OpenCV-Python for Data Science

Michael Komodromos

Contents

1 Introduction

OpenCV is a library that includes several hundred computer vision (CV) algorithms. It was started in 1999 by Gary Bradsky and first released in 2000. OpenCV supports a wide range of programming languages such as: C++, Java, JavaScript and Python.

1.1 OpenCV-Python

OpenCV-Python brings together the best qualities of OpenCV and Python. Allowing for great performance and the stylistic qualities of Python. This is achieved by creating Python wrappers around the C/C++ modules. Meaning code is just as fast as if it was written in C/C++ (as C/C++ is being executed in the background).

The Python OpenCV modules are used in conjunction with numpy, a highly optimized library for numerical operations. Providing a MATLAB style syntax. In the background OpenCV array structures are converted to and from Numpy arrays.

Remark. OpenCV Python is the appropriate tool for fast prototyping.

1.2 Installation

The following installation instructions are for Python version 3.6+. It is possible to use older version of Python, however unless there is a specific reason to do so, it is not recommended.

Remark. Python version 3.6 comes with python-venv. The module allows you to easily create virtual environments.

Note. Virtual environments become extremely useful and future proofs your system from mismatched dependencies.

1.2.1 Setting up a virtual environment

From your terminal run the following:

```
python3 -m venv env
. env/bin/activate
```

You should now see `(env)` at the start of the command line.

1.2.2 Installing OpenCV-Python

```
pip install opencv-python
```

That's it! All dependencies (numpy) are installed alongside opencv. To test everything is working type:

```
>>> import cv2
```

2 Numpy

Basic knowledge of nunpy is needed to work with OpenCV. Working knowledge of Python is assumed.

2.1 Arrays

The main object in Numpy is the homogeneous multidimensional array. A table of elements typically numbers all of the same type.

Remark. homogeneous implies that all elements are of the same type.

Note. In numpy dimensions are called axes

Example 2.1.1. A numpy array type

```
[[ 1., 0., 0.],
 [ 0., 1., 2.]]
```

2 axes, length 3

Numpy's array class is called `ndarray`. It is also know by `array` in the numpy namespace of course.

2.1.1 Creating Arrays

There are several ways to create arrays. *Note:* the array type is deduced from the type of the elements in the sequence.

Example 2.1.2. Creating arrays in numpy

```
>>> import numpy as np
>>> a = np.array([1, 2, 3])
```

Creates a 1 dimensional, or an array with 1 axis, of type int64.

The `array` function transforms multiple sequences into the respective multi-dimensional array.

Example 2.1.3. Creating a 3 dimensional array

```
>>> b = np.array([
    [
        [1,2,3],
        [2,3,4]
    ],
    [
        [3,4,5],
        [4,5,6]
    ]
])
>>> b.ndim
3
```

Note. The data type can be set with the dtype parameter
ex: `np.array( [ [1,2], [3,4] ], dtype=complex)`

2.1.2 Attributes

ndim
returns number of axes.

shape
returns a tuple with the number of rows `n` and the number of columns `m` *i.e.* `(n, m)`.

size
returns total number of elements in array.

dtype
returns an object describing the elements data type, float64, int64 etc.

2.1.3 Functions

`np.zeros(shape, dtype=float, order='C')`
Creates an array full of zeros.

Example 2.1.4. np.zeros

```
>>> np.zeros((5, 7))
```

`np.ones(shape, dtype=float, order='C')`
Creates an array full of ones.

Example 2.1.5. np.ones

```
>>> np.ones((1, 3))
```

`np.empty(shape, dtype=float, order='C')`
Creates an array whose initial content is random.

Example 2.1.6. np.empty

```
>>> np.empty((3,4))
```

`np.full(shape, val, dtype=None, order='C')`
Return a new array of given shape and type, filled with val.

Example 2.1.7. np.full

```
>>> np.full((3,4), 42)
```

`np.arange(start, stop, step, dtype=None)`
similar to native `range` function, but returns anumpy array object.

Example 2.1.8. np.arange

```
>>> np.arange(0, 100, 5)
```

```
np.linspace(start, stop, num=50, endpoint=True, retstep=False, dtype=None,
 axis=0)
```
returns an array of floating point integers, that are linearly spaced.

Example 2.1.9. np.linespace

```
>>> np.linspace(0,2,9)
```

2.2 Basic Operations

Arithmetic operations on arrays are applied elementwise, resulting in an new array being created and filled with the result.

Example 2.2.1. Arithmetic

```
>>> a = np.ones(3)
>>> b = np.zeros(3)
>>> b[1] = 5
>>> a * 5
array([ 5., 5., 5.]
>>> a * b
array([0., 5., 0.])
```

Note. the matrix product can be performed using the `@` operator, or with the `dot()` method.

Example 2.2.2. Matrix Multiplication

```
>>> A = np.array([[1,2],[0,1]])
>>> B = np.array([[2,0],[3,4]])
>>> A @ B
array([[5, 4],
       [3, 4]])
>>> A.dot(B)
array([[5, 4],
       [3, 4]])
```

Some operators like `+=` and `*=` modify the existing array.

Example 2.2.3. Modifying arrays

```
>>> a = np.ones((2,3), dtype=int)
>>> b = np.random.random((2,3))
>>> a *= 3
>>> a
array([[3, 3, 3],
       [3, 3, 3]])
>>> b += a
>>> b
array([[ 3.417022  ,  3.72032449,  3.00011437],
       [ 3.30233257,  3.14675589,  3.09233859]])
```

2.2.1 Unary Operation

Def 1. Unary Operations

An operation that takes a single operand. *ex:* `not` is an operation that logically negates an expression.

`np.sum(axis=None, dtype=None, out=None, where=True)`
Returns the sum of all elements in an array

Example 2.2.4. np.sum

```
>>> a = np.ones(2)
>>> a.sum()
2
```

`np.min(iterable, *args)`
Returns the minimum element of an array.

Example 2.2.5. np.min

```
>>> a = np.array([1, 5, 10])
>>> a.min()
1
```

`np.max(iterable, *args)`
Returns the maximum element of an array.

Example 2.2.6. np.max

```
>>> a = np.array([1, 5, 10])
>>> a.max()
10
```

2.2.2 Universal Functions

Numpy provides many familiar mathematical functios.

`np.sin(x, out=None, *args)`
Applies the sine operation on a given data type.

Example 2.2.7. np.sin

```
>>> A = np.arange(3)
>>> np.sin(B)
array([0.         , 0.84147098, 0.90929743])
```

`np.exp(x, out=None, *args)`
Applies an exponential operation to a given data type.

Example 2.2.8. np.exp

```
>>> A = np.arange(3)
>>> np.exp(A)
array([ 1.         ,  2.71828183,  7.3890561 ])
```

`np.cumsum(a, axis=None, dtype=None, out=None)`
Returns an array comataining the cumulative sum of elements.

Example 2.2.9. np.cumsum

```
>>> A = np.ones(3)
>>> np.cumsum(A)
array([ 1.,  2., 3.])
```

Function	Operation
all	whether elements along an axis evaluate to a truthy value
any	Tests whether any element along an axis evaluates to a truthy value
argmax	returns the indecie of the maximum values along and axis
argmin	returns the indicie of the miimum values along and axis
average	computes the weighted average along a given axis
bincount	counts the number of occurrences of each value in an array of non negative integers
ceil	returns the ceiling of an array element wise; *i.e.* the smallest int i such that $i \geq x$
clip	given an interval $a \leq i \leq b$ element i outside the interval is set to the interval edges
conj	returns the complex conjugate element wise
corrcoef	returns the Pearson product moment correlation coefficient
cov	estimates a covariance matrix given data and weights
cross	returns the cross product of two vectors
cumprod	returns the cumulative product of elements along a given axis
cumsum	return the cumulative sum of elements along a given axis
diff	calculate the n-th discrete difference along the given axis
dot	dot product of two arrays
floor	return the floor of the input, element-wise
inner	inner product of two arrays
maximum	element-wise maximum of array elements
mean	compute the arithmetic mean along the specified axis
median	compute the median along the specified axis
minimum	element-wise minimum of array elements
nonzero	return the indices of the elements that are non-zero
outer	compute the outer product of two vectors
prod	return the product of array elements over a given axis
sort	return a sorted copy of an array
std	compute the standard deviation along the specified axis
sum	sum of array elements over a given axis
trace	return the sum along diagonals of the array
transpose	permute the dimensions of an array
var	compute the variance along the specified axis.
vdot	return the dot product of two vectors
where	return elements chosen from x or y depending on condition

Table 1: functions in numpy

2.2.3 Indexing, Slicing and Iterating

One-dimensional arrays can be indexed, sliced and iterated over, similar to native python lists.

Example 2.2.10. Indexing

```
>>> a = np.arange(10)**3
>>> a
array([  0,   1,   8,  27,  64, 125, 216, 343, 512, 729])
>>> a[2]
8
```

Example 2.2.11. Slicing

```
>>> a[2:5]
array([ 8, 27, 64])
```

Example 2.2.12. Iterating

```
>>> a[:6:2] = -1000    # equivalent to a[0:6:2] = -1000;
    from start to position 6, exclusive, set every 2nd
    element to -1000
>>> a
array([-1000,     1, -1000,    27, -1000,   125,   216,
    343,   512,   729])
>>> a[ : :-1] # reversed
array([  729,   512,   343,   216,   125, -1000,    27,
    -1000,     1, -1000])
>>> for i in a:
...     print(i)
...
0
1
```

Multidimensional arrays can have one index per axis.

Example 2.2.13. Indexing Multidimensional Arrays

```
>>> def f(x,y):
...     return 10*x+y
...
>>> b = np.fromfunction(f,(5,4),dtype=int)
>>> b
array([[ 0,  1,  2,  3],
       [10, 11, 12, 13],
       [20, 21, 22, 23],
       [30, 31, 32, 33],
       [40, 41, 42, 43]])
>>> b[2,3]
23
```

Example 2.2.14. Slicing Multidimensional Arrays

```
>>> b[0:5, 1] # each row in the second column of b
array([ 1, 11, 21, 31, 41])
>>> b[ : ,1]  # equivalent to the previous example
array([ 1, 11, 21, 31, 41])
>>> b[1:3, : ] # columns in the second and third row of b
array([[10, 11, 12, 13],
       [20, 21, 22, 23]])
```

Example 2.2.15. Iterating over Multidimensional Arrays

```
>>> for row in b:
...     print(row)
...
[0 1 2 3]
[10 11 12 13]
[20 21 22 23]
[30 31 32 33]
[40 41 42 43]
```

Note. Iterating is done w.r.t. the first axis

The `flat` attribute can be used to create an iterator over all the elements in an array

Example 2.2.16. The `flat` attribute

```
>>> for element in b.flat:
...     print(element)
...
0
1
2
# truncated
```

2.3 Shape Manipulation

2.3.1 Changing the shape of an array

An array's shape is determined by the number of elements along each axis.

Example 2.3.1. Array shape

```
>>> a = np.floor(10*np.random.random((3,4)))
>>> a.shape
(3, 4)
```

Various methods are used to change the shape of an array.

`np.ravel(a, order='C')`
Returns a flatten version of an array.

Example 2.3.2. np.ravel

```
>>> a.ravel()
array([ 2.,  8.,  0.,  6.,  4.,  5.,  1.,  1.,  8.,  9.,
    3.,  6.])
```

Note. The command produces the same output as `np.array(list(a.flat))`

`np.reshape(a, newshape, order='C')`
Returns an array with a modified shape.

Example 2.3.3. np.reshape

```
>>> a.reshape(6,2)
array([[ 2.,  8.],
       [ 0.,  6.],
       ...
       [ 3.,  6.]])
```

Note. If a dimension of -1 is given then the dimension is automatically calculated.

`ndarray.T`
Returns the transposed array.

Example 2.3.4. T

```
>>> a.T  # returns the array, transposed
array([[ 2.,  4.,  8.],
       [ 8.,  5.,  9.],
       [ 0.,  1.,  3.],
       [ 6.,  1.,  6.]])
```

Note. The `resize` method modifies the array itself.

2.3.2 Stacking arrays

Several arrays can be stacked together along different axes.

`np.vstack(tup)`
Stacks arrays in sequence vertically (row wise).

Example 2.3.5. np.vstack

```
>>> a = np.floor(10*np.random.random((2,2)))
>>> a
array([[ 8.,  8.],
       [ 0.,  0.]])
>>> b = np.floor(10*np.random.random((2,2)))
>>> b
array([[ 1.,  8.],
       [ 0.,  4.]])
>>> np.vstack((a,b))
array([[ 8.,  8.],
       [ 0.,  0.],
       [ 1.,  8.],
       [ 0.,  4.]])
```

`np.hstack(tup)`
Stacks arrays in sequence horizontally (col wise).

Example 2.3.6. np.hstack

```
>>> np.hstack((a,b))
array([[ 8.,  8.,  1.,  8.],
       [ 0.,  0.,  0.,  4.]])
```

`np.column_stack(tup)`
Stack 1D arrays as columns into 2D arrays.

Example 2.3.7. np.column_stack

```
>>> a = np.array((1,2,3))
>>> b = np.array((2,3,4))
>>> np.column_stack((a,b))
array([[1, 2],
       [2, 3],
       [3, 4]])
```

`np.concatenate((a1, a2, ...), axis=0, out=None)`
Join a sequence of arrays along an existing axis.

Example 2.3.8. np.concatenate

```
>>> a = np.array([[1, 2], [3, 4]])
>>> b = np.array([[5, 6]])
```

```
>>> np.concatenate((a, b), axis=0)
array([[1, 2],
       [3, 4],
       [5, 6]])
>>> np.concatenate((a, b.T), axis=1)
array([[1, 2, 5],
       [3, 4, 6]])
>>> np.concatenate((a, b), axis=None)
array([1, 2, 3, 4, 5, 6])
```

2.3.3 Splitting Arrays

`np.hsplit(arr, indices_or_sections)`
Splits an array along its horizontal axis.

Example 2.3.9. np.hsplit

```
>>> x = np.arange(16.0).reshape(4, 4)
>>> x
array([[ 0.,   1.,   2.,   3.],
       [ 4.,   5.,   6.,   7.],
       [ 8.,   9.,  10.,  11.],
       [12.,  13.,  14.,  15.]])
>>> np.hsplit(x, 2)
[array([[  0.,   1.],
       [  4.,   5.],
       [  8.,   9.],
       [12.,  13.]]),
 array([[  2.,   3.],
       [  6.,   7.],
       [10.,  11.],
       [14.,  15.]])]
```

`np.vsplit(arr, indices_or_sections)`
Splits an array along the vertical axis (row wise).

Example 2.3.10.

```
>>> np.vsplit(x, 2)
[array([[0., 1., 2., 3.],
       [4., 5., 6., 7.]]), array([[ 8.,  9., 10., 11.],
       [12., 13., 14., 15.]])]
```

2.4 Copies and Views

Important to distinguish when the data of an array has been copied or not. There are three cases to consider.

2.4.1 No Copy

Assignments make no copy of an array.

Example 2.4.1. Assignment and copying

```
>>> a = np.arange(12)
>>> b = a            # no new object is created
>>> b is a           # a and b are two names for the same
    ndarray object
True
>>> b.shape = 3,4    # changes the shape of a
>>> a.shape
(3, 4)
```

2.4.2 Shallow Copies

`ndarray.view(dtype=None, type=None)`
New view of array with the same data

Example 2.4.2. ndarray.view

```
>>> c = a.view()
>>> c is a
False
>>> c.base is a                        # c is a view of the
    data owned by a
True
>>> c.flags.owndata
False
```

Note. The data type can be changed with the

Note. Changes in c's data also result in a's data changing

Remark. Slicing arrays also result in a view being returned.

2.4.3 Deep Copies

`ndarray.copy(a, order='K')`
Returns an array copy of the given object.

Example 2.4.3. ndarray.copy

```
>>> d = a.copy()                          # a new array
    object with new data is created
>>> d is a
False
```

2.5 Broadcasting

Def 2. Broadcasting

Describes how numpy treats arrays with different shapes during arithmetic operations. The smaller array is "broadcast" across the larger array *s.t.* they have compatible shapes.

Remark. Numpy operations are usually done on pairs of arrays element-wise. The simplest case is that two arrays have the same shape.

Example 2.5.1. Broadcasting simplest case

```
>>> a = np.array([1.0, 2.0, 3.0])
>>> b = np.array([2.0, 2.0, 2.0])
>>> a * b
array([ 2.,  4.,  6.])
```

Remark. This constraint is relaxed when an array and a scalar are combined in an operation

Example 2.5.2. Broadcasting: arrays and scalars

```
>>> a = np.array([1.0, 2.0, 3.0])
>>> b = 2.0
>>> a * b
array([ 2.,  4.,  6.])
```

These rules generalize as follows, two dimensions are compatible when:

1. they are equal
2. one dimension if of length 1

Note. If these conditions are not met then a `ValueError` is raised

Example 2.5.3. Broadcasting

```
A      (4d array):  8 x 1 x 6 x 1
B      (3d array):      7 x 1 x 5
Result (4d array):  8 x 7 x 6 x 5

A      (3d array):  15 x 3 x 5
B      (3d array):  15 x 1 x 5
Result (3d array):  15 x 3 x 5

A      (3d array):  15 x 3 x 5
B      (2d array):       3 x 5
Result (3d array):  15 x 3 x 5

A      (3d array):  15 x 3 x 5
B      (2d array):       3 x 1
Result (3d array):  15 x 3 x 5
```

2.6 Indexing

2.6.1 Indexing with arrays of indices

Arrays containing indices can be used to interact with other arrays.

Example 2.6.1. Indexing with arrays containing indices

```
>>> a = np.arange(12) ** 2 # square numbers
>>> i = np.array([1, 1, 3, 8, 5 ]) # indices
>>> a[i] # elements at positions i
array([ 1,  1,  9, 64, 25])
```

Remark. For multidimensional arrays a single array of indices refers to the first dimension

Note. Multiple indexes can be given to interact with more dimensions.

Example 2.6.2. Indexing multidimensional arrays

```
>>> a = np.arange(12).reshape(3,4)
>>> a
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>> i = np.array( [ [0,1], [1,2] ] )
>>> j = np.array( [ [2,1], [3,3] ] )
>>> a[i,j] # i and j must have equal shape
array([[ 2,  5],
       [ 7, 11]])
```

```

# i, row entries
[ [0, 1],
  [1, 2] ]
# j, col entries
[ [2, 1],
  [3, 3] ]

# applied to
[[ 0,  1,  2,  3],
 [ 4,  5,  6,  7],
 [ 8,  9, 10, 11]]

# results
[[ a[0, 2], a[1, 1] ],
 [ a[1, 3], a[2, 3] ]]

i (0, 0)  j (0, 0)      i (1, 0)  j (0, 1)
i (1, 0)  j (1, 0)      i (1, 1)  j (1, 1)
```

2.6.2 Indexing with Boolean Arrays

Boolean indexing uses an array with the same shape as the original array in order to extract elements. A boolean operation is performed on the original array, for instance $\leq$, $>$, $==$, $isnan()$ and so forth to obtain a boolean indexed array.

Example 2.6.3. Boolean Indexing

```
>>> a = np.arange(12).reshape(3,4)
>>> b = a > 4
>>> b # boolean with a's shape
array([[False, False, False, False],
       [False,  True,  True,  True],
       [ True,  True,  True,  True]])
>>> a[b] # 1D array with the selected elements
array([ 5,  6,  7,  8,  9, 10, 11])
```

Remark. Can be extremely useful for value assignment

Example 2.6.4. Assigning values

```
>>> a[b] = 0 # All elements of 'a' higher than 4 become 0
>>> a
array([[0, 1, 2, 3],
       [4, 0, 0, 0],
       [0, 0, 0, 0]])
```

Note. Alternatively a 1D boolean array can be used to extract the relevant slices of an array

Example 2.6.5. Slicing with boolean arrays

```
>>> a = np.arange(12).reshape(3,4)
>>> b1 = np.array([False,True,True]) # first dim selection
array([[ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

Note. The length of the array must coincide with the length of the dimension

2.7 Constants

Constant	Numpy Var	Definition
Positive Infinity	**PINF**	floating point representation of positive infinity
Negative Infinity	**NINF**	floating point representation of negative infinity
Positive Zero	**PZERO**	floating point representation of positive zero
Negative Zero	**NZERO**	floating point representation of negative zero
Euler's Constant	**e**	base of natural logarithms, e = 2.718281...
Pi	**pi**	pi = 3.14159...
NaN	**nan**	not a number

Table 2: constants in numpy

3 Matplotlib

Matplotlib is a Python 2D plotting library used for producing publication quality figures.

Throughout OpenCV matplotlib is used to interact with images; a basic understanding of matplotlib is required.

To install matplotlib in a given virtual environment run

```
pip install matplotlib
```

3.1 General Concepts

Matplotlib is organised in a hierarchical structure. At the top of the hierarchy is the state-machine environment, provided by the `matplotlib.pyplot` module. The next level down is the object-oriented interface, where the user explicitly creates and keeps track of figures and axes objects. Beyond that a purely object oriented approach can be taken for fine grain control, *ie.* building GUI apps.

Def 3. State Machine

A state machine consists of a set of states s and a set of events Σ which change the current state. In terms of matplotlib the state is a plot, typically references as `plt` and the set of events are methods that change the plot; adding points, lines, titles etc.

3.2 The Structure of Figures

3.2.1 Figures

Figure are used to keep track of the canvas, child axes, titles, legends etc.

Note. The canvas is responsible for drawing the plot

Note. A figure needs at least one set of axes

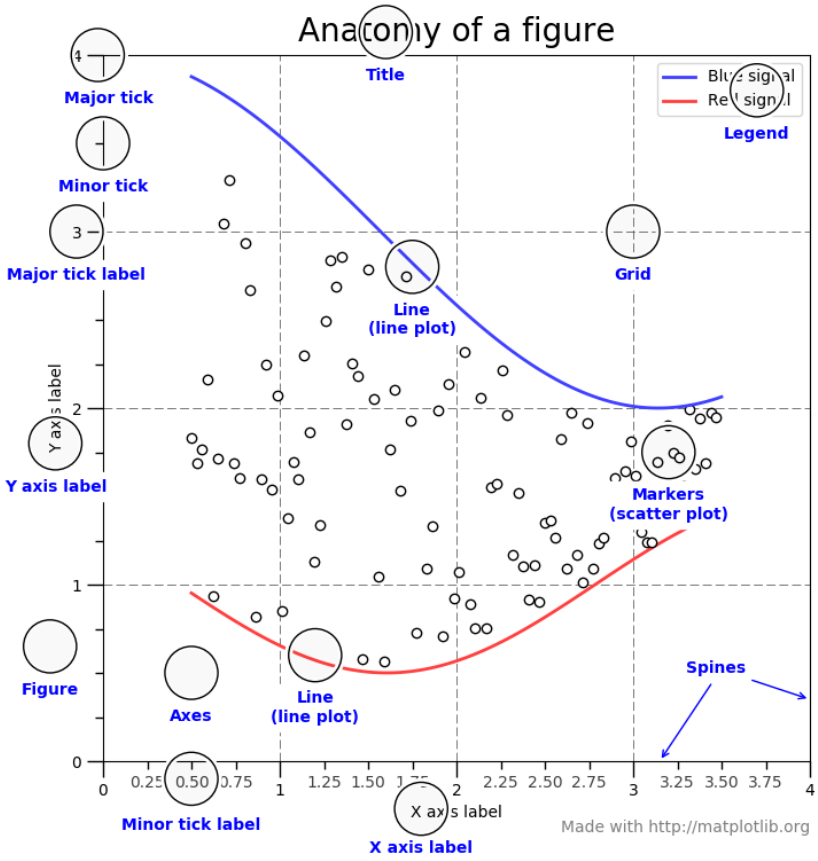

Figure 1: Anatomy of a matplotlib figure

3.2.2 Axes

Axes can be thought of as a plot. A given figure can have multiple axes but an axes can only be in one figure. Axes are made up of `Axis` objects, representing the dimensionality of the plot, *i.e.* 3 axis $\Rightarrow$ 3D plot.

3.2.3 Axis

Number-line like objects. The set the graph limits and generate ticks (marks along the axis) and tick labels.

3.2.4 Artist

Everything that is visible on a figure is an artist: text, Line2D , collection, patch objects and so on. All artists are drawn to the canvas.

3.2.5 Input types

All the plotting functions expect a numpy array as input.

Note. Array like class like pandas data objects and numpy martix may not work as intended. It is best to convert these to numpy array object beforehand.

3.3 Pyplot

`matplotlib.pyplot` is a collection of command style functions used to make changes to figures. Function calls states are preserved across states.

Example 3.3.1. A Simple Plot

```
>>> import matplotlib.pyplot as plt
>>> plt.plot([1, 2, 3, 4])
>>> plt.show()
```

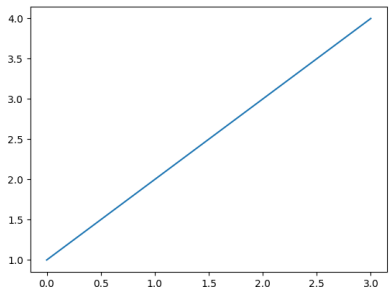

Figure 2: A simple plot

`pyplot.plot(*args, scalex=True, scaley=True, data=None, **kwargs)`
Plots y versus x as lines and/or markers.

Example 3.3.2. pyplot.plot() command

```
>>> plot(x, y) # plot x and y using default line style and
    color
>>> plot(x, y, 'bo') # plot x and y using blue circle
    markers
>>> plot(y)
>>> plot('xlabel', 'ylabel', data=obj) # plot labelled data
```

3.3.1 plot() formatting

There are a number of formatting options available, format strings or the `fmt` arg takes the following form: `[marker][line][color]`.

Note. Other combinations are supported but their interpretation is ambiguous.

Character	Description
.	point marker
,	pixel marker
o	circle marker
v	triangle_down marker
^	triangle_up marker
<	triangle_left marker
>	triangle_right marker
1	tri_down marker
2	tri_up marker
3	tri_left marker
4	tri_right marker
s	square marker
p	pentagon marker
*	star marker
h	hexagon1 marker
H	hexagon2 marker
+	plus marker
x	x marker
D	diamond marker
d	thin_diamond marker
|	vline marker
_	hline marker

Table 3: Markers

Character	Description
-	solid line style
–	dashed line style
-.	dash-dot line style
:	dotted line style

Table 4: Line Styles

Character	Description
b	blue
g	green
r	red
c	cyan
m	magenta
y	yellow
k	black
w	white

Table 5: Colors

Example 3.3.3. Plot format strings

```
'b'    # blue markers with default shape
'or'   # red circles
'-g'   # green solid line
'--'   # dashed line with default color
```

Example 3.3.4. Plot formatting in practise

```
>>> plt.plot([1, 2, 3, 4], [1, 4, 9, 16], '.--m')
>>> plt.axis([0, 6, 0, 20])
>>> plt.show()
```

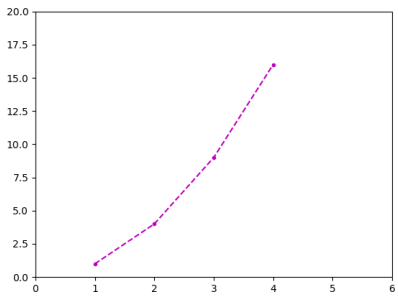

Figure 3: A plot with magenta markers

`pyplot.set_xlim(self, left=None, right=None, auto=False)`
Set the x axis limits

Example 3.3.5. set_xlim

```
>>> set_xlim(left, right)
```

Note. Passing limits in reverse order will flip the direction of the axis

```
>>> set_xlim(5000, 0)
```

`pyplot.set_ylim(self, bottom=None, top=None, auto=False)`
Set the y axis limits

Example 3.3.6. set_ylim

```
>>> set_ylim(bottom, top)
```

Note. Passing limits in reverse order will flip the direction of the axis

```
>>> set_ylim(5000, 0)
```

3.3.2 Plotting with keyword strings

Its often useful to have data in a format that allows for particular variables to be accessed. In Python dictionary objects are used, providing a key-value pair object. These can be used to plot categorical variables.

Example 3.3.7. Using the plot() data argument

```
>>> kwdata = {'a': np.arange(50), 'b': np.random.randint(0,
    50, 50)}
>>> plt.scatter('a', 'b', data=kwdata)
>>> plt.show()
```

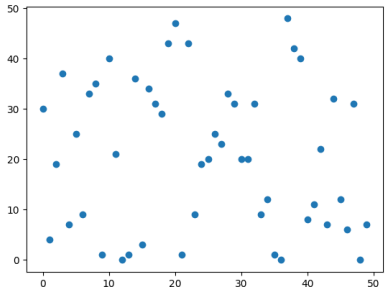

Figure 4: Plotting with keyword strings

`pyplot.scatter(x, y ,data=None, *args, **kwargs)`
Creates a scatter plot of y vs x with varying options

Note. plot function is faster for scatter plots when markers do not vary in size or color.

Note. fundamentally scatter plots work with 1D arrays

3.3.3 Plotting categorical variables

Example 3.3.8. Plotting categorical variables

```
>>> names = ['group_a', 'group_b', 'group_c']
>>> values = [1, 10, 100]
>>> plt.figure(figsize=(9, 3))
>>> plt.subplot(131)
>>> plt.bar(names, values)
>>> plt.subplot(132)
>>> plt.scatter(names, values)
>>> plt.subplot(133)
>>> plt.plot(names, values)
>>> plt.suptitle('Categorical Plotting')
>>> plt.show()
```

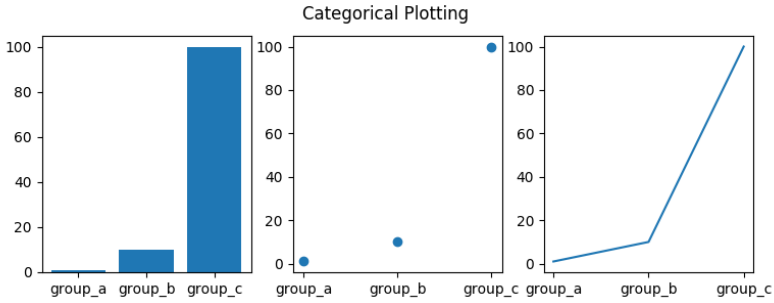

Figure 5: Plotting categorical variables

`pyplot.subplot(*args, **kwargs)`
Adds a subplot to the current figure. Returns an `Axes` object

Example 3.3.9. subplot()

```
plt.subplot(221)
ax1=plt.subplot(2, 2, 1) # equivalent
```

`pyplot.suptitle(t, **kwargs)`
Adds a centred title to a figure

Note. Takes other parameters such as: `horizontalalignment, verticalalignment , fontsize, fontweight` aliased by `ha, va, size, weight`

3.3.4 Line Properties

Lines under the `matplotlib.lines.Line2D` class have many properties that can be set.

3.3.5 Working with Text

`pyplot.title(label, fontdict=None, loc='center', pad=None, **kwargs)`
Set a title for the axes.

Note. The location or `loc` can be set to one of `center, left, right`

`pyplot.xlabel(xlabel, fontdict=None, labelpad=None, **kwargs)`
Set the label for the x-axis.

`pyplot.ylabel(ylabel, fontdict=None, labelpad=None)`
Set the label for the y-axis.

`pyplot.text(x, y, s, fontdict=None, **kwargs)`
Add text to the axes. Where `s` is the text, `x, y` are data coordinates.

Property	Value Type
alpha	float
animated	[True — False]
antialiased or aa	[True — False]
clip_box	a matplotlib.transform.Bbox instance
clip_on	[True — False]
clip_path	a Path instance and a Transform instance, a Patch
color or c	any matplotlib color
contains	the hit testing function
dash_capstyle	['butt' — 'round' — 'projecting']
dash_joinstyle	['miter' — 'round' — 'bevel']
dashes	sequence of on/off ink in points
data	(np.array xdata, np.array ydata)
figure	a matplotlib.figure.Figure instance
label	any string
linestyle or ls	['-' — '–' — '-.' — ':' — 'steps' — ...]
linewidth or lw	float value in points
marker	['+' — ',' — '.' — '1' — '2' — '3' — '4']
markeredgecolor or mec	any matplotlib color
markeredgewidth or mew	float value in points
markerfacecolor or mfc	any matplotlib color
markersize or ms	float
markevery	[None — integer — (startind, stride)]
picker	used in interactive line selection
pickradius	the line pick selection radius
solid_capstyle	['butt' — 'round' — 'projecting']
solid_joinstyle	['miter' — 'round' — 'bevel']
transform	a matplotlib.transforms.Transform instance
visible	[True — False]
xdata	np.array
ydata	np.array
zorder	any number

Table 6: Line2D Properties

Note. All the text commands returns a `matplotlib.text.Text` instance.

Note. `matplotlib` accepts TEX equation expressions in any text expression.

Example 3.3.10. TEX in `matplotlib`

```
plt.title(r'$\sigma_i=15$')
```

Property	Value Type
alpha	float
backgroundcolor	any matplotlib color
color	any matplotlib color
horizontalalignment or ha	['center' — 'right' — 'left']
label	any string
linespacing	float
name or fontname	string e.g., ['Sans' — 'Courier' — 'Helvetica' ...]
position	(x, y)
rotation	[angle in degrees — 'vertical' — 'horizontal']
size or fontsize	[size in points — relative size, e.g., 'smaller', 'x-large']
style or fontstyle	['normal' — 'italic' — 'oblique']
variant	['normal' — 'small-caps']
verticalalignment or va	['center' — 'top' — 'bottom' — 'baseline']
visible	bool
weight or fontweight	['normal' — 'bold' — 'heavy' — 'light' — 'ultrabold' — 'ultralight']
x	float
y	float

Table 7: Notable line properties

3.3.6 Annotating Text

A common use for text is to annotate some feature of a plot. The `annotate()` method provides help functionality to make annotations easy.

`pyplot.annotate(s, xy, *args, **kwargs)`
Annotates the point `xy` with given text `s`.

Note. `xy` is a tuple of floats `(x, y)`

Example 3.3.11. Annotations

```
t = np.arange(0.0, 5.0, 0.01)
s = np.cos(2*np.pi*t)
line, = plt.plot(t, s, lw=2)

plt.annotate('local max', xy=(2, 1), xytext=(3, 1.5),
    arrowprops=dict(facecolor='black', shrink=0.0))

plt.ylim(-2, 2)
plt.show()
```

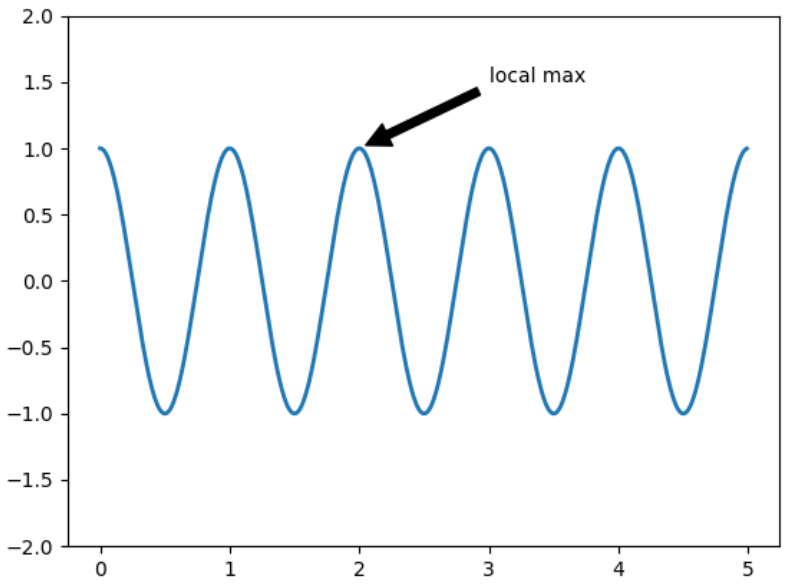

Figure 6: Annotating a plot

3.3.7 Non-linear Axes

Matplotlib supports non linear, logarithmic and logit scales for axes. As well as the ability to create custom scales for plots; more information can be found in the `matplotlib` docs.

`pyplot.yscale(value, **kwargs)`
Set the y-axis scale.

`pyplot.xscale(value, **kwargs)`
Set the x-axis scale.

Note. Values that can be applied:

- linear
- log
- symlog
- logit

Example 3.3.12. Non-linear axes

```
y = np.random.normal(loc=0.5, scale=0.4, size=1000)
y = y[(y > 0) & (y < 1)]
y.sort()
x = np.arange(len(y))

# plot with various axes scales
plt.figure()

# linear
plt.subplot(121)
plt.plot(x, y)
plt.yscale('linear')
plt.title('linear')
plt.grid(True)

# log
plt.subplot(122)
plt.plot(x, y)
plt.yscale('log')
plt.title('log')
plt.grid(True)

plt.show()
```

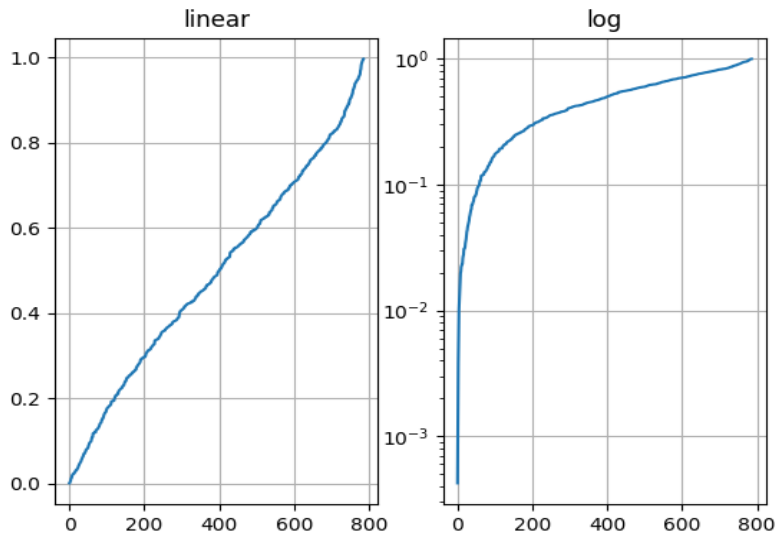

Figure 7: Changing the axis scale

3.4 Images

The matplotlib image module found under `matplotlib.image` and typically imported as `mpimg` supports basic image loading, rescaling and display operations.

3.4.1 Importing image data into Numpy arrays

Natively matplotlib only supports PNG images.

`image.imread(fname, format=None)`
Reads an image file into an numpy arra

Note. The returned shape of the array is either

- `(M, N)` for grey scale images
- `(M, N, 3)` for RGB images
- `(M, N, 4)` for RGBA images

Example 3.4.1. Importing images

```
>>> import matplotlib.image as mpimg
>>> img = mpimg.imread("./grasshopper.png")
>>> print(img)
[[0.40784314 0.40784314 0.40784314 ... 0.42745098 0.42745098
    0.42745098]
 ...
 [0.44313726 0.4509804  0.4509804  ... 0.44705883 0.44705883
    0.44313726]]
```

3.4.2 Plotting numpy arrays as images

`pyplot.imshow(Arr, data=None, **kwargs)`
Display an image.

Remark. Colouring schemes are set by the `cmap` parameter and include: `"hot"`, `"jet"`, `"nipy_spectral"`, `"gnuplot"`, `"ocean"`, as well as many others.

Example 3.4.2. Display images

```
>>> plt.imshow(img)
>>> plt.show()
```

Note. A pseudocolour or false colour has been applied as the image is imported as a grey scale image with no colour set.

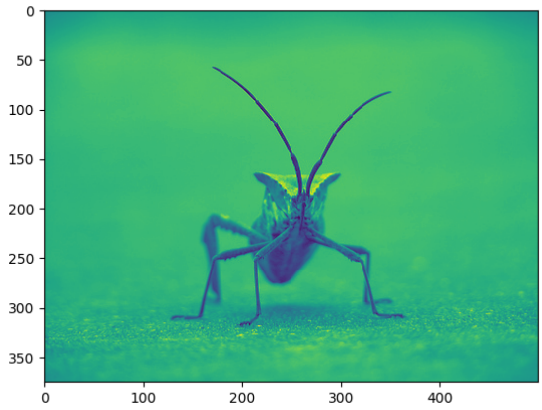

Figure 8: Grasshopper

3.4.3 Colour scale reference

`pyplot.colorbar(mappable=None, cax=None, ax=None, **kw)`
Adds a colorbar to a plot

Example 3.4.3. Adding a colorbar

```
>>> plt.imshow(img, cmap="gnuplot")
>>> plt.colorbar()
>>> plt.show()
```

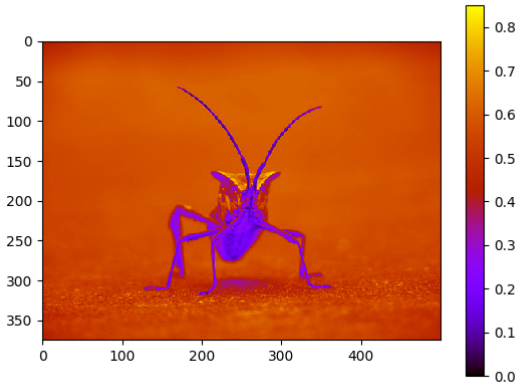

Figure 9: GNUplot & colorbar

3.4.4 Examining Image Data Ranges

A useful tool to understand images in grater depth are histograms, we can visualise the range of colours in an image and in turn enhance the contrast, find important colours and so forth.

`pyplot.hist(x, bins=None, range=None, density=None, weights=None, cumulative=False, log=False, color=None, stacked=False`
Plot a histogram

Note. There are boolean options like `cumulative`, `density`, `log` *etc.* used to change the histogram.

Example 3.4.4. Examining images

```
>>> plt.hist(img.ravel(), bins=256, range=(0.0, 1.0), fc="k"
    )
>>> plt.show()
```

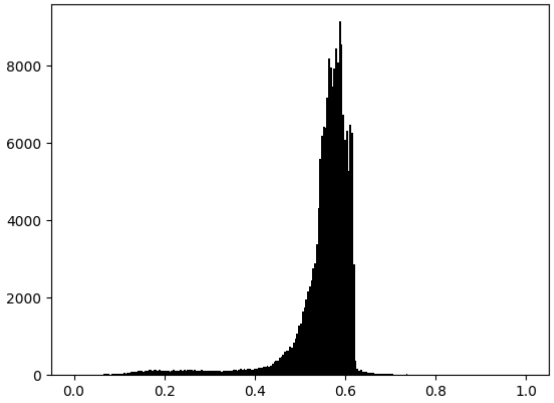

Figure 10: Grasshopper histogram

Remark. An interesting point is the peak at around 0.6. In order to look at the image within these limits more closely the `clim` arg can be used.

Example 3.4.5. Setting a clim

```
>>> plt.imshow(img, cmap="gnuplot", clim=(0, 0.6)))
```

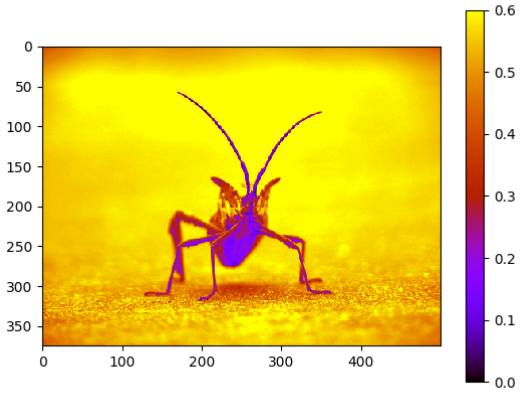

Figure 11: Setting a clim

3.5 Interactive Mode

Matplotlib has multiple backends that allow the user to work with images. When using OpenCV it is helpful to visually interpret whats happening to a plot as it happens, *ex.* facial recognition, object detection and so on.

`pyplot.ion()`
Turns interactive mode on

Example 3.5.1. Interactive mode

```
>>> plt.ion()
>>> plt.plot([1, 2])
```

Remark. A window should appear with a plot being displayed

The title can be set with:

```
>>> plt.title("An interactive title")
```

and the title can be updated

```
>>> plt.title("A new title")
```

For certain backends some functions require that the plot it explicitly drawn with:

```
>>> plt.draw()
```

4 OpenCV: Basics

Typically the OpenCV namespace known as `cv2` is imported as `cv`.

4.1 Images

When working with images in OpenCV it is important to note that:

1. the following file types are supported: `bmp, jpg, jp2, png, webp, pbm, pgm, ppm, pxm, ppm, st, pfm, tiff, tif, exr, hdr, pic`
2. color images are decoded as **B G R** instead of RBG
3. pixel length is limited to 2^{30}

4.1.1 Reading Images

`cv.imread(filename, flags=IMREAD_COLOR)`
Loads an image from a file.

Flags can be found under the `cv` namespace. Available flags include:

Enumerator	Description
IMREAD_UNCHANGED	return the loaded image as is
IMREAD_GRAYSCALE	always convert image to the single channel grayscale
IMREAD_COLOR	always convert image to the 3 channel BGR
IMREAD_ANYDEPTH	return 16-bit/32-bit image
IMREAD_ANYCOLOR	the image is read in any possible color format.
IMREAD_LOAD_GDAL	use the gdal driver for loading the image.
IMREAD_REDUCED_GRAYSCALE_2	convert image to single channel grayscale and reduce size by 1/2.
IMREAD_REDUCED_COLOR_2	convert image to 3 channel BGR and reduce size by 1/2.
IMREAD_REDUCED_GRAYSCALE_4	convert image to single channel grayscale and reduce size by 1/4.
IMREAD_REDUCED_COLOR_4	convert image to 3 channel BGR and reduce size by 1/4.
IMREAD_REDUCED_GRAYSCALE_8	convert image to single channel grayscale and reduce size by 1/8.
IMREAD_REDUCED_COLOR_8	convert image to the 3 channel BGR and reduce size by 1/8.
IMREAD_IGNORE_ORIENTATION	do not rotate the image

Example 4.1.1. Reading an image

```
import matplotlib.pyplot as plt
import numpy as np
import cv2 as cv


img = cv.imread("./grasshopper.png")

cv.imshow('grasshopper', img)
cv.waitKey(0)
cv.destroyAllWindows()
```

A window should appear with the specified image. In order to close the window and terminate the program any key can be pressed.

4.1.2 Useful functions for managing images

`cv.imshow(winname, img)`
Display an OpenGL 2D texture (img) in a specified window.

`cv.waitKey(delay=0)`
Waits for a key to be pressed.

Note. Delay of 0 means indefinitely; otherwise the time is in milliseconds.

`cv.destroyAllWindows()`
Destroys all the HighGUI windows.

`cv.destroyWindow(winname)`
Destroys specified window.

`cv.imwrite(filename, img, *)`
Saves an image to a specified file

Example 4.1.2. Writing an image

```
cv.imwrite('grasshopper.png',img)
```

4.1.3 Using Matplotlib

There is an important distinction in the way matplotlib and OpenCV read images. Matplotlib reads images as RGB whereas OpenCV reads images as BGR. Therefore displaying an image that has been read with OpenCV in matplotlib or vice versa will result in the wrong colour representation.

To amend this issue either use the same library or if you import an image with OpenCV you can reverse the numpy array.

Example 4.1.3. Displaying images with matplotlib

```
img = cv.imread('grasshopper.jpg')
img2 = img[:, :, ::-1]

# or

cv.cvtColor(img, cv.COLOR_BGR2RGB)
```

Example 4.1.4. Incorrectly displaying and image with matplotlib

```
>>> img = cv.imread("./duck.webp")
>>> img2 = img[:, :, ::-1]
>>> plt.subplot(121)
>>> plt.imshow(img)
>>> plt.subplot(122)
>>> plt.imshow(img2)
>>> plt.show()
```

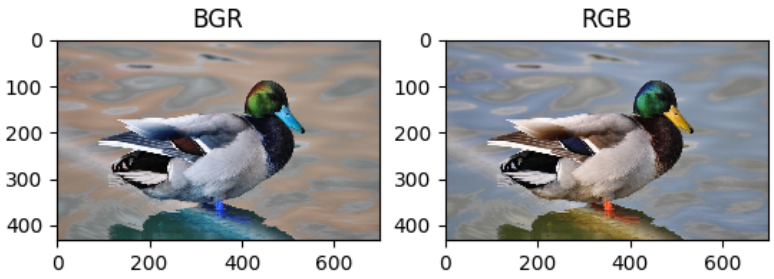

Figure 12: Inverted Colours

4.2 Video

4.2.1 Capturing Video

To capture video a `VideoCapture` object is contructed.

`cv.VideoCapture`
Class for capturing video from files, image sequences or cameras. The constuctor can either be passed a filename or an integer specifying which camera to use (starting from 0)

Example 4.2.1. Capturing Video

```
cap = cv.VideoCapture(0)
if not cap.isOpened():
    print("Cannot open camera")
    exit()

while True:
    res, frame = cap.read() # combines cap.grab() and cap.
    retrieve()

    if not res:
        print("Can't receive frame")
        break

    gray_frame = cv.cvtColor(frame, cv.COLOR_BGR2GRAY)
    cv.imshow('frame', gray_frame) # display

    if cv.waitKey(1) == ord('q'):
        break

cap.release() # close capture stream
cv.destroyAllWindows()
```

4.2.2 `VideoCapture` methods

`VideoCapture.isOpened()`
Returns true if the video capturing has been initalized already.

`VideoCapture.grab()`
Grabs the next frame in the video file or capturing device. Returns true in case of success.

`VideoCapture.retrieve()`
Decodes and returns the grabbed video frame.

`VideoCapture.read()`
Grabs, decodes and returns the next video frame. Returns false if no frame has been grabbed. Shorthand for calling both `grab` and `retrieve`.

`VideoCapture.release()`
Closes the video file or capturing device.

`VideoCapture.get(propId)`
Returns the specified VideoCapture property.

`VideoCapture.set(propId, value)`
Sets a property in the VideoCapture.

Enumerator	Description
CAP_PROP_POS_MSEC	Current position of the video file in milliseconds
CAP_PROP_POS_FRAMES	0-based index of the frame to be decoded/captured next
CAP_PROP_FRAME_WIDTH	Width of the frames in the video stream
CAP_PROP_FRAME_HEIGHT	Height of the frames in the video stream
CAP_PROP_FPS	Frame rate
CAP_PROP_FRAME_COUNT	Number of frames in the video file
CAP_PROP_FORMAT	Format of the Mat objects returned by VideoCapture::retrieve()
CAP_PROP_BRIGHTNESS	Brightness of the image
CAP_PROP_CONVERT_RGB	Boolean flags indicating whether images should be converted to RGB

Table 8: Commonly Used VideoCapture properties

4.2.3 Reading Video from file

Reading videos from file can uses the `VideoCapture` class.

Example 4.2.2. Reading Videos from File

```
cap = cv.VideoCapture("./video.mp4")

while True:
    res, frame = cap.read() # combines cap.grab() and cap.
    retrieve()

    if not res:
        break
```

```
    gray_frame = cv.cvtColor(frame, cv.COLOR_BGR2GRAY)
    cv.imshow('frame', gray_frame) # display

    if cv.waitKey(1) == ord('q'):
        break
...
```

Note. Videos do not play back seems faster than intended. This is because there is no limit to how many frames can be played a seconds. It may be interesting to wait for fps^{-1} seconds between frames.

4.2.4 Writing Video

The `VideoWriter` class is ueed to wite (save) videos.

`cv.VideoWriter(filename, fourcc, fps, size, isColor=True`
Initializes a video writer.

Def 4. FourCC (four-character code)
A sequence of four bytes used to identify data formats. In terms of OpenCV the 4 char code is used to compress the frames.

Some character codes include: `DIVX`, `XVID`, `X264`, `WMV1`, `MJPG`

Note. A FourCC code can be obtained via `cv.VideoWriter.fourcc(*'XVID')`

Example 4.2.3. Writing Video

```
cap = cv.VideoCapture(0)
fourcc = cv.VideoWriter.fourcc(*"XVID")
out = cv.VideoWriter('output.avi', fourcc, 20.0, (640,  480)
    )

while cap.isOpened():
    ret, frame = cap.read()
    if not ret:
        print("Can't receive frame")
        break

    frame = cv.flip(frame, 1) # vertically flips image

    out.write(frame)
    cv.imshow('frame', frame)
    if cv.waitKey(1) == ord('q'):
        break

cap.release()
out.release()
cv.destroyAllWindows()
```

`cv.flip(frame, flipCode)`
Flips a 2D array around horizontal, vertical or both axes.

flipCode < 0 flips both axes
flipCode $= 0$ flips horizontally
flipCode > 0 flip vertically

4.2.5 `VideoWriter` methods

`VideoWriter.open()`
Initializes or reinitializes the video writer. Returns `true` if successful.

`VideoWriter.isOpened()`
Returns `true` if video writer successfully Initialized.

`VideoWriter.release()`
Closes the video writer.

`VideoWriter.write(img)`
Writes the next video frame.

`VideoWriter.getBackendName()`
Returns used backend API.

`VideoWriter.set(propId, val)`
Sets a `VideoWriter` property

`VideoWriter.get()`
Returns the specified `VideoWriter` property.

Enumerator	Description
VIDEOWRITER_QUALITY	Current quality (0..100%) of the encoded videostream. Can be adjusted dynamically in some codecs
VIDEOWRITER_FRAMEBYTES	(Read-only): Size of just encoded video frame. Note that the encoding order may be different from representation order
VIDEOWRITER_NSTRIPES	Number of stripes for parallel encoding. -1 for auto detection

Table 9: `VideoWriter` properties

4.3 Drawing Functions

All drawing classes, functions and enumerations in OpenCV are found under `Imgproc_draw` and are imported into the main `cv2` namespace.

4.3.1 Lines

`cv.line(img, pt1, pt2, col, thickness, shift=0`
Draws a line connecting two points `pt1, pt2`

Example 4.3.1. Drawing Lines

```
img = np.full((512, 512, 3), 251.0)
cv.line(img, (0, 0), (511, 511), (255, 0, 0), 2)
cv.line(img, (511, 0), (0, 511), (0, 0, 255), 2)
```

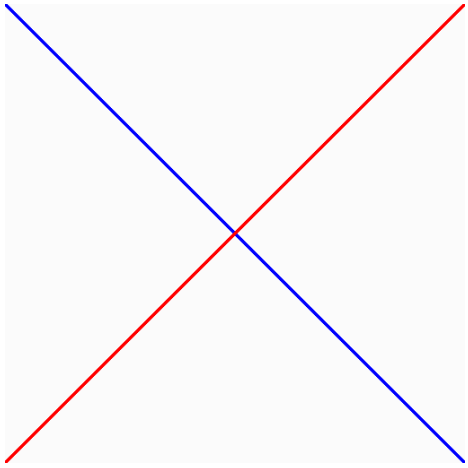

Figure 13: Drawing Lines

Remark. When plottings lines or more generally when dealing with images, the **top left** pixel is point $(0, 0)$.

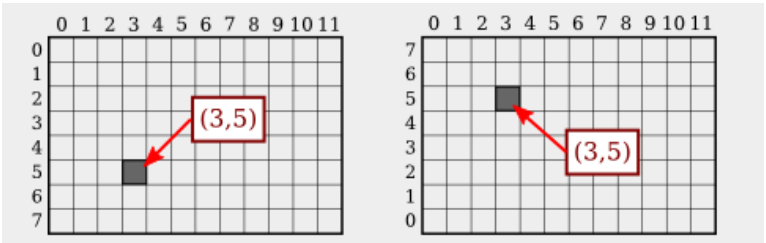

Figure 14: Computer Graphics

4.3.2 Rectangles

`cv.rectangle(img, pt1, pt2, col, thickness=1, lineType=LINE_0, shift=0)`
Draws a rectangle, negative thickness means the rectangle will be filled.

Example 4.3.2. Drawing Rectangles

```
img = np.full((512, 512, 3), 251.0)
cv.rectangle(img, (63, 63), (447, 447), (0, 255, 0), -1)
```

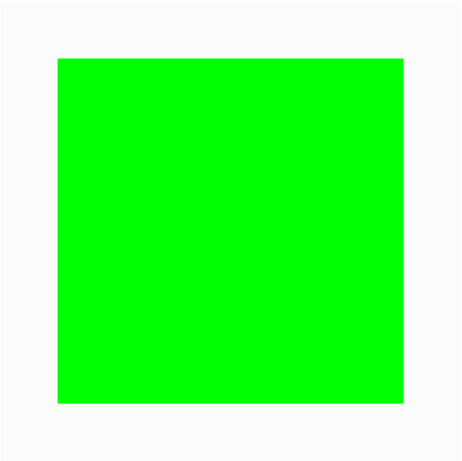

Figure 15: Drawing Rectangles

4.3.3 Circles

`cv.circle(ing, center, rad, col, thickness=1, lineType=LINE_0, shift=0)`
Draws a circle with a given center and radius

Example 4.3.3. Drawing Circles

```
cv.circle(img, (255, 255), 128, (0, 255, 100), 2)
```

4.3.4 Ellipses

`cv.ellipse(ing, center, axes, rotation, startAngle, endAngle, col, thickness =1, lineType=LINE_0, shift=0)`
Draws an ellipse, can be filled by changing the line type.

Example 4.3.4. Drawing Ellipses

```
cv.ellipse(img, (255, 255),
    (100, 200), # axes
    0, 90, 320, # rotation, start, end
    (0, 0, 255), -1)
```

Figure 16: Drawing Ellipses

4.3.5 Polygons

`cv.polylines(img, *pts, isClosed, col, thickness,... )`
Draws several polygoonal curves.

Example 4.3.5. Drawing Polygons

```
points = np.array([(63,63), (447, 127), (63, 195), (447,
    255), (63, 319), (447, 383)], np.int32)
points = points.reshape((-1, 1, 2))
cv.polylines(img, [points], True, 255, 1)
```

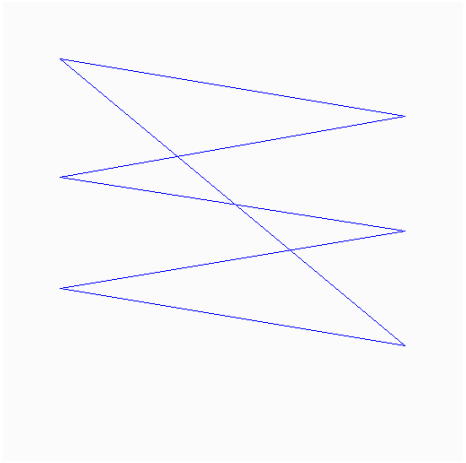

Figure 17: Drawing Polygons

4.3.6 Text

`cv.putText(img, text, org, fontFace, fontScale, col, thickness=1, lineType =LINE_0, bottomLeftOrigin=False)`
Draws a text string. Symbols that cannot be rendered are replaced by question marks.

Example 4.3.6. Drawing Text

```
font = cv.FONT_HERSHEY_SIMPLEX
cv.putText(img, "oCV", (200, 300), font, 2, (0, 0, 0), 2,cv.
    LINE_AA)
```

Figure 18: Drawing Text

4.3.7 Miscellaneous

`cv.arrowLine(img, pt1, pt2, col, thickness=1, tipLength=0.1)`
Draws an arrow segments pointing from pt1 to pt2.

`cv.drawMarker(img, pt, col, markerType=MARKER_CROSS, markerSize=20)`
Draws a marker at a given point.

`cv.drawContours(img, *contours, countourIdx, col, thickness=1, hierarchy=[], maxLevel, offset)`
Draws contours outlines or filled contours.

4.4 GUIs

Graphical user interfaces or GUIs are constructed with the `highgui` module. The module is responsible for: creating and managing windows, creating GUI elements mainly trackbars and dealing with window interactions such as mouse events.

4.4.1 Creating Windows

`cv.namedWindow(winname, flags=WINDOW_AUTOSIZE)`
Creates a window that can be used as a place holder for images and track bars.

Note. If a window with the same name exists the function does nothing.

Enumerator	Description
WINDOW_NORMAL	window can be resized
WINDOW_ AUTOSIZE	window cannot be resized, size determined by the image being displayed
WINDOW_OPENGL	window with opengl support
WINDOW_FULLSCREEN	fullscreen window
WINDOW_FREERATIO	the image expends as much as it can (no ratio constraint)
WINDOW_KEEPRATIO	the ratio of the image is respected
WINDOW_GUI_EXPANDED	window with status bar and tool bar
WINDOW_GUI_NORMAL	old fashioned way, not recommended

Table 10: Window Flags

`cv.resizeWindow(winname, width, height)`
Resizes window to the specified size.

`cv.moveWindow(winname, x, y)`
Moves window to the specified position.

`cv.setWindowTitle(winname, title)`
Updates window title.

`cv.setWindowProperty(winname, prop_id, prop_value)`
Changes parameters of a window dynamically.

`cv.getWindowProperty(winname, prop_id)`
Provides parameters of a window.

4.4.2 Mouse Events

OpenCV can be used to listen for mouse events. In order for OpenCV to interact with mouse data, the `setMouseCallback` function and a callback function are needed.

Def 5. Callback Function
A function that is passed into another function as an argument and is invoked inside the outer function to complete some kind of action.

`cv.setMouseCallback(winame, onMouse, userdata=0)`
Sets mouse handler for the specified window. `onMouse` is a callback function for mouse events.

`cv.MouseCallback(event, x, y, flags, userdata)`
Callback function for mouse events.

Available mouse events include:

Enumerator	Description
EVENT_MOUSEMOVE	indicates that the mouse pointer has moved over the window
EVENT_LBUTTONDOWN	indicates that the left mouse button is pressed
EVENT_RBUTTONDOWN	indicates that the right mouse button is pressed
EVENT_MBUTTONDOWN	indicates that the middle mouse button is pressed
EVENT_LBUTTONUP	indicates that left mouse button is released
EVENT_RBUTTONUP	indicates that right mouse button is released
EVENT_MBUTTONUP	indicates that middle mouse button is released
EVENT_LBUTTONDBLCLK	indicates that left mouse button is double clicked
EVENT_RBUTTONDBLCLK	indicates that right mouse button is double clicked
EVENT_MBUTTONDBLCLK	indicates that middle mouse button is double clicked
EVENT_MOUSEWHEEL	positive and negative values mean forward and backward scrolling, respectively
EVENT_MOUSEHWHEEL	positive and negative values mean right and left scrolling, respectively

Table 11: Mouse Events

Example 4.4.1. Mouse Events

```
img = np.full((512, 512, 3), 251.0)
cv.namedWindow('image')

def draw_circle(event, x, y, flags, userdata):
    ''' Callback function that draws a circle '''
    if event == cv.EVENT_LBUTTONDBLCLK:
        cv.circle(img, (x, y), 100, (255, 0, 0), 1)

cv.setMouseCallback('image', draw_circle)

while(True):
    cv.imshow('image', img)
    if cv.waitKey(20) & 0xFF == 27: # esc key
        break

cv.destroyAllWindows()
```

Note. When there is a change in the state of the mouse, the callback if fired containing information such as `x, y` position, the `event` and additional `flags`.

Event flags can be used to narrow down events even further. Available flags include:

Enumerator	Description
EVENT_FLAG_LBUTTON	indicates that the left mouse button is down
EVENT_FLAG_RBUTTON	indicates that the right mouse button is down
EVENT_FLAG_MBUTTON	indicates that the middle mouse button is down
EVENT_FLAG_CTRLKEY	indicates that CTRL Key is pressed
EVENT_FLAG_SHIFTKEY	indicates that SHIFT Key is pressed
EVENT_FLAG_ALTKEY	indicates that ALT Key is pressed

Table 12: Event Flags

4.4.3 Trackbars

Trackbars are an extremely useful tool for quickly interacting with a range of values. They can be used to change image properties such as brightness, function parameters such as drawing function colours etc. Trackbars are created using the `createTrackbar` function.

`cv.createTrackbar(trackbarname, winname, val, count, onChange=0, data=0)`
Creates a trackbar and attaches it to the specified window. `value` is the initial value of the trackbar.

`cv.TrackbarCallback(pos, userdata)`
Callback function for Trackbar.

Much like Mouse Events trackbars also require a callback.

Example 4.4.2. Using Trackbars

```
...
def draw_circle(event, x, y, flags, params):
    ''' Callback function that draws a circle '''
    if event == cv.EVENT_LBUTTONDBLCLK:
        red = cv.getTrackbarPos("R", "image")
        thickness = cv.getTrackbarPos("T", "image")
        cv.circle(img, (x, y), 100, (0, 0, red), thickness)

def empty_func(x): pass

cv.createTrackbar("R", "image", 122, 255, empty_func)
cv.createTrackbar("T", "image", 5, 50, empty_func)
cv.setMouseCallback('image', draw_circle)
...
```

Some additional function for interacting with trackbars:

`cv.getTrackbarPos(trackbarname, winname)`
Returns the position of the trackbar.

`cv.setTrackbarPos(trackbarname, winame, pos)`
Set the trackbar position.

`cv.setTrackbarMax(trackbarname, winname, maxval)`
Set the trackbar maximum position.

`cv.setTrackbarMin(trackbarname, winname, minval)`
Set the trackbar minimum position.

5 OpenCV: Image Processing

Image processing plays a fundamental role in OpenCV, before information can be extracted from an image about say the location of an object, the presence of a face etc. An image must be manipulated in such way that removes unnecessary information or enhances existing information. Therefore, images need to be combined, noise removed, motion de-blurred and so on. The following section explores the available options for processing images in OpenCV.

5.1 Colour Space

Def 6. Colour Space

An abstract mathematical model that describes colours. They are also referred to as colour models.

5.1.1 HSV Colour Space

HSV stands for Hue, Saturation, Visibility. It is a commonly used colour space and provides a more natural way or thinking about colour. Rather than RGB / BGR which considers colour as a combinations of the three primary colours.

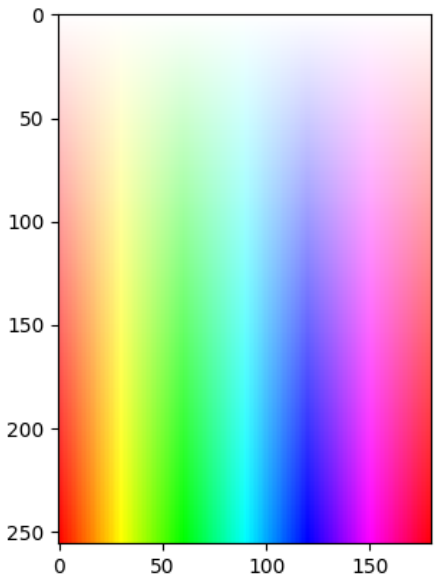

Figure 19: HSV Colour Space

Note. The visibility is constant in this case, however it can be thought of as the amount of darkness / greyness in the image. A low visibility means that the colour is darker.

The range for each value in OpenCV is:

- H $[0, 179]$
- S $[0, 255]$
- V $[0, 255]$

5.1.2 Changing Colour Space

The `cvtColor` function is used to convert between colour spaces.

`cvtColor(img, dst, code`
Converts an image from one colour space to another.

Example 5.1.1. Converting BGR TO HSV

```
duck = cv.imread("./duck.png")
hsv_duck = cv.cvtColor(duck, cv.COLOR_BGR2HSV)
cv.imshow("image", hsv_duck)
```

Note. The image has been converted to HSV however is being displayed in BGR; appearing unusual.

Conversion	Description
COLOR_BGR2GRAY	BGR and grayscale
COLOR_BGR2BGR565	BGR and BGR565 (16-bit images)
COLOR_GRAY2BGR565	grayscale to BGR565 (16-bit images)
COLOR_BGR2XYZ	BGR to CIE XYZ
COLOR_BGR2YCrCb	BGR to luma-chroma (aka YCC)
COLOR_BGR2HSV	BGR to HSV (hue saturation value)
COLOR_BGR2Lab	BGR to CIE Lab
COLOR_BGR2Luv	BGR to CIE Luv
COLOR_BGR2HLS	BGR to HLS (hue lightness saturation)

Table 13: Colour Space Conversions

Remark. There are over 150 colour conversions available.

```
>>> [print(i) for i in dir(cv) if i.startswith('COLOR_')]
```

Prints the available conversions

5.2 Image Arithmetic

Image arithmetic applies the standard arithmetic operations or logical operators to two or more images. Operators are applied on a pixel by pixel basis, therefore the value of the pixel is dependant only on the corresponding image pixel value. Typically logical operators are applied on binary (black and white) images, in order to combine or remove elements.

Throughout the example the following image will be used

```
ints = np.array([(127, 63), (63, 191), (191, 191)],np.int32)
points.reshape((-1, 1, 2))

triangle = cv.fillPoly(np.zeros((255, 255), np.uint8), [
    points], 170)
rectangle = cv.rectangle(np.zeros((255, 255), np.uint8),
    (63, 63), (191, 191), 80, -1)
```

5.2.1 Addition

Addition is performed element wise

$$Q(i,j) = P_1(i,j) + P_2(i,j) \tag{1}$$

Alternatively pixel value can be added to a constant C:

$$Q(i,j) = P_1(i,j) + C$$

Example 5.2.1. Addition

```
rectangle + triangle
rectangle + 100
```

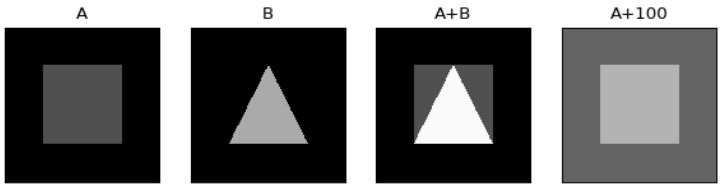

Figure 20: Addition

5.2.2 Subtraction

Pixel subtraction takes two images and produces an output image whose pixels values are those of the first image minus the corresponding pixel values from the second image. It is also possible to subtract a constant from the image.

$$Q(i,j) = P_1(i,j) - P_2(i,j) \tag{2}$$

Alternatively a constant C can be subtracted:

$$Q(i,j) = P_1(i,j) - C$$

Example 5.2.2. Subtraction

```
# Rectangle an Traingle pixel values changed to 200
rectangle - triangle
rectangle - 200
```

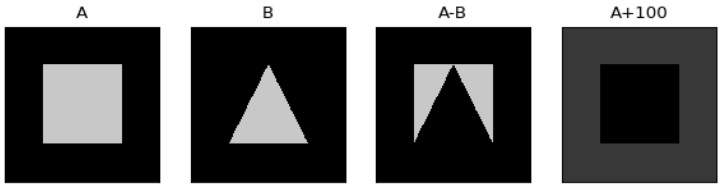

Figure 21: Subtraction

Remark. Implementations of the operator vary as to what it does if the output pixels are negative. For instance a pixel value of -30 appears as 226

Image subtraction can be useful for removing uneven lighting in an image. For instance in microscopy; an image may be taken when the lighting is poor of some item, a similar image can be taken without the item but with the same uneven light source, subtracting one image from the other would result in a darker but evenly lit image.

5.2.3 Multiplication

Similar to other arithmetic operations pixel multiplication can consist of multiplying one pixel value by another, or multiplying by a constant.

$$Q(i,j) = P_1(i,j) \times P_2(i,j) \tag{3}$$

$$Q(i,j) = P_1(i,j) \times C$$

Example 5.2.3. Multiplication

```
rectangle * 0.5
triangle  * 1.5
```

Figure 22: Multiplication

Remark. Pixel multiplication can come in useful when trying to increase / decrease the brightness of an image. For instance a scaling factor $C > 1$ would increase the brightness, and a factor $C < 1$ would decrease the brightness.

Note. Using pixel multiplication often gives better results than just adding the same value to each pixel as the relative contrast of the image is preserved.

5.2.4 Division

Image division take two images as input and produces a third image whose pixel values are the pixel values of the first image divided by the corresponding values in the second image.

$$Q(i,j) = \frac{P_1(i,j)}{P_2(i,j)} \tag{4}$$

$$Q(i,j) = \frac{P_1(i,j)}{C}$$

One of the most important uses of division is in change detection. This is similar to subtraction but instead of giving the absolute change for each pixel from one to the next, it gives the ratio between corresponding pixel values.

Example 5.2.4. Division

```
div1 = cv.imread("./images/division1.png", 0)
div2 = cv.imread("./images/division2.png", 0)
(div1 / div2) * 100) # scaled for visibility
```

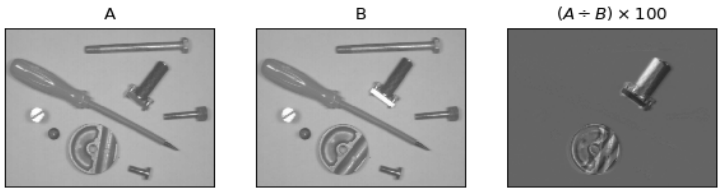

Figure 23: Division

5.2.5 Logical AND

The AND operator takes in two images and determines the values of the third images pixels based on the following truth table:

A	B	Q
0	0	0
0	1	0
1	0	0
1	1	1

Mathematically the operator is defined as:

$$Q(i,j) = P_1(i,j) \wedge P_2(i,j) \tag{5}$$

`np.logical_and(x1, x2)`
Computes the truth value of x1 AND x2 element-wise.

Example 5.2.5. AND

```
circle_r = cv.circle(np.zeros((255, 255), np.uint8), (159,
    127), 64, 255, -1)
circle_l = cv.circle(np.zeros((255, 255), np.uint8), (93,
    127), 64, 255, -1)

np.logical_and(circle_r, circle_l)
```

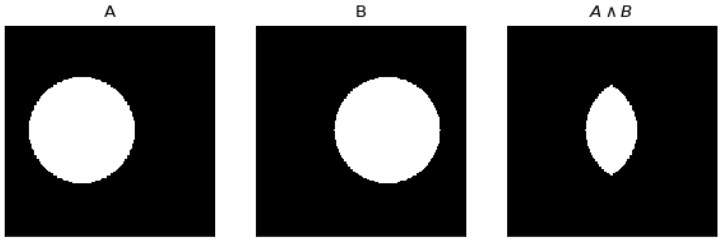

Figure 24: AND

5.2.6 Logical OR

The OR operator takes in two images and determines the values of the third images pixels based on the following truth table:

A	B	Q
0	0	0
0	1	1
1	0	1
1	1	1

Mathematically it is defined as:

$$Q(i,j) = P_1(i,j) \vee P_2(i,j) \tag{6}$$

`np.logical_or(x1, x2)`
Computes the truth value of x1 OR x2 element-wise.

Example 5.2.6. OR

```
np.logical_or(circle_r, circle_l)
```

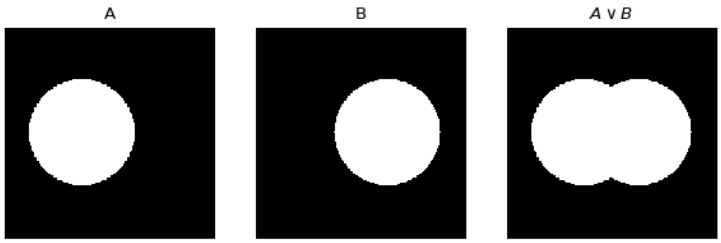

Figure 25: OR

Remark. The OR operator can be used to computed the union of images, hence highlighting all pixels that are in the first or second image.

5.2.7 Logical XOR

The exclusive or (XOR) operator takes in two images and determines the values of the third images pixels based on the following truth table:

A	B	Q
0	0	0
0	1	1
1	0	1
1	1	0

Mathematically it is defined as:

$$Q(i,j) = P_1(i,j) \veebar P_2(i,j) \tag{7}$$

`np.logical_xor(x1, x2)`
Computes the truth value of x1 XOR x2 element-wise.

Example 5.2.7. XOR

```
np.logical_xor(circle_r, circle_l)
```

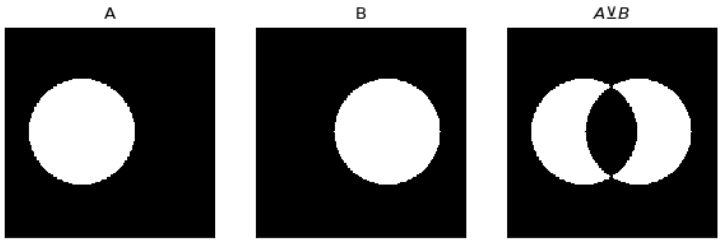

Figure 26: XOR

Remark. XOR can be used to detect changes in images, since pixels which did not change in an image are removed.

5.2.8 Logical NOT

The NOT operator takes in an image and determines the values of the output image pixels based on the following truth table:

A	Q
1	0
0	1

Mathematically it is defined as:

$$Q(i,j) = \neg P_1(i,j) \tag{8}$$

`np.logical_not(x)`
Computes the truth value of NOT x

Example 5.2.8. NOT

```
np.logical_not(circle)
```

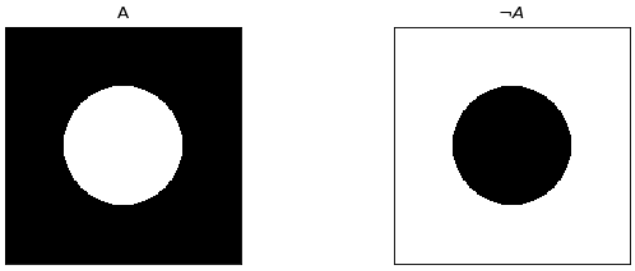

Figure 27: NOT

5.3 Image Thresholding

5.3.1 Simple Thresholding

Thresholding provides a straightforward way to perform image segmentation. For instance in many applications it is necessary to separate objects of interest, from the background regions of an image. It is also useful to see what regions of an image consist of pixels whose values are in a specific range.

The simplest way to apply thresholding is to set every pixel above a certain limit T to the maximum value M and those below to 0.

$$s = \begin{cases} M, & \text{if } r > T. \\ 0, & otherwise. \end{cases} \tag{9}$$

`cv.threshold(src, thresh, maxValue, type)`
Applies a fixed-level threshold to each array element.

Example 5.3.1. Basic Thresholding

```
thresh = 160
img = cv.imread("./range.png")
arr, res1 = cv.threshold(img, thresh, 255, type=cv.
    THRESH_BINARY)
arr, res2 = cv.threshold(img, thresh, 255, type=cv.
    THRESH_BINARY_INV)
arr, res3 = cv.threshold(img, thresh, 255, type=cv.
    THRESH_TRUNC)
arr, res4 = cv.threshold(img, thresh, 255, type=cv.
    THRESH_TOZERO)
arr, res5 = cv.threshold(img, thresh, 255, type=cv.
    THRESH_TOZERO_INV)
```

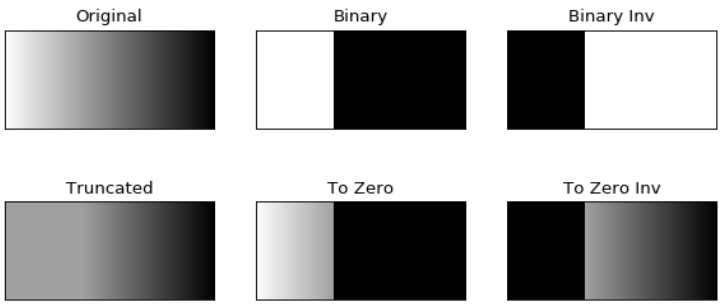

Figure 28: Basic Thresholding

The available thresholds in OpenCV are:

Enumerator	Description
THRESH_BINARY	$s = \begin{cases} M, & \text{if } r > T. \\ 0, & otherwise. \end{cases}$
THRESH_BINARY_INV	$s = \begin{cases} 0, & \text{if } r > T. \\ M, & otherwise. \end{cases}$
THRESH_TRUNC	$s = \begin{cases} M, & \text{if } r > T. \\ r, & otherwise. \end{cases}$
THRESH_TOZERO	$s = \begin{cases} r, & \text{if } r > T. \\ 0, & otherwise. \end{cases}$
THRESH_TOZERO_INV	$s = \begin{cases} 0, & \text{if } r > T. \\ r, & otherwise. \end{cases}$
THRESH_OTSU	Use Otsu algorithm to choose the optimal threshold value.
THRESH_TRIANGLE	Use Triangle algorithm to choose the optimal threshold value.

Table 14: Structuring Elements

5.3.2 Adaptive Thresholding

Adaptive thresholding takes a grayscale image and outputs a binary image representing the segmentation. For each pixel a threshold is calculated. The threshold is then used to determine if the value of the pixel is set to the maximum or 0.

There are two methods available in OpenCV: mean thresholding and Gaussian thresholding. Both methods examine a local neighbourhood of pixels in order to determine the optimal threshold value.

Remark. The size of the neighbourhood has be to sufficiently large enough to cover both foreground and background pixels.

`cv.adaptiveThreshold(src, maxVal, adaptiveMethod, thresholdType, size, C)`
Applies a threshold to an array, the constant C is subtracted from the mean / weighted mean.

Example 5.3.2. Adaptive Thresholding

```
img = cv.imread("./sudoku.png", 0)
img = cv.medianBlur(img, 5)

res_mean = cv.adaptiveThreshold(img, 255, cv.
    ADAPTIVE_THRESH_MEAN_C, cv.THRESH_BINARY, 11, 2)
res_gaus = cv.adaptiveThreshold(img, 255, cv.
    ADAPTIVE_THRESH_GAUSSIAN_C, cv.THRESH_BINARY, 11, 2)
```

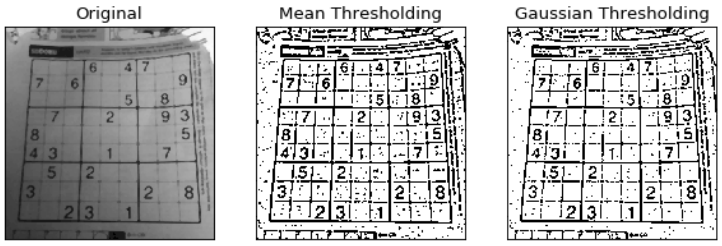

Figure 29: Adaptive Thresholding

5.3.3 Otsu's Binarization

Otsu binarization is used to automatically determine an optimal threshold t based on the distribution of pixel intensity. For instance consider a bimodal image (an image with two intensity peaks), a good threshold would exist somewhere between the two peaks. Otsu's method determines the optimal value by separating the two regions such that the weighted within-class variance is minimized, *i.e.*:

$$\sigma_w^2(t) = q_1(t)\sigma_1^2(t) + q_2(t)\sigma_2^2(t) \tag{10}$$

Where:

$$q_1(t) = \sum_{i=1}^{t} P(i) \qquad q_2(t) = \sum_{i=t+1}^{I} P(i)$$

$$\mu_1(t) = \sum_{i=1}^{t} \frac{iP(i)}{q_1(t)} \qquad \mu_2(t) = \sum_{i=t+1}^{I} \frac{iP(i)}{q_2(t)}$$

$$\sigma_1^2(t) = \sum_{i=1}^{t} [i - \mu_1(t)]^2 \frac{P(i)}{q_1(t)} \qquad \sigma_2^2(t) = \sum_{i=t+1}^{I} [i - \mu_2(t)]^2 \frac{P(i)}{q_2(t)}$$

In order to apply Otsu Binarization in OpenCV the `threshold` function is used.

Example 5.3.3. Otsu Binarization

```
img = cv.imread("./noisy.png", 0)
img = cv.GaussianBlur(img, (5, 5), 0)

thresh, bin_img = cv.threshold(img, 0, 255, cv.THRESH_BINARY
    + cv.THRESH_OTSU)
```

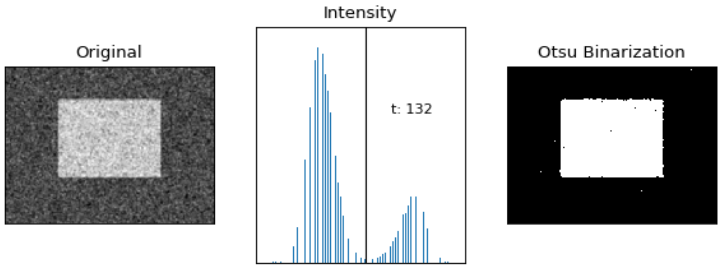

Figure 30: Otsu Binarization applied to a noisy image

Remark. A line has been added to the histogram to show that the optimal threshold $t = 132$.

5.4 Image Smoothing

5.4.1 A note on Convolution

Def 7. Convolution

A mathematical operation denoted by $*$ applied on two functions f, g that expresses how the shape of f is modified by g.

For continuous functions it is defined as:

$$(f * g)(t) \triangleq \int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau \tag{11}$$

For discrete functions it is defined as:

$$(f * g)[n] = \sum_{m=-\infty}^{\infty} f[m]g[n-m] \tag{12}$$

Some properties of convolution include:

Commutativity $f * g = g * f$
Associativity $f * (g * h) = (f * g) * h$
Distributivity $f * (g + h) = (f * g) + (f * h)$
Multiplicative Identity $f * \delta = f$

In image processing convolution provides a means of combining two arrays generally of different sizes but of the same dimensionality; to produce a third array.

Example 5.4.1. Input and Kernel Arrays

Consider and Image $\boldsymbol{I}$

$$\begin{bmatrix} I_{1,1} & I_{1,2} & \dots & I_{1,n-1} & I_{1,n} \\ I_{2,1} & I_{2,2} & \dots & I_{2,n-1} & I_{2,n} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ I_{m-1,1} & I_{m-1,2} & \dots & I_{m-1,n-1} & I_{m-1,n} \\ I_{m,1} & I_{m,2} & \dots & I_{m,n-1} & I_{m,n} \end{bmatrix}$$

And Kernel $\boldsymbol{K}$

$$\begin{bmatrix} K_{1,1} & K_{1,2} & K_{1,3} \\ K_{2,1} & K_{2,2} & K_{2,3} \end{bmatrix}$$

The convolution is performed by sliding the Kernel $\boldsymbol{K}$ over the Image $\boldsymbol{I}$, such that all the positions fit $\boldsymbol{K}$ entirely. Each position corresponds to a single output value for example:

$$O_{5,7} = I_{5,7}K_{1,1} + I_{5,8}K_{1,2} + I_{5,9}K_{1,3} + I_{6,7}K_{2,1} + I_{6,8}K_{2,2} + I_{6,9}K_{2,3}$$

Or more generally:

$$O(i,j) = \sum_{k=1}^{m} \sum_{l=1}^{n} I(i+k-1, j+l-1) K(k,l) \tag{13}$$

Note. Many implementations of convolution relax the constraint that the kernel can only be moved into places where it fits entirely; producing a larger output image due to the borders.

Remark. Convolution can be used to implement many different operators, in particular: spatial filters, feature detectors, image smoothing, edge detectors and so forth.

5.4.2 Image Smoothing Theory

Smoothing is a simple and frequently used image processing operation. It is commonly used to reduce noise and produce a less pixelated image. The most basic smoothing operation is linear smoothing. Smoothing operations are typically defined as:

$$g(i,j) = \sum_{k,l} f(i+k, j+l) h(k,l) \tag{14}$$

Where:

$g(i,j)$ pixel value

$f(i+k, j+l)$ input pixel values

$h(k,l)$ kernel

Remark. The kernel is just a matrix of coefficients.

5.4.3 Normalized Box or Mean Filter

Normalized box or mean filtering is a simple filter, where each output pixel is the mean of its kernel neighbours, each neighbour's value contributes with equal weight to the value of the output pixel O.

$$K = \frac{1}{K_{width} \cdot K_{height}} \begin{bmatrix} 1 & \dots & 1 \\ \vdots & \ddots & \vdots \\ 1 & \dots & 1 \end{bmatrix} \tag{15}$$

Example 5.4.2. 3×3 Mean Filter
Consider the Kernel $\boldsymbol{K}$

$$\frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

And a region of an Image $\boldsymbol{I}$

·	**1**	**2**	**3**
1	123	125	126
2	122	124	126
3	118	120	150

The value of the pixel at $2, 2$ $\boldsymbol{p}$ is given by:

$$\begin{aligned} O_{2,2} &= \frac{1}{9}(123 + 122 + 118 + 125 + 124 + 120 + 126 + 126 + 150) \\ &= 126 \end{aligned}$$

Remark. A Mean filter provides a simple way to smooth an image, however this also provides the non-desirable effect of smoothing edges.

5.4.4 Median Filtering

Median filtering considers each pixel of the image in turn and looks at its nearby neighbours to determine whether or not its representative of the surroundings.

Example 5.4.3. Median Filtering

·	·	·	·	·	·	·	
·	123	125	126	130	140	·	
·	122	**124**	**126**	**127**	135	·	
·	118	**120**	**150**	**125**	134	·	→ median value: 124
·	119	**115**	**119**	**123**	133	·	
·	111	116	110	120	130	·	
·	·	·	·	·	·	·	

(16)

The central point 150 is unrepresentative of the surrounding pixels: $115, 119, 120, 123, 124, 125, 126, 127, 150$ and so it is replaced with a value of 124.

Note. In the above example a 3×3 neighbourhood is used; larger neighbourhoods will produce more severe smoothing.

Remark. In practise median filtering does a better job than mean (box) filtering in terms of preserving detail in an image.

Remark. Median filters are also much better at preserving sharp edges as they do not generate entirely new pixel values but use values already present in the image.

5.4.5 Gaussian Filter

Gaussian Filtering or Gaussian Blur Filtering is amongst one of the more useful filters, however is more computationally intense (and therefore slower). Gaussian filtering provides greater weight to the pixels closest to the output pixel, and decreases the weight as the spatial distance of the pixels increases.

$$G_\sigma(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{17}$$

Which produces the following distribution

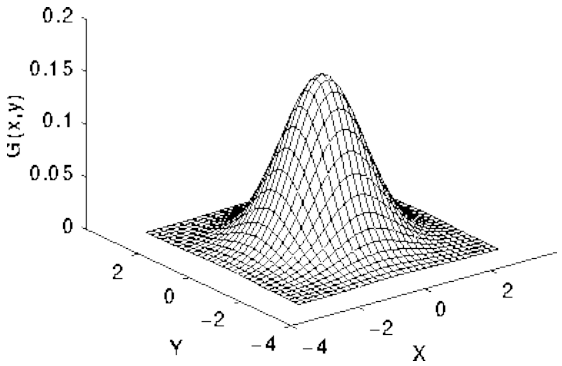

Figure 31: 2D Gaussian Distribution

Remark. The idea behind Gaussian smoothing is to use the 2D distribution as a point-spread function, i.e. produce a discrete approximation to the Gaussian function s.t. we can perform the convolution. In reality the 2D Gaussian Distribution is infinitely large; which would produce a kernel that is infinitely large as such an approximation is needed.

The 2D Gaussian Kernel is given by:

$$G_\sigma(x) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2}{2\sigma^2}} \tag{18}$$

The Gaussian Blur Filtered Image is defined as:

$$GB[I]_p = \sum_{q\in\mathcal{S}} G_\sigma(|\boldsymbol{p}-\boldsymbol{q}|) I_q \tag{19}$$

Example 5.4.4. Gaussian Kernel $\sigma = 1.0$,

$$G_1 = \frac{1}{273}\begin{bmatrix} 1 & 4 & 7 & 4 & 1 \\ 4 & 16 & 26 & 16 & 1 \\ 7 & 26 & 41 & 26 & 1 \\ 4 & 16 & 26 & 16 & 1 \\ 1 & 4 & 7 & 4 & 1 \end{bmatrix}$$

Once a suitable kernel is calculated then the Gaussian smoothing can be performed.

Remark. Given that the Kernel is circularly symmetric the convolution can be done with 1D matrices first in the x direction and then in the y

5.4.6 Bilateral Filtering

Bilateral filtering is a simple, non-iterative scheme of edge-preserving smoothing.

Remark. Bilateral Filtering was discovered in 1995 and used in 1997 as part of the SUSAN framework. Since then it has grown in popularity and is widely adopted in image processing applications [**bilateral˙filtering**]

Formally Bilateral Filtering is given by:

$$BF(I)_p = \frac{1}{W_p} \sum_{q \in S} G_{\sigma_s}(|\boldsymbol{p} - \boldsymbol{q}|) G_{\sigma_\tau}(I_p - I_q) I_q \tag{20}$$

Where W_p is a normalization factor:

$$W_p = \sum_{q \in S} G_{\sigma_s}(|\boldsymbol{p} - \boldsymbol{q}|) G_{\sigma_\tau}(I_p - I_q) \tag{21}$$

Bilateral Filtering has several qualities:

- intuitive, pixels are replaced by an average of their neighbours.
- dependant on two parameters, size and contrast.
- Non-iterative, parameter effect is not cumulative.

Typically it is assumed that spatial variation occurs slowly in images so it is appropriate for nearer pixels to have similar values. Simple filters such as Normalized Box do not account for the increased spacial variance that occurs at edges and tends to blur them. Bilateral Filtering is a solution that smooths out regions and prevents averaging of pixel values around edges.

Example 5.4.5. Intuition behind Bilateral Filtering

Consider:

$$G_{\sigma_s}(|\boldsymbol{p} - \boldsymbol{q}|) I_q$$

This is the definition for the Gaussian Blur Filter.

Hence the main component behind BF is:

$$G_{\sigma_\tau}(I_p - I_q)$$

The effect is controlled by σ_τ. For instance larger values of σ_τ will smooth pixels over larger ranges of contrast.

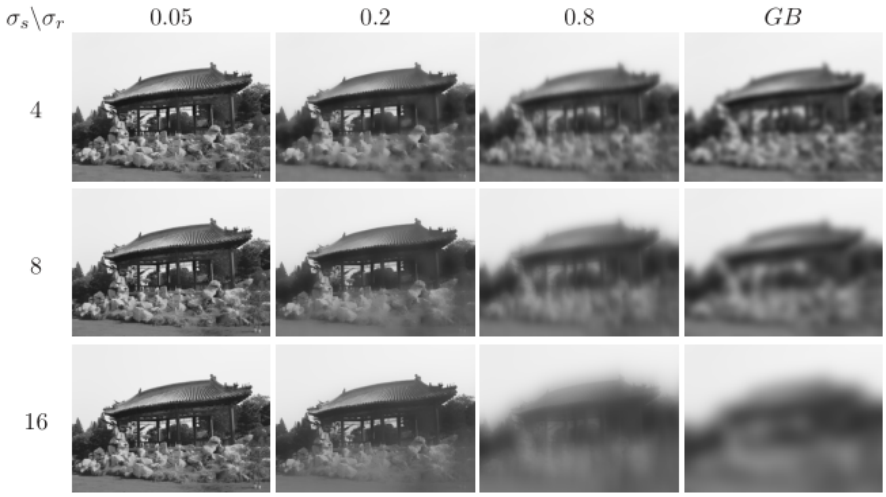

Figure 32: Bilateral Filtering Coefficients

Note. The GB column represents the image under a Gaussian Blur Filter with parameter σ_s

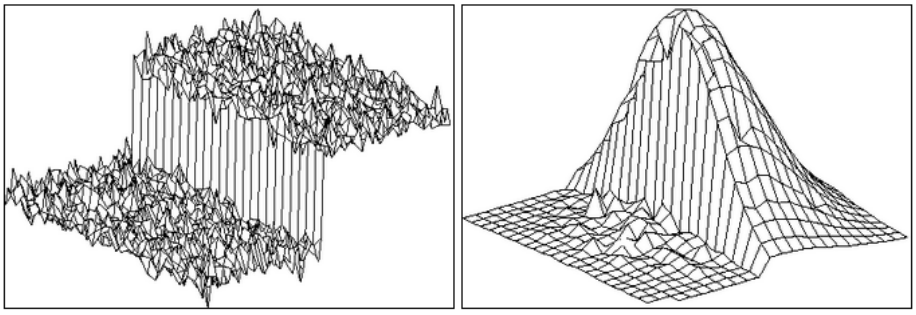

Figure 33: Bilateral Filtering

The effect of Bilateral Filtering is demonstrated above, where both regions of the image have been smoothed and the contrast / edge is preserved in the image.

5.4.7 Smoothing Functions in OpenCV

`cv.blur(src, dst, ksize, anchor=(-1, -1), borderType=BORDER_DEFAULT)`
Blurs an image using a normalized box filter.

Example 5.4.6. Mean Filtering

```
lena = cv.imread("./lena.png")
lena_1 = cv.blur(lena, (10,10))
lena_2 = cv.blur(lena, (30, 30))
```

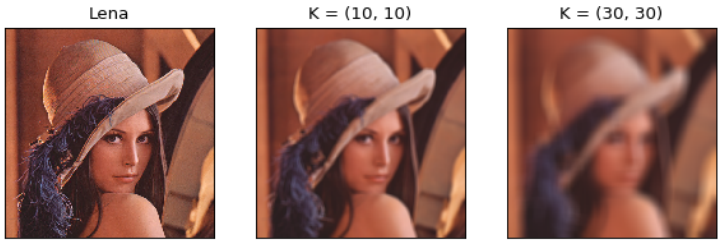

Figure 34: Mean Filtering

`cv.medianBlur(src, dst, ksize)`
Blurs an image using the median filter. ksize must be an odd number greater than 1, *ex:* 3, 5, 7

Example 5.4.7. Median Filtering

```
lena_1 = cv.medianBlur(lena, 9)
lena_2 = cv.medianBlur(lena, 91)
```

Figure 35: Median Filtering

`cv.GaussianBlur(src, dst, ksize, sigmaX, sigmaY=0)`
Blurs an image using a Gaussian filter.

Example 5.4.8. Gaussian Filtering

```
lena_1 = cv.GaussianBlur(lena, (9, 9), 1)
lena_2 = cv.GaussianBlur(lena, (31, 31), 5)
```

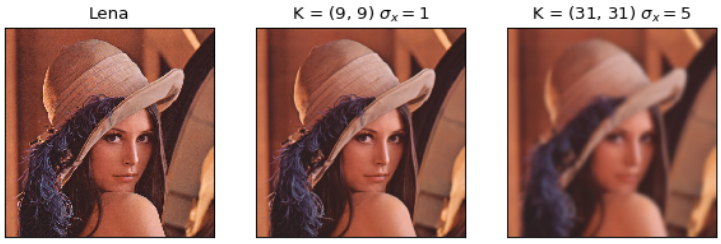

Figure 36: Gaussian Filtering

`cv.bilateralFilter(src, dst, d, sigmaColor, sigmaSpace)`
Applies the bilateral filter to an image, for simplicity sigma values can be the same, large sigma values > 150 will have a strong effect, giving a cartoonish look. Filter size $d > 5$, are slow. $d = 5$ is recommended, $d = 9$ is fine for offline applications.

Example 5.4.9. Bilateral Filtering

```
lena_1 = cv.bilateralFilter(lena, 5, 75, 75)
lena_2 = cv.bilateralFilter(lena, 19, 400, 30)
```

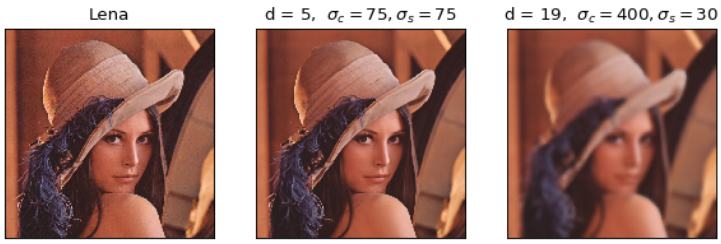

Figure 37: Bilateral Filtering

6 OpenCV: Image Processing - Transformations

6.1 Geometric Transformations

6.1.1 Scaling

Scaling compresses or expands an image along the coordinate directions.

An image or region of pixels can be compressed by sub sampling, this method is computationally simple, for instance a random pixel can be selected from a neighbourhood or an interpolation of pixel values can be used.

Example 6.1.1. Compression

$$I = \begin{matrix} 2 & 2 & 2 & 2 \\ 6 & 6 & 6 & 6 \\ 2 & 2 & 2 & 2 \\ 6 & 6 & 6 & 6 \end{matrix}$$

Single Pixel Selection	Interpolation
2 2	4 4
2 2	4 4

Remark. When a single pixel is selected (potentially at random) it is used to represent the compressed region. However a linear (or other functional) interpolation can provide a better representation of the image.

Similarly when an image zoomed it can be done through pixel replication or interpolation.

Example 6.1.2. Zooming

$$I = \begin{matrix} 2 & 5 \\ 2 & 5 \end{matrix}$$

Replication	Interpolation
2 2 5 5	2 3 4 5
2 2 5 5	2 3 4 5
2 2 5 5	2 3 4 5
2 2 5 5	2 3 4 5

Image scaling in OpenCV is done with the `resize` function.

`cv.resize(src, dsize, fx, fy, interpolation=INTER_LINEAR)`
Resizes an image down or up to the specified size

Example 6.1.3. Resizing

```
img = cv.imread("./lena.png")
res = cv.resize(img, None, fx=2, fy=2)
# height, width = img.shape[:2]
# res = cv.resize(img, (1*width, 2*height))
```

Note. `dsize` is a tuple specifying an exact width and height to resize to.

Enumerator	Description
INTER_NEAREST	nearest neighbor interpolation
INTER_LINEAR	bilinear interpolation
INTER_CUBIC	bicubic interpolation
INTER_AREA	resampling using pixel area relation. It may be a preferred method for image decimation, as it gives moiré-free results.
INTER_LANCZOS4	Lanczos interpolation over 8×8 neighbourhood
INTER_LINEAR_EXACT	Bit exact bilinear interpolation
INTER_MAX	mask for interpolation codes
WARP_FILL_OUTLIERS	Fills all of the destination image pixels. If some of them correspond to outliers in the source image, they are set to zero

Table 15: Interpolation Flags

6.1.2 Rotation

Rotation is most commonly used to improve the visual appearance of an image. The operation is performed by geometric transformation which maps the input position (x_1, y_1) to the output position (x', y') via a rotation about an origin O over a given angle θ.

In OpenCV the new pixel positions (x', y') are determined using the matrix $\boldsymbol{M}$:

$$\boldsymbol{M} = \begin{bmatrix} \alpha & \beta & (1-\alpha)x_0 - \beta y_0 \\ -\beta & \alpha & (1-\alpha)y_0 + \beta x_0 \\ 0 & 0 & 1 \end{bmatrix} \tag{22}$$

Where:

$$\alpha = scale \cdot \cos(\theta)$$
$$\beta = scale \cdot \sin(\theta)$$
$$x_0, y_0 : \text{ origin}$$

(x', y') are given by:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \boldsymbol{M} \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix}$$

The rotation matrix $\boldsymbol{M}$ is determined by the `getRotationMatrix2D` function, and the rotation carried out by `warpAffine`.

`cv.getRotationMatrix2D(center, angle, scale)`
Calculates an affine matrix of 2D rotation.

`cv.warpAffine(src, M, dsize, flags=INTER_LINEAR)`
Applies an affine transformation to an image.

Example 6.1.4. Rotation

```
rows, cols = img.shape
center = ((cols - 1)/2.0, (rows - 1)/2.0)
M = cv.getRotationMatrix2D(center, 45, 1)
res = cv.warpAffine(img, M, (cols, rows))
```

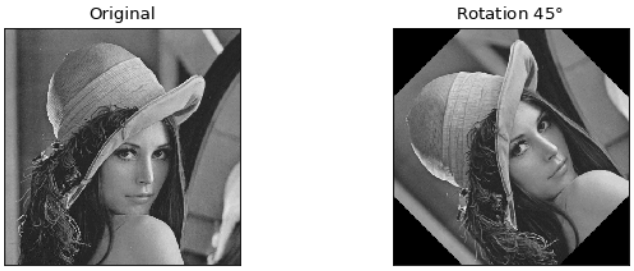

Figure 38: Rotation

6.1.3 Translation

Translation is often used to improve visual appearance. It also has a role as a preprocessor in applications where two or more images are involved. Translation maps each pixel in an input image (x_1, y_1) to a new position in an output image (x', y'). The dimensionality of the two images is often (but not strictly) the same.

Note. Translation is a special case of affine transformation.

Translation is performed with the affine matrix $\boldsymbol{M}$ given by:

$$M = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \end{bmatrix} \tag{23}$$

Hence (x', y') is determined by:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \boldsymbol{M} \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix}$$

Example 6.1.5. Translation

```
M =np.float32([
    [1, 0, 100],
    [0, 1, 100]
])
res = cv.warpAffine(img, M, img.shape)
```

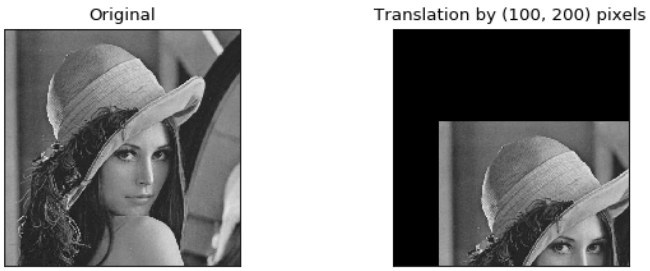

Figure 39: Translation

6.1.4 Affine Transformation

Occasionally geometric distortion can be introduced to an image due to perspective irregularities. Perspective irregularities typically occur due to the position of the camera with respect to the scene; altering the apparent dimensions of the scene geometry. Applying an affine transformation to a uniformly distorted image can correct the distortion.

Affine transformation is commonly written as:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \boldsymbol{A} \begin{bmatrix} x_1 \\ y_1 \end{bmatrix} + \boldsymbol{B} \tag{24}$$

Affine matrices can be used to carry out different geometric operations as shown previously.

Example 6.1.6. Affine transformation matrices and Geometric operations

Scaling

$$\boldsymbol{A} = \begin{bmatrix} a_{11} & 0 \\ 0 & a_{22} \end{bmatrix},\ \boldsymbol{B} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

Rotation

$$\boldsymbol{A} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix},\ \boldsymbol{B} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

Translation

$$\boldsymbol{A} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix},\ \boldsymbol{B} = \begin{bmatrix} t_x \\ t_y \end{bmatrix}$$

In order to calculate the affine matrix in OpenCV `getAffineTransform` is used.

`cv.getAffineTransform(src, dst)`
Calculates the affine transform from three pairs of corresponding points.

Example 6.1.7. Affine Transformation

```
pts1 = np.float32([
    [20, 20],
    [20, 100],
    [100, 20]
])
pts2 = np.float32([
    [10, 50],
    [25, 110],
    [70, 60]
])
M = cv.getAffineTransform(pts1, pts2)
res = cv.warpAffine(img, M, (cols, rows))
```

Remark. Several different affine transformations can be combined to produce a resultant image. It is important to note that the order at which the operations occur is significant, for instance the effect of rotation followed by transformation may not be equivalent to the converse.

6.1.5 Perspective Transformation

Perspective transformation uses a 3×3 Matrix in order to map input points to the new points (x', y').

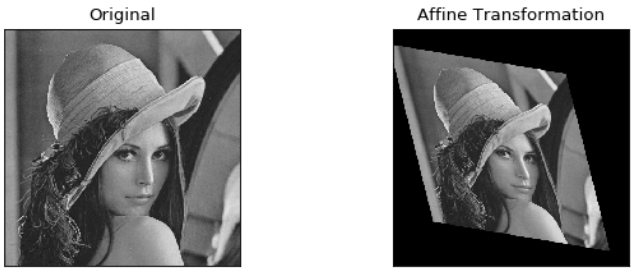

Figure 40: Affine Transformation

`cv.getPerspectiveTransform(src, dst, solveMethod=DECOMP_LU`
Calculates a perspective transform matrix from four pairs of corresponding points.

`cv.warpPerspective(src, M, dsize, flags=INTER_LINEAR)`
Applies a perspective transformation to an image.

Example 6.1.8. Perspective Transformation

```
img = cv.imread("./sudoku.png")

pts1 = np.float32([[56, 65], [368, 52], [28, 387], [389,
    390]])
pts2 = np.float32([[0, 0], [300, 0], [0, 300], [300, 300]])

M = cv.getPerspectiveTransform(pts1, pts2)
res = cv.warpPerspective(img,M,(300,300))
```

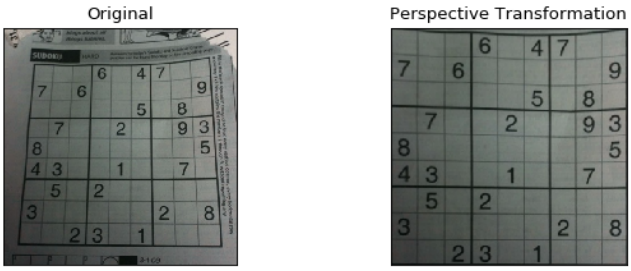

Figure 41: Perspective Transformation

Remark. Among the four points chosen 3 should no be collinear.

6.2 Morphological Transformations

6.2.1 Mathematical Morphology

Mathematical morphology contributes a wide range of operators to image processing, all based around a few mathematical concepts from set theory. The operators are particularly useful for the analysis of binary images and common usages include edge detection, noise removal, image enhancement and image segmentation.

The basic operations in morphology are erosion and dilation. Both operations require and image I and a kernel K. For binary images, white pixels are taken as the foreground and black pixels denote the background. For grayscale images, the intensity represents the height above the base plane.

Conceptually erosion and dilation work by translating the Kernel over various points in the input image and examining the intersection between the kernel coordinates and the input image coordinates. In the case of erosion the output image O consists of the points in which K can be translated while remaining within the input image.

Remark. Other operators are defined as a combination of erosion and dilation along with set operators. *ex.* opening, closing, skeletonization.

6.2.2 Erosion

Erosion operators takes an image I and a Kernel K also referred to a structuring element. The structuring element determines the effect of the erosion on the input image.

Def 8. Erosion

The erosion of I by K is the set of all points x s.t. $K_x \subseteq I$.

$$I \ominus K = \{K_x \subseteq I\} \tag{25}$$

Where:

I is a set of Euclidean coordinates corresponding to the input binary image.

K is the set of coordinates for the structuring element.

K_x denotes the translation of K s.t. the origin is at x

Remark. The mathematical definition for grayscale erosion is identical except the set of coordinates associated with the image are 3D instead of 2D.

Example 6.2.1. Erosion

$$I = \begin{array}{cccccccc} & \vdots & \vdots & \vdots & \vdots & \vdots & \\ \dots & 1 & 1 & 1 & 1 & 1 & \dots \\ \dots & 1 & 1 & 1 & 1 & 1 & \dots \\ \dots & 1 & 1 & 1 & 1 & 1 & \dots \\ \dots & 1 & 1 & \mathbf{0} & 1 & 1 & \dots \\ \dots & 1 & 1 & 1 & 1 & 1 & \dots \\ & \vdots & \vdots & \vdots & \vdots & \vdots & \end{array}$$

$$K = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

For each pixel p, K is superimposed over I if it's fully contained then the value is retained else it is set to 0.

The erosion of I by K is:

$$I \ominus K = \begin{array}{cccccccc} & \vdots & \vdots & \vdots & \vdots & \vdots & \\ \ldots & 1 & 1 & 1 & 1 & 1 & \ldots \\ \ldots & 1 & 1 & 1 & 1 & 1 & \ldots \\ \ldots & 1 & \mathbf{0} & \mathbf{0} & \mathbf{0} & 1 & \ldots \\ \ldots & 1 & \mathbf{0} & \mathbf{0} & \mathbf{0} & 1 & \ldots \\ \ldots & 1 & \mathbf{0} & \mathbf{0} & \mathbf{0} & 1 & \ldots \\ & \vdots & \vdots & \vdots & \vdots & \vdots & \end{array}$$

Remark. In this case it is assumed that the origin of I is the centre.

Grayscale erosion results in a decrease in intensity. Generally darkening the image. Bright regions are shrunk in size and dark regions grow. The effect is noticeable when there is a sharp increase in intensity. Regions with uniform intensity are left practically unchanged.

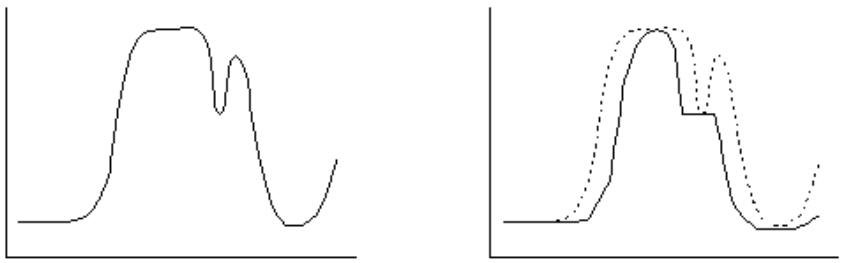

Figure 42: Grayscale erosion intensity change

Note. Eroding foreground pixels is the same as dilating background pixels. In other words Erosion is the *dual* to dilation.

6.2.3 Dilation

Dilation is the second basic operator in the field of mathematical morphology. The basic effect is to gradually enlarge the boundaries of regions of foreground pixels. The dilation operator takes an input image I and a Kernel or structuring element K.

Def 9. Dilation

The dilation of I by K is the set of all points x s.t. the intersect is non empty.

$$I \oplus K = \{K_x \cap I = \emptyset\} \tag{26}$$

Where:

I is a set of Euclidean coordinates corresponding to the input binary image.

K is the set of coordinates for the structuring element.

K_x denotes the translation of K s.t. the origin is at x

Example 6.2.2. Dilation

$$I = \begin{array}{cccccccc} & \vdots & \vdots & \vdots & \vdots & \vdots & \\ \ldots & \mathbf{0} & 1 & 1 & 1 & 1 & \ldots \\ \ldots & 1 & 1 & 1 & 1 & 1 & \ldots \\ \ldots & 1 & 1 & \mathbf{0} & \mathbf{0} & \mathbf{0} & \ldots \\ \ldots & 1 & 1 & \mathbf{0} & \mathbf{0} & \mathbf{0} & \ldots \\ \ldots & 1 & 1 & \mathbf{0} & \mathbf{0} & \mathbf{0} & \ldots \\ & \vdots & \vdots & \vdots & \vdots & \vdots & \end{array}$$

$$K = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

For each pixel p, K is superimposed over I if there is an intersect then the value is retained else it is set to 0.

The dilation of I by K is:

$$I \oplus K = \begin{array}{cccccccc} & \vdots & \vdots & \vdots & \vdots & \vdots & \\ \ldots & 1 & 1 & 1 & 1 & 1 & \ldots \\ \ldots & 1 & 1 & 1 & 1 & 1 & \ldots \\ \ldots & 1 & 1 & 1 & 1 & 1 & \ldots \\ \ldots & 1 & 1 & 1 & \mathbf{0} & \mathbf{0} & \ldots \\ \ldots & 1 & 1 & 1 & \mathbf{0} & \mathbf{0} & \ldots \\ & \vdots & \vdots & \vdots & \vdots & \vdots & \end{array}$$

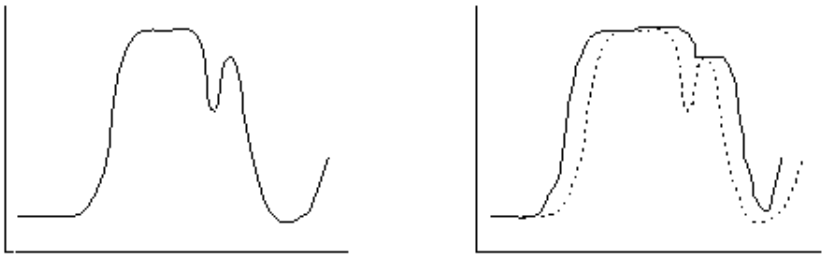

Figure 43: Dilation image intensity change

Dilation has several use cases. It can be used to fill small spurious holes ("pepper noise"). Edge detection; performed by first taking the dilation of an image and subtracting the original image. Finally dilation forms the basis of other morphology operations in conjunction with logical operators, *i.e.* region filling.

6.2.4 Opening

Opening is derived from the fundamental morphology operations, dilation and erosion. As such the operator requires an input image I and a structuring element K. The effect of the operator is to preserve foreground regions that have a similarly shape to the structuring element, or that contain the structuring element, while eliminating all other regions of foreground pixels.

Def 10. Opening
Defined by erosion of I by K followed by a dilation of the product.

$$I \circ K = (I \ominus K) \oplus K \tag{27}$$

Remark. Opening can be used to remove "salt noise" from an image.

6.2.5 Closing

Similar to opening, closing is defined from basic morphology operators. Closing tends to enlarge boundaries of foreground regions, but is less destructive of the original image. The operation is determined by the structuring element.

Def 11. Closing
Defined by dilation of I by K followed by an erosion of the product.

$$I \bullet K = (I \oplus K) \ominus K \tag{28}$$

Remark. The primary use is to fill small background holes in an image ("pepper noise"). A downside is that the dilation will distort all regions of pixels.

6.2.6 Other Operations

Def 12. Morphological Gradient
Is the difference between the dilation and the erosion of an image.

$$I \triangle K = (I \oplus K) - (I \ominus K) \tag{29}$$

Remark. The resulting image will be the outline of the input image.

Def 13. Top Hat
Is the difference between the input image and the opening of the image.

$$I - (I \circ K) \tag{30}$$

Def 14. Black Hat
Is the difference between the input image and the closing of the image.

$$I - (I \bullet K) \tag{31}$$

6.2.7 Morphological Operations in OpenCV

`cv.erode(src, kernel, iterations=1)`
Erodes an image with a specific structuring element.

Example 6.2.3. Erosion

```
img = cv.imread("./coins.png")
kernel = np.ones((5,5),np.uint8)
erosion = cv.erode(img, kernel, iterations=1)
```

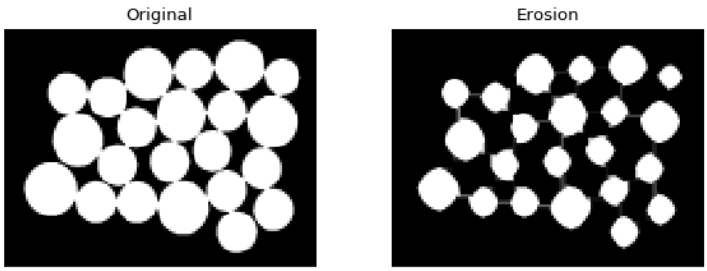

Figure 44: Erosion

`cv.dilate(src, kernel, iterations=1)`
Dilates an image by using a specific structuring element

Example 6.2.4. Dilation

```
kernel = np.ones((3,3),np.uint8)
dilation = cv.dilate(img, kernel, iterations=1)
```

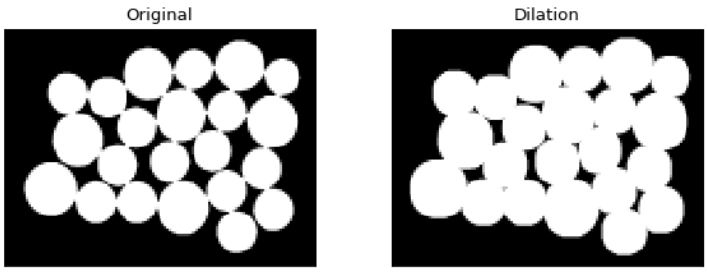

Figure 45: Dilation

`cv.morphologyEx(src, op, kernel, iterations=1)`
Performs advanced morphological transformations

Example 6.2.5. Advanced Morphology

```
kernel = cv.getStructuringElement(cv.MORPH_RECT, (3, 3))
grad = cv.morphologyEx(img, cv.MORPH_GRADIENT, kernel)
```

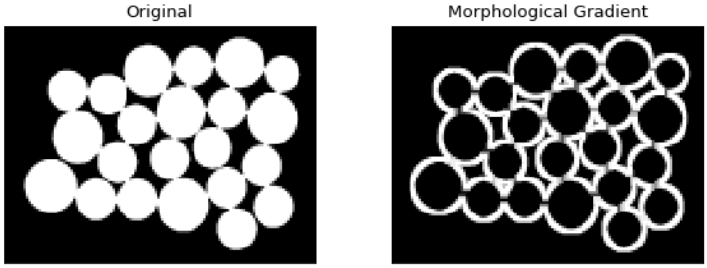

Figure 46: Morphological Gradient

`cv.getStructuringElement(shape, ksize)`
Returns a structuring element of the specified size and shape for morphological operation

Enumerator	Description
MORPH_RECT	Rectangular shaped structuring element
MORPH_CROSS	Cross-shaped structuring element
MORPH_ELIPSE	An elliptic structuring element inscribed into a rectangle

Table 16: Structuring Elements

The morphological operations are available as enumerators, and can be passed to `morphologyEx()`.

Enumerator	Description
MORPH_ERODE	Erosion
MORPH_DILATE	Dilation
MORPH_OPEN	opening operation
MORPH_CLOSE	closing operation
MORPH_GRADIENT	morphological gradient
MORPH_TOPHAT	"top hat"
MORPH_BLACKHAT	"black hat"
MORPH_HITMISS	"Hit or miss"

Table 17: Morphological Operations

6.3 Image Gradients

One of the most important uses of convolution is the computation of derivatives in an image. Image derivatives are important for detecting edges. For instance it is easy to determine an edge due to the sudden change in pixel intensity; a convenient way of expressing this change is with derivatives.

6.3.1 Sobel Operator

The Sobel operator performs a 2D spatial gradient measurement on an image. It emphasises regions that correspond to edges. The operator works using a pair of kernels $G_x$and G_y. The kernels are designed to respond maximally to edges running vertically and horizontally. Combining the products of the convolution results in the absolute magnitude.

The Sobel operator is given by:

$$G = \sqrt{G_x^2 + G_y^2} \tag{32}$$

Where:

$$G_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} * I$$

$$G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix} * I$$

The Sobel operation is performed with the `Sobel` function.

`cv.Sobel(src, ddepth, dx, dy, ksize=3, scale=1, delta=0)`
Calculates the first, second, third or mixed image derivative using an extended Sobel operator. There is also kszie $= -1$ which corresponds to a 3×3 Scharr filter.

Example 6.3.1. Sobel Filtering

```
img = cv.imread("./images/sudoku.png")
res_x = cv.Sobel(img, cv.CV_64F, 1, 0, ksize=5)
res_y = cv.Sobel(img, cv.CV_64F, 0, 1, ksize=5)
```

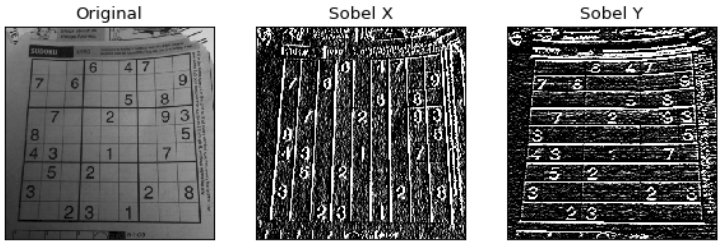

Figure 47: Sobel Filtering

Note. By setting the Kernel size to -1 the Scharr Kernels can be applied instead of the Sobel Kernels. Scharr filtering often produces better results than Sobel filtering for smaller sized kernels. Alternatively the `Scharr` function can be used.

6.3.2 Laplace Operator

The Laplace Operator, denoted as $\Delta\cdot$, is defined in terms of second derivatives. The formal mathematical definition is:

$$\Delta f = \sum_{i=1}^{n} \frac{\partial^2 f}{\partial x_i^2} \tag{33}$$

In terms of images the Laplace operator takes the second derivative with respect to the x and y axis.

$$\Delta I = \frac{\partial^2 I}{\partial x^2} + \frac{\partial^2 I}{\partial y^2} \tag{34}$$

The Laplace operator can be used as an edge detector, for instance the first derivative of an image grows rapidly when there is a change from low to high intensity, i.e. at an edge, and then decreases at a discontinuity. This implies an edge occurs at a local maxima.

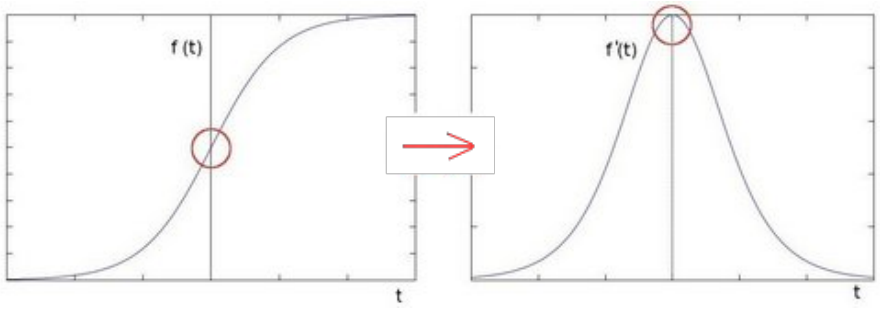

Figure 48: First Derivative

Hence the second derivative of an image implies that edges occur at values around 0, i.e. the point of the local maxima.

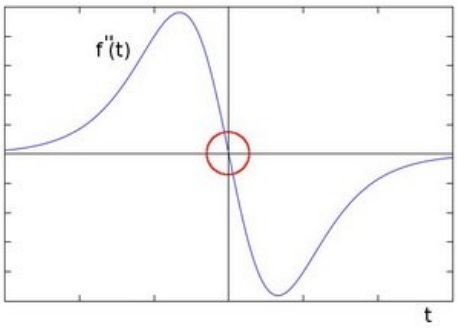

Figure 49: Second Derivative

`cv.Laplace(src, ddepth, ksize=1, scale=1, delta=0)`
Calculates the Laplacian of an image.

Example 6.3.2. Laplace Operator

```
res_lap = cv.Laplacian(img, cv.CV_64F)
```

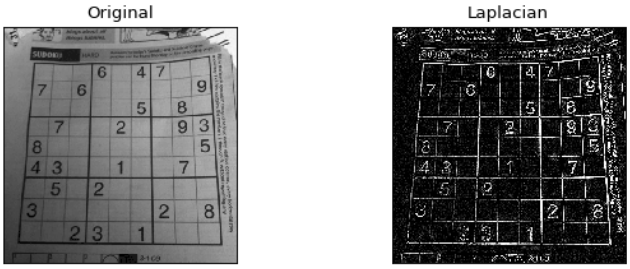

Figure 50: Laplace Operator

Remark. The OpenCV implementation of the Laplace Operator uses the Sobel Operator in its computation.

6.4 Canny Edge Detection

Canny edge detection uses a more sophisticated multi-staged techniques to detect edges. The function takes a grey scale image as input and produces an image showing the positions of intensity change. The operator is based on a set of particular criteria, there are other detectors that also claim to be optimal with slightly different criteria.

6.4.1 Theory

Canny Edge Detection begins by reducing the noise in an image with a 5×5 Gaussian filter. The smoothed image is then filtered with a Sobel Kernel in both the horizontal and vertical directions. The first derivatives G_x and G_y are then used to find the gradient and the direction of each pixel.

The gradient is given by:

$$G = \sqrt{G_x^2 + G_y^2}$$

The direction or angle is given by:

$$\theta = \tan^{-1}\left(\frac{G_y}{G_x}\right) \tag{35}$$

Note. Gradient direction is always perpendicular to edges, angles within a given range are rounded to one of four angles representing vertical, horizontal or diagonal directions.

After each pixel's gradient and direction is established, the pixel is tested within its local neighbourhood to established whether it is a local maxima or not. If a pixel is not the local maxima it is set to 0, otherwise it is considered for the next stage.

Finally a minimum and maximum threshold, T_{min} and T_{max} are used to determine which pixels are actually edges. Pixels with values above T_{max} are considered sure edges. Pixels between T_{max} and T_{min} but connected to a 'sure edge' are considered edges. Otherwise the pixel is discarded.

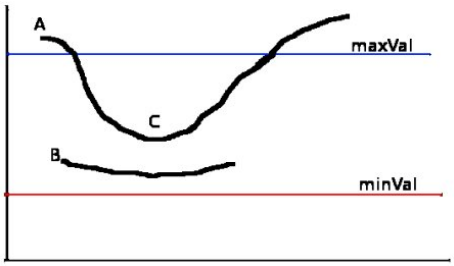

Figure 51: Canny Edge Detection

For instance in the above figure, Point C is connected to a sure edge (A) and so is considered as part of the final result. Point B is between the valid threshold, but not connected to a sure edge and so is discarded. The final stage is useful for removing small lines and noise which is often not considered as an edge.

6.4.2 Canny Edge Detection in OpenCV

`cv.Canny(img, minVal, maxVal, apertureSize=3, L2gradient=False)`
Finds edges in an image using the Canny algorithm. The apertureSize is the size of the Sobel Kernel

Example 6.4.1. Canny Edge Detection

```
img = cv.imread("./images/grasshopper.png", 0)
canny = cv.Canny(img, 80, 180)

# plotting
plt.figure(num=None,figsize=(9,3), dpi=80, facecolor='w',
    edgecolor='k')

plt.subplot(1, 2, 1)
plt.xticks([]), plt.yticks([]),
plt.title("Original")
plt.imshow(img, cmap="gray")

plt.subplot(1, 2, 2)
plt.xticks([]), plt.yticks([]),
plt.title("Canny")
plt.imshow(canny, cmap="gray")

plt.show()
```

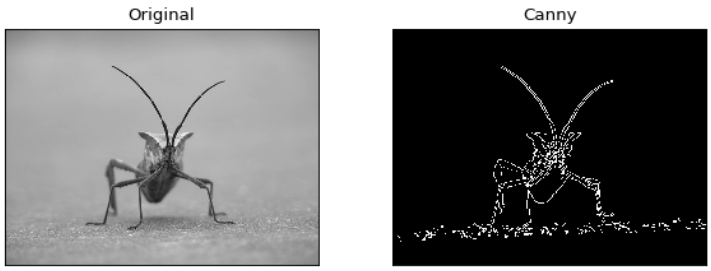

Figure 52: Canny Edge Detection Example

6.5 Hough Transforms

6.5.1 Hough Line Transform Theory

In the Cartesian coordinate system a line is expressed using two parameters: (m, c) giving the result: $y = mx + c$. In Polar coordinates the same result can be expressed with parameters (θ, ρ) as:

$$y = \left(-\frac{\cos\theta}{\sin\theta} \right) x + \left(\frac{\rho}{\sin\theta} \right) \tag{36}$$

Rearranging gives:

$$\rho = x\cos\theta + y\sin\theta$$

Hence the θ, ρ plane can represent the set of lines that pass through a point (x, y) as shown in the following plot of the curve $\rho = 8\cos\theta + 6\sin\theta$

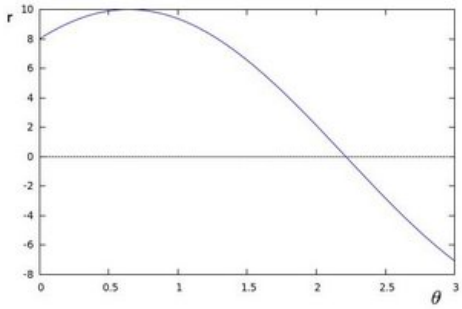

Figure 53: Set of lines that pass through $x = 8, y = 6$

Now consider the points: $(8, 6), (4, 9), (12, 3)$; drawing the curves for each in the θ, ρ plane gives:

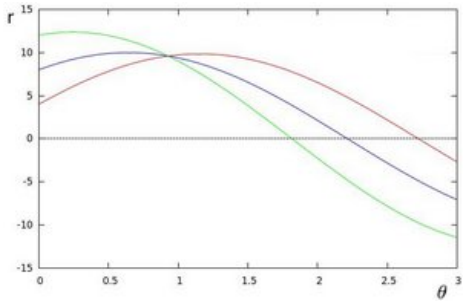

Figure 54: Set of lines for points $(8, 6), (4, 9), (12, 3)$

Intersecting curves imply that all the points belong to the same line.

The Hough Line Transform keeps track of the intersections between the curves of every point in an image. If the number of intersections is above some threshold, then it declares it as a line with the parameters (θ, ρ_θ) of the intersection point.

6.5.2 Hough Line transform in OpenCV

OpenCV implements two kinds of Hough line transforms: Standard and Probabilistic.

Note. The Probabilistic implementation only considers a random subset of points in order to detect lines.

`cv.HoughLines(img, rho, theta, threshold)`
Finds lines in a binary image using the Standard Hough transform.

`cv.HoughLinesP(img, rho, theta, threshold)`
Finds line segments in a binary image using the Probabilistic Hough transform.

Example 6.5.1. Hough Line Transform

```
img = cv.imread("./images/sudoku.png")
img_2 = img.copy()
gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY)
edges = cv.Canny(gray, 50, 150)

# Standard
hough_pts_std = cv.HoughLines(edges, 1, np.pi/100, 200)

for line in hough_pts_std:
    rho,theta = line[0]
    a = np.cos(theta)
    b = np.sin(theta)
    x0 = a*rho
    y0 = b*rho
    x1 = int(x0 + 1000*(-b))
    y1 = int(y0 + 1000*(a))
    x2 = int(x0 - 1000*(-b))
    y2 = int(y0 - 1000*(a))
    cv.line(img,(x1,y1),(x2,y2),(0,0,255),2)

# Probabilistic
hough_pts_prob = cv.HoughLinesP(edges, 1, np.pi/100, 100,
    minLineLength=100, maxLineGap=10)

for line in hough_pts_prob:
    x1,y1,x2,y2 = line[0]
    cv.line(img_2,(x1,y1),(x2,y2),(0,255,0),2)
```

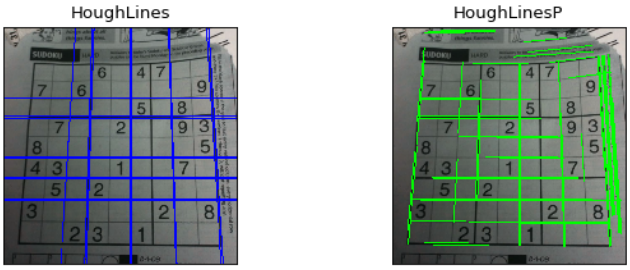

Figure 55: Hough Line Transform

6.5.3 Hough Circle Transform Theory

The Hough Circle Transform works similarly to the Hough Line Transform. However three parameters are needed to define a circle:

$$(x_{center}, y_{center}, r)$$

The increased dimensionality of the problem, *i.e.* the consideration of the radius r, means the problem is computationally more intense; OpenCV implements a method known as the Hough gradient method to determine these values.

The Hough gradient method works by first passing an image through an edge detector, in this case the `cv.Canny()` function. For every non-zero pixel in the edge image, the local gradient is computed via the first-order Sobel derivatives. Using the gradient, a set of lines is determined constrained by the specified minimum and maximum radial distance, every point that lies on one of these lines is added to an accumulator. Using the non-zero pixels in the edge image, the candidate centres are determined and sorted in descending order based on their accumulator values. Therefore the centres with the most supporting pixels appear first. Using the pixels in the accumulator list described earlier, the pixels are sorted based on their distance from the centre, a radius is selected which is closest to the specified maximum radius. A centre is kept if it has sufficient support from the non-zero pixels in the edge image and if it is a sufficient distance from any previously selected circle.

6.5.4 Hough Circle Transform in OpenCV

`cv.HoughCircles(img, method, dp, minDist, param1=100, param2=100, minRadius =0, maxRadius=0)`
Finds circles in a grayscale image using the Hough Transform. Returns an output vector of found circles.

`dp`	is the accumulator resolution, i.e. a value of 2 means the accumulator has half the resolution as the input image
`minDist`	is the minimum distance between circles
`param1`	is the higher threshold passed to the Canny edge detector
`param2`	is the accumulator threshold, a lower value may detect more false circles.

Example 6.5.2. Hough Circle Transform

```
# load image
img = cv.imread("./images/coins_2.png")

# process image
img = cv.medianBlur(img, 7)
img_gr = cv.cvtColor(img, cv.COLOR_BGR2GRAY)
img_edg = cv.Canny(img_gr, 100, 200)

# apply hough circle transform
circles = cv.HoughCircles(img_gr, cv.HOUGH_GRADIENT, 1,
    minDist=60, param1=200, param2=40,
    minRadius=30, maxRadius=200)

circles = np.uint16(np.around(circles))

for cir in circles[0,:]:
    center = (cir[0], cir[1])
    rad = cir[2]
    # draw the outer circle
    cv.circle(img, center, rad, (0,255,0), 20)
    # draw marker at center
    cv.circle(img, center, 20, (0,0,255), -1)
```

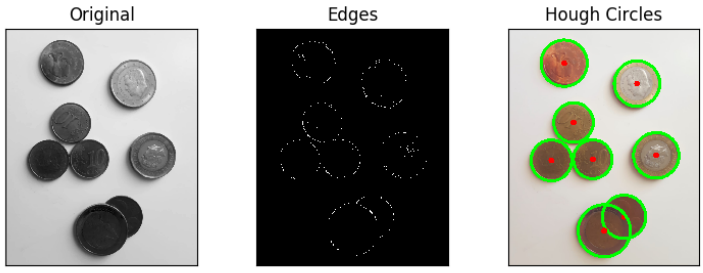

Figure 56: Hough Circle Transform with Coins

7 OpenCV: Image Processing - Segmentation

7.1 Image Contours