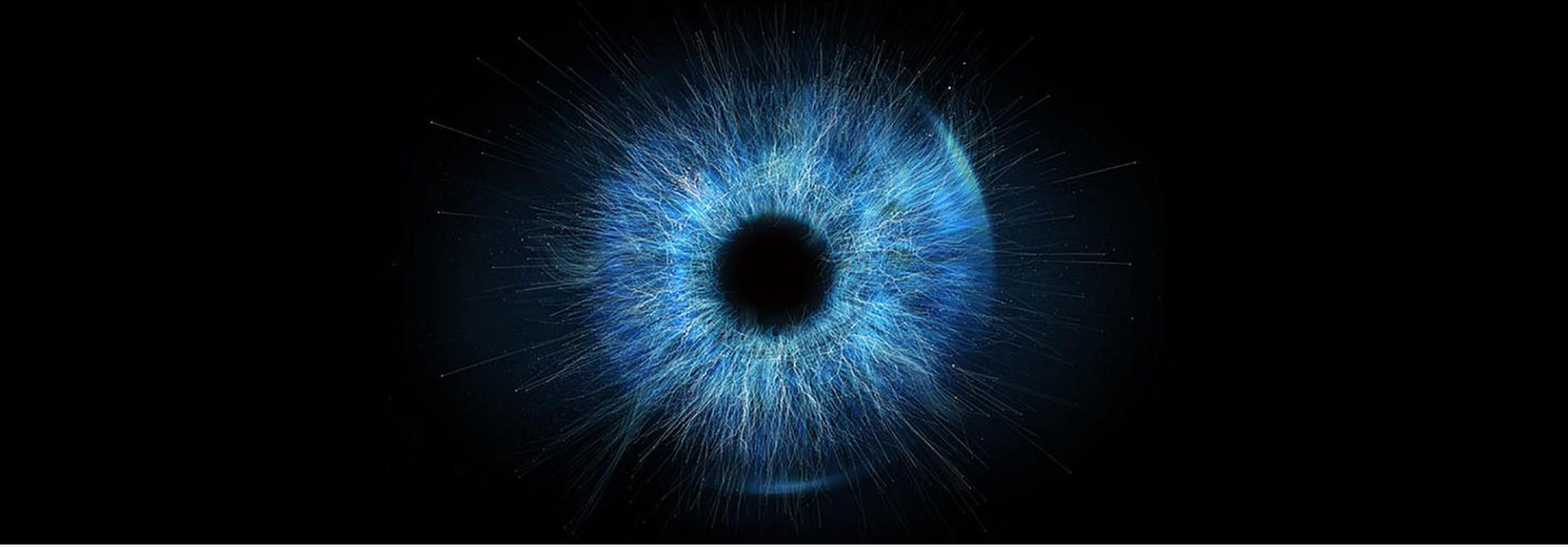


Exercise 8

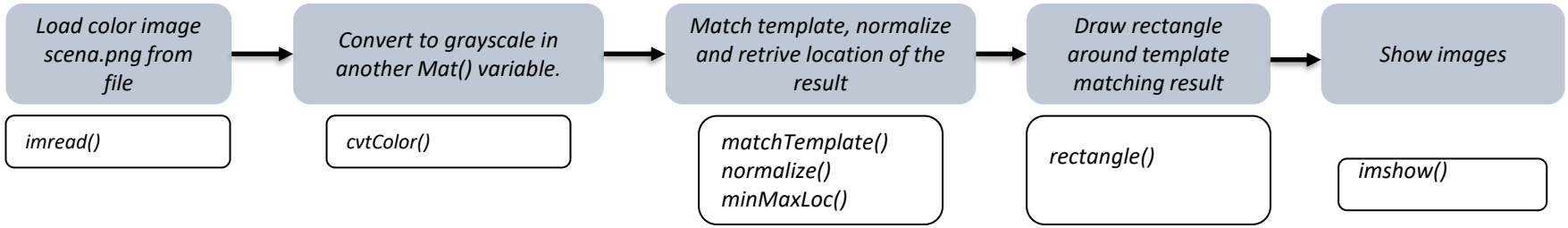
Fundamentals of machine vision algorithms

dr.sc. Filip Šuligoj
fsuligoj@fsb.hr

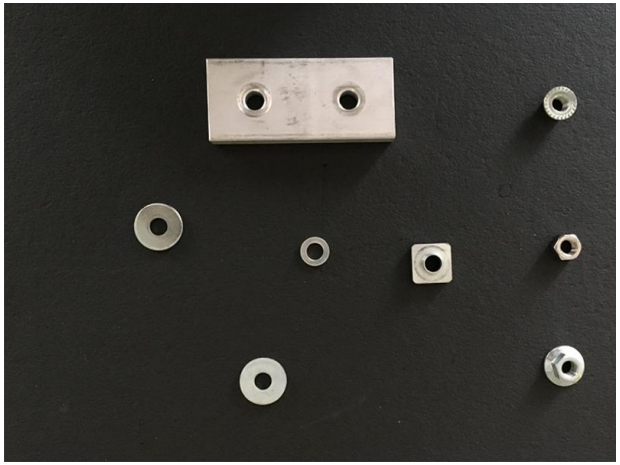
University of Zagreb
FSB Faculty of mechanical engineering and naval arhitecture

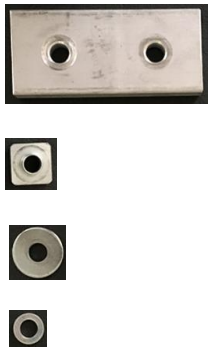
Practical task 1

Template matching



Test different methods and templates





Practical task 1

Template matching

matchTemplate()

```
void cv::matchTemplate ( InputArray  image,
                         InputArray  templ,
                         OutputArray result,
                         int         method,
                         InputArray  mask = noArray()
                       )
```

Python:

```
cv.matchTemplate( image, templ, method[, result[, mask]] ) -> result
```

`#include <opencv2/imgproc.hpp>`

Compares a template against overlapped image regions.

The function slides through image , compares the overlapped patches of size $w \times h$ against templ using the specified method and stores the comparison results in result . Here are the formulae for the available comparison methods (I denotes image, T template, R result). The summation is done over template and/or the image patch: $x' = 0...w-1, y' = 0...h-1$

After the function finishes the comparison, the best matches can be found as global minimums (when **TM_SQDIFF** was used) or maximums (when **TM_CCORR** or **TM_CCOEFF** was used) using the **minMaxLoc** function. In case of a color image, template summation in the numerator and each sum in the denominator is done over all of the channels and separate mean values are used for each channel. That is, the function can take a color template and a color image. The result will still be a single-channel image, which is easier to analyze.

Parameters

- **image** Image where the search is running. It must be 8-bit or 32-bit floating-point.
- **templ** Searched template. It must be not greater than the source image and have the same data type.
- **result** Map of comparison results. It must be single-channel 32-bit floating-point. If image is $W \times H$ and templ is $w \times h$, then result is $(W-w+1) \times (H-h+1)$.
- **method** Parameter specifying the comparison method, see **TemplateMatchModes**
- **mask** Mask of searched template. It must have the same datatype and size with templ. It is not set by default. Currently, only the **TM_SQDIFF** and **TM_CCORR_NORMED** methods are supported.

Examples:

samples/cpp/tutorial_code/Histograms_Matching/MatchTemplate_Demo.cpp.

normalize() [1/2]

```
void cv::normalize ( InputArray       src,
                     InputOutputArray dst,
                     double           alpha = 1,
                     double           beta = 0,
                     int              norm_type = NORM_L2,
                     int              dtype = -1,
                     InputArray       mask = noArray()
                   )
```

Python:

```
cv.normalize( src, dst[, alpha[, beta[, norm_type[, dtype[, mask]]]]] ) -> dst
```

`#include <opencv2/core.hpp>`

Normalizes the norm or value range of an array.

The function **cv::normalize** normalizes scale and shift the input array elements so that

$$\|\texttt{dst}\|_{L_p} = \texttt{alpha}$$

(where p=Inf, 1 or 2) when normType=NORM_INF, NORM_L1, or NORM_L2, respectively; or so that

$$\min_I \texttt{dst}(I) = \texttt{alpha}, \ \max_I \texttt{dst}(I) = \texttt{beta}$$

minMaxLoc() [1/2]

```
void cv::minMaxLoc ( InputArray src,
                     double *   minVal,
                     double *   maxVal = 0,
                     Point *    minLoc = 0,
                     Point *    maxLoc = 0,
                     InputArray mask = noArray()
                   )
```

Python:

```
cv.minMaxLoc( src[, mask] ) -> minVal, maxVal, minLoc, maxLoc
```

`#include <opencv2/core.hpp>`

Finds the global minimum and maximum in an array.

The function **cv::minMaxLoc** finds the minimum and maximum element values and their positions. The extremums are searched across the whole array or, if mask is not an empty array, in the specified array region.

The function do not work with multi-channel arrays. If you need to find minimum or maximum elements across all the channels, use **Mat::reshape** first to reinterpret the array as single-channel. Or you may extract the particular channel using either extractImageCOI , or mixChannels , or split .

Parameters

- **src** input single-channel array.
- **minVal** pointer to the returned minimum value; NULL is used if not required.
- **maxVal** pointer to the returned maximum value; NULL is used if not required.
- **minLoc** pointer to the returned minimum location (in 2D case); NULL is used if not required.
- **maxLoc** pointer to the returned maximum location (in 2D case); NULL is used if not required.
- **mask** optional mask used to select a sub-array.

See also

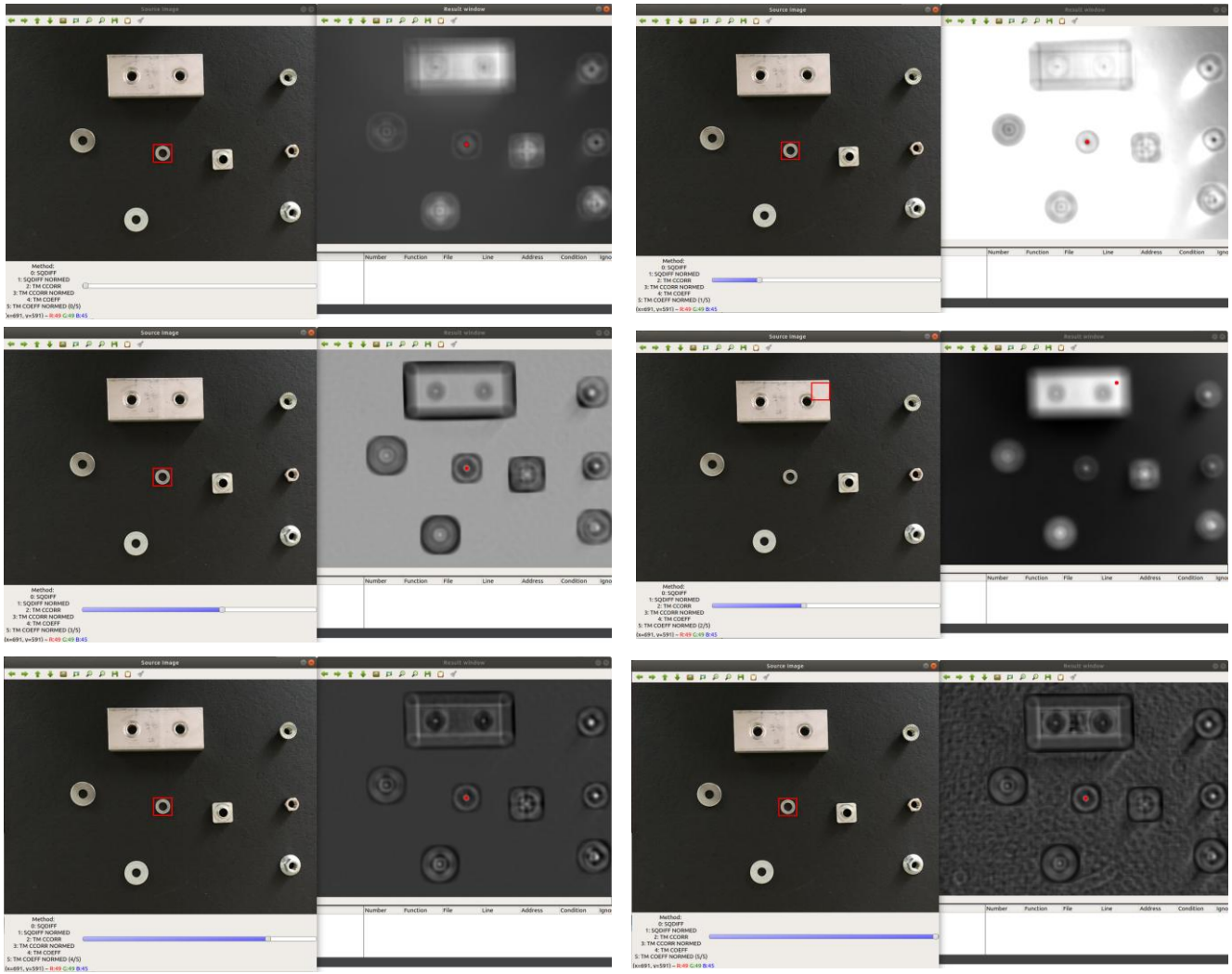
max, min, compare, inRange, extractImageCOI, mixChannels, split, Mat::reshape

Examples:

samples/cpp/image_alignment.cpp, samples/cpp/tutorial_code/Histograms_Matching/MatchTemplate_Demo.cpp, samples/dnn/classification.cpp, samples/dnn/object_detection.cpp, samples/dnn/openpose.cpp, and samples/dnn/text_detection.cpp.

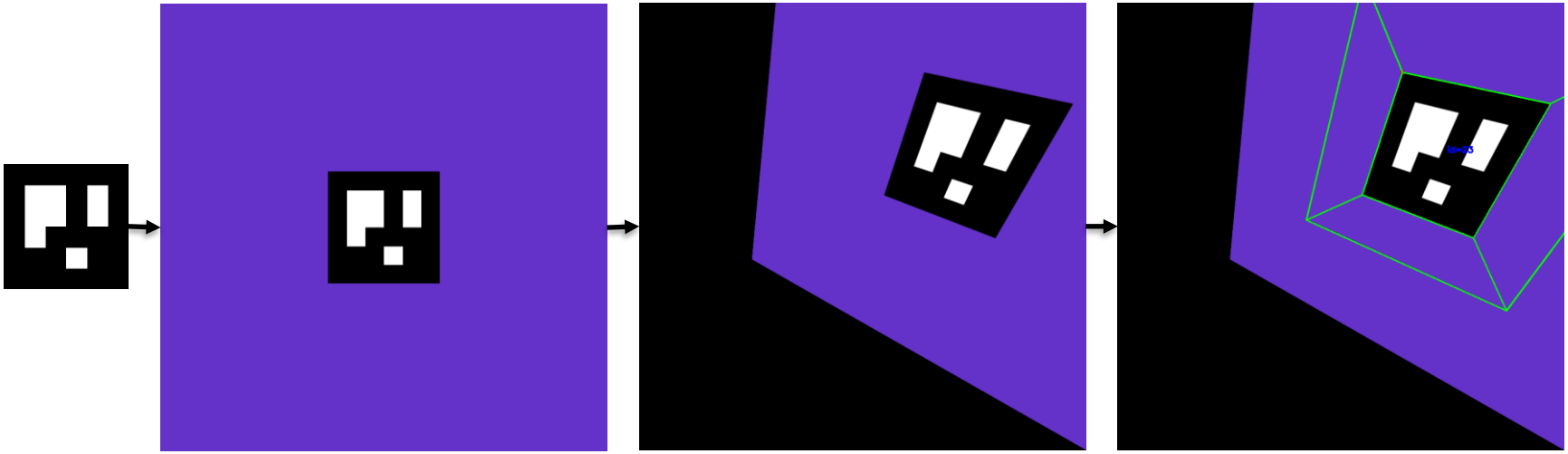
Practical task 1

Template matching - example of results



Practical task 2

Load and generate marker	Create colored background	Apply rotation and warp perspective	Detect markers and estimate pose	Draw cube
getPredefinedDictionary() *generateImageMarker()*	*cvtColor()*	*Rodrigues()* *warpPerspective()*	*detectMarkers()* *solvePnP()*	*projectPoints()* *line()*



Practical task 2

Purpose: Converts a rotation vector into a 3×3 rotation matrix.

Rodrigues()

```
void cv::Rodrigues ( InputArray  src,
                     OutputArray dst,
                     OutputArray jacobian = noArray()
                   )
```

Python:

```
cv.Rodrigues( src[, dst[, jacobian]] ) -> dst, jacobian
```

```
#include <opencv2/calib3d.hpp>
```

Converts a rotation matrix to a rotation vector or vice versa.

Parameters

src	Input rotation vector (3x1 or 1x3) or rotation matrix (3x3).
dst	Output rotation matrix (3x3) or rotation vector (3x1 or 1x3), respectively.
jacobian	Optional output Jacobian matrix, 3x9 or 9x3, which is a matrix of partial derivatives of the output array components with respect to the input array components.

$$\theta \leftarrow norm(r)$$
$$r \leftarrow r/\theta$$
$$R = \cos(\theta) I + (1 - \cos\theta) r r^T + \sin(\theta) \begin{bmatrix} 0 & -r_z & r_y \\ r_z & 0 & -r_x \\ -r_y & r_x & 0 \end{bmatrix}$$

Inverse transformation can be also done easily, since

$$\sin(\theta) \begin{bmatrix} 0 & -r_z & r_y \\ r_z & 0 & -r_x \\ -r_y & r_x & 0 \end{bmatrix} = \frac{R - R^T}{2}$$

A rotation vector is a convenient and most compact representation of a rotation matrix (since any rotation matrix has just 3 degrees of freedom). The representation is used in the global 3D geometry optimization procedures like **calibrateCamera**, **stereoCalibrate**, or **solvePnP** .

Purpose: Applies a perspective (homography) warp to simulate a 3D transformation in 2D space.

warpPerspective()

```
void cv::warpPerspective ( InputArray  src,
                           OutputArray dst,
                           InputArray  M,
                           Size        dsize,
                           int         flags = INTER_LINEAR ,
                           int         borderMode = BORDER_CONSTANT ,
                           const Scalar & borderValue = Scalar()
                         )
```

Python:

```
cv.warpPerspective( src, M, dsize[, dst[, flags[, borderMode[, borderValue]]]] ) -> dst
```

```
#include <opencv2/imgproc.hpp>
```

Applies a perspective transformation to an image.

The function warpPerspective transforms the source image using the specified matrix:

$$\texttt{dst}(x, y) = \texttt{src}\left(\frac{M_{11}x + M_{12}y + M_{13}}{M_{31}x + M_{32}y + M_{33}}, \frac{M_{21}x + M_{22}y + M_{23}}{M_{31}x + M_{32}y + M_{33}} \right)$$

when the flag **WARP_INVERSE_MAP** is set. Otherwise, the transformation is first inverted with invert and then put in the formula above instead of M. The function cannot operate in-place.

Parameters

src	input image.
dst	output image that has the size dsize and the same type as src .
M	3×3 transformation matrix.
dsize	size of the output image.
flags	combination of interpolation methods (**INTER_LINEAR** or **INTER_NEAREST**) and the optional flag **WARP_INVERSE_MAP**, that sets M as the inverse transformation ($\texttt{dst} \rightarrow \texttt{src}$).
borderMode	pixel extrapolation method (**BORDER_CONSTANT** or **BORDER_REPLICATE**).
borderValue	value used in case of a constant border; by default, it equals 0.

See also

warpAffine, **resize**, **remap**, **getRectSubPix**, **perspectiveTransform**

Practical task 2

Purpose: Estimates the 3D pose (rotation and translation) of an object based on known geometry and its 2D image.

solvePnP()

```
bool cv::solvePnP ( InputArray    objectPoints,
                    InputArray    imagePoints,
                    InputArray    cameraMatrix,
                    InputArray    distCoeffs,
                    OutputArray   rvec,
                    OutputArray   tvec,
                    bool          useExtrinsicGuess = false ,
                    int           flags = SOLVEPNP_ITERATIVE
                  )
```

Python:

```
cv.solvePnP( objectPoints, imagePoints, cameraMatrix, distCoeffs[, rvec[, tvec[, useExtrinsicGuess[, flags]]]] ) -> retval, rvec, tvec
```

```
#include <opencv2/calib3d.hpp>
```

Finds an object pose from 3D-2D point correspondences.

Purpose: Projects 3D object points to 2D image coordinates using estimated pose.

projectPoints()

```
void cv::projectPoints ( InputArray    objectPoints,
                         InputArray    rvec,
                         InputArray    tvec,
                         InputArray    cameraMatrix,
                         InputArray    distCoeffs,
                         OutputArray   imagePoints,
                         OutputArray   jacobian = noArray() ,
                         double        aspectRatio = 0
                       )
```

Python:

```
cv.projectPoints( objectPoints, rvec, tvec, cameraMatrix, distCoeffs[, imagePoints[, jacobian[, aspectRatio]]] ) -> imagePoints, jacobian
```

```
#include <opencv2/calib3d.hpp>
```

Projects 3D points to an image plane.

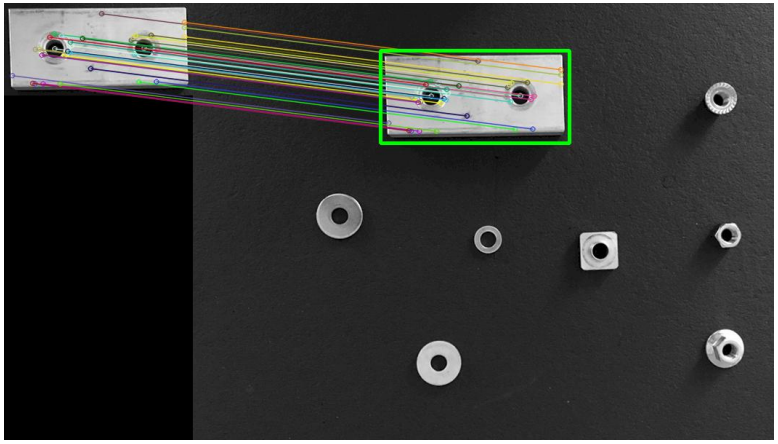
Parameters

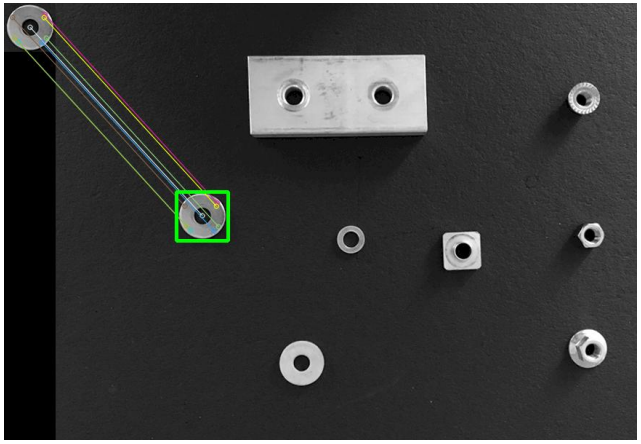
objectPoints	Array of object points expressed wrt. the world coordinate frame. A 3xN/Nx3 1-channel or 1xN/Nx1 3-channel (or vector<Point3f>), where N is the number of points in the view.
rvec	The rotation vector (**Rodrigues**) that, together with tvec, performs a change of basis from world to camera coordinate system, see **calibrateCamera** for details.
tvec	The translation vector, see parameter description above.
cameraMatrix	Camera intrinsic matrix $A = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$.
distCoeffs	Input vector of distortion coefficients $(k_1, k_2, p_1, p_2[, k_3[, k_4, k_5, k_6[, s_1, s_2, s_3, s_4[, \tau_x, \tau_y]]]])$ of $4, 5, 8, 12$ or 14 elements . If the vector is empty, the zero distortion coefficients are assumed.
imagePoints	Output array of image points, 1xN/Nx1 2-channel, or vector<Point2f> .
jacobian	Optional output 2Nx(10+<numDistCoeffs>) jacobian matrix of derivatives of image points with respect to components of the rotation vector, translation vector, focal lengths, coordinates of the principal point and the distortion coefficients. In the old interface different components of the jacobian are returned via different output parameters.
aspectRatio	Optional "fixed aspect ratio" parameter. If the parameter is not 0, the function assumes that the aspect ratio (f_x/f_y) is fixed and correspondingly adjusts the jacobian matrix.

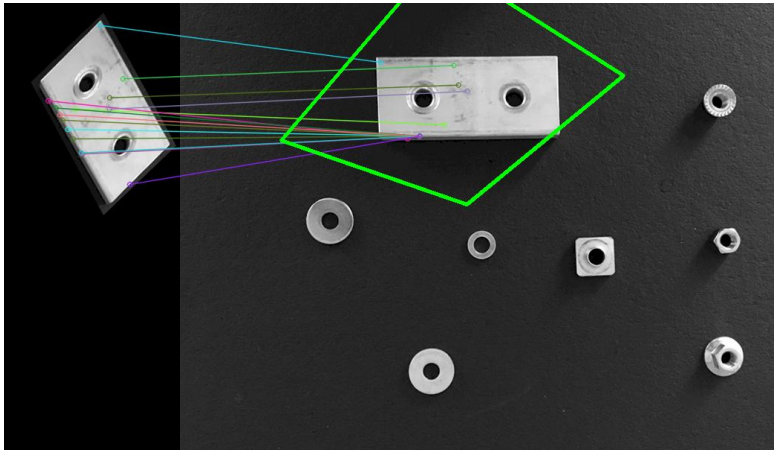
The function computes the 2D projections of 3D points to the image plane, given intrinsic and extrinsic camera parameters. Optionally, the function computes Jacobians -matrices of partial derivatives of image points coordinates (as functions of all the input parameters) with respect to the particular parameters, intrinsic and/or extrinsic. The Jacobians are used during the global optimization in **calibrateCamera**, **solvePnP**, and **stereoCalibrate**. The function itself can also be used to compute a re-projection error, given the current intrinsic and extrinsic parameters.

Practical task 3

SURF for feature matching – prepared code







Practical task 3

create()

```
static Ptr<SURF> cv::xfeatures2d::SURF::create ( double hessianThreshold = 100 ,
                                                 int    nOctaves = 4 ,
                                                 int    nOctaveLayers = 3 ,
                                                 bool   extended = false ,
                                                 bool   upright = false
                                               )
```

Python:

```
cv.xfeatures2d.SURF_create( [, hessianThreshold[, nOctaves[, nOctaveLayers[, extended[, upright]]]]] ) -> retval
```

Parameters

hessianThreshold	Threshold for hessian keypoint detector used in **SURF**.
nOctaves	Number of pyramid octaves the keypoint detector will use.
nOctaveLayers	Number of octave layers within each octave.
extended	Extended descriptor flag (true - use extended 128-element descriptors; false - use 64-element descriptors).
upright	Up-right or rotated features flag (true - do not compute orientation of features; false - compute orientation).

Purpose: Detects keypoints and computes descriptors in a single call (used with feature detectors like ORB, SIFT, etc.).

detectAndCompute()

```
virtual void cv::Feature2D::detectAndCompute ( InputArray                 image,
                                               InputArray                 mask,
                                               std::vector< KeyPoint > &  keypoints,
                                               OutputArray                descriptors,
                                               bool                       useProvidedKeypoints = false
                                             )
```

Python:

```
cv.Feature2D.detectAndCompute( image, mask[, descriptors[, useProvidedKeypoints]] ) -> keypoints, descriptors
```

Detects keypoints and computes the descriptors

knnMatch() [1/2]

```
void cv::DescriptorMatcher::knnMatch ( InputArray                          queryDescriptors,
                                       InputArray                          trainDescriptors,
                                       std::vector< std::vector< DMatch > > & matches,
                                       int                                 k,
                                       InputArray                          mask = noArray() ,
                                       bool                                compactResult = false
                                     ) const
```

Python:

```
cv.DescriptorMatcher.knnMatch( queryDescriptors, trainDescriptors, k[, mask[, compactResult]] ) -> matches
cv.DescriptorMatcher.knnMatch( queryDescriptors, k[, masks[, compactResult]]                  ) -> matches
```

Finds the k best matches for each descriptor from a query set.

Parameters

queryDescriptors	Query set of descriptors.
trainDescriptors	Train set of descriptors. This set is not added to the train descriptors collection stored in the class object.
mask	Mask specifying permissible matches between an input query and train matrices of descriptors.
matches	Matches. Each matches[i] is k or less matches for the same query descriptor.
k	Count of best matches found per each query descriptor or less if a query descriptor has less than k possible matches in total.
compactResult	Parameter used when the mask (or masks) is not empty. If compactResult is false, the matches vector has the same size as queryDescriptors rows. If compactResult is true, the matches vector does not contain matches for fully masked-out query descriptors.

These extended variants of **DescriptorMatcher::match** methods find several best matches for each query descriptor. The matches are returned in the distance increasing order. See **DescriptorMatcher::match** for the details about query and train descriptors.

Purpose: Finds the k-nearest best matches between descriptor sets using distance metrics (usually for feature matching).

Practical task 3

Purpose: Applies a homography matrix to a set of 2D points to map them into a new perspective.

perspectiveTransform()

```
void cv::perspectiveTransform ( InputArray  src,
                                OutputArray dst,
                                InputArray  m
                              )
```

Python:

```
cv.perspectiveTransform( src, m[, dst] ) -> dst
```

```
#include <opencv2/core.hpp>
```

Performs the perspective matrix transformation of vectors.

The function **cv::perspectiveTransform** transforms every element of src by treating it as a 2D or 3D vector, in the following way:

$$(x, y, z) \rightarrow (x'/w, y'/w, z'/w)$$

where

$$(x', y', z', w') = \texttt{mat} \cdot [x \quad y \quad z \quad 1]$$

and

$$w = \begin{cases} w' & \text{if } w' \neq 0 \\ \infty & \text{otherwise} \end{cases}$$

Here a 3D vector transformation is shown. In case of a 2D vector transformation, the z component is omitted.

Note

The function transforms a sparse set of 2D or 3D vectors. If you want to transform an image using perspective transformation, use warpPerspective . If you have an inverse problem, that is, you want to compute the most probable perspective transformation out of several pairs of corresponding points, you can use getPerspectiveTransform or findHomography .

Parameters

src input two-channel or three-channel floating-point array; each element is a 2D/3D vector to be transformed.

dst output array of the same size and type as src.

m 3x3 or 4x4 floating-point transformation matrix.

See also

transform, **warpPerspective**, **getPerspectiveTransform**, **findHomography**

Purpose: Computes the best-fit 3×3 homography matrix that maps one set of 2D points to another.

findHomography() [1/2]

```
Mat cv::findHomography ( InputArray  srcPoints,
                         InputArray  dstPoints,
                         int         method = 0 ,
                         double      ransacReprojThreshold = 3 ,
                         OutputArray mask = noArray() ,
                         const int   maxIters = 2000 ,
                         const double confidence = 0.995
                       )
```

Python:

```
cv.findHomography( srcPoints, dstPoints[, method[, ransacReprojThreshold[, mask[, maxIters[, confidence]]]]] ) -> retval, mask
```

```
#include <opencv2/calib3d.hpp>
```

Finds a perspective transformation between two planes.

Parameters

srcPoints Coordinates of the points in the original plane, a matrix of the type CV_32FC2 or vector<Point2f> .

dstPoints Coordinates of the points in the target plane, a matrix of the type CV_32FC2 or a vector<Point2f> .

method Method used to compute a homography matrix. The following methods are possible:

- **0** - a regular method using all the points, i.e., the least squares method
- **RANSAC** - RANSAC-based robust method
- **LMEDS** - Least-Median robust method
- **RHO** - PROSAC-based robust method

ransacReprojThreshold Maximum allowed reprojection error to treat a point pair as an inlier (used in the RANSAC and RHO methods only). That is, if

$$\| \texttt{dstPoints}_i - \texttt{convertPointsHomogeneous}(\texttt{H} * \texttt{srcPoints}_i) \|_2 > \texttt{ransacReprojThreshold}$$

then the point i is considered as an outlier. If srcPoints and dstPoints are measured in pixels, it usually makes sense to set this parameter somewhere in the range of 1 to 10.

mask Optional output mask set by a robust method (RANSAC or LMeDS). Note that the input mask values are ignored.

maxIters The maximum number of RANSAC iterations.

confidence Confidence level, between 0 and 1.

Student assignment (seminar)

Student assignment - seminar

- *Find ArUco Markers – Detect at least four ArUco markers in an input image (e.g., photo of a printed A4 sheet with markers at the corners).*
- *Rectify Image – Use the markers to compute a homography and warp the perspective, rectifying the image to a top-down view.*
- *Crop Image – Automatically crop the rectified image based on the marker layout without knowing the original aspect ratio.*
- *Threshold Image – Convert the rectified image to grayscale and apply adaptive binary thresholding to highlight darker objects.*
- *Show Contours – Detect and draw contours of all dark shapes on a white background.*