# OpenCV Cheat Sheet

In [ ]:
```
import cv2 as cv
```

## GUI

In [ ]:
```
cv.imshow('image', img)
cv.waitKey(0)  # 0=indefinitely, otherwise delay in ms
cv.destroyAllWindows()
```

In [ ]:
```
cv.namedWindow('window')
cv.setMouseCallback('Image mouse', mouse_callback, param=None)
def mouse_callback(event, x, y, flags, param):
    if event == cv.EVENT_LBUTTONDOWN | cv.EVENT_LBUTTONUP |cv.EVENT_LBUTTONDBLC
                cv.EVENT_MOUSEMOVE | cv.EVENT_MOUSEWHEEL:
    pass
```

## Colors

In [ ]:
```
b = img_OpenCV[:, :, 0]
g = img_OpenCV[:, :, 1]
r = img_OpenCV[:, :, 2]
# --- or ---
b, g, r = cv.split(img)

img = cv.merge((r, g, b))

img_rgb = img_bgr[:, :, ::-1]
# --- or ---
img_rgb = cv.cvtColor(img_bgr, cv.COLOR_BGR2RGB)
```

In [ ]:
```
img_gry = cv.cvtColor(img_bgr, cv.COLOR_BGR2GRAY)
img_col = cv.applyColorMap(img_gry, cv.COLORMAP_JET)
```

## Image Manipulation

In [ ]:
```
pix_b = img[0, 0, 0]
pix_bgr = img[0, 0]
img_b = img[:, :, 0]
img_slice = img[0:10, 0:20]

img.fill(128)
img[:] = 128
img[:, :, 0] = 0
```

In [ ]:
```
# stack horizontally
img_lr = np.concatenate((img_l, img_r), axis=1)
```

## File I/O

In [ ]:
```
img = cv.imread('img.png')
img = cv.imread('img.png', cv.IMREAD_GRAYSCALE)
```

In [ ]:
```
cv.imwrite('img.jpg', img)
```

## Video

In [ ]:
```
capture = cv.VideoCapture(0)  # 0=index_camera, also video filename
assert capture.isOpened()

width = capture.get(cv.CAP_PROP_FRAME_WIDTH)
height = capture.get(cv.CAP_PROP_FRAME_HEIGHT)
fps = capture.get(cv.CAP_PROP_FPS)

while capture.isOpened():
    ret, frame = capture.read()
    if not ret: break
capture.release()
```

In [ ]:
```
fourcc = cv.VideoWriter_fourcc(*'AVC1')
# https://gist.github.com/takuma7/44f9ecb028ff00e2132e
writer = cv.VideoWriter(video_path, fourcc, fps, width, height, is_color)
writer.write(frame)
writer.release()
```

In [ ]:
```
# navigating video files
num_frames = capture.get(cv.CAP_PROP_FRAME_COUNT)
capture.set(cv.CAP_PROP_POS_FRAMES, <FRAME_INDEX>)
```

## Drawing Shapes

In [ ]:
```
pt1, pt2 = (0, 0), (100, 100)
pts = np.array([[250, 5], [220, 80], [280, 80]], np.int32).reshape((-1, 1, 2))
color = (255, 255, 255)
lineType = cv.LINE_4 | cv.LINE_8 | cv.LINE_AA
thicknes = -1  # fill shape
```

In [ ]:
```
cv.line(img, pt1, pt2, color, thickness=1, lineType=8, shift=0)
cv.arrowedLine(img, pt1, pt2, color, thickness=1, lineType=8, shift=0, tipLengt
cv.rectangle(img, pt1, pt2, color, thickness=1, lineType=8, shift=0)
cv.circle(img, center, radius, color, thickness=1, lineType=8, shift=0)
cv.ellipse(img, center, axes, angle, startAngle, endAngle, color,
           thickness=1, lineType=8, shift=0)
cv.polylines(img, pts, is_closed, color, thickness=1, lineType=8, shift=0)
```

In [ ]:
```
rect = (0, 0, 50, 50)
is_intersecting, pt1, pt2 = clipLine(rect, pt1, pt2)
```

## Drawing Text

In [ ]:
```
font_face = cv.FONT_HERSHEY_SIMPLEX or cv.FONT_HERSHEY_DUPLEX or ...
cv.putText(img, text, org, fontFace, fontScale, color,
           thickness=1, lineType=8, bottomLeftOrigin=False)  # !bottomLeftOrigi
```

In [ ]:
```
font_scale = cv.getFontScaleFromHeight(fontFace, pixelHeight, thickness=1)
(width, height), baseLine = cv.getTextSize(text, fontFace, fontScale, thickness
```

## Geometric Transformations

In [ ]:
```
# Resizing
interpolation = cv.INTER_NEAREST | cv.INTER_LINEAR | cv.INTER_CUBIC |
                cv.INTER_AREA | cv.INTER_LANCZOS4
image = cv.resize(img, (width, height), interpolation=cv.INTER_LINEAR)
image = cv.resize(img, None, fx=0.5, fy=0.5)  # dSize=None => auto-calc
# ---
image = cv.pyrDown(src[, dst[, dSize[, borderType]]])  # Blur and downsample (2
image = cv.pyrUp(src[, dst[, dSize[, borderType]]])    # Upsample (2X) and blur
```

In [ ]:
```
# Translation
M = np.float32([[1, 0, translate_x],
                [0, 1, translate_y]])
image = cv.warpAffine(img, M, (outputWidth, outputHeight))
```

In [ ]:
```
# Rotation
M = cv.getRotationMatrix2D((centerX, centerY), angleDeg, scaleFactor)
image = cv.warpAffine(img, M, (outputWidth, outputHeight))
# ---
image = cv.transpose(img)  ## Rotate 90-deg CCW
```

In [ ]:
```
# Affine Transformation
pts_1 = np.float32([[0,0], [0,1], [1,0]])
pts_2 = np.float32([[1,1], [1,3], [4,1]])
M = cv.getAffineTransform(pts_1, pts_2)
image = cv.warpAffine(img, M, (outputWidth, outputHeight))
```

In [ ]:
```
# Perspective Transformation
pts_1 = np.float32([[0,0], [0,1], [1,0], [1,1]])
pts_1 = np.float32([[0,0], [0,2], [2,0], [3,3]])
M = cv.getPerspectiveTransform(pts_1, pts_2)
image = cv.warpPerspective(img, M, (300, 300))
```

In [ ]:
```
# Horz/Vert Flipping
# flipCode: 0 => Horz, 1 => Vert, -1 => Both
image = cv.flip(img, flipCode)
```

## Image Filtering

In [ ]:
```
kernel = np.ones((5, 5), np.float32) / 25
# ddepth=-1 => output will have same depth as source
image = cv.filter2D(img, ddepth, kernel)
```

__Sharpening Kernels__

```
|||||----   |||||----|-   |||||----
|----|----|  ---|----||   |----|----|
|0|-1|0|   -1|-1|-1|   |1|1|1|
|-1|4|-1    |-1|8|-1|   |1|-8|1
||0|-1|0    |-1|-1|-1   ||1|1|1
|           |            |
```

__Sobel Kernels__

```
|||||----   |||||----   |||||----
|----|----|  |----|----|  |----|----|
|-1|0|1|   |-1|-2|1    |1|1|1|
|-2|0|2    ||0|0|0      |1|-8|1
||-1|0|1    ||-1|-2|     ||1|1|1
|           1|           |
```

__Laplacian Kernels__

```
|||||----   |||||----
|----|----|  |----|----|
|0|1|0|   |1|4|1|
||1|-4|1    |4|-20|
||0|1|0    4||1|4|
|           1|
```

In [ ]:
```
# Sobel
# dx, dy: order of derivative
# cv.Sobel(src, ddepth, dx, dy[, dst[, ksize=3[, ...]]])
image_x = cv.Sobel(img, cv.CV_32F, 0, 1, ksize=5)
image_y = cv.Sobel(img, cv.CV_32F, 1, 0, ksize=5)
# Laplacian
# cv.Laplacian(src, ddepth[, dst[, ksize[, ...]]])
image = cv.Laplacian(img, cv.CV_32F)
```

In [ ]:
```
# Unsharp Mask
smoothed = cv.GaussianBlur(img, ksize, sigmaX)
# cv.addWeighted(src1, alpha, src2, beta, gamma)
# dst = src1*alpha + src2*beta + gamma
unsharped = cv.addWeighted(img, 1.5, smoothed, -0.5, 0)
```

In [ ]:
```
ksize = (width, height)
# Box Blur
image = cv.blur(img, ksize)
# Gausssian Blur
# sigmaX=0 => computed from ksize.width and ksize.height
image = cv.GaussianBlur(img, ksize, sigmaX)
```

In [ ]:
```
# Median Blur
ksize1 = 5  # width == height
image = cv.medianBlur(img, ksize1)
```

In [ ]:
```
# Bilateral Blur
# dia<0 => computed from sigmaSpatial
image = cv.bilateralFilter(img, dia, sigmaColor, SigmaSpatial)
```

In [ ]:
```
# Canny Edge
image = cv.Canny(img, loThreshold1, hiThreshold, sobelApertSize=3)
```

## NLM Denoising

- cv.fastNlMeansDenoising() - single grayscale image
- cv.fastNlMeansDenoisingColored() - color image
- cv.fastNlMeansDenoisingMulti() - sequence of grayscale images
- cv.fastNlMeansDenoisingColoredMulti() - sequence of color images

In [ ]:
```
fastNlMeansDenoising(img[, dst[, h=3.0[, hColor=3.0[, templateWindowSize=7[, se
```

## Arithmetic Ops

In [ ]:
```
# Saturation Arithmetic
# src1, src2: array or scalar
image = cv.add(src1, src2)
image = cv.subtract(src1, src2)
```

In [ ]:
```
# Blending
```

```
image = cv.addWeighted(src1, alpha, src2, beta, gamma)
```

In [ ]:
```
# Bitwise
image = cv.bitwise_not(img)
image = cv.bitwise_and(src1, src2)
image = cv.bitwise_or(src1, src2)
image = cv.bitwise_xor(src1, src2)
```

In [ ]:
```
# lowerb: inclusive lower-bound array/scalar
# upperb: inclusive upper-bound array/scalar
mask = cv.inRange(img, lowerb, upperb)
```

## Morphological Ops

In [ ]:
```
shape =  cv.MORPH_RECT | cv.MORPH_ELLIPSE | cv.MORPH_CROSS
cv.getStructuringElement(shape, ksize)
```

In [ ]:
```
image = cv.dilate(img, kernel, iterations=1)
image = cv.erode(img, kernel, iterations=1)
```

In [ ]:
```
image = cv.morphologyEx(img, cv.MORPH_OPEN, kernel)       # erosion → dilation
image = cv.morphologyEx(img, cv.MORPH_CLOSE, kernel)      # dilation → erosion
image = cv.morphologyEx(img, cv.MORPH_GRADIENT, kernel)   # dilation - erosion
image = cv.morphologyEx(img, cv.MORPH_TOPHAT, kernel)     # original - opening
image = cv.morphologyEx(img, cv.MORPH_BLACKHAT, kernel)   # closing - original
```

## Histogram

NOTE: cv.calcHist() is much faster than np.histogram() and plt.hist()

In [ ]:
```
# images: list of images
# channels: list of channel idxs, e.g. grayscale: [0], color: [0, 1, 2]
# mask : None => no mask
# histSize: list of # hist bins
# ranges: range of intensity to measure (upper non-inclusive), e.g. [0, 256]
hist = cv.calcHist([image], [channels], mask, [histSize], [ranges])
```

In [ ]:
```
# Masks
mask = np.zeros((100, 100), np.uint8)
mask[10:90, 10:90] = 255
```

## Histogram Equalization

In [ ]:
```
# Grayscale
image = cv.equalizeHist(img_gry)

# Color
H, S, V = cv.split(cv.cvtColor(img, cv.COLOR_BGR2HSV))
V_eq = cv.equalizeHist(V)
image = cv.cvtColor(cv.merge([H, S, eq_V]), cv.COLOR_HSV2BGR)
```

In [ ]:
```
# CLAHE
# cv.createCLAHE(clipLimit, tileGridSize=(8,8))
```

```
clahe = cv.createCLAHE(clipLimit=2.0)
image = clahe.apply(img_gry)
```

## Thresholding

In [ ]:
```
threshType = (cv.THRESH_BINARY | cv.THRESH_BINARY_INV | cv.THRESH_TRUNC) + cv.T
retval, image = cv.threshold(img, thresh, maxval, threshType)
```

In [ ]:
```
adaptMethod = cv.ADAPTIVE_THRESH_MEAN_C | cv.ADAPTIVE_THRESH_GAUSSIAN_C
# ADAPTIVE_THRESH_GAUSSIAN_C => cross-correlation with Gaussian window (sigma c
# blocksize: int
# threshOffs: constant subtracted from the (weighted) mean
image = adaptiveThreshold(img, maxValue, adaptMethod, threshType, blockSize, th
```

## Contours

In [ ]:
```
img_edge = cv.Canny(img, 30, 200)
# mode: cv.RETR_EXTERNAL => Outer only, cv.RETR_LIST => All, cv.RETR_TREE => Al
method = cv.CHAIN_APPROX_NONE | cv.CHAIN_APPROX_SIMPLE | cv.CHAIN_APPROX_TC89_I
contours, hierarchy = cv.findContours(img_edge, mode, method)
# contourIdx: -1 => all
cv.drawContours(img, contours, contourIdx, color, thickness, lineType)
```

In [ ]:
```
# Contour Length
closed = True  # whether contour is closed
epsilon = 0.03 * cv.arcLength(contour, closed)

# Vertices Reduction (Polygon Approximation)
# epsilon: max dist between original contour and its approximation
approx = cv.approxPolyDP(contour, epsilon, closed)

# Convex Hull
hull = cv.convexHull(contour)
```

In [ ]:
```
# Contour Area
area = cv.contourArea(contour)

# Contour Moments
mo = cv.moments(contours)
cx = int(mo['m10'] / mo['m00'])
cy = int(mo['m01'] / mo['m00'])
```

## Line / Circle Detection

In [ ]:
```
# rho: Dist resolution of accumulator (px)
# theta: Angle resolution of accumulator (rads)
# threshold: Accummmulator (votes) threshold
# [min_theta], [max_theta]: min/max angle to check for lines
lines = cv.HoughLines(img_edge, rho, theta, threshold)

# minLineLength: shorter line segments will be rejected
# maxLineGap: max allowed gap between points to be on the same line
lines = cv.HoughLinesP(img_edge, rho, theta, threshold[, lines[, minLineLength[
```

```
# dp: image_resolution / accum_resolution
# minDist: min dist between circle centers
method = cv.HOUGH_GRADIENT | cv.HOUGH_GRADIENT_ALT
circles = cv.HoughCircles(img_gry, method, dp, minDist[, circles[, param1[, par
```

In [ ]: