moments

Calculates all of the moments up to the third order of a polygon or rasterized shape.

**C++:**Moments **moments**(InputArray **array**, bool **binaryImage**=false )

**Python:**cv2.**moments**(array[, binaryImage]) → retval

**C:**void **cvMoments**(const CvArr\* **arr**, CvMoments\* **moments**, int **binary**=0 )

**Python:**cv.**Moments**(arr, binary=0) → moments

|  |  |
| --- | --- |
| **Parameters:** | * **array** – Raster image (single-channel, 8-bit or floating-point 2D array) or an array (  or  ) of 2D points (Point or Point2f ). * **binaryImage** – If it is true, all non-zero image pixels are treated as 1’s. The parameter is used for images only. * **moments** – Output moments. |

The function computes moments, up to the 3rd order, of a vector shape or a rasterized shape. The results are returned in the structure Moments defined as:

**class** **Moments**

{

**public**:

Moments();

Moments(double m00, double m10, double m01, double m20, double m11,

double m02, double m30, double m21, double m12, double m03 );

Moments( **const** CvMoments& moments );

**operator** CvMoments() **const**;

*// spatial moments*

double m00, m10, m01, m20, m11, m02, m30, m21, m12, m03;

*// central moments*

double mu20, mu11, mu02, mu30, mu21, mu12, mu03;

*// central normalized moments*

double nu20, nu11, nu02, nu30, nu21, nu12, nu03;

}

In case of a raster image, the spatial moments  are computed as:

The central moments  are computed as:

where  is the mass center:

The normalized central moments  are computed as:

**Note**

,   , hence the values are not stored.

The moments of a contour are defined in the same way but computed using the Green’s formula (see <http://en.wikipedia.org/wiki/Green_theorem>). So, due to a limited raster resolution, the moments computed for a contour are slightly different from the moments computed for the same rasterized contour.

**Note**

Since the contour moments are computed using Green formula, you may get seemingly odd results for contours with self-intersections, e.g. a zero area (m00) for butterfly-shaped contours.

**See also**

**[contourArea()](https://docs.opencv.org/2.4/modules/imgproc/doc/structural_analysis_and_shape_descriptors.html" \l "double contourArea(InputArray contour, bool oriented))**, **[arcLength()](https://docs.opencv.org/2.4/modules/imgproc/doc/structural_analysis_and_shape_descriptors.html" \l "double arcLength(InputArray curve, bool closed))**

approxPolyDP

Approximates a polygonal curve(s) with the specified precision.

**C++:**void **approxPolyDP**(InputArray **curve**, OutputArray **approxCurve**, double **epsilon**, bool **closed**)

**Python:**cv2.**approxPolyDP**(curve, epsilon, closed[, approxCurve]) → approxCurve

**C:**CvSeq\* **cvApproxPoly**(const void\* **src\_seq**, int **header\_size**, CvMemStorage\* **storage**, int **method**, double **eps**, int **recursive**=0 )

|  |  |
| --- | --- |
| **Parameters:** | * **curve** –   Input vector of a 2D point stored in:   * + std::vector or Mat (C++ interface)   + Nx2 numpy array (Python interface)   + CvSeq or `` CvMat (C interface) * **approxCurve** – Result of the approximation. The type should match the type of the input curve. In case of C interface the approximated curve is stored in the memory storage and pointer to it is returned. * **epsilon** – Parameter specifying the approximation accuracy. This is the maximum distance between the original curve and its approximation. * **closed** – If true, the approximated curve is closed (its first and last vertices are connected). Otherwise, it is not closed. * **header\_size** – Header size of the approximated curve. Normally, sizeof(CvContour) is used. * **storage** – Memory storage where the approximated curve is stored. * **method** – Contour approximation algorithm. Only CV\_POLY\_APPROX\_DP is supported. * **recursive** – Recursion flag. If it is non-zero and curve is CvSeq\*, the function cvApproxPoly approximates all the contours accessible from curve by h\_next and v\_next links. |

The functions approxPolyDP approximate a curve or a polygon with another curve/polygon with less vertices so that the distance between them is less or equal to the specified precision. It uses the Douglas-Peucker algorithm <http://en.wikipedia.org/wiki/Ramer-Douglas-Peucker_algorithm>

See <https://github.com/opencv/opencv/tree/master/samples/cpp/contours2.cpp> for the function usage model.

arcLength

Calculates a contour perimeter or a curve length.

**C++:**double **arcLength**(InputArray **curve**, bool **closed**)

**Python:**cv2.**arcLength**(curve, closed) → retval

**C:**double **cvArcLength**(const void\* **curve**, CvSlice **slice**=CV\_WHOLE\_SEQ, int **is\_closed**=-1 )

**Python:**cv.**ArcLength**(curve, slice=CV\_WHOLE\_SEQ, isClosed=-1) → float

|  |  |
| --- | --- |
| **Parameters:** | * **curve** – Input vector of 2D points, stored in std::vector or Mat. * **closed** – Flag indicating whether the curve is closed or not. |

The function computes a curve length or a closed contour perimeter.