3.2 Geometric Image Transformations

The functions in this section perform various geometrical transformations of 2D images. They do not change the image content but deform the pixel grid and map this deformed grid to the destination image. In fact, to avoid sampling artifacts, the mapping is done in the reverse order, from destination to the source. That is, for each pixel (x, y) of the destination image, the functions compute coordinates of the corresponding "donor" pixel in the source image and copy the pixel value:

$$\texttt{dst}(x, y) = \texttt{src}(f_x(x, y), f_y(x, y))$$

In case when you specify the forward mapping $\langle g_x, g_y \rangle$: `src` $\rightarrow$ `dst` , the OpenCV functions first compute the corresponding inverse mapping $\langle f_x, f_y \rangle$: `dst` $\rightarrow$ `src` and then use the above formula.

The actual implementations of the geometrical transformations, from the most generic `remap()` and to the simplest and the fastest `resize()` , need to solve two main problems with the above formula:

- Extrapolation of non-existing pixels. Similarly to the filtering functions described in the previous section, for some (x, y) , either one of $f_x(x, y)$, or $f_y(x, y)$, or both of them may fall outside of the image. In this case, an extrapolation method needs to be used. OpenCV provides the same selection of extrapolation methods as in the filtering functions. In addition, it provides the method `BORDER_TRANSPARENT` . This means that the corresponding pixels in the destination image will not be modified at all.
- Interpolation of pixel values. Usually $f_x(x, y)$ and $f_y(x, y)$ are floating-point numbers. This means that $\langle f_x, f_y \rangle$ can be either an affine or perspective transformation, or radial lens distortion correction, and so on. So, a pixel value at fractional coordinates needs to be retrieved. In the simplest case, the coordinates can be just rounded to the nearest integer coordinates and the corresponding pixel can be used. This is called a nearest-neighbor interpolation. However, a better result can be achieved by using more sophisticated interpolation methods , where a polynomial function is fit into some neighborhood of the computed pixel $(f_x(x, y), f_y(x, y))$, and then the value of the polynomial at $(f_x(x, y), f_y(x, y))$ is taken as the interpolated pixel value. In OpenCV, you can choose between several interpolation methods. See `resize()` for details.

convertMaps

Converts image transformation maps from one representation to another.

C++: void **convertMaps**(InputArray **map1**, InputArray **map2**, OutputArray **dstmap1**, OutputArray **dstmap2**, int **dstmap1type**, bool **nninterpolation**=false)

Python: cv2.**convertMaps**(map1, map2, dstmap1type[, dstmap1[, dstmap2[, nninterpolation]]]) $\rightarrow$ dstmap1, dstmap2

Parameters

map1 – The first input map of type `CV_16SC2` , `CV_32FC1` , or `CV_32FC2` .

map2 – The second input map of type `CV_16UC1` , `CV_32FC1` , or none (empty matrix), respectively.

dstmap1 – The first output map that has the type `dstmap1type` and the same size as `src` .

dstmap2 – The second output map.

dstmap1type – Type of the first output map that should be `CV_16SC2` , `CV_32FC1` , or `CV_32FC2` .

nninterpolation – Flag indicating whether the fixed-point maps are used for the nearest-neighbor or for a more complex interpolation.

The function converts a pair of maps for `remap()` from one representation to another. The following options (`(map1.type(), map2.type())` $\rightarrow$ `(dstmap1.type(), dstmap2.type())`) are supported:

- `(CV_32FC1, CV_32FC1)` → `(CV_16SC2, CV_16UC1)` . This is the most frequently used conversion operation, in which the original floating-point maps (see `remap()`) are converted to a more compact and much faster fixed-point representation. The first output array contains the rounded coordinates and the second array (created only when `nninterpolation=false`) contains indices in the interpolation tables.
- `(CV_32FC2)` → `(CV_16SC2, CV_16UC1)` . The same as above but the original maps are stored in one 2-channel matrix.
- Reverse conversion. Obviously, the reconstructed floating-point maps will not be exactly the same as the originals.

See Also:

`remap()`, `undistort()`, `initUndistortRectifyMap()`

getAffineTransform

Calculates an affine transform from three pairs of the corresponding points.

C++: Mat **getAffineTransform**(const Point2f* **src**, const Point2f* **dst**)

Python: cv2.**getAffineTransform**(src, dst) → retval

C: CvMat* **cvGetAffineTransform**(const CvPoint2D32f* **src**, const CvPoint2D32f* **dst**, CvMat* **mapMatrix**)

Python: cv.**GetAffineTransform**(src, dst, mapMatrix) → None

Parameters

src – Coordinates of triangle vertices in the source image.

dst – Coordinates of the corresponding triangle vertices in the destination image.

The function calculates the 2×3 matrix of an affine transform so that:

$$\begin{bmatrix} x_i' \\ y_i' \end{bmatrix} = \texttt{map\_matrix} \cdot \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix}$$

where

$$dst(i) = (x_i', y_i'), src(i) = (x_i, y_i), i = 0, 1, 2$$

See Also:

`warpAffine()`, `transform()`

getPerspectiveTransform

Calculates a perspective transform from four pairs of the corresponding points.

C++: Mat **getPerspectiveTransform**(const Point2f* **src**, const Point2f* **dst**)

Python: cv2.**getPerspectiveTransform**(src, dst) → retval

C: CvMat* **cvGetPerspectiveTransform**(const CvPoint2D32f* **src**, const CvPoint2D32f* **dst**, CvMat* **mapMatrix**)

Python: cv.**GetPerspectiveTransform**(src, dst, mapMatrix) → None

Parameters

src – Coordinates of quadrangle vertices in the source image.

dst – Coordinates of the corresponding quadrangle vertices in the destination image.

The function calculates the 3×3 matrix of a perspective transform so that:

$$\begin{bmatrix} t_i x'_i \\ t_i y'_i \\ t_i \end{bmatrix} = \texttt{map\_matrix} \cdot \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix}$$

where

$$dst(i) = (x'_i, y'_i), src(i) = (x_i, y_i), i = 0, 1, 2, 3$$

See Also:

`findHomography()`, `warpPerspective()`, `perspectiveTransform()`

getRectSubPix

Retrieves a pixel rectangle from an image with sub-pixel accuracy.

C++: void **getRectSubPix**(InputArray **image**, Size **patchSize**, Point2f **center**, OutputArray **dst**, int **patchType**=-1)

Python: cv2.**getRectSubPix**(image, patchSize, center[, patch[, patchType]]) → patch

C: void **cvGetRectSubPix**(const CvArr* **src**, CvArr* **dst**, CvPoint2D32f **center**)

Python: cv.**GetRectSubPix**(src, dst, center) → None

Parameters

src – Source image.

patchSize – Size of the extracted patch.

center – Floating point coordinates of the center of the extracted rectangle within the source image. The center must be inside the image.

dst – Extracted patch that has the size `patchSize` and the same number of channels as `src` .

patchType – Depth of the extracted pixels. By default, they have the same depth as `src` .

The function `getRectSubPix` extracts pixels from `src` :

$$dst(x, y) = src(x + \texttt{center.x} - (\texttt{dst.cols} - 1) * 0.5, y + \texttt{center.y} - (\texttt{dst.rows} - 1) * 0.5)$$

where the values of the pixels at non-integer coordinates are retrieved using bilinear interpolation. Every channel of multi-channel images is processed independently. While the center of the rectangle must be inside the image, parts of the rectangle may be outside. In this case, the replication border mode (see `borderInterpolate()`) is used to extrapolate the pixel values outside of the image.

See Also:

`warpAffine()`, `warpPerspective()`

getRotationMatrix2D

Calculates an affine matrix of 2D rotation.

C++: Mat **getRotationMatrix2D**(Point2f **center**, double **angle**, double **scale**)

Python: cv2.**getRotationMatrix2D**(center, angle, scale) → retval

C: CvMat* **cv2DRotationMatrix**(CvPoint2D32f **center**, double **angle**, double **scale**, CvMat* **mapMatrix**)

Python: cv.**GetRotationMatrix2D**(center, angle, scale, mapMatrix) → None

Parameters

center – Center of the rotation in the source image.

angle – Rotation angle in degrees. Positive values mean counter-clockwise rotation (the coordinate origin is assumed to be the top-left corner).

scale – Isotropic scale factor.

mapMatrix – The output affine transformation, 2x3 floating-point matrix.

The function calculates the following matrix:

$$\begin{bmatrix} \alpha & \beta & (1-\alpha)\cdot \texttt{center.x} - \beta \cdot \texttt{center.y} \\ -\beta & \alpha & \beta \cdot \texttt{center.x} + (1-\alpha)\cdot \texttt{center.y} \end{bmatrix}$$

where

$$\begin{aligned} \alpha &= \texttt{scale} \cdot \cos \texttt{angle}, \\ \beta &= \texttt{scale} \cdot \sin \texttt{angle} \end{aligned}$$

The transformation maps the rotation center to itself. If this is not the target, adjust the shift.

See Also:

getAffineTransform(), warpAffine(), transform()

invertAffineTransform

Inverts an affine transformation.

C++: void **invertAffineTransform**(InputArray **M**, OutputArray **iM**)

Python: cv2.**invertAffineTransform**(M[, iM]) → iM

Parameters

M – Original affine transformation.

iM – Output reverse affine transformation.

The function computes an inverse affine transformation represented by 2×3 matrix M:

$$\begin{bmatrix} a_{11} & a_{12} & b_1 \\ a_{21} & a_{22} & b_2 \end{bmatrix}$$

The result is also a 2×3 matrix of the same type as M.

LogPolar

Remaps an image to log-polar space.

C: void **cvLogPolar**(const CvArr* **src**, CvArr* **dst**, CvPoint2D32f **center**, double **M**, int **flags**=CV_INTER_LINEAR+CV_WARP_FILL_OUTLIERS)

Python: cv.**LogPolar**(src, dst, center, M, flags=CV_INNER_LINEAR+CV_WARP_FILL_OUTLIERS) → None

Parameters

src – Source image

dst – Destination image

center – The transformation center; where the output precision is maximal

M – Magnitude scale parameter. See below

flags – A combination of interpolation methods and the following optional flags:

- **CV_WARP_FILL_OUTLIERS** fills all of the destination image pixels. If some of them correspond to outliers in the source image, they are set to zero
- **CV_WARP_INVERSE_MAP** See below

The function `cvLogPolar` transforms the source image using the following transformation:

- Forward transformation (`CV_WARP_INVERSE_MAP` is not set):

$$dst(\phi, \rho) = src(x, y)$$

- Inverse transformation (`CV_WARP_INVERSE_MAP` is set):

$$dst(x, y) = src(\phi, \rho)$$

where

$$\rho = M \cdot \log \sqrt{x^2 + y^2}, \phi = atan(y/x)$$

The function emulates the human "foveal" vision and can be used for fast scale and rotation-invariant template matching, for object tracking and so forth. The function can not operate in-place.

remap

Applies a generic geometrical transformation to an image.

C++: void **remap**(InputArray **src**, OutputArray **dst**, InputArray **map1**, InputArray **map2**, int **interpolation**, int **borderMode**=BORDER_CONSTANT, const Scalar& **borderValue**=Scalar())

Python: cv2.**remap**(src, map1, map2, interpolation[, dst[, borderMode[, borderValue]]]) → dst

C: void **cvRemap**(const CvArr* **src**, CvArr* **dst**, const CvArr* **mapx**, const CvArr* **mapy**, int **flags**=CV_INTER_LINEAR+CV_WARP_FILL_OUTLIERS, CvScalar **fillval**=cvScalarAll(0))

Python: cv.**Remap**(src, dst, mapx, mapy, flags=CV_INNER_LINEAR+CV_WARP_FILL_OUTLIERS, fillval=(0, 0, 0, 0)) → None

Parameters

src – Source image.

dst – Destination image. It has the same size as `map1` and the same type as `src` .

map1 – The first map of either `(x,y)` points or just `x` values having the type `CV_16SC2` , `CV_32FC1` , or `CV_32FC2` . See `convertMaps()` for details on converting a floating point representation to fixed-point for speed.

map2 – The second map of `y` values having the type `CV_16UC1` , `CV_32FC1` , or none (empty map if `map1` is `(x,y)` points), respectively.

interpolation – Interpolation method (see `resize()`). The method `INTER_AREA` is not supported by this function.

borderMode – Pixel extrapolation method (see `borderInterpolate()`). When `borderMode=BORDER_TRANSPARENT` , it means that the pixels in the destination image that corresponds to the "outliers" in the source image are not modified by the function.

borderValue – Value used in case of a constant border. By default, it is 0.

The function `remap` transforms the source image using the specified map:

$$\texttt{dst}(x,y) = \texttt{src}(map_x(x,y), map_y(x,y))$$

where values of pixels with non-integer coordinates are computed using one of available interpolation methods. map_x and map_y can be encoded as separate floating-point maps in map_1 and map_2 respectively, or interleaved floating-point maps of (x,y) in map_1 , or fixed-point maps created by using `convertMaps()` . The reason you might want to convert from floating to fixed-point representations of a map is that they can yield much faster (~2x) remapping operations. In the converted case, map_1 contains pairs `(cvFloor(x), cvFloor(y))` and map_2 contains indices in a table of interpolation coefficients.

This function cannot operate in-place.

resize

Resizes an image.

C++: void **resize**(InputArray **src**, OutputArray **dst**, Size **dsize**, double **fx**=0, double **fy**=0, int **interpolation**=INTER_LINEAR)

Python: cv2.**resize**(src, dsize[, dst[, fx[, fy[, interpolation]]]]) → dst

C: void **cvResize**(const CvArr* **src**, CvArr* **dst**, int **interpolation**=CV_INTER_LINEAR)

Python: cv.**Resize**(src, dst, interpolation=CV_INTER_LINEAR) → None

Parameters

src – Source image.

dst – Destination image. It has the size `dsize` (when it is non-zero) or the size computed from `src.size()` , `fx` , and `fy` . The type of `dst` is the same as of `src` .

dsize – Destination image size. If it is zero, it is computed as:

```
dsize = Size(round(fx*src.cols), round(fy*src.rows))
```

Either `dsize` or both `fx` and `fy` must be non-zero.

fx – Scale factor along the horizontal axis. When it is 0, it is computed as

```
(double)dsize.width/src.cols
```

fy – Scale factor along the vertical axis. When it is 0, it is computed as

```
(double)dsize.height/src.rows
```

interpolation – Interpolation method:

- **INTER_NEAREST** - a nearest-neighbor interpolation
- **INTER_LINEAR** - a bilinear interpolation (used by default)
- **INTER_AREA** - resampling using pixel area relation. It may be a preferred method for image decimation, as it gives moire'-free results. But when the image is zoomed, it is similar to the `INTER_NEAREST` method.
- **INTER_CUBIC** - a bicubic interpolation over 4x4 pixel neighborhood
- **INTER_LANCZOS4** - a Lanczos interpolation over 8x8 pixel neighborhood

The function `resize` resizes the image `src` down to or up to the specified size. Note that the initial `dst` type or size are not taken into account. Instead, the size and type are derived from the `src`,``dsize``,``fx`` , and `fy` . If you want to resize `src` so that it fits the pre-created `dst` , you may call the function as follows:

```
// explicitly specify dsize=dst.size(); fx and fy will be computed from that.
resize(src, dst, dst.size(), 0, 0, interpolation);
```

If you want to decimate the image by factor of 2 in each direction, you can call the function this way:

```
// specify fx and fy and let the function compute the destination image size.
resize(src, dst, Size(), 0.5, 0.5, interpolation);
```

To shrink an image, it will generally look best with CV_INTER_AREA interpolation, whereas to enlarge an image, it will generally look best with CV_INTER_CUBIC (slow) or CV_INTER_LINEAR (faster but still looks OK).

See Also:

`warpAffine()`, `warpPerspective()`, `remap()`

warpAffine

Applies an affine transformation to an image.

C++: void **warpAffine**(InputArray **src**, OutputArray **dst**, InputArray **M**, Size **dsize**, int **flags**=INTER_LINEAR, int **borderMode**=BORDER_CONSTANT, const Scalar& **borderValue**=Scalar())

Python: cv2.**warpAffine**(src, M, dsize[, dst[, flags[, borderMode[, borderValue]]]]) → dst

C: void **cvWarpAffine**(const CvArr* **src**, CvArr* **dst**, const CvMat* **mapMatrix**, int **flags**=CV_INTER_LINEAR+CV_WARP_FILL_OUTLIERS, CvScalar **fillval**=cvScalarAll(0))

Python: cv.**WarpAffine**(src, dst, mapMatrix, flags=CV_INTER_LINEAR+CV_WARP_FILL_OUTLIERS, fillval=(0, 0, 0, 0)) → None

C: void **cvGetQuadrangleSubPix**(const CvArr* **src**, CvArr* **dst**, const CvMat* **mapMatrix**)

Python: cv.**GetQuadrangleSubPix**(src, dst, mapMatrix) → None

Parameters

src – Source image.

dst – Destination image that has the size `dsize` and the same type as `src` .

M – 2×3 transformation matrix.

dsize – Size of the destination image.

flags – Combination of interpolation methods (see `resize()`) and the optional flag `WARP_INVERSE_MAP` that means that `M` is the inverse transformation (`dst` $\rightarrow$ `src`).

borderMode – Pixel extrapolation method (see `borderInterpolate()`). When `borderMode=BORDER_TRANSPARENT` , it means that the pixels in the destination image corresponding to the "outliers" in the source image are not modified by the function.

borderValue – Value used in case of a constant border. By default, it is 0.

The function `warpAffine` transforms the source image using the specified matrix:

$$\texttt{dst}(x,y) = \texttt{src}(\texttt{M}_{11}x + \texttt{M}_{12}y + \texttt{M}_{13}, \texttt{M}_{21}x + \texttt{M}_{22}y + \texttt{M}_{23})$$

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

See Also:

`warpPerspective()`, `resize()`, `remap()`, `getRectSubPix()`, `transform()`

Note: `cvGetQuadrangleSubPix` is similar to `cvWarpAffine`, but the outliers are extrapolated using replication border mode.

---

warpPerspective

Applies a perspective transformation to an image.

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

Python: `cv2.`**warpPerspective**(src, M, dsize[, dst[, flags[, borderMode[, borderValue]]]]) $\rightarrow$ dst

C: void **cvWarpPerspective**(const CvArr* **src**, CvArr* **dst**, const CvMat* **mapMatrix**, int **flags**=CV_INTER_LINEAR+CV_WARP_FILL_OUTLIERS, CvScalar **fillval**=cvScalarAll(0))

Python: `cv.`**WarpPerspective**(src, dst, mapMatrix, flags=CV_INNER_LINEAR+CV_WARP_FILL_OUTLIERS, fillval=(0, 0, 0, 0)) $\rightarrow$ None

Parameters

src – Source image.

dst – Destination image that has the size `dsize` and the same type as `src` .

M – 3×3 transformation matrix.

dsize – Size of the destination image.

flags – Combination of interpolation methods (see `resize()`) and the optional flag `WARP_INVERSE_MAP` that means that `M` is the inverse transformation (`dst` $\rightarrow$ `src`).

borderMode – Pixel extrapolation method (see `borderInterpolate()`). When `borderMode=BORDER_TRANSPARENT` , it means that the pixels in the destination image that corresponds to the "outliers" in the source image are not modified by the function.

borderValue – Value used in case of a constant border. By default, it is 0.

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.

See Also:

`warpAffine()`, `resize()`, `remap()`, `getRectSubPix()`, `perspectiveTransform()`

initUndistortRectifyMap

Computes the undistortion and rectification transformation map.

C++: void **initUndistortRectifyMap**(InputArray **cameraMatrix**, InputArray **distCoeffs**, InputArray **R**, InputArray **newCameraMatrix**, Size **size**, int **m1type**, OutputArray **map1**, OutputArray **map2**)

Python: cv2.**initUndistortRectifyMap**(cameraMatrix, distCoeffs, R, newCameraMatrix, size, m1type[, map1[, map2]]) → map1, map2

C: void **cvInitUndistortRectifyMap**(const CvMat* **cameraMatrix**, const CvMat* **distCoeffs**, const CvMat* **R**, const CvMat* **newCameraMatrix**, CvArr* **map1**, CvArr* **map2**)

C: void **cvInitUndistortMap**(const CvMat* **cameraMatrix**, const CvMat* **distCoeffs**, CvArr* **map1**, CvArr* **map2**)

Python: cv.**InitUndistortRectifyMap**(cameraMatrix, distCoeffs, R, newCameraMatrix, map1, map2) → None

Python: cv.**InitUndistortMap**(cameraMatrix, distCoeffs, map1, map2) → None

Parameters

cameraMatrix – Input camera 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 $(k_1, k_2, p_1, p_2[, k_3[, k_4, k_5, k_6]])$ of 4, 5, or 8 elements. If the vector is NULL/empty, the zero distortion coefficients are assumed.

R – Optional rectification transformation in the object space (3x3 matrix). `R1` or `R2` , computed by `stereoRectify()` can be passed here. If the matrix is empty, the identity transformation is assumed. In `cvInitUndistortMap` R assumed to be an identity matrix.

newCameraMatrix – New camera matrix $A' = \begin{bmatrix} f'_x & 0 & c'_x \\ 0 & f'_y & c'_y \\ 0 & 0 & 1 \end{bmatrix}$.

size – Undistorted image size.

m1type – Type of the first output map that can be `CV_32FC1` or `CV_16SC2` . See `convertMaps()` for details.

map1 – The first output map.

map2 – The second output map.

The function computes the joint undistortion and rectification transformation and represents the result in the form of maps for `remap()` . The undistorted image looks like original, as if it is captured with a camera using the camera matrix `=newCameraMatrix` and zero distortion. In case of a monocular camera, `newCameraMatrix` is usually equal to `cameraMatrix` , or it can be computed by `getOptimalNewCameraMatrix()` for a better control over scaling. In case of a stereo camera, `newCameraMatrix` is normally set to `P1` or `P2` computed by `stereoRectify()` .

Also, this new camera is oriented differently in the coordinate space, according to `R` . That, for example, helps to align two heads of a stereo camera so that the epipolar lines on both images become horizontal and have the same y-coordinate (in case of a horizontally aligned stereo camera).

The function actually builds the maps for the inverse mapping algorithm that is used by `remap()` . That is, for each pixel (u, v) in the destination (corrected and rectified) image, the function computes the corresponding coordinates in the source image (that is, in the original image from camera). The following process is applied:

$$\begin{array}{l} x \leftarrow (u - c'_x)/f'_x \\ y \leftarrow (v - c'_y)/f'_y \\ [X\,Y\,W]^T \leftarrow R^{-1} * [x\,y\,1]^T \\ x' \leftarrow X/W \\ y' \leftarrow Y/W \\ x'' \leftarrow x'(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) + 2p_1 x' y' + p_2(r^2 + 2x'^2) \\ y'' \leftarrow y'(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) + p_1(r^2 + 2y'^2) + 2p_2 x' y' \\ map_x(u, v) \leftarrow x'' f_x + c_x \\ map_y(u, v) \leftarrow y'' f_y + c_y \end{array}$$

where $(k_1, k_2, p_1, p_2[, k_3])$ are the distortion coefficients.

In case of a stereo camera, this function is called twice: once for each camera head, after `stereoRectify()` , which in its turn is called after `stereoCalibrate()` . But if the stereo camera was not calibrated, it is still possible to compute the rectification transformations directly from the fundamental matrix using `stereoRectifyUncalibrated()` . For each camera, the function computes homography `H` as the rectification transformation in a pixel domain, not a rotation matrix `R` in 3D space. `R` can be computed from `H` as

$$R = \texttt{cameraMatrix}^{-1} \cdot H \cdot \texttt{cameraMatrix}$$

where `cameraMatrix` can be chosen arbitrarily.

getDefaultNewCameraMatrix

Returns the default new camera matrix.

C++: Mat **getDefaultNewCameraMatrix**(InputArray **cameraMatrix**, Size **imgsize**=Size(), bool **centerPrincipalPoint**=false)

Python: cv2.**getDefaultNewCameraMatrix**(cameraMatrix[, imgsize[, centerPrincipalPoint]]) → retval

Parameters

cameraMatrix – Input camera matrix.

imgsize – Camera view image size in pixels.

centerPrincipalPoint – Location of the principal point in the new camera matrix. The parameter indicates whether this location should be at the image center or not.

The function returns the camera matrix that is either an exact copy of the input `cameraMatrix` (when `centerPrinicipalPoint=false`), or the modified one (when `centerPrincipalPoint=true`).

In the latter case, the new camera matrix will be:

$$\begin{bmatrix} f_x & 0 & (\texttt{imgSize.width} - 1) * 0.5 \\ 0 & f_y & (\texttt{imgSize.height} - 1) * 0.5 \\ 0 & 0 & 1 \end{bmatrix},$$

where f_x and f_y are $(0,0)$ and $(1,1)$ elements of `cameraMatrix` , respectively.

By default, the undistortion functions in OpenCV (see `initUndistortRectifyMap()`, `undistort()`) do not move the principal point. However, when you work with stereo, it is important to move the principal points in both views to the same y-coordinate (which is required by most of stereo correspondence algorithms), and may be to the same x-coordinate too. So, you can form the new camera matrix for each view where the principal points are located at the center.

undistort

Transforms an image to compensate for lens distortion.

C++: void **undistort**(InputArray **src**, OutputArray **dst**, InputArray **cameraMatrix**, InputArray **distCoeffs**, InputArray **newCameraMatrix**=noArray())

Python: cv2.**undistort**(src, cameraMatrix, distCoeffs[, dst[, newCameraMatrix]]) → dst

C: void **cvUndistort2**(const CvArr* **src**, CvArr* **dst**, const CvMat* **cameraMatrix**, const CvMat* **distCoeffs**, const CvMat* **newCameraMatrix**=NULL)

Python: cv.**Undistort2**(src, dst, cameraMatrix, distCoeffs) → None

Parameters

src – Input (distorted) image.

dst – Output (corrected) image that has the same size and type as `src` .

cameraMatrix – Input camera 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 $(k_1, k_2, p_1, p_2[, k_3[, k_4, k_5, k_6]])$ of 4, 5, or 8 elements. If the vector is NULL/empty, the zero distortion coefficients are assumed.

newCameraMatrix – Camera matrix of the distorted image. By default, it is the same as `cameraMatrix` but you may additionally scale and shift the result by using a different matrix.

The function transforms an image to compensate radial and tangential lens distortion.

The function is simply a combination of `initUndistortRectifyMap()` (with unity `R`) and `remap()` (with bilinear interpolation). See the former function for details of the transformation being performed.

Those pixels in the destination image, for which there is no correspondent pixels in the source image, are filled with zeros (black color).

A particular subset of the source image that will be visible in the corrected image can be regulated by `newCameraMatrix` . You can use `getOptimalNewCameraMatrix()` to compute the appropriate `newCameraMatrix` depending on your requirements.

The camera matrix and the distortion parameters can be determined using `calibrateCamera()` . If the resolution of images is different from the resolution used at the calibration stage, f_x, f_y, c_x and c_y need to be scaled accordingly, while the distortion coefficients remain the same.

undistortPoints

Computes the ideal point coordinates from the observed point coordinates.

C++: void **undistortPoints**(InputArray **src**, OutputArray **dst**, InputArray **cameraMatrix**, InputArray **distCoeffs**, InputArray **R**=noArray(), InputArray **P**=noArray())

C: void **cvUndistortPoints**(const CvMat* **src**, CvMat* **dst**, const CvMat* **cameraMatrix**, const CvMat* **distCoeffs**, const CvMat* **R**=NULL, const CvMat* **P**=NULL)

Python: cv.**UndistortPoints**(src, dst, cameraMatrix, distCoeffs, R=None, P=None) → None

Parameters

src – Observed point coordinates, 1xN or Nx1 2-channel (CV_32FC2 or CV_64FC2).

dst – Output ideal point coordinates after undistortion and reverse perspective transformation.

cameraMatrix – Camera matrix $\begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$.

distCoeffs – Input vector of distortion coefficients $(k_1, k_2, p_1, p_2[, k_3[, k_4, k_5, k_6]])$ of 4, 5, or 8 elements. If the vector is NULL/empty, the zero distortion coefficients are assumed.

R – Rectification transformation in the object space (3x3 matrix). `R1` or `R2` computed by `stereoRectify()` can be passed here. If the matrix is empty, the identity transformation is used.

P – New camera matrix (3x3) or new projection matrix (3x4). `P1` or `P2` computed by `stereoRectify()` can be passed here. If the matrix is empty, the identity new camera matrix is used.

The function is similar to `undistort()` and `initUndistortRectifyMap()` but it operates on a sparse set of points instead of a raster image. Also the function performs a reverse transformation to `projectPoints()` . In case of a 3D object, it does not reconstruct its 3D coordinates, but for a planar object, it does, up to a translation vector, if the proper `R` is specified.

```
// (u,v) is the input point, (u', v') is the output point
// camera_matrix=[fx 0 cx; 0 fy cy; 0 0 1]
// P=[fx' 0 cx' tx; 0 fy' cy' ty; 0 0 1 tz]
x" = (u - cx)/fx
y" = (v - cy)/fy
(x',y') = undistort(x",y",dist_coeffs)
[X,Y,W]T = R*[x' y' 1]T
x = X/W, y = Y/W
u' = x*fx' + cx'
v' = y*fy' + cy',
```

where `undistort()` is an approximate iterative algorithm that estimates the normalized original point coordinates out of the normalized distorted point coordinates ("normalized" means that the coordinates do not depend on the camera matrix).

The function can be used for both a stereo camera head or a monocular camera (when R is empty).