Camera Calibration and 3D Reconstruction

Topics

Fisheye camera model

Detailed Description

The functions in this section use a so-called pinhole camera model. The view of a scene is obtained by projecting a scene's 3D point P_w into the image plane using a perspective transformation which forms the corresponding pixel p. Both P_w and p are represented in homogeneous coordinates, i.e. as 3D and 2D homogeneous vector respectively. You will find a brief introduction to projective geometry, homogeneous vectors and homogeneous transformations at the end of this section's introduction. For more succinct notation, we often drop the 'homogeneous' and say vector instead of homogeneous vector.

The distortion-free projective transformation given by a pinhole camera model is shown below.

$$s \; p = A \begin{bmatrix} R|t \end{bmatrix} P_w,$$

where P_w is a 3D point expressed with respect to the world coordinate system, p is a 2D pixel in the image plane, A is the camera intrinsic matrix, R and t are the rotation and translation that describe the change of coordinates from world to camera coordinate systems (or camera frame) and s is the projective transformation's arbitrary scaling and not part of the camera model.

The camera intrinsic matrix A (notation used as in [319] and also generally notated as K) projects 3D points given in the camera coordinate system to 2D pixel coordinates, i.e.

$$p = A P_c.$$

The camera intrinsic matrix A is composed of the focal lengths f_x and f_y, which are expressed in pixel units, and the principal point (c_x, c_y), that is usually close to the image center:

$$A = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix},$$

and thus

$$s \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X_c \\ Y_c \\ Z_c \end{bmatrix}.$$

The matrix of intrinsic parameters does not depend on the scene viewed. So, once estimated, it can be re-used as long as the focal length is fixed (in case of a zoom lens). Thus, if an image from the camera is scaled by a factor, all of these parameters need to be scaled (multiplied/divided, respectively) by the same factor.

The joint rotation-translation matrix $[R|t]$ is the matrix product of a projective transformation and a homogeneous transformation. The 3-by-4 projective transformation maps 3D points represented in camera coordinates to 2D points in the image plane and represented in normalized camera coordinates $x' = X_c / Z_c$ and $y' = Y_c / Z_c$:

$$Z_c \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{bmatrix}.$$

The homogeneous transformation is encoded by the extrinsic parameters R and t and represents the change of basis from world coordinate system w to the camera coordinate sytem c. Thus, given the representation of the point P in world coordinates, P_w, we obtain P's representation in the camera coordinate system, P_c, by

$$P_c = \begin{bmatrix} R & t \\ 0 & 1 \end{bmatrix} P_w,$$

This homogeneous transformation is composed out of R, a 3-by-3 rotation matrix, and t, a 3-by-1 translation vector:

$$\begin{bmatrix} R & t \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

and therefore

$$\begin{bmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix}.$$

Combining the projective transformation and the homogeneous transformation, we obtain the projective transformation that maps 3D points in world coordinates into 2D points in the image plane and in normalized camera coordinates:

$$Z_c \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} R|t \end{bmatrix} \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \end{bmatrix} \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix},$$

with $x' = X_c / Z_c$ and $y' = Y_c / Z_c$. Putting the equations for instrincs and extrinsics together, we can write out $s \; p = A \begin{bmatrix} R|t \end{bmatrix} P_w$ as

$$s \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \end{bmatrix} \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix}.$$

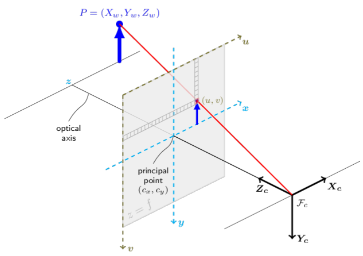
If $Z_c \ne 0$, the transformation above is equivalent to the following,

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} f_x X_c / Z_c + c_x \\ f_y Y_c / Z_c + c_y \end{bmatrix}$$

with

$$\begin{bmatrix} X_c \\ Y_c \\ Z_c \end{bmatrix} = \begin{bmatrix} R|t \end{bmatrix} \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix}.$$

The following figure illustrates the pinhole camera model.



Pinhole camera model

Real lenses usually have some distortion, mostly radial distortion, and slight tangential distortion. So, the above model is extended as:

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} f_x x'' + c_x \\ f_y y'' + c_y \end{bmatrix}$$

where

$$\begin{bmatrix} x'' \\ y'' \end{bmatrix} = \begin{bmatrix} x' \frac{1 + k_1 r^2 + k_2 r^4 + k_3 r^6}{1 + k_4 r^2 + k_5 r^4 + k_6 r^6} + 2 p_1 x' y' + p_2(r^2 + 2 x'^2) + s_1 r^2 + s_2 r^4 \\ y' \frac{1 + k_1 r^2 + k_2 r^4 + k_3 r^6}{1 + k_4 r^2 + k_5 r^4 + k_6 r^6} + p_1 (r^2 + 2 y'^2) + 2 p_2 x' y' + s_3 r^2 + s_4 r^4 \end{bmatrix}$$

with

$$r^2 = x'^2 + y'^2$$

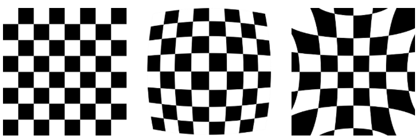
and

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} X_c/Z_c \\ Y_c/Z_c \end{bmatrix},$$

if $Z_c \ne 0$.

The distortion parameters are the radial coefficients k_1, k_2, k_3, k_4, k_5, and k_6 , p_1 and p_2 are the tangential distortion coefficients, and s_1, s_2, s_3, and s_4, are the thin prism distortion coefficients. Higher-order coefficients are not considered in OpenCV.

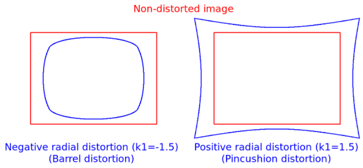
The next figures show two common types of radial distortion: barrel distortion ($1 + k_1 r^2 + k_2 r^4 + k_3 r^6$ monotonically decreasing) and pincushion distortion ($1 + k_1 r^2 + k_2 r^4 + k_3 r^6$ monotonically increasing). Radial distortion is always monotonic for real lenses, and if the estimator produces a non-monotonic result, this should be considered a calibration failure. More generally, radial distortion must be monotonic and the distortion function must be bijective. A failed estimation result may look deceptively good near the image center but will work poorly in e.g. AR/SFM applications. The optimization method used in OpenCV camera calibration does not include these constraints as the framework does not support the required integer programming and polynomial inequalities. See issue #15992 for additional information.



No distortion

Negative radial distortion (Barrel distortion)

Positive radial distortion (Pincushion distortion)



In some cases, the image sensor may be tilted in order to focus an oblique plane in front of the camera (Scheimpflug principle). This can be useful for particle image velocimetry (PIV) or triangulation with a laser fan. The tilt causes a perspective distortion of x'' and y''. This distortion can be modeled in the following way, see e.g. [171].

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} f_x x''' + c_x \\ f_y y''' + c_y \end{bmatrix},$$

where

$$s \begin{bmatrix} x''' \\ y''' \\ 1 \end{bmatrix} = \begin{bmatrix} R_{33}(\tau_x, \tau_y) & 0 & -R_{13}(\tau_x, \tau_y) \\ 0 & R_{33}(\tau_x, \tau_y) & -R_{23}(\tau_x, \tau_y) \\ 0 & 0 & 1 \end{bmatrix} R(\tau_x, \tau_y) \begin{bmatrix} x'' \\ y'' \\ 1 \end{bmatrix}$$

and the matrix $R(\tau_x, \tau_y)$ is defined by two rotations with angular parameter τ_x and τ_y, respectively,

$$R(\tau_x, \tau_y) = \begin{bmatrix} \cos(\tau_y) & 0 & -\sin(\tau_y) \\ 0 & 1 & 0 \\ \sin(\tau_y) & 0 & \cos(\tau_y) \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\tau_x) & \sin(\tau_x) \\ 0 & -\sin(\tau_x) & \cos(\tau_x) \end{bmatrix} = \begin{bmatrix} \cos(\tau_y) & \sin(\tau_y)\sin(\tau_x) & -\sin(\tau_y)\cos(\tau_x) \\ 0 & \cos(\tau_x) & \sin(\tau_x) \\ \sin(\tau_y) & -\cos(\tau_y)\sin(\tau_x) & \cos(\tau_y)\cos(\tau_x) \end{bmatrix}.$$

In the functions below the coefficients are passed or returned as

$$(k_1, k_2, p_1, p_2[, k_3[, k_4, k_5, k_6[, s_1, s_2, s_3, s_4[, \tau_x, \tau_y]]]])$$

vector. That is, if the vector contains four elements, it means that $k_3 = 0$. The distortion coefficients do not depend on the scene viewed. Thus, they also belong to the intrinsic camera parameters. And they remain the same regardless of the captured image resolution. If, for example, a camera has been calibrated on images of 320 x 240 resolution, absolutely the same distortion coefficients can be used for 640 x 480 images from the same camera while f_x, f_y , c_x, and c_y need to be scaled appropriately.

The functions below use the above model to do the following:

- Project 3D points to the image plane given intrinsic and extrinsic parameters.
- Compute extrinsic parameters given intrinsic parameters, a few 3D points, and their projections.
- Estimate intrinsic and extrinsic camera parameters from several views of a known calibration pattern (every view is described by several 3D-2D point correspondences).
- Estimate the relative position and orientation of the stereo camera "heads" and compute the rectification* transformation that makes the camera optical axes parallel.

Homogeneous Coordinates
Homogeneous Coordinates are a system of coordinates that are used in projective geometry. Their use allows to represent points at infinity by finite coordinates and simplifies formulas when compared to the cartesian counterparts, e.g. they have the advantage that affine transformations can be expressed as linear homogeneous transformation.

One obtains the homogeneous vector P_h by appending a 1 along an n-dimensional cartesian vector P e.g. for a 3D cartesian vector the mapping $P \rightarrow P_h$ is:

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \rightarrow \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}.$$

For the inverse mapping $P_h \rightarrow P$, one divides all elements of the homogeneous vector by its last element, e.g. for a 3D homogeneous vector one gets its 2D cartesian counterpart by:

$$\begin{bmatrix} X \\ Y \\ W \end{bmatrix} \rightarrow \begin{bmatrix} X/W \\ Y/W \end{bmatrix},$$

if $W \neq 0$.

Due to this mapping, all multiples kP_h, for $k \neq 0$, of a homogeneous point represent the same point P_h. An intuitive understanding of this property is that under a projective transformation, all multiples of P_h are mapped to the same point. This is the physical observation one does for pinhole cameras, as all points along a ray through the camera's pinhole are projected to the same image point, e.g. all points along the red ray in the image of the pinhole camera model above would be mapped to the same image coordinate. This property is also the source for the scale ambiguity s in the equation of the pinhole camera model.

As mentioned, by using homogeneous coordinates we can express any change of basis parameterized by R and t as a linear transformation, e.g. for the change of basis from coordinate system 0 to coordinate system 1 becomes:

$$P_1 = RP_0 + t \rightarrow P_{h_1} = \begin{bmatrix} R & t \\ 0 & 1 \end{bmatrix} P_{h_0}.$$

Note

- Many functions in this module take a camera intrinsic matrix as an input parameter. Although all functions assume the same structure of this parameter, they may name it differently. The parameter's description, however, will be clear in that a camera intrinsic matrix with the structure shown above is required.
- A calibration sample for 3 cameras in a horizontal position can be found at opencv_source_code/samples/cpp/3calibration.cpp
- A calibration sample based on a sequence of images can be found at opencv_source_code/samples/cpp/calibration.cpp
- A calibration sample in order to do 3D reconstruction can be found at opencv_source_code/samples/cpp/build3dmodel.cpp
- A calibration example on stereo calibration can be found at opencv_source_code/samples/cpp/stereo_calib.cpp
- A calibration example on stereo matching can be found at opencv_source_code/samples/cpp/stereo_match.cpp
- (Python) A camera calibration sample can be found at opencv_source_code/samples/python/calibrate.py

Classes

struct **cv::CirclesGridFinderParameters**

class **cv::LMSolver**

class **cv::StereoBM**
Class for computing stereo correspondence using the block matching algorithm, introduced and contributed to OpenCV by K. Konolige. More...

class **cv::StereoMatcher**
The base class for stereo correspondence algorithms. More...

class **cv::StereoSGBM**
The class implements the modified H. Hirschmuller algorithm [125] that differs from the original one as follows: More...

struct **cv::UsacParams**

Typedefs

typedef **CirclesGridFinderParameters** **cv::CirclesGridFinderParameters2**

Enumerations

```
enum {
  cv::LMEDS = 4 ,
  cv::RANSAC = 8 ,
  cv::RHO = 16 ,
  cv::USAC_DEFAULT = 32 ,
  cv::USAC_PARALLEL = 33 ,
  cv::USAC_FM_8PTS = 34 ,
  cv::USAC_FAST = 35 ,
  cv::USAC_ACCURATE = 36 ,
  cv::USAC_PROSAC = 37 ,
  cv::USAC_MAGSAC = 38
}
```
type of the robust estimation algorithm More...

```
enum {
  cv::CALIB_CB_ADAPTIVE_THRESH = 1 ,
  cv::CALIB_CB_NORMALIZE_IMAGE = 2 ,
  cv::CALIB_CB_FILTER_QUADS = 4 ,
  cv::CALIB_CB_FAST_CHECK = 8 ,
  cv::CALIB_CB_EXHAUSTIVE = 16 ,
  cv::CALIB_CB_ACCURACY = 32 ,
  cv::CALIB_CB_LARGER = 64 ,
  cv::CALIB_CB_MARKER = 128 ,
  cv::CALIB_CB_PLAIN = 256
}
enum {
  cv::CALIB_CB_SYMMETRIC_GRID = 1 ,
  cv::CALIB_CB_ASYMMETRIC_GRID = 2 ,
  cv::CALIB_CB_CLUSTERING = 4
}
enum {
  cv::CALIB_NINTRINSIC = 18 ,
  cv::CALIB_USE_INTRINSIC_GUESS = 0x00001 ,
  cv::CALIB_FIX_ASPECT_RATIO = 0x00002 ,
  cv::CALIB_FIX_PRINCIPAL_POINT = 0x00004 ,
  cv::CALIB_ZERO_TANGENT_DIST = 0x00008 ,
  cv::CALIB_FIX_FOCAL_LENGTH = 0x00010 ,
  cv::CALIB_FIX_K1 = 0x00020 ,
  cv::CALIB_FIX_K2 = 0x00040 ,
  cv::CALIB_FIX_K3 = 0x00080 ,
  cv::CALIB_FIX_K4 = 0x00800 ,
  cv::CALIB_FIX_K5 = 0x01000 ,
  cv::CALIB_FIX_K6 = 0x02000 ,
  cv::CALIB_RATIONAL_MODEL = 0x04000 ,
  cv::CALIB_THIN_PRISM_MODEL = 0x08000 ,
  cv::CALIB_FIX_S1_S2_S3_S4 = 0x10000 ,
  cv::CALIB_TILTED_MODEL = 0x40000 ,
  cv::CALIB_FIX_TAUX_TAUY = 0x80000 ,
  cv::CALIB_USE_QR = 0x100000 ,
  cv::CALIB_FIX_TANGENT_DIST = 0x200000 ,
  cv::CALIB_FIX_INTRINSIC = 0x00100 ,
  cv::CALIB_SAME_FOCAL_LENGTH = 0x00200 ,
  cv::CALIB_ZERO_DISPARITY = 0x00400 ,
  cv::CALIB_USE_LU = (1 << 17) ,
  cv::CALIB_USE_EXTRINSIC_GUESS = (1 << 22)
```

}

enum {
cv::FM_7POINT = 1 ,
cv::FM_8POINT = 2 ,
cv::FM_LMEDS = 4 ,
cv::FM_RANSAC = 8
}
the algorithm for finding fundamental matrix More...

enum cv::HandEyeCalibrationMethod {
cv::CALIB_HAND_EYE_TSAI = 0 ,
cv::CALIB_HAND_EYE_PARK = 1 ,
cv::CALIB_HAND_EYE_HORAUD = 2 ,
cv::CALIB_HAND_EYE_ANDREFF = 3 ,
cv::CALIB_HAND_EYE_DANIILIDIS = 4
}

enum cv::LocalOptimMethod {
cv::LOCAL_OPTIM_NULL =0 ,
cv::LOCAL_OPTIM_INNER_LO =1 ,
cv::LOCAL_OPTIM_INNER_AND_ITER_LO =2 ,
cv::LOCAL_OPTIM_GC =3 ,
cv::LOCAL_OPTIM_SIGMA =4
}

enum cv::NeighborSearchMethod {
cv::NEIGH_FLANN_KNN =0 ,
cv::NEIGH_GRID =1 ,
cv::NEIGH_FLANN_RADIUS =2
}

enum cv::PolishingMethod {
cv::NONE_POLISHER =0 ,
cv::LSQ_POLISHER =1 ,
cv::MAGSAC =2 ,
cv::COV_POLISHER =3
}

enum cv::RobotWorldHandEyeCalibrationMethod {
cv::CALIB_ROBOT_WORLD_HAND_EYE_SHAH = 0 ,
cv::CALIB_ROBOT_WORLD_HAND_EYE_LI = 1
}

enum cv::SamplingMethod {
cv::SAMPLING_UNIFORM =0 ,
cv::SAMPLING_PROGRESSIVE_NAPSAC =1 ,
cv::SAMPLING_NAPSAC =2 ,
cv::SAMPLING_PROSAC =3
}

enum cv::ScoreMethod {
cv::SCORE_METHOD_RANSAC =0 ,
cv::SCORE_METHOD_MSAC =1 ,
cv::SCORE_METHOD_MAGSAC =2 ,
cv::SCORE_METHOD_LMEDS =3
}

enum cv::SolvePnPMethod {
cv::SOLVEPNP_ITERATIVE = 0 ,
cv::SOLVEPNP_EPNP = 1 ,
cv::SOLVEPNP_P3P = 2 ,
cv::SOLVEPNP_DLS = 3 ,
cv::SOLVEPNP_UPNP = 4 ,
cv::SOLVEPNP_AP3P = 5 ,
cv::SOLVEPNP_IPPE = 6 ,
cv::SOLVEPNP_IPPE_SQUARE = 7 ,
cv::SOLVEPNP_SQPNP = 8
}

enum cv::UndistortTypes {
cv::PROJ_SPHERICAL_ORTHO = 0 ,
cv::PROJ_SPHERICAL_EQRECT = 1
}
cv::undistort mode More...

Functions

double cv::calibrateCamera (InputArrayOfArrays objectPoints, InputArrayOfArrays imagePoints, Size imageSize, InputOutputArray cameraMatrix, InputOutputArray distCoeffs, OutputArrayOfArrays rvecs, OutputArrayOfArrays tvecs, int flags=0, TermCriteria criteria=TermCriteria(TermCriteria::COUNT+TermCriteria::EPS, 30, DBL_EPSILON))

double cv::calibrateCamera (InputArrayOfArrays objectPoints, InputArrayOfArrays imagePoints, Size imageSize, InputOutputArray cameraMatrix, InputOutputArray distCoeffs, OutputArrayOfArrays rvecs, OutputArrayOfArrays tvecs, OutputArray stdDeviationsIntrinsics, OutputArray stdDeviationsExtrinsics, OutputArray perViewErrors, int flags=0, TermCriteria criteria=TermCriteria(TermCriteria::COUNT+TermCriteria::EPS, 30, DBL_EPSILON))
Finds the camera intrinsic and extrinsic parameters from several views of a calibration pattern.

double cv::calibrateCameraRO (InputArrayOfArrays objectPoints, InputArrayOfArrays imagePoints, Size imageSize, int iFixedPoint, InputOutputArray cameraMatrix, InputOutputArray distCoeffs, OutputArrayOfArrays rvecs, OutputArrayOfArrays tvecs, OutputArray newObjPoints, int flags=0, TermCriteria criteria=TermCriteria(TermCriteria::COUNT+TermCriteria::EPS, 30, DBL_EPSILON))

double cv::calibrateCameraRO (InputArrayOfArrays objectPoints, InputArrayOfArrays imagePoints, Size imageSize, int iFixedPoint, InputOutputArray cameraMatrix, InputOutputArray distCoeffs, OutputArrayOfArrays rvecs, OutputArrayOfArrays tvecs, OutputArray newObjPoints, OutputArray stdDeviationsIntrinsics, OutputArray stdDeviationsExtrinsics, OutputArray stdDeviationsObjPoints, OutputArray perViewErrors, int flags=0, TermCriteria criteria=TermCriteria(TermCriteria::COUNT+TermCriteria::EPS, 30, DBL_EPSILON))
Finds the camera intrinsic and extrinsic parameters from several views of a calibration pattern.

void cv::calibrateHandEye (InputArrayOfArrays R_gripper2base, InputArrayOfArrays t_gripper2base, InputArrayOfArrays R_target2cam, InputArrayOfArrays t_target2cam, OutputArray R_cam2gripper, OutputArray t_cam2gripper, HandEyeCalibrationMethod method=CALIB_HAND_EYE_TSAI)
Computes Hand-Eye calibration: ${}^{g}\mathbf{T}_c$.

void cv::calibrateRobotWorldHandEye (InputArrayOfArrays R_world2cam, InputArrayOfArrays t_world2cam, InputArrayOfArrays R_base2gripper, InputArrayOfArrays t_base2gripper, OutputArray R_base2world, OutputArray t_base2world, OutputArray R_gripper2cam, OutputArray t_gripper2cam, RobotWorldHandEyeCalibrationMethod method=CALIB_ROBOT_WORLD_HAND_EYE_SHAH)
Computes Robot-World/Hand-Eye calibration: ${}^{w}\mathbf{T}_b$ and ${}^{c}\mathbf{T}_g$.

void cv::calibrationMatrixValues (InputArray cameraMatrix, Size imageSize, double apertureWidth, double apertureHeight, double &fovx, double &fovy, double &focalLength, Point2d &principalPoint, double &aspectRatio)
Computes useful camera characteristics from the camera intrinsic matrix.

bool cv::checkChessboard (InputArray img, Size size)

void cv::composeRT (InputArray rvec1, InputArray tvec1, InputArray rvec2, InputArray tvec2, OutputArray rvec3, OutputArray tvec3, OutputArray dr3dr1=noArray(), OutputArray dr3dt1=noArray(), OutputArray dr3dr2=noArray(), OutputArray dr3dt2=noArray(), OutputArray dt3dr1=noArray(), OutputArray dt3dt1=noArray(), OutputArray dt3dr2=noArray(), OutputArray dt3dt2=noArray())
Combines two rotation-and-shift transformations.

void cv::computeCorrespondEpilines (InputArray points, int whichImage, InputArray F, OutputArray lines)
For points in an image of a stereo pair, computes the corresponding epilines in the other image.

void cv::convertPointsFromHomogeneous (InputArray src, OutputArray dst)
Converts points from homogeneous to Euclidean space.

void cv::convertPointsHomogeneous (InputArray src, OutputArray dst)
Converts points to/from homogeneous coordinates.

void cv::convertPointsToHomogeneous (InputArray src, OutputArray dst)
Converts points from Euclidean to homogeneous space.

void cv::correctMatches (InputArray F, InputArray points1, InputArray points2, OutputArray newPoints1, OutputArray newPoints2)
Refines coordinates of corresponding points.

void cv::decomposeEssentialMat (InputArray E, OutputArray R1, OutputArray R2, OutputArray t)
Decompose an essential matrix to possible rotations and translation.

int cv::decomposeHomographyMat (InputArray H, InputArray K, OutputArrayOfArrays rotations, OutputArrayOfArrays translations, OutputArrayOfArrays normals)
Decompose a homography matrix to rotation(s), translation(s) and plane normal(s).

void cv::decomposeProjectionMatrix (InputArray projMatrix, OutputArray cameraMatrix, OutputArray rotMatrix, OutputArray transVect, OutputArray rotMatrixX=noArray(), OutputArray rotMatrixY=noArray(), OutputArray rotMatrixZ=noArray(), OutputArray eulerAngles=noArray())
Decomposes a projection matrix into a rotation matrix and a camera intrinsic matrix.

void cv::drawChessboardCorners (InputOutputArray image, Size patternSize, InputArray corners, bool patternWasFound)
Renders the detected chessboard corners.

void cv::drawFrameAxes (InputOutputArray image, InputArray cameraMatrix, InputArray distCoeffs, InputArray rvec, InputArray tvec, float length, int thickness=3)
Draw axes of the world/object coordinate system from pose estimation.

cv::Mat cv::estimateAffine2D (InputArray from, InputArray to, OutputArray inliers=noArray(), int method=RANSAC, double ransacReprojThreshold=3, size_t maxIters=2000, double confidence=0.99, size_t refineIters=10)
Computes an optimal affine transformation between two 2D point sets.

cv::Mat cv::estimateAffine2D (InputArray pts1, InputArray pts2, OutputArray inliers, const UsacParams &params)

cv::Mat cv::estimateAffine3D (InputArray src, InputArray dst, double *scale=nullptr, bool force_rotation=true)
Computes an optimal affine transformation between two 3D point sets.

int cv::estimateAffine3D (InputArray src, InputArray dst, OutputArray out, OutputArray inliers, double ransacThreshold=3, double confidence=0.99)
Computes an optimal affine transformation between two 3D point sets.

cv::Mat cv::estimateAffinePartial2D (InputArray from, InputArray to, OutputArray inliers=noArray(), int method=RANSAC, double ransacReprojThreshold=3, size_t maxIters=2000, double confidence=0.99, size_t refineIters=10)
Computes an optimal limited affine transformation with 4 degrees of freedom between two 2D point sets.

Scalar cv::estimateChessboardSharpness (InputArray image, Size patternSize, InputArray corners, float rise_distance=0.8F, bool vertical=false, OutputArray sharpness=noArray())
Estimates the sharpness of a detected chessboard.

int cv::estimateTranslation3D (InputArray src, InputArray dst, OutputArray out, OutputArray inliers, double ransacThreshold=3, double confidence=0.99)
Computes an optimal translation between two 3D point sets.

void cv::filterHomographyDecompByVisibleRefpoints (InputArrayOfArrays rotations, InputArrayOfArrays normals, InputArray beforePoints, InputArray afterPoints, OutputArray possibleSolutions, InputArray pointsMask=noArray())
Filters homography decompositions based on additional information.

void cv::filterSpeckles (InputOutputArray img, double newVal, int maxSpeckleSize, double maxDiff, InputOutputArray buf=noArray())
Filters off small noise blobs (speckles) in the disparity map.

bool cv::find4QuadCornerSubpix (InputArray img, InputOutputArray corners, Size region_size)
finds subpixel-accurate positions of the chessboard corners

bool cv::findChessboardCorners (InputArray image, Size patternSize, OutputArray corners, int flags=CALIB_CB_ADAPTIVE_THRESH+CALIB_CB_NORMALIZE_IMAGE)
Finds the positions of internal corners of the chessboard.

bool cv::findChessboardCornersSB (InputArray image, Size patternSize, OutputArray corners, int flags, OutputArray meta)

	Finds the positions of internal corners of the chessboard using a sector based approach.
bool	cv::findChessboardCornersSB (InputArray image, Size patternSize, OutputArray corners, int flags=0)
bool	cv::findCirclesGrid (InputArray image, Size patternSize, OutputArray centers, int flags, const Ptr< FeatureDetector > &blobDetector, const CirclesGridFinderParameters &parameters) Finds centers in the grid of circles.
bool	cv::findCirclesGrid (InputArray image, Size patternSize, OutputArray centers, int flags=CALIB_CB_SYMMETRIC_GRID, const Ptr< FeatureDetector > &blobDetector=SimpleBlobDetector::create())
Mat	cv::findEssentialMat (InputArray points1, InputArray points2, double focal, Point2d pp, int method, double prob, double threshold, OutputArray mask)
Mat	cv::findEssentialMat (InputArray points1, InputArray points2, double focal=1.0, Point2d pp=Point2d(0, 0), int method=RANSAC, double prob=0.999, double threshold=1.0, int maxIters=1000, OutputArray mask=noArray())
Mat	cv::findEssentialMat (InputArray points1, InputArray points2, InputArray cameraMatrix, int method, double prob, double threshold, OutputArray mask)
Mat	cv::findEssentialMat (InputArray points1, InputArray points2, InputArray cameraMatrix, int method=RANSAC, double prob=0.999, double threshold=1.0, int maxIters=1000, OutputArray mask=noArray()) Calculates an essential matrix from the corresponding points in two images.
Mat	cv::findEssentialMat (InputArray points1, InputArray points2, InputArray cameraMatrix1, InputArray cameraMatrix2, InputArray dist_coeff1, InputArray dist_coeff2, OutputArray mask, const UsacParams &params)
Mat	cv::findEssentialMat (InputArray points1, InputArray points2, InputArray cameraMatrix1, InputArray distCoeffs1, InputArray cameraMatrix2, InputArray distCoeffs2, int method=RANSAC, double prob=0.999, double threshold=1.0, OutputArray mask=noArray()) Calculates an essential matrix from the corresponding points in two images from potentially two different cameras.
Mat	cv::findFundamentalMat (InputArray points1, InputArray points2, int method, double ransacReprojThreshold, double confidence, int maxIters, OutputArray mask=noArray()) Calculates a fundamental matrix from the corresponding points in two images.
Mat	cv::findFundamentalMat (InputArray points1, InputArray points2, int method=FM_RANSAC, double ransacReprojThreshold=3., double confidence=0.99, OutputArray mask=noArray())
Mat	cv::findFundamentalMat (InputArray points1, InputArray points2, OutputArray mask, const UsacParams &params)
Mat	cv::findFundamentalMat (InputArray points1, InputArray points2, OutputArray mask, int method=FM_RANSAC, double ransacReprojThreshold=3., double confidence=0.99)
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) Finds a perspective transformation between two planes.
Mat	cv::findHomography (InputArray srcPoints, InputArray dstPoints, OutputArray mask, const UsacParams &params)
Mat	cv::findHomography (InputArray srcPoints, InputArray dstPoints, OutputArray mask, int method=0, double ransacReprojThreshold=3)
Mat	cv::getDefaultNewCameraMatrix (InputArray cameraMatrix, Size imgsize=Size(), bool centerPrincipalPoint=false) Returns the default new camera matrix.
Mat	cv::getOptimalNewCameraMatrix (InputArray cameraMatrix, InputArray distCoeffs, Size imageSize, double alpha, Size newImgSize=Size(), Rect *validPixROI=0, bool centerPrincipalPoint=false) Returns the new camera intrinsic matrix based on the free scaling parameter.
Rect	cv::getValidDisparityROI (Rect roi1, Rect roi2, int minDisparity, int numberOfDisparities, int blockSize) computes valid disparity ROI from the valid ROIs of the rectified images (that are returned by stereoRectify)
Mat	cv::initCameraMatrix2D (InputArrayOfArrays objectPoints, InputArrayOfArrays imagePoints, Size imageSize, double aspectRatio=1.0) Finds an initial camera intrinsic matrix from 3D-2D point correspondences.
void	cv::initInverseRectificationMap (InputArray cameraMatrix, InputArray distCoeffs, InputArray R, InputArray newCameraMatrix, const Size &size, int m1type, OutputArray map1, OutputArray map2) Computes the projection and inverse-rectification transformation map. In essense, this is the inverse of initUndistortRectifyMap to accomodate stereo-rectification of projectors ('inverse-cameras') in projector-camera pairs.
void	cv::initUndistortRectifyMap (InputArray cameraMatrix, InputArray distCoeffs, InputArray R, InputArray newCameraMatrix, Size size, int m1type, OutputArray map1, OutputArray map2) Computes the undistortion and rectification transformation map.
float	cv::initWideAngleProjMap (InputArray cameraMatrix, InputArray distCoeffs, Size imageSize, int destImageWidth, int m1type, OutputArray map1, OutputArray map2, enum UndistortTypes projType=PROJ_SPHERICAL_EQRECT, double alpha=0) initializes maps for remap for wide-angle
static float	cv::initWideAngleProjMap (InputArray cameraMatrix, InputArray distCoeffs, Size imageSize, int destImageWidth, int m1type, OutputArray map1, OutputArray map2, int projType, double alpha=0)
void	cv::matMulDeriv (InputArray A, InputArray B, OutputArray dABdA, OutputArray dABdB) Computes partial derivatives of the matrix product for each multiplied matrix.
void	cv::projectPoints (InputArray objectPoints, InputArray rvec, InputArray tvec, InputArray cameraMatrix, InputArray distCoeffs, OutputArray imagePoints, OutputArray jacobian=noArray(), double aspectRatio=0) Projects 3D points to an image plane.
int	cv::recoverPose (InputArray E, InputArray points1, InputArray points2, InputArray cameraMatrix, OutputArray R, OutputArray t, double distanceThresh, InputOutputArray mask=noArray(), OutputArray triangulatedPoints=noArray())
int	cv::recoverPose (InputArray E, InputArray points1, InputArray points2, InputArray cameraMatrix, OutputArray R, OutputArray t, InputOutputArray mask=noArray()) Recovers the relative camera rotation and the translation from an estimated essential matrix and the corresponding points in two images, using chirality check. Returns the number of inliers that pass the check.
int	cv::recoverPose (InputArray E, InputArray points1, InputArray points2, OutputArray R, OutputArray t, double focal=1.0, Point2d pp=Point2d(0, 0), InputOutputArray mask=noArray())
int	cv::recoverPose (InputArray points1, InputArray points2, InputArray cameraMatrix1, InputArray distCoeffs1, InputArray cameraMatrix2, InputArray distCoeffs2, OutputArray E, OutputArray R, OutputArray t, int method=cv::RANSAC, double prob=0.999, double threshold=1.0, InputOutputArray mask=noArray()) Recovers the relative camera rotation and the translation from corresponding points in two images from two different cameras, using cheirality check. Returns the number of inliers that pass the check.
float	cv::rectify3Collinear (InputArray cameraMatrix1, InputArray distCoeffs1, InputArray cameraMatrix2, InputArray distCoeffs2, InputArray cameraMatrix3, InputArray distCoeffs3, InputArrayOfArrays imgpt1, InputArrayOfArrays imgpt3, Size imageSize, InputArray R12, InputArray T12, InputArray R13, InputArray T13, OutputArray R1, OutputArray R2, OutputArray R3, OutputArray P1, OutputArray P2, OutputArray P3, OutputArray Q, double alpha, Size newImgSize, Rect *roi1, Rect *roi2, int flags) computes the rectification transformations for 3-head camera, where all the heads are on the same line.
void	cv::reprojectImageTo3D (InputArray disparity, OutputArray _3dImage, InputArray Q, bool handleMissingValues=false, int ddepth=-1) Reprojects a disparity image to 3D space.
void	cv::Rodrigues (InputArray src, OutputArray dst, OutputArray jacobian=noArray()) Converts a rotation matrix to a rotation vector or vice versa.
Vec3d	cv::RQDecomp3x3 (InputArray src, OutputArray mtxR, OutputArray mtxQ, OutputArray Qx=noArray(), OutputArray Qy=noArray(), OutputArray Qz=noArray()) Computes an RQ decomposition of 3x3 matrices.
double	cv::sampsonDistance (InputArray pt1, InputArray pt2, InputArray F) Calculates the Sampson Distance between two points.
int	cv::solveP3P (InputArray objectPoints, InputArray imagePoints, InputArray cameraMatrix, InputArray distCoeffs, OutputArrayOfArrays rvecs, OutputArrayOfArrays tvecs, int flags) Finds an object pose from 3 3D-2D point correspondences.
bool	cv::solvePnP (InputArray objectPoints, InputArray imagePoints, InputArray cameraMatrix, InputArray distCoeffs, OutputArray rvec, OutputArray tvec, bool useExtrinsicGuess=false, int flags=SOLVEPNP_ITERATIVE) Finds an object pose from 3D-2D point correspondences.
int	cv::solvePnPGeneric (InputArray objectPoints, InputArray imagePoints, InputArray cameraMatrix, InputArray distCoeffs, OutputArrayOfArrays rvecs, OutputArrayOfArrays tvecs, bool useExtrinsicGuess=false, SolvePnPMethod flags=SOLVEPNP_ITERATIVE, InputArray rvec=noArray(), InputArray tvec=noArray(), OutputArray reprojectionError=noArray()) Finds an object pose from 3D-2D point correspondences.
bool	cv::solvePnPRansac (InputArray objectPoints, InputArray imagePoints, InputArray cameraMatrix, InputArray distCoeffs, OutputArray rvec, OutputArray tvec, bool useExtrinsicGuess=false, int iterationsCount=100, float reprojectionError=8.0, double confidence=0.99, OutputArray inliers=noArray(), int flags=SOLVEPNP_ITERATIVE) Finds an object pose from 3D-2D point correspondences using the RANSAC scheme.
bool	cv::solvePnPRansac (InputArray objectPoints, InputArray imagePoints, InputOutputArray cameraMatrix, InputArray distCoeffs, OutputArray rvec, OutputArray tvec, OutputArray inliers, const UsacParams &params=UsacParams())
void	cv::solvePnPRefineLM (InputArray objectPoints, InputArray imagePoints, InputArray cameraMatrix, InputArray distCoeffs, InputOutputArray rvec, InputOutputArray tvec, TermCriteria criteria=TermCriteria(TermCriteria::EPS+TermCriteria::COUNT, 20, FLT_EPSILON)) Refine a pose (the translation and the rotation that transform a 3D point expressed in the object coordinate frame to the camera coordinate frame) from a 3D-2D point correspondences and starting from an initial solution.
void	cv::solvePnPRefineVVS (InputArray objectPoints, InputArray imagePoints, InputArray cameraMatrix, InputArray distCoeffs, InputOutputArray rvec, InputOutputArray tvec, TermCriteria criteria=TermCriteria(TermCriteria::EPS+TermCriteria::COUNT, 20, FLT_EPSILON), double VVSlambda=1) Refine a pose (the translation and the rotation that transform a 3D point expressed in the object coordinate frame to the camera coordinate frame) from a 3D-2D point correspondences and starting from an initial solution.
double	cv::stereoCalibrate (InputArrayOfArrays objectPoints, InputArrayOfArrays imagePoints1, InputArrayOfArrays imagePoints2, InputOutputArray cameraMatrix1, InputOutputArray distCoeffs1, InputOutputArray cameraMatrix2, InputOutputArray distCoeffs2, Size imageSize, InputOutputArray R, InputOutputArray T, OutputArray E, OutputArray F, OutputArray perViewErrors, int flags=CALIB_FIX_INTRINSIC, TermCriteria criteria=TermCriteria(TermCriteria::COUNT+TermCriteria::EPS, 30, 1e-6)) This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.
double	cv::stereoCalibrate (InputArrayOfArrays objectPoints, InputArrayOfArrays imagePoints1, InputArrayOfArrays imagePoints2, InputOutputArray cameraMatrix1, InputOutputArray distCoeffs1, InputOutputArray cameraMatrix2, InputOutputArray distCoeffs2, Size imageSize, InputOutputArray R, InputOutputArray T, OutputArray E, OutputArray F, OutputArrayOfArrays rvecs, OutputArrayOfArrays tvecs, OutputArray perViewErrors, int flags=CALIB_FIX_INTRINSIC, TermCriteria criteria=TermCriteria(TermCriteria::COUNT+TermCriteria::EPS, 30, 1e-6)) Calibrates a stereo camera set up. This function finds the intrinsic parameters for each of the two cameras and the extrinsic parameters between the two cameras.
double	cv::stereoCalibrate (InputArrayOfArrays objectPoints, InputArrayOfArrays imagePoints1, InputArrayOfArrays imagePoints2, InputOutputArray cameraMatrix1, InputOutputArray distCoeffs1, InputOutputArray cameraMatrix2, InputOutputArray distCoeffs2, Size imageSize, OutputArray R, OutputArray T, OutputArray E, OutputArray F, int flags=CALIB_FIX_INTRINSIC, TermCriteria criteria=TermCriteria(TermCriteria::COUNT+TermCriteria::EPS, 30, 1e-6)) This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.
void	cv::stereoRectify (InputArray cameraMatrix1, InputArray distCoeffs1, InputArray cameraMatrix2, InputArray distCoeffs2, Size imageSize, InputArray R, InputArray T, OutputArray R1, OutputArray R2, OutputArray P1, OutputArray P2, OutputArray Q, int flags=CALIB_ZERO_DISPARITY, double alpha=-1, Size newImageSize=Size(), Rect *validPixROI1=0, Rect *validPixROI2=0) Computes rectification transforms for each head of a calibrated stereo camera.
bool	cv::stereoRectifyUncalibrated (InputArray points1, InputArray points2, InputArray F, Size imgSize, OutputArray H1, OutputArray H2, double threshold=5) Computes a rectification transform for an uncalibrated stereo camera.
void	cv::triangulatePoints (InputArray projMatr1, InputArray projMatr2, InputArray projPoints1, InputArray projPoints2, OutputArray points4D) This function reconstructs 3-dimensional points (in homogeneous coordinates) by using their observations with a stereo camera.
void	cv::undistort (InputArray src, OutputArray dst, InputArray cameraMatrix, InputArray distCoeffs, InputArray newCameraMatrix=noArray()) Transforms an image to compensate for lens distortion.
void	cv::undistortImagePoints (InputArray src, OutputArray dst, InputArray cameraMatrix, InputArray distCoeffs, TermCriteria=TermCriteria(TermCriteria::MAX_ITER+TermCriteria::EPS, 5, 0.01)) Compute undistorted image points position.
void	cv::undistortPoints (InputArray src, OutputArray dst, InputArray cameraMatrix, InputArray distCoeffs, InputArray R, InputArray P, TermCriteria criteria)
void	cv::undistortPoints (InputArray src, OutputArray dst, InputArray cameraMatrix, InputArray distCoeffs, InputArray R=noArray(), InputArray P=noArray()) Computes the ideal point coordinates from the observed point coordinates.

void	cv::validateDisparity (InputOutputArray disparity, InputArray cost, int minDisparity, int numberOfDisparities, int disp12MaxDisp=1) validates disparity using the left-right check. The matrix "cost" should be computed by the stereo correspondence algorithm

Typedef Documentation

◆ CirclesGridFinderParameters2

typedef CirclesGridFinderParameters cv::CirclesGridFinderParameters2

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

Enumeration Type Documentation

◆ anonymous enum

anonymous enum

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

type of the robust estimation algorithm

Enumerator	
LMEDS Python: cv.LMEDS	least-median of squares algorithm
RANSAC Python: cv.RANSAC	RANSAC algorithm.
RHO Python: cv.RHO	RHO algorithm.
USAC_DEFAULT Python: cv.USAC_DEFAULT	USAC algorithm, default settings.
USAC_PARALLEL Python: cv.USAC_PARALLEL	USAC, parallel version.
USAC_FM_8PTS Python: cv.USAC_FM_8PTS	USAC, fundamental matrix 8 points.
USAC_FAST Python: cv.USAC_FAST	USAC, fast settings.
USAC_ACCURATE Python: cv.USAC_ACCURATE	USAC, accurate settings.
USAC_PROSAC Python: cv.USAC_PROSAC	USAC, sorted points, runs PROSAC.
USAC_MAGSAC Python: cv.USAC_MAGSAC	USAC, runs MAGSAC++.

◆ anonymous enum

anonymous enum

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

Enumerator	
CALIB_CB_ADAPTIVE_THRESH Python: cv.CALIB_CB_ADAPTIVE_THRESH	
CALIB_CB_NORMALIZE_IMAGE Python: cv.CALIB_CB_NORMALIZE_IMAGE	
CALIB_CB_FILTER_QUADS Python: cv.CALIB_CB_FILTER_QUADS	
CALIB_CB_FAST_CHECK Python: cv.CALIB_CB_FAST_CHECK	
CALIB_CB_EXHAUSTIVE Python: cv.CALIB_CB_EXHAUSTIVE	
CALIB_CB_ACCURACY Python: cv.CALIB_CB_ACCURACY	
CALIB_CB_LARGER Python: cv.CALIB_CB_LARGER	
CALIB_CB_MARKER Python: cv.CALIB_CB_MARKER	
CALIB_CB_PLAIN Python: cv.CALIB_CB_PLAIN	

◆ anonymous enum

anonymous enum

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

Enumerator	
CALIB_CB_SYMMETRIC_GRID Python: cv.CALIB_CB_SYMMETRIC_GRID	
CALIB_CB_ASYMMETRIC_GRID Python: cv.CALIB_CB_ASYMMETRIC_GRID	
CALIB_CB_CLUSTERING Python: cv.CALIB_CB_CLUSTERING	

◆ anonymous enum

anonymous enum

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

Enumerator	
CALIB_NINTRINSIC Python: cv.CALIB_NINTRINSIC	
CALIB_USE_INTRINSIC_GUESS Python: cv.CALIB_USE_INTRINSIC_GUESS	
CALIB_FIX_ASPECT_RATIO Python: cv.CALIB_FIX_ASPECT_RATIO	
CALIB_FIX_PRINCIPAL_POINT Python: cv.CALIB_FIX_PRINCIPAL_POINT	
CALIB_ZERO_TANGENT_DIST Python: cv.CALIB_ZERO_TANGENT_DIST	
CALIB_FIX_FOCAL_LENGTH Python: cv.CALIB_FIX_FOCAL_LENGTH	
CALIB_FIX_K1 Python: cv.CALIB_FIX_K1	
CALIB_FIX_K2 Python: cv.CALIB_FIX_K2	
CALIB_FIX_K3 Python: cv.CALIB_FIX_K3	
CALIB_FIX_K4 Python: cv.CALIB_FIX_K4	
CALIB_FIX_K5 Python: cv.CALIB_FIX_K5	
CALIB_FIX_K6 Python: cv.CALIB_FIX_K6	
CALIB_RATIONAL_MODEL Python: cv.CALIB_RATIONAL_MODEL	
CALIB_THIN_PRISM_MODEL Python: cv.CALIB_THIN_PRISM_MODEL	
CALIB_FIX_S1_S2_S3_S4 Python: cv.CALIB_FIX_S1_S2_S3_S4	
CALIB_TILTED_MODEL Python: cv.CALIB_TILTED_MODEL	
CALIB_FIX_TAUX_TAUY Python: cv.CALIB_FIX_TAUX_TAUY	
CALIB_USE_QR Python: cv.CALIB_USE_QR	use QR instead of SVD decomposition for solving. Faster but potentially less precise
CALIB_FIX_TANGENT_DIST Python: cv.CALIB_FIX_TANGENT_DIST	
CALIB_FIX_INTRINSIC Python: cv.CALIB_FIX_INTRINSIC	
CALIB_SAME_FOCAL_LENGTH Python: cv.CALIB_SAME_FOCAL_LENGTH	
CALIB_ZERO_DISPARITY Python: cv.CALIB_ZERO_DISPARITY	
CALIB_USE_LU Python: cv.CALIB_USE_LU	use LU instead of SVD decomposition for solving. much faster but potentially less precise
CALIB_USE_EXTRINSIC_GUESS Python: cv.CALIB_USE_EXTRINSIC_GUESS	for stereoCalibrate

◆ anonymous enum

anonymous enum

`#include <opencv2/calib3d.hpp>`

the algorithm for finding fundamental matrix

Enumerator	
FM_7POINT Python: cv.FM_7POINT	7-point algorithm
FM_8POINT Python: cv.FM_8POINT	8-point algorithm
FM_LMEDS Python: cv.FM_LMEDS	least-median algorithm. 7-point algorithm is used.
FM_RANSAC Python: cv.FM_RANSAC	RANSAC algorithm. It needs at least 15 points. 7-point algorithm is used.

◆ HandEyeCalibrationMethod

enum cv::HandEyeCalibrationMethod

`#include <opencv2/calib3d.hpp>`

Enumerator	
CALIB_HAND_EYE_TSAI Python: cv.CALIB_HAND_EYE_TSAI	A New Technique for Fully Autonomous and Efficient 3D Robotics Hand/Eye Calibration [274].
CALIB_HAND_EYE_PARK Python: cv.CALIB_HAND_EYE_PARK	Robot Sensor Calibration: Solving AX = XB on the Euclidean Group [213].
CALIB_HAND_EYE_HORAUD Python: cv.CALIB_HAND_EYE_HORAUD	Hand-eye Calibration [126].
CALIB_HAND_EYE_ANDREFF Python: cv.CALIB_HAND_EYE_ANDREFF	On-line Hand-Eye Calibration [12].
CALIB_HAND_EYE_DANIILIDIS Python: cv.CALIB_HAND_EYE_DANIILIDIS	Hand-Eye Calibration Using Dual Quaternions [65].

◆ LocalOptimMethod

enum cv::LocalOptimMethod

`#include <opencv2/calib3d.hpp>`

Enumerator	
LOCAL_OPTIM_NULL Python: cv.LOCAL_OPTIM_NULL	
LOCAL_OPTIM_INNER_LO Python: cv.LOCAL_OPTIM_INNER_LO	
LOCAL_OPTIM_INNER_AND_ITER_LO Python: cv.LOCAL_OPTIM_INNER_AND_ITER_LO	
LOCAL_OPTIM_GC Python: cv.LOCAL_OPTIM_GC	
LOCAL_OPTIM_SIGMA Python: cv.LOCAL_OPTIM_SIGMA	

◆ NeighborSearchMethod

enum cv::NeighborSearchMethod

`#include <opencv2/calib3d.hpp>`

Enumerator	
NEIGH_FLANN_KNN Python: cv.NEIGH_FLANN_KNN	
NEIGH_GRID Python: cv.NEIGH_GRID	
NEIGH_FLANN_RADIUS Python: cv.NEIGH_FLANN_RADIUS	

◆ PolishingMethod

enum cv::PolishingMethod

`#include <opencv2/calib3d.hpp>`

Enumerator	
NONE_POLISHER Python: cv.NONE_POLISHER	
LSQ_POLISHER Python: cv.LSQ_POLISHER	
MAGSAC Python: cv.MAGSAC	
COV_POLISHER Python: cv.COV_POLISHER	

◆ RobotWorldHandEyeCalibrationMethod

enum cv::RobotWorldHandEyeCalibrationMethod

`#include <opencv2/calib3d.hpp>`

Enumerator	
CALIB_ROBOT_WORLD_HAND_EYE_SHAH Python: cv.CALIB_ROBOT_WORLD_HAND_EYE_SHAH	Solving the robot-world/hand-eye calibration problem using the kronecker product [244].
CALIB_ROBOT_WORLD_HAND_EYE_LI Python: cv.CALIB_ROBOT_WORLD_HAND_EYE_LI	Simultaneous robot-world and hand-eye calibration using dual-quaternions and kronecker product [162].

◆ SamplingMethod

enum cv::SamplingMethod

`#include <opencv2/calib3d.hpp>`

Enumerator	
SAMPLING_UNIFORM Python: cv.SAMPLING_UNIFORM	
SAMPLING_PROGRESSIVE_NAPSAC Python: cv.SAMPLING_PROGRESSIVE_NAPSAC	
SAMPLING_NAPSAC Python: cv.SAMPLING_NAPSAC	
SAMPLING_PROSAC Python: cv.SAMPLING_PROSAC	

◆ ScoreMethod

enum cv::ScoreMethod

`#include <opencv2/calib3d.hpp>`

Enumerator	
SCORE_METHOD_RANSAC Python: cv.SCORE_METHOD_RANSAC	
SCORE_METHOD_MSAC Python: cv.SCORE_METHOD_MSAC	
SCORE_METHOD_MAGSAC Python: cv.SCORE_METHOD_MAGSAC	
SCORE_METHOD_LMEDS Python: cv.SCORE_METHOD_LMEDS	

◆ SolvePnPMethod

enum **cv::SolvePnPMethod**

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

Enumerator	
SOLVEPNP_ITERATIVE Python: cv.SOLVEPNP_ITERATIVE	Pose refinement using non-linear Levenberg-Marquardt minimization scheme [178] [77] Initial solution for non-planar "objectPoints" needs at least 6 points and uses the DLT algorithm. Initial solution for planar "objectPoints" needs at least 4 points and uses pose from homography decomposition.
SOLVEPNP_EPNP Python: cv.SOLVEPNP_EPNP	EPnP: Efficient Perspective-n-Point Camera Pose Estimation [157].
SOLVEPNP_P3P Python: cv.SOLVEPNP_P3P	Complete Solution Classification for the Perspective-Three-Point Problem [97].
SOLVEPNP_DLS Python: cv.SOLVEPNP_DLS	**Broken implementation. Using this flag will fallback to EPnP.** A Direct Least-Squares (DLS) Method for PnP [124]
SOLVEPNP_UPNP Python: cv.SOLVEPNP_UPNP	**Broken implementation. Using this flag will fallback to EPnP.** Exhaustive Linearization for Robust Camera Pose and Focal Length Estimation [214]
SOLVEPNP_AP3P Python: cv.SOLVEPNP_AP3P	An Efficient Algebraic Solution to the Perspective-Three-Point Problem [144].
SOLVEPNP_IPPE Python: cv.SOLVEPNP_IPPE	Infinitesimal Plane-Based Pose Estimation [61] Object points must be coplanar.
SOLVEPNP_IPPE_SQUARE Python: cv.SOLVEPNP_IPPE_SQUARE	Infinitesimal Plane-Based Pose Estimation [61] This is a special case suitable for marker pose estimation. 4 coplanar object points must be defined in the following order: • point 0: [-squareLength / 2, squareLength / 2, 0] • point 1: [squareLength / 2, squareLength / 2, 0] • point 2: [squareLength / 2, -squareLength / 2, 0] • point 3: [-squareLength / 2, -squareLength / 2, 0]
SOLVEPNP_SQPNP Python: cv.SOLVEPNP_SQPNP	SQPnP: A Consistently Fast and Globally OptimalSolution to the Perspective-n-Point Problem [268].

◆ UndistortTypes

enum **cv::UndistortTypes**

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

cv::undistort mode

Enumerator	
PROJ_SPHERICAL_ORTHO Python: cv.PROJ_SPHERICAL_ORTHO	
PROJ_SPHERICAL_EQRECT Python: cv.PROJ_SPHERICAL_EQRECT	

Function Documentation

◆ calibrateCamera() [1/2]

```
double cv::calibrateCamera ( InputArrayOfArrays   objectPoints,
                             InputArrayOfArrays   imagePoints,
                             Size                 imageSize,
                             InputOutputArray     cameraMatrix,
                             InputOutputArray     distCoeffs,
                             OutputArrayOfArrays  rvecs,
                             OutputArrayOfArrays  tvecs,
                             int                  flags = 0,
                             TermCriteria         criteria = TermCriteria(TermCriteria::COUNT+TermCriteria::EPS, 30, DBL_EPSILON) )
```

Python:

```
cv.calibrateCamera( objectPoints, imagePoints, imageSize, cameraMatrix, distCoeffs[, rvecs[, tvecs[, flags[, criteria]]]] ) -> retval, cameraMatrix, distCoeffs, rvecs, tvecs
cv.calibrateCameraExtended( objectPoints, imagePoints, imageSize, cameraMatrix, distCoeffs[, rvecs[, tvecs[, stdDeviationsIntrinsics[, stdDeviationsExtrinsics[, perViewErrors[, flags[, criteria]]]]]]] ) -> retval, cameraMatrix, distCoeffs, rvecs, tvecs, stdDeviationsIntrinsics, stdDeviationsExtrinsics, perViewErrors
```

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

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

◆ calibrateCamera() [2/2]

```
double cv::calibrateCamera ( InputArrayOfArrays   objectPoints,
                             InputArrayOfArrays   imagePoints,
                             Size                 imageSize,
                             InputOutputArray     cameraMatrix,
                             InputOutputArray     distCoeffs,
                             OutputArrayOfArrays  rvecs,
                             OutputArrayOfArrays  tvecs,
                             OutputArray          stdDeviationsIntrinsics,
                             OutputArray          stdDeviationsExtrinsics,
                             OutputArray          perViewErrors,
                             int                  flags = 0,
                             TermCriteria         criteria = TermCriteria(TermCriteria::COUNT+TermCriteria::EPS, 30, DBL_EPSILON) )
```

Python:

```
cv.calibrateCamera( objectPoints, imagePoints, imageSize, cameraMatrix, distCoeffs[, rvecs[, tvecs[, flags[, criteria]]]] ) -> retval, cameraMatrix, distCoeffs, rvecs, tvecs
cv.calibrateCameraExtended( objectPoints, imagePoints, imageSize, cameraMatrix, distCoeffs[, rvecs[, tvecs[, stdDeviationsIntrinsics[, stdDeviationsExtrinsics[, perViewErrors[, flags[, criteria]]]]]]] ) -> retval, cameraMatrix, distCoeffs, rvecs, tvecs, stdDeviationsIntrinsics, stdDeviationsExtrinsics, perViewErrors
```

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

Finds the camera intrinsic and extrinsic parameters from several views of a calibration pattern.

Parameters

- **objectPoints** In the new interface it is a vector of vectors of calibration pattern points in the calibration pattern coordinate space (e.g. std::vector<std::vector<cv::Vec3f>>). The outer vector contains as many elements as the number of pattern views. If the same calibration pattern is shown in each view and it is fully visible, all the vectors will be the same. Although, it is possible to use partially occluded patterns or even different patterns in different views. Then, the vectors will be different. Although the points are 3D, they all lie in the calibration pattern's XY coordinate plane (thus 0 in the Z-coordinate), if the used calibration pattern is a planar rig. In the old interface all the vectors of object points from different views are concatenated together.
- **imagePoints** In the new interface it is a vector of vectors of the projections of calibration pattern points (e.g. std::vector<std::vector<cv::Vec2f>>). imagePoints.size() and objectPoints.size(), and imagePoints[i].size() and objectPoints[i].size() for each i, must be equal, respectively. In the old interface all the vectors of object points from different views are concatenated together.
- **imageSize** Size of the image used only to initialize the camera intrinsic matrix.
- **cameraMatrix** Input/output 3x3 floating-point camera intrinsic matrix $A = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$. If **CALIB_USE_INTRINSIC_GUESS** and/or **CALIB_FIX_ASPECT_RATIO**, **CALIB_FIX_PRINCIPAL_POINT** or **CALIB_FIX_FOCAL_LENGTH** are specified, some or all of fx, fy, cx, cy must be initialized before calling the function.
- **distCoeffs** Input/output 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.
- **rvecs** Output vector of rotation vectors (**Rodrigues**) estimated for each pattern view (e.g. std::vector<cv::Mat>>). That is, each i-th rotation vector together with the corresponding i-th translation vector (see the next output parameter description) brings the calibration pattern from the object coordinate space (in which object points are specified) to the camera coordinate space. In more technical terms, the tuple of the i-th rotation and translation vector performs a change of basis from object coordinate space to camera coordinate space. Due to its duality, this tuple is equivalent to the position of the calibration pattern with respect to the camera coordinate space.
- **tvecs** Output vector of translation vectors estimated for each pattern view, see parameter describtion above.
- **stdDeviationsIntrinsics** Output vector of standard deviations estimated for intrinsic parameters. Order of deviations values: $(f_x, f_y, c_x, c_y, k_1, k_2, p_1, p_2, k_3, k_4, k_5, k_6, s_1, s_2, s_3, s_4, \tau_x, \tau_y)$ If one of parameters is not estimated, it's deviation is equals to zero.
- **stdDeviationsExtrinsics** Output vector of standard deviations estimated for extrinsic parameters. Order of deviations values: $(R_0, T_0, \dots, R_{M-1}, T_{M-1})$ where M is the number of pattern views. R_i, T_i are concatenated 1x3 vectors.
- **perViewErrors** Output vector of the RMS re-projection error estimated for each pattern view.
- **flags** Different flags that may be zero or a combination of the following values:
  - **CALIB_USE_INTRINSIC_GUESS** cameraMatrix contains valid initial values of fx, fy, cx, cy that are optimized further. Otherwise, (cx, cy) is initially set to the image center (imageSize is used), and focal distances are computed in a least-squares fashion. Note, that if intrinsic parameters are known, there is no need to use this function just to estimate extrinsic parameters. Use **solvePnP** instead.
  - **CALIB_FIX_PRINCIPAL_POINT** The principal point is not changed during the global optimization. It stays at the center or at a different location specified when **CALIB_USE_INTRINSIC_GUESS** is set too.
  - **CALIB_FIX_ASPECT_RATIO** The functions consider only fy as a free parameter. The ratio fx/fy stays the same as in the input cameraMatrix . When **CALIB_USE_INTRINSIC_GUESS** is not set, the actual input values of fx and fy are ignored, only their ratio is computed and used further.
  - **CALIB_ZERO_TANGENT_DIST** Tangential distortion coefficients (p_1, p_2) are set to zeros and stay zero.
  - **CALIB_FIX_FOCAL_LENGTH** The focal length is not changed during the global optimization if **CALIB_USE_INTRINSIC_GUESS** is set.
  - **CALIB_FIX_K1**,..., **CALIB_FIX_K6** The corresponding radial distortion coefficient is not changed during the optimization. If **CALIB_USE_INTRINSIC_GUESS** is set, the coefficient from the supplied distCoeffs matrix is used. Otherwise, it is set to 0.
  - **CALIB_RATIONAL_MODEL** Coefficients k4, k5, and k6 are enabled. To provide the backward compatibility, this extra flag should be explicitly specified to make the calibration function use the rational model and return 8 coefficients or more.
  - **CALIB_THIN_PRISM_MODEL** Coefficients s1, s2, s3 and s4 are enabled. To provide the backward compatibility, this extra flag should be explicitly specified to make the calibration function use the thin prism model and return 12 coefficients or more.
  - **CALIB_FIX_S1_S2_S3_S4** The thin prism distortion coefficients are not changed during the optimization. If **CALIB_USE_INTRINSIC_GUESS** is set, the coefficient from the supplied distCoeffs matrix is used. Otherwise, it is set to 0.
  - **CALIB_TILTED_MODEL** Coefficients tauX and tauY are enabled. To provide the backward compatibility, this extra flag should be explicitly specified to make the calibration function use the tilted sensor model and return 14 coefficients.
  - **CALIB_FIX_TAUX_TAUY** The coefficients of the tilted sensor model are not changed during the optimization. If **CALIB_USE_INTRINSIC_GUESS** is set, the coefficient from the supplied distCoeffs matrix is used. Otherwise, it is set to 0.
- **criteria** Termination criteria for the iterative optimization algorithm.

Returns

the overall RMS re-projection error.

The function estimates the intrinsic camera parameters and extrinsic parameters for each of the views. The algorithm is based on [319] and [37] . The coordinates of 3D object points and their corresponding 2D projections in each view must be specified. That may be achieved by using an object with known geometry and easily detectable feature points. Such an object is called a calibration rig or calibration pattern, and OpenCV has built-in support for a chessboard as a calibration rig (see **findChessboardCorners**). Currently, initialization of intrinsic parameters (when **CALIB_USE_INTRINSIC_GUESS** is not set) is only implemented for planar calibration patterns (where Z-coordinates of the object points must be all zeros). 3D calibration rigs can also be used as long as initial cameraMatrix is provided.

The algorithm performs the following steps:

- Compute the initial intrinsic parameters (the option only available for planar calibration patterns) or read them from the input parameters. The distortion coefficients are all set to zeros initially unless some of CALIB_FIX_K? are specified.
- Estimate the initial camera pose as if the intrinsic parameters have been already known. This is done using **solvePnP** .
- Run the global Levenberg-Marquardt optimization algorithm to minimize the reprojection error, that is, the total sum of squared distances between the observed feature points imagePoints and the projected (using the current estimates for camera parameters and the poses) object points objectPoints. See **projectPoints** for details.

> **Note**
> If you use a non-square (i.e. non-N-by-N) grid and **findChessboardCorners** for calibration, and **calibrateCamera** returns bad values (zero distortion coefficients, c_x and c_y very far from the image center, and/or large differences between f_x and f_y (ratios of 10:1 or more)), then you are probably using patternSize=cvSize(rows,cols) instead of using patternSize=cvSize(cols,rows) in **findChessboardCorners**.
> The function may throw exceptions, if unsupported combination of parameters is provided or the system is underconstrained.

See also

calibrateCameraRO, findChessboardCorners, solvePnP, initCameraMatrix2D, stereoCalibrate, undistort

◆ calibrateCameraRO() [1/2]

```
double cv::calibrateCameraRO ( InputArrayOfArrays   objectPoints,
                               InputArrayOfArrays   imagePoints,
                               Size                 imageSize,
                               int                  iFixedPoint,
                               InputOutputArray     cameraMatrix,
                               InputOutputArray     distCoeffs,
                               OutputArrayOfArrays  rvecs,
                               OutputArrayOfArrays  tvecs,
                               OutputArray          newObjPoints,
                               int                  flags = 0,
                               TermCriteria         criteria = TermCriteria(TermCriteria::COUNT+TermCriteria::EPS, 30, DBL_EPSILON) )
```

Python:

```
cv.calibrateCameraRO( objectPoints, imagePoints, imageSize, iFixedPoint, cameraMatrix, distCoeffs[, rvecs[, tvecs[, newObjPoints[, flags[, criteria]]]]] ) -> retval, cameraMatrix, distCoeffs, rvecs, tvecs, newObjPoints
cv.calibrateCameraROExtended( objectPoints, imagePoints, imageSize, iFixedPoint, cameraMatrix, distCoeffs[, rvecs[, tvecs[, newObjPoints[, stdDeviationsIntrinsics[, stdDeviationsExtrinsics[, stdDeviationsObjPoints[, perViewErrors[, flags[, criteria]]]]]]]]] ) -> retval, cameraMatrix, distCoeffs, rvecs, tvecs, newObjPoints, stdDeviationsIntrinsic stdDeviationsExtrinsic stdDeviationsObjPoin perViewErrors
```

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

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

◆ calibrateCameraRO() [2/2]

```
double cv::calibrateCameraRO ( InputArrayOfArrays   objectPoints,
                               InputArrayOfArrays   imagePoints,
                               Size                 imageSize,
                               int                  iFixedPoint,
                               InputOutputArray     cameraMatrix,
                               InputOutputArray     distCoeffs,
                               OutputArrayOfArrays  rvecs,
                               OutputArrayOfArrays  tvecs,
                               OutputArray          newObjPoints,
                               OutputArray          stdDeviationsIntrinsics,
                               OutputArray          stdDeviationsExtrinsics,
                               OutputArray          stdDeviationsObjPoints,
                               OutputArray          perViewErrors,
                               int                  flags = 0,
                               TermCriteria         criteria = TermCriteria(TermCriteria::COUNT+TermCriteria::EPS, 30, DBL_EPSILON) )
```

Python:

```
cv.calibrateCameraRO( objectPoints, imagePoints, imageSize, iFixedPoint, cameraMatrix, distCoeffs[, rvecs[, tvecs[, newObjPoints[, flags[, criteria]]]]] ) -> retval, cameraMatrix, distCoeffs, rvecs, tvecs, newObjPoints
cv.calibrateCameraROExtended( objectPoints, imagePoints, imageSize, iFixedPoint, cameraMatrix, distCoeffs[, rvecs[, tvecs[, newObjPoints[, stdDeviationsIntrinsics[, stdDeviationsExtrinsics[, stdDeviationsObjPoints[, perViewErrors[, flags[, criteria]]]]]]]]] ) -> retval, cameraMatrix, distCoeffs, rvecs, tvecs, newObjPoints, stdDeviationsIntrinsic stdDeviationsExtrinsic stdDeviationsObjPoin perViewErrors
```

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

Finds the camera intrinsic and extrinsic parameters from several views of a calibration pattern.

This function is an extension of **calibrateCamera** with the method of releasing object which was proposed in [255]. In many common cases with inaccurate, unmeasured, roughly planar targets (calibration plates), this method can dramatically improve the precision of the estimated camera parameters. Both the object-releasing method and standard method are supported by this function. Use the parameter **iFixedPoint** for method selection. In the internal implementation, **calibrateCamera** is a wrapper for this function.

Parameters

objectPoints	Vector of vectors of calibration pattern points in the calibration pattern coordinate space. See **calibrateCamera** for details. If the method of releasing object to be used, the identical calibration board must be used in each view and it must be fully visible, and all objectPoints[i] must be the same and all points should be roughly close to a plane. **The calibration target has to be rigid, or at least static if the camera (rather than the calibration target) is shifted for grabbing images.**
imagePoints	Vector of vectors of the projections of calibration pattern points. See **calibrateCamera** for details.
imageSize	Size of the image used only to initialize the intrinsic camera matrix.
iFixedPoint	The index of the 3D object point in objectPoints[0] to be fixed. It also acts as a switch for calibration method selection. If object-releasing method to be used, pass in the parameter in the range of [1, objectPoints[0].size()-2], otherwise a value out of this range will make standard calibration method selected. Usually the top-right corner point of the calibration board grid is recommended to be fixed when object-releasing method being utilized. According to [255], two other points are also fixed. In this implementation, objectPoints[0].front and objectPoints[0].back.z are used. With object-releasing method, accurate rvecs, tvecs and newObjPoints are only possible if coordinates of these three fixed points are accurate enough.
cameraMatrix	Output 3x3 floating-point camera matrix. See **calibrateCamera** for details.
distCoeffs	Output vector of distortion coefficients. See **calibrateCamera** for details.
rvecs	Output vector of rotation vectors estimated for each pattern view. See **calibrateCamera** for details.
tvecs	Output vector of translation vectors estimated for each pattern view.
newObjPoints	The updated output vector of calibration pattern points. The coordinates might be scaled based on three fixed points. The returned coordinates are accurate only if the above mentioned three fixed points are accurate. If not needed, **noArray()** can be passed in. This parameter is ignored with standard calibration method.
stdDeviationsIntrinsics	Output vector of standard deviations estimated for intrinsic parameters. See **calibrateCamera** for details.
stdDeviationsExtrinsics	Output vector of standard deviations estimated for extrinsic parameters. See **calibrateCamera** for details.
stdDeviationsObjPoints	Output vector of standard deviations estimated for refined coordinates of calibration pattern points. It has the same size and order as objectPoints[0] vector. This parameter is ignored with standard calibration method.
perViewErrors	Output vector of the RMS re-projection error estimated for each pattern view.
flags	Different flags that may be zero or a combination of some predefined values. See **calibrateCamera** for details. If the method of releasing object is used, the calibration time may be much longer. CALIB_USE_QR or CALIB_USE_LU could be used for faster calibration with potentially less precise and less stable in some rare cases.
criteria	Termination criteria for the iterative optimization algorithm.

Returns

the overall RMS re-projection error.

The function estimates the intrinsic camera parameters and extrinsic parameters for each of the views. The algorithm is based on [319], [37] and [255]. See **calibrateCamera** for other detailed explanations.

See also

calibrateCamera, **findChessboardCorners**, **solvePnP**, **initCameraMatrix2D**, **stereoCalibrate**, **undistort**

◆ calibrateHandEye()

```
void cv::calibrateHandEye ( InputArrayOfArrays          R_gripper2base,
                            InputArrayOfArrays          t_gripper2base,
                            InputArrayOfArrays          R_target2cam,
                            InputArrayOfArrays          t_target2cam,
                            OutputArray                 R_cam2gripper,
                            OutputArray                 t_cam2gripper,
                            HandEyeCalibrationMethod    method = CALIB_HAND_EYE_TSAI )
```

Python:

```
cv.calibrateHandEye( R_gripper2base, t_gripper2base, R_target2cam, t_target2cam[, R_cam2gripper[, t_cam2gripper[, method]]] ) -> R_cam2gripper, t_cam2gripper
```

`#include <opencv2/calib3d.hpp>`

Computes Hand-Eye calibration: ${}^{g}\mathrm{T}_c$.

Parameters

`[in]`	**R_gripper2base**	Rotation part extracted from the homogeneous matrix that transforms a point expressed in the gripper frame to the robot base frame (${}^{b}\mathrm{T}_g$). This is a vector (`vector<Mat>`) that contains the rotation, `(3x3)` rotation matrices or `(3x1)` rotation vectors, for all the transformations from gripper frame to robot base frame.
`[in]`	**t_gripper2base**	Translation part extracted from the homogeneous matrix that transforms a point expressed in the gripper frame to the robot base frame (${}^{b}\mathrm{T}_g$). This is a vector (`vector<Mat>`) that contains the `(3x1)` translation vectors for all the transformations from gripper frame to robot base frame.
`[in]`	**R_target2cam**	Rotation part extracted from the homogeneous matrix that transforms a point expressed in the target frame to the camera frame (${}^{c}\mathrm{T}_t$). This is a vector (`vector<Mat>`) that contains the rotation, `(3x3)` rotation matrices or `(3x1)` rotation vectors, for all the transformations from calibration target frame to camera frame.
`[in]`	**t_target2cam**	Rotation part extracted from the homogeneous matrix that transforms a point expressed in the target frame to the camera frame (${}^{c}\mathrm{T}_t$). This is a vector (`vector<Mat>`) that contains the `(3x1)` translation vectors for all the transformations from calibration target frame to camera frame.
`[out]`	**R_cam2gripper**	Estimated `(3x3)` rotation part extracted from the homogeneous matrix that transforms a point expressed in the camera frame to the gripper frame (${}^{g}\mathrm{T}_c$).
`[out]`	**t_cam2gripper**	Estimated `(3x1)` translation part extracted from the homogeneous matrix that transforms a point expressed in the camera frame to the gripper frame (${}^{g}\mathrm{T}_c$).
`[in]`	**method**	One of the implemented Hand-Eye calibration method, see **cv::HandEyeCalibrationMethod**

The function performs the Hand-Eye calibration using various methods. One approach consists in estimating the rotation then the translation (separable solutions) and the following methods are implemented:

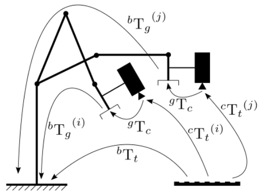
- R. Tsai, R. Lenz A New Technique for Fully Autonomous and Efficient 3D Robotics Hand/EyeCalibration [274]
- F. Park, B. Martin Robot Sensor Calibration: Solving AX = XB on the Euclidean Group [213]
- R. Horaud, F. Dornaika Hand-Eye Calibration [126]

Another approach consists in estimating simultaneously the rotation and the translation (simultaneous solutions), with the following implemented methods:

- N. Andreff, R. Horaud, B. Espiau On-line Hand-Eye Calibration [12]
- K. Daniilidis Hand-Eye Calibration Using Dual Quaternions [65]

The following picture describes the Hand-Eye calibration problem where the transformation between a camera ("eye") mounted on a robot gripper ("hand") has to be estimated. This configuration is called eye-in-hand.

The eye-to-hand configuration consists in a static camera observing a calibration pattern mounted on the robot end-effector. The transformation from the camera to the robot base frame can then be estimated by inputting the suitable transformations to the function, see below.



The calibration procedure is the following:

- a static calibration pattern is used to estimate the transformation between the target frame and the camera frame
- the robot gripper is moved in order to acquire several poses
- for each pose, the homogeneous transformation between the gripper frame and the robot base frame is recorded using for instance the robot kinematics

$$\begin{bmatrix} X_b \\ Y_b \\ Z_b \\ 1 \end{bmatrix} = \begin{bmatrix} {}^{b}\mathrm{R}_g & {}^{b}\mathrm{t}_g \\ 0_{1\times 3} & 1 \end{bmatrix} \begin{bmatrix} X_g \\ Y_g \\ Z_g \\ 1 \end{bmatrix}$$

- for each pose, the homogeneous transformation between the calibration target frame and the camera frame is recorded using for instance a pose estimation method (PnP) from 2D-3D point correspondences

$$\begin{bmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{bmatrix} = \begin{bmatrix} {}^{c}\mathrm{R}_t & {}^{c}\mathrm{t}_t \\ 0_{1\times 3} & 1 \end{bmatrix} \begin{bmatrix} X_t \\ Y_t \\ Z_t \\ 1 \end{bmatrix}$$

The Hand-Eye calibration procedure returns the following homogeneous transformation

$$\begin{bmatrix} X_g \\ Y_g \\ Z_g \\ 1 \end{bmatrix} = \begin{bmatrix} {}^{g}\mathrm{R}_c & {}^{g}\mathrm{t}_c \\ 0_{1\times 3} & 1 \end{bmatrix} \begin{bmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{bmatrix}$$

This problem is also known as solving the $\mathbf{AX} = \mathbf{XB}$ equation:

- for an eye-in-hand configuration

$$\begin{array}{c} {}^{b}{\mathrm{T}_g}^{(1)}\,{}^{g}\mathrm{T}_c\,{}^{c}{\mathrm{T}_t}^{(1)} = {}^{b}{\mathrm{T}_g}^{(2)}\,{}^{g}\mathrm{T}_c\,{}^{c}{\mathrm{T}_t}^{(2)} \\ ({}^{b}{\mathrm{T}_g}^{(2)})^{-1}\,{}^{b}{\mathrm{T}_g}^{(1)}\,{}^{g}\mathrm{T}_c = {}^{g}\mathrm{T}_c\,{}^{c}{\mathrm{T}_t}^{(2)}({}^{c}{\mathrm{T}_t}^{(1)})^{-1} \\ \mathrm{A}_i\mathrm{X} = \mathrm{X}\mathrm{B}_i \end{array}$$

- for an eye-to-hand configuration

$$\begin{array}{c} {}^{g}{\mathrm{T}_b}^{(1)}\,{}^{b}\mathrm{T}_c\,{}^{c}{\mathrm{T}_t}^{(1)} = {}^{g}{\mathrm{T}_b}^{(2)}\,{}^{b}\mathrm{T}_c\,{}^{c}{\mathrm{T}_t}^{(2)} \\ ({}^{g}{\mathrm{T}_b}^{(2)})^{-1}\,{}^{g}{\mathrm{T}_b}^{(1)}\,{}^{b}\mathrm{T}_c = {}^{b}\mathrm{T}_c\,{}^{c}{\mathrm{T}_t}^{(2)}({}^{c}{\mathrm{T}_t}^{(1)})^{-1} \\ \mathrm{A}_i\mathrm{X} = \mathrm{X}\mathrm{B}_i \end{array}$$

Note
Additional information can be found on this website.
A minimum of 2 motions with non parallel rotation axes are necessary to determine the hand-eye transformation. So at least 3 different poses are required, but it is strongly recommended to use many more poses.

◆ calibrateRobotWorldHandEye()

```
void cv::calibrateRobotWorldHandEye ( InputArrayOfArrays  R_world2cam,
                                      InputArrayOfArrays  t_world2cam,
                                      InputArrayOfArrays  R_base2gripper,
                                      InputArrayOfArrays  t_base2gripper,
                                      OutputArray         R_base2world,
                                      OutputArray         t_base2world,
                                      OutputArray         R_gripper2cam,
                                      OutputArray         t_gripper2cam,
                                      RobotWorldHandEyeCalibrationMethod method = CALIB_ROBOT_WORLD_HAND_EYE_SHAH )
```

Python:

```
cv.calibrateRobotWorldHandEye( R_world2cam, t_world2cam, R_base2gripper, t_base2gripper[, R_base2world[, t_base2world[, R_gripper2cam[, t_gripper2cam[, method]]]]] ) -> R_base2world, t_base2world, R_gripper2cam, t_gripper2cam
```

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

Computes Robot-World/Hand-Eye calibration: ${}^{w}\mathrm{T}_b$ and ${}^{c}\mathrm{T}_g$.

Parameters

[in]	**R_world2cam**	Rotation part extracted from the homogeneous matrix that transforms a point expressed in the world frame to the camera frame (${}^{c}\mathrm{T}_w$). This is a vector (`vector<Mat>`) that contains the rotation, `(3x3)` rotation matrices or `(3x1)` rotation vectors, for all the transformations from world frame to the camera frame.
[in]	**t_world2cam**	Translation part extracted from the homogeneous matrix that transforms a point expressed in the world frame to the camera frame (${}^{c}\mathrm{T}_w$). This is a vector (`vector<Mat>`) that contains the `(3x1)` translation vectors for all the transformations from world frame to the camera frame.
[in]	**R_base2gripper**	Rotation part extracted from the homogeneous matrix that transforms a point expressed in the robot base frame to the gripper frame (${}^{g}\mathrm{T}_b$). This is a vector (`vector<Mat>`) that contains the rotation, `(3x3)` rotation matrices or `(3x1)` rotation vectors, for all the transformations from robot base frame to the gripper frame.
[in]	**t_base2gripper**	Rotation part extracted from the homogeneous matrix that transforms a point expressed in the robot base frame to the gripper frame (${}^{g}\mathrm{T}_b$). This is a vector (`vector<Mat>`) that contains the `(3x1)` translation vectors for all the transformations from robot base frame to the gripper frame.
[out]	**R_base2world**	Estimated `(3x3)` rotation part extracted from the homogeneous matrix that transforms a point expressed in the robot base frame to the world frame (${}^{w}\mathrm{T}_b$).
[out]	**t_base2world**	Estimated `(3x1)` translation part extracted from the homogeneous matrix that transforms a point expressed in the robot base frame to the world frame (${}^{w}\mathrm{T}_b$).
[out]	**R_gripper2cam**	Estimated `(3x3)` rotation part extracted from the homogeneous matrix that transforms a point expressed in the gripper frame to the camera frame (${}^{c}\mathrm{T}_g$).
[out]	**t_gripper2cam**	Estimated `(3x1)` translation part extracted from the homogeneous matrix that transforms a point expressed in the gripper frame to the camera frame (${}^{c}\mathrm{T}_g$).
[in]	**method**	One of the implemented Robot-World/Hand-Eye calibration method, see **cv::RobotWorldHandEyeCalibrationMethod**

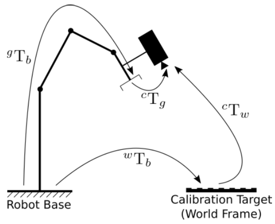
The function performs the Robot-World/Hand-Eye calibration using various methods. One approach consists in estimating the rotation then the translation (separable solutions):

- M. Shah, Solving the robot-world/hand-eye calibration problem using the kronecker product [244]

Another approach consists in estimating simultaneously the rotation and the translation (simultaneous solutions), with the following implemented method:

- A. Li, L. Wang, and D. Wu, Simultaneous robot-world and hand-eye calibration using dual-quaternions and kronecker product [162]

The following picture describes the Robot-World/Hand-Eye calibration problem where the transformations between a robot and a world frame and between a robot gripper ("hand") and a camera ("eye") mounted at the robot end-effector have to be estimated.



The calibration procedure is the following:

- a static calibration pattern is used to estimate the transformation between the target frame and the camera frame
- the robot gripper is moved in order to acquire several poses
- for each pose, the homogeneous transformation between the gripper frame and the robot base frame is recorded using for instance the robot kinematics

$$\begin{bmatrix} X_g \\ Y_g \\ Z_g \\ 1 \end{bmatrix} = \begin{bmatrix} {}^{g}\mathrm{R}_b & {}^{g}\mathrm{t}_b \\ 0_{1\times 3} & 1 \end{bmatrix} \begin{bmatrix} X_b \\ Y_b \\ Z_b \\ 1 \end{bmatrix}$$

- for each pose, the homogeneous transformation between the calibration target frame (the world frame) and the camera frame is recorded using for instance a pose estimation method (PnP) from 2D-3D point correspondences

$$\begin{bmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{bmatrix} = \begin{bmatrix} {}^{c}\mathrm{R}_w & {}^{c}\mathrm{t}_w \\ 0_{1\times 3} & 1 \end{bmatrix} \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix}$$

The Robot-World/Hand-Eye calibration procedure returns the following homogeneous transformations

$$\begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix} = \begin{bmatrix} {}^{w}\mathrm{R}_b & {}^{w}\mathrm{t}_b \\ 0_{1\times 3} & 1 \end{bmatrix} \begin{bmatrix} X_b \\ Y_b \\ Z_b \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{bmatrix} = \begin{bmatrix} {}^{c}\mathrm{R}_g & {}^{c}\mathrm{t}_g \\ 0_{1\times 3} & 1 \end{bmatrix} \begin{bmatrix} X_g \\ Y_g \\ Z_g \\ 1 \end{bmatrix}$$

This problem is also known as solving the $\mathbf{A}\mathbf{X}=\mathbf{Z}\mathbf{B}$ equation, with:

- $\mathbf{A} \Leftrightarrow {}^{c}\mathrm{T}_w$
- $\mathbf{X} \Leftrightarrow {}^{w}\mathrm{T}_b$
- $\mathbf{Z} \Leftrightarrow {}^{c}\mathrm{T}_g$
- $\mathbf{B} \Leftrightarrow {}^{g}\mathrm{T}_b$

Note
At least 3 measurements are required (input vectors size must be greater or equal to 3).

◆ calibrationMatrixValues()

```
void cv::calibrationMatrixValues ( InputArray  cameraMatrix,
                                   Size        imageSize,
                                   double      apertureWidth,
                                   double      apertureHeight,
                                   double &    fovx,
                                   double &    fovy,
                                   double &    focalLength,
                                   Point2d &   principalPoint,
                                   double &    aspectRatio )
```

Python:

```
cv.calibrationMatrixValues( cameraMatrix, imageSize, apertureWidth, apertureHeight ) -> fovx, fovy, focalLength, principalPoint, aspectRatio
```

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

Computes useful camera characteristics from the camera intrinsic matrix.

Parameters

cameraMatrix	Input camera intrinsic matrix that can be estimated by **calibrateCamera** or **stereoCalibrate** .
imageSize	Input image size in pixels.
apertureWidth	Physical width in mm of the sensor.
apertureHeight	Physical height in mm of the sensor.
fovx	Output field of view in degrees along the horizontal sensor axis.
fovy	Output field of view in degrees along the vertical sensor axis.
focalLength	Focal length of the lens in mm.
principalPoint	Principal point in mm.
aspectRatio	f_y/f_x

The function computes various useful camera characteristics from the previously estimated camera matrix.

Note
Do keep in mind that the unity measure 'mm' stands for whatever unit of measure one chooses for the chessboard pitch (it can thus be any value).

◆ checkChessboard()

```
bool cv::checkChessboard ( InputArray  img,
                           Size        size )
```

Python:

```
cv.checkChessboard( img, size ) -> retval
```

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

◆ composeRT()

```
void cv::composeRT ( InputArray  rvec1,
                     InputArray  tvec1,
                     InputArray  rvec2,
                     InputArray  tvec2,
                     OutputArray rvec3,
                     OutputArray tvec3,
                     OutputArray dr3dr1 = noArray(),
                     OutputArray dr3dt1 = noArray(),
                     OutputArray dr3dr2 = noArray(),
                     OutputArray dr3dt2 = noArray(),
                     OutputArray dt3dr1 = noArray(),
                     OutputArray dt3dt1 = noArray(),
                     OutputArray dt3dr2 = noArray(),
                     OutputArray dt3dt2 = noArray() )
```

Python:

```
cv.composeRT( rvec1, tvec1, rvec2, tvec2[, rvec3[, tvec3[, dr3dr1[, dr3dt1[, dr3dr2[, dr3dt2[, dt3dr1[, dt3dt1[, dt3dr2[, dt3dt2]]]]]]]]]] ) -> rvec3, tvec3, dr3dr1, dr3dt1, dr3dr2, dr3dt2, dt3dr1, dt3dt1, dt3dr2, dt3dt2
```

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

Combines two rotation-and-shift transformations.

Parameters

- **rvec1** First rotation vector.
- **tvec1** First translation vector.
- **rvec2** Second rotation vector.
- **tvec2** Second translation vector.
- **rvec3** Output rotation vector of the superposition.
- **tvec3** Output translation vector of the superposition.
- **dr3dr1** Optional output derivative of rvec3 with regard to rvec1
- **dr3dt1** Optional output derivative of rvec3 with regard to tvec1
- **dr3dr2** Optional output derivative of rvec3 with regard to rvec2
- **dr3dt2** Optional output derivative of rvec3 with regard to tvec2
- **dt3dr1** Optional output derivative of tvec3 with regard to rvec1
- **dt3dt1** Optional output derivative of tvec3 with regard to tvec1
- **dt3dr2** Optional output derivative of tvec3 with regard to rvec2
- **dt3dt2** Optional output derivative of tvec3 with regard to tvec2

The functions compute:

$$\begin{array}{l} \texttt{rvec3} = \mathrm{rodrigues}^{-1} (\mathrm{rodrigues} (\texttt{rvec2}) \cdot \mathrm{rodrigues} (\texttt{rvec1})) \\ \texttt{tvec3} = \mathrm{rodrigues} (\texttt{rvec2}) \cdot \texttt{tvec1} + \texttt{tvec2} \end{array} ,$$

where $\mathrm{rodrigues}$ denotes a rotation vector to a rotation matrix transformation, and $\mathrm{rodrigues}^{-1}$ denotes the inverse transformation. See **Rodrigues** for details.

Also, the functions can compute the derivatives of the output vectors with regards to the input vectors (see **matMulDeriv**). The functions are used inside **stereoCalibrate** but can also be used in your own code where Levenberg-Marquardt or another gradient-based solver is used to optimize a function that contains a matrix multiplication.

◆ computeCorrespondEpilines()

```
void cv::computeCorrespondEpilines ( InputArray  points,
                                     int         whichImage,
                                     InputArray  F,
                                     OutputArray lines )
```

Python:

```
cv.computeCorrespondEpilines( points, whichImage, F[, lines] ) -> lines
```

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

For points in an image of a stereo pair, computes the corresponding epilines in the other image.

Parameters

- **points** Input points. $N \times 1$ or $1 \times N$ matrix of type CV_32FC2 or vector<Point2f> .
- **whichImage** Index of the image (1 or 2) that contains the points .
- **F** Fundamental matrix that can be estimated using **findFundamentalMat** or **stereoRectify** .
- **lines** Output vector of the epipolar lines corresponding to the points in the other image. Each line $ax + by + c = 0$ is encoded by 3 numbers (a, b, c) .

For every point in one of the two images of a stereo pair, the function finds the equation of the corresponding epipolar line in the other image.

From the fundamental matrix definition (see **findFundamentalMat**), line $l^{(2)}_i$ in the second image for the point $p^{(1)}_i$ in the first image (when whichImage=1) is computed as:

$$l^{(2)}_i = F p^{(1)}_i$$

And vice versa, when whichImage=2, $l^{(1)}_i$ is computed from $p^{(2)}_i$ as:

$$l^{(1)}_i = F^T p^{(2)}_i$$

Line coefficients are defined up to a scale. They are normalized so that $a_i^2 + b_i^2 = 1$.

◆ convertPointsFromHomogeneous()

```
void cv::convertPointsFromHomogeneous ( InputArray  src,
                                        OutputArray dst )
```

Python:

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

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

Converts points from homogeneous to Euclidean space.

Parameters

- **src** Input vector of N-dimensional points.
- **dst** Output vector of N-1-dimensional points.

The function converts points homogeneous to Euclidean space using perspective projection. That is, each point (x1, x2, ... x(n-1), xn) is converted to (x1/xn, x2/xn, ..., x(n-1)/xn). When xn=0, the output point coordinates will be (0,0,0,...).

◆ convertPointsHomogeneous()

```
void cv::convertPointsHomogeneous ( InputArray  src,
                                    OutputArray dst )
```

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

Converts points to/from homogeneous coordinates.

Parameters

- **src** Input array or vector of 2D, 3D, or 4D points.
- **dst** Output vector of 2D, 3D, or 4D points.

The function converts 2D or 3D points from/to homogeneous coordinates by calling either **convertPointsToHomogeneous** or **convertPointsFromHomogeneous**.

> **Note**
> The function is obsolete. Use one of the previous two functions instead.

◆ convertPointsToHomogeneous()

```
void cv::convertPointsToHomogeneous ( InputArray  src,
                                      OutputArray dst )
```

Python:

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

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

Converts points from Euclidean to homogeneous space.

Parameters

- **src** Input vector of N-dimensional points.
- **dst** Output vector of N+1-dimensional points.

The function converts points from Euclidean to homogeneous space by appending 1's to the tuple of point coordinates. That is, each point (x1, x2, ..., xn) is converted to (x1, x2, ..., xn, 1).

◆ correctMatches()

```
void cv::correctMatches ( InputArray  F,
                          InputArray  points1,
                          InputArray  points2,
                          OutputArray newPoints1,
                          OutputArray newPoints2 )
```

Python:

```
cv.correctMatches( F, points1, points2[, newPoints1[, newPoints2]] ) -> newPoints1, newPoints2
```

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

Refines coordinates of corresponding points.

Parameters

F	3x3 fundamental matrix.
points1	1xN array containing the first set of points.
points2	1xN array containing the second set of points.
newPoints1	The optimized points1.
newPoints2	The optimized points2.

The function implements the Optimal Triangulation Method (see Multiple View Geometry [116] for details). For each given point correspondence points1[i] <-> points2[i], and a fundamental matrix F, it computes the corrected correspondences newPoints1[i] <-> newPoints2[i] that minimize the geometric error $d(points1[i], newPoints1[i])^2 + d(points2[i], newPoints2[i])^2$ (where $d(a, b)$ is the geometric distance between points a and b) subject to the epipolar constraint $newPoints2^T \cdot F \cdot newPoints1 = 0$.

◆ decomposeEssentialMat()

```
void cv::decomposeEssentialMat ( InputArray  E,
                                 OutputArray R1,
                                 OutputArray R2,
                                 OutputArray t )
```

Python:

```
cv.decomposeEssentialMat( E[, R1[, R2[, t]]] ) -> R1, R2, t
```

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

Decompose an essential matrix to possible rotations and translation.

Parameters

E	The input essential matrix.
R1	One possible rotation matrix.
R2	Another possible rotation matrix.
t	One possible translation.

This function decomposes the essential matrix E using svd decomposition [116]. In general, four possible poses exist for the decomposition of E. They are $[R_1, t]$, $[R_1, -t]$, $[R_2, t]$, $[R_2, -t]$.

If E gives the epipolar constraint $[p_2; 1]^T A^{-T} E A^{-1} [p_1; 1] = 0$ between the image points p_1 in the first image and p_2 in second image, then any of the tuples $[R_1, t]$, $[R_1, -t]$, $[R_2, t]$, $[R_2, -t]$ is a change of basis from the first camera's coordinate system to the second camera's coordinate system. However, by decomposing E, one can only get the direction of the translation. For this reason, the translation t is returned with unit length.

◆ decomposeHomographyMat()

```
int cv::decomposeHomographyMat ( InputArray           H,
                                 InputArray           K,
                                 OutputArrayOfArrays rotations,
                                 OutputArrayOfArrays translations,
                                 OutputArrayOfArrays normals )
```

Python:

```
cv.decomposeHomographyMat( H, K[, rotations[, translations[, normals]]] ) -> retval, rotations, translations, normals
```

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

Decompose a homography matrix to rotation(s), translation(s) and plane normal(s).

Parameters

H	The input homography matrix between two images.
K	The input camera intrinsic matrix.
rotations	Array of rotation matrices.
translations	Array of translation matrices.
normals	Array of plane normal matrices.

This function extracts relative camera motion between two views of a planar object and returns up to four mathematical solution tuples of rotation, translation, and plane normal. The decomposition of the homography matrix H is described in detail in [180].

If the homography H, induced by the plane, gives the constraint

$$s_i \begin{bmatrix} x'_i \\ y'_i \\ 1 \end{bmatrix} \sim H \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix}$$

on the source image points p_i and the destination image points p'_i, then the tuple of rotations[k] and translations[k] is a change of basis from the source camera's coordinate system to the destination camera's coordinate system. However, by decomposing H, one can only get the translation normalized by the (typically unknown) depth of the scene, i.e. its direction but with normalized length.

If point correspondences are available, at least two solutions may further be invalidated, by applying positive depth constraint, i.e. all points must be in front of the camera.

◆ decomposeProjectionMatrix()

```
void cv::decomposeProjectionMatrix ( InputArray  projMatrix,
                                     OutputArray cameraMatrix,
                                     OutputArray rotMatrix,
                                     OutputArray transVect,
                                     OutputArray rotMatrixX = noArray(),
                                     OutputArray rotMatrixY = noArray(),
                                     OutputArray rotMatrixZ = noArray(),
                                     OutputArray eulerAngles = noArray() )
```

Python:

```
cv.decomposeProjectionMatrix( projMatrix[, cameraMatrix[, rotMatrix[, transVect[, rotMatrixX[, rotMatrixY[, rotMatrixZ[, eulerAngles]]]]]]] ) -> cameraMatrix, rotMatrix, transVect, rotMatrixX, rotMatrixY, rotMatrixZ, eulerAngles
```

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

Decomposes a projection matrix into a rotation matrix and a camera intrinsic matrix.

Parameters

projMatrix	3x4 input projection matrix P.
cameraMatrix	Output 3x3 camera intrinsic matrix $A = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$.
rotMatrix	Output 3x3 external rotation matrix R.
transVect	Output 4x1 translation vector T.
rotMatrixX	Optional 3x3 rotation matrix around x-axis.
rotMatrixY	Optional 3x3 rotation matrix around y-axis.
rotMatrixZ	Optional 3x3 rotation matrix around z-axis.
eulerAngles	Optional three-element vector containing three Euler angles of rotation in degrees.

The function computes a decomposition of a projection matrix into a calibration and a rotation matrix and the position of a camera.

It optionally returns three rotation matrices, one for each axis, and three Euler angles that could be used in OpenGL. Note, there is always more than one sequence of rotations about the three principal axes that results in the same orientation of an object, e.g. see [248] . Returned three rotation matrices and corresponding three Euler angles are only one of the possible solutions.

The function is based on RQDecomp3x3 .

◆ drawChessboardCorners()

```
void cv::drawChessboardCorners ( InputOutputArray image,
                                 Size             patternSize,
                                 InputArray       corners,
                                 bool             patternWasFound )
```

Python:

```
cv.drawChessboardCorners( image, patternSize, corners, patternWasFound ) -> image
```

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

Renders the detected chessboard corners.

Parameters

image	Destination image. It must be an 8-bit color image.
patternSize	Number of inner corners per a chessboard row and column (patternSize = cv::Size(points_per_row,points_per_column)).
corners	Array of detected corners, the output of findChessboardCorners.
patternWasFound	Parameter indicating whether the complete board was found or not. The return value of findChessboardCorners should be passed here.

The function draws individual chessboard corners detected either as red circles if the board was not found, or as colored corners connected with lines if the board was found.

◆ drawFrameAxes()

```
void cv::drawFrameAxes ( InputOutputArray image,
                         InputArray       cameraMatrix,
                         InputArray       distCoeffs,
                         InputArray       rvec,
                         InputArray       tvec,
                         float            length,
                         int              thickness = 3 )
```

Python:

```
cv.drawFrameAxes( image, cameraMatrix, distCoeffs, rvec, tvec, length[, thickness] ) -> image
```

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

Draw axes of the world/object coordinate system from pose estimation.

See also

solvePnP

Parameters

image	Input/output image. It must have 1 or 3 channels. The number of channels is not altered.
cameraMatrix	Input 3x3 floating-point matrix of camera intrinsic parameters. $A = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$
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.
rvec	Rotation vector (see Rodrigues) that, together with tvec, brings points from the model coordinate system to the camera coordinate system.
tvec	Translation vector.
length	Length of the painted axes in the same unit than tvec (usually in meters).
thickness	Line thickness of the painted axes.

This function draws the axes of the world/object coordinate system w.r.t. to the camera frame. OX is drawn in red, OY in green and OZ in blue.

◆ estimateAffine2D() [1/2]

```
cv::Mat cv::estimateAffine2D ( InputArray  from,
                               InputArray  to,
                               OutputArray inliers = noArray(),
                               int         method = RANSAC,
                               double      ransacReprojThreshold = 3,
                               size_t      maxIters = 2000,
                               double      confidence = 0.99,
                               size_t      refineIters = 10 )
```

Python:

```
cv.estimateAffine2D( from_, to[, inliers[, method[, ransacReprojThreshold[, maxIters[, confidence[, refineIters]]]]]] ) -> retval, inliers
cv.estimateAffine2D( pts1, pts2, params[, inliers]                                                                   ) -> retval, inliers
```

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

Computes an optimal affine transformation between two 2D point sets.

It computes

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} X \\ Y \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$$

Parameters

from	First input 2D point set containing (X, Y).
to	Second input 2D point set containing (x, y).
inliers	Output vector indicating which points are inliers (1-inlier, 0-outlier).
method	Robust method used to compute transformation. The following methods are possible: • RANSAC - RANSAC-based robust method • LMEDS - Least-Median robust method RANSAC is the default method.
ransacReprojThreshold	Maximum reprojection error in the RANSAC algorithm to consider a point as an inlier. Applies only to RANSAC.
maxIters	The maximum number of robust method iterations.
confidence	Confidence level, between 0 and 1, for the estimated transformation. Anything between 0.95 and 0.99 is usually good enough. Values too close to 1 can slow down the estimation significantly. Values lower than 0.8-0.9 can result in an incorrectly estimated transformation.
refineIters	Maximum number of iterations of refining algorithm (Levenberg-Marquardt). Passing 0 will disable refining, so the output matrix will be output of robust method.

Returns

Output 2D affine transformation matrix 2×3 or empty matrix if transformation could not be estimated. The returned matrix has the following form:

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

The function estimates an optimal 2D affine transformation between two 2D point sets using the selected robust algorithm.

The computed transformation is then refined further (using only inliers) with the Levenberg-Marquardt method to reduce the re-projection error even more.

> **Note**
> The RANSAC method can handle practically any ratio of outliers but needs a threshold to distinguish inliers from outliers. The method LMeDS does not need any threshold but it works correctly only when there are more than 50% of inliers.

See also

estimateAffinePartial2D, getAffineTransform

◆ estimateAffine2D() [2/2]

```
cv::Mat cv::estimateAffine2D ( InputArray           pts1,
                               InputArray           pts2,
                               OutputArray          inliers,
                               const UsacParams &   params )
```

Python:

```
cv.estimateAffine2D( from_, to[, inliers[, method[, ransacReprojThreshold[, maxIters[, confidence[, refineIters]]]]]] ) -> retval, inliers
cv.estimateAffine2D( pts1, pts2, params[, inliers]                                                                   ) -> retval, inliers
```

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

◆ estimateAffine3D() [1/2]

```
cv::Mat cv::estimateAffine3D ( InputArray src,
                               InputArray dst,
                               double *   scale = nullptr,
                               bool       force_rotation = true )
```

Python:

```
cv.estimateAffine3D( src, dst[, out[, inliers[, ransacThreshold[, confidence]]]] ) -> retval, out, inliers
cv.estimateAffine3D( src, dst[, force_rotation]                                  ) -> retval, scale
```

`#include <opencv2/calib3d.hpp>`

Computes an optimal affine transformation between two 3D point sets.

It computes R, s, t minimizing $\sum idst_i - c \cdot R \cdot src_i$ where R is a 3x3 rotation matrix, t is a 3x1 translation vector and s is a scalar size value. This is an implementation of the algorithm by Umeyama [278] . The estimated affine transform has a homogeneous scale which is a subclass of affine transformations with 7 degrees of freedom. The paired point sets need to comprise at least 3 points each.

Parameters

src	First input 3D point set.
dst	Second input 3D point set.
scale	If null is passed, the scale parameter c will be assumed to be 1.0. Else the pointed-to variable will be set to the optimal scale.
force_rotation	If true, the returned rotation will never be a reflection. This might be unwanted, e.g. when optimizing a transform between a right- and a left-handed coordinate system.

Returns

3D affine transformation matrix 3×4 of the form

$$T = \begin{bmatrix} R & t \end{bmatrix}$$

◆ estimateAffine3D() [2/2]

```
int cv::estimateAffine3D ( InputArray  src,
                           InputArray  dst,
                           OutputArray out,
                           OutputArray inliers,
                           double      ransacThreshold = 3,
                           double      confidence = 0.99 )
```

Python:

```
cv.estimateAffine3D( src, dst[, out[, inliers[, ransacThreshold[, confidence]]]] ) -> retval, out, inliers
cv.estimateAffine3D( src, dst[, force_rotation]                                  ) -> retval, scale
```

`#include <opencv2/calib3d.hpp>`

Computes an optimal affine transformation between two 3D point sets.

It computes

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

Parameters

src	First input 3D point set containing (X, Y, Z).
dst	Second input 3D point set containing (x, y, z).
out	Output 3D affine transformation matrix 3×4 of the form $$\begin{bmatrix} a_{11} & a_{12} & a_{13} & b_1 \\ a_{21} & a_{22} & a_{23} & b_2 \\ a_{31} & a_{32} & a_{33} & b_3 \end{bmatrix}$$
inliers	Output vector indicating which points are inliers (1-inlier, 0-outlier).
ransacThreshold	Maximum reprojection error in the RANSAC algorithm to consider a point as an inlier.
confidence	Confidence level, between 0 and 1, for the estimated transformation. Anything between 0.95 and 0.99 is usually good enough. Values too close to 1 can slow down the estimation significantly. Values lower than 0.8-0.9 can result in an incorrectly estimated transformation.

The function estimates an optimal 3D affine transformation between two 3D point sets using the RANSAC algorithm.

◆ estimateAffinePartial2D()

```
cv::Mat cv::estimateAffinePartial2D ( InputArray  from,
                                      InputArray  to,
                                      OutputArray inliers = noArray(),
                                      int         method = RANSAC,
                                      double      ransacReprojThreshold = 3,
                                      size_t      maxIters = 2000,
                                      double      confidence = 0.99,
                                      size_t      refineIters = 10 )
```

Python:

```
cv.estimateAffinePartial2D( from_, to[, inliers[, method[, ransacReprojThreshold[, maxIters[, confidence[, refineIters]]]]]] ) -> retval, inliers
```

`#include <opencv2/calib3d.hpp>`

Computes an optimal limited affine transformation with 4 degrees of freedom between two 2D point sets.

Parameters

from	First input 2D point set.
to	Second input 2D point set.
inliers	Output vector indicating which points are inliers.
method	Robust method used to compute transformation. The following methods are possible: • **RANSAC** - RANSAC-based robust method • **LMEDS** - Least-Median robust method RANSAC is the default method.
ransacReprojThreshold	Maximum reprojection error in the RANSAC algorithm to consider a point as an inlier. Applies only to RANSAC.
maxIters	The maximum number of robust method iterations.
confidence	Confidence level, between 0 and 1, for the estimated transformation. Anything between 0.95 and 0.99 is usually good enough. Values too close to 1 can slow down the estimation significantly. Values lower than 0.8-0.9 can result in an incorrectly estimated transformation.
refineIters	Maximum number of iterations of refining algorithm (Levenberg-Marquardt). Passing 0 will disable refining, so the output matrix will be output of robust method.

Returns

Output 2D affine transformation (4 degrees of freedom) matrix 2×3 or empty matrix if transformation could not be estimated.

The function estimates an optimal 2D affine transformation with 4 degrees of freedom limited to combinations of translation, rotation, and uniform scaling. Uses the selected algorithm for robust estimation.

The computed transformation is then refined further (using only inliers) with the Levenberg-Marquardt method to reduce the re-projection error even more.

Estimated transformation matrix is:

$$\begin{bmatrix} \cos(\theta) \cdot s & -\sin(\theta) \cdot s & t_x \\ \sin(\theta) \cdot s & \cos(\theta) \cdot s & t_y \end{bmatrix}$$

Where θ is the rotation angle, s the scaling factor and t_x, t_y are translations in x, y axes respectively.

> **Note**
> The RANSAC method can handle practically any ratio of outliers but need a threshold to distinguish inliers from outliers. The method LMeDS does not need any threshold but it works correctly only when there are more than 50% of inliers.

See also

estimateAffine2D, getAffineTransform

◆ estimateChessboardSharpness()

```
Scalar cv::estimateChessboardSharpness ( InputArray   image,
                                         Size         patternSize,
                                         InputArray   corners,
                                         float        rise_distance = 0.8F,
                                         bool         vertical = false,
                                         OutputArray  sharpness = noArray() )
```

Python:

cv.estimateChessboardSharpness(image, patternSize, corners[, rise_distance[, vertical[, sharpness]]]) -> retval, sharpness

`#include <opencv2/calib3d.hpp>`

Estimates the sharpness of a detected chessboard.

Image sharpness, as well as brightness, are a critical parameter for accuracte camera calibration. For accessing these parameters for filtering out problematic calibraiton images, this method calculates edge profiles by traveling from black to white chessboard cell centers. Based on this, the number of pixels is calculated required to transit from black to white. This width of the transition area is a good indication of how sharp the chessboard is imaged and should be below ~3.0 pixels.

Parameters

image	Gray image used to find chessboard corners
patternSize	Size of a found chessboard pattern
corners	Corners found by findChessboardCornersSB
rise_distance	Rise distance 0.8 means 10% ... 90% of the final signal strength
vertical	By default edge responses for horizontal lines are calculated
sharpness	Optional output array with a sharpness value for calculated edge responses (see description)

The optional sharpness array is of type CV_32FC1 and has for each calculated profile one row with the following five entries: 0 = x coordinate of the underlying edge in the image 1 = y coordinate of the underlying edge in the image 2 = width of the transition area (sharpness) 3 = signal strength in the black cell (min brightness) 4 = signal strength in the white cell (max brightness)

Returns

Scalar(average sharpness, average min brightness, average max brightness,0)

◆ estimateTranslation3D()

```
int cv::estimateTranslation3D ( InputArray   src,
                                InputArray   dst,
                                OutputArray  out,
                                OutputArray  inliers,
                                double       ransacThreshold = 3,
                                double       confidence = 0.99 )
```

Python:

cv.estimateTranslation3D(src, dst[, out[, inliers[, ransacThreshold[, confidence]]]]) -> retval, out, inliers

`#include <opencv2/calib3d.hpp>`

Computes an optimal translation between two 3D point sets.

It computes

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

Parameters

src	First input 3D point set containing (X, Y, Z).
dst	Second input 3D point set containing (x, y, z).
out	Output 3D translation vector 3×1 of the form $$\begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$$
inliers	Output vector indicating which points are inliers (1-inlier, 0-outlier).
ransacThreshold	Maximum reprojection error in the RANSAC algorithm to consider a point as an inlier.
confidence	Confidence level, between 0 and 1, for the estimated transformation. Anything between 0.95 and 0.99 is usually good enough. Values too close to 1 can slow down the estimation significantly. Values lower than 0.8-0.9 can result in an incorrectly estimated transformation.

The function estimates an optimal 3D translation between two 3D point sets using the RANSAC algorithm.

◆ filterHomographyDecompByVisibleRefpoints()

```
void cv::filterHomographyDecompByVisibleRefpoints ( InputArrayOfArrays rotations,
                                                    InputArrayOfArrays normals,
                                                    InputArray         beforePoints,
                                                    InputArray         afterPoints,
                                                    OutputArray        possibleSolutions,
                                                    InputArray         pointsMask = noArray() )
```

Python:

cv.filterHomographyDecompByVisibleRefpoints(rotations, normals, beforePoints, afterPoints[, possibleSolutions[, pointsMask]]) -> possibleSolutions

`#include <opencv2/calib3d.hpp>`

Filters homography decompositions based on additional information.

Parameters

rotations	Vector of rotation matrices.
normals	Vector of plane normal matrices.
beforePoints	Vector of (rectified) visible reference points before the homography is applied
afterPoints	Vector of (rectified) visible reference points after the homography is applied
possibleSolutions	Vector of int indices representing the viable solution set after filtering
pointsMask	optional Mat/Vector of 8u type representing the mask for the inliers as given by the findHomography function

This function is intended to filter the output of the decomposeHomographyMat based on additional information as described in [180] . The summary of the method: the decomposeHomographyMat function returns 2 unique solutions and their "opposites" for a total of 4 solutions. If we have access to the sets of points visible in the camera frame before and after the homography transformation is applied, we can determine which are the true potential solutions and which are the opposites by verifying which homographies are consistent with all visible reference points being in front of the camera. The inputs are left unchanged; the filtered solution set is returned as indices into the existing one.

◆ filterSpeckles()

```
void cv::filterSpeckles ( InputOutputArray img,
                          double           newVal,
                          int              maxSpeckleSize,
                          double           maxDiff,
                          InputOutputArray buf = noArray() )
```

Python:

cv.filterSpeckles(img, newVal, maxSpeckleSize, maxDiff[, buf]) -> img, buf

`#include <opencv2/calib3d.hpp>`

Filters off small noise blobs (speckles) in the disparity map.

Parameters

img	The input 16-bit signed disparity image
newVal	The disparity value used to paint-off the speckles
maxSpeckleSize	The maximum speckle size to consider it a speckle. Larger blobs are not affected by the algorithm
maxDiff	Maximum difference between neighbor disparity pixels to put them into the same blob. Note that since StereoBM, StereoSGBM and may be other algorithms return a fixed-point disparity map, where disparity values are multiplied by 16, this scale factor should be taken into account when specifying this parameter value.
buf	The optional temporary buffer to avoid memory allocation within the function.

◆ find4QuadCornerSubpix()

```
bool cv::find4QuadCornerSubpix ( InputArray       img,
                                 InputOutputArray corners,
                                 Size             region_size )
```

Python:

cv.find4QuadCornerSubpix(img, corners, region_size) -> retval, corners

`#include <opencv2/calib3d.hpp>`

finds subpixel-accurate positions of the chessboard corners

◆ findChessboardCorners()

```
bool cv::findChessboardCorners ( InputArray  image,
                                 Size        patternSize,
                                 OutputArray corners,
                                 int         flags = CALIB_CB_ADAPTIVE_THRESH+CALIB_CB_NORMALIZE_IMAGE )
```

Python:

cv.findChessboardCorners(image, patternSize[, corners[, flags]]) -> retval, corners

`#include <opencv2/calib3d.hpp>`

Finds the positions of internal corners of the chessboard.

Parameters

image	Source chessboard view. It must be an 8-bit grayscale or color image.
patternSize	Number of inner corners per a chessboard row and column (patternSize = cv::Size(points_per_row,points_per_colum) = cv::Size(columns,rows)).
corners	Output array of detected corners.
flags	Various operation flags that can be zero or a combination of the following values:

- **CALIB_CB_ADAPTIVE_THRESH** Use adaptive thresholding to convert the image to black and white, rather than a fixed threshold level (computed from the average image brightness).
- **CALIB_CB_NORMALIZE_IMAGE** Normalize the image gamma with **equalizeHist** before applying fixed or adaptive thresholding.
- **CALIB_CB_FILTER_QUADS** Use additional criteria (like contour area, perimeter, square-like shape) to filter out false quads extracted at the contour retrieval stage.
- **CALIB_CB_FAST_CHECK** Run a fast check on the image that looks for chessboard corners, and shortcut the call if none is found. This can drastically speed up the call in the degenerate condition when no chessboard is observed.
- **CALIB_CB_PLAIN** All other flags are ignored. The input image is taken as is. No image processing is done to improve to find the checkerboard. This has the effect of speeding up the execution of the function but could lead to not recognizing the checkerboard if the image is not previously binarized in the appropriate manner.

The function attempts to determine whether the input image is a view of the chessboard pattern and locate the internal chessboard corners. The function returns a non-zero value if all of the corners are found and they are placed in a certain order (row by row, left to right in every row). Otherwise, if the function fails to find all the corners or reorder them, it returns 0. For example, a regular chessboard has 8 x 8 squares and 7 x 7 internal corners, that is, points where the black squares touch each other. The detected coordinates are approximate, and to determine their positions more accurately, the function calls **cornerSubPix**. You also may use the function **cornerSubPix** with different parameters if returned coordinates are not accurate enough.

Sample usage of detecting and drawing chessboard corners: :

```
Size patternsize(8,6); //interior number of corners
Mat gray = ....; //source image
vector<Point2f> corners; //this will be filled by the detected corners

//CALIB_CB_FAST_CHECK saves a lot of time on images
//that do not contain any chessboard corners
bool patternfound = findChessboardCorners(gray, patternsize, corners,
        CALIB_CB_ADAPTIVE_THRESH + CALIB_CB_NORMALIZE_IMAGE
        + CALIB_CB_FAST_CHECK);

if(patternfound)
  cornerSubPix(gray, corners, Size(11, 11), Size(-1, -1),
    TermCriteria(CV_TERMCRIT_EPS + CV_TERMCRIT_ITER, 30, 0.1));

drawChessboardCorners(img, patternsize, Mat(corners), patternfound);
```

Note

The function requires white space (like a square-thick border, the wider the better) around the board to make the detection more robust in various environments. Otherwise, if there is no border and the background is dark, the outer black squares cannot be segmented properly and so the square grouping and ordering algorithm fails.

Use gen_pattern.py (**Create calibration pattern**) to create checkerboard.

◆ findChessboardCornersSB() [1/2]

```
bool cv::findChessboardCornersSB ( InputArray   image,
                                   Size         patternSize,
                                   OutputArray  corners,
                                   int          flags,
                                   OutputArray  meta )
```

Python:

cv.findChessboardCornersSB(image, patternSize[, corners[, flags]]) -> retval, corners

cv.findChessboardCornersSBWithMeta(image, patternSize, flags[, corners[, meta]]) -> retval, corners, meta

`#include <opencv2/calib3d.hpp>`

Finds the positions of internal corners of the chessboard using a sector based approach.

Parameters

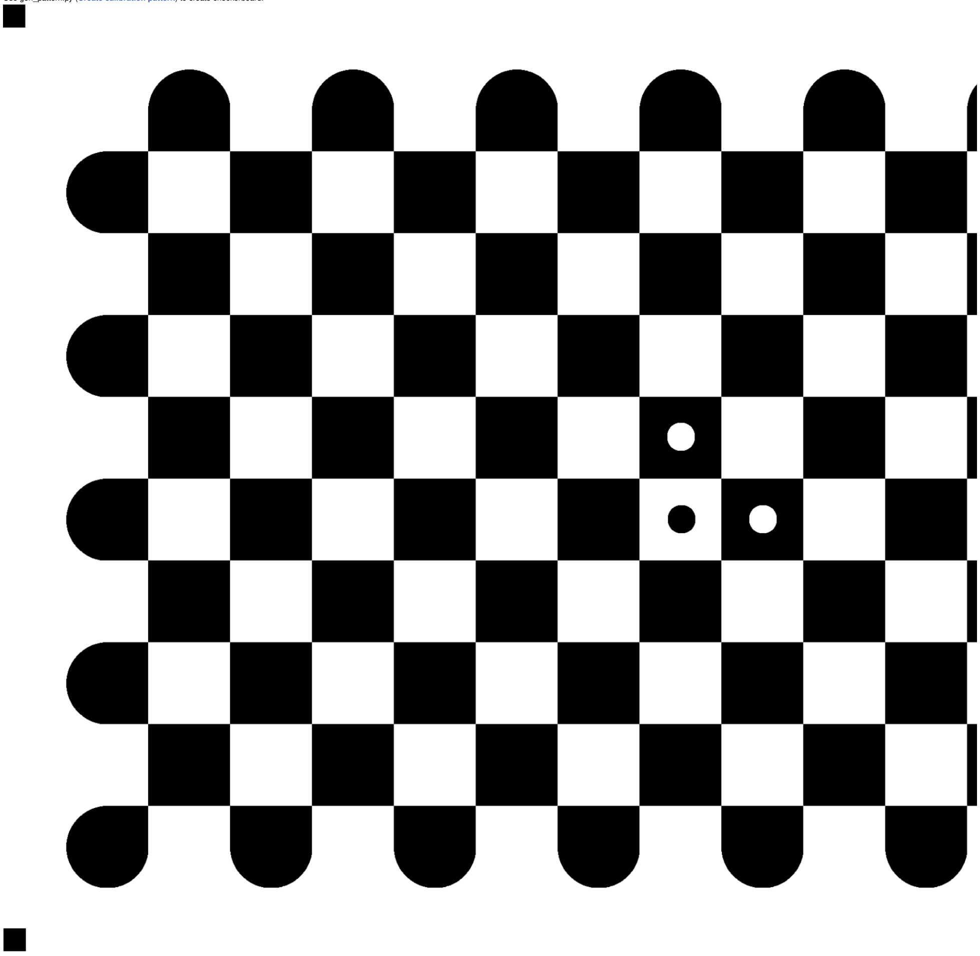
- **image** Source chessboard view. It must be an 8-bit grayscale or color image.
- **patternSize** Number of inner corners per a chessboard row and column (patternSize = cv::Size(points_per_row,points_per_colum) = cv::Size(columns,rows)).
- **corners** Output array of detected corners.
- **flags** Various operation flags that can be zero or a combination of the following values:
  - CALIB_CB_NORMALIZE_IMAGE Normalize the image gamma with equalizeHist before detection.
  - CALIB_CB_EXHAUSTIVE Run an exhaustive search to improve detection rate.
  - CALIB_CB_ACCURACY Up sample input image to improve sub-pixel accuracy due to aliasing effects.
  - CALIB_CB_LARGER The detected pattern is allowed to be larger than patternSize (see description).
  - CALIB_CB_MARKER The detected pattern must have a marker (see description). This should be used if an accurate camera calibration is required.
- **meta** Optional output arrray of detected corners (CV_8UC1 and size = cv::Size(columns,rows)). Each entry stands for one corner of the pattern and can have one of the following values:
  - 0 = no meta data attached
  - 1 = left-top corner of a black cell
  - 2 = left-top corner of a white cell
  - 3 = left-top corner of a black cell with a white marker dot
  - 4 = left-top corner of a white cell with a black marker dot (pattern origin in case of markers otherwise first corner)

The function is analog to findChessboardCorners but uses a localized radon transformation approximated by box filters being more robust to all sort of noise, faster on larger images and is able to directly return the sub-pixel position of the internal chessboard corners. The M
Calibration" demonstrating that the returned sub-pixel positions are more accurate than the one returned by cornerSubPix allowing a precise camera calibration for demanding applications.

In the case, the flags CALIB_CB_LARGER or CALIB_CB_MARKER are given, the result can be recovered from the optional meta array. Both flags are helpful to use calibration patterns exceeding the field of view of the camera. These oversized patterns allow more accurate
consistent coordinate system across all images, the optional marker (see image below) can be used to move the origin of the board to the location where the black circle is located.

> **Note**
> The function requires a white boarder with roughly the same width as one of the checkerboard fields around the whole board to improve the detection in various environments. In addition, because of the localized radon transformation it is beneficial to use round corners for t
> checkerboard optimized for the detection. However, any other checkerboard can be used as well.

Use gen_pattern.py (Create calibration pattern) to create checkerboard.



◆ findChessboardCornersSB() [2/2]

```
bool cv::findChessboardCornersSB ( InputArray   image,
                                   Size         patternSize,
                                   OutputArray  corners,
                                   int          flags = 0 )
```
inline

Python:

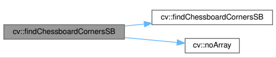
cv.findChessboardCornersSB(image, patternSize[, corners[, flags]]) -> retval, corners

cv.findChessboardCornersSBWithMeta(image, patternSize, flags[, corners[, meta]]) -> retval, corners, meta

`#include <opencv2/calib3d.hpp>`

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Here is the call graph for this function:



◆ findCirclesGrid() [1/2]

```
bool cv::findCirclesGrid ( InputArray                          image,
                           Size                                patternSize,
                           OutputArray                         centers,
                           int                                 flags,
                           const Ptr< FeatureDetector > &      blobDetector,
                           const CirclesGridFinderParameters & parameters )
```

Python:

```
cv.findCirclesGrid( image, patternSize, flags, blobDetector, parameters[, centers] ) -> retval, centers
cv.findCirclesGrid( image, patternSize[, centers[, flags[, blobDetector]]]         ) -> retval, centers
```

`#include <opencv2/calib3d.hpp>`

Finds centers in the grid of circles.

Parameters

- **image** grid view of input circles; it must be an 8-bit grayscale or color image.
- **patternSize** number of circles per row and column (patternSize = Size(points_per_row, points_per_colum)).
- **centers** output array of detected centers.
- **flags** various operation flags that can be one of the following values:
  - CALIB_CB_SYMMETRIC_GRID uses symmetric pattern of circles.
  - CALIB_CB_ASYMMETRIC_GRID uses asymmetric pattern of circles.
  - CALIB_CB_CLUSTERING uses a special algorithm for grid detection. It is more robust to perspective distortions but much more sensitive to background clutter.
- **blobDetector** feature detector that finds blobs like dark circles on light background. If `blobDetector` is NULL then `image` represents Point2f array of candidates.
- **parameters** struct for finding circles in a grid pattern.

The function attempts to determine whether the input image contains a grid of circles. If it is, the function locates centers of the circles. The function returns a non-zero value if all of the centers have been found and they have been placed in a certain order (row by row, left to right in every row). Otherwise, if the function fails to find all the corners or reorder them, it returns 0.

Sample usage of detecting and drawing the centers of circles: :

```
Size patternsize(7,7); //number of centers
Mat gray = ...; //source image
vector<Point2f> centers; //this will be filled by the detected centers

bool patternfound = findCirclesGrid(gray, patternsize, centers);

drawChessboardCorners(img, patternsize, Mat(centers), patternfound);
```

> **Note**
> The function requires white space (like a square-thick border, the wider the better) around the board to make the detection more robust in various environments.

◆ findCirclesGrid() [2/2]

```
bool cv::findCirclesGrid ( InputArray                  image,
                           Size                        patternSize,
                           OutputArray                 centers,
                           int                         flags = CALIB_CB_SYMMETRIC_GRID,
                           const Ptr< FeatureDetector > & blobDetector = SimpleBlobDetector::create() )
```

Python:

```
cv.findCirclesGrid( image, patternSize, flags, blobDetector, parameters[, centers] ) -> retval, centers
cv.findCirclesGrid( image, patternSize[, centers[, flags[, blobDetector]]]         ) -> retval, centers
```

`#include <opencv2/calib3d.hpp>`

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

◆ findEssentialMat() [1/6]

```
Mat cv::findEssentialMat ( InputArray  points1,
                           InputArray  points2,
                           double      focal,
                           Point2d     pp,
                           int         method,
                           double      prob,
                           double      threshold,
                           OutputArray mask )
```

Python:

```
cv.findEssentialMat( points1, points2, cameraMatrix[, method[, prob[, threshold[, maxIters[, mask]]]]]                                    ) -> retval, mask
cv.findEssentialMat( points1, points2[, focal[, pp[, method[, prob[, threshold[, maxIters[, mask]]]]]]]                                  ) -> retval, mask
cv.findEssentialMat( points1, points2, cameraMatrix1, distCoeffs1, cameraMatrix2, distCoeffs2[, method[, prob[, threshold[, mask]]]] ) -> retval, mask
cv.findEssentialMat( points1, points2, cameraMatrix1, cameraMatrix2, dist_coeff1, dist_coeff2, params[, mask]                         ) -> retval, mask
```

`#include <opencv2/calib3d.hpp>`

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

◆ findEssentialMat() [2/6]

```
Mat cv::findEssentialMat ( InputArray  points1,
                           InputArray  points2,
                           double      focal = 1.0,
                           Point2d     pp = Point2d(0, 0),
                           int         method = RANSAC,
                           double      prob = 0.999,
                           double      threshold = 1.0,
                           int         maxIters = 1000,
                           OutputArray mask = noArray() )
```

Python:

```
cv.findEssentialMat( points1, points2, cameraMatrix[, method[, prob[, threshold[, maxIters[, mask]]]]]                                    ) -> retval, mask
cv.findEssentialMat( points1, points2[, focal[, pp[, method[, prob[, threshold[, maxIters[, mask]]]]]]]                                  ) -> retval, mask
cv.findEssentialMat( points1, points2, cameraMatrix1, distCoeffs1, cameraMatrix2, distCoeffs2[, method[, prob[, threshold[, mask]]]] ) -> retval, mask
cv.findEssentialMat( points1, points2, cameraMatrix1, cameraMatrix2, dist_coeff1, dist_coeff2, params[, mask]                         ) -> retval, mask
```

`#include <opencv2/calib3d.hpp>`

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Parameters

- **points1** Array of N (N >= 5) 2D points from the first image. The point coordinates should be floating-point (single or double precision).
- **points2** Array of the second image points of the same size and format as points1 .
- **focal** focal length of the camera. Note that this function assumes that points1 and points2 are feature points from cameras with same focal length and principal point.
- **pp** principal point of the camera.
- **method** Method for computing a fundamental matrix.
  - RANSAC for the RANSAC algorithm.
  - LMEDS for the LMedS algorithm.
- **threshold** Parameter used for RANSAC. It is the maximum distance from a point to an epipolar line in pixels, beyond which the point is considered an outlier and is not used for computing the final fundamental matrix. It can be set to something like 1-3, depending on the accuracy of the point localization, image resolution, and the image noise.
- **prob** Parameter used for the RANSAC or LMedS methods only. It specifies a desirable level of confidence (probability) that the estimated matrix is correct.
- **mask** Output array of N elements, every element of which is set to 0 for outliers and to 1 for the other points. The array is computed only in the RANSAC and LMedS methods.
- **maxIters** The maximum number of robust method iterations.

This function differs from the one above that it computes camera intrinsic matrix from focal length and principal point:

$$A = \begin{bmatrix} f & 0 & x_{pp} \\ 0 & f & y_{pp} \\ 0 & 0 & 1 \end{bmatrix}$$

◆ findEssentialMat() [3/6]

```
Mat cv::findEssentialMat ( InputArray  points1,
                           InputArray  points2,
                           InputArray  cameraMatrix,
                           int         method,
                           double      prob,
                           double      threshold,
                           OutputArray mask )
```

Python:

```
cv.findEssentialMat( points1, points2, cameraMatrix[, method[, prob[, threshold[, maxIters[, mask]]]]]                                    ) -> retval, mask
cv.findEssentialMat( points1, points2[, focal[, pp[, method[, prob[, threshold[, maxIters[, mask]]]]]]]                                  ) -> retval, mask
cv.findEssentialMat( points1, points2, cameraMatrix1, distCoeffs1, cameraMatrix2, distCoeffs2[, method[, prob[, threshold[, mask]]]] ) -> retval, mask
cv.findEssentialMat( points1, points2, cameraMatrix1, cameraMatrix2, dist_coeff1, dist_coeff2, params[, mask]                         ) -> retval, mask
```

`#include <opencv2/calib3d.hpp>`

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

◆ findEssentialMat() [4/6]

```
Mat cv::findEssentialMat ( InputArray  points1,
                           InputArray  points2,
                           InputArray  cameraMatrix,
                           int         method = RANSAC,
                           double      prob = 0.999,
                           double      threshold = 1.0,
                           int         maxIters = 1000,
                           OutputArray mask = noArray() )
```

Python:

```
cv.findEssentialMat( points1, points2, cameraMatrix[, method[, prob[, threshold[, maxIters[, mask]]]]] ) -> retval, mask
cv.findEssentialMat( points1, points2[, focal[, pp[, method[, prob[, threshold[, maxIters[, mask]]]]]]] ) -> retval, mask
cv.findEssentialMat( points1, points2, cameraMatrix1, distCoeffs1, cameraMatrix2, distCoeffs2[, method[, prob[, threshold[, mask]]]] ) -> retval, mask
cv.findEssentialMat( points1, points2, cameraMatrix1, cameraMatrix2, dist_coeff1, dist_coeff2, params[, mask] ) -> retval, mask
```

`#include <opencv2/calib3d.hpp>`

Calculates an essential matrix from the corresponding points in two images.

Parameters

- **points1** Array of N (N >= 5) 2D points from the first image. The point coordinates should be floating-point (single or double precision).
- **points2** Array of the second image points of the same size and format as points1.
- **cameraMatrix** Camera intrinsic matrix $A = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$. Note that this function assumes that points1 and points2 are feature points from cameras with the same camera intrinsic matrix. If this assumption does not hold for your use case, use another function overload or **undistortPoints** with `P = cv::NoArray()` for both cameras to transform image points to normalized image coordinates, which are valid for the identity camera intrinsic matrix. When passing these coordinates, pass the identity matrix for this parameter.
- **method** Method for computing an essential matrix.
  - **RANSAC** for the RANSAC algorithm.
  - **LMEDS** for the LMedS algorithm.
- **prob** Parameter used for the RANSAC or LMedS methods only. It specifies a desirable level of confidence (probability) that the estimated matrix is correct.
- **threshold** Parameter used for RANSAC. It is the maximum distance from a point to an epipolar line in pixels, beyond which the point is considered an outlier and is not used for computing the final fundamental matrix. It can be set to something like 1-3, depending on the accuracy of the point localization, image resolution, and the image noise.
- **mask** Output array of N elements, every element of which is set to 0 for outliers and to 1 for the other points. The array is computed only in the RANSAC and LMedS methods.
- **maxIters** The maximum number of robust method iterations.

This function estimates essential matrix based on the five-point algorithm solver in [208] . [252] is also a related. The epipolar geometry is described by the following equation:

$$[p_2; 1]^T K^{-T} E K^{-1} [p_1; 1] = 0$$

where E is an essential matrix, p_1 and p_2 are corresponding points in the first and the second images, respectively. The result of this function may be passed further to **decomposeEssentialMat** or **recoverPose** to recover the relative pose between cameras.

◆ findEssentialMat() [5/6]

```
Mat cv::findEssentialMat ( InputArray        points1,
                           InputArray        points2,
                           InputArray        cameraMatrix1,
                           InputArray        cameraMatrix2,
                           InputArray        dist_coeff1,
                           InputArray        dist_coeff2,
                           OutputArray       mask,
                           const UsacParams & params )
```

Python:

```
cv.findEssentialMat( points1, points2, cameraMatrix[, method[, prob[, threshold[, maxIters[, mask]]]]] ) -> retval, mask
cv.findEssentialMat( points1, points2[, focal[, pp[, method[, prob[, threshold[, maxIters[, mask]]]]]]] ) -> retval, mask
cv.findEssentialMat( points1, points2, cameraMatrix1, distCoeffs1, cameraMatrix2, distCoeffs2[, method[, prob[, threshold[, mask]]]] ) -> retval, mask
cv.findEssentialMat( points1, points2, cameraMatrix1, cameraMatrix2, dist_coeff1, dist_coeff2, params[, mask] ) -> retval, mask
```

`#include <opencv2/calib3d.hpp>`

◆ findEssentialMat() [6/6]

```
Mat cv::findEssentialMat ( InputArray  points1,
                           InputArray  points2,
                           InputArray  cameraMatrix1,
                           InputArray  distCoeffs1,
                           InputArray  cameraMatrix2,
                           InputArray  distCoeffs2,
                           int         method = RANSAC,
                           double      prob = 0.999,
                           double      threshold = 1.0,
                           OutputArray mask = noArray() )
```

Python:

```
cv.findEssentialMat( points1, points2, cameraMatrix[, method[, prob[, threshold[, maxIters[, mask]]]]] ) -> retval, mask
cv.findEssentialMat( points1, points2[, focal[, pp[, method[, prob[, threshold[, maxIters[, mask]]]]]]] ) -> retval, mask
cv.findEssentialMat( points1, points2, cameraMatrix1, distCoeffs1, cameraMatrix2, distCoeffs2[, method[, prob[, threshold[, mask]]]] ) -> retval, mask
cv.findEssentialMat( points1, points2, cameraMatrix1, cameraMatrix2, dist_coeff1, dist_coeff2, params[, mask] ) -> retval, mask
```

`#include <opencv2/calib3d.hpp>`

Calculates an essential matrix from the corresponding points in two images from potentially two different cameras.

Parameters

- **points1** Array of N (N >= 5) 2D points from the first image. The point coordinates should be floating-point (single or double precision).
- **points2** Array of the second image points of the same size and format as points1.
- **cameraMatrix1** Camera matrix for the first camera $K = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$.
- **cameraMatrix2** Camera matrix for the second camera $K = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$.
- **distCoeffs1** Input vector of distortion coefficients for the first camera $(k_1, k_2, p_1, p_2[, k_3[, k_4, k_5, k_6[, s_1, s_2, s_3, s_4[, \tau_x, \tau_y]]]])$ of 4, 5, 8, 12 or 14 elements. If the vector is NULL/empty, the zero distortion coefficients are assumed.
- **distCoeffs2** Input vector of distortion coefficients for the second camera $(k_1, k_2, p_1, p_2[, k_3[, k_4, k_5, k_6[, s_1, s_2, s_3, s_4[, \tau_x, \tau_y]]]])$ of 4, 5, 8, 12 or 14 elements. If the vector is NULL/empty, the zero distortion coefficients are assumed.
- **method** Method for computing an essential matrix.
  - **RANSAC** for the RANSAC algorithm.
  - **LMEDS** for the LMedS algorithm.
- **prob** Parameter used for the RANSAC or LMedS methods only. It specifies a desirable level of confidence (probability) that the estimated matrix is correct.
- **threshold** Parameter used for RANSAC. It is the maximum distance from a point to an epipolar line in pixels, beyond which the point is considered an outlier and is not used for computing the final fundamental matrix. It can be set to something like 1-3, depending on the accuracy of the point localization, image resolution, and the image noise.
- **mask** Output array of N elements, every element of which is set to 0 for outliers and to 1 for the other points. The array is computed only in the RANSAC and LMedS methods.

This function estimates essential matrix based on the five-point algorithm solver in [208] . [252] is also a related. The epipolar geometry is described by the following equation:

$$[p_2; 1]^T K^{-T} E K^{-1} [p_1; 1] = 0$$

where E is an essential matrix, p_1 and p_2 are corresponding points in the first and the second images, respectively. The result of this function may be passed further to **decomposeEssentialMat** or **recoverPose** to recover the relative pose between cameras.

◆ findFundamentalMat() [1/4]

```
Mat cv::findFundamentalMat ( InputArray   points1,
                             InputArray   points2,
                             int          method,
                             double       ransacReprojThreshold,
                             double       confidence,
                             int          maxIters,
                             OutputArray  mask = noArray() )
```

Python:

```
cv.findFundamentalMat( points1, points2, method, ransacReprojThreshold, confidence, maxIters[, mask] ) -> retval, mask
cv.findFundamentalMat( points1, points2[, method[, ransacReprojThreshold[, confidence[, mask]]]]     ) -> retval, mask
cv.findFundamentalMat( points1, points2, params[, mask]                                              ) -> retval, mask
```

`#include <opencv2/calib3d.hpp>`

Calculates a fundamental matrix from the corresponding points in two images.

Parameters

points1	Array of N points from the first image. The point coordinates should be floating-point (single or double precision).
points2	Array of the second image points of the same size and format as points1 .
method	Method for computing a fundamental matrix. • **FM_7POINT** for a 7-point algorithm. $N = 7$ • **FM_8POINT** for an 8-point algorithm. $N \ge 8$ • **FM_RANSAC** for the RANSAC algorithm. $N \ge 8$ • **FM_LMEDS** for the LMedS algorithm. $N \ge 8$
ransacReprojThreshold	Parameter used only for RANSAC. It is the maximum distance from a point to an epipolar line in pixels, beyond which the point is considered an outlier and is not used for computing the final fundamental matrix. It can be set to something like 1-3, depending on the accuracy of the point localization, image resolution, and the image noise.
confidence	Parameter used for the RANSAC and LMedS methods only. It specifies a desirable level of confidence (probability) that the estimated matrix is correct.
[out] **mask**	optional output mask
maxIters	The maximum number of robust method iterations.

The epipolar geometry is described by the following equation:

$$[p_2; 1]^T F [p_1; 1] = 0$$

where F is a fundamental matrix, p_1 and p_2 are corresponding points in the first and the second images, respectively.

The function calculates the fundamental matrix using one of four methods listed above and returns the found fundamental matrix. Normally just one matrix is found. But in case of the 7-point algorithm, the function may return up to 3 solutions (9×3 matrix that stores all 3 matrices sequentially).

The calculated fundamental matrix may be passed further to **computeCorrespondEpilines** that finds the epipolar lines corresponding to the specified points. It can also be passed to **stereoRectifyUncalibrated** to compute the rectification transformation. :

```
// Example. Estimation of fundamental matrix using the RANSAC algorithm
int point_count = 100;
vector<Point2f> points1(point_count);
vector<Point2f> points2(point_count);

// initialize the points here ...
for( int i = 0; i < point_count; i++ )
{
    points1[i] = ...;
    points2[i] = ...;
}

Mat fundamental_matrix =
 findFundamentalMat(points1, points2, FM_RANSAC, 3, 0.99);
```

◆ findFundamentalMat() [2/4]

```
Mat cv::findFundamentalMat ( InputArray   points1,
                             InputArray   points2,
                             int          method = FM_RANSAC,
                             double       ransacReprojThreshold = 3.,
                             double       confidence = 0.99,
                             OutputArray  mask = noArray() )
```

Python:

```
cv.findFundamentalMat( points1, points2, method, ransacReprojThreshold, confidence, maxIters[, mask] ) -> retval, mask
cv.findFundamentalMat( points1, points2[, method[, ransacReprojThreshold[, confidence[, mask]]]]     ) -> retval, mask
cv.findFundamentalMat( points1, points2, params[, mask]                                              ) -> retval, mask
```

`#include <opencv2/calib3d.hpp>`

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

◆ findFundamentalMat() [3/4]

```
Mat cv::findFundamentalMat ( InputArray          points1,
                             InputArray          points2,
                             OutputArray         mask,
                             const UsacParams &  params )
```

Python:

```
cv.findFundamentalMat( points1, points2, method, ransacReprojThreshold, confidence, maxIters[, mask] ) -> retval, mask
cv.findFundamentalMat( points1, points2[, method[, ransacReprojThreshold[, confidence[, mask]]]]     ) -> retval, mask
cv.findFundamentalMat( points1, points2, params[, mask]                                              ) -> retval, mask
```

`#include <opencv2/calib3d.hpp>`

◆ findFundamentalMat() [4/4]

```
Mat cv::findFundamentalMat ( InputArray   points1,
                             InputArray   points2,
                             OutputArray  mask,
                             int          method = FM_RANSAC,
                             double       ransacReprojThreshold = 3.,
                             double       confidence = 0.99 )
```

Python:

```
cv.findFundamentalMat( points1, points2, method, ransacReprojThreshold, confidence, maxIters[, mask] ) -> retval, mask
cv.findFundamentalMat( points1, points2[, method[, ransacReprojThreshold[, confidence[, mask]]]]     ) -> retval, mask
cv.findFundamentalMat( points1, points2, params[, mask]                                              ) -> retval, mask
```

`#include <opencv2/calib3d.hpp>`

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

◆ findHomography() [1/3]

```
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
cv.findHomography( srcPoints, dstPoints, params[, mask]                                                    ) -> 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

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

  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.

The function finds and returns the perspective transformation H between the source and the destination planes:

$$s_i \begin{bmatrix} x'_i \\ y'_i \\ 1 \end{bmatrix} \sim H \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix}$$

so that the back-projection error

$$\sum_i \left(x'_i - \frac{h_{11} x_i + h_{12} y_i + h_{13}}{h_{31} x_i + h_{32} y_i + h_{33}} \right)^2 + \left(y'_i - \frac{h_{21} x_i + h_{22} y_i + h_{23}}{h_{31} x_i + h_{32} y_i + h_{33}} \right)^2$$

is minimized. If the parameter method is set to the default value 0, the function uses all the point pairs to compute an initial homography estimate with a simple least-squares scheme.

However, if not all of the point pairs ($srcPoints_i, dstPoints_i$) fit the rigid perspective transformation (that is, there are some outliers), this initial estimate will be poor. In this case, you can use one of the three robust methods. The methods RANSAC, LMeDS and RHO try many different random subsets of the corresponding point pairs (of four pairs each, collinear pairs are discarded), estimate the homography matrix using this subset and a simple least-squares algorithm, and then compute the quality/goodness of the computed homography (which is the number of inliers for RANSAC or the least median re-projection error for LMeDS). The best subset is then used to produce the initial estimate of the homography matrix and the mask of inliers/outliers.

Regardless of the method, robust or not, the computed homography matrix is refined further (using inliers only in case of a robust method) with the Levenberg-Marquardt method to reduce the re-projection error even more.

The methods RANSAC and RHO can handle practically any ratio of outliers but need a threshold to distinguish inliers from outliers. The method LMeDS does not need any threshold but it works correctly only when there are more than 50% of inliers. Finally, if there are no outliers and the noise is rather small, use the default method (method=0).

The function is used to find initial intrinsic and extrinsic matrices. Homography matrix is determined up to a scale. If h_{33} is non-zero, the matrix is normalized so that $h_{33} = 1$.

> **Note**
> Whenever an H matrix cannot be estimated, an empty one will be returned.

See also
getAffineTransform, estimateAffine2D, estimateAffinePartial2D, getPerspectiveTransform, warpPerspective, perspectiveTransform

◆ findHomography() [2/3]

```
Mat cv::findHomography ( InputArray           srcPoints,
                         InputArray           dstPoints,
                         OutputArray          mask,
                         const UsacParams &   params )
```

Python:

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

`#include <opencv2/calib3d.hpp>`

◆ findHomography() [3/3]

```
Mat cv::findHomography ( InputArray    srcPoints,
                         InputArray    dstPoints,
                         OutputArray   mask,
                         int           method = 0,
                         double        ransacReprojThreshold = 3 )
```

Python:

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

`#include <opencv2/calib3d.hpp>`

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

◆ getDefaultNewCameraMatrix()

```
Mat cv::getDefaultNewCameraMatrix ( InputArray  cameraMatrix,
                                    Size        imgsize = Size(),
                                    bool        centerPrincipalPoint = false )
```

Python:

```
cv.getDefaultNewCameraMatrix( cameraMatrix[, imgsize[, centerPrincipalPoint]] ) -> retval
```

`#include <opencv2/calib3d.hpp>`

Returns the default new camera matrix.

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.

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.

◆ getOptimalNewCameraMatrix()

```
Mat cv::getOptimalNewCameraMatrix ( InputArray cameraMatrix,
                                    InputArray distCoeffs,
                                    Size       imageSize,
                                    double     alpha,
                                    Size       newImgSize = Size(),
                                    Rect *     validPixROI = 0,
                                    bool       centerPrincipalPoint = false )
```

Python:

cv.getOptimalNewCameraMatrix(cameraMatrix, distCoeffs, imageSize, alpha[, newImgSize[, centerPrincipalPoint]]) -> retval, validPixROI

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

Returns the new camera intrinsic matrix based on the free scaling parameter.

Parameters

- **cameraMatrix** Input camera intrinsic matrix.
- **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 NULL/empty, the zero distortion coefficients are assumed.
- **imageSize** Original image size.
- **alpha** Free scaling parameter between 0 (when all the pixels in the undistorted image are valid) and 1 (when all the source image pixels are retained in the undistorted image). See **stereoRectify** for details.
- **newImgSize** Image size after rectification. By default, it is set to imageSize .
- **validPixROI** Optional output rectangle that outlines all-good-pixels region in the undistorted image. See roi1, roi2 description in **stereoRectify** .
- **centerPrincipalPoint** Optional flag that indicates whether in the new camera intrinsic matrix the principal point should be at the image center or not. By default, the principal point is chosen to best fit a subset of the source image (determined by alpha) to the corrected image.

Returns

new_camera_matrix Output new camera intrinsic matrix.

The function computes and returns the optimal new camera intrinsic matrix based on the free scaling parameter. By varying this parameter, you may retrieve only sensible pixels alpha=0 , keep all the original image pixels if there is valuable information in the corners alpha=1 , or get something in between. When alpha>0 , the undistorted result is likely to have some black pixels corresponding to "virtual" pixels outside of the captured distorted image. The original camera intrinsic matrix, distortion coefficients, the computed new camera intrinsic matrix, and newImageSize should be passed to **initUndistortRectifyMap** to produce the maps for **remap** .

◆ getValidDisparityROI()

```
Rect cv::getValidDisparityROI ( Rect roi1,
                                Rect roi2,
                                int  minDisparity,
                                int  numberOfDisparities,
                                int  blockSize )
```

Python:

cv.getValidDisparityROI(roi1, roi2, minDisparity, numberOfDisparities, blockSize) -> retval

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

computes valid disparity ROI from the valid ROIs of the rectified images (that are returned by **stereoRectify**)

◆ initCameraMatrix2D()

```
Mat cv::initCameraMatrix2D ( InputArrayOfArrays objectPoints,
                             InputArrayOfArrays imagePoints,
                             Size               imageSize,
                             double             aspectRatio = 1.0 )
```

Python:

cv.initCameraMatrix2D(objectPoints, imagePoints, imageSize[, aspectRatio]) -> retval

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

Finds an initial camera intrinsic matrix from 3D-2D point correspondences.

Parameters

- **objectPoints** Vector of vectors of the calibration pattern points in the calibration pattern coordinate space. In the old interface all the per-view vectors are concatenated. See **calibrateCamera** for details.
- **imagePoints** Vector of vectors of the projections of the calibration pattern points. In the old interface all the per-view vectors are concatenated.
- **imageSize** Image size in pixels used to initialize the principal point.
- **aspectRatio** If it is zero or negative, both f_x and f_y are estimated independently. Otherwise, $f_x = f_y \cdot \texttt{aspectRatio}$.

The function estimates and returns an initial camera intrinsic matrix for the camera calibration process. Currently, the function only supports planar calibration patterns, which are patterns where each object point has z-coordinate =0.

◆ initInverseRectificationMap()

```
void cv::initInverseRectificationMap ( InputArray   cameraMatrix,
                                       InputArray   distCoeffs,
                                       InputArray   R,
                                       InputArray   newCameraMatrix,
                                       const Size & size,
                                       int          m1type,
                                       OutputArray  map1,
                                       OutputArray  map2 )
```

Python:

cv.initInverseRectificationMap(cameraMatrix, distCoeffs, R, newCameraMatrix, size, m1type[, map1[, map2]]) -> map1, map2

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

Computes the projection and inverse-rectification transformation map. In essense, this is the inverse of **initUndistortRectifyMap** to accomodate stereo-rectification of projectors ('inverse-cameras') in projector-camera pairs.

The function computes the joint projection and inverse rectification transformation and represents the result in the form of maps for **remap**. The projected image looks like a distorted version of the original which, once projected by a projector, should visually match the original. 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 projector-camera pair, newCameraMatrix is normally set to P1 or P2 computed by **stereoRectify** .

The projector is oriented differently in the coordinate space, according to R. In case of projector-camera pairs, this helps align the projector (in the same manner as **initUndistortRectifyMap** for the camera) to create a stereo-rectified pair. This allows epipolar lines on both images to become horizontal and have the same y-coordinate (in case of a horizontally aligned projector-camera pair).

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

$$
\begin{array}{l}
\text{newCameraMatrix}\\
x \leftarrow (u - {c'}_x)/{f'}_x \\
y \leftarrow (v - {c'}_y)/{f'}_y \\
\\
\text{Undistortion}\\
\scriptsize{\textit{though equation shown is for radial undistortion, function implements cv::undistortPoints()}}\\
r^2 \leftarrow x^2 + y^2 \\
\theta \leftarrow \frac{1 + k_1 r^2 + k_2 r^4 + k_3 r^6}{1 + k_4 r^2 + k_5 r^4 + k_6 r^6}\\
x' \leftarrow \frac{x}{\theta} \\
y' \leftarrow \frac{y}{\theta} \\
\\
\text{Rectification}\\
{[X\,Y\,W]}^T \leftarrow R*[x'\,y'\,1]^T \\
x'' \leftarrow X/W \\
y'' \leftarrow Y/W \\
\\
\text{cameraMatrix}\\
map_x(u,v) \leftarrow x'' f_x + c_x \\
map_y(u,v) \leftarrow y'' f_y + c_y
\end{array}
$$

where $(k_1, k_2, p_1, p_2[, k_3[, k_4, k_5, k_6[, s_1, s_2, s_3, s_4[, \tau_x, \tau_y]]]])$ are the distortion coefficients vector distCoeffs.

In case of a stereo-rectified projector-camera pair, this function is called for the projector while **initUndistortRectifyMap** is called for the camera head. This is done after **stereoRectify**, which in turn is called after **stereoCalibrate**. If the projector-camera pair is not calibrated, it is still possible to compute the rectification transformations directly from the fundamental matrix using **stereoRectifyUncalibrated**. For the projector and 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

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

where cameraMatrix can be chosen arbitrarily.

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 $(k_1, k_2, p_1, p_2[, k_3[, k_4, k_5, k_6[, s_1, s_2, s_3, s_4[, \tau_x, \tau_y]]]])$ of 4, 5, 8, 12 or 14 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.
- **newCameraMatrix** New camera matrix $A' = \begin{bmatrix} f_x' & 0 & c_x' \\ 0 & f_y' & c_y' \\ 0 & 0 & 1 \end{bmatrix}$.
- **size** Distorted image size.
- **m1type** Type of the first output map. Can be CV_32FC1, CV_32FC2 or CV_16SC2, see **convertMaps**
- **map1** The first output map for **remap**.
- **map2** The second output map for **remap**.

◆ initUndistortRectifyMap()

```
void cv::initUndistortRectifyMap ( InputArray   cameraMatrix,
                                   InputArray   distCoeffs,
                                   InputArray   R,
                                   InputArray   newCameraMatrix,
                                   Size         size,
                                   int          m1type,
                                   OutputArray  map1,
                                   OutputArray  map2 )
```

Python:

cv.initUndistortRectifyMap(cameraMatrix, distCoeffs, R, newCameraMatrix, size, m1type[, map1[, map2]]) -> map1, map2

`#include <opencv2/calib3d.hpp>`

Computes the undistortion and rectification transformation 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:

$$
\begin{array}{l}
x \leftarrow (u - {c'}_x)/{f'}_x \\
y \leftarrow (v - {c'}_y)/{f'}_y \\
{[X\,Y\,W]}^T \leftarrow R^{-1} * [x\,y\,1]^T \\
x' \leftarrow X/W \\
y' \leftarrow Y/W \\
r^2 \leftarrow x'^2 + y'^2 \\
x'' \leftarrow x' \frac{1 + k_1 r^2 + k_2 r^4 + k_3 r^6}{1 + k_4 r^2 + k_5 r^4 + k_6 r^6} + 2p_1 x' y' + p_2(r^2 + 2 x'^2) + s_1 r^2 + s_2 r^4 \\
y'' \leftarrow y' \frac{1 + k_1 r^2 + k_2 r^4 + k_3 r^6}{1 + k_4 r^2 + k_5 r^4 + k_6 r^6} + p_1 (r^2 + 2 y'^2) + 2 p_2 x' y' + s_3 r^2 + s_4 r^4 \\
s\vecthree{x'''}{y'''}{1} = \vecthreethree{R_{33}(\tau_x, \tau_y)}{0}{-R_{13}((\tau_x, \tau_y)}{0}{R_{33}(\tau_x, \tau_y)}{-R_{23}(\tau_x, \tau_y)}{0}{0}{1} R(\tau_x, \tau_y) \vecthree{x''}{y''}{1} \\
map_x(u,v) \leftarrow x''' f_x + c_x \\
map_y(u,v) \leftarrow y''' f_y + c_y
\end{array}
$$

where $(k_1, k_2, p_1, p_2[, k_3[, k_4, k_5, k_6[, s_1, s_2, s_3, s_4[, \tau_x, \tau_y]]]])$ 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

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

where cameraMatrix can be chosen arbitrarily.

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 $(k_1, k_2, p_1, p_2[, k_3[, k_4, k_5, k_6[, s_1, s_2, s_3, s_4[, \tau_x, \tau_y]]]])$ of 4, 5, 8, 12 or 14 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 **initUndistortRectifyMap** 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, CV_32FC2 or CV_16SC2, see **convertMaps**
map1	The first output map.
map2	The second output map.

◆ initWideAngleProjMap() [1/2]

```
float cv::initWideAngleProjMap ( InputArray          cameraMatrix,
                                 InputArray          distCoeffs,
                                 Size                imageSize,
                                 int                 destImageWidth,
                                 int                 m1type,
                                 OutputArray         map1,
                                 OutputArray         map2,
                                 enum UndistortTypes projType = PROJ_SPHERICAL_EQRECT,
                                 double              alpha = 0 )
```

`#include <opencv2/calib3d.hpp>`

initializes maps for **remap** for wide-angle

◆ initWideAngleProjMap() [2/2]

```
static float cv::initWideAngleProjMap ( InputArray   cameraMatrix,
                                        InputArray   distCoeffs,
                                        Size         imageSize,
                                        int          destImageWidth,
                                        int          m1type,
                                        OutputArray  map1,
                                        OutputArray  map2,
                                        int          projType,
                                        double       alpha = 0 )
```

inline static

`#include <opencv2/calib3d.hpp>`

Here is the call graph for this function:

cv::initWideAngleProjMap → cv::initWideAngleProjMap

◆ matMulDeriv()

```
void cv::matMulDeriv ( InputArray   A,
                       InputArray   B,
                       OutputArray  dABdA,
                       OutputArray  dABdB )
```

Python:

cv.matMulDeriv(A, B[, dABdA[, dABdB]]) -> dABdA, dABdB

`#include <opencv2/calib3d.hpp>`

Computes partial derivatives of the matrix product for each multiplied matrix.

Parameters

A	First multiplied matrix.
B	Second multiplied matrix.
dABdA	First output derivative matrix d(A*B)/dA of size $\texttt{A.rows*B.cols} \times {A.rows*A.cols}$.
dABdB	Second output derivative matrix d(A*B)/dB of size $\texttt{A.rows*B.cols} \times {B.rows*B.cols}$.

The function computes partial derivatives of the elements of the matrix product $A*B$ with regard to the elements of each of the two input matrices. The function is used to compute the Jacobian matrices in **stereoCalibrate** but can also be used in any other similar optimization function.

◆ 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.

> **Note**
> By setting rvec = tvec = $[0, 0, 0]$, or by setting cameraMatrix to a 3x3 identity matrix, or by passing zero distortion coefficients, one can get various useful partial cases of the function. This means, one can compute the distorted coordinates for a sparse set of points or apply a perspective transformation (and also compute the derivatives) in the ideal zero-distortion setup.

◆ recoverPose() [1/4]

```
int cv::recoverPose ( InputArray        E,
                      InputArray        points1,
                      InputArray        points2,
                      InputArray        cameraMatrix,
                      OutputArray       R,
                      OutputArray       t,
                      double            distanceThresh,
                      InputOutputArray  mask = noArray(),
                      OutputArray       triangulatedPoints = noArray() )
```

Python:

```
cv.recoverPose( points1, points2, cameraMatrix1, distCoeffs1, cameraMatrix2, distCoeffs2[, E[, R[, t[, method[, prob[, threshold[, mask]]]]]]] ) -> retval, E, R, t, mask
cv.recoverPose( E, points1, points2, cameraMatrix[, R[, t[, mask]]] ) -> retval, R, t, mask
cv.recoverPose( E, points1, points2[, R[, t[, focal[, pp[, mask]]]]] ) -> retval, R, t, mask
cv.recoverPose( E, points1, points2, cameraMatrix, distanceThresh[, R[, t[, mask[, triangulatedPoints]]]] ) -> retval, R, t, mask, triangulatedPoints
```

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

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Parameters

- **E** The input essential matrix.
- **points1** Array of N 2D points from the first image. The point coordinates should be floating-point (single or double precision).
- **points2** Array of the second image points of the same size and format as points1.
- **cameraMatrix** Camera intrinsic matrix $A = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$. Note that this function assumes that points1 and points2 are feature points from cameras with the same camera intrinsic matrix.
- **R** Output rotation matrix. Together with the translation vector, this matrix makes up a tuple that performs a change of basis from the first camera's coordinate system to the second camera's coordinate system. Note that, in general, t can not be used for this tuple, see the parameter description below.
- **t** Output translation vector. This vector is obtained by **decomposeEssentialMat** and therefore is only known up to scale, i.e. t is the direction of the translation vector and has unit length.
- **distanceThresh** threshold distance which is used to filter out far away points (i.e. infinite points).
- **mask** Input/output mask for inliers in points1 and points2. If it is not empty, then it marks inliers in points1 and points2 for the given essential matrix E. Only these inliers will be used to recover pose. In the output mask only inliers which pass the chirality check.
- **triangulatedPoints** 3D points which were reconstructed by triangulation.

This function differs from the one above that it outputs the triangulated 3D point that are used for the chirality check.

◆ recoverPose() [2/4]

```
int cv::recoverPose ( InputArray       E,
                      InputArray       points1,
                      InputArray       points2,
                      InputArray       cameraMatrix,
                      OutputArray      R,
                      OutputArray      t,
                      InputOutputArray mask = noArray() )
```

Python:

```
cv.recoverPose( points1, points2, cameraMatrix1, distCoeffs1, cameraMatrix2, distCoeffs2[, E[, R[, t[, method[, prob[, threshold[, mask]]]]]]] ) -> retval, E, R, t, mask
cv.recoverPose( E, points1, points2, cameraMatrix[, R[, t[, mask]]] ) -> retval, R, t, mask
cv.recoverPose( E, points1, points2[, R[, t[, focal[, pp[, mask]]]]] ) -> retval, R, t, mask
cv.recoverPose( E, points1, points2, cameraMatrix, distanceThresh[, R[, t[, mask[, triangulatedPoints]]]] ) -> retval, R, t, mask, triangulatedPoints
```

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

Recovers the relative camera rotation and the translation from an estimated essential matrix and the corresponding points in two images, using chirality check. Returns the number of inliers that pass the check.

Parameters

- **E** The input essential matrix.
- **points1** Array of N 2D points from the first image. The point coordinates should be floating-point (single or double precision).
- **points2** Array of the second image points of the same size and format as points1 .
- **cameraMatrix** Camera intrinsic matrix $A = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$. Note that this function assumes that points1 and points2 are feature points from cameras with the same camera intrinsic matrix.
- **R** Output rotation matrix. Together with the translation vector, this matrix makes up a tuple that performs a change of basis from the first camera's coordinate system to the second camera's coordinate system. Note that, in general, t can not be used for this tuple, see the parameter described below.
- **t** Output translation vector. This vector is obtained by **decomposeEssentialMat** and therefore is only known up to scale, i.e. t is the direction of the translation vector and has unit length.
- **mask** Input/output mask for inliers in points1 and points2. If it is not empty, then it marks inliers in points1 and points2 for the given essential matrix E. Only these inliers will be used to recover pose. In the output mask only inliers which pass the chirality check.

This function decomposes an essential matrix using **decomposeEssentialMat** and then verifies possible pose hypotheses by doing chirality check. The chirality check means that the triangulated 3D points should have positive depth. Some details can be found in [208].

This function can be used to process the output E and mask from **findEssentialMat**. In this scenario, points1 and points2 are the same input for **findEssentialMat** :

```
// Example. Estimation of fundamental matrix using the RANSAC algorithm
int point_count = 100;
vector<Point2f> points1(point_count);
vector<Point2f> points2(point_count);

// initialize the points here ...
for( int i = 0; i < point_count; i++ )
{
    points1[i] = ...;
    points2[i] = ...;
}

// cametra matrix with both focal lengths = 1, and principal point = (0, 0)
Mat cameraMatrix = Mat::eye(3, 3, CV_64F);

Mat E, R, t, mask;

E = findEssentialMat(points1, points2, cameraMatrix, RANSAC, 0.999, 1.0, mask);
recoverPose(E, points1, points2, cameraMatrix, R, t, mask);
```

◆ recoverPose() [3/4]

```
int cv::recoverPose ( InputArray       E,
                      InputArray       points1,
                      InputArray       points2,
                      OutputArray      R,
                      OutputArray      t,
                      double           focal = 1.0,
                      Point2d          pp = Point2d(0, 0),
                      InputOutputArray mask = noArray() )
```

Python:

```
cv.recoverPose( points1, points2, cameraMatrix1, distCoeffs1, cameraMatrix2, distCoeffs2[, E[, R[, t[, method[, prob[, threshold[, mask]]]]]]] ) -> retval, E, R, t, mask
cv.recoverPose( E, points1, points2, cameraMatrix[, R[, t[, mask]]] ) -> retval, R, t, mask
cv.recoverPose( E, points1, points2[, R[, t[, focal[, pp[, mask]]]]] ) -> retval, R, t, mask
cv.recoverPose( E, points1, points2, cameraMatrix, distanceThresh[, R[, t[, mask[, triangulatedPoints]]]] ) -> retval, R, t, mask, triangulatedPoints
```

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

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

Parameters

- **E** The input essential matrix.
- **points1** Array of N 2D points from the first image. The point coordinates should be floating-point (single or double precision).
- **points2** Array of the second image points of the same size and format as points1 .
- **R** Output rotation matrix. Together with the translation vector, this matrix makes up a tuple that performs a change of basis from the first camera's coordinate system to the second camera's coordinate system. Note that, in general, t can not be used for this tuple, see the parameter description below.
- **t** Output translation vector. This vector is obtained by **decomposeEssentialMat** and therefore is only known up to scale, i.e. t is the direction of the translation vector and has unit length.
- **focal** Focal length of the camera. Note that this function assumes that points1 and points2 are feature points from cameras with same focal length and principal point.
- **pp** principal point of the camera.
- **mask** Input/output mask for inliers in points1 and points2. If it is not empty, then it marks inliers in points1 and points2 for the given essential matrix E. Only these inliers will be used to recover pose. In the output mask only inliers which pass the chirality check.

This function differs from the one above that it computes camera intrinsic matrix from focal length and principal point:

$$A = \begin{bmatrix} f & 0 & x_{pp} \\ 0 & f & y_{pp} \\ 0 & 0 & 1 \end{bmatrix}$$

◆ recoverPose() [4/4]

```
int cv::recoverPose ( InputArray         points1,
                      InputArray         points2,
                      InputArray         cameraMatrix1,
                      InputArray         distCoeffs1,
                      InputArray         cameraMatrix2,
                      InputArray         distCoeffs2,
                      OutputArray        E,
                      OutputArray        R,
                      OutputArray        t,
                      int                method = cv::RANSAC,
                      double             prob = 0.999,
                      double             threshold = 1.0,
                      InputOutputArray   mask = noArray() )
```

Python:

cv.recoverPose(points1, points2, cameraMatrix1, distCoeffs1, cameraMatrix2, distCoeffs2[, E[, R[, t[, method[, prob[, threshold[, mask]]]]]]]) -> retval, E, R, t, mask

cv.recoverPose(E, points1, points2, cameraMatrix[, R[, t[, mask]]]) -> retval, R, t, mask

cv.recoverPose(E, points1, points2[, R[, t[, focal[, pp[, mask]]]]]) -> retval, R, t, mask

cv.recoverPose(E, points1, points2, cameraMatrix, distanceThresh[, R[, t[, mask[, triangulatedPoints]]]]) -> retval, R, t, mask, triangulatedPoints

`#include <opencv2/calib3d.hpp>`

Recovers the relative camera rotation and the translation from corresponding points in two images from two different cameras, using cheirality check. Returns the number of inliers that pass the check.

Parameters

points1	Array of N 2D points from the first image. The point coordinates should be floating-point (single or double precision).
points2	Array of the second image points of the same size and format as points1 .
cameraMatrix1	Input/output camera matrix for the first camera, the same as in **calibrateCamera**. Furthermore, for the stereo case, additional flags may be used, see below.
distCoeffs1	Input/output vector of distortion coefficients, the same as in **calibrateCamera**.
cameraMatrix2	Input/output camera matrix for the first camera, the same as in **calibrateCamera**. Furthermore, for the stereo case, additional flags may be used, see below.
distCoeffs2	Input/output vector of distortion coefficients, the same as in **calibrateCamera**.
E	The output essential matrix.
R	Output rotation matrix. Together with the translation vector, this matrix makes up a tuple that performs a change of basis from the first camera's coordinate system to the second camera's coordinate system. Note that, in general, t can not be used for this tuple, see the parameter described below.
t	Output translation vector. This vector is obtained by **decomposeEssentialMat** and therefore is only known up to scale, i.e. t is the direction of the translation vector and has unit length.
method	Method for computing an essential matrix. • **RANSAC** for the RANSAC algorithm. • **LMEDS** for the LMedS algorithm.
prob	Parameter used for the RANSAC or LMedS methods only. It specifies a desirable level of confidence (probability) that the estimated matrix is correct.
threshold	Parameter used for RANSAC. It is the maximum distance from a point to an epipolar line in pixels, beyond which the point is considered an outlier and is not used for computing the final fundamental matrix. It can be set to something like 1-3, depending on the accuracy of the point localization, image resolution, and the image noise.
mask	Input/output mask for inliers in points1 and points2. If it is not empty, then it marks inliers in points1 and points2 for then given essential matrix E. Only these inliers will be used to recover pose. In the output mask only inliers which pass the cheirality check.

This function decomposes an essential matrix using **decomposeEssentialMat** and then verifies possible pose hypotheses by doing cheirality check. The cheirality check means that the triangulated 3D points should have positive depth. Some details can be found in [208].

This function can be used to process the output E and mask from **findEssentialMat**. In this scenario, points1 and points2 are the same input for findEssentialMat.:

```
// Example. Estimation of fundamental matrix using the RANSAC algorithm
int point_count = 100;
vector<Point2f> points1(point_count);
vector<Point2f> points2(point_count);

// initialize the points here ...
for( int i = 0; i < point_count; i++ )
{
    points1[i] = ...;
    points2[i] = ...;
}

// Input: camera calibration of both cameras, for example using intrinsic chessboard calibration.
Mat cameraMatrix1, distCoeffs1, cameraMatrix2, distCoeffs2;

// Output: Essential matrix, relative rotation and relative translation.
Mat E, R, t, mask;

recoverPose(points1, points2, cameraMatrix1, distCoeffs1, cameraMatrix2, distCoeffs2, E, R, t, mask);
```

◆ rectify3Collinear()

```
float cv::rectify3Collinear ( InputArray           cameraMatrix1,
                              InputArray           distCoeffs1,
                              InputArray           cameraMatrix2,
                              InputArray           distCoeffs2,
                              InputArray           cameraMatrix3,
                              InputArray           distCoeffs3,
                              InputArrayOfArrays   imgpt1,
                              InputArrayOfArrays   imgpt3,
                              Size                 imageSize,
                              InputArray           R12,
                              InputArray           T12,
                              InputArray           R13,
                              InputArray           T13,
                              OutputArray          R1,
                              OutputArray          R2,
                              OutputArray          R3,
                              OutputArray          P1,
                              OutputArray          P2,
                              OutputArray          P3,
                              OutputArray          Q,
                              double               alpha,
                              Size                 newImgSize,
                              Rect *               roi1,
                              Rect *               roi2,
                              int                  flags )
```

Python:

cv.rectify3Collinear(cameraMatrix1, distCoeffs1, cameraMatrix2, distCoeffs2, cameraMatrix3, distCoeffs3, imgpt1, imgpt3, imageSize, R12, T12, R13, T13, alpha, newImgSize, flags[, R1[, R2[, R3[, P1[, P2[, P3[, Q]]]]]]]) -> retval, R1, R2, R3, P1, P2, P3, Q, roi1, roi2

`#include <opencv2/calib3d.hpp>`

computes the rectification transformations for 3-head camera, where all the heads are on the same line.

◆ reprojectImageTo3D()

```
void cv::reprojectImageTo3D ( InputArray   disparity,
                              OutputArray  _3dImage,
                              InputArray   Q,
                              bool         handleMissingValues = false,
                              int          ddepth = -1 )
```

Python:

```
cv.reprojectImageTo3D( disparity, Q[, _3dImage[, handleMissingValues[, ddepth]]] ) -> _3dImage
```

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

Reprojects a disparity image to 3D space.

Parameters

- **disparity** Input single-channel 8-bit unsigned, 16-bit signed, 32-bit signed or 32-bit floating-point disparity image. The values of 8-bit / 16-bit signed formats are assumed to have no fractional bits. If the disparity is 16-bit signed format, as computed by **StereoBM** or **StereoSGBM** and maybe other algorithms, it should be divided by 16 (and scaled to float) before being used here.
- **_3dImage** Output 3-channel floating-point image of the same size as disparity. Each element of _3dImage(x,y) contains 3D coordinates of the point (x,y) computed from the disparity map. If one uses Q obtained by **stereoRectify**, then the returned points are represented in the first camera's rectified coordinate system.
- **Q** 4×4 perspective transformation matrix that can be obtained with **stereoRectify**.
- **handleMissingValues** Indicates, whether the function should handle missing values (i.e. points where the disparity was not computed). If handleMissingValues=true, then pixels with the minimal disparity that corresponds to the outliers (see **StereoMatcher::compute**) are transformed to 3D points with a very large Z value (currently set to 10000).
- **ddepth** The optional output array depth. If it is -1, the output image will have CV_32F depth. ddepth can also be set to CV_16S, CV_32S or CV_32F.

The function transforms a single-channel disparity map to a 3-channel image representing a 3D surface. That is, for each pixel (x,y) and the corresponding disparity d=disparity(x,y) , it computes:

$$\begin{bmatrix} X \\ Y \\ Z \\ W \end{bmatrix} = Q \begin{bmatrix} x \\ y \\ \texttt{disparity}(x, y) \\ 1 \end{bmatrix}.$$

See also

To reproject a sparse set of points {(x,y,d),...} to 3D space, use **perspectiveTransform**.

◆ 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.

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

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** .

> **Note**
> More information about the computation of the derivative of a 3D rotation matrix with respect to its exponential coordinate can be found in:
> - A Compact Formula for the Derivative of a 3-D Rotation in Exponential Coordinates, Guillermo Gallego, Anthony J. Yezzi [96]
>
> Useful information on SE(3) and Lie Groups can be found in:
> - A tutorial on SE(3) transformation parameterizations and on-manifold optimization, Jose-Luis Blanco [29]
> - Lie Groups for 2D and 3D Transformation, Ethan Eade [78]
> - A micro Lie theory for state estimation in robotics, Joan Solà, Jérémie Deray, Dinesh Atchuthan [249]

◆ RQDecomp3x3()

```
Vec3d cv::RQDecomp3x3 ( InputArray   src,
                        OutputArray  mtxR,
                        OutputArray  mtxQ,
                        OutputArray  Qx = noArray(),
                        OutputArray  Qy = noArray(),
                        OutputArray  Qz = noArray() )
```

Python:

```
cv.RQDecomp3x3( src[, mtxR[, mtxQ[, Qx[, Qy[, Qz]]]]] ) -> retval, mtxR, mtxQ, Qx, Qy, Qz
```

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

Computes an RQ decomposition of 3x3 matrices.

Parameters

- **src** 3x3 input matrix.
- **mtxR** Output 3x3 upper-triangular matrix.
- **mtxQ** Output 3x3 orthogonal matrix.
- **Qx** Optional output 3x3 rotation matrix around x-axis.
- **Qy** Optional output 3x3 rotation matrix around y-axis.
- **Qz** Optional output 3x3 rotation matrix around z-axis.

The function computes a RQ decomposition using the given rotations. This function is used in **decomposeProjectionMatrix** to decompose the left 3x3 submatrix of a projection matrix into a camera and a rotation matrix.

It optionally returns three rotation matrices, one for each axis, and the three Euler angles in degrees (as the return value) that could be used in OpenGL. Note, there is always more than one sequence of rotations about the three principal axes that results in the same orientation of an object, e.g. see [248] . Returned three rotation matrices and corresponding three Euler angles are only one of the possible solutions.

◆ sampsonDistance()

```
double cv::sampsonDistance ( InputArray pt1,
                             InputArray pt2,
                             InputArray F )
```

Python:

```
cv.sampsonDistance( pt1, pt2, F ) -> retval
```

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

Calculates the Sampson Distance between two points.

The function **cv::sampsonDistance** calculates and returns the first order approximation of the geometric error as:

$$sd(\texttt{pt1} , \texttt{pt2})= \frac{(\texttt{pt2}^t \cdot \texttt{F} \cdot \texttt{pt1})^2} {((\texttt{F} \cdot \texttt{pt1})(0))^2 + ((\texttt{F} \cdot \texttt{pt1})(1))^2 + ((\texttt{F}^t \cdot \texttt{pt2})(0))^2 + ((\texttt{F}^t \cdot \texttt{pt2})(1))^2}$$

The fundamental matrix may be calculated using the **findFundamentalMat** function. See [116] 11.4.3 for details.

Parameters

- **pt1** first homogeneous 2d point
- **pt2** second homogeneous 2d point
- **F** fundamental matrix

Returns

The computed Sampson distance.

◆ solveP3P()

```
int cv::solveP3P ( InputArray           objectPoints,
                   InputArray           imagePoints,
                   InputArray           cameraMatrix,
                   InputArray           distCoeffs,
                   OutputArrayOfArrays  rvecs,
                   OutputArrayOfArrays  tvecs,
                   int                  flags )
```

Python:

```
cv.solveP3P( objectPoints, imagePoints, cameraMatrix, distCoeffs, flags[, rvecs[, tvecs]] ) -> retval, rvecs, tvecs
```

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

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

See also

Perspective-n-Point (PnP) pose computation

Parameters

- **objectPoints** Array of object points in the object coordinate space, 3x3 1-channel or 1x3/3x1 3-channel. vector<Point3f> can be also passed here.
- **imagePoints** Array of corresponding image points, 3x2 1-channel or 1x3/3x1 2-channel. vector<Point2f> can be also passed here.
- **cameraMatrix** Input 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 NULL/empty, the zero distortion coefficients are assumed.
- **rvecs** Output rotation vectors (see Rodrigues) that, together with tvecs, brings points from the model coordinate system to the camera coordinate system. A P3P problem has up to 4 solutions.
- **tvecs** Output translation vectors.
- **flags** Method for solving a P3P problem:
  - SOLVEPNP_P3P Method is based on the paper of X.S. Gao, X.-R. Hou, J. Tang, H.-F. Chang "Complete Solution Classification for the Perspective-Three-Point Problem" ([97]).
  - SOLVEPNP_AP3P Method is based on the paper of T. Ke and S. Roumeliotis. "An Efficient Algebraic Solution to the Perspective-Three-Point Problem" ([144]).

The function estimates the object pose given 3 object points, their corresponding image projections, as well as the camera intrinsic matrix and the distortion coefficients.

> **Note**
> The solutions are sorted by reprojection errors (lowest to highest).

◆ 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.

See also

Perspective-n-Point (PnP) pose computation

This function returns the rotation and the translation vectors that transform a 3D point expressed in the object coordinate frame to the camera coordinate frame, using different methods:

- P3P methods (SOLVEPNP_P3P, SOLVEPNP_AP3P): need 4 input points to return a unique solution.
- SOLVEPNP_IPPE Input points must be >= 4 and object points must be coplanar.
- SOLVEPNP_IPPE_SQUARE Special case suitable for marker pose estimation. Number of input points must be 4. Object points must be defined in the following order:
  - point 0: [-squareLength / 2, squareLength / 2, 0]
  - point 1: [squareLength / 2, squareLength / 2, 0]
  - point 2: [squareLength / 2, -squareLength / 2, 0]
  - point 3: [-squareLength / 2, -squareLength / 2, 0]
- for all the other flags, number of input points must be >= 4 and object points can be in any configuration.

Parameters

- **objectPoints** Array of object points in the object coordinate space, Nx3 1-channel or 1xN/Nx1 3-channel, where N is the number of points. vector<Point3d> can be also passed here.
- **imagePoints** Array of corresponding image points, Nx2 1-channel or 1xN/Nx1 2-channel, where N is the number of points. vector<Point2d> can be also passed here.
- **cameraMatrix** Input 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 NULL/empty, the zero distortion coefficients are assumed.
- **rvec** Output rotation vector (see Rodrigues) that, together with tvec, brings points from the model coordinate system to the camera coordinate system.
- **tvec** Output translation vector.
- **useExtrinsicGuess** Parameter used for SOLVEPNP_ITERATIVE. If true (1), the function uses the provided rvec and tvec values as initial approximations of the rotation and translation vectors, respectively, and further optimizes them.
- **flags** Method for solving a PnP problem: see calib3d_solvePnP_flags

More information about Perspective-n-Points is described in Perspective-n-Point (PnP) pose computation

> **Note**
> - An example of how to use solvePnP for planar augmented reality can be found at opencv_source_code/samples/python/plane_ar.py
> - If you are using Python:
>   - Numpy array slices won't work as input because solvePnP requires contiguous arrays (enforced by the assertion using cv::Mat::checkVector() around line 55 of modules/calib3d/src/solvepnp.cpp version 2.4.9)
>   - The P3P algorithm requires image points to be in an array of shape (N,1,2) due to its calling of undistortPoints (around line 75 of modules/calib3d/src/solvepnp.cpp version 2.4.9) which requires 2-channel information.
>   - Thus, given some data D = np.array(...) where D.shape = (N,M), in order to use a subset of it as, e.g., imagePoints, one must effectively copy it into a new array: imagePoints = np.ascontiguousarray(D[:,:2]).reshape((N,1,2))
> - The methods SOLVEPNP_DLS and SOLVEPNP_UPNP cannot be used as the current implementations are unstable and sometimes give completely wrong results. If you pass one of these two flags, SOLVEPNP_EPNP method will be used instead.
> - The minimum number of points is 4 in the general case. In the case of SOLVEPNP_P3P and SOLVEPNP_AP3P methods, it is required to use exactly 4 points (the first 3 points are used to estimate all the solutions of the P3P problem, the last one is used to retain the best solution that minimizes the reprojection error).
> - With SOLVEPNP_ITERATIVE method and `useExtrinsicGuess=true`, the minimum number of points is 3 (3 points are sufficient to compute a pose but there are up to 4 solutions). The initial solution should be close to the global solution to converge.
> - With SOLVEPNP_IPPE input points must be >= 4 and object points must be coplanar.
> - With SOLVEPNP_IPPE_SQUARE this is a special case suitable for marker pose estimation. Number of input points must be 4. Object points must be defined in the following order:
>   - point 0: [-squareLength / 2, squareLength / 2, 0]
>   - point 1: [squareLength / 2, squareLength / 2, 0]
>   - point 2: [squareLength / 2, -squareLength / 2, 0]
>   - point 3: [-squareLength / 2, -squareLength / 2, 0]

- With SOLVEPNP_SQPNP input points must be >= 3

◆ solvePnPGeneric()

```
int cv::solvePnPGeneric ( InputArray            objectPoints,
                          InputArray            imagePoints,
                          InputArray            cameraMatrix,
                          InputArray            distCoeffs,
                          OutputArrayOfArrays   rvecs,
                          OutputArrayOfArrays   tvecs,
                          bool                  useExtrinsicGuess = false,
                          SolvePnPMethod        flags = SOLVEPNP_ITERATIVE,
                          InputArray            rvec = noArray(),
                          InputArray            tvec = noArray(),
                          OutputArray           reprojectionError = noArray() )
```

Python:

```
cv.solvePnPGeneric( objectPoints, imagePoints, cameraMatrix, distCoeffs[, rvecs[, tvecs[, useExtrinsicGuess[, flags[, rvec[, tvec[, reprojectionError]]]]]]] ) -> retval, rvecs, tvecs, reprojectionError
```

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

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

See also
    Perspective-n-Point (PnP) pose computation

This function returns a list of all the possible solutions (a solution is a <rotation vector, translation vector> couple), depending on the number of input points and the chosen method:

- P3P methods (**SOLVEPNP_P3P**, **SOLVEPNP_AP3P**): 3 or 4 input points. Number of returned solutions can be between 0 and 4 with 3 input points.
- **SOLVEPNP_IPPE** Input points must be >= 4 and object points must be coplanar. Returns 2 solutions.
- **SOLVEPNP_IPPE_SQUARE** Special case suitable for marker pose estimation. Number of input points must be 4 and 2 solutions are returned. Object points must be defined in the following order:
  - point 0: [-squareLength / 2, squareLength / 2, 0]
  - point 1: [squareLength / 2, squareLength / 2, 0]
  - point 2: [squareLength / 2, -squareLength / 2, 0]
  - point 3: [-squareLength / 2, -squareLength / 2, 0]
- for all the other flags, number of input points must be >= 4 and object points can be in any configuration. Only 1 solution is returned.

Parameters

objectPoints	Array of object points in the object coordinate space, Nx3 1-channel or 1xN/Nx1 3-channel, where N is the number of points. vector<Point3d> can be also passed here.
imagePoints	Array of corresponding image points, Nx2 1-channel or 1xN/Nx1 2-channel, where N is the number of points. vector<Point2d> can be also passed here.
cameraMatrix	Input 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 NULL/empty, the zero distortion coefficients are assumed.
rvecs	Vector of output rotation vectors (see **Rodrigues**) that, together with tvecs, brings points from the model coordinate system to the camera coordinate system.
tvecs	Vector of output translation vectors.
useExtrinsicGuess	Parameter used for **SOLVEPNP_ITERATIVE**. If true (1), the function uses the provided rvec and tvec values as initial approximations of the rotation and translation vectors, respectively, and further optimizes them.
flags	Method for solving a PnP problem: see **calib3d_solvePnP_flags**
rvec	Rotation vector used to initialize an iterative PnP refinement algorithm, when flag is **SOLVEPNP_ITERATIVE** and useExtrinsicGuess is set to true.
tvec	Translation vector used to initialize an iterative PnP refinement algorithm, when flag is **SOLVEPNP_ITERATIVE** and useExtrinsicGuess is set to true.
reprojectionError	Optional vector of reprojection error, that is the RMS error ($\text{RMSE} = \sqrt{\frac{\sum_i^N (\hat{y_i} - y_i)^2}{N}}$) between the input image points and the 3D object points projected with the estimated pose.

More information is described in **Perspective-n-Point (PnP) pose computation**

Note

- An example of how to use solvePnP for planar augmented reality can be found at opencv_source_code/samples/python/plane_ar.py
- If you are using Python:
  - Numpy array slices won't work as input because solvePnP requires contiguous arrays (enforced by the assertion using **cv::Mat::checkVector()** around line 55 of modules/calib3d/src/solvepnp.cpp version 2.4.9)
  - The P3P algorithm requires image points to be in an array of shape (N,1,2) due to its calling of **undistortPoints** (around line 75 of modules/calib3d/src/solvepnp.cpp version 2.4.9) which requires 2-channel information.
  - Thus, given some data D = np.array(...) where D.shape = (N,M), in order to use a subset of it as, e.g., imagePoints, one must effectively copy it into a new array: imagePoints = np.ascontiguousarray(D[:,:2]).reshape((N,1,2))
- The methods **SOLVEPNP_DLS** and **SOLVEPNP_UPNP** cannot be used as the current implementations are unstable and sometimes give completely wrong results. If you pass one of these two flags, **SOLVEPNP_EPNP** method will be used instead.
- The minimum number of points is 4 in the general case. In the case of **SOLVEPNP_P3P** and **SOLVEPNP_AP3P** methods, it is required to use exactly 4 points (the first 3 points are used to estimate all the solutions of the P3P problem, the last one is used to retain the best solution that minimizes the reprojection error).
- With **SOLVEPNP_ITERATIVE** method and `useExtrinsicGuess=true`, the minimum number of points is 3 (3 points are sufficient to compute a pose but there are up to 4 solutions). The initial solution should be close to the global solution to converge.
- With **SOLVEPNP_IPPE** input points must be >= 4 and object points must be coplanar.
- With **SOLVEPNP_IPPE_SQUARE** this is a special case suitable for marker pose estimation. Number of input points must be 4. Object points must be defined in the following order:
  - point 0: [-squareLength / 2, squareLength / 2, 0]
  - point 1: [squareLength / 2, squareLength / 2, 0]
  - point 2: [squareLength / 2, -squareLength / 2, 0]
  - point 3: [-squareLength / 2, -squareLength / 2, 0]

◆ solvePnPRansac() [1/2]

```
bool cv::solvePnPRansac ( InputArray   objectPoints,
                          InputArray   imagePoints,
                          InputArray   cameraMatrix,
                          InputArray   distCoeffs,
                          OutputArray  rvec,
                          OutputArray  tvec,
                          bool         useExtrinsicGuess = false,
                          int          iterationsCount = 100,
                          float        reprojectionError = 8.0,
                          double       confidence = 0.99,
                          OutputArray  inliers = noArray(),
                          int          flags = SOLVEPNP_ITERATIVE )
```

Python:

```
cv.solvePnPRansac( objectPoints, imagePoints, cameraMatrix, distCoeffs[, rvec[, tvec[, useExtrinsicGuess[, iterationsCount[, reprojectionError[, confidence[, inliers[, flags]]]]]]]] ) -> retval, rvec, tvec, inliers
cv.solvePnPRansac( objectPoints, imagePoints, cameraMatrix, distCoeffs[, rvec[, tvec[, inliers[, params]]]] ) -> retval, cameraMatrix, rvec, tvec, inliers
```

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

Finds an object pose from 3D-2D point correspondences using the RANSAC scheme.

See also
    Perspective-n-Point (PnP) pose computation

Parameters

objectPoints	Array of object points in the object coordinate space, Nx3 1-channel or 1xN/Nx1 3-channel, where N is the number of points. vector<Point3d> can be also passed here.
imagePoints	Array of corresponding image points, Nx2 1-channel or 1xN/Nx1 2-channel, where N is the number of points. vector<Point2d> can be also passed here.
cameraMatrix	Input 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 NULL/empty, the zero distortion coefficients are assumed.
rvec	Output rotation vector (see **Rodrigues**) that, together with tvec, brings points from the model coordinate system to the camera coordinate system.
tvec	Output translation vector.
useExtrinsicGuess	Parameter used for **SOLVEPNP_ITERATIVE**. If true (1), the function uses the provided rvec and tvec values as initial approximations of the rotation and translation vectors, respectively, and further optimizes them.
iterationsCount	Number of iterations.
reprojectionError	Inlier threshold value used by the RANSAC procedure. The parameter value is the maximum allowed distance between the observed and computed point projections to consider it an inlier.
confidence	The probability that the algorithm produces a useful result.
inliers	Output vector that contains indices of inliers in objectPoints and imagePoints .
flags	Method for solving a PnP problem (see **solvePnP**).

The function estimates an object pose given a set of object points, their corresponding image projections, as well as the camera intrinsic matrix and the distortion coefficients. This function finds such a pose that minimizes reprojection error, that is, the sum of squared distances between the observed projections imagePoints and the projected (using **projectPoints**) objectPoints. The use of RANSAC makes the function resistant to outliers.

Note

- An example of how to use solvePNPRansac for object detection can be found at opencv_source_code/samples/cpp/tutorial_code/calib3d/real_time_pose_estimation/
- The default method used to estimate the camera pose for the Minimal Sample Sets step is **SOLVEPNP_EPNP**. Exceptions are:
  - if you choose **SOLVEPNP_P3P** or **SOLVEPNP_AP3P**, these methods will be used.
  - if the number of input points is equal to 4, **SOLVEPNP_P3P** is used.
- The method used to estimate the camera pose using all the inliers is defined by the flags parameters unless it is equal to **SOLVEPNP_P3P** or **SOLVEPNP_AP3P**. In this case, the method **SOLVEPNP_EPNP** will be used instead.

◆ solvePnPRansac() [2/2]

```
bool cv::solvePnPRansac ( InputArray            objectPoints,
                          InputArray            imagePoints,
                          InputOutputArray      cameraMatrix,
                          InputArray            distCoeffs,
                          OutputArray           rvec,
                          OutputArray           tvec,
                          OutputArray           inliers,
                          const UsacParams &    params = UsacParams() )
```

Python:

```
cv.solvePnPRansac( objectPoints, imagePoints, cameraMatrix, distCoeffs[, rvec[, tvec[, useExtrinsicGuess[, iterationsCount[, reprojectionError[, confidence[, inliers[, flags]]]]]]]] ) -> retval, rvec, tvec, inliers
cv.solvePnPRansac( objectPoints, imagePoints, cameraMatrix, distCoeffs[, rvec[, tvec[, inliers[, params]]]] ) -> retval, cameraMatrix, rvec, tvec, inliers
```

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

◆ solvePnPRefineLM()

```
void cv::solvePnPRefineLM ( InputArray          objectPoints,
                            InputArray          imagePoints,
                            InputArray          cameraMatrix,
                            InputArray          distCoeffs,
                            InputOutputArray    rvec,
                            InputOutputArray    tvec,
                            TermCriteria        criteria = TermCriteria(TermCriteria::EPS+TermCriteria::COUNT, 20, FLT_EPSILON) )
```

Python:

```
cv.solvePnPRefineLM( objectPoints, imagePoints, cameraMatrix, distCoeffs, rvec, tvec[, criteria] ) -> rvec, tvec
```

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

Refine a pose (the translation and the rotation that transform a 3D point expressed in the object coordinate frame to the camera coordinate frame) from a 3D-2D point correspondences and starting from an initial solution.

See also
Perspective-n-Point (PnP) pose computation

Parameters

- **objectPoints** Array of object points in the object coordinate space, Nx3 1-channel or 1xN/Nx1 3-channel, where N is the number of points. vector<Point3d> can also be passed here.
- **imagePoints** Array of corresponding image points, Nx2 1-channel or 1xN/Nx1 2-channel, where N is the number of points. vector<Point2d> can also be passed here.
- **cameraMatrix** Input 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 NULL/empty, the zero distortion coefficients are assumed.
- **rvec** Input/Output rotation vector (see Rodrigues) that, together with tvec, brings points from the model coordinate system to the camera coordinate system. Input values are used as an initial solution.
- **tvec** Input/Output translation vector. Input values are used as an initial solution.
- **criteria** Criteria when to stop the Levenberg-Marquard iterative algorithm.

The function refines the object pose given at least 3 object points, their corresponding image projections, an initial solution for the rotation and translation vector, as well as the camera intrinsic matrix and the distortion coefficients. The function minimizes the projection error with respect to the rotation and the translation vectors, according to a Levenberg-Marquardt iterative minimization [178] [77] process.

◆ solvePnPRefineVVS()

```
void cv::solvePnPRefineVVS ( InputArray          objectPoints,
                             InputArray          imagePoints,
                             InputArray          cameraMatrix,
                             InputArray          distCoeffs,
                             InputOutputArray    rvec,
                             InputOutputArray    tvec,
                             TermCriteria        criteria = TermCriteria(TermCriteria::EPS+TermCriteria::COUNT, 20, FLT_EPSILON),
                             double              VVSlambda = 1 )
```

Python:

```
cv.solvePnPRefineVVS( objectPoints, imagePoints, cameraMatrix, distCoeffs, rvec, tvec[, criteria[, VVSlambda]] ) -> rvec, tvec
```

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

Refine a pose (the translation and the rotation that transform a 3D point expressed in the object coordinate frame to the camera coordinate frame) from a 3D-2D point correspondences and starting from an initial solution.

See also
Perspective-n-Point (PnP) pose computation

Parameters

- **objectPoints** Array of object points in the object coordinate space, Nx3 1-channel or 1xN/Nx1 3-channel, where N is the number of points. vector<Point3d> can also be passed here.
- **imagePoints** Array of corresponding image points, Nx2 1-channel or 1xN/Nx1 2-channel, where N is the number of points. vector<Point2d> can also be passed here.
- **cameraMatrix** Input 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 NULL/empty, the zero distortion coefficients are assumed.
- **rvec** Input/Output rotation vector (see Rodrigues) that, together with tvec, brings points from the model coordinate system to the camera coordinate system. Input values are used as an initial solution.
- **tvec** Input/Output translation vector. Input values are used as an initial solution.
- **criteria** Criteria when to stop the Levenberg-Marquard iterative algorithm.
- **VVSlambda** Gain for the virtual visual servoing control law, equivalent to the α gain in the Damped Gauss-Newton formulation.

The function refines the object pose given at least 3 object points, their corresponding image projections, an initial solution for the rotation and translation vector, as well as the camera intrinsic matrix and the distortion coefficients. The function minimizes the projection error with respect to the rotation and the translation vectors, using a virtual visual servoing (VVS) [51] [182] scheme.

◆ stereoCalibrate() [1/3]

```
double cv::stereoCalibrate ( InputArrayOfArrays  objectPoints,
                             InputArrayOfArrays  imagePoints1,
                             InputArrayOfArrays  imagePoints2,
                             InputOutputArray    cameraMatrix1,
                             InputOutputArray    distCoeffs1,
                             InputOutputArray    cameraMatrix2,
                             InputOutputArray    distCoeffs2,
                             Size                imageSize,
                             InputOutputArray    R,
                             InputOutputArray    T,
                             OutputArray         E,
                             OutputArray         F,
                             OutputArray         perViewErrors,
                             int                 flags = CALIB_FIX_INTRINSIC,
                             TermCriteria        criteria = TermCriteria(TermCriteria::COUNT+TermCriteria::EPS, 30, 1e-6) )
```

Python:

```
cv.stereoCalibrate( objectPoints, imagePoints1, imagePoints2, cameraMatrix1, distCoeffs1, cameraMatrix2, distCoeffs2, imageSize[, R[, T[, E[, F[, flags[, criteria]]]]]] ) -> retval, cameraMatrix1, distCoeffs1, cameraMatrix2, distCoeffs2, R, T, E, F
cv.stereoCalibrate( objectPoints, imagePoints1, imagePoints2, cameraMatrix1, distCoeffs1, cameraMatrix2, distCoeffs2, imageSize, R, T[, E[, F[, perViewErrors[, flags[, criteria]]]]] ) -> retval, cameraMatrix1, distCoeffs1, cameraMatrix2, distCoeffs2, R, T, E, F, perViewErrors
cv.stereoCalibrateExtended( objectPoints, imagePoints1, imagePoints2, cameraMatrix1, distCoeffs1, cameraMatrix2, distCoeffs2, imageSize, R, T[, E[, F[, rvecs[, tvecs[, perViewErrors[, flags[, criteria]]]]]]] ) -> retval, cameraMatrix1, distCoeffs1, cameraMatrix2, distCoeffs2, R, T, E, F, rvecs, tvecs, perViewErrors
```

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

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

◆ stereoCalibrate() [2/3]

```
double cv::stereoCalibrate ( InputArrayOfArrays   objectPoints,
                             InputArrayOfArrays   imagePoints1,
                             InputArrayOfArrays   imagePoints2,
                             InputOutputArray     cameraMatrix1,
                             InputOutputArray     distCoeffs1,
                             InputOutputArray     cameraMatrix2,
                             InputOutputArray     distCoeffs2,
                             Size                 imageSize,
                             InputOutputArray     R,
                             InputOutputArray     T,
                             OutputArray          E,
                             OutputArray          F,
                             OutputArrayOfArrays  rvecs,
                             OutputArrayOfArrays  tvecs,
                             OutputArray          perViewErrors,
                             int                  flags = CALIB_FIX_INTRINSIC,
                             TermCriteria         criteria = TermCriteria(TermCriteria::COUNT+TermCriteria::EPS, 30, 1e-6) )
```

Python:

```
cv.stereoCalibrate( objectPoints, imagePoints1, imagePoints2, cameraMatrix1, distCoeffs1, cameraMatrix2, distCoeffs2, imageSize[, R[, T[, E[, F[, flags[, criteria]]]]]] ) -> retval, cameraMatrix1, distCoeffs1, cameraMatrix2, distCoeffs2, R, T, E, F
cv.stereoCalibrate( objectPoints, imagePoints1, imagePoints2, cameraMatrix1, distCoeffs1, cameraMatrix2, distCoeffs2, imageSize, R, T[, E[, F[, perViewErrors[, flags[, criteria]]]]] ) -> retval, cameraMatrix1, distCoeffs1, cameraMatrix2, distCoeffs2, R, T, E, F, perViewErrors
cv.stereoCalibrateExtended( objectPoints, imagePoints1, imagePoints2, cameraMatrix1, distCoeffs1, cameraMatrix2, distCoeffs2, imageSize, R, T[, E[, F[, rvecs[, tvecs[, perViewErrors[, flags[, criteria]]]]]]] ) -> retval, cameraMatrix1, distCoeffs1, cameraMatrix2, distCoeffs2, R, T, E, F, rvecs, tvecs, perViewErrors
```

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

Calibrates a stereo camera set up. This function finds the intrinsic parameters for each of the two cameras and the extrinsic parameters between the two cameras.

Parameters

- **objectPoints** Vector of vectors of the calibration pattern points. The same structure as in **calibrateCamera**. For each pattern view, both cameras need to see the same object points. Therefore, objectPoints.size(), imagePoints1.size(), and imagePoints2.size() need to be equal as well as objectPoints[i].size(), imagePoints1[i].size(), and imagePoints2[i].size() need to be equal for each i.
- **imagePoints1** Vector of vectors of the projections of the calibration pattern points, observed by the first camera. The same structure as in **calibrateCamera**.
- **imagePoints2** Vector of vectors of the projections of the calibration pattern points, observed by the second camera. The same structure as in **calibrateCamera**.
- **cameraMatrix1** Input/output camera intrinsic matrix for the first camera, the same as in **calibrateCamera**. Furthermore, for the stereo case, additional flags may be used, see below.
- **distCoeffs1** Input/output vector of distortion coefficients, the same as in **calibrateCamera**.
- **cameraMatrix2** Input/output second camera intrinsic matrix for the second camera. See description for cameraMatrix1.
- **distCoeffs2** Input/output lens distortion coefficients for the second camera. See description for distCoeffs1.
- **imageSize** Size of the image used only to initialize the camera intrinsic matrices.
- **R** Output rotation matrix. Together with the translation vector T, this matrix brings points given in the first camera's coordinate system to points in the second camera's coordinate system. In more technical terms, the tuple of R and T performs a change of basis from the first camera's coordinate system to the second camera's coordinate system. Due to its duality, this tuple is equivalent to the position of the first camera with respect to the second camera coordinate system.
- **T** Output translation vector, see description above.
- **E** Output essential matrix.
- **F** Output fundamental matrix.
- **rvecs** Output vector of rotation vectors (**Rodrigues**) estimated for each pattern view in the coordinate system of the first camera of the stereo pair (e.g. std::vector<cv::Mat>). More in detail, each i-th rotation vector together with the corresponding i-th translation vector (see the next output parameter description) brings the calibration pattern from the object coordinate space (in which object points are specified) to the camera coordinate space of the first camera of the stereo pair. In more technical terms, the tuple of the i-th rotation and translation vector performs a change of basis from object coordinate space to camera coordinate space of the first camera of the stereo pair.
- **tvecs** Output vector of translation vectors estimated for each pattern view, see parameter description of previous output parameter (rvecs).
- **perViewErrors** Output vector of the RMS re-projection error estimated for each pattern view.
- **flags** Different flags that may be zero or a combination of the following values:
  - **CALIB_FIX_INTRINSIC** Fix cameraMatrix? and distCoeffs? so that only R, T, E, and F matrices are estimated.
  - **CALIB_USE_INTRINSIC_GUESS** Optimize some or all of the intrinsic parameters according to the specified flags. Initial values are provided by the user.
  - **CALIB_USE_EXTRINSIC_GUESS** R and T contain valid initial values that are optimized further. Otherwise R and T are initialized to the median value of the pattern views (each dimension separately).
  - **CALIB_FIX_PRINCIPAL_POINT** Fix the principal points during the optimization.
  - **CALIB_FIX_FOCAL_LENGTH** Fix $f^{(j)}_x$ and $f^{(j)}_y$.
  - **CALIB_FIX_ASPECT_RATIO** Optimize $f^{(j)}_y$. Fix the ratio $f^{(j)}_x/f^{(j)}_y$
  - **CALIB_SAME_FOCAL_LENGTH** Enforce $f^{(0)}_x = f^{(1)}_x$ and $f^{(0)}_y = f^{(1)}_y$.
  - **CALIB_ZERO_TANGENT_DIST** Set tangential distortion coefficients for each camera to zeros and fix there.
  - **CALIB_FIX_K1**,..., **CALIB_FIX_K6** Do not change the corresponding radial distortion coefficient during the optimization. If **CALIB_USE_INTRINSIC_GUESS** is set, the coefficient from the supplied distCoeffs matrix is used. Otherwise, it is set to 0.
  - **CALIB_RATIONAL_MODEL** Enable coefficients k4, k5, and k6. To provide the backward compatibility, this extra flag should be explicitly specified to make the calibration function use the rational model and return 8 coefficients. If the flag is not set, the function computes and returns only 5 distortion coefficients.
  - **CALIB_THIN_PRISM_MODEL** Coefficients s1, s2, s3 and s4 are enabled. To provide the backward compatibility, this extra flag should be explicitly specified to make the calibration function use the thin prism model and return 12 coefficients. If the flag is not set, the function computes and returns only 5 distortion coefficients.
  - **CALIB_FIX_S1_S2_S3_S4** The thin prism distortion coefficients are not changed during the optimization. If **CALIB_USE_INTRINSIC_GUESS** is set, the coefficient from the supplied distCoeffs matrix is used. Otherwise, it is set to 0.
  - **CALIB_TILTED_MODEL** Coefficients tauX and tauY are enabled. To provide the backward compatibility, this extra flag should be explicitly specified to make the calibration function use the tilted sensor model and return 14 coefficients. If the flag is not set, the function computes and returns only 5 distortion coefficients.
  - **CALIB_FIX_TAUX_TAUY** The coefficients of the tilted sensor model are not changed during the optimization. If **CALIB_USE_INTRINSIC_GUESS** is set, the coefficient from the supplied distCoeffs matrix is used. Otherwise, it is set to 0.
- **criteria** Termination criteria for the iterative optimization algorithm.

The function estimates the transformation between two cameras making a stereo pair. If one computes the poses of an object relative to the first camera and to the second camera, (R_1, T_1) and (R_2, T_2), respectively, for a stereo camera where the relative position and orientation between the two cameras are fixed, then those poses definitely relate to each other. This means, if the relative position and orientation (R, T) of the two cameras is known, it is possible to compute (R_2, T_2) when (R_1, T_1) is given. This is what the described function does. It computes (R, T) such that:

$$R_2 = R R_1$$

$$T_2 = R T_1 + T.$$

Therefore, one can compute the coordinate representation of a 3D point for the second camera's coordinate system when given the point's coordinate representation in the first camera's coordinate system:

$$\begin{bmatrix} X_2 \\ Y_2 \\ Z_2 \\ 1 \end{bmatrix} = \begin{bmatrix} R & T \\ 0 & 1 \end{bmatrix} \begin{bmatrix} X_1 \\ Y_1 \\ Z_1 \\ 1 \end{bmatrix}.$$

Optionally, it computes the essential matrix E:

$$E = \begin{bmatrix} 0 & -T_2 & T_1 \\ T_2 & 0 & -T_0 \\ -T_1 & T_0 & 0 \end{bmatrix} R$$

where T_i are components of the translation vector $T : T = [T_0, T_1, T_2]^T$. And the function can also compute the fundamental matrix F:

$$F = cameraMatrix2^{-T} \cdot E \cdot cameraMatrix1^{-1}$$

Besides the stereo-related information, the function can also perform a full calibration of each of the two cameras. However, due to the high dimensionality of the parameter space and noise in the input data, the function can diverge from the correct solution. If the intrinsic parameters can be estimated with high accuracy for each of the cameras individually (for example, using **calibrateCamera**), you are recommended to do so and then pass **CALIB_FIX_INTRINSIC** flag to the function along with the computed intrinsic parameters. Otherwise, if all the parameters are estimated at once, it makes sense to restrict some parameters, for example, pass **CALIB_SAME_FOCAL_LENGTH** and **CALIB_ZERO_TANGENT_DIST** flags, which is usually a reasonable assumption.

Similarly to **calibrateCamera**, the function minimizes the total re-projection error for all the points in all the available views from both cameras. The function returns the final value of the re-projection error.

◆ stereoCalibrate() [3/3]

```
double cv::stereoCalibrate ( InputArrayOfArrays   objectPoints,
                             InputArrayOfArrays   imagePoints1,
                             InputArrayOfArrays   imagePoints2,
                             InputOutputArray     cameraMatrix1,
                             InputOutputArray     distCoeffs1,
                             InputOutputArray     cameraMatrix2,
                             InputOutputArray     distCoeffs2,
                             Size                 imageSize,
                             OutputArray          R,
                             OutputArray          T,
                             OutputArray          E,
                             OutputArray          F,
                             int                  flags = CALIB_FIX_INTRINSIC,
                             TermCriteria         criteria = TermCriteria(TermCriteria::COUNT+TermCriteria::EPS, 30, 1e-6) )
```

Python:

```
cv.stereoCalibrate( objectPoints, imagePoints1, imagePoints2, cameraMatrix1, distCoeffs1, cameraMatrix2, distCoeffs2, imageSize[, R[, T[, E[, F[, flags[, criteria]]]]]] ) -> retval, cameraMatrix1, distCoeffs1, cameraMatrix2, distCoeffs2, R, T, E, F
cv.stereoCalibrate( objectPoints, imagePoints1, imagePoints2, cameraMatrix1, distCoeffs1, cameraMatrix2, distCoeffs2, imageSize, R, T[, E[, F[, perViewErrors[, flags[, criteria]]]]] ) -> retval, cameraMatrix1, distCoeffs1, cameraMatrix2, distCoeffs2, R, T, E, F, perViewErrors
cv.stereoCalibrateExtended( objectPoints, imagePoints1, imagePoints2, cameraMatrix1, distCoeffs1, cameraMatrix2, distCoeffs2, imageSize, R, T[, E[, F[, rvecs[, tvecs[, perViewErrors[, flags[, criteria]]]]]]] ) -> retval, cameraMatrix1, distCoeffs1, cameraMatrix2, distCoeffs2, R, T, E, F, rvecs, tvecs, perViewErrors
```

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

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

◆ stereoRectify()

```
void cv::stereoRectify ( InputArray   cameraMatrix1,
                         InputArray   distCoeffs1,
                         InputArray   cameraMatrix2,
                         InputArray   distCoeffs2,
                         Size         imageSize,
                         InputArray   R,
                         InputArray   T,
                         OutputArray  R1,
                         OutputArray  R2,
                         OutputArray  P1,
                         OutputArray  P2,
                         OutputArray  Q,
                         int          flags = CALIB_ZERO_DISPARITY,
                         double       alpha = -1,
                         Size         newImageSize = Size(),
                         Rect *       validPixROI1 = 0,
                         Rect *       validPixROI2 = 0 )
```

Python:

```
cv.stereoRectify( cameraMatrix1, distCoeffs1, cameraMatrix2, distCoeffs2, imageSize, R, T[, R1[, R2[, P1[, P2[, Q[, flags[, alpha[, newImageSize]]]]]]]] ) -> R1, R2, P1, P2, Q, validPixROI1, validPixROI2
```

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

Computes rectification transforms for each head of a calibrated stereo camera.

Parameters

cameraMatrix1	First camera intrinsic matrix.
distCoeffs1	First camera distortion parameters.
cameraMatrix2	Second camera intrinsic matrix.
distCoeffs2	Second camera distortion parameters.
imageSize	Size of the image used for stereo calibration.
R	Rotation matrix from the coordinate system of the first camera to the second camera, see **stereoCalibrate**.
T	Translation vector from the coordinate system of the first camera to the second camera, see **stereoCalibrate**.
R1	Output 3x3 rectification transform (rotation matrix) for the first camera. This matrix brings points given in the unrectified first camera's coordinate system to points in the rectified first camera's coordinate system. In more technical terms, it performs a change of basis from the unrectified first camera's coordinate system to the rectified first camera's coordinate system.
R2	Output 3x3 rectification transform (rotation matrix) for the second camera. This matrix brings points given in the unrectified second camera's coordinate system to points in the rectified second camera's coordinate system. In more technical terms, it performs a change of basis from the unrectified second camera's coordinate system to the rectified second camera's coordinate system.
P1	Output 3x4 projection matrix in the new (rectified) coordinate systems for the first camera, i.e. it projects points given in the rectified first camera coordinate system into the rectified first camera's image.
P2	Output 3x4 projection matrix in the new (rectified) coordinate systems for the second camera, i.e. it projects points given in the rectified first camera coordinate system into the rectified second camera's image.
Q	Output 4×4 disparity-to-depth mapping matrix (see **reprojectImageTo3D**).
flags	Operation flags that may be zero or **CALIB_ZERO_DISPARITY** . If the flag is set, the function makes the principal points of each camera have the same pixel coordinates in the rectified views. And if the flag is not set, the function may still shift the images in the horizontal or vertical direction (depending on the orientation of epipolar lines) to maximize the useful image area.
alpha	Free scaling parameter. If it is -1 or absent, the function performs the default scaling. Otherwise, the parameter should be between 0 and 1. alpha=0 means that the rectified images are zoomed and shifted so that only valid pixels are visible (no black areas after rectification). alpha=1 means that the rectified image is decimated and shifted so that all the pixels from the original images from the cameras are retained in the rectified images (no source image pixels are lost). Any intermediate value yields an intermediate result between those two extreme cases.
newImageSize	New image resolution after rectification. The same size should be passed to **initUndistortRectifyMap** (see the stereo_calib.cpp sample in OpenCV samples directory). When (0,0) is passed (default), it is set to the original imageSize . Setting it to a larger value can help you preserve details in the original image, especially when there is a big radial distortion.
validPixROI1	Optional output rectangles inside the rectified images where all the pixels are valid. If alpha=0 , the ROIs cover the whole images. Otherwise, they are likely to be smaller (see the picture below).
validPixROI2	Optional output rectangles inside the rectified images where all the pixels are valid. If alpha=0 , the ROIs cover the whole images. Otherwise, they are likely to be smaller (see the picture below).

The function computes the rotation matrices for each camera that (virtually) make both camera image planes the same plane. Consequently, this makes all the epipolar lines parallel and thus simplifies the dense stereo correspondence problem. The function takes the matrices computed by **stereoCalibrate** as input. As output, it provides two rotation matrices and also two projection matrices in the new coordinates. The function distinguishes the following two cases:

- **Horizontal stereo**: the first and the second camera views are shifted relative to each other mainly along the x-axis (with possible small vertical shift). In the rectified images, the corresponding epipolar lines in the left and right cameras are horizontal and have the same y-coordinate. P1 and P2 look like:

$$\texttt{P1} = \begin{bmatrix} f & 0 & cx_1 & 0 \\ 0 & f & cy & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

$$\texttt{P2} = \begin{bmatrix} f & 0 & cx_2 & T_x \cdot f \\ 0 & f & cy & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix},$$

$$\texttt{Q} = \begin{bmatrix} 1 & 0 & 0 & -cx_1 \\ 0 & 1 & 0 & -cy \\ 0 & 0 & 0 & f \\ 0 & 0 & -\frac{1}{T_x} & \frac{cx_1 - cx_2}{T_x} \end{bmatrix}$$

  where T_x is a horizontal shift between the cameras and $cx_1 = cx_2$ if **CALIB_ZERO_DISPARITY** is set.

- **Vertical stereo**: the first and the second camera views are shifted relative to each other mainly in the vertical direction (and probably a bit in the horizontal direction too). The epipolar lines in the rectified images are vertical and have the same x-coordinate. P1 and P2 look like:

$$\texttt{P1} = \begin{bmatrix} f & 0 & cx & 0 \\ 0 & f & cy_1 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

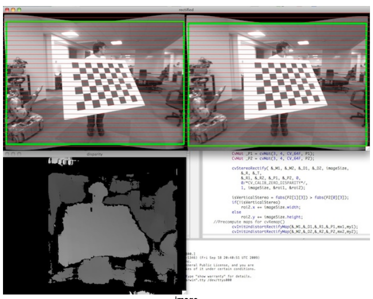
$$\texttt{P2} = \begin{bmatrix} f & 0 & cx & 0 \\ 0 & f & cy_2 & T_y \cdot f \\ 0 & 0 & 1 & 0 \end{bmatrix},$$

$$\texttt{Q} = \begin{bmatrix} 1 & 0 & 0 & -cx \\ 0 & 1 & 0 & -cy_1 \\ 0 & 0 & 0 & f \\ 0 & 0 & -\frac{1}{T_y} & \frac{cy_1 - cy_2}{T_y} \end{bmatrix}$$

  where T_y is a vertical shift between the cameras and $cy_1 = cy_2$ if **CALIB_ZERO_DISPARITY** is set.

As you can see, the first three columns of P1 and P2 will effectively be the new "rectified" camera matrices. The matrices, together with R1 and R2 , can then be passed to **initUndistortRectifyMap** to initialize the rectification map for each camera.

See below the screenshot from the stereo_calib.cpp sample. Some red horizontal lines pass through the corresponding image regions. This means that the images are well rectified, which is what most stereo correspondence algorithms rely on. The green rectangles are roi1 and roi2 . You see that their interiors are all valid pixels.



image

- stereoRectifyUncalibrated()

```
bool cv::stereoRectifyUncalibrated ( InputArray   points1,
                                     InputArray   points2,
                                     InputArray   F,
                                     Size         imgSize,
                                     OutputArray  H1,
                                     OutputArray  H2,
                                     double       threshold = 5 )
```

Python:

```
cv.stereoRectifyUncalibrated( points1, points2, F, imgSize[, H1[, H2[, threshold]]] ) -> retval, H1, H2
```

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

Computes a rectification transform for an uncalibrated stereo camera.

Parameters

- **points1** Array of feature points in the first image.
- **points2** The corresponding points in the second image. The same formats as in **findFundamentalMat** are supported.
- **F** Input fundamental matrix. It can be computed from the same set of point pairs using **findFundamentalMat** .
- **imgSize** Size of the image.
- **H1** Output rectification homography matrix for the first image.
- **H2** Output rectification homography matrix for the second image.
- **threshold** Optional threshold used to filter out the outliers. If the parameter is greater than zero, all the point pairs that do not comply with the epipolar geometry (that is, the points for which $|\texttt{points2[i]}^T \cdot \texttt{F} \cdot \texttt{points1[i]}| > \texttt{threshold}$) are rejected prior to computing the homographies. Otherwise, all the points are considered inliers.

The function computes the rectification transformations without knowing intrinsic parameters of the cameras and their relative position in the space, which explains the suffix "uncalibrated". Another related difference from **stereoRectify** is that the function outputs not the rectification transformations in the object (3D) space, but the planar perspective transformations encoded by the homography matrices H1 and H2 . The function implements the algorithm **[117]** .

> **Note**
> While the algorithm does not need to know the intrinsic parameters of the cameras, it heavily depends on the epipolar geometry. Therefore, if the camera lenses have a significant distortion, it would be better to correct it before computing the fundamental matrix and calling this function. For example, distortion coefficients can be estimated for each head of stereo camera separately by using **calibrateCamera** . Then, the images can be corrected using **undistort** , or just the point coordinates can be corrected with **undistortPoints** .

◆ triangulatePoints()

```
void cv::triangulatePoints ( InputArray   projMatr1,
                             InputArray   projMatr2,
                             InputArray   projPoints1,
                             InputArray   projPoints2,
                             OutputArray  points4D )
```

Python:

```
cv.triangulatePoints( projMatr1, projMatr2, projPoints1, projPoints2[, points4D] ) -> points4D
```

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

This function reconstructs 3-dimensional points (in homogeneous coordinates) by using their observations with a stereo camera.

Parameters

- **projMatr1** 3x4 projection matrix of the first camera, i.e. this matrix projects 3D points given in the world's coordinate system into the first image.
- **projMatr2** 3x4 projection matrix of the second camera, i.e. this matrix projects 3D points given in the world's coordinate system into the second image.
- **projPoints1** 2xN array of feature points in the first image. In the case of the c++ version, it can be also a vector of feature points or two-channel matrix of size 1xN or Nx1.
- **projPoints2** 2xN array of corresponding points in the second image. In the case of the c++ version, it can be also a vector of feature points or two-channel matrix of size 1xN or Nx1.
- **points4D** 4xN array of reconstructed points in homogeneous coordinates. These points are returned in the world's coordinate system.

> **Note**
> Keep in mind that all input data should be of float type in order for this function to work.
> If the projection matrices from **stereoRectify** are used, then the returned points are represented in the first camera's rectified coordinate system.

See also

reprojectImageTo3D

◆ undistort()

```
void cv::undistort ( InputArray   src,
                     OutputArray  dst,
                     InputArray   cameraMatrix,
                     InputArray   distCoeffs,
                     InputArray   newCameraMatrix = noArray() )
```

Python:

```
cv.undistort( src, cameraMatrix, distCoeffs[, dst[, newCameraMatrix]] ) -> dst
```

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

Transforms an image to compensate for lens distortion.

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.

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 $(k_1, k_2, p_1, p_2[, k_3[, k_4, k_5, k_6[, s_1, s_2, s_3, s_4[, \tau_x, \tau_y]]]])$ of 4, 5, 8, 12 or 14 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.

◆ undistortImagePoints()

```
void cv::undistortImagePoints ( InputArray   src,
                                OutputArray  dst,
                                InputArray   cameraMatrix,
                                InputArray   distCoeffs,
                                TermCriteria = TermCriteria(TermCriteria::MAX_ITER+TermCriteria::EPS, 5, 0.01) )
```

Python:

```
cv.undistortImagePoints( src, cameraMatrix, distCoeffs[, dst[, arg1]] ) -> dst
```

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

Compute undistorted image points position.

Parameters

- **src** Observed points position, 2xN/Nx2 1-channel or 1xN/Nx1 2-channel (CV_32FC2 or CV_64FC2) (or vector<Point2f>).
- **dst** Output undistorted points position (1xN/Nx1 2-channel or vector<Point2f>).
- **cameraMatrix** Camera matrix $\begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$.
- **distCoeffs** Distortion coefficients

◆ undistortPoints() [1/2]

```
void cv::undistortPoints ( InputArray   src,
                           OutputArray  dst,
                           InputArray   cameraMatrix,
                           InputArray   distCoeffs,
                           InputArray   R,
                           InputArray   P,
                           TermCriteria criteria )
```

Python:

```
cv.undistortPoints(     src, cameraMatrix, distCoeffs[, dst[, R[, P]]]      ) -> dst
cv.undistortPointsIter( src, cameraMatrix, distCoeffs, R, P, criteria[, dst] ) -> dst
```

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

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

> **Note**
> Default version of **undistortPoints** does 5 iterations to compute undistorted points.

◆ undistortPoints() [2/2]

```
void cv::undistortPoints ( InputArray   src,
                           OutputArray  dst,
                           InputArray   cameraMatrix,
                           InputArray   distCoeffs,
                           InputArray   R = noArray(),
                           InputArray   P = noArray() )
```

Python:

```
cv.undistortPoints(     src, cameraMatrix, distCoeffs[, dst[, R[, P]]]      ) -> dst
cv.undistortPointsIter( src, cameraMatrix, distCoeffs, R, P, criteria[, dst] ) -> dst
```

`#include <opencv2/calib3d.hpp>`

Computes the ideal point coordinates from the observed point coordinates.

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.

For each observed point coordinate (u, v) the function computes:

$$
\begin{array}{l}
x^{"} \leftarrow (u - c_x)/f_x \\
y^{"} \leftarrow (v - c_y)/f_y \\
(x', y') = undistort(x^{"}, y^{"}, \texttt{distCoeffs}) \\
{[X\,Y\,W]}^T \leftarrow R * [x'\,y'\,1]^T \\
x \leftarrow X/W \\
y \leftarrow Y/W \\
\text{only performed if P is specified:} \\
u' \leftarrow x {f'}_x + {c'}_x \\
v' \leftarrow y {f'}_y + {c'}_y
\end{array}
$$

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).

Parameters

src	Observed point coordinates, 2xN/Nx2 1-channel or 1xN/Nx1 2-channel (CV_32FC2 or CV_64FC2) (or vector<Point2f>).
dst	Output ideal point coordinates (1xN/Nx1 2-channel or vector<Point2f>) after undistortion and reverse perspective transformation. If matrix P is identity or omitted, dst will contain normalized point coordinates.
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 $(k_1, k_2, p_1, p_2[, k_3[, k_4, k_5, k_6[, s_1, s_2, s_3, s_4[, \tau_x, \tau_y]]]])$ of 4, 5, 8, 12 or 14 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) $\begin{bmatrix} {f'}_x & 0 & {c'}_x & t_x \\ 0 & {f'}_y & {c'}_y & t_y \\ 0 & 0 & 1 & t_z \end{bmatrix}$. P1 or P2 computed by **stereoRectify** can be passed here. If the matrix is empty, the identity new camera matrix is used.

◆ validateDisparity()

```
void cv::validateDisparity ( InputOutputArray disparity,
                             InputArray       cost,
                             int              minDisparity,
                             int              numberOfDisparities,
                             int              disp12MaxDisp = 1 )
```

Python:

```
cv.validateDisparity( disparity, cost, minDisparity, numberOfDisparities[, disp12MaxDisp] ) -> disparity
```

`#include <opencv2/calib3d.hpp>`

validates disparity using the left-right check. The matrix "cost" should be computed by the stereo correspondence algorithm