OpenCV Wrapper Documentation

Release 0.2.3

Andreas Bergem

Aug 17, 2019

Contents

Release v0.2.3

CHAPTER 1

Getting Started

1.1 Installation

Getting started is easy, using pip or pipenv!:

```
pip(env) install opencv-wrapper
```

1.2 Reading and writing

Reading and writing images is done using plain OpenCV:

```
import cv2 as cv

image = cv.imread("path/to/infile")
cv.imwrite("path/to/outfile", image)
```

Reading videos, however, is a bit more tedious in OpenCV. We therefore have a more pythonic approach, using OpenCV Wrapper:

```
import cv2 as cv
import opencv_wrapper as cvw

with cvw.load_video("path/to/file") as video:
    for frame in video:
        cv.imshow("Frame", frame)
        cvw.wait_key(1)
```

Alternatively, we can read a range of frames:

```
import cv2 as cv
import opencv_wrapper as cvw
```

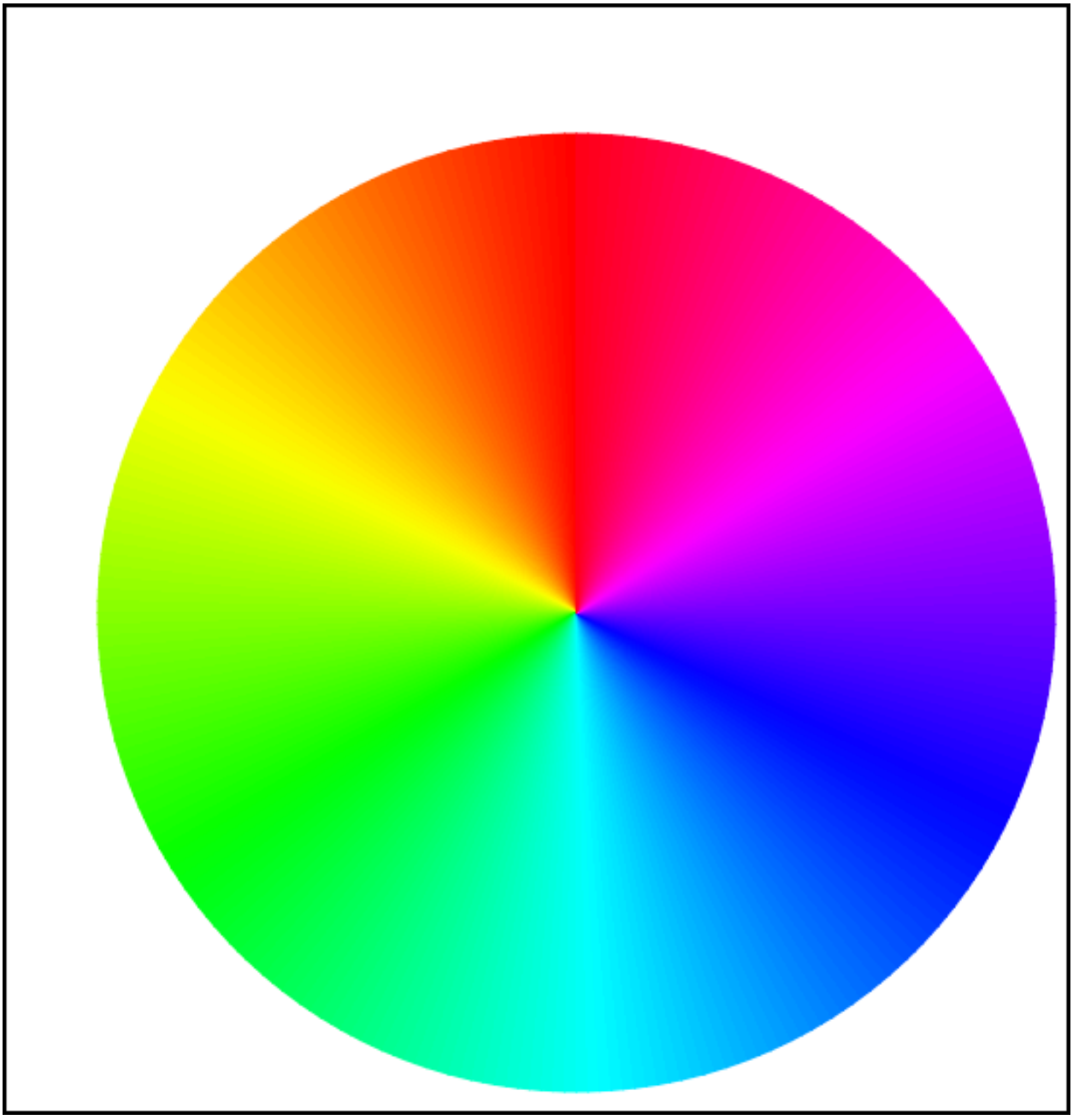
(continues on next page)

(continued from previous page)

```
with cvw.load_video("path/to/file") as video:
    for frame in cvw.read_frames(video, start, stop, step):
        cv.imshow("Frame", frame)
        cvw.wait_key(1)
```

1.3 Larger Example - Rotate A Color Wheel

Say we have the following color wheel image, which we want to rotate.



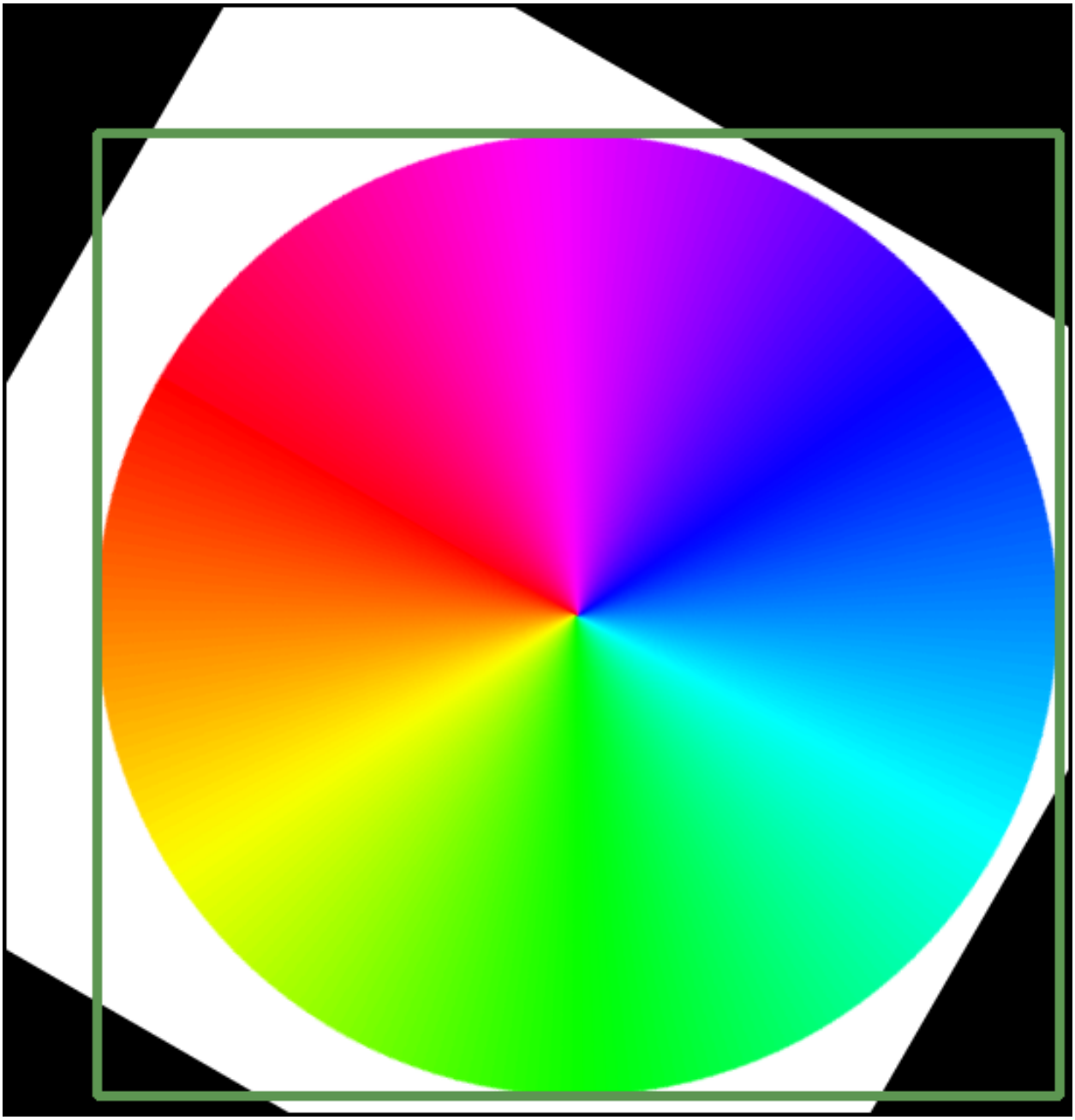
We of course want to rotate it at it's center, which is not in the center of the image. A possible solution using OpenCV would be:

```
import cv2 as cv
import random

img = cv.imread("resources/color_wheel_invert.png")
gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY)
_, otsu = cv.threshold(gray, 250, 255, cv.THRESH_BINARY_INV)
_, contours, _ = cv.findContours(otsu, cv.RETR_EXTERNAL, cv.CHAIN_APPROX_SIMPLE)
contour = contours[0]
rect = cv.boundingRect(contour)  # Gives a tuple (x, y, w, h)
x, y, w, h = rect

color = [random.randint(0, 255) for _ in range(3)]

degrees = 60
center = (x + w / 2), (y + h / 2)
rotation_matrix = cv.getRotationMatrix2D(center, degrees, 1)
rotated_image = cv.warpAffine(img, rotation_matrix, gray.shape[::-1])

cv.rectangle(rotated_image, (x, y), (x + w, y + h), color)

cv.imshow("Image", rotated_image)
cv.waitKey(0)
```

We first convert the image to gray scale. The color wheel in gray scale does not contain any values of pure white. We can therefore threshold the image at a high threshold, to segment the color wheel.

We then find contours in the image (which in this case only will be one contour), and find the bounding rectangle enclosing the contour. From this rectangle we can find the center point by the means of the top left corner, the height and width. We use this to create a rotation matrix, and call the affine warp function. Lastly, we draw a rectangle around the found contour. This is just for viewing pruposes.

We get the following result.



Although a perfectly fine solution, there are a lot of things that can be simplified. We also cannot help but rotate the whole image. Here is a solution using opencv_wrapper:

```
import cv2 as cv
import opencv_wrapper as cvw

img = cv.imread("resources/color_wheel_invert.png")
gray = cvw.bgr2gray(img)
otsu = cvw.threshold_binary(gray, 250, inverse=True)
contours = cvw.find_external_contours(otsu)
contour = contours[0]
rect = contour.bounding_rect  # Gives a Rect object
degrees = 60
```

(continues on next page)

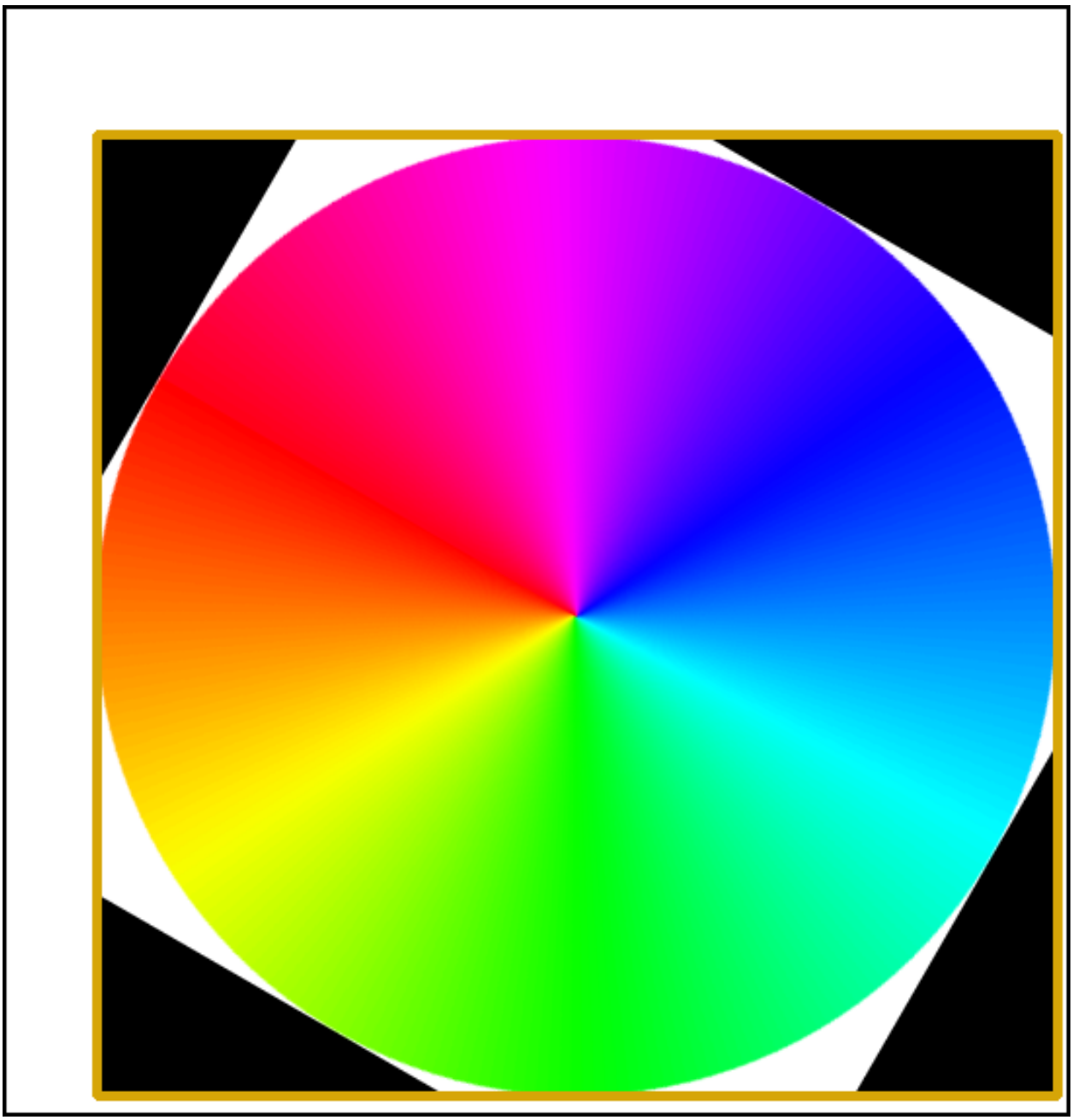
(continued from previous page)

```
center = rect.center  # Gives a Point object
top_left = rect.tl  # Gives a Point object
new_center = center - top_left
img[rect.slice] = cvw.rotate_image(
    img[rect.slice], new_center, degrees, unit=cvw.AngleUnit.DEGREES
)
cvw.rectangle(img, rect, cvw.Color.RANDOM)

cv.imshow("Image", img)
cvw.wait_key(0)
```

We again follow the same approach. However, with the Contour class, we can simply call the bounding rect property. This yields a Rect object, which has a center property. Convenient.

Where we before were left with no (obvious) choice but to rotate the whole image, we can now simply slice the image at the rectangle, only rotating the figure itself. For this exact purpose, it doesn't make much difference, but it is a demonstration. We find the new center from which to rotate, and simply call the rotate image function. We can here choose whether to use degrees or radians. Lastly we draw a rectangle with a random color.

We get the following result.



Not only is this a tad less tedious to write, but we are also easily able to rotate only the relevant part of the circle by slicing[1]. The contour, rectangle and point objects are also an ease to work with.

Other Area of Ease While not providing examples, there are many other parts of the OpenCV that become an ease to work with, when using opencv_wrapper. Areas include

- Morphology
- Image normalization
- Color conversion

[1] Disclosure: The slicing is not that hard to accomplish, from *x, y, w, h*. We can create it like this:

```
our_slice = (slice(y, y+h), slice(x, x+w))
```

- Thresholding
- Image smoothing

CHAPTER 2

Developer Interface

2.1 Model

class opencv_wrapper.model.**Contour**(*points*)

Model class for a contour.

The points come from cv2.findContours(). Using *find_external_contours()* is preferred.

area

Return the area computed from cv.moments(points).

Return type float

Returns The area of the contour

bounding_rect

Return the bounding rectangle around the contour.

Uses cv2.boundingRect(points).

Return type *Rect*

Returns The bounding rectangle of the contour

center

Return the center point of the area.

Due to skewed densities, the center of the bounding rectangle is preferred to the center from moments.

Return type *Point*

Returns The center of the bounding rectangle

class opencv_wrapper.model.**Point**(*x*, *y*)

Model class for a point.

Points can be negated, added, subtracted and iterated over, yielding x, y.

cartesian()

Return type Tuple[float, float]

Returns The point represented as cartesian coordinates

norm
Return the absolute L2 norm of the point. Alias for *cvw.norm(point)*.

Return type float

Returns The absolute L2 norm of the point

classmethod origin()

Return type *Point*

Returns Return the origin point, Point(0, 0)

polar()

Return type Tuple[float, float]

Returns The point represented as polar coordinates

class opencv_wrapper.model.**Rect**(*x*, *y*, *width*, *height*, *, *padding=0*)
Model class of a rectangle.

Rectangles can be iterated over, yielding x, y, width, height.

Rectangles can also be divided. The division is applied to x, y, the width and the height of the rectangle. The makes the rectangle fit to an image shrinked by the same factor.

A test whether or not a point is located inside the rectangle can be checked by the *in* keyword: *if point in rect*.

If padding is given, the rectangle will be *padding* pixels larger in each of its sides. The padding can be negative.

Parameters

- **x** (float) – The top-left x coordinate.
- **y** (float) – The top-left y coordinate.
- **width** (float) – The width of the rectangle.
- **height** (float) – The height of the rectangle.
- **padding** (float) – The padding to be applied to the rectangle.

area

Return type float

Returns The area of the rectangle.

bl

Return type *Point*

Returns The bottom-left corner of the rectangle.

br

Return type *Point*

Returns The bottom-right corner of the rectangle.

cartesian_corners()
Yields the rectangle as top-left and bottom-right points, as used in cv2.rectangle.

Return type Tuple[Tuple[float, float], Tuple[float, float]]

Returns The top-left and bottom-right corners of the rectangle as cartesian two-tuples.

center

Return type *Point*

Returns The center point of the rectangle.

empty()

Return type bool

Returns Whether or not the rectangle is empty.

slice

Creates a slice of the rectangle, to be used on a 2-D numpy array.

For example *image[rect.slice] = 255* will fill the area represented by the rectangle as white, in a gray-scale, uint8 image.

Return type Tuple[*slice*, *slice*]

Returns The slice of the rectangle.

tl

Return type *Point*

Returns The top-left corner of the rectangle.

tr

Return type *Point*

Returns The top-right corner of the rectangle.

2.2 Display

class opencv_wrapper.display.**LineStyle**

An enumeration.

opencv_wrapper.display.**circle**(*image*, *center*, *radius*, *color*, *thickness=1*)

Draw a circle on *image* at *center* with *radius*.

Parameters

- **image** (ndarray) – The image to draw the circle
- **center** (Union[*Point*, Tuple[int, int]]) – The center at which to draw the circle
- **radius** (int) – The radius of the circle
- **color** (Union[*Color*, int, Tuple[int, int, int]]) – The color of the circle.
- **thickness** (int) – The thickness of the circle; can be -1 to fill the circle.

opencv_wrapper.display.**draw_contour**(*image*, *contour*, *color*, *thickness=1*)

Draw a contour on an image.

Parameters

- **image** (ndarray) – Image to draw on
- **contour** (*Contour*) – Contour to draw
- **color** (Union[*Color*, int, Tuple[int, int, int]]) – Color to draw

- **thickness** – Thickness to draw with

opencv_wrapper.display.**draw_contours**(*image*, *contours*, *color*, *thickness=1*)
Draw multiple contours on an image

Parameters

- **image** (ndarray) – Image to draw on
- **contours** (Sequence[*Contour*]) – Contours to draw
- **color** (Union[*Color*, int, Tuple[int, int, int]]) – Color to draw with
- **thickness** – Thickness to draw with

opencv_wrapper.display.**line**(*image*, *point1*, *point2*, *color*, *thickness=1*, *line_style=<LineStyle.SOLID: 1>*)
Draw a line from *point1* to *point2* on *image*.

Parameters

- **image** (ndarray) – The image to draw the line
- **point1** (Union[*Point*, Tuple[int, int]]) – The starting point
- **point2** (Union[*Point*, Tuple[int, int]]) – The ending point
- **color** (Union[*Color*, int, Tuple[int, int, int]]) – The color of the line
- **thickness** (int) – The thickness of the line
- **line_style** (*LineStyle*) – The line style to draw. For LineStyle.DASHED, only thickness 1 is currently supported.

opencv_wrapper.display.**put_text**(*image*, *text*, *origin*, *color*, *thickness=1*, *scale=1*)
Put *text* on *image* at *origin*.

Parameters

- **image** (ndarray) – The image to draw the text
- **text** (str) – The text to be drawn
- **origin** (Union[*Point*, Tuple[int, int]]) – The origin to start the text. The bottom of the first character is set in the origin.
- **color** (Union[*Color*, int, Tuple[int, int, int]]) – The color of the text
- **thickness** (int) – The thickness of the text
- **scale** (float) – The scale of the text.

opencv_wrapper.display.**rectangle**(*image*, *rect*, *color*, *thickness=1*, *line_style=<LineStyle.SOLID: 1>*)
Draw a rectangle on *image*.

Parameters

- **image** (ndarray) – The image to draw the rectangle
- **rect** (*Rect*) – The rectangle to be drawn
- **color** (Union[*Color*, int, Tuple[int, int, int]]) – The color of the rectangle
- **thickness** (int) – The thickness of the lines; can be -1 to fill the rectangle.
- **line_style** (*LineStyle*) – The line style to draw. For LineStyle.DASHED, only thickness 1 is currently supported.

opencv_wrapper.display.**wait_key**(*delay*)

Wait for a key event infinitely (if *delay* is 0) or *delay* amount of milliseconds.

An alias for cv.waitKey(delay) & 0xFF. See cv.waitKey(delay) for further documentation. Comparison of the key pressed can be found by *ord(str)*. For example

```
>>> if wait_key(0) == ord('q'): continue
```

Parameters **delay** (int) – Amount of milliseconds to wait, or 0 for infinitely.

Return type str

Returns The key pressed.

2.3 Utilities

class opencv_wrapper.utils.**Color**

Color enum for predefined colors.

Color.RANDOM returns a random color. Colors can be added together.

BLACK = (0, 0, 0)

BLUE = (200, 0, 0)

CYAN = (255, 255, 0)

GREEN = (0, 200, 0)

MAGENTA = (255, 0, 255)

RED = (0, 0, 200)

WHITE = (255, 255, 255)

YELLOW = (0, 255, 255)

2.4 Image Operations

class opencv_wrapper.image_operations.**AngleUnit**

Enum for which angle unit to use.

DEGREES = 2

RADIANS = 1

class opencv_wrapper.image_operations.**MorphShape**

Enum for determining shape in morphological operations.

Alias for OpenCV's morph enums.

CIRCLE = 2

CROSS = 1

RECT = 0

class opencv_wrapper.image_operations.**AngleUnit**

Enum for which angle unit to use.

class opencv_wrapper.image_operations.**MorphShape**
Enum for determining shape in morphological operations.

Alias for OpenCV's morph enums.

opencv_wrapper.image_operations.**bgr2gray**(*image*)
Convert image from BGR to gray

Parameters **image** (ndarray) – Image to be converted

Return type ndarray

Returns Converted image

opencv_wrapper.image_operations.**bgr2hls**(*image*)
Convert image from BGR to HLS color space

Parameters **image** (ndarray) – Image to be converted

Return type ndarray

Returns Converted image

opencv_wrapper.image_operations.**bgr2hsv**(*image*)
Convert image from BGR to HSV color space

Parameters **image** (ndarray) – Image to be converted

Return type ndarray

Returns Converted image

opencv_wrapper.image_operations.**bgr2luv**(*image*)
Convert image from BGR to CIE LUV color space

Parameters **image** (ndarray) – Image to be converted

Return type ndarray

Returns Converted image

opencv_wrapper.image_operations.**bgr2xyz**(*image*)
Convert image from BGR to CIE XYZ color space

Parameters **image** (ndarray) – Image to be converted

Return type ndarray

Returns Converted image

opencv_wrapper.image_operations.**canny**(*image*, *low_threshold*, *high_threshold*, *high_pass_size=3*, *l2_gradient=True*)
Perform Canny's edge detection on *image*.

Parameters

- **image** (ndarray) – The image to be processed.
- **low_threshold** (float) – The lower threshold in the hysteresis thresholding.
- **high_threshold** (float) – The higher threshold in the hysteresis thresholding.
- **high_pass_size** (int) – The size of the Sobel filter, used to find gradients.
- **l2_gradient** – Whether to use the L2 gradient. The L1 gradient is used if false.

Return type ndarray

Returns Binary image of thinned edges.

opencv_wrapper.image_operations.**dilate**(*image*, *kernel_size*, *shape=<MorphShape.RECT: 0>*, *iterations=1*)

Dilate *image* with *kernel_size* and *shape*.

Parameters

- **image** (ndarray) – Image to be dilated
- **kernel_size** (int) – Kernel size to dilate with
- **shape** (*MorphShape*) – Shape of kernel
- **iterations** (int) – Number of iterations to perform dilation

Return type ndarray

Returns The dilated image

opencv_wrapper.image_operations.**erode**(*image*, *kernel_size*, *shape=<MorphShape.RECT: 0>*, *iterations=1*)

Erode *image* with *kernel_size* and *shape*.

Parameters

- **image** (ndarray) – Image to be eroded
- **kernel_size** (int) – Kernel size to erode with
- **shape** (*MorphShape*) – Shape of kernel
- **iterations** (int) – Number of iterations to perform erosion

Return type ndarray

Returns The eroded image

opencv_wrapper.image_operations.**find_external_contours**(*image*)

Find the external contours in the *image*.

Alias for *cv2.findContours(image, cv.RETR_EXTERNAL, cv.CHAIN_APPROX_SIMPLE)*

Parameters **image** (ndarray) – The image in with to find the contours

Return type Tuple[*Contour*, ...]

Returns A tuple of Contour objects

opencv_wrapper.image_operations.**gray2bgr**(*image*)

Convert image from gray to BGR

Parameters **image** (ndarray) – Image to be converted

Return type ndarray

Returns Converted image

opencv_wrapper.image_operations.**morph_close**(*image*, *kernel_size*, *shape=<MorphShape.RECT: 0>*, *iterations=1*)

Morphologically close *image* with *kernel_size* and *shape*.

Parameters

- **image** (ndarray) – Image to be closed
- **kernel_size** (int) – Kernel size to close with
- **shape** (*MorphShape*) – Shape of kernel
- **iterations** – Number of iterations to perform closing

Return type ndarray

Returns The closed image

opencv_wrapper.image_operations.**morph_open**(*image*, *kernel_size*, *shape=<MorphShape.RECT: 0>*, *iterations=1*)

Morphologically open *image* with *kernel_size* and *shape*.

Parameters

- **image** (ndarray) – Image to be opened
- **kernel_size** (int) – Kernel size to open with
- **shape** (*MorphShape*) – Shape of kernel
- **iterations** – Number of iterations to perform opening

Return type ndarray

Returns The opened image

opencv_wrapper.image_operations.**normalize**(*image*, *min=0*, *max=255*, *dtype=None*)

Normalize image to range [*min*, *max*].

Parameters

- **image** (ndarray) – Image to be normalized
- **min** (int) – New minimum value of image
- **max** (int) – New maximum value of image
- **dtype** (Optional[dtype]) – Output type of image. Default is same as *image*.

Return type ndarray

Returns The normalized image

opencv_wrapper.image_operations.**resize**(*image*, ***, *factor=None*, *shape=None*)

Resize an image with the given factor or shape.

Either shape or factor must be provided. Using *factor* of 2 gives an image of half the size. Using *shape* gives an image of the given shape.

Parameters

- **image** (ndarray) – Image to resize
- **factor** (Optional[int]) – Shrink factor. A factor of 2 halves the image size.
- **shape** (Optional[Tuple[int, ...]]) – Output image size.

Return type ndarray

Returns A resized image

opencv_wrapper.image_operations.**rotate_image**(*image*, *center*, *angle*, *unit=<AngleUnit.RADIANS: 1>*)

Rotate *image angle* degrees at *center*. *unit* specifies if *angle* is given in degrees or radians.

Parameters

- **image** (ndarray) – The image to be rotated.
- **center** (*Point*) – The center of the rotation
- **angle** (float) – The angle to be rotated

- **unit** (*AngleUnit*) – The unit of the angle

Return type ndarray

Returns The rotated image.

opencv_wrapper.image_operations.**threshold_adaptive**(*image*, *block_size*, *c=0*, ***, *weighted=True*, *inverse=False*)

Adaptive thresholding of *image*, using a (weighted) local mean.

A local threshold value is determined for each *block_size x block_size* window. If *weighted* is true, the gaussian weighted mean is used. If not, the mean is used.

Parameters

- **image** (ndarray) – Input image.
- **block_size** (int) – The size of the local windows.
- **c** (int) – Constant to be subtracted from the (weighted) mean.
- **weighted** (bool) – Whether or not to weight the mean with a gaussian weighting.
- **inverse** (bool) – Whether or not to inverse the image output.

Return type ndarray

Returns The thresholded image.

2.5 Tracking

opencv_wrapper.tracking.**dense_optical_flow**(*prev_frame*, *next_frame*, *pyr_scale=0.5*, *levels=3*, *winsize=11*, *iterations=1*, *poly_n=4*, *poly_sigma=1.1*, *gaussian_window=True*, *initial_flow=None*)

Calculate the dense optical flow between two frames, using the Farnebäck method.

For further documentation on the parameters, see OpenCV documentation for cv2.calcOpticalFlowFarnaback.

Parameters

- **prev_frame** (ndarray) – The initial frame
- **next_frame** (ndarray) – The frame after *prev_frame*, with displacement.

Return type ndarray

Returns An image with the shape

2.6 Video

Convenience functions for reading and writing videos.

Usage:

```
>>> import cv2 as cv
>>> with load_video("path/to/file") as video:
>>>     for frame in video:
>>>         cv.imshow("Frame", frame)
>>>         cv.waitKey(1)
```

class opencv_wrapper.video.**VideoCapture**(*source*)

A video capture object for displaying videos.

For normal use, use *load_camera()* and *load_camera()* instead.

The object can be created using either an index of a connected camera, or a filename of a video file.

Parameters **source** (Union[int, str]) – Either index of camera or filename of video file.

class opencv_wrapper.video.**VideoWriter**(*filename*, *fps=None*, *capture=None*, *fourcc='MJPG'*)

A video writer for writing videos, using OpenCV's *cv.VideoWriter*.

The video writer is lazy, in that it waits to receive the first frame, before determining the frame size for the video writer. This is in contrast to OpenCV's video writer, which expects a frame size up front.

Either *fps* or *capture* must be provided.

For additional documentation, see cv2.VideoWriter documentation

Parameters

- **filename** (str) – Name of the output video file.
- **fps** (Optional[int]) – Framerate of the created video stream.
- **capture** (Optional[Any]) – A capture object from cv2.VideoCapture or *load_video()*. Used to retrieve fps if *fps* is not provided.
- **fourcc** (str) – 4-character code of codec used to compress the frames. See documentation

write(*frame*)

Write a frame to the video.

The frame must be the same size each time the frame is written.

Parameters **frame** – Image to be written

opencv_wrapper.video.**load_camera**(*index=0*)

Open a camera for video capturing.

Parameters **index** (int) – Index of the camera to open.

For more details see cv2.VideoCapture(index) documentation

Return type Iterator[Any]

opencv_wrapper.video.**load_video**(*filename*)

Open a video file

Parameters **filename** (str) – It an be:

- Name of video file
- An image sequence
- A URL of a video stream

For more details see cv2.VideoCapture(filename) documentation

Return type Iterator[Any]

opencv_wrapper.video.**read_frames**(*video*, *start=0*, *stop=None*, *step=1*)
Read frames of a video object.

Start, stop and step work as built-in range.

Parameters

- **video** (*VideoCapture*) – Video object to read from.
- **start** (int) – Frame number to skip to.
- **stop** (Optional[int]) – Frame number to stop reading, exclusive.
- **step** (int) – Step to iterate over frames. Similar to range's step. Must be greater than 0.

Return type Iterator[ndarray]

2.7 Miscellaneous Functions

opencv_wrapper.misc_functions.**line_iterator**(*image*, *p1*, *p2*)
Produces and array that consists of the coordinates and intensities of each pixel in a line between two points.

Credit: https://stackoverflow.com/questions/32328179/opencv-3-0-python-lineiterator

Parameters

- **image** (ndarray) – The image being processed
- **p1** (*Point*) – The first point
- **p2** (*Point*) – The second point

Return type ndarray

Returns An array that consists of the coordinates and intensities of each pixel on the line. (shape: [numPixels, 3(5)], row = [x,y, intensity(b, g, r)]), for gray-scale(bgr) image.

opencv_wrapper.misc_functions.**norm**(*input*)
Calculates the absolute L2 norm of the point or array. :type input: Union[*Point*, ndarray] :param input: The n-dimensional point :rtype: float :return: The L2 norm of the n-dimensional point

opencv_wrapper.misc_functions.**rect_intersection**(*rect1*, *rect2*)
Calculate the intersection between two rectangles.

Parameters

- **rect1** (*Rect*) – First rectangle
- **rect2** (*Rect*) – Second rectangle

Return type Optional[*Rect*]

Returns A rectangle representing the intersection between *rect1* and *rect2* if it exists, else None.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

o

Index

M

N

O

P

R

S

T

V

W

Y