{kp_o,kp_l}(i,j) \cdot s(i,j), \tag{22}$$

where M_{kp_o,kp_l} denotes the gradient magnitude image of octave kp_o and layer kp_l. $s(i,j)$

is a Gaussian scaling factor which prefers points closer to the keypoint center x, y:

$$s(i,j) = e^{-\frac{(i-x)^2+(j-y)^2}{2\cdot\sigma^2}}, \tag{23}$$

where $\sigma = 1.8$.

Figure 9: By estimating the dominant gradient orientation we can normalize rotated patches.

After iterating through the neighborhood, we have to assign the dominant gradient orientation to our keypoint kp. To this end, we find the maximum bin b_{max} of the histogram H and set the angle of the keypoint to $(b_{max} + 0.5) \cdot \frac{360}{nAngleBins}$.

`Hint`: If two histogram bins are equal take the first one.

3.5 Patch Rectification - `rectifyPatch` (0.5 Points)

In this section, we implement a simple helper function which rectifies a patch using the previously computed keypoint location (x, y), scale and angle θ. We implement this function in `rectifyPatch`.

To rectify a patch around the keypoint kp, we need to transform the image with an affine transform M. More specifically, we define the transform as

$$M = \begin{bmatrix} s\cdot\cos(\theta) & -s\cdot\sin(\theta) & (-s\cdot\cos(\theta)+s\cdot\sin(\theta))\cdot\frac{patchSize}{2}+x \\ s\cdot\sin(\theta) & s\cdot\cos(\theta) & (-s\cdot\sin(\theta)-s\cdot\cos(\theta))\cdot\frac{patchSize}{2}+y \end{bmatrix}, \tag{24}$$

where $s = \frac{kp.size}{patchSize}$ denotes the scaling factor of the transform. We set s to 1 if $kp.size \leq 0$. θ denotes the angle of the keypoint. If $\theta < 0$, we set $\theta = 0$.

The output of this function is a rectified patch of size $patchSize \times patchSize$. `Hint`: Angles are stored in degrees. std::sin, std::cos need the angle in radiants.

Helpful OpenCV functions

- `warpAffine(...)`, make use of the specified flags in the function.

3.6 Weak Learners Response - `computeWLResponse` (0.5 Points)

Figure 10: BinBoost descriptor consists of rectangular sums on oriented gradient images. We illustrate here 3 features (different rectangles) and 3 out of 8 gradient images.

The BinBoost [4] descriptor consists of rectangular sums on our previously computed oriented gradient images (see Fig. 10). The descriptor learns several so-called "weak learners" on a dataset of similar and dissimilar patches. Every "weak learner" defines a rectangular feature on a single gradient image. In Fig. 10 we illustrate three such weak learners (colored red, green, blue). In this function we implement feature computation for a single weak learner, i. e. the sum of one rectangle.

Formally, we define G as the vector of patches with size $(rows \times cols)$ of the gradients in $W = orientQuant$ different directions around a previously determined feature point. d is the direction of the current weak learner. Further, let x_{min}, x_{max}, y_{min}, y_{max} and threshold t be properties of the weak learner. We can compute the the weak learner response R at this feature by

$$R = \frac{\sum_{i=x_{min}}^{i<x_{max}} \sum_{j=y_{min}}^{j<y_{max}} G_d(i,j)}{\sum_{k=0}^{k<W} \sum_{i=x_{min}}^{i<x_{max}} \sum_{j=y_{min}}^{j<y_{max}} G_k(i,j)} - t. \tag{25}$$

This equation is rather complex and expensive to compute in this form. Fortunately, we have the integral images of these gradient images. By using Equation (20) and the integrals of the gradients I (where the $orientQuant$-th element in I is the sum of all previous integral images, as described in Section 3.3), we can simplify the above formula:

$$R = \frac{I_d(x_{min}, y_{min}) + I_d(x_{max}+1, y_{max}+1) - I_d(x_{max}+1, y_{min}) - I_d(x_{min}, y_{max}+1)}{I_W(x_{min}, y_{min}) + I_W(x_{max}+1, y_{max}+1) - I_W(x_{max}+1, y_{min}) - I_W(x_{min}, y_{max}+1)} - t \tag{26}$$

In the function $computeWLResponse$ we are given the weak learner WL, the number of different directions $orientQuant$ and a vector of the integrals of patches in the gradient images (including this last "sum of all" layer). Using above equation we can compute the response of the weak learner. We avoid division by zero. If the denominator is zero, we return zero. Otherwise, we return the result of the equation. The properties of the weak learner WL are called $WL.x_min$, $WL.x_max$, $WL.y_min$, $WL.y_max$ and $WL.thresh$.

4 Image Matching (2 Points)

In this section we match the previously extracted keypoint descriptors from the first image with the ones from the second image and estimate a homography transformation with these matches.

4.1 Feature Matching - `matchFeatures` (1 Point)

In this step we compute matches between our feature descriptors. This part of the task is done in the function `matchFeatures(...)`. A match consists of two feature points. One of the source image and one of the target image. The matching is done with a DescriptorMatcher (BFMatcher) provided by OpenCV. We use the `knnMatch` method to obtain the $k = 2$ best matches for each source descriptor.

The matcher returns a list of matches and our task is to distinguish between good and bad matches. In our case the number of matches is $k = 2$, which means that the best match and second best match is returned for every feature point of the source image. To make the result more robust we check every match for a sufficiently big distance between the best match and the second best match. So we add only matches with

$$\text{distance}(\text{query}, \text{match}_0) \leq \texttt{match_distance_factor} \cdot \text{distance}(\text{query}, \text{match}_1) \tag{27}$$

to the output parameter (`matches`).

To identify the matches in the images we also add their source and target point to a list for the source and target image. This means that (for every match) we have to add the `queryIdx` point in the `source_features` vector to the `good_matches_source` vector and `trainIdx` point in the `target_features` vector to the `good_matches_target` vector.

The result of this step is a large number of matches (see Fig. 11).

Figure 11: Shows the good matches between two similar pictures using the Descriptor-Matcher of OpenCV. Every match is illustrated as a colored line between its feature points.

Helpful OpenCV functions:

- `cv::DescriptorMatcher::knnMatch(...)`

4.2 Calculate Homography - `calculateHomography` (0.5 Points)

The calculation of the homography between the two images is done with the matches from the previous task in the function `calculateHomography(...)`. For that we use the `cv::findHomography`-function provided by OpenCV. We use the `CV_RANSAC` method with a maximum allowed reprojection error of 3.

Helpful OpenCV functions:

- `cv::findHomography(...)`

4.3 Transformation of the Image - `transformImage` (0.5 Points)

This task is done in the function `transformImage(...)`. As input we get the right image, the left image and a homography, which transforms the left image to the right image. The main objective of this function is to warp the right image into the left with the OpenCV

function `warpPerspective(...)` (see Fig. 12). The resulting image should have the same size as the left image. We use linear interpolation (`INTER_LINEAR`).

Figure 12: Comparison between the resized image (left) and the source image (right).

Helpful OpenCV functions:

- `cv::warpPerspective(...)`

5 Compute Differences (3 Points)

In this section we compute the differences of the two rectified images.

5.1 Compute Difference Points - `computeDiffs` (1 Point)

In this step we implement the function `computeDiffs(...)`, which computes the difference between the transformed and resized image and the source image (Fig. 12).

The first step is to pre-process both images to remove noise due to resizing and transformation artifacts. Therefore, we blur both, our left and right image, with a Gaussian filter of σ. We set the kernel size of the Gaussian to $\lceil 3 \cdot \sigma \rceil \cdot 2 + 1$.

In the next step, we compute the absolute differences of these blurred images B_l,B_r:

$$D = |B_l - B_r|, \tag{28}$$

where D corresponds to the `diffImg` parameter. To get the "average" difference of a pixel, for the sake of simplicity, we simply convert this image to a grayscale image. From this grayscale difference image D_g we compute a thresholded image D_t with the `threshold` using the `cv::threshold(...)` function with flag THRESH_BINARY. Therefore, $D_t = D_g >$ threshold . The result is a binary difference image in which each connected component represents a difference (parameter `img_threshold`).

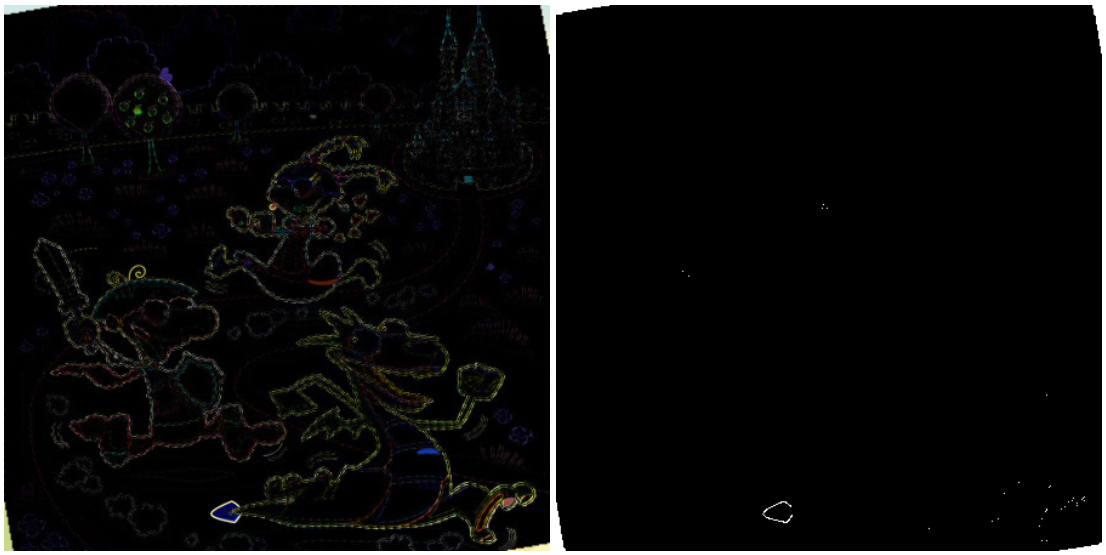

Figure 13: Absolute difference image (left). Thresholded image (right).

To get a description of the "white" pixels in our thresholded image without going through all of them we use a Canny edge detector (`cv::Canny(...)`) and the `findContours(...)`-function. The resulting list of contours should be saved in the contours parameter. We use a `lowThreshold` of 0 and a `highThreshold` of 255 and a kernel size of 3 for the Canny edge detector. Further, we call `findContours` with the detected edges and provide the `RETR_TREE` as mode flag and `CHAIN_APPROX_SIMPLE` as method flag.

Helpful OpenCV functions:

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

5.2 Mark the Difference Points - `markDiffPoints` (2 Points)

In the final step we need to mark the differences between the images as rectangles and circles. In the function `markDiffPoints(...)` we get the previously computed contours list. Every contour represents a difference between the source and the target image. For each contour, we want to create a circle and a bounding rectangle representing the contour.

To create a circle, we need to compute the center of the contour. We define the center of the circle as average x- and y-coordinate of the contour. Formally, the center $c_i = (x_i, y_i)$ of the i-th contour is defined as $x_i = \frac{1}{|contour_i|} \sum_{pt \in contour_i} pt_x$ and $y_i = \frac{1}{|contour_i|} \sum_{pt \in contour_i} pt_y$, where $|contour_i|$ denotes the number of elements of the i-th contour and pt_x/pt_y the x/y-coordinate of the point pt of $contour_i$. We further define the diameter as maximum of the width and height of the bounding rectangle of $contour_i$ (`cv::boundingRect(...)`). As we see in our output image, we detect a large number of difference-points (Fig. 14)

As we detect a large number of near duplicate difference points, we also create bounding rectangles r for each contour, if the corresponding circle is within our size constraints, i. e. $diameter_i \leq maxDiameter$. In order to remove near duplicates we perform a form of non-maxima suppression by merging overlapping rectangles. Specifically, we define the bounding rectangle r_i for contour i as minimum enclosing bounding rectangle of contour i (as obtained by `cv::boundingRect(...)`) padded by 10 pixels.

To eliminate multiple near duplicate bounding rectangles, the bounding rectangle r_i should be compared with all previously computed rectangles r_j in the rectangle-list `diffRectangles` if r_i overlaps with r_j. To check if the rectangles overlap, we provide a helper function `doRectOverlap(...)`. If they overlap with each other, we merge both rectangles and repeat the process. Specifically, we update bounding rectangle r_i to a new bounding rectangle representing the old and the new rectangle:

$$x = \min(r_i.x, r_j.x) \tag{29}$$

$$y = \min(r_i.y, r_j.y) \tag{30}$$

$$width = \max(r_i.x + r_i.width, r_j.x + r_j.width) - x \tag{31}$$

$$height = \max(r_i.y + r_i.height, r_j.y + r_j.height) - y. \tag{32}$$

We further remove the old bounding box r_j from our list `diffRectangles` and repeat this process for the now enlarged bounding box r_i. Finally, if no overlaps are found, we add r_i to the list `diffRectangles`.

Figure 14: Detected differences as circles (without near duplicate detection).

We visualize the result of this step in Fig. 15.

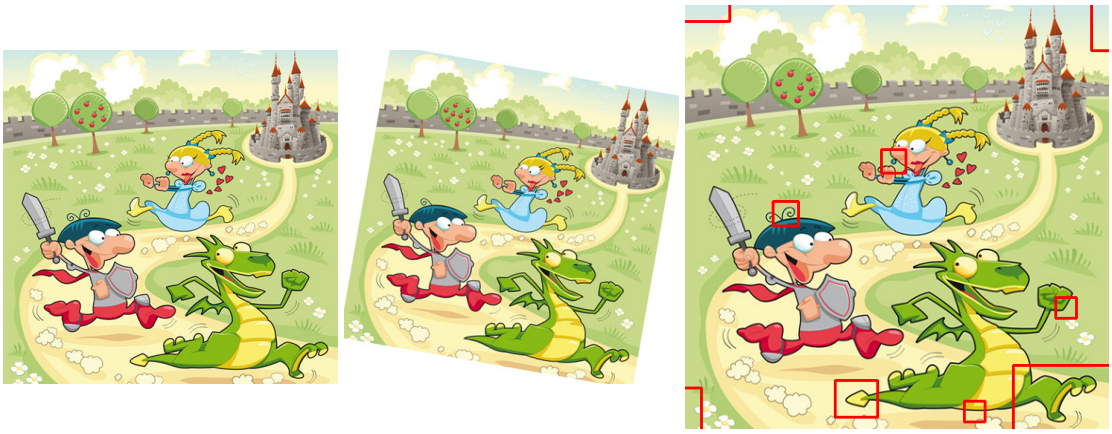

Figure 15: Detected differences with their bounding rectangles.

Helpful OpenCV functions:

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

6 Bonus FAST feature descriptor (3 Points)

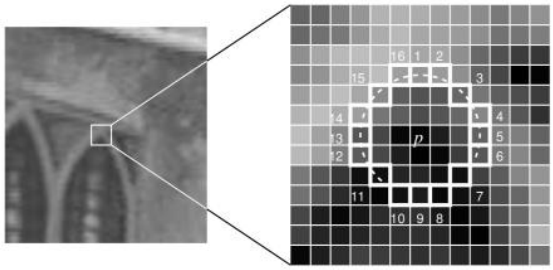

Figure 16: Zoom in to a corner with all pixels marked that are used by FAST to find pixels that are part of the corner [3].

This task has to be implemented in the function `Fast (...)`. The idea behind the FAST features (Features from Accelerated Segment Test) [3] is to find a corner by looking at the pixels that are nearby. The FAST algorithm checks pixels that are lying on a circle and checks if they have a higher or lower intensity than the center pixel. In our case we will use a circle with the radius $r = 3$. All points that are on that circle are given in the parameter (`circle`).

To compute the FAST detections, we consider all pixels (x, y) for which the Harris cornerness response is bigger than `harrisThreshold`, which are at least 9 pixels away from the border of the image. If this is the case, we can test the point (x, y) if it is a FAST feature point. To this end, we compute a lower a t_- and an upper threshold t_+

$$\begin{aligned} t_+ &= I(x,y) + t \\ t_- &= I(x,y) - t \end{aligned} \tag{33}$$

t is given by the parameter `thresh`.

Next, we check for every point of the circle if N consecutive points are higher or lower than the threshold calculated before. If that is the case we found a feature point. Hint: To avoid errors due to boundary conditions at the start/end of the circle we can go through the circle twice. At the same time (ı.e. in the same loop) we calculate the sum of absolute differences (SAD) between the center and the current pixel on the circle by using the formula provided. The SAD is needed for the non-maxima suppression.

$$SAD(x,y) = \sum_{(x_c,y_c)\in C} |I(x,y) - I(x_c,y_c)| \tag{34}$$

When all feature points are found, we perform a non-maxima suppression. This is done to

Figure 17: Example of non-maxima suppression[2].

avoid having more than one feature descriptor for a single feature. To this end, we iterate every previously detected FAST feature and check for every other feature point that has an Euclidean distance ≤ 9 if its SAD is higher than the current feature point. If this is the case, the current point is no local maxima and can therefore be removed from the list of keypoints.

[2] https://github.com/tejaslodaya/car-detection-yolo/blob/master/README.md

7 Parameters of the testcases

The following parameters define the behavior of the program. You find examples in the given .json files.

- Path of the images: `folder`
- Name of the testcase: `name`
- Name of the left image: `input_left`
- Name of the right image: `input_right`
- Acceptable distance factor (used as factor * distance): `match_distance_factor`
- Name of the binboost file: `wl_bin_name`
- Threshold: `diff_threshold`
- Testcase is only suitable for bonus implementation: `bonus_only: 1`
- There are no differences in this testcase: `no_diff: 1`

8 Framework

The following functionalities are provided by the given framework and to not have to be implemented:

- Reading in the config (JSON) file
- Reading in the given images
- Calling the functions in the right order
- Creating the output images

9 Submission

Your tasks consist of multiple sections which build up together, but will be graded independently. This is to ensure objective grading and to grand points for solutions that are incomplete.

We want to point out that all tasks have to be solved independently. If you share you code in any form (voluntarily or by mistake) the relevant task will be graded with 0 points for everyone involved, regardless of who wrote the code in the first place. It is also not allowed to use code from the internet, books or other sources. A plagiarism check is conducted both automatically and manually.

Submission of the assignment and the organization of the interview slots is managed by a web portal. Submission in any other form (i. e. email) will not be accepted. There is a description of how to hand in in the Teach Center.

The test system tests your submission automatically. There is a timeout after 7 minutes, so make sure your program is faster than that.

Since the programs are tested automatically, all parameters have to be passed by the according config files just as stated in the tasks. Especially waiting for parameters to be passed by hand is NOT allowed because it will cause a timeout. If you change something in the configuration systems that makes the execution of the test data impossible, the task will be graded with 0 points. The config data is given to you as JSON-files, to use them you can use `rapidjson`. The framework explains how to use them.

Every config file contains at least one testcase and its configuration.

It is up to you to implement your own extensions, but please make sure they are deactivated in the code you submit, so we can check your output with our references.

The framework you get is exactly the same the reference solution is based on. The only difference is that some parts are missing so you can implement them. You do not have to use the framework but it will save you a lot of time and work if do so.

10 Test images

The images used in the testcases are images from `www.puzzlesandriddles.com` and `www.tinyurl.com/y646hu99`.

References

[1] David G Lowe. Distinctive Image Features from Scale-Invariant Keypoints. International Journal of Computer Vision, 60(2):91–110, 2004.

[2] Krystian Mikolajczyk and Cordelia Schmid. Scale & affine invariant interest point detectors. International Journal of Computer Vision, 60(1):63–86, 2004.

[3] Edward Rosten and Tom Drummond. Machine learning for high-speed corner detection. In European Conference on Computer Vision, 2006.

[4] Tomasz Trzcinski, Mario Christoudias, and Vincent Lepetit. Learning image descriptors with boosting. IEEE Transactions on Pattern Analysis and Machine Intelligence, 37(3):597–610, 2015.