Temat: Operacje punktowe.

W tym ćwiczeniu przerobimy:

- Przekształcenia geometryczne – translacja, rotacja , skalowanie, transformacje afiniczne i transformacja perspektywy.
- Przycinanie obrazów za pomocą określonych funkcji NumPy i OpenCV.
- Omówimy obrazy binarne i sposób wykonywania operacji arytmetycznych na tych obrazach.
- Przyjrzymy się kilku rzeczywistym zastosowaniom, w których powyższe techniki mogą się przydać.

Przerabiane zagadnienia:

Spis treści

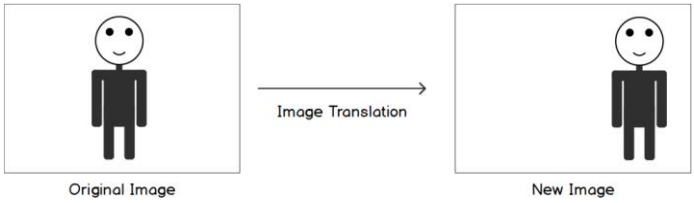
1 Przekształcenia geometryczne

- Często podczas przetwarzania obrazu konieczna jest transformacja geometrii obrazu (szerokość i wysokość obrazu).
- Ten proces nazywa się transformacją geometryczną.
- Ponieważ obrazy są macierzami, to stosując jakąś dowolną operację na obrazach (macierzach) otrzymamy inną macierz – nazywamy to przekształceniem.
- Ta podstawowa idea będzie szeroko wykorzystywana do zrozumienia i zastosowania różnych rodzajów przekształceń geometrycznych.

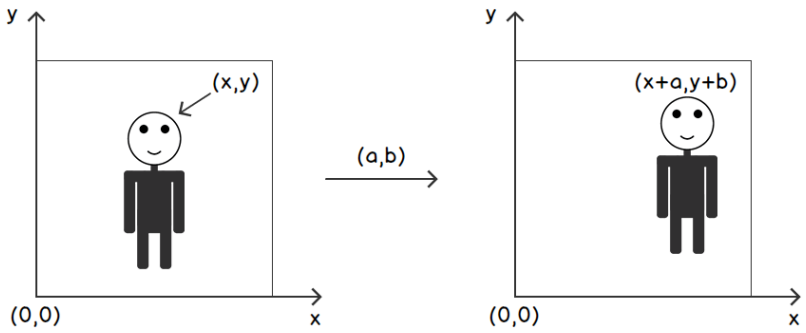
Oto przekształcenia geometryczne, które omówimy:
- Translacje
- Obrót
- Skalowanie lub zmiana rozmiaru obrazu
- Transformacje afiniczne
- Transformacje perspektywy

1.1 Translacje obrazu (przesunięcie)

- translacje to przesunięcia obrazu
- podstawowa intencja to przesunięcie obrazu wzdłuż linii



- w przykładzie przemieszczamy człowieczka do prawej strony
- obraz można przesuwać w obu kierunkach x i y jednocześnie lub każdym kierunku oddzielnie
- w tym przetwarzaniu każdy piksel obrazu przesuwamy w pewnym kierunku



- na podstawie powyższego rysunku można przedstawić translację obrazu w postaci następującego równania macierzowego

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} x + a \\ y + b \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} a \\ b \end{bmatrix}$$

gdzie:
x' i y' reprezentują nowe współrzędne piksela po jego przesunięciu o a jednostek w kierunku x i b jednostek w kierunku y

- powyższe przekształcenie jest w rzeczywistości bardzo rzadko używane w przetwarzaniu obrazów, a jeśli nawet występuje, to zwykle w połączeniu z innymi transformacjami, jako transformacje afiniczne
- ponieważ obrazy w środowisku OpenCV + Python są tablicami biblioteki NumPy, translację obrazu można wykonać przez zastosowanie równania macierzowego i w tym sensie to samo równanie macierzowe można zapisać jako tablicę NumPy:

```
M = np.array([[1,0,a],[0,1,b],[0,0,1])
```

- pierwsze dwie kolumny tej macierzy tworzą macierz jednostkową ([[1,0],[0,1]]), co oznacza, że chociaż przekształcamy obraz, jego wymiary (szerokość i wysokość) pozostaną takie same
- ostatnia kolumna składa się z a i b, co oznacza przesunięcie obrazu o a jednostek w kierunku x oraz b jednostek w kierunku y
- ostatni wiersz [0,0,1] służy tylko do utworzenia macierzy kwadratowej M o tej samej liczbie wierszy i kolumn
- obraz img jest tablicą NumPy w której każda kolumna oznacza jeden piksel obrazu, przy czym pierwszy wiersz dotyczy współrzędnych x, drugi wiersz dotyczy współrzędnych y, a trzeci wiersz jest wypełniony jedynkami w celu zachowania wymiaru tej macierzy potrzebnego do mnożenia
- teraz kiedy chcemy przesunąć obraz, wystarczy pomnożyć obraz img (jako tablicę NumPy) przez tablicę M, co może być wykonane w następujący sposób:

```
output = M@img
```

gdzie `output` jest obrazem po przesunięciu

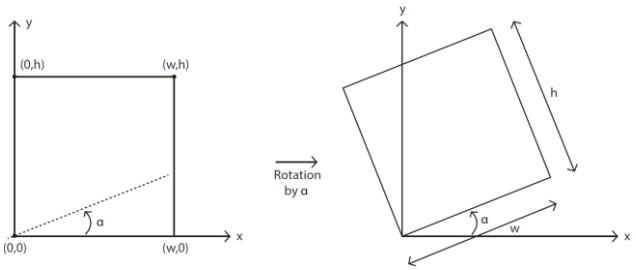
Wykonaj poniższe ćwiczenie:

```
# -*- coding: utf-8 -*-
"""
Przesunięcie obrazu
"""

# %%
# 1. Import modułu
import numpy as np

# %%
# 2. Określenie 3 punktów które chcemy przesunąć
# [2,3], [0,0], [1,2]

points = np.array([[2,0,1],
                   [3,0,2],
                   [1,1,1]])

print(points)

# %%
# 3. Parametry przesunięcia

# Przesunięcie o 2 jednostki w kierunku osi x
a = 2
# Przesunięcie o 3 jednostki w kierunku osi y
b = 3

# %%
# 4. Punkty po przesunięciu - obliczenie "ręczne" za pomocą dodawania
```

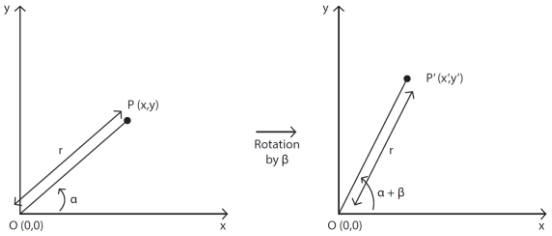
```
outPoints = np.array([[2 + a, 0 + a, 1 + a],
            [3 + b, 0 + b, 2 + b],
            [1, 1, 1]])

print(outPoints)

# %%
# 5. Macierz translacji
M = np.array([[1,0,a],[0,1,b],[0,0,1]])

print(M)


# %%
# 6. Wykonanie przesunięcia za pomocą NumPy
output = M@points

print(output)
```

1.2 Obrót obrazu

- w podobny sposób wyprowadza się zależności dotyczące określenia wartości macierzy transformacji przy wykonywaniu obracania obrazu
- na rysunku obraz ma wymiary w i h, obracamy obraz o kąt α wokół punktu, który jest na rysunku początkiem układu współrzędnych



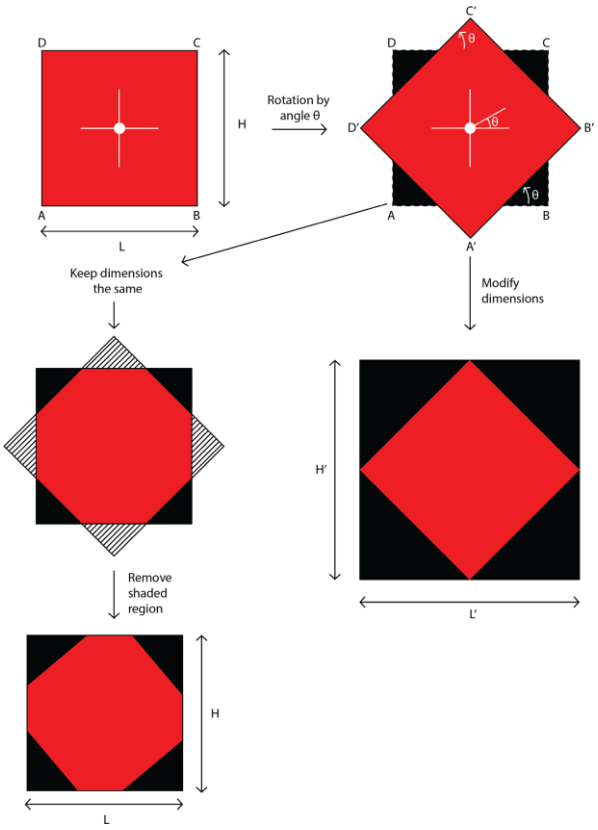
- problem jest w tym przypadku podzielony na dwa zadania:
  - określenie macierzy obrotu
  - określenie wielkości obrazu po przekształceniu
- pomijając wyprowadzenie wzoru, jeżeli punkt jest obracany wokół początku układu współrzędnych (0,0) o kąt β, to macierzowe równanie współrzędnych x' i y' jest następujące:



$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\beta & -\sin\beta \\ \sin\beta & \cos\beta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

- równanie to można zastosować do każdego piksela obrazu, aby obrócić go o zadany kąt

- znajdowanie wielkości obrazu po obrocie, rozpatruje się dwa przypadki:
    - po obrocie obraz zachowuje swoją wielkość
    - po obrocie wielkość obrazu jest modyfikowana



- dobra wiadomość: biblioteka OpenCV posiada funkcję `cv2.getRotationMatrix2D` która wspomoże nas przy wykonywaniu obracania obrazu

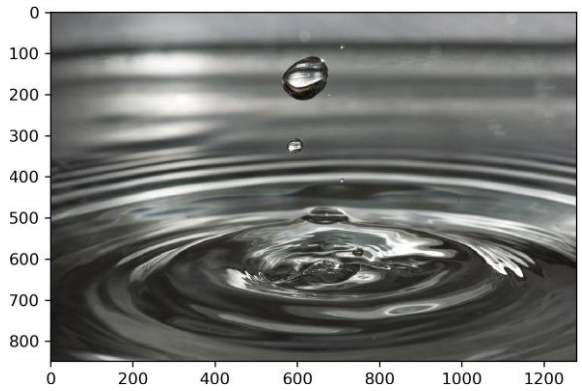
- w poniższej tabeli podsumowano obie transformacje: przesunięcie i obrót

Przekształcenie	**Macierz Transformacji**	**Generowanie Macierzy Transformacji**
Przesunięcie	$\begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \end{bmatrix}$ gdzie: t_x i t_y oznaczają przesunięcie odpowiednio w kierunku x i y	**M=np.float32([[1,0,t_x],[0,1,t_y]]** gdzie: t_x, t_y oznaczają przesunięcie
Obrót	$\begin{bmatrix} \alpha & \beta & (1-\alpha)\cdot center.x - \beta \cdot center.y \\ -\beta & \alpha & \beta \cdot center.x + (1-\alpha)\cdot center.y \end{bmatrix}$ gdzie: $\alpha = scale \cdot \cos\theta$, $\beta = scale \cdot \sin\theta$ **center.x** i **center.y** reprezentują współrzędne x i y punktu (środka) wokół którego obraz jest obracany	**M=cv2.getRotationMatrix2D((centerX,centerY),angle,scale)** gdzie: **(centerX, centerY)** są współrzędnymi punktu wokół którego będzie wykonane obrót, **angle** jest kątem obrotu, **scale** jest współczynnikiem przez który wyjściowy obraz będzie przeskalowany w górę lub w dół

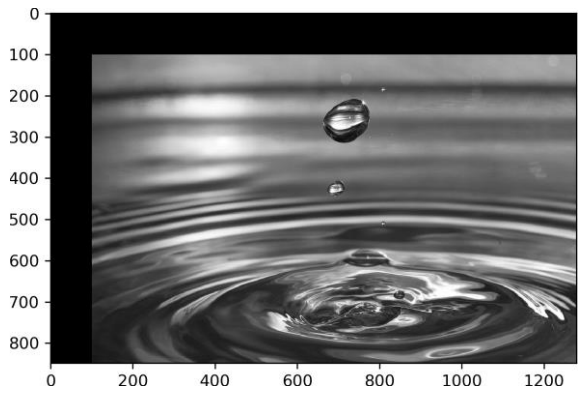
1.3 Transformacje afiniczne

- Transformacje afiniczne są przykładem transformacji geometrycznych w wizji komputerowej.
- Transformacja afiniczna może łączyć efekty translacji, rotacji i zmiany rozmiaru w jedną transformację.
- Transformacja afiniczna w OpenCV wykorzystuje macierz 2×3, a następnie stosuje odpowiednie efekty za sprawą tej macierzy poprzez funkcję **cv2.warpAffine**.
- Funkcja ta przyjmuje trzy argumenty:
    - **`cv2.warpAffine(src, M, dsize)`**
        - **src** – obraz który chcemy zastosować do transformacji
        - **M** – macierz transformacji
        - **dsize** – kształt obrazu wyjściowego (liczba kolumn, liczba wierszy)

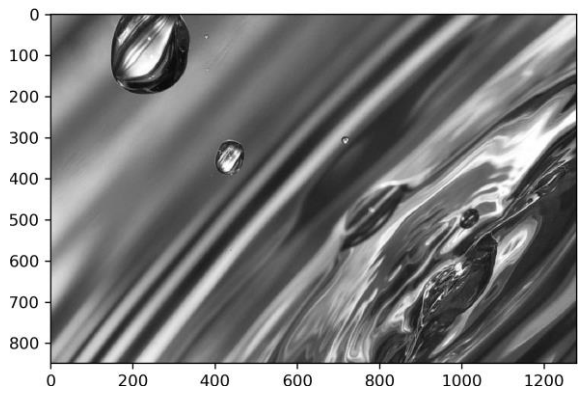
```
import cv2
import numpy as np
import matplotlib.pyplot as plt
img = cv2.imread("images/drip.jpg")

# Wyświetlenie obrazu przy użyciu matplotlib
plt.imshow(img[:,:,::-1])
plt.show()
```




```
# Konwersja obrazu do skali szarości
img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

# Zapamiętanie wysokości i szerokości obrazu
height,width = img.shape

# Translacja
tx = 100
ty = 100
M = np.float32([[1,0,tx],[0,1,ty]])
dst = cv2.warpAffine(img,M,(width,height))
plt.imshow(dst,cmap="gray")
plt.show()
```




```
# Obrót
angle = 45
center = (width//2, height//2)
scale = 2
M = cv2.getRotationMatrix2D(center,angle,scale)
dst = cv2.warpAffine(img,M,(width,height))
plt.imshow(dst,cmap="gray")
plt.show()
```




```
# Zmiana rozmiaru
print("Szerokość obrazu = {}, Wysokość obrazu = {}"\
.format(width, height))
dst = cv2.resize(img, None, fx=2, fy=2, \
interpolation=cv2.INTER_LINEAR)
height, width = dst.shape
print("Szerokość obrazu = {}, Wysokość obrazu = {}"\
.format(width, height))
```

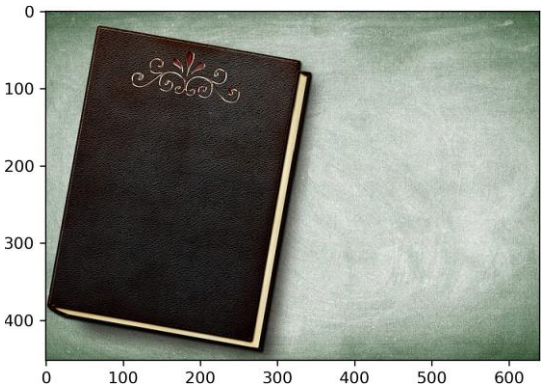
Wynik przetwarzania:

```
Szerokość obrazu = 1280, Wysokość obrazu = 849
Szerokość obrazu = 2560, Wysokość obrazu = 1698
```

1.4 Transformacje perspektywy

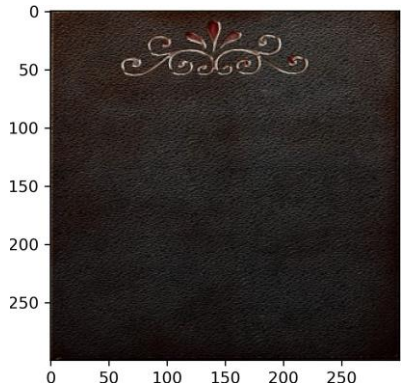
- Do wykonania transformacji perspektywy biblioteka OpenCV dostarcza dwie funkcje
  - tworzenie macierzy M – funkcja: `cv2.getPerspectiveTransform`
  - transformacja – funkcja: `cv2.warpPerspective`
- Będą potrzebne cztery punkty na obrazie wejściowym i współrzędne tych samych punktów na obrazie wyjściowym.
- Punkty te nie powinny być współliniowe.

```
# Import modułów
import cv2
import numpy as np
import matplotlib.pyplot as plt

# Wczytanie obrazu
img = cv2.imread("images/book.jpg")

# Wyświetlenie obrazu
plt.imshow(img[:,:,::-1])
plt.show()
```



- Określ cztery punkty na obrazie, tak aby leżały w czterech rogach przedniej okładki.
- Ponieważ potrzebna jest tylko przednia okładka, punkty wyjściowe będą niczym innym jak punktami narożnymi końcowego obrazu 300×300.
- Zwróć uwagę, że kolejność punktów powinna pozostać taka sama dla punktów wejściowych i wyjściowych:

```
inputPts = np.float32([[4,381],[266,429], [329,68], [68,20]])
outputPts = np.float32([[0,300], [300,300], [300,0], [0,0]])

# Macierz transformacji
M = cv2.getPerspectiveTransform(inputPts,outputPts)

# Zastosowanie macierzy transformacji do transformacji perspektywy
dst = cv2.warpPerspective(img,M,(300,300))

# Wyświetlenie wyniku
plt.imshow(dst[:,:,::-1])
plt.show()
```



- W tym ćwiczeniu zobaczyliśmy, jak można użyć przekształceń geometrycznych, aby wyodrębnić przednią okładkę książki z danego obrazu.
- Może być to przydatne przy skanowaniu dokumentu gdy chcemy uzyskać odpowiednio zorientowany obraz dokumentu.

2 Arytmetyka na obrazach

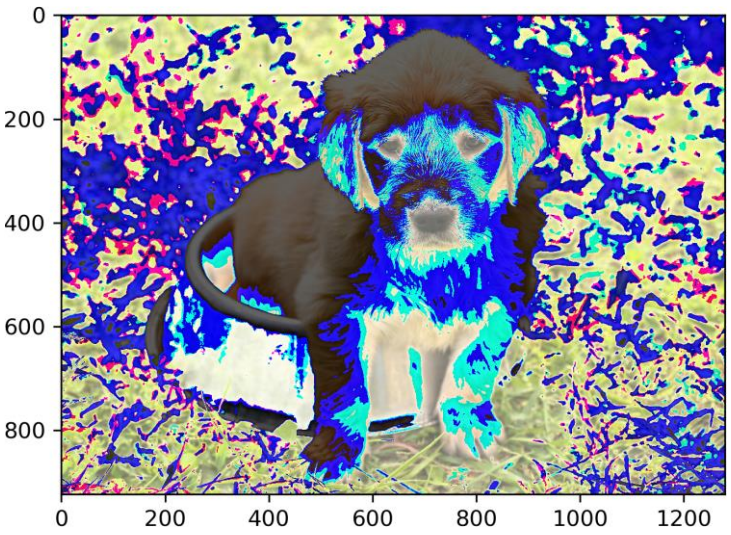
- Obrazy to nic innego jak macierze, a na macierzach można wykonywać operacje arytmetyczne, to możemy wykonywać je również na obrazach.
- Na obrazach możemy wykonać następujące operacje:
    - Dodawanie i odejmowanie dwóch obrazów
    - Dodawanie i odejmowanie stałej wartości do/od obrazu
    - Mnożenie stałej wartości przez obraz
- Można oczywiście pomnożyć dwa obrazy, jeśli założymy, że są to macierze, ale jeśli chodzi o obrazy, mnożenie dwóch obrazów nie ma większego sensu, chyba że odbywa się to na pikselach.

2.1 Dodawanie stałej do obrazu

- Przy dodawaniu obrazów (lub wartości stałych do obrazów) należy pamiętać, że zakres wartości pikseli to zwykle (0 – 255).
- Po sumowaniu okaże się, że część pikseli będzie miała wartość większą niż dopuszczalna i mogą zajść wówczas dwa przypadki:
    - końcowa wartość zostanie przycięta do maksymalnej (255)
    - wartość ostateczna dla określonego piksela zostanie wyznaczona z operacji modulo 255
- Wyniki uzyskane w obu podejściach są różne, ale rekomendowane jest podejście z OpencCV
- Poniższe przykłady przedstawiają oba przypadki.

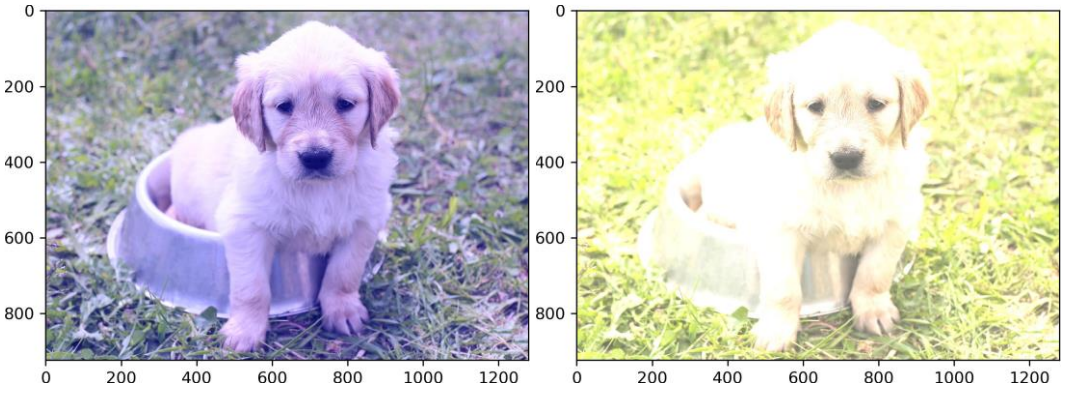
```
# Import bibliotek
import cv2
import numpy as np
import matplotlib.pyplot as plt

# Wczytanie obrazu
img = cv2.imread("images/puppy.jpg")

# Wyświetlenie obrazu
plt.imshow(img[:,:,::-1])
plt.show()
```

```
# Dodanie wartości 100 do obrazu
numpyImg = img + 100

# Wyświetlenie obrazu
plt.imshow(numpyImg[:,:,::-1])
plt.show()
```



- Dodanie wartości 100 poważnie zniekształciło obraz. Wynika to z operacji modulo wykonywanej przez NumPy na nowych wartościach pikseli. Powinno to również dać wyobrażenie, dlaczego podejście NumPy nie jest zalecanym podejściem do użycia podczas dodawania stałej wartości do obrazu.
- Teraz zrobimy to samo przy pomocy biblioteki OpenCV.

```
# Użycie OpenCV
# 1 wariant
opencvImg = cv2.add(img,np.array([100]))

# 2 wariant
# opencvImg = cv2.add(img,100)


# Wyświetlenie obrazu
plt.imshow(opencvImg[:,:,::-1])
plt.show()
```



- Jak pokazano pierwszy obraz ma wzmocniony niebieski odcień, dzieje się tak, ponieważ wartość 100 została dodana tylko do pierwszego kanału obrazu, którym jest kanał niebieski. W drugim przypadku wartość 100 została dodana do wszystkich kanałów (kanały R i G zostały nasycone maksymalnie wartością 255)
- We wcześniejszych implementacjach funkcji cv2.add() nie można było dodawać wartości jednocześnie do wszystkich kanałów, zatem, aby wykonać takie dodawanie należało dodawać dwa obrazy, a nie stałą do obrazu, jak poniżej:

```
#  Sprawdzenie kształtu obrazu
img.shape
```

- Teraz utwórz obraz o takim samym kształcie jak obraz oryginalny, który ma wartość stałą pikseli = 100.
- Robimy to, ponieważ chcemy dodać wartość 100 do każdego kanału oryginalnego obrazu:

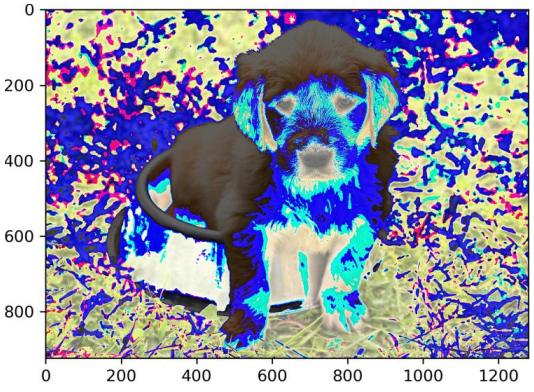
```
nparr = np.ones((924,1280,3),dtype=np.uint8) * 100
```

- Dodanie `nparr` do obrazu i wizualizacja

```
opencvImg = cv2.add(img,nparr)

plt.imshow(opencvImg[:,:,::-1])
plt.show()
```

- Teraz gdy ten sam obraz zostanie dodany za pomocą OpenCV, zauważymy, że otrzymane wyniki są takie same, jak w przypadku NumPy:

```
npImg = img + nparr
plt.imshow(npImg[:,:,::-1])
plt.show()
```



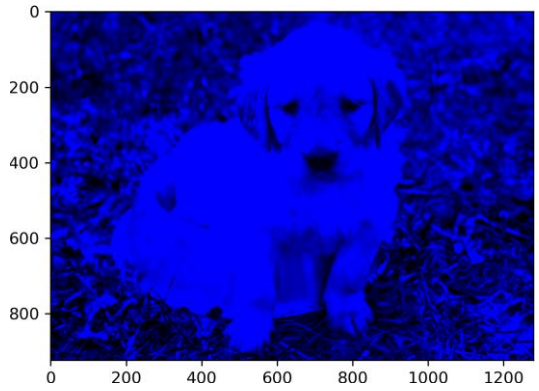
- Ważnym spostrzeżeniem w otrzymanych wynikach jest to, że dodanie wartości do obrazu za pomocą OpenCV powoduje zwiększenie jego jasności.
- Możesz spróbować odjąć wartość (lub dodać wartość ujemną) i sprawdzić, czy odwrotne zachowanie będzie również prawdziwe.
- W tym ćwiczeniu dodaliśmy stałą wartość do obrazu i porównaliśmy dane wyjściowe uzyskane za pomocą funkcji `cv2.add()` oraz operatora dodawania NumPy (+).
- Widzieliśmy również, że podczas korzystania z funkcji `cv2.add()` wartość jest dodawana do wszystkich trzech kanałów.

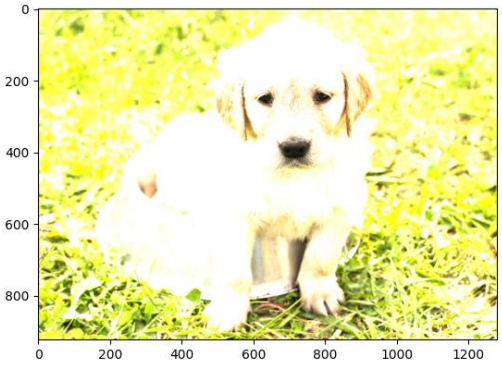
2.2 Mnożenie obrazów

- Mnożenie obrazów jest bardzo podobne do dodawania obrazów i można je przeprowadzić za pomocą funkcji `cv2.multiply()` OpenCV, co jest zalecane lub NumPy.
- Funkcja OpenCV jest zalecana z tego samego powodu, co w przypadku `cv2.add()` w poprzednim ćwiczeniu.

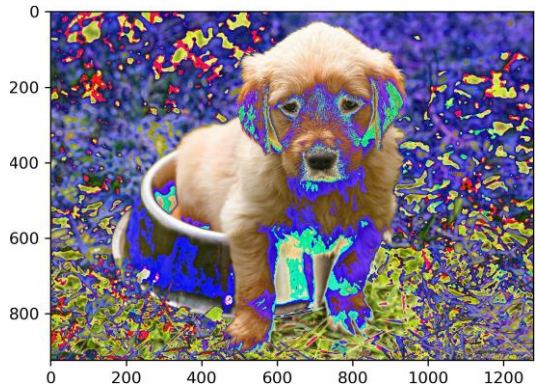
```
# Import bibliotek
import cv2
import numpy as np
import matplotlib.pyplot as plt


# Wczytanie i wyświetlenie obrazu
img = cv2.imread("images/puppy.jpg")
plt.imshow(img[:,:,::-1])
plt.show()
```


```
# Mnożenie obrazu przez 2 i wyświetlenie wyniku
# 1 Wersja
cvImg = cv2.multiply(img,np.array([2]))

# 2 2 Wersja
# cvImg = cv2.multiply(img,2)

plt.imshow(cvImg[:,:,::-1])
plt.show()
```






```
# Mnożenie z użyciem NumPy
npImg = img*2

plt.imshow(npImg[:,:,::-1])
plt.show()
```




```
# Zbadanie kształtu obrazu
img.shape

# Tablica 1-nek o rozmiarze obrazu
nparr = np.ones((924,1280,3),dtype=np.uint8) * 2


# Mnożenie tablic
cvImg = cv2.multiply(img,nparr)
plt.imshow(cvImg[:,:,::-1])
plt.show()
```

- Mnożenie to nic innego jak powtarzalne dodawanie, więc sensowne jest uzyskanie jaśniejszego obrazu za pomocą mnożenia.

- Do tej pory omawialiśmy przekształcenia geometryczne i arytmetykę obrazów. Przejdźmy teraz do nieco bardziej zaawansowanego tematu dotyczącego wykonywania operacji bitowych na obrazach. Zanim to omówimy, spójrzmy na obrazy binarne.

3 Obrazy binarne

3.1 Wprowadzenie

- Obrazy binarne potrzebują tylko jednego bitu do reprezentowania wartości piksela.
- Te obrazy są powszechnie używane jako maski do zaznaczania lub usuwania określonego obszaru obrazu.
- To właśnie na tych obrazach powszechnie stosuje się operacje bitowe.
- Gdzie można zobaczyć binarne obrazy w rzeczywistości?
- Takie czarno-białe obrazy można znaleźć dość często w kodach QR.
- Obrazy binarne są szeroko stosowane do analizy dokumentów, a nawet w przemysłowych zadaniach widzenia maszynowego. Oto przykładowy obraz binarny:

- Zobaczmy teraz, jak możemy przekonwertować obraz na obraz binarny.
- Technika ta należy do kategorii progowania.
- Progowanie odnosi się do procesu konwersji obrazu kolorowego na obraz binarny.
- Dostępnych jest wiele technik progowania, ale tutaj skupimy się tylko na bardzo prostej technice progowania – progowaniu binarnym – ponieważ pracujemy z obrazami binarnymi.
- Koncepcja progowania binarnego jest bardzo prosta:
    - Wybierasz wartość progową, a wszystkie wartości pikseli poniżej progu i równe progowi są zastępowane przez 0, podczas gdy wszystkie wartości pikseli powyżej progu są zastępowane określoną wartością (zwykle 1 lub 255).
    - W ten sposób otrzymujesz obraz, który ma tylko dwie unikalne wartości pikseli, czyli właśnie to jest obraz binarny.

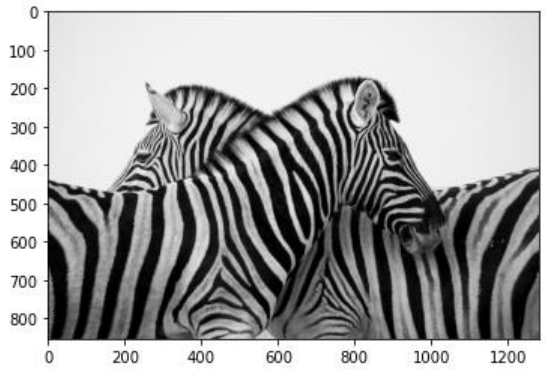
```
# Przykład ustawienia progu i wartości max
thresh = 125
maxValue = 255
# Binarny próg
th, dst = cv2.threshold(img, thresh, maxValue, cv2.THRESH_BINARY)
```

3.2 Zadanie. Konwersja obrazu w obraz binarny

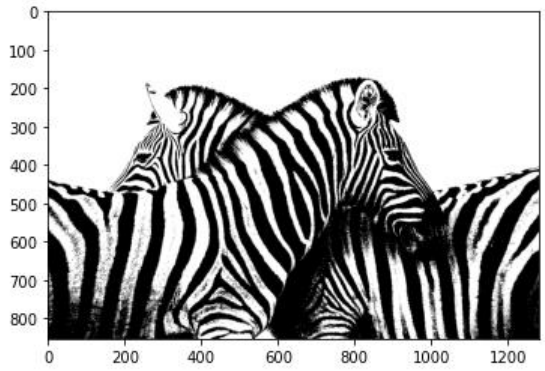
Utwórz nowy plik i wykonaj poniższe ćwiczenie

```
import cv2
import numpy as np
```

```
import matplotlib.pyplot as plt

img = cv2.imread("images/zebra.jpg")
img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

plt.imshow(img, cmap='gray')
plt.show()
```




```
# Przykład ustawienia progu i wartości max
thresh = 125
maxValue = 255
th, dst = cv2.threshold(img, thresh, maxValue, cv2.THRESH_BINARY)

plt.imshow(dst, cmap='gray')
plt.show()
```



- W ramach ćwiczenia zmodyfikuj wartość progu i powtórz czynności

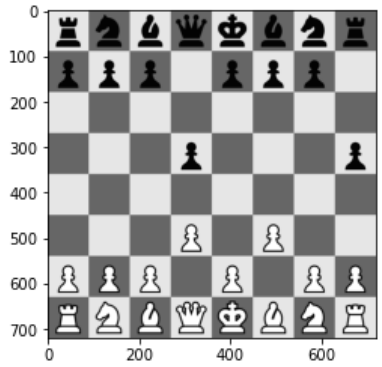
3.3 Operacje bitowe na obrazach

<table>
<tr><th>Bitwise Operation</th><th>Table</th></tr>
<tr><td>NOT
Used for generating the negative of a binary image.
Function: cv2.bitwise_not</td><td>
<table>
<tr><td>Input Bit</td><td>Output Bit</td></tr>
<tr><td>0</td><td>1</td></tr>
<tr><td>1</td><td>0</td></tr>
</table>
</td></tr>
<tr><td>OR
The OR operation will return a 1 if at least one of the images has a 1 in that pixel. This can be used to generate unions of two binary images.
Function: cv2.bitwise_or</td><td>
<table>
<tr><td>Input Bit 1</td><td>Input Bit 2</td><td>Output Bit</td></tr>
<tr><td>0</td><td>0</td><td>0</td></tr>
<tr><td>0</td><td>1</td><td>1</td></tr>
<tr><td>1</td><td>0</td><td>1</td></tr>
<tr><td>1</td><td>1</td><td>1</td></tr>
</table>
</td></tr>
<tr><td>AND
The AND operation will return a 1, but only if both of the images have a 1 in that specific pixel. This can be used to generate the intersection of two binary images.
Function: cv2.bitwise_and</td><td>
<table>
<tr><td>Input Bit 1</td><td>Input Bit 2</td><td>Output Bit</td></tr>
<tr><td>0</td><td>0</td><td>0</td></tr>
<tr><td>0</td><td>1</td><td>0</td></tr>
<tr><td>1</td><td>0</td><td>0</td></tr>
<tr><td>1</td><td>1</td><td>1</td></tr>
</table>
</td></tr>
<tr><td>XOR
The XOR operation will return a 1, but only if one of the pixels is 1 for the images. This can be used to identify the moving object in two subsequent frames.
Function: cv2.bitwise_xor</td><td>
<table>
<tr><td>Input Bit 1</td><td>Input Bit 2</td><td>Output Bit</td></tr>
<tr><td>0</td><td>0</td><td>0</td></tr>
<tr><td>0</td><td>1</td><td>1</td></tr>
<tr><td>1</td><td>0</td><td>1</td></tr>
<tr><td>1</td><td>1</td><td>0</td></tr>
</table>
</td></tr>
</table>

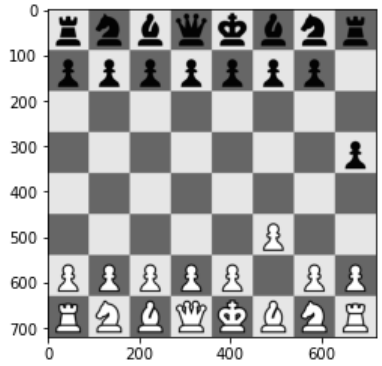
Zadanie. Przy pomocy funkcji XOR sprawdzić, które bierki szachowe ruszyły się podczas pewnego fragmentu tej samej gry.

Wskazany restart kernel

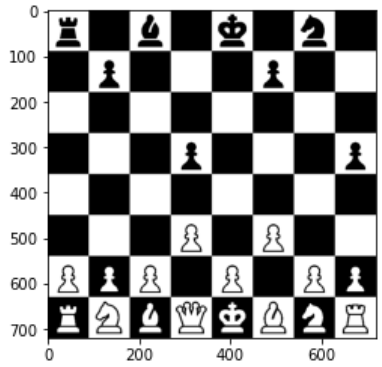
```
import cv2
import numpy as np
import matplotlib.pyplot as plt

# Wczytanie obrazów i konwersja do skali szarości
img1 = cv2.imread("images/board.png")
img2 = cv2.imread("images/board2.png")
img1 = cv2.cvtColor(img1, cv2.COLOR_BGR2GRAY)
img2 = cv2.cvtColor(img2, cv2.COLOR_BGR2GRAY)

plt.imshow(img1,cmap="gray")
plt.show()
```



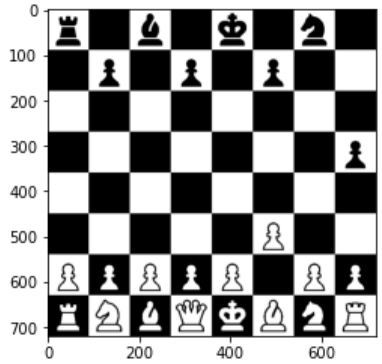

```
plt.imshow(img2,cmap="gray")
plt.show()
```



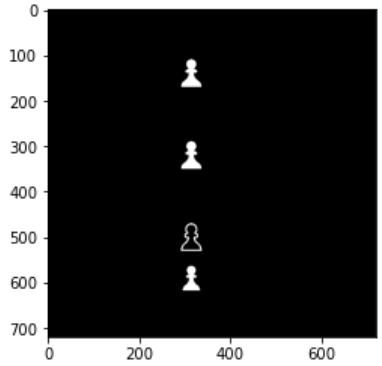

```
# Ustawienie progu i wartości max
thresh = 150
maxValue = 255
# Binarny próg
th, dst1 = cv2.threshold(img1, thresh, maxValue, cv2.THRESH_BINARY)
# Binary threshold
th, dst2 = cv2.threshold(img2, thresh, maxValue, cv2.THRESH_BINARY)


# Wyświetlenie binarnych obrazów przy pomocy matplotlib
plt.imshow(dst1, cmap='gray')
plt.show()
```




```
plt.imshow(dst2, cmap='gray')
plt.show()
```

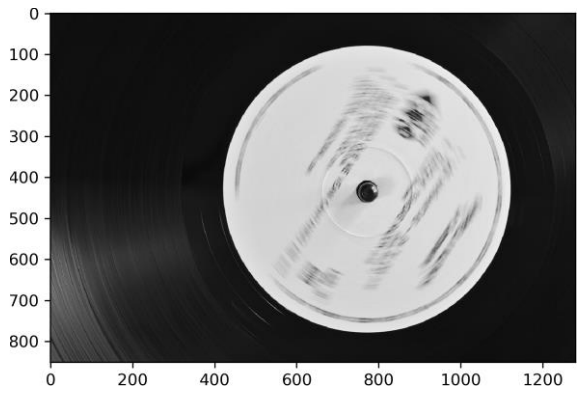



```
# Użycie operacji bitowej XOR aby znaleźć ruszone bierki
dst = cv2.bitwise_xor(dst1,dst2)

# Wyświetlenie wyniku
plt.imshow(dst, cmap='gray')
plt.show()
```

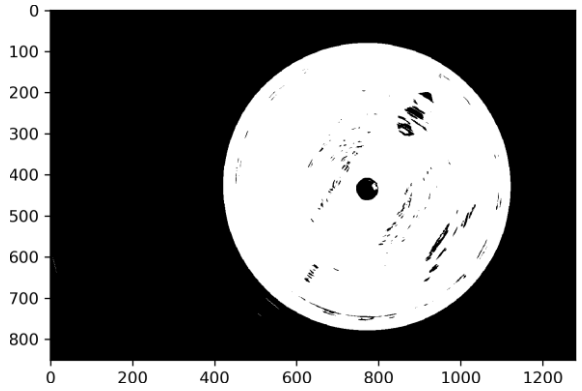


3.4 Maskowanie

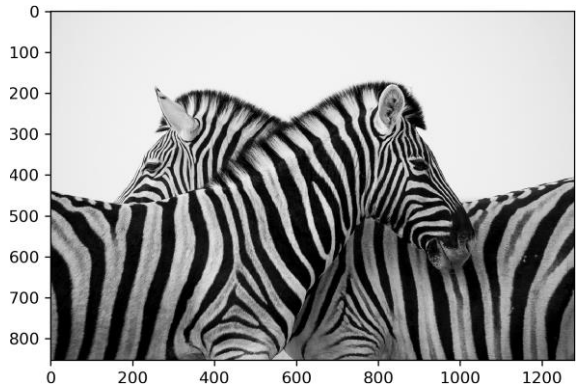
- Omówmy ostatnią koncepcję związaną z obrazami binarnymi.
- Obrazy binarne są dość często używane jako maska.
- Rozważmy na przykład następujący obraz. Użyjemy obrazu płyty gramofonowej:



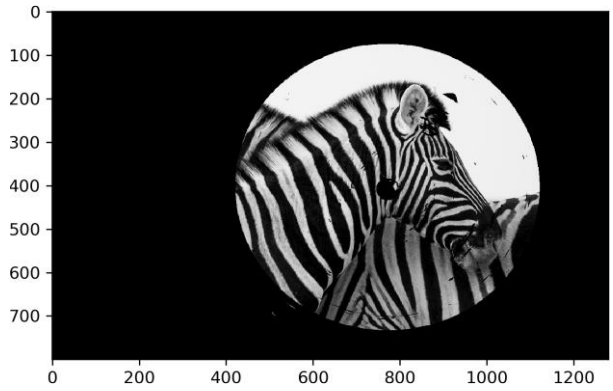
- Po progowaniu maska będzie wyglądała tak:



Co się stanie jeżeli zastosujemy maskowanie na obrazie z zebrami:



- Oto wynik:



- Do maskowania używana jest funkcja `numpy` o nazwie `where()`:

```
result = np.where(mask, image, 0)
```

- Funkcja `np.where` NumPy mówi, że gdziekolwiek maska (pierwszy argument) jest niezerowa, zwraca wartość obrazu (drugi argument); w przeciwnym razie zwróć 0 (trzeci argument).

3.5 Samodzielne ćwiczenie:

W tym ćwiczeniu będziesz używać maskowania i innych czynności, które wykonywałeś w tym i poprzednim ćwiczeniu, aby odtworzyć wynik pokazany na rysunku maskowania zebr. Będziemy używać koncepcji zmiany rozmiaru obrazu, progowania i maskowania obrazu, aby wyświetlić tylko

głowy zebr. Podobną koncepcję można zastosować do tworzenia ładnych portretów zdjęć, na których widoczna jest tylko twarz osoby, a reszta regionu/tła jest zaciemniona.

Kroki, które musisz zrobić, aby wykonać to ćwiczenie:

1. Utwórz nowy plik.
2. Zaimportuj potrzebne biblioteki – OpenCV, NumPy i Matplotlib.
3. Odczytaj plik **recording.jpg** i przekonwertuj go do skali szarości.
4. Wykonaj progowanie używając progu o wartości 150 i wartości max=255.
5. Odczytaj plik z zebrami **zebras.jpg** i przekonwertuj go do skali szarości
6. Wydrukuj kształty obu obrazów (zebr i płyty).
7. Zauważysz, że obrazy mają różne wymiary. Zmień rozmiar obu obrazów do 1280×800 pikseli. Oznacza to, że szerokość obrazu o zmienionym rozmiarze powinna wynosić 1280 pikseli, a wysokość powinna wynosić 800 pikseli. Trzeba będzie użyć funkcji **cv2.resize** do zmiany rozmiaru. Użyj interpolacji liniowej podczas zmiany rozmiaru obrazów.
8. Następnie użyj polecenia **where** NumPy, aby zachować tylko piksele, w których piksele płyty są białe. Pozostałe piksele należy zastąpić kolorem czarnym.
9. Wynik powinien wyglądać tak: