```
import numpy as np
import matplotlib.pyplot as plt
import os
import cv2
import torch
```

#1. NumPy

Arrays and Matrices

```
a = np.array([1, 2, 3])
b = np.array([10, 20, 30])
print(f'A is {a}. B is {b}')
print("Object Type:",type(a))
```

```
A is [1 2 3]. B is [10 20 30]
Object Type: <class 'numpy.ndarray'>
```

Elemen dari array dapat diakses dengan menggunakan notasi `[]`. Untuk mengakses elemen dari belakang, kita dapat menggunakan angka negatif.

```
print(a[1])
print(b[-2])
print(repr(a[0:3:2]))
print(a[0:3:2])
```

```
2
20
array([1, 3])
[1 3]
```

In Python, print(repr(a[0:3:2])) is a statement that prints the representation of a slice of the list a. Let's break down the components:

a: This is assumed to be a list or any other sequence (e.g., a string or tuple) from which you are taking a slice.

[0:3:2]: This is the slice notation. It represents a slice of the list a starting from index 0, up to (but not including) index 3, with a step of 2. In other words, it selects elements at index 0 and index 2, skipping every other element.

repr(): This is a built-in Python function that returns the canonical string representation of an object. For most built-in data types, repr() produces a string that, when passed to eval(), would recreate the original object. For lists, it will represent the list with its elements and the slice notation.

Karena array memiliki ukuran yang tetap, kita dapat memperoleh ukuran ini dengan menggunakan atribut `shape`

```
print(a.shape)
print(b.shape)
```

```
(3,)
(3,)
```

Karena `a` dan `b` adalah numpy arrays, kita dapat melakukan operasi matematika terhadap mereka

```
print("Numpy math:")
print(f'{a} + {b} = {a + b}')

# If they weren't numpy arrays
print("\nVanilla Python List math:")
print(list(a),"+",list(b),"=",list(a) + list(b))
```

```
Numpy math:
[1 2 3] + [10 20 30] = [11 22 33]

Vanilla Python List math:
[1, 2, 3] + [10, 20, 30] = [1, 2, 3, 10, 20, 30]
```

Matriks


```
my_matrix = np.array(
    [
        [1, 2, 3],
        [4, 5, 6]
    ]
)
print(repr(my_matrix))
print("matrix shape: ", my_matrix.shape)

print("\n")
print(my_matrix)
```

```
array([[1, 2, 3],
       [4, 5, 6]])
matrix shape:  (2, 3)


[[1 2 3]
 [4 5 6]]
```

Kita dapat mengakses array pada matriks menggunakan indexing

```
# Get row 0, all columns except the last one
print(repr(my_matrix[0,:-1]))

# Get columns 1 and 2.
print(repr(my_matrix[:, 1:3]))
```

```
array([1, 2])
array([[2, 3],
       [5, 6]])
```

Tidak seperti nested lists, dimana setiap list dapat memiliki jumlah elemen yang berbeda, Setiap baris pada matriks harus memiliki ukuran yang sama.

```
print(my_matrix.ravel())
```

```
[1 2 3 4 5 6]
```

ravel(): This is a method available for NumPy arrays. It returns a flattened version of the array. Flattening means that it converts the two-dimensional array into a one-dimensional array by concatenating all the elements row-wise.

Kita dapat mengubah 6 elemen ini menjadi bentuk apapun yang diinginkan

```
my_reshaped_matrix = my_matrix.reshape((3,2))
my_reshaped_matrix
```

```
array([[1, 2],
       [3, 4],
       [5, 6]])
```

Tambahan Dimensi

Kita dapat menambah sebuah dimensi terhadap array 1d untuk membuat matriks 2d.

Data pada computer vision biasanya berbentuk (Tinggi x Lebar x 3 warna)

```
my_tensor = np.array(range(2 * 3 * 4)).reshape(2,3,4)
print(my_tensor)
print("tensor shape: ", my_tensor.shape)
```

```
[[[ 0  1  2  3]
  [ 4  5  6  7]
  [ 8  9 10 11]]

 [[12 13 14 15]
  [16 17 18 19]
  [20 21 22 23]]]
tensor shape:  (2, 3, 4)
```

Let's break down the code:

np.array(range(2 * 3 * 4)): This creates a 1-dimensional NumPy array containing elements from 0 to 23. The range() function generates numbers from 0 up to (but not including) the specified value (2 * 3 * 4 = 24 in this case).

.reshape(2, 3, 4): After creating the 1-dimensional array, the reshape() method is applied to change its shape to (2, 3, 4). The elements are then arranged in a 3-dimensional manner, as specified by the provided shape.

Membuat Array

```
desired_shape = (2, 3)
print(repr(  np.zeros(desired_shape)          ),end="\n\n")
print(repr(  np.ones(desired_shape)           ),end="\n\n")
print(repr(  np.eye(3)                        ),end="\n\n")
print(repr(  np.full(desired_shape, 7)        ),end="\n\n")
print(repr(  np.random.random(desired_shape)  ),end="\n\n")
```

```
array([[0., 0., 0.],
       [0., 0., 0.]])

array([[1., 1., 1.],
       [1., 1., 1.]])

array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])

array([[7, 7, 7],
       [7, 7, 7]])

array([[0.39306054, 0.94904859, 0.72710692],
       [0.08327287, 0.51069647, 0.84291125]])
```

Mengatur elemen

```
my_tensor[0, 0, 0] = 100
print(my_tensor)

print()

# Get the 0th matrix slice. Rows 1 and 2. Get every other column starting from 0
print(my_tensor[0, 1:3, 0::2])
```

```
[[[100   1   2   3]
  [  4   5   6   7]
  [  8   9  10  11]]

 [[ 12  13  14  15]
  [ 16  17  18  19]
  [ 20  21  22  23]]]

[[ 4  6]
 [ 8 10]]
```

my_tensor[0, 1:3, 0::2]: my_tensor: This is the NumPy array you created earlier, which is a 3-dimensional tensor with shape (2, 3, 4). [0, 1:3, 0::2]: This is the slicing notation used to extract a portion of the array. Now, let's interpret each part of the slicing notation:

0: This selects the first element along the first axis of the tensor. So, it selects the first 2x3 matrix within the tensor (the matrix at index 0).

1:3: This selects elements starting from index 1 up to (but not including) index 3 along the second axis of the tensor. This means it selects rows with indices 1 and 2 within the selected 2x3 matrix.

0::2: This selects elements starting from index 0 to the end, with a step of 2 along the third axis of the tensor. This means it selects columns with indices 0 and 2 within the selected rows.

Menggunakan Array Integer sebagai indeks

```
my_array = np.array(range(5)) * 10 + 3
print("my_array:")
print(my_array)
# We want the elements in this order
array_of_indices = np.array([4, 1, 3, 0, 2])
print("new order: ", repr(array_of_indices))
print(my_array[array_of_indices])
```



```
my_array:
[ 3 13 23 33 43]
new order:  array([4, 1, 3, 0, 2])
[43 13 33  3 23]
```

my_array = np.array(range(5)) * 10 + 3

Let's break down the code step by step:

range(5): The range() function generates numbers from 0 up to (but not including) the specified value. In this case, it generates numbers from 0 to 4.

np.array(range(5)): The np.array() function converts the generated numbers from the range() function into a NumPy array.

- 10: This multiplies each element in the NumPy array by 10.
- 3: This adds 3 to each element in the NumPy array.

The result is an array where each element is generated by taking the number from 0 to 4, multiplying it by 10, and then adding 3 to it.

Kita juga dapat menggunakan Array Boolean untuk menutup nilai matriks

```
# Make a selector array
selector = (np.random.random((3,4)) * len(my_array)).astype(int)

# np.random.random((3, 4)): This generates a 3x4 NumPy array filled with random floating-point numbers between 0 (inc
# * len(my_array): This multiplies each element of the random array by the length of my_array, which is 5 in your pre
# .astype(int): This converts the elements of the resulting array to integers. It truncates the decimal part of the n

print("selector array: ")
print(my_array,"my array")
print(selector)
# Now, we can use these to get elements from our original array!
my_array[selector]
```

```
selector array:
[ 3 13 23 33 43] my array
[[1 1 1 1]
 [4 2 0 0]
 [3 4 4 1]]
array([[13, 13, 13, 13],
       [43, 23,  3,  3],
       [33, 43, 43, 13]])
```

Kita dapat mengatur elemen elemen seperti ini

```
# create a mask
my_mask = np.array([1, 0, 0, 1, 1], dtype=bool)
print("mask: ", repr(my_mask))
print("Original")
print(repr(my_array))
print("Masked with",repr(my_mask))
print(repr(my_array[my_mask]))
```

```
mask:  array([ True, False, False,  True,  True])
Original
array([ 3, 13, 23, 33, 43])
Masked with array([ True, False, False,  True,  True])
array([ 3, 33, 43])
```

```
print(repr(my_array))
replacement = np.array([600, 700, 800])
my_array[my_mask] = replacement
print(repr(my_array))
```

```
array([ 3, 13, 23, 33, 43])
array([600,  13,  23, 700, 800])
```

operator < dan > akan menutup sebagian elemen pada array

```
# for example, you want to change pixels with gray-level larger than 20
print(my_array > 20)
print(my_array[my_array > 20])
```

```
[ True False  True  True  True]
[600  23 700 800]
```

Operasi Matriks

```
# create arrays
a = np.array(range(10)).reshape(2,5)
b = np.array(range(100, 1100, 100)).reshape(2,5)
print(f'a = {a}')
print("b =",repr(  b  ))
```

```
a = [[0 1 2 3 4]
 [5 6 7 8 9]]
b = array([[ 100,  200,  300,  400,  500],
       [ 600,  700,  800,  900, 1000]])
```

```
# sum
print("a + b = ", repr(a + b))
# multiply
print("a * b = ", a * b)
# broadcasting
print("a * 2 = ", repr(a * 2))
```

```
a + b =  array([[ 100,  201,  302,  403,  504],
       [ 605,  706,  807,  908, 1009]])
a * b =  [[   0  200  600 1200 2000]
 [3000 4200 5600 7200 9000]]
a * 2 =  array([[ 0,  2,  4,  6,  8],
       [10, 12, 14, 16, 18]])
```

Matematika Matriks

```
# create matrices
A = np.array( range(6) ). reshape((3,2))
B = np.array( range(10,16) ). reshape((2,3))

print("A =", repr(A))
print("B =", repr(B))
```

```
A = array([[0, 1],
       [2, 3],
       [4, 5]])
B = array([[10, 11, 12],
       [13, 14, 15]])
```

```
# matrix multiplication
print("AB = ", repr(np.matmul(A, B)))
# also with @
print("A @ B = ", repr(A @ B))
```

```
AB =  array([[ 13,  14,  15],
       [ 59,  64,  69],
       [105, 114, 123]])
A @ B =  array([[ 13,  14,  15],
       [ 59,  64,  69],
       [105, 114, 123]])
```

```
A = np.array([
    [3, 2, 1],
    [4, 8, 2],
    [1, 2, 3]
])
print("Transpose:\n", repr(A.T))
A_inv = np.linalg.inv(A)
print("Inverse:", repr(A_inv), sep="\n")
```

```
Transpose:
 array([[3, 4, 1],
       [2, 8, 2],
       [1, 2, 3]])
Inverse:
array([[ 0.5 , -0.1 , -0.1 ],
       [-0.25,  0.2 , -0.05],
       [ 0.  , -0.1 ,  0.4 ]])
```



```
# create some matrix
A = np.array(range(10)).reshape(2,5)
A
```

```
array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])
```

```
print("A's Shape is", A.shape)
col_sum = np.sum(A, axis=0) # A.sum(0)
row_sum = np.sum(A, axis=1) # A.sum(1)
total_sum = np.sum(A) # A.sum()
print("Row sums (shape: %s) - sum all values on axis 0 (along the row)" % str(row_sum.shape))
print(repr(row_sum))
print("Col sums (shape: %s) - sum all values on axis 1 (along the column)" % str(col_sum.shape))
print(repr(col_sum))
print("Full array sum (scalar)")
print(repr(total_sum))
```

```
A's Shape is (2, 5)
Row sums (shape: (2,)) - sum all values on axis 0 (along the row)
array([10, 35])
Col sums (shape: (5,)) - sum all values on axis 1 (along the column)
array([ 5,  7,  9, 11, 13])
Full array sum (scalar)
45
```

Menumpuk Arrays

```
A = np.array((range(10))).reshape(2,5)
B = np.array((range(10, 20))).reshape(2,5)
print("A=", repr(A), sep="\n")
print("B=", repr(B), sep="\n")

# stack arrays on a new axis
stacked = np.stack([A,B])
print("A shape: %s  ---  B shape: %s  ---  stacked shape: %s" % (A.shape, B.shape, stacked.shape))
print(stacked)
```

```
A=
array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])
B=
array([[10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19]])
A shape: (2, 5)  ---  B shape: (2, 5)  ---  stacked shape: (2, 2, 5)
[[[ 0  1  2  3  4]
  [ 5  6  7  8  9]]

 [[10 11 12 13 14]
  [15 16 17 18 19]]]
```

Menyambung arrays

```
catted = np.concatenate([A,B], axis=0)
print("A shape: %s  ---  B shape: %s  ---  catted shape (meow!): %s" % (A.shape, B.shape, catted.shape))
print(catted)
```

```
A shape: (2, 5)  ---  B shape: (2, 5)  ---  catted shape (meow!): (4, 5)
[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]
 [15 16 17 18 19]]
```

#2. PyTorch

```
data = [[1, 2], [3, 4]]
x_data = torch.tensor(data)
print(x_data, x_data.shape)
```

```
tensor([[1, 2],
        [3, 4]]) torch.Size([2, 2])
```

```
np_array = np.array(data)
x_np = torch.from_numpy(np_array)
```

```
print(x_np, x_np.shape)
```

```
tensor([[1, 2],
        [3, 4]], dtype=torch.int32) torch.Size([2, 2])
```

```
x_ones = torch.ones_like(x_data) # retains the properties of x_data
print(f"Ones Tensor: \n {x_ones} \n")

x_rand = torch.rand_like(x_data, dtype=torch.float) # overrides the datatype of x_data
print(f"Random Tensor: \n {x_rand} \n")
```

```
Ones Tensor:
 tensor([[1, 1],
        [1, 1]])

Random Tensor:
 tensor([[0.3945, 0.7436],
        [0.9964, 0.6925]])
```

```
shape = (2, 3,)
rand_tensor = torch.rand(shape)
ones_tensor = torch.ones(shape)
zeros_tensor = torch.zeros(shape)

print(f"Random Tensor: \n {rand_tensor} \n")
print(f"Ones Tensor: \n {ones_tensor} \n")
print(f"Zeros Tensor: \n {zeros_tensor}")
```

```
Random Tensor:
 tensor([[0.9196, 0.7228, 0.7443],
        [0.7374, 0.9749, 0.3799]])

Ones Tensor:
 tensor([[1., 1., 1.],
        [1., 1., 1.]])

Zeros Tensor:
 tensor([[0., 0., 0.],
        [0., 0., 0.]])
```

```
tensor = torch.rand(3, 4)

print(f"Shape of tensor: {tensor.shape}")
print(f"Datatype of tensor: {tensor.dtype}")
print(f"Device tensor is stored on: {tensor.device}")
```

```
Shape of tensor: torch.Size([3, 4])
Datatype of tensor: torch.float32
Device tensor is stored on: cpu
```

Diferensiasi menggunakan Autograd

PyTorch mempunyai mekanisme diferensiasi bawaan yang disebut dengan `torch.autograd`. Mari kita lihat bagaimana `autograd` mengumpulkan gradien. Buat 2 tensor `a` dan `b` dengan `requires_grad=True`. Ini memberi arahan bagi `autograd` bahwa setiap operasi terhadap tensor ini harus dilacak.

```
x = torch.rand(5, 5)
print(x)
y = torch.rand(5, 5)
z = torch.rand((5, 5), requires_grad=True)

a = x + y
print(f"Does `a` require gradients? : {a.requires_grad}")
b = x + z
print(f"Does `b` require gradients?: {b.requires_grad}")
```

```
tensor([[0.0648, 0.9199, 0.8221, 0.8119, 0.0685],
        [0.5371, 0.4856, 0.2043, 0.6096, 0.6139],
        [0.4206, 0.0072, 0.6887, 0.6801, 0.8029],
        [0.0842, 0.6448, 0.3777, 0.9436, 0.4110],
        [0.6014, 0.4895, 0.6297, 0.3384, 0.4816]])
Does `a` require gradients? : False
Does `b` require gradients?: True
```

```
a = torch.tensor([2., 3.], requires_grad=True)
b = torch.tensor([6., 4.], requires_grad=True)
```



```
print(a)
print(b)
```

```
tensor([2., 3.], requires_grad=True)
tensor([6., 4.], requires_grad=True)
```

```
print(a**3)
print(3*a**3)
print(b**2)
```

```
tensor([ 8., 27.], grad_fn=<PowBackward0>)
tensor([24., 81.], grad_fn=<MulBackward0>)
tensor([36., 16.], grad_fn=<PowBackward0>)
```

Kita membuat tensor lain `Q` dari `a` dan `b`.

$$Q = 3a^3 - b^2$$

```
Q = 3*a**3 - b**2
Q
```

```
tensor([-12.,  65.], grad_fn=<SubBackward0>)
```

Asumsikan `a` dan `b` sebagai parameter dari NN, dan `Q` eror. Di proses pelatihan NN, kita ingin gradien dari eror terhadap parameter, dimana:

$$\frac{\partial Q}{\partial a} = 9a^2$$
$$\frac{\partial Q}{\partial b} = -2b$$

Ketika kita memanggil `.backward()` pada `Q`, autograd menghitung gradien ini dan. menyimpannya dalam atribut `.grad` dari tensor tersebut.

Kita harus menggunakan argumen `gradient` di `Q.backward()` karena ini adalah sebuah vektor. `gradient` adalah sebuah tensor dari bentuk yang sama dengan `Q`, dan merepresentasikan gradien dari Q terhadap dirinya sendiri, dimana:

$$\frac{dQ}{dQ} = 1$$

Kita dapat pula agregasi Q menjadi sebuah skalar dan memanggil backward secara implisit, seperti `Q.sum().backward()`.

```
external_grad = torch.tensor([1., 1.])
Q.backward(gradient=external_grad)
```

Gradien sekarang telah disimpan dalam `a.grad` dan `b.grad`

```
# cek apakah gradien yang tersimpan terhitung benar
print(9*a**2 == a.grad)
print(-2*b == b.grad)
```

```
tensor([True, True])
tensor([True, True])
```

Buatlah loss function sebagai berikut:

$$Y = \Sigma ln(x)$$

Gunakan autograd untuk menghitung gradiennya terhadap parameter x

Hitung turunannya secara analitik dan bandingkan

```
x = torch.tensor([2., 3.], requires_grad=True)
Y = torch.sum(torch.log(x))
# backward pass
Y.backward()
x.grad
```

```
tensor([0.5000, 0.3333])
```

To find the gradient of the function

$$Y = \Sigma ln(x)$$

, we need to determine the derivative of (Y) with respect to (x).

Assuming that the sum is taken over some finite set of values of (x), we can write the function as:

$$Y = \ln(x_1) + \ln(x_2) + \ldots + \ln(x_n)$$

where (x_1, x_2, ..., x_n) are the values of (x) over which the sum is taken.

Now, let's find the derivative of (Y) with respect to (x):

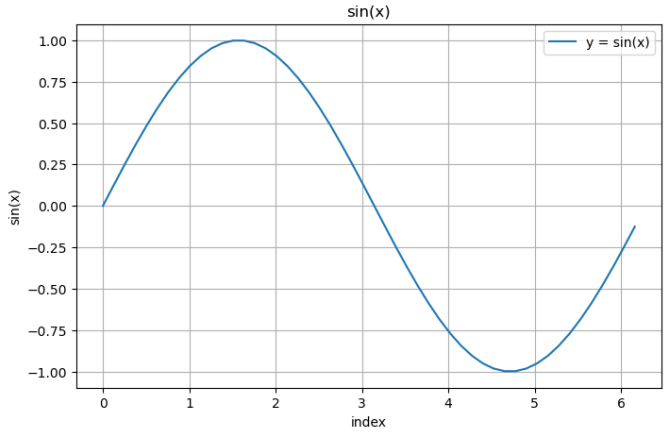
$$\begin{aligned}\frac{dY}{dx} &= \frac{d}{dx}(\ln(x_1) + \ln(x_2) + \ldots + \ln(x_n)) \\ &= \frac{d}{dx}\ln(x_1) + \frac{d}{dx}\ln(x_2) + \ldots + \frac{d}{dx}\ln(x_n) \quad \text{(using linearity of the derivative)} \\ &= \frac{1}{x_1} + \frac{1}{x_2} + \ldots + \frac{1}{x_n} \quad (\text{since } \frac{d}{dx}\ln(x) = \frac{1}{x}) \\ &= \sum \frac{1}{x_i}\end{aligned}$$

#3. Matplotlib

Menggambar grafik garis

Terhadap indeks

```
x = np.arange(50) * 2 * np.pi / 50
y = np.sin(x)
fig = plt.figure(figsize=(8, 5)) # create a figure, just like in matlab
ax = fig.add_subplot(1, 1 ,1) # create a subplot of certain size
ax.plot(x, y, label="y = sin(x)")
ax.set_xlabel('index')
ax.set_ylabel("sin(x)")
ax.set_title("sin(x)")
ax.grid()
ax.legend()
plt.show()
```



*argsint, (int, int, index), or SubplotSpec, default: (1, 1, 1) Three integers (nrows, ncols, index). The subplot will take the index position on a grid with nrows rows and ncols columns. index starts at 1 in the upper left corner and increases to the right. index can also be a two-tuple specifying the (first, last) indices (1-based, and including last) of the subplot, e.g., fig.add_subplot(3, 1, (1, 2)) makes a subplot that spans the upper 2/3 of the figure.

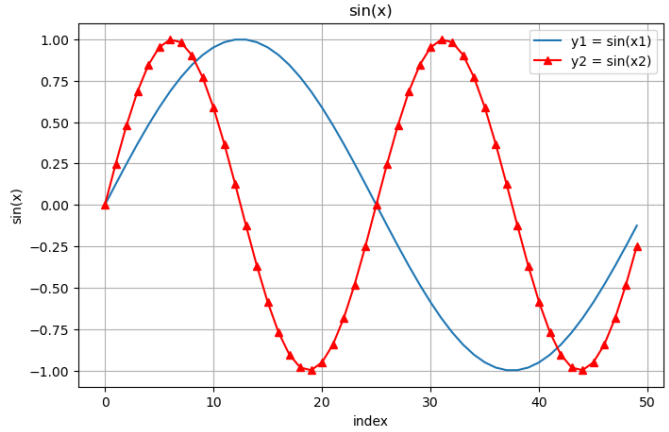
Beberapa Garis

```
x2 = np.arange(50) * 2 * np.pi / 25
y2 = np.sin(x2)
fig = plt.figure(figsize=(8, 5)) # create a figure, just like in matlab
ax = fig.add_subplot(1, 1 ,1) # create a subplot of certain size
ax.plot(y, label="y1 = sin(x1)")
ax.plot(y2, 'r-^', label="y2 = sin(x2)")
```



```
ax.set_xlabel('index')
ax.set_ylabel("sin(x)")
ax.set_title("sin(x)")
ax.grid()
ax.legend()
```

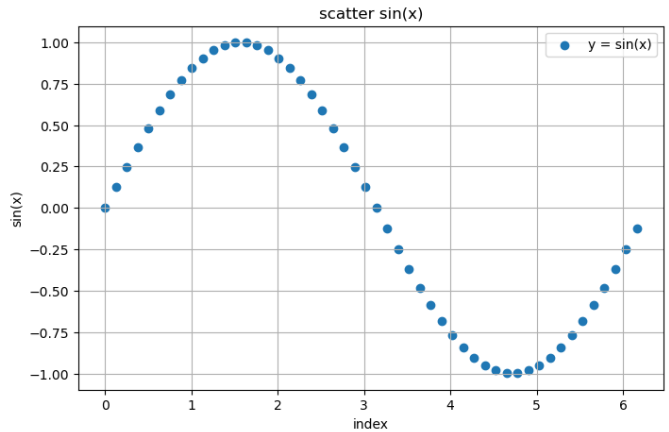
```
<matplotlib.legend.Legend at 0x1af9b21c2e0>
```



Grafik Scatter

```
x = np.arange(50) * 2 * np.pi / 50
y = np.sin(x)
fig = plt.figure(figsize=(8, 5)) # create a figure, just like in matlab
ax = fig.add_subplot(1, 1 ,1) # create a subplot of certain size
ax.scatter(x, y, label="y = sin(x)")
ax.set_xlabel('index')
ax.set_ylabel("sin(x)")
ax.set_title("scatter sin(x)")
ax.grid()
ax.legend()
```

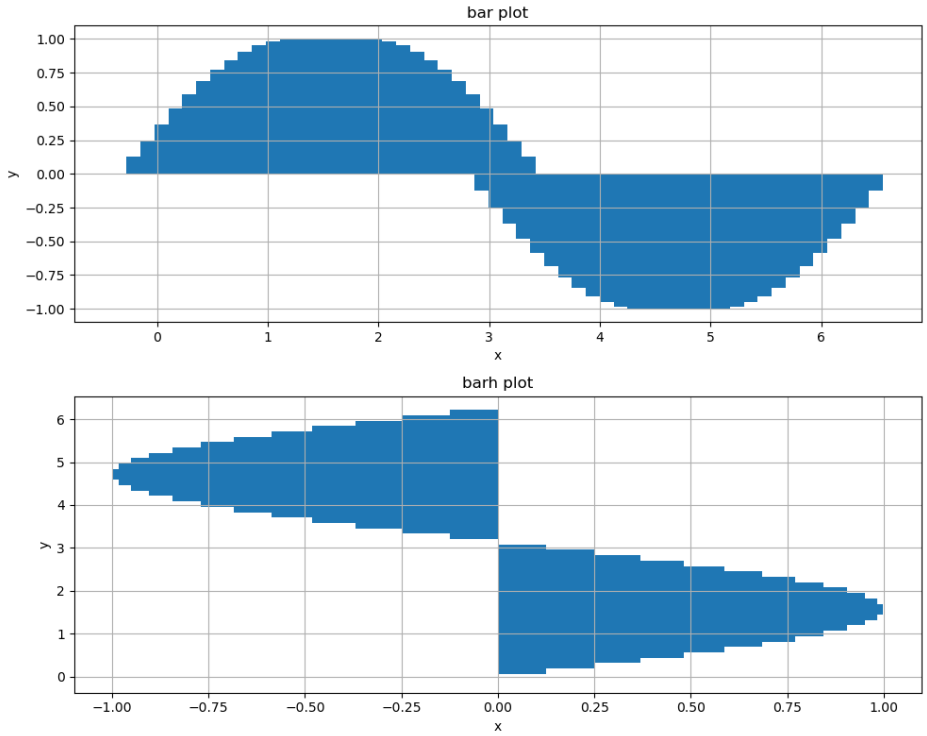
```
<matplotlib.legend.Legend at 0x1af9b29fd00>
```



Grafik Batang

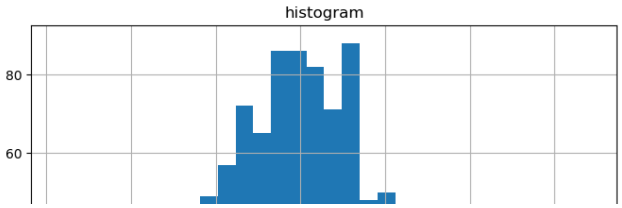
```
fig = plt.figure(figsize=(10, 8)) # create a figure, just like in matlab
ax1 = fig.add_subplot(2, 1 ,1) # create a subplot of certain size
```

```
ax1.bar(x, y)
ax1.set_xlabel('x')
ax1.set_ylabel("y")
ax1.set_title("bar plot")
ax1.grid()

ax2 = fig.add_subplot(2, 1 ,2) # create a subplot of certain size
ax2.barh(x, y, height=x[1]-x[0])
ax2.set_xlabel('x')
ax2.set_ylabel("y")
ax2.set_title("barh plot")
ax2.grid()

plt.tight_layout()
```



Histogram

```
fig = plt.figure(figsize=(8, 5)) # create a figure, just like in matlab
ax = fig.add_subplot(1, 1 ,1) # create a subplot of certain size
ax.hist(np.random.randn(1000), 30) # 30 is the number of bins
ax.set_title("histogram")
ax.grid()
```

#4. OpenCV

Membaca

```
apple_image = cv2.imread("./imgs/apple.jpeg")
```

gambar di OpenCV berbentuk numpy arrays

```
type(apple_image), apple_image.shape, apple_image.dtype
```

```
(numpy.ndarray, (1067, 800, 3), dtype('uint8'))
```

Channels dan Formats

```
apple_image.shape
```

```
(1067, 800, 3)
```

```
apple_image[0,0] # Get the pixel located at (0,0) from the top left
```

```
array([0, 0, 0], dtype=uint8)
```

```
apple_image_rgb = cv2.cvtColor(apple_image, cv2.COLOR_BGR2RGB)
print('bgr',apple_image)
print('rgb',apple_image_rgb)
```

```
...
 [0 0 0]
 [0 0 0]
 [0 0 0]]

[[0 0 0]
 [0 0 0]
 [0 0 0]
 ...
 [0 0 0]
 [0 0 0]
 [0 0 0]]

[[0 0 0]
 [0 0 0]
 [0 0 0]
 ...
 [0 0 0]
 [0 0 0]
 [0 0 0]]]
```

Menampilkan Gambar

```
fig = plt.figure(figsize=(8, 5)) # create a figure, just like in matlab
ax = fig.add_subplot(1, 1 ,1) # create a subplot of certain size
ax.imshow(apple_image_rgb)
ax.set_title("apple")
ax.set_axis_off()
```

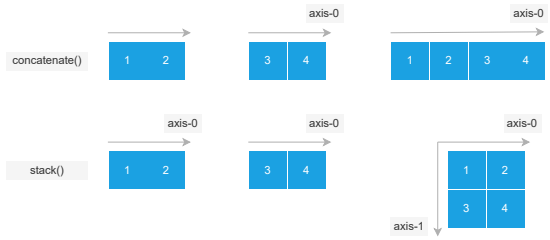
Mengubah images

Mengubah Warna

```
apple_gray = cv2.cvtColor(apple_image, cv2.COLOR_BGR2GRAY)
fig = plt.figure(figsize=(8, 5)) # create a figure, just like in matlab
ax = fig.add_subplot(1, 1 ,1) # create a subplot of certain size
ax.imshow(apple_gray, cmap="gray")
ax.set_title("apple hitam putih")
ax.set_axis_off()
```

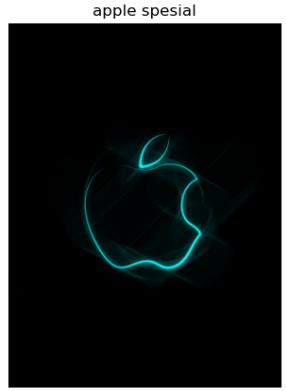


```
from IPython.display import Image
img_url = "https://www.pythontutorial.net/wp-content/uploads/2022/08/numpy-stack-vs-concatenate.svg"
Image(url=img_url)
```




```
empty_arr = np.zeros(apple_gray.shape, dtype=np.uint8)
print(f'empty_arr: {empty_arr}')
# stack them, making the 3rd axis
# Purple
special_apple = np.stack([ apple_gray,empty_arr,apple_gray ], axis=2)
# Green Blue
special_apple = np.stack([ empty_arr,apple_gray,apple_gray ], axis=2)
# Yellow
# special_apple = np.stack([ apple_gray,apple_gray,empty_arr ], axis=2)
print("created image of shape", special_apple.shape)

fig = plt.figure(figsize=(8, 5)) # create a figure, just like in matlab
ax = fig.add_subplot(1, 1, 1) # create a subplot of certain size
ax.imshow(special_apple)
ax.set_title("apple spesial")
ax.set_axis_off()
```

```
empty_arr: [[0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
 ...
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]]
created image of shape (1067, 800, 3)
```



Mengubah Ukuran

```
image_height, image_width, image_num_channels = special_apple.shape
new_height = image_height * 4
new_width = image_width * 4

# Resize it to 3x the width, and 3x the height, so we expect some distortion.
# (To display it in the browser, the image is being scaled down anyway, so resizing it 2 x 2 will not be obvious)

bigger_special_apple = cv2.resize(special_apple, (new_width, new_height))
print("resized to image of shape", bigger_special_apple.shape)

fig = plt.figure(figsize=(15, 15)) # create a figure, just like in matlab
ax = fig.add_subplot(1, 1, 1) # create a subplot of certain size
ax.imshow(bigger_special_apple)
ax.set_title("apple lebih besar")
ax.set_axis_off()
```

```
resized to image of shape (4268, 3200, 3)
```

Menulis ke File atau Variabel

```
output_path = "./imgs/output_apple.png"
cv2.imwrite(output_path, bigger_special_apple)
```

```
True
```

```
test_read_output = cv2.imread(output_path)
print("Read file of shape:", test_read_output.shape, "type",test_read_output.dtype)
fig = plt.figure(figsize=(20, 15)) # create a figure, just like in matlab
ax = fig.add_subplot(1, 1, 1) # create a subplot of certain size
ax.imshow(test_read_output)
ax.set_title("test_read_output")
ax.set_axis_off()
```

Video

Objek `VideoCapture` akan membaca video dari webcams, dan files.

Kita dapat menggunakan objek `VideoWriter` untuk menuliskan video ke dalam file.

↳ 2 cells hidden

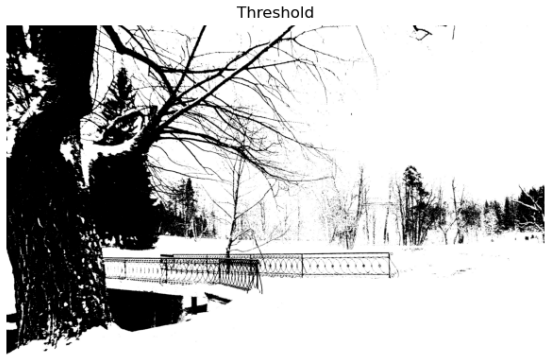
Binary thresholding

Binary thresholding pada gambar hitam putih

```
help(cv2.threshold)
```

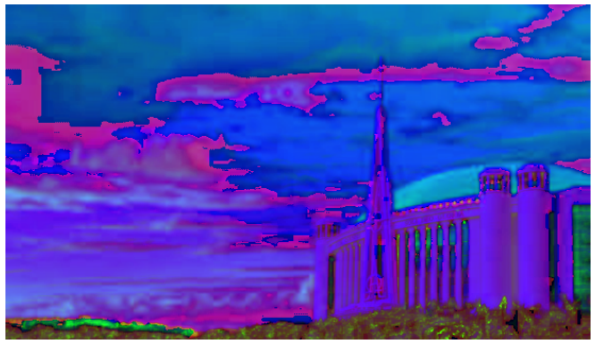
```
Help on built-in function threshold:

threshold(...)
    threshold(src, thresh, maxval, type[, dst]) -> retval, dst
    .   @brief Applies a fixed-level threshold to each array element.
    .
    .   The function applies fixed-level thresholding to a multiple-channel array. The function is typically
    .   used to get a bi-level (binary) image out of a grayscale image ( #compare could be also used for
    .   this purpose) or for removing a noise, that is, filtering out pixels with too small or too large
    .   values. There are several types of thresholding supported by the function. They are determined by
    .   type parameter.
    .
    .   Also, the special values #THRESH_OTSU or #THRESH_TRIANGLE may be combined with one of the
    .   above values. In these cases, the function determines the optimal threshold value using the Otsu's
    .   or Triangle algorithm and uses it instead of the specified thresh.
    .
    .   @note Currently, the Otsu's and Triangle methods are implemented only for 8-bit single-channel images.
    .
    .   @param src input array (multiple-channel, 8-bit or 32-bit floating point).
    .   @param dst output array of the same size  and type and the same number of channels as src.
    .   @param thresh threshold value.
    .   @param maxval maximum value to use with the #THRESH_BINARY and #THRESH_BINARY_INV thresholding
    .   types.
    .   @param type thresholding type (see #ThresholdTypes).
    .   @return the computed threshold value if Otsu's or Triangle methods used.
    .
    .   @sa  adaptiveThreshold, findContours, compare, min, max
```

```
# threshold for grayscale image
_, threshold_img = cv2.threshold(gray_img, 60, 255, cv2.THRESH_BINARY)

threshold_img = cv2.cvtColor(threshold_img, cv2.COLOR_GRAY2RGB)
fig = plt.figure(figsize=(8, 5)) # create a figure, just like in matlab
ax = fig.add_subplot(1, 1 ,1) # create a subplot of certain size
ax.imshow(threshold_img, cmap="gray")
ax.set_title("Threshold")
ax.set_axis_off()
```

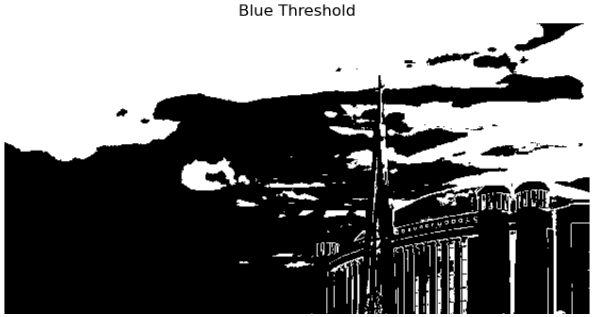


Binary thresholding pada gambar warna


```
img = cv2.imread('./imgs/grii-pusat.jpeg')
#img = cv2.imread('Pavlovsk_Railing_of_bridge_Yellow_palace_Winter.jpeg')

# convert image to RGB color for matplotlib
img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
img_hsv = cv2.cvtColor(img, cv2.COLOR_BGR2HSV)
fig = plt.figure(figsize=(8, 5)) # create a figure, just like in matlab
ax = fig.add_subplot(1, 1 ,1) # create a subplot of certain size
ax.imshow(cv2.cvtColor(img, cv2.COLOR_BGR2RGB))
ax.set_title("Image")
ax.set_axis_off()

fig = plt.figure(figsize=(8, 5)) # create a figure, just like in matlab
ax = fig.add_subplot(1, 1 ,1) # create a subplot of certain size
ax.imshow(img_hsv)
ax.set_title("Image")
ax.set_axis_off()
```




```
# threshold for hue channel in blue range
blue_min = np.array([85, 60, 60], np.uint8)
blue_max = np.array([150, 255, 255], np.uint8)
threshold_blue_img = cv2.inRange(img, blue_min, blue_max)

# show threshold bits
threshold_blue_img = cv2.cvtColor(threshold_blue_img, cv2.COLOR_GRAY2RGB)

fig = plt.figure(figsize=(8, 5))
ax = fig.add_subplot(1, 1 ,1)
ax.imshow(threshold_blue_img)
ax.set_title("Blue Threshold")
ax.set_axis_off()
```



Binary thresholding untuk masking

```
#img = cv2.imread('borobudur.jpeg')
img = cv2.imread('./imgs/prambanan.jpeg')
img_hsv = cv2.cvtColor(img, cv2.COLOR_BGR2HSV)

# mask out the sky
mask_inverse = cv2.inRange(img_hsv, blue_min, blue_max) # 1 for the sky
mask = cv2.bitwise_not(mask_inverse) # 0 for the sky

# apply the mask
# convert single channel mask back into 3 channels
mask_rgb = cv2.cvtColor(mask, cv2.COLOR_GRAY2RGB)

# perform bitwise and on mask to obtain cut-out image that is not blue
masked_img = cv2.bitwise_and(img, mask_rgb)

# replace the cut-out parts with white
masked_replace_white = cv2.addWeighted(masked_img, 1, \
                                       cv2.cvtColor(mask_inverse, cv2.COLOR_GRAY2RGB), 1, 0)

fig = plt.figure(figsize=(20, 10)) # create a figure, just like in matlab
ax = fig.add_subplot(1, 3 ,1)
ax.imshow(cv2.cvtColor(img_hsv, cv2.COLOR_HSV2RGB))
ax.set_title("Image")
ax.set_axis_off()

ax = fig.add_subplot(1, 3 ,2)
ax.imshow(cv2.cvtColor(mask, cv2.COLOR_GRAY2RGB))
ax.set_title("Mask")
ax.set_axis_off()

ax = fig.add_subplot(1, 3 ,3)
ax.imshow(cv2.cvtColor(masked_replace_white, cv2.COLOR_BGR2RGB))
ax.set_title("Masked Image")
ax.set_axis_off()

```

Gaussian Blur

```
# load a sample image
img = cv2.imread('./imgs/prambanan.jpeg')
fig = plt.figure(figsize=(16, 10)) # create a figure, just like in matlab
ax = fig.add_subplot(1, 1, 1) # create a subplot of certain size
```



```
ax.imshow(cv2.cvtColor(img, cv2.COLOR_BGR2RGB))
ax.set_title("Original Image")
ax.set_axis_off()
```


```
# preproccess with blurring, with 5x5 kernel
img_blur_small = cv2.GaussianBlur(img, (5,5), 25) # last parameter is the variance of the gaussian
fig = plt.figure(figsize=(16, 10)) # create a figure, just like in matlab
ax = fig.add_subplot(1, 1, 1) # create a subplot of certain size
ax.imshow(cv2.cvtColor(img_blur_small, cv2.COLOR_BGR2RGB))
ax.set_title("Blurred Image")
ax.set_axis_off()
```


```
# threshold on regular image
gray_img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
_, threshold_img = cv2.threshold(gray_img, 100, 255, cv2.THRESH_BINARY)

# threshold on blurred image
gray_blur_img = cv2.cvtColor(img_blur_small, cv2.COLOR_BGR2GRAY)
_, threshold_img_blur = cv2.threshold(gray_blur_img, 100, 255, cv2.THRESH_BINARY)

```

```
fig = plt.figure(figsize=(20, 10))
ax = fig.add_subplot(1, 2 ,1)
ax.imshow(cv2.cvtColor(threshold_img, cv2.COLOR_GRAY2RGB))
ax.set_title("Threshold Original")
ax.set_axis_off()

ax = fig.add_subplot(1, 2 ,2)
ax.imshow(cv2.cvtColor(threshold_img_blur, cv2.COLOR_GRAY2RGB))
ax.set_title("Threshold Blurred")
ax.set_axis_off()
```

```
    # imports for the tutorial
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image
import cv2
from scipy import signal
```

It looks like you've imported several Python libraries commonly used for image processing and manipulation. Here's a brief description of each import statement:

1. `import numpy as np`: NumPy is a fundamental package for scientific computing with Python. It provides support for arrays and matrices, as well as mathematical functions to operate on these arrays. The `as np` alias allows you to refer to NumPy functions with the shorthand "np."
2. `import matplotlib.pyplot as plt`: Matplotlib is a popular data visualization library in Python. The `pyplot` module from Matplotlib provides functions to create various types of plots and graphs. The alias `plt` is commonly used for convenience.
3. `from PIL import Image`: PIL (Python Imaging Library) is used for opening, manipulating, and saving various image file formats. It allows you to work with images in a variety of ways.
4. `import cv2`: OpenCV (Open Source Computer Vision Library) is a powerful library for computer vision tasks, including image and video processing. The `cv2` module provides a wide range of functions for working with images and videos.
5. `from scipy import signal`: SciPy is a library for scientific and technical computing. The `signal` submodule contains functions related to signal processing, including convolution and filtering.

```
# plot images function
def plot_images(image_list, title_list, subplot_shape=(1,1), axis='off', fontsize=30, figsize=(4,4), cmap=['gray'], c
    plt.figure(figsize=figsize)
    for ii, im in enumerate(image_list):
        c_title = title_list[ii]
        if len(cmap) > 1:
            c_cmap = cmap[ii]
        else:
            c_cmap = cmap[0]
        plt.subplot(subplot_shape[0], subplot_shape[1],ii+1)
        plt.imshow(im, cmap=c_cmap)
        plt.title(c_title, fontsize=fontsize)
        plt.axis(axis)
        if cbar:
            plt.colorbar()
```

The `plot_images` function you've provided is a useful utility for displaying a list of images with corresponding titles and various customizations. Here's a breakdown of the function's parameters and its purpose:

- `image_list`: This is a list of images you want to display.
- `title_list`: This is a list of titles for each image in `image_list`.
- `subplot_shape`: This parameter defines the shape of the subplot grid where the images will be displayed. It's a tuple with two values, specifying the number of rows and columns in the grid.
- `axis`: This parameter controls whether axis labels are displayed on the images. By default, it's set to 'off,' which means that axis labels are not displayed. You can change it to 'on' if you want to display axis labels.
- `fontsize`: The font size for the image titles.
- `figsize`: The size of the entire figure where the images will be plotted. It's a tuple specifying the width and height in inches.
- `cmap`: This is a list of colormaps to apply to the images. If you provide a single colormap, it will be applied to all images. If you provide a list of colormaps with the same length as `image_list`, each image will have its own colormap.
- `cbar`: If set to `True`, a colorbar will be displayed next to each image. A colorbar is a scale that shows the mapping between colors and data values in the image.

Here's how you can use this function:

```
# Example usage
image_list = [image1, image2, image3]  # List of your images
title_list = ["Image 1", "Image 2", "Image 3"]  # Titles for the images

# Plot the images in a 1x3 grid with gray colormap and colorbars
plot_images(image_list, title_list, subplot_shape=(1, 3), cmap=['gray', 'gray', 'gray'], cbar=True)
```

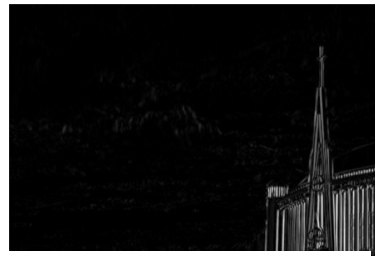


```
# Show the plot
plt.show()
```

This function allows you to quickly visualize multiple images in a customizable layout.

Filter Sobel

```
im = cv2.imread('./images/grii-pusat.jpeg')
im = cv2.cvtColor(im, cv2.COLOR_BGR2GRAY)
ksize = 3
im_x = np.abs(cv2.Sobel(im, cv2.CV_64F, 1, 0, ksize=ksize))
im_y = np.abs(cv2.Sobel(im, cv2.CV_64F, 0, 1, ksize=ksize))

plot_images([im,im_x,im_y], ['','',''], subplot_shape=(1,3), figsize=(30,20))
```



It seems like you're performing image gradient calculations using the Sobel operator and then attempting to visualize the original image, its horizontal gradient (along the x-axis), and its vertical gradient (along the y-axis) using the `plot_images` function you provided. However, it appears there's an issue with your usage of the function. The titles you've provided in the `title_list` are empty strings (''), which might lead to confusion in the visualization.

Here's an example of how you can modify your code to display these images with more informative titles:

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

# Load the image and convert it to grayscale
im = cv2.imread('grii-pusat.jpeg')
im = cv2.cvtColor(im, cv2.COLOR_BGR2GRAY)

# Calculate the image gradients using Sobel operator
ksize = 3
im_x = np.abs(cv2.Sobel(im, cv2.CV_64F, 1, 0, ksize=ksize))
im_y = np.abs(cv2.Sobel(im, cv2.CV_64F, 0, 1, ksize=ksize))

# Create informative titles for the images
titles = ['Original Image', 'Horizontal Gradient (im_x)', 'Vertical Gradient (im_y)']

# Plot the images with titles
plot_images([im, im_x, im_y], titles, subplot_shape=(1, 3), figsize=(30, 20))

# Show the plot
plt.show()
```

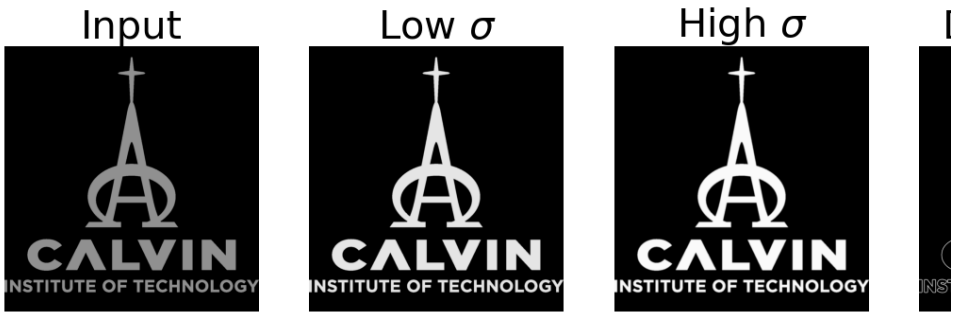
By providing informative titles in the `titles` list, you can better understand the purpose of each image in the visualization. This will make it easier to interpret the results.

DoG

```
# DoG example on opencv logo
img = cv2.imread('./images/CIT-logo.png')
img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

```

```
# Apply 3x3 and 7x7 Gaussian blur
low_sigma = cv2.GaussianBlur(img,(3,3),0)
high_sigma = cv2.GaussianBlur(img,(7,7),0)

print(f'Log Sigma: {low_sigma}')
print(f'High Sigma: {high_sigma}')

# Calculate the DoG by subtracting
dog = low_sigma - high_sigma
print(f'DoG: {dog}')

# show the result
plot_images([img,low_sigma,high_sigma,dog], ['Input','Low $\sigma$','High $\sigma$','DoG result'],
            subplot_shape=(1,4), figsize=(16,4))
```

```
Log Sigma: [[0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
 ...
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]]
High Sigma: [[0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
 ...
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]]
DoG: [[0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
 ...
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]]
```



It appears that you're applying the Difference of Gaussians (DoG) filter to an image, specifically the OpenCV logo in grayscale, to enhance certain features. Here's a breakdown of your code and how it works:

1. You load the 'CIT-logo.png' image and convert it to grayscale.
2. You apply Gaussian blurs with different kernel sizes to the grayscale image. `low_sigma` corresponds to a 3x3 Gaussian blur, while `high_sigma` corresponds to a 7x7 Gaussian blur. These blurs smooth the image at different scales.
3. You calculate the DoG image by subtracting the `high_sigma` blurred image from the `low_sigma` blurred image. This operation enhances edges and other high-frequency details in the image.
4. Finally, you use the `plot_images` function to display the original image, the two blurred versions, and the DoG result, each with its corresponding title.

Your code looks correct for applying the DoG filter and visualizing the results. When you run this code, you should see the input image, the low sigma blurred image, the high sigma blurred image, and the DoG result side by side in the visualization. The DoG result will highlight edges and features present in the image.

Find Edges with Convolution Kernels and Gradient

```
im_o = cv2.imread('./images/borobudur.jpeg')
im = cv2.cvtColor(im_o, cv2.COLOR_BGR2GRAY)
```

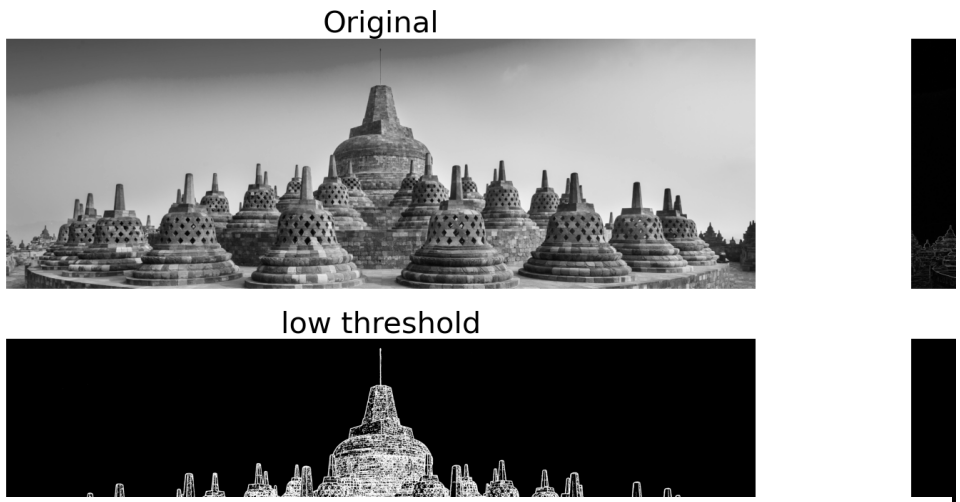


```
ksize = 5

# blur image
im = cv2.GaussianBlur(im, (ksize, ksize), 0)

# edge extraction:
k_x = np.array([[0, 0, 0], [-1, 0, 1], [0, 0, 0]])
k_y = np.array([[0, -1, 0], [0, 0, 0], [0, 1, 0]])
im_x = cv2.filter2D(im, cv2.CV_64F, k_x)
im_y = cv2.filter2D(im, cv2.CV_64F, k_y)

# gradient magnitude:
im_grad_mag = np.sqrt(np.square(im_x) + np.square(im_y))

low_thresh = im_grad_mag > 7 # low threshold
high_thresh = im_grad_mag > 30 # High threshold

# plot_images([im, im_x, im_y], ['Original', 'Deriv. X', 'Deriv. Y'], subplot_shape=(1,3), figsize=(18,6))
```

It appears that you're performing edge detection and gradient magnitude calculations on an image using custom convolution kernels and thresholds. Here's an explanation of the code:

1. You load the 'borobudur.jpeg' image and convert it to grayscale.
2. You apply Gaussian blur to the grayscale image using a 5x5 kernel size to reduce noise and make the edges more pronounced.
3. You define two convolution kernels `k_x` and `k_y` for edge detection. These kernels perform horizontal (x-direction) and vertical (y-direction) gradient calculations, respectively.
4. You use the `cv2.filter2D` function to convolve the blurred image with the `k_x` and `k_y` kernels, resulting in `im_x` and `im_y`, which represent the gradient in the x and y directions, respectively.
5. You calculate the gradient magnitude of the image using the Euclidean formula, `im_grad_mag = sqrt(im_x^2 + im_y^2)`. This represents the overall gradient magnitude at each pixel.
6. You define two thresholding conditions, `low_thresh` and `high_thresh`, to identify edges in the image. Pixels with gradient magnitudes above a certain threshold are considered edges. The `low_thresh` and `high_thresh` arrays contain boolean values indicating whether the gradient magnitude at each pixel meets the respective thresholds.

```
# figures:
plot_images([im, im_grad_mag, low_thresh, high_thresh], ['Original', 'Magnitude', 'low threshold', 'higher threshold'
            subplot_shape=(2,2), figsize=(30,10))
```



It looks like you've added code to plot the original image, gradient magnitude image, and thresholded images using the `plot_images` function. Here's an explanation of what this code does:

1. The `plot_images` function is called with four images:
   - `im`: The original grayscale image.

- `im_grad_mag`: The gradient magnitude image.
- `low_thresh`: The image resulting from the low threshold condition (boolean values indicating where the gradient magnitude is above the low threshold).
- `high_thresh`: The image resulting from the high threshold condition (boolean values indicating where the gradient magnitude is above the high threshold).

2. You've provided corresponding titles for each image in the `title_list`.
3. `subplot_shape` is set to (2, 2), indicating a 2x2 grid for arranging the four images.
4. `figsize` specifies the size of the entire figure where the images will be plotted.
5. Finally, `plt.show()` is used to display the figure containing the images with their titles.

Canny Edge Detection

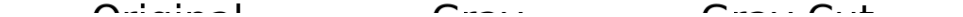
```
# Load Image
img = cv2.imread('./images/monas.jpeg')

# convert to RGB for matplotlib
img_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

# convert to gray and crop for canny
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
in_x_min = 100
in_x_max = 700
in_y_min = 30
in_y_max = 850
gray_cut = gray[in_y_min:in_y_max, in_x_min:in_x_max]
```

In the provided code, you're loading an image, converting it to RGB color space, and then converting it to grayscale. Additionally, you're cropping the grayscale image for further processing using the Canny edge detection algorithm. Here's a breakdown of the code:

1. `img = cv2.imread('monas.jpeg')`: This line loads an image named 'monas.jpeg' using OpenCV's `imread` function. The resulting `img` variable contains the image in the BGR color space (commonly used by OpenCV).
2. `img_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)`: This line converts the loaded image from BGR to RGB color space. Converting to RGB is often done when you plan to display the image using matplotlib, as matplotlib expects images in RGB format for proper display.
3. `gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)`: This line converts the original BGR image to grayscale using OpenCV's `cvtColor` function. Grayscale images have a single channel, which is useful for various image processing tasks.
4. Cropping the Grayscale Image:
   - `in_x_min`, `in_x_max`, `in_y_min`, and `in_y_max` define the region of interest (ROI) for cropping the grayscale image. These values specify the minimum and maximum x and y coordinates for the cropping rectangle.
   - `gray_cut = gray[in_y_min:in_y_max, in_x_min:in_x_max]`: This line crops the grayscale image based on the specified coordinates, resulting in a smaller region of the image that you can use for further processing.

After this code, you can perform various image processing operations on the `gray_cut` image, such as edge detection using the Canny algorithm or any other operations you may need for your specific application.

```
plot_images([img_rgb, gray, gray_cut], ['Original', 'Gray', 'Gray Cut'], cmap=[None, 'gray', 'gray'],
            subplot_shape=(1, 3), figsize=(12, 4))
```

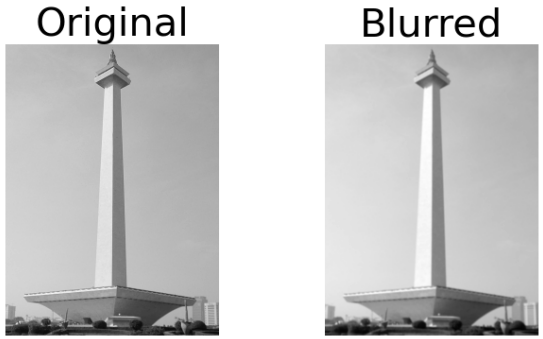



It seems like you want to visualize the original RGB image, the grayscale version of the image, and the cropped grayscale image using the `plot_images` function. Here's a breakdown of how this code works:

1. `plot_images`: This is the custom function you defined earlier for plotting images with titles and other customizations.
2. `img_rgb`: The original RGB image converted to RGB color space.
3. `gray`: The grayscale version of the original image.
4. `gray_cut`: The cropped grayscale image, which is a region of interest (ROI) from the original grayscale image.
5. `['Original', 'Gray', 'Gray Cut']`: This list provides titles for each of the three images you want to display.
6. `cmap`: This list specifies the colormaps to be used for displaying the images. The original RGB image doesn't require a colormap, so it's set to `None`. The grayscale images are displayed using the 'gray' colormap, which is suitable for grayscale images.
7. `subplot_shape`: It's set to (1, 3), indicating that you want to arrange the three images in a single row with three columns.
8. `figsize`: This parameter sets the size of the entire figure where the images will be plotted.

Canny Detection - Step 1 - Blur

```
# Gaussian Blurring
blur = cv2.GaussianBlur(gray_cut, (9,9), 0)
plot_images([gray_cut, blur], ['Original','Blurred'], cmap=['gray', 'gray'], subplot_shape=(1,2), figsize=(8,4))
```



In this code snippet, you are applying Gaussian blurring to the cropped grayscale image (`gray_cut`) using OpenCV's `cv2.GaussianBlur` function and then visualizing both the original and the blurred images. Here's an explanation of the code:

1. `blur = cv2.GaussianBlur(gray_cut, (9, 9), 0)`: This line applies Gaussian blur to the `gray_cut` image using a Gaussian kernel with a size of `(9, 9)` and a standard deviation of `0`. The resulting `blur` image is the smoothed version of the original cropped grayscale image.
2. `plot_images`: You are using your custom `plot_images` function to visualize the original (`gray_cut`) and the blurred (`blur`) images side by side.
3. `['Original', 'Blurred']`: These are the titles for the two images being displayed.
4. `cmap=['gray', 'gray']`: Since both images are grayscale, you specify the 'gray' colormap for both.
5. `subplot_shape=(1, 2)`: This parameter sets up a subplot grid with one row and two columns, allowing you to display the original and blurred images side by side.
6. `figsize=(8, 4)`: It defines the size of the figure where the images will be displayed.

Canny Detection - Step 2 - Find Magnitude and Orientation of Gradient

```
# Apply Sobel:
sobelx_64 = cv2.Sobel(blur, cv2.CV_64F, 1, 0, ksize=3)
sobely_64 = cv2.Sobel(blur, cv2.CV_64F, 0, 1, ksize=3)

# From gradients calculate the magnitude and changing
mag = np.hypot(sobelx_64, sobely_64)
mag = mag / mag.max()
```

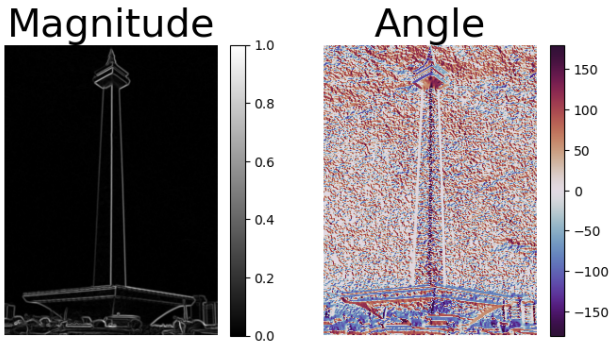
```
 8
 9 # Find the direction and change it to degree
10 theta = np.arctan2(sobely_64, sobelx_64)
11 angle = np.rad2deg(theta)
```

In this code snippet, you are applying the Sobel operator to the blurred image to calculate gradient information, including gradient magnitude and gradient direction (angle). Here's an explanation of the code:

1. `sobelx_64 = cv2.Sobel(blur, cv2.CV_64F, 1, 0, ksize=3)` and `sobely_64 = cv2.Sobel(blur, cv2.CV_64F, 0, 1, ksize=3)`: These lines use the `cv2.Sobel` function to calculate the horizontal gradient (`sobelx_64`) and vertical gradient (`sobely_64`) of the blurred image (`blur`) using a Sobel kernel of size 3x3. The gradients are calculated in the x and y directions.
2. `mag = np.hypot(sobelx_64, sobely_64)`: This line calculates the gradient magnitude (`mag`) at each pixel using the `np.hypot` function. The gradient magnitude represents the strength of the gradient at each pixel.
3. `mag = mag / mag.max()`: This normalizes the gradient magnitude values to a range between 0 and 1 by dividing them by the maximum magnitude value. This normalization is often done for better visualization or further processing.
4. `theta = np.arctan2(sobely_64, sobelx_64)`: This line calculates the gradient direction (`theta`) at each pixel using the `np.arctan2` function. The gradient direction is represented as an angle in radians.
5. `angle = np.rad2deg(theta)`: This converts the gradient directions from radians to degrees, resulting in the `angle` variable, which represents the gradient directions in degrees.

Now, you have calculated both the gradient magnitude and gradient direction for the blurred image. These gradient values can be useful for various image processing tasks, such as edge detection and feature extraction.

```
1 plot_images([mag, angle], ['Magnitude', 'Angle'], subplot_shape=(1,2), figsize=(8,4), cmap=['gray', 'twilight_shifted
2              cbar=True)
```

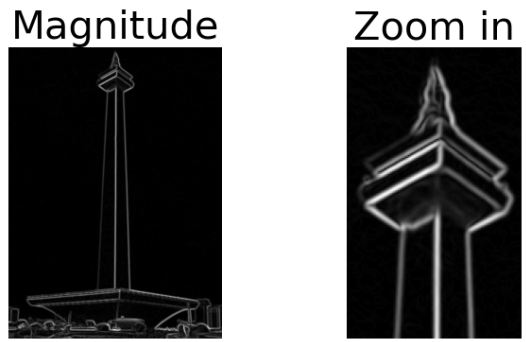


In this code snippet, you're using the `plot_images` function to visualize the gradient magnitude (`mag`) and gradient direction (`angle`) calculated from the Sobel operator. Here's a breakdown of the code:

1. `plot_images`: You are using your custom `plot_images` function to display two images side by side.
2. `[mag, angle]`: These are the two images you want to display. `mag` represents the gradient magnitude, and `angle` represents the gradient direction.
3. `['Magnitude', 'Angle']`: These are the titles for the two images being displayed.
4. `subplot_shape=(1, 2)`: This parameter sets up a subplot grid with one row and two columns, allowing you to display the magnitude and angle images side by side.
5. `figsize=(8, 4)`: It defines the size of the figure where the images will be displayed.
6. `cmap=['gray', 'twilight_shifted']`: You specify the colormaps to be used for displaying the images. The gradient magnitude (`mag`) is displayed using the 'gray' colormap, and the gradient direction (`angle`) is displayed using the 'twilight_shifted' colormap.
7. `cbar=True`: This parameter adds colorbars to the images, which can provide information about the color mapping of pixel values to colors.

Canny Detection - Step 3 - Localization


```
1 # Zoom in at the problem (ridge instead of edge)
2 in_x_min, in_x_max, in_y_min, in_y_max = (240, 360, 0, 200)
3 mag_cut = mag[in_y_min:in_y_max, in_x_min:in_x_max]
4 # ang_cut = mag[in_y_min:in_y_max, in_x_min:in_x_max]
5 plot_images([mag, mag_cut], ['Magnitude','Zoom in'], subplot_shape=(1,2), figsize=(8,4))
```



In this code snippet, you are zooming in on a specific region of interest (ROI) within the gradient magnitude image (`mag`) to examine a problem related to ridges. Here's an explanation of the code:

1. `in_x_min, in_x_max, in_y_min, in_y_max = (240, 360, 0, 200)`: These variables define the coordinates for the region of interest (ROI) that you want to zoom in on. Specifically:
   - `in_x_min` and `in_x_max` determine the horizontal (x-axis) boundaries of the ROI.
   - `in_y_min` and `in_y_max` determine the vertical (y-axis) boundaries of the ROI.
2. `mag_cut = mag[in_y_min:in_y_max, in_x_min:in_x_max]`: This line creates a new image (`mag_cut`) by cropping the gradient magnitude image (`mag`) based on the specified ROI coordinates. You are extracting a specific area of the image for closer examination.
3. `plot_images`: You are using the `plot_images` function to display two images side by side:
   - The original gradient magnitude image (`mag`) with the title "Magnitude."
   - The cropped ROI (`mag_cut`) with the title "Zoom in."
4. `subplot_shape=(1, 2)`: This parameter sets up a subplot grid with one row and two columns, allowing you to display the original and zoomed-in images side by side.
5. `figsize=(8, 4)`: It defines the size of the figure where the images will be displayed.

Two images side by side will be shown:

- The original gradient magnitude image with the title "Magnitude."
- A zoomed-in view of the gradient magnitude image, focusing on the specific ROI, with the title "Zoom in."

This visualization allows you to closely examine the gradient magnitude in the selected region of interest, which can be helpful for identifying and understanding features or patterns related to ridges in the image.

Canny Detection - Step 3 - Non-Maximum Suppression (NMS)

- Periksa apakah pixel berada pada maxima lokal sepanjang arah gradien, lalu pilih satu maximum sepanjang lebar dari batas
- Memerlukan pemeriksaan interpolasi pixel p dan r

```
 1 # Find the neighbouring pixels (b,c) in the rounded gradient direction
 2 # and then apply non-max suppression
 3 M, N = mag.shape
 4 Non_max = np.zeros((M,N), dtype= np.float64)
 5
 6 for i in range(1,M-1):
 7     for j in range(1,N-1):
 8         # Horizontal 0
 9         if (0 <= angle[i,j] < 22.5) or (157.5 <= angle[i,j] <= 180) or (-22.5 <= angle[i,j] < 0)\
10         or (-180 <= angle[i,j] < -157.5):
11             b = mag[i, j+1]
12             c = mag[i, j-1]
13         # Diagonal 45
14         elif (22.5 <= angle[i,j] < 67.5) or (-157.5 <= angle[i,j] < -112.5):
15             b = mag[i+1, j+1]
```

```
            c = mag[i-1, j-1]
        # Vertical 90
        elif (67.5 <= angle[i,j] < 112.5) or (-112.5 <= angle[i,j] < -67.5):
            b = mag[i+1, j]
            c = mag[i-1, j]
        # Diagonal 135
        elif (112.5 <= angle[i,j] < 157.5) or (-67.5 <= angle[i,j] < -22.5):
            b = mag[i+1, j-1]
            c = mag[i-1, j+1]

        # Non-max Suppression
        if (mag[i,j] >= b) and (mag[i,j] >= c):
            Non_max[i,j] = mag[i,j]
        else:
            Non_max[i,j] = 0
```

In this code snippet, you are performing non-maximum suppression on the gradient magnitude image (`mag`) based on the gradient direction (`angle`). Non-maximum suppression is a technique used in edge detection to thin out detected edges and keep only the strongest ones. Here's an explanation of the code:

1. `M` and `N` are used to get the dimensions of the gradient magnitude image (`mag`), assuming `mag` is a NumPy array.
2. `Non_max` is initialized as an empty array of the same dimensions as `mag` to store the results of non-maximum suppression.
3. Two nested loops iterate over the rows and columns of the gradient magnitude image, excluding the border pixels (i.e., from index 1 to M-2 and 1 to N-2) because non-maximum suppression is not typically applied to border pixels.
4. Based on the gradient direction at each pixel (`angle[i, j]`), you determine the neighboring pixels `b` and `c`. The choice of neighboring pixels depends on the angle:
   - If the angle is close to horizontal (0 degrees or 180 degrees), you choose the pixels to the left and right: `b = mag[i, j+1]` and `c = mag[i, j-1]`.
   - If the angle is close to 45 degrees, you choose the pixels diagonally: `b = mag[i+1, j+1]` and `c = mag[i-1, j-1]`.
   - If the angle is close to vertical (90 degrees or -90 degrees), you choose the pixels above and below: `b = mag[i+1, j]` and `c = mag[i-1, j]`.
   - If the angle is close to 135 degrees, you choose the pixels diagonally in the opposite direction: `b = mag[i+1, j-1]` and `c = mag[i-1, j+1]`.
5. Non-maximum suppression is then applied as follows:
   - If the magnitude at the current pixel (`mag[i, j]`) is greater than or equal to both neighboring magnitudes (`b` and `c`), the current pixel's value is preserved in the `Non_max` image.
   - Otherwise, the current pixel's value is set to 0 in the `Non_max` image.

This process ensures that only local maxima in the gradient magnitude image are retained, which typically corresponds to edges in the image. The non-maximum suppression step is an essential part of the Canny edge detection algorithm.

After running this code, you will have an image `Non_max` where non-maximum suppression has been applied, retaining only the strongest edges in the image.

```
mag_cut = mag[in_y_min:in_y_max, in_x_min:in_x_max]
Non_max_cut = Non_max[in_y_min:in_y_max, in_x_min:in_x_max]
plot_images([mag, Non_max, mag_cut, Non_max_cut], ['Magnitude', 'Non-max', 'Magnitude', 'Non-max'],
            subplot_shape=(1,4), figsize=(16,8))
```


In this code snippet, you are visualizing the results of the gradient magnitude (`mag`) and non-maximum suppression (`Non_max`) both for the entire image and a zoomed-in region of interest. Here's an explanation of the code:

1. `mag_cut` and `Non_max_cut` are created by cropping the gradient magnitude image (`mag`) and the non-maximum suppression result (`Non_max`) based on the specified region of interest (ROI) coordinates `in_x_min`, `in_x_max`, `in_y_min`, and `in_y_max`. These cropped images focus on a specific area of the original image for closer examination.
2. The `plot_images` function is used to display four images side by side:
   - The original gradient magnitude image (`mag`) with the title "Magnitude."
   - The non-maximum suppression result (`Non_max`) with the title "Non-max."
   - The cropped gradient magnitude image (`mag_cut`) with the title "Magnitude."
   - The cropped non-maximum suppression result (`Non_max_cut`) with the title "Non-max."
3. `subplot_shape=(1, 4)`: This parameter sets up a subplot grid with one row and four columns, allowing you to display all four images side by side.
4. `figsize=(16, 8)`: It defines the size of the figure where the images will be displayed.

Canny Detection - Step 4 - Hysteresis Thresholding

- Periksa apakah nilai maksimum dari gradien sudah cukup besar
  - Dropouts? gunakan histeresis
    - Gunakan threshold besar untuk memulai kurva batas dan threshold kecil untuk melanjutkan

```
# Hysteresis - step 1 - two tresholds
# Set high and low threshold
highThreshold = 40 / 255
lowThreshold = 10 / 255

M, N = Non_max.shape
out = np.zeros((M,N), dtype= np.float64)

# If edge intensity is greater than 'High' it is a sure-edge
# below 'low' threshold, it is a sure non-edge
strong_i, strong_j = np.where(Non_max >= highThreshold)
zeros_i, zeros_j = np.where(Non_max < lowThreshold)

# weak edges
weak_i, weak_j = np.where((Non_max <= highThreshold) & (Non_max >= lowThreshold))

# Set same intensity value for all edge pixels
out[strong_i, strong_j] = 255
out[zeros_i, zeros_j ] = 0
out[weak_i, weak_j] = 75
```

In this code snippet, you are performing the first step of hysteresis thresholding to classify pixels into three categories: strong edges, weak edges, and non-edges. Here's an explanation of the code:

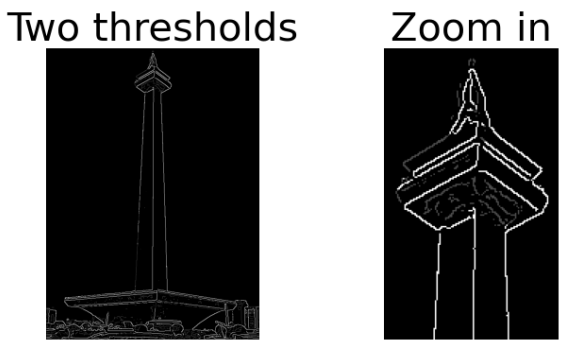
1. `highThreshold` and `lowThreshold` are defined as threshold values. These values determine the intensity levels used to classify pixels into different categories. Pixels with intensity values greater than or equal to `highThreshold` are considered "strong edges," and pixels with intensity values less than `lowThreshold` are considered "non-edges." Pixels with intensity values between `lowThreshold` and `highThreshold` are considered "weak edges."
2. `M` and `N` are used to get the dimensions of the `Non_max` image, assuming `Non_max` is a NumPy array.
3. `out` is initialized as an empty array of the same dimensions as `Non_max` to store the results of the hysteresis thresholding.
4. `strong_i` and `strong_j` store the indices of pixels in the `Non_max` image where the intensity is greater than or equal to `highThreshold`. These are the "strong edge" pixels.
5. `zeros_i` and `zeros_j` store the indices of pixels in the `Non_max` image where the intensity is less than `lowThreshold`. These are the "non-edge" pixels.

6. `weak_i` and `weak_j` store the indices of pixels in the `Non_max` image where the intensity is between `lowThreshold` and `highThreshold`. These are the "weak edge" pixels.
7. The following assignments are made to the `out` image:
    - Pixels classified as strong edges (`strong_i` and `strong_j`) are set to an intensity value of 255, indicating strong edges.
    - Pixels classified as non-edges (`zeros_i` and `zeros_j`) are set to an intensity value of 0, indicating non-edges.
    - Pixels classified as weak edges (`weak_i` and `weak_j`) are set to an intensity value of 75, which is typically used to represent weak edges.

This step separates the pixels into different categories based on their intensity values and prepares them for the second step of hysteresis thresholding, where weak edges can be further classified as strong edges or non-edges based on their connectivity to strong edges.

After running this code, you will have an image (`out`) where pixels are categorized as strong edges, weak edges, or non-edges based on the defined thresholds.

```
plot_images([out, out[in_y_min:in_y_max, in_x_min:in_x_max]], ['Two thresholds', 'Zoom in'],
            subplot_shape=(1,2), figsize=(8,4))
```



In this code snippet, you are visualizing the results of the hysteresis thresholding, specifically the categorization of pixels into strong edges, weak edges, and non-edges. Here's an explanation of the code:

1. `plot_images`: You are using the `plot_images` function to display two images side by side.
2. `[out, out[in_y_min:in_y_max, in_x_min:in_x_max]]`: These are the two images you want to display.
    - `out` represents the result of hysteresis thresholding for the entire image, where pixels are categorized into strong edges (255), weak edges (75), or non-edges (0) based on the defined thresholds.
    - `out[in_y_min:in_y_max, in_x_min:in_x_max]` represents a zoomed-in view of the hysteresis thresholding result for a specific region of interest (ROI) based on the earlier cropping coordinates.
3. `['Two thresholds', 'Zoom in']`: These are the titles for the two images being displayed.
4. `subplot_shape=(1, 2)`: This parameter sets up a subplot grid with one row and two columns, allowing you to display both images side by side.
5. `figsize=(8, 4)`: It defines the size of the figure where the images will be displayed.

When you run this code, you will see two images side by side in a single row:

- The result of hysteresis thresholding for the entire image with the title "Two thresholds." This image shows the categorization of pixels into strong edges, weak edges, or non-edges based on the defined thresholds.
- A zoomed-in view of the hysteresis thresholding result for a specific region of interest (ROI) with the title "Zoom in." This allows you to inspect the thresholding result in a specific area of the image.

This visualization helps you understand how pixels are categorized based on their intensity values and how this process can be used to identify edges in the image.

Canny Detection - Step 4 - Edge Linking

- Asumsikan titik yang ditandai adalah titik batas
- Lalu kita mengkonstruksi nilai tan dari kurva batas (arah normal dari gradien titik ini) dan menggunakan ini untuk memprediksi titik selanjutnya (antara r atau s)

Sambungkan batas

```
# Hysteresis - step 2
out1 = np.copy(out)
new_points = []
M, N = out1.shape
for i in range(1, M-1):
    for j in range(1, N-1):
        if (out1[i,j] == 75):
            # 8 neighbors search:
            if 255 in [out1[i+1, j-1], out1[i+1, j], out1[i+1, j+1], out1[i, j-1], out1[i, j+1],
                       out1[i-1, j-1], out1[i-1, j], out1[i-1, j+1]]:
                out1[i, j] = 255
                new_points.append(np.array([i,j]))
            else:
                out1[i, j] = 0
new_points = np.stack(new_points,-1)
```

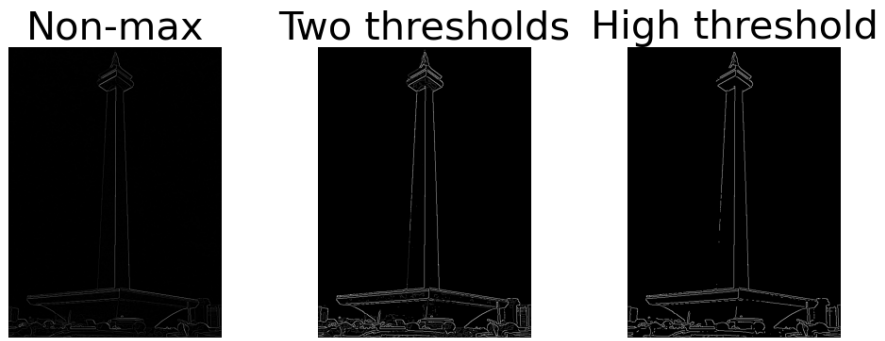
In this code snippet, you are performing the second step of hysteresis thresholding, which involves further classifying weak edges (intensity value 75) as either strong edges (255) or non-edges (0) based on their connectivity to strong edge pixels. Here's an explanation of the code:

1. `out1` is created as a copy of the previously obtained result `out`. This copy will be modified in the second step of hysteresis thresholding to refine the classification of weak edges.
2. `new_points` is initialized as an empty list. This list will be used to store the coordinates of weak edge pixels that are connected to strong edge pixels.
3. `M` and `N` are used to get the dimensions of the `out1` image, assuming `out1` is a NumPy array.
4. Two nested loops iterate over the rows and columns of the `out1` image, excluding the border pixels (i.e., from index 1 to M-2 and 1 to N-2).
5. For each pixel in `out1` with an intensity value of 75 (a weak edge pixel), you perform the following steps:
    - You search for the presence of strong edge pixels (intensity value 255) in the 8 neighboring pixels (top-left, top, top-right, left, right, bottom-left, bottom, bottom-right).
    - If at least one of the neighboring pixels is a strong edge pixel, you classify the current weak edge pixel as a strong edge pixel (set its intensity to 255), and its coordinates are added to the `new_points` list.
    - If none of the neighboring pixels are strong edge pixels, you classify the current weak edge pixel as a non-edge pixel (set its intensity to 0).
6. After processing all weak edge pixels in the image, you stack the coordinates of the newly classified strong edge pixels (`new_points`) into a NumPy array.

This step of hysteresis thresholding refines the classification of weak edge pixels based on their connectivity to strong edges. Pixels that are connected to strong edges are promoted to strong edges, while those not connected to strong edges are classified as non-edges.

After running this code, you will have an updated `out1` image where weak edges are further classified as strong edges or non-edges based on their connectivity to strong edge pixels.

```
# Hysteresis output
plot_images([Non_max, out, out > 75], ['Non-max', 'Two thresholds', 'High threshold'], subplot_shape=(1,3), figsize=(
```



In this code snippet, you are visualizing the results of the hysteresis thresholding process, specifically the output after the second step of hysteresis where weak edges are further classified as strong edges or non-edges based on their connectivity to strong edge pixels. Here's an explanation of the code:

1. `plot_images` : You are using the `plot_images` function to display three images side by side.
2. `[Non_max, out, out > 75]` : These are the three images you want to display:
    - `Non_max` represents the original gradient magnitude image.
    - `out` represents the result of the two-threshold hysteresis thresholding, where pixels are categorized into strong edges (255), weak edges (75), or non-edges (0).
    - `out > 75` creates a binary image where pixels with values greater than 75 (strong edges and weak edges) are set to True, while others are set to False. This is used to highlight pixels classified as strong edges after the second step of hysteresis.
3. `['Non-max', 'Two thresholds', 'High threshold']` : These are the titles for the three images being displayed. "Non-max" represents the original gradient magnitude, "Two thresholds" represents the result of two-threshold hysteresis, and "High threshold" represents the pixels classified as strong edges after the second step of hysteresis.
4. `subplot_shape=(1, 3)` : This parameter sets up a subplot grid with one row and three columns, allowing you to display all three images side by side.
5. `figsize=(12, 4)` : It defines the size of the figure where the images will be displayed.

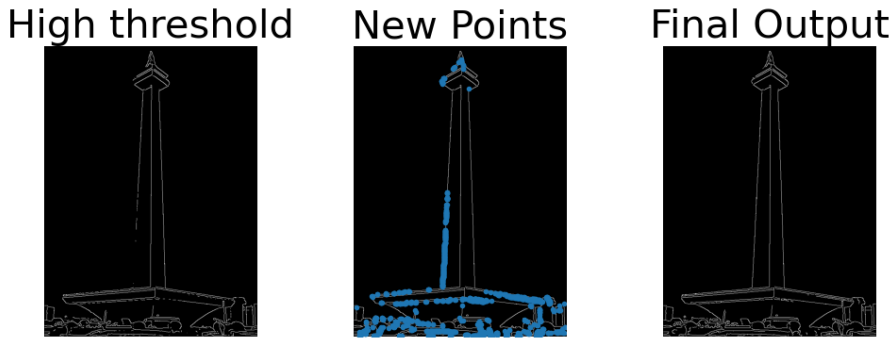
When you run this code, you will see three images side by side in a single row:

- The original gradient magnitude image with the title "Non-max."
- The result of the two-threshold hysteresis thresholding with the title "Two thresholds." This image shows the categorization of pixels into strong edges (255), weak edges (75), or non-edges (0).
- A binary image with the title "High threshold," where pixels classified as strong edges after the second step of hysteresis are highlighted.

This visualization helps you understand how the hysteresis thresholding process refines the edge classification and highlights the final strong edge pixels.

```
plt.figure(figsize=(12,4))
plt.subplot(1,3,1)
plt.imshow(out > 75,cmap='gray')
plt.title('High threshold',fontsize=30)
plt.axis('off')
plt.subplot(1,3,2)
plt.imshow(out1,cmap='gray')
plt.title('New Points',fontsize=30)
plt.plot(new_points[1],new_points[0],'.')
plt.axis('off')
plt.subplot(1,3,3)
plt.imshow(out1,cmap='gray')
plt.title('Final Output',fontsize=30)
plt.axis('off')
```

```
(-0.5, 599.5, 819.5, -0.5)
```



In this code snippet, you are creating a custom figure with three subplots to display the results of the hysteresis thresholding process, including the high threshold, new points, and the final output. Here's an explanation of the code:

1. `plt.figure(figsize=(12,4))` : This line creates a new figure with a specified figure size.

2. `plt.subplot(1,3,1)` : This sets up the first subplot in a grid with one row and three columns. The subsequent code will be applied to this subplot.
3. `plt.imshow(out > 75, cmap='gray')` : This displays the binary image created by applying a high threshold (values greater than 75) as an image using a grayscale colormap.
4. `plt.title('High threshold', fontsize=30)` : This sets the title of the first subplot to "High threshold" with a specified font size.
5. `plt.axis('off')` : This removes the axis labels and ticks from the first subplot.
6. `plt.subplot(1,3,2)` : This sets up the second subplot in the same grid.
7. `plt.imshow(out1, cmap='gray')` : This displays the `out1` image, which represents the result after the second step of hysteresis, where weak edges are further classified based on connectivity to strong edges.
8. `plt.title('New Points', fontsize=30)` : This sets the title of the second subplot to "New Points" with a specified font size.
9. `plt.plot(new_points[1], new_points[0], '.')` : This overlays points on the second subplot to visualize the coordinates of new points (pixels classified as strong edges after the second step of hysteresis).
10. `plt.axis('off')` : This removes the axis labels and ticks from the second subplot.
11. `plt.subplot(1,3,3)` : This sets up the third subplot in the same grid.
12. `plt.imshow(out1, cmap='gray')` : This displays the `out1` image again.
13. `plt.title('Final Output', fontsize=30)` : This sets the title of the third subplot to "Final Output" with a specified font size.
14. `plt.axis('off')` : This removes the axis labels and ticks from the third subplot.

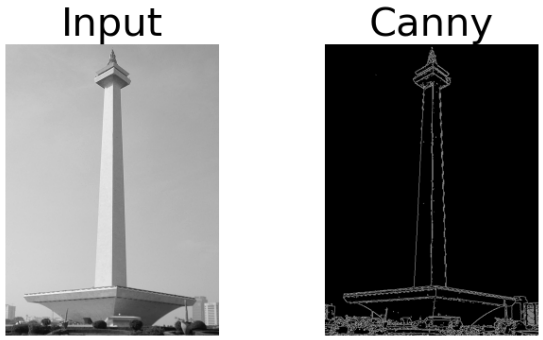
When you run this code, you will see a custom figure with three subplots, each displaying different aspects of the hysteresis thresholding process:

- The first subplot shows the result of applying a high threshold.
- The second subplot shows the `out1` image and overlays the coordinates of new points.
- The third subplot shows the `out1` image without the overlay, representing the final output of the hysteresis thresholding process.

This custom visualization provides insight into the hysteresis thresholding steps and the classification of pixels as strong edges.

Canny dalam 1 baris (OpenCV)

```
# Canny
im_canny = cv2.Canny(gray_cut, 70, 100)  # 50, 260
plot_images([gray_cut, im_canny], ['Input', 'Canny'], subplot_shape=(1,2), figsize=(8,4))
```



In this code snippet, you are applying the Canny edge detection algorithm to the cropped grayscale image `gray_cut` . Here's an explanation of the code:

1. `cv2.Canny(gray_cut, 70, 100)` : This line of code applies the Canny edge detection algorithm to the `gray_cut` image. The parameters `70` and `100` are the lower and upper thresholds for edge detection. These values determine which edges are considered strong and which are considered weak. Edges with gradient magnitudes between these thresholds are considered weak edges, and those above the upper threshold are considered strong edges.
2. `plot_images([gray_cut, im_canny], ['Input', 'Canny'], subplot_shape=(1,2), figsize=(8,4))` : This line uses the `plot_images` function to display two images side by side:
    - The original grayscale image (`gray_cut`) with the title "Input."

- The result of applying the Canny edge detection algorithm (`im_canny`) with the title "Canny."

3. `subplot_shape=(1, 2)`: This parameter sets up a subplot grid with one row and two columns, allowing you to display both images side by side.
4. `figsize=(8, 4)`: It defines the size of the figure where the images will be displayed.

When you run this code, you will see two images side by side in a single row:

- The original grayscale image with the title "Input."
- The result of the Canny edge detection algorithm with the title "Canny." This image will highlight the detected edges in white against a black background.

The Canny edge detection algorithm is commonly used for detecting edges in images due to its ability to suppress noise and accurately locate edges. The choice of threshold values (`70` and `100` in this case) affects the sensitivity of edge detection, and you can adjust them based on your specific requirements.

Transformasi Hough

```
# Read the image
img_orig = cv2.imread('./images/tianglistrik.jpeg')
img = img_orig.copy()

# Convert to grayscale
gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)

# Find the edges using Canny detector
edges = cv2.Canny(gray,50,150,apertureSize = 3)

# Apply the hough transform
lines = cv2.HoughLines(edges,1,np.pi/180,200)

# Draw the lines
for line in lines:
    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))
    cv2.line(img,(x1,y1),(x2,y2),(0,0,255),2)
```

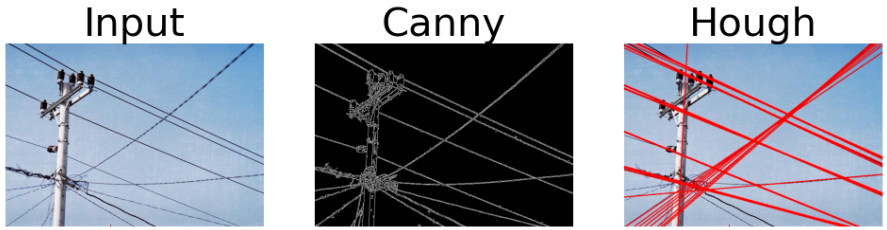
In this code snippet, you are performing edge detection using the Canny detector and then applying the Hough Line Transform to detect and draw lines in an image. Here's an explanation of the code:

1. `cv2.imread('tianglistrik.jpeg')`: This line reads the image named 'tianglistrik.jpeg' and stores it in the `img_orig` variable.
2. `img = img_orig.copy()`: A copy of the original image is created to avoid modifying the original image directly.
3. `cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)`: This converts the image from color (BGR) to grayscale and stores it in the `gray` variable. Grayscale images are often used for edge detection.
4. `cv2.Canny(gray, 50, 150, apertureSize=3)`: The Canny edge detection algorithm is applied to the grayscale image (`gray`) with specified thresholds (50 and 150). `apertureSize` specifies the size of the Sobel kernel used for edge detection. The result is stored in the `edges` variable.
5. `cv2.HoughLines(edges, 1, np.pi/180, 200)`: The Hough Line Transform is applied to the binary edge image (`edges`). The parameters are as follows:
   - `1`: The resolution of the accumulator in pixels.
   - `np.pi/180`: The resolution of the accumulator in radians (1 degree).
   - `200`: The threshold value that determines how many votes a line must receive to be considered a line. Lines with votes above this threshold are detected and returned in the `lines` variable.
6. `for line in lines:`: This loop iterates over the detected lines.
7. Inside the loop:
   - `rho, theta = line[0]`: The polar representation of the line is extracted from the `line` variable, where `rho` is the distance from the origin to the line and `theta` is the angle of the line.

   - `a = np.cos(theta)` and `b = np.sin(theta)`: The cosine and sine of the angle `theta` are calculated.
   - `x0 = a * rho` and `y0 = b * rho`: These equations compute a point `(x0, y0)` on the line.
   - `x1, y1, x2,` and `y2` are calculated to determine two points (`(x1, y1)` and `(x2, y2)`) on the line, which are used to draw the line segment.
   - `cv2.line(img, (x1, y1), (x2, y2), (0, 0, 255), 2)`: This draws a line segment on the `img` image using the calculated points. The arguments are as follows:
     - `(x1, y1)` and `(x2, y2)` are the endpoints of the line.
     - `(0, 0, 255)` specifies the color of the line in BGR format (red in this case).
     - `2` is the thickness of the line.

After running this code, you will have the original image with red lines drawn on it, representing the detected lines in the image using the Hough Line Transform.

```
# show the result
img_orig = cv2.cvtColor(img_orig, cv2.COLOR_BGR2RGB)
img_lines = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
plot_images([img_orig, edges, img_lines], ['Input', 'Canny', 'Hough'], subplot_shape=(1,3), figsize=(12,4))
```



In this code snippet, you are displaying the results of the edge detection and Hough Line Transform on the original image. Here's an explanation of the code:

1. `cv2.cvtColor(img_orig, cv2.COLOR_BGR2RGB)`: This line converts the original image from the BGR color space to the RGB color space. This step is necessary because Matplotlib expects images in RGB format for proper display.
2. `cv2.cvtColor(img, cv2.COLOR_BGR2RGB)`: Similarly, this line converts the image with drawn lines (`img`) from BGR to RGB color space.
3. `plot_images([img_orig, edges, img_lines], ['Input', 'Canny', 'Hough'], subplot_shape=(1,3), figsize=(12,4))`: You use the `plot_images` function to display three images side by side:
   - The original input image (`img_orig`) with the title "Input."
   - The result of the Canny edge detection (`edges`) with the title "Canny."
   - The image with detected lines using the Hough Line Transform (`img_lines`) with the title "Hough."
4. `subplot_shape=(1, 3)`: This parameter sets up a subplot grid with one row and three columns, allowing you to display all three images side by side.
5. `figsize=(12, 4)`: It defines the size of the figure where the images will be displayed.

When you run this code, you will see three images side by side in a single row:

- The original input image with the title "Input."
- The result of the Canny edge detection, highlighting detected edges, with the title "Canny."
- The original input image with red lines drawn on it, representing the lines detected using the Hough Line Transform, with the title "Hough."

This visualization allows you to compare the original image, edge detection results, and the detected lines, providing insights into the image processing steps applied to detect edges and lines in the image.

Ransac

```
from sklearn import linear_model, datasets

# create inlier and outlier dataset
n_samples, n_outliers = 1000, 50
X, y, coef = datasets.make_regression(n_samples=n_samples, n_features=1,
                                      n_informative=1, noise=10,
                                      coef=True, random_state=0)
```

```
np.random.seed(0)
X[:n_outliers] = 3 + 0.5 * np.random.normal(size=(n_outliers, 1))
y[:n_outliers] = -3 + 10 * np.random.normal(size=n_outliers)

# Fit line using all data
lr = linear_model.LinearRegression()
lr.fit(X, y)

# Robustly fit line with RANSAC algorithm
ransac = linear_model.RANSACRegressor()
ransac.fit(X, y)
inlier_mask = ransac.inlier_mask_
outlier_mask = np.logical_not(inlier_mask)

# Predict data of estimated models
line_X = np.arange(X.min(), X.max())[:, np.newaxis]
line_y = lr.predict(line_X)
line_y_ransac = ransac.predict(line_X)
```

In this code snippet, you are using scikit-learn to create a dataset with both inliers and outliers, fitting a linear regression model to the data, and then fitting a robust linear regression model using the RANSAC algorithm. Here's a breakdown of the code:

1. Import the necessary libraries:
   - `linear_model` and `datasets` from `sklearn` for linear regression and dataset creation.
2. Create the dataset:
   - `n_samples, n_outliers`: Define the total number of samples and the number of outliers in the dataset.
   - `datasets.make_regression`: Generate a dataset with a single feature (`n_features=1`) and one informative feature. The dataset contains noise, and the true coefficient of the linear model is also provided (`coef=True`).
   - Introduce outliers by adding noise to a subset of the data points.
3. Fit a linear regression model:
   - Create an instance of `linear_model.LinearRegression()` as `lr`.
   - Use `lr.fit(X, y)` to fit the linear regression model to the entire dataset.
4. Fit a robust linear regression model with RANSAC:
   - Create an instance of `linear_model.RANSACRegressor()` as `ransac`.
   - Use `ransac.fit(X, y)` to fit the robust linear regression model to the data. RANSAC is used to robustly estimate the model by iteratively fitting the model to a subset of the data, identifying inliers, and re-fitting to the inliers.
5. Create masks to separate inliers and outliers:
   - `ransac.inlier_mask_` is a Boolean mask indicating which data points are considered inliers by the RANSAC algorithm.
   - `np.logical_not(inlier_mask)` creates an outlier mask by negating the inlier mask.
6. Predict data for both models:
   - Generate a range of data points (`line_X`) for making predictions.
   - Use the linear regression model (`lr`) to predict `line_y`.
   - Use the RANSAC-based model (`ransac`) to predict `line_y_ransac`.

After running this code, you will have the following:

- The linear regression model (`lr`) fitted to the entire dataset.
- The RANSAC-based robust linear regression model (`ransac`) that is more robust to outliers.
- Predictions for both models for a range of data points (`line_X`).

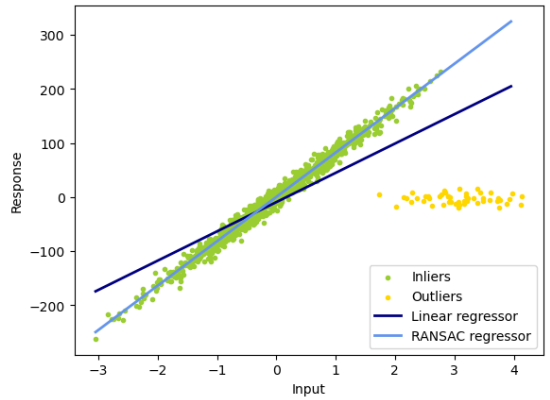
You can use these models and predictions for further analysis or visualization, such as plotting the original data points, inliers, outliers, and the fitted lines.

```
# Compare estimated coefficients
print('True Slope: {:.2f}'.format(coef))
print('Est. LS: {:.2f}'.format(lr.coef_[0]))
print('Est. RANSAC: {:.2f}'.format(ransac.estimator_.coef_[0]))

# plot the fit
plt.scatter(X[inlier_mask], y[inlier_mask], color='yellowgreen', marker='.',
            label='Inliers')
plt.scatter(X[outlier_mask], y[outlier_mask], color='gold', marker='.',
            label='Outliers')
plt.plot(line_X, line_y, color='navy', linewidth=2, label='Linear regressor')
plt.plot(line_X, line_y_ransac, color='cornflowerblue', linewidth=2,
```



```
         label='RANSAC regressor')
plt.legend(loc='lower right')
plt.xlabel("Input")
plt.ylabel("Response")
plt.show()
```

```
True Slope: 82.19
Est. LS: 54.17
Est. RANSAC: 82.09
```



In this code snippet, you are comparing the estimated coefficients of the true linear model, the linear regression model, and the RANSAC-based robust linear regression model. You are also visualizing the data points, inliers, outliers, and the fitted lines. Here's an explanation of the code:

1. Compare estimated coefficients:
   - `print('True Slope: {:.2f}'.format(coef))`: This line prints the true slope (coefficient) of the linear model.
   - `print('Est. LS: {:.2f}'.format(lr.coef_[0]))`: This line prints the estimated slope (coefficient) of the linear regression model.
   - `print('Est. RANSAC: {:.2f}'.format(ransac.estimator_.coef_[0]))`: This line prints the estimated slope (coefficient) of the RANSAC-based robust linear regression model. The `ransac.estimator_` attribute provides access to the underlying linear regression model fitted by RANSAC.
2. Plot the fit:
   - `plt.scatter(X[inlier_mask], y[inlier_mask], color='yellowgreen', marker='.', label='Inliers')`: This scatter plot displays the inlier data points in yellow-green.
   - `plt.scatter(X[outlier_mask], y[outlier_mask], color='gold', marker='.', label='Outliers')`: This scatter plot displays the outlier data points in gold.
   - `plt.plot(line_X, line_y, color='navy', linewidth=2, label='Linear regressor')`: This line plot shows the fit of the linear regression model in navy blue.
   - `plt.plot(line_X, line_y_ransac, color='cornflowerblue', linewidth=2, label='RANSAC regressor')`: This line plot shows the fit of the RANSAC-based robust linear regression model in cornflower blue.
   - `plt.legend(loc='lower right')`: This adds a legend to the plot to label the different elements.
   - `plt.xlabel("Input")` and `plt.ylabel("Response")`: These labels the x-axis and y-axis, respectively.

After running this code, you will see a plot that visually compares the fits of the linear regression model and the RANSAC-based robust linear regression model to the data. Inliers and outliers are differentiated, and the estimated coefficients of the models are printed for comparison. This visualization helps you understand the impact of outliers on the model estimation and how the RANSAC algorithm can provide more robust results in the presence of outliers.