

OpenCV 4.x Cheat Sheet (Python version)

A summary of: <https://docs.opencv.org/master/>

I/O

<code>i = imread("name.png")</code>	Loads image as BGR (if grayscale, B=G=R)
<code>i = imread("name.png", IMREAD_UNCHANGED)</code>	Loads image as is (inc. transparency if available)
<code>i = imread("name.png", IMREAD_GRAYSCALE)</code>	Loads image as grayscale
<code>imshow("Title", i)</code>	Displays image I
<code>imwrite("name.png", i)</code>	Saves image I
<code>waitKey(500)</code>	Wait 0.5 seconds for keypress (0 waits forever)
<code>destroyAllWindows()</code>	Releases and closes all windows

Color/Intensity

<code>i_gray = cvtColor(i, COLOR_BGR2GRAY)</code>	BGR to gray conversion
<code>i_rgb = cvtColor(i, COLOR_BGR2RGB)</code>	BGR to RGB (useful for <code>matplotlib</code>)
<code>i = cvtColor(i, COLOR_GRAY2RGB)</code>	Converts grayscale to RGB (R=G=B)
<code>i = equalizeHist(i)</code>	Histogram equalization
<code>i = normalize(i, None, 0, 255, NORM_MINMAX, CV_8U)</code>	Normalizes I between 0 and 255
<code>i = normalize(i, None, 0, 1, NORM_MINMAX, CV_32F)</code>	Normalizes I between 0 and 1

Other useful color spaces

<code>COLOR_BGR2HSV</code>	BGR to HSV (Hue, Saturation, Value)
<code>COLOR_BGR2LAB</code>	BGR to Lab (Lightness, Green/Magenta, Blue/Yellow)
<code>COLOR_BGR2LUV</code>	BGR to Luv (\approx Lab, but different normalization)
<code>COLOR_BGR2YCrCb</code>	BGR to YCrCb (Luma, Blue-Luma, Red-Luma)

Channel manipulation

<code>b, g, r = split(i)</code>	Splits the image I into channels
<code>b, g, r, a = split(i)</code>	Same as above, but I has alpha channel
<code>i = merge((b, g, r))</code>	Merges channels into image

Arithmetic operations

<code>i = add(i1, i2)</code>	$\min(I_1 + I_2, 255)$, i.e. saturated addition if uint8
<code>i = addWeighted(i1, alpha, i2, beta, gamma)</code>	$\min(\alpha I_1 + \beta I_2 + \gamma, 255)$, i.e. image blending
<code>i = subtract(i1, i2)</code>	$\max(I_1 - I_2, 0)$, i.e. saturated subtraction if uint8
<code>i = absdiff(i1, i2)</code>	$ I_1 - I_2 $, i.e. absolute difference

Note: one of the images can be replaced by a scalar.

Logical operations

<code>i = bitwise_not(i)</code>	Inverts every bit in I (e.g. mask inversion)
<code>i = bitwise_and(i1, i2)</code>	Logical <i>and</i> between I_1 and I_2 (e.g. mask image)
<code>i = bitwise_or(i1, i2)</code>	Logical <i>or</i> between I_1 and I_2 (e.g. merge 2 masks)
<code>i = bitwise_xor(i1, i2)</code>	Exclusive <i>or</i> between I_1 and I_2

Statistics

<code>mB, mG, mR, mA = mean(i)</code>	Average of each channel (i.e. BGRA)
<code>ms, sds = meanStdDev(i)</code>	Mean and SDev p/channel (3 or 4 rows each)
<code>h = calcHist([i], [c], None, [256], [0,256])</code>	Histogram of channel c , no mask, 256 bins (0-255)
<code>h = calcHist([i], [0,1], None, [256,256], [0,256, 0,256])</code>	2D histogram using channels 0 and 1, with “resolution” 256 in each dimension

Filtering

<code>i = blur(i, (5, 5))</code>	Filters I with 5×5 box filter (i.e. average filter)
<code>i = GaussianBlur(i, (5,5), sigmaX=0, sigmaY=0)</code>	Filters I with 5×5 Gaussian; auto σ s; (I is float)
<code>i = GaussianBlur(i, None, sigmaX=2, sigmaY=2)</code>	Blurs, auto kernel dimension
<code>i = filter2D(i, -1, k)</code>	Filters with 2D kernel using cross-correlation
<code>kx = getGaussianKernel(5, -1)</code>	1D Gaussian kernel with length 5 (auto StDev)
<code>i = sepFilter2D(i, -1, kx, ky)</code>	Filter using separable kernel (same output type)
<code>i = medianBlur(i, 3)</code>	Median filter with size=3 (size ≥ 3)
<code>i = bilateralFilter(i, -1, 10, 50)</code>	Bilateral filter with $\sigma_r = 10$, $\sigma_s = 50$, auto size

Borders

All filtering operations have parameter `borderType` which can be set to:

<code>BORDER_CONSTANT</code>	Pads with constant border (requires additional parameter <code>value</code>)
<code>BORDER_REPLICATE</code>	Replicates the first/last row and column onto the padding
<code>BORDER_REFLECT</code>	Reflects the image borders onto the padding
<code>BORDER_REFLECT_101</code>	Same as previous, but doesn't include the pixel at the border (the default)
<code>BORDER_WRAP</code>	Wraps around the image borders to build the padding

Borders can also be added with custom widths:

<code>i = copyMakeBorder(i, 2, 2, 3, 1, borderType=BORDER_WRAP)</code>	Widths: top, bottom, left, right
--	----------------------------------

Differential operators

<code>i_x = Sobel(i, CV_32F, 1, 0)</code>	Sobel in the x-direction: $I_x = \frac{\partial}{\partial x} I$
<code>i_y = Sobel(i, CV_32F, 0, 1)</code>	Sobel in the y-direction: $I_y = \frac{\partial}{\partial y} I$
<code>i_x, i_y = spatialGradient(i, 3)</code>	The gradient: ∇I (using 3×3 Sobel): needs uint8 image
<code>m = magnitude(i_x, i_y)</code>	$\ \nabla I\ $; I_x, I_y must be float (for conversion, see <code>np.astype()</code>)
<code>m, d = cartToPolar(i_x, i_y)</code>	$\ \nabla I\ $; $\theta \in [0, 2\pi]$; <code>angleInDegrees=False</code> ; needs float32 I_x, I_y
<code>l = Laplacian(i, CV_32F, ksize=5)</code>	ΔI , Laplacian with kernel size of 5

Geometric transforms

<code>i = resize(i, (width, height))</code>	Resizes image to width×height
<code>i = resize(i, None, fx=0.2, fy=0.1)</code>	Scales image to 20% width and 10% height
<code>M = getRotationMatrix2D((xc, yc), deg, scale)</code>	Returns 2×3 rotation matrix M , arbitrary (x_c, y_c)
<code>M = getAffineTransform(pts1,pts2)</code>	Affine transform matrix M from 3 correspondences
<code>i = warpAffine(i, M, (cols,rows))</code>	Applies Affine transform M to I , output size=(cols, rows)
<code>M = getPerspectiveTransform(pts1,pts2)</code>	Perspective transform matrix M from 4 correspondences
<code>M, s = findHomography(pts1, pts2)</code>	Persp transf mx M from all $\gg 4$ corresps (Least squares)
<code>M, s = findHomography(pts1, pts2, RANSAC)</code>	Persp transf mx M from best $\gg 4$ corresps (RANSAC)
<code>i = warpPerspective(i, M, (cols, rows))</code>	Applies perspective transform M to image I

Interpolation methods

`resize`, `warpAffine` and `warpPerspective` use bilinear interpolation by default. It can be changed by parameter `interpolation` for `resize`, and `flags` for the others:

<code>flags=INTER_NEAREST</code>	Simplest, fastest (or <code>interpolation=INTER_NEAREST</code>)
<code>flags=INTER_LINEAR</code>	Bilinear interpolation: Default
<code>flags=INTER_CUBIC</code>	Bicubic interpolation

Segmentation

<code>_, i_t = threshold(i, t, 255, THRESH_BINARY)</code>	Manually thresholds image I given threshold level t
<code>t, i_t = threshold(i, 0, 255, THRESH_OTSU)</code>	Returns thresh level and thresholded image using Otsu
<code>i_t = adaptiveThreshold(i, 255, ADAPTIVE_THRESH_MEAN_C, THRESH_BINARY, b, c)</code>	Adaptive mean-c with block size b and constant c
<code>bp = calcBackProject([i_hsv], [0,1], h, [0,180, 0,256], 1)</code>	Back-projects histogram h onto the image <code>i_hsv</code> using only hue and saturation; no scaling (i.e. 1)
<code>cp, la, ct = kmeans(feats, K, None, crit, 10, KMEANS_RANDOM_CENTERS)</code>	Returns the labels <code>la</code> and centers <code>ct</code> of K clusters, best compactness <code>cp</code> out of 10; 1 feat/column

Features

<code>e = Canny(i, t1, th)</code>	Returns the Canny edges (e is binary)
<code>l = HoughLines(e, 1, pi/180, 150)</code>	Returns all $(\rho, \theta) \geq 150$ votes, Bin res: $\rho = 1$ pix, $\theta = 1$ deg
<code>l = HoughLinesP(e, 1, pi/180, 150, None, 100, 20)</code>	Probabilistic Hough, min length=100, max gap=20
<code>c = HoughCircles(i, HOUGH_GRADIENT, 1, minDist=50, param1=200, param2=18, minRadius=20, maxRadius=60)</code>	Returns all (x_c, y_c, r) with at least 18 votes, bin resolution=1, param1 is the t_h of Canny, and the centers must be at least 50 pixels away from each other
<code>r = cornerHarris(i, 3, 5, 0.04)</code>	Harris corners' R s per pixel, window=3, Sobel=5, $\alpha = 0.04$
<code>f = FastFeatureDetector_create()</code>	Instantiates the Star feature detector
<code>k = f.detect(i, None)</code>	Detects keypoints on grayscale image I
<code>i_k = drawKeypoints(i, k, None)</code>	Draws keypoints k on color image I
<code>d = xfeatures2d.BriefDescriptorExtractor_create()</code>	Instantiates a BRIEF descriptor
<code>k, ds = d.compute(i, k)</code>	Computes the descriptors of keypoints k over I
<code>dd = AKAZE_create()</code>	Instantiates the AKAZE detector/descriptor
<code>m = BFMatcher.create(NORM_HAMMING, crossCheck=True)</code>	Instantiates a brute-force matcher, with x-checking, and Hamming distance
<code>ms = m.match(ds_l, ds_r)</code>	Matches the left and right descriptors
<code>i_m = drawMatches(i_l, k_l, i_r, k_r, ms, None)</code>	Draws matches from the left keypoints k_l on left image I_l to right I_r , using matches ms

Detection

<code>ccs = matchTemplate(i, t, TM_CCORR_NORMED)</code>	Matches template T to image I (normalized X-correl)
<code>m, M, m_l, M_l = minMaxLoc(ccs)</code>	Min, max values and respective coordinates in ccs
<code>c = CascadeClassifier()</code>	Creates an instance of an “empty” cascade classifier
<code>r = c.load("file.xml")</code>	Loads a pre-trained model from file; r is True/False
<code>objs = c.detectMultiScale(i)</code>	Returns 1 tuple (x, y, w, h) per detected object

Motion and Tracking

<code>pts = goodFeaturesToTrack(i, 100, 0.5, 10)</code>	Returns 100 Shi-Tomasi corners with, at least, 0.5 quality, and 10 pixels away from each other
<code>pts1, st, e = calcOpticalFlowPyrLK(i0, i1, pts0, None)</code>	New positions of pts from estimated optical flow between I_0 and I_1 ; st [<i>i</i>] is 1 if flow for point i was found, or 0 otherwise
<code>t = TrackerCSRT_create()</code>	Instantiates the CSRT tracker
<code>r = t.init(f, bbox)</code>	Initializes tracker with frame and bounding box
<code>r, bbox = t.update(f)</code>	Returns new bounding box, given next frame

Drawing on the image

<code>line(i,(x0, y0),(x1, y1), (b, g, r), t)</code>	Line
<code>rectangle(i, (x0, y0), (x1, y1), (b, g, r), t)</code>	Rectangle
<code>circle(i,(x0, y0), radius, (b, g, r), t)</code>	Circle
<code>polylines(i,[pts], True, (b, g, r), t)</code>	Closed (True) polygon (pts is array of points)
<code>putText(i, "Hi", (x,y), FONT_HERSHEY_SIMPLEX, 1, (r,g,b), 2, LINE_AA)</code>	Writes “Hi” at (x, y) , font size=1, thickness=2

Parameters

(x0, y0)	Origin/Start/Top left corner (note that it’s not (row,column))
(x1, y1)	End/Bottom right corner
(b, g, r)	Line color (uint8)
t	Line thickness (fills, if negative)

Calibration and Stereo

<code>r, crns = findChessboardCorners(i, (n_x,n_y))</code>	2D coords of detected corners; i is gray; r is the status; (n_x , n_y) is size of calib target
<code>crnrs = cornerSubPix(i, crns, (5,5), (-1,-1), crit)</code>	Improves coordinates with sub-pixel accuracy
<code>r, K, D, ExRs, ExTs = calibrateCamera(crns_3D, crns_2D, i.shape[:2], None, None)</code>	Calculates intrinsics (inc. distortion coeffs), & extrinsics (i.e. 1 R+T per target view); crns_3D contains 1 array of 3D corner coords p/target view; crns_2D contains the respective arrays of 2D corner coordinates (i.e. 1 crns p/target view)
<code>drawChessboardCorners(i, (n_x, n_y), crns, r)</code>	Draws corners on I (may be color); r is status from corner detection
<code>u = undistort(i, K, D)</code>	Undistorts I using the intrinsics
<code>s = StereoSGBM_create(minDisparity = 0, numDisparities = 32, blockSize = 11)</code>	Instantiates Semi-Global Block Matching method
<code>s = StereoBM_create(32, 11)</code>	Instantiates a simpler block matching method
<code>d = s.compute(i_L, i_R)</code>	Computes disparity map (α^{-1} depth map)

Termination criteria (used in e.g. K-Means, Camera calibration)

<code>crit = (TERM_CRITERIA_MAX_ITER, 20, 0)</code>	Stops after 20 iterations
<code>crit = (TERM_CRITERIA_EPS, 0, 1.0)</code>	Stop if “movement” is less than 1.0
<code>crit = (TERM_CRITERIA_MAX_ITER TERM_CRITERIA_EPS, 20, 1.0)</code>	Stops whatever happens first

Useful stuff

Numpy (np.)

<code>m = mean(i)</code>	Mean/average of array I
<code>m = average(i, weights)</code>	Weighted mean/average of array I
<code>v = var(i)</code>	Variance of array/image I
<code>s = std(i)</code>	Standard deviation of array/image I
<code>h,b = histogram(i.ravel(),256,[0,256])</code>	numpy histogram also returns the bins b
<code>i = clip(i, 0, 255)</code>	numpy ’s saturation/clamping function
<code>i = i.astype(np.float32)</code>	Converts the image type to float32 (vs. uint8, float64)
<code>x, _, _ = linalg.lstsq(A, b)</code>	Solves the least squares problem $\frac{1}{2} Ax - b ^2$
<code>i = hstack((i1, i2))</code>	Merges I_1 and I_2 side-by-side
<code>i = vstack((i1, i2))</code>	Merges I_1 above I_2
<code>i = fliplr(i)</code>	Flips image left-right
<code>i = flipud(i)</code>	Flips image up-down
<code>i = pad(i, ((1, 1), (3, 3)), 'reflect')</code>	Alternative to copyMakeBorder (also top, bottom, left, right)
<code>idx = argmax(i)</code>	Linear index of maximum in I (i.e. index of flattened I)
<code>r, c = unravel_index(idx, i.shape)</code>	2D coordinate of the index with respect to shape of i
<code>b = any(M > 5)</code>	Returns True if any element in array M is greater than 5
<code>b = all(M > 5)</code>	Returns True if all elements in array M are greater than 5
<code>rows, cols = where(M > 5)</code>	Returns indices of the rows and cols where elems in M are >5
<code>coords = list(zip(rows, cols))</code>	Creates a list with the elements of rows and cols paired
<code>M_inv = linalg.inv(M)</code>	Inverse of M
<code>rad = deg2rad(deg)</code>	Converts degrees into radians

Matplotlib.pyplot (plt.)

<code>imshow(i, cmap="gray", vmin=0, vmax=255)</code>	matplotlib ’s imshow preventing auto-normalization
<code>quiver(xx, yy, i_x, -i_y, color="green")</code>	Plots the gradient direction at positions xx , yy
<code>savefig("name.png")</code>	Saves the plot as an image

Copyright © 2019 António Anjos (Rev: 2020-07-01)
Most up-to-date version: <https://github.com/a-anjos/python-opencv>