OpenCV 4.x Cheat Sheet (Python version)

A summary of: `https://docs.opencv.org/master/`

I/O

`= imread("name.png")`	Loads image as BGR (if grayscale, B=G=R)
`= imread("name.png", IMREAD_UNCHANGED)`	Loads image as is (inc. transparency if available)
`= imread("name.png", IMREAD_GRAYSCALE)`	Loads image as grayscale
`mshow("Title", i)`	Displays image I
`mwrite("name.png", i)`	Saves image I
`aitKey(500)`	Wait 0.5 seconds for keypress (0 waits forever)
`lestroyAllWindows()`	Releases and closes all windows

Color/Intensity

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

Other useful color spaces

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

Channel manipulation

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

Arithmetic operations

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

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

Logical operations

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

Statistics

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

Filtering

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

Borders

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

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

Borders can also be added with custom widths:

`i = copyMakeBorder(i, 2, 2, 3, 1, borderType=BORDER_WRAP)`	Widths: top, bottom, left, right

Differential operators

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

Geometric transforms

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

`flags=INTER_NEAREST`	Simplest, fastest (or `interpolation=INTER_NEAREST`)
`flags=INTER_LINEAR`	Bilinear interpolation: Default
`flags=INTER_CUBIC`	Bicubic interpolation

Segmentation

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

Features

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

Detection

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

Motion and Tracking

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

Drawing on the image

- `line(i,(x0, y0),(x1, y1), (b, g, r), t)` — Line
- `rectangle(i, (x0, y0), (x1, y1), (b, g, r), t)` — Rectangle
- `circle(i,(x0, y0), radius, (b, g, r), t)` — Circle
- `polylines(i,[pts], True, (b, g, r), t)` — Closed (True) polygon (pts is array of points)
- `putText(i, "Hi", (x,y), FONT_HERSHEY_SIMPLEX, 1, (r,g,b), 2, LINE_AA)` — 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

- `r, crns = findChessboardCorners(i, (n_x,n_y))` — 2D coords of detected corners; i is gray; r is the status; (n_x, n_y) is size of calib target
- `crnrs = cornerSubPix(i, crns, (5,5), (-1,-1), crit)` — Improves coordinates with sub-pixel accuracy
- `r, K, D, ExRs, ExTs = calibrateCamera(crns_3D, crns_2D, i.shape[:2], None, None)` — 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 2D corner coordinates (i.e. 1 crns p/target v
- `drawChessboardCorners(i, (n_x, n_y), crns, r)` — Draws corners on I (may be color); r is status from corner detection
- `u = undistort(i, K, D)` — Undistorts I using the intrinsics
- `s = StereoSGBM_create(minDisparity = 0, numDisparities = 32, blockSize = 11)` — Instantiates Semi-Global Block Matching met
- `s = StereoBM_create(32, 11)` — Instantiates a simpler block matching method
- `d = s.compute(i_L, i_R)` — Computes disparity map ($\propto^{-1}$ depth map)

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

- `crit = (TERM_CRITERIA_MAX_ITER, 20, 0)` — Stops after 20 iterations
- `crit = (TERM_CRITERIA_EPS, 0, 1.0)` — Stop if "movement" is less than 1.0
- `crit = (TERM_CRITERIA_MAX_ITER | TERM_CRITERIA_EPS, 20, 1.0)` — Stops whatever happens first

Useful stuff

Numpy (np.)

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

Matplotlib.pyplot (plt.)

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

Most up-to-date version: https://github.com/a-anjos/python-opencv