Lecture 02
Digital Image Basics

2024-02-01

Sébastien Valade

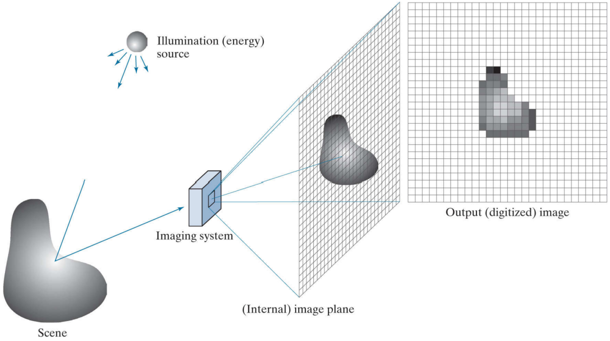
VNIVERSIDAD NACIONAL
AVTONOMA DE
MEXICO

1. What is a digital image?
    1. image acquisition
    2. sampling and quantization
    3. 3D projection on 2D plane
    4. color image
    5. color spaces
    6. image histogram
2. Point operations
3. Image processing levels
4. Image manipulation with Python

1.1. image acquisition

1. energy from an **illumination source** is reflected from a **scene**
2. the **imaging system** collects the incoming energy and focuses it onto an **image plane**

   NB: light-sensing instruments typically use 2-D arrays of photosensors to record incoming light intensity I(x): the CCD (*Charge-Coupled Device*)
3. the image plane is sampled and quantized to produce a **digital image**

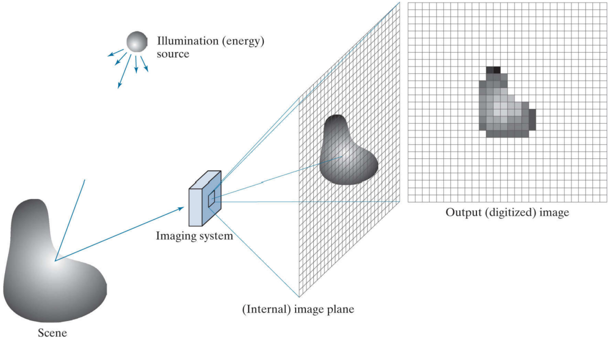


Credit: Gonzalez & Woods 2018

1.1. image acquisition

1. energy from an **illumination source** is reflected from a **scene**
2. the **imaging system** collects the incoming energy and focuses it onto an **image plane**

   NB: light-sensing instruments typically use 2-D arrays of photosensors to record incoming light intensity I(x): the CCD (*Charge-Coupled Device*)
3. the image plane is sampled and quantized to produce a **digital image**

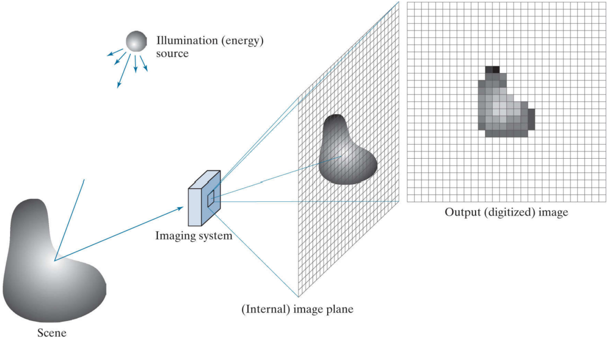


Credit: Gonzalez & Woods 2018

1.1. image acquisition

1. energy from an **illumination source** is reflected from a **scene**
2. the **imaging system** collects the incoming energy and focuses it onto an **image plane**

   NB: light-sensing instruments typically use 2-D arrays of photosensors to record incoming light intensity I(x): the CCD (*Charge-Coupled Device*)
3. the image plane is sampled and quantized to produce a **digital image**

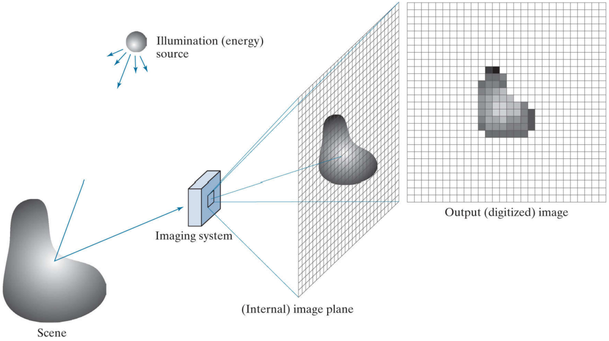


Credit: Gonzalez & Woods 2018

1.1. image acquisition

1. energy from an **illumination source** is reflected from a **scene**
2. the **imaging system** collects the incoming energy and focuses it onto an **image plane**

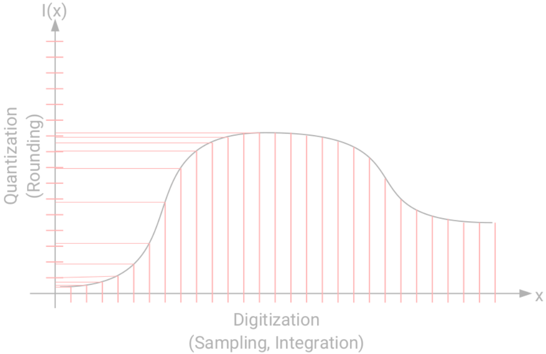
   NB: light-sensing instruments typically use 2-D arrays of photosensors to record incoming light intensity I(x): the CCD (*Charge-Coupled Device*)
3. the image plane is sampled and quantized to produce a **digital image**



Credit: Gonzalez & Woods 2018

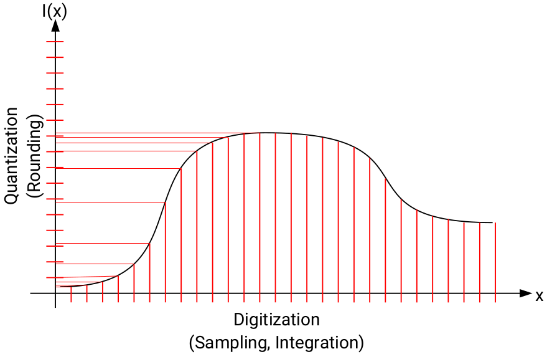
1.2. sampling and quantization

- each photosensor records incident light
- digitalization of an analog signal involves two operations
  - **spatial sampling** (= discretization of space domain)
  - **intensity quantization** (= discretization of incoming light signal)



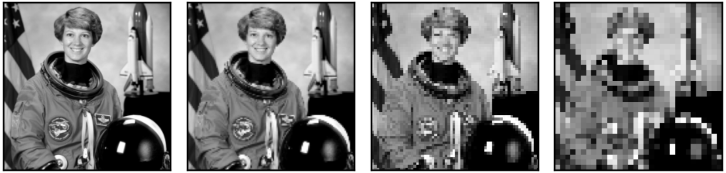
1.2. sampling and quantization

- each photosensor records incident light
- digitalization of an analog signal involves two operations
  - **spatial sampling** (= discretization of space domain)
  - **intensity quantization** (= discretization of incoming light signal)



spatial sampling (= discretization of space domain)

⇒ smallest element resulting from the discretization of the space is called a pixel (=picture element)



(512, 512)

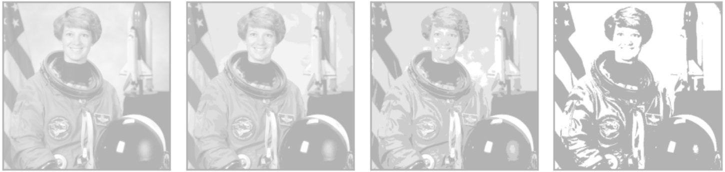
(128, 128)

(64, 64)

(32, 32)

intensity quantization (= discretization of light intensity signal)

⇒ typically, 256 levels (8 bits/pixel = 2^8 values) suffices to represent the intensity



8-bit resolution
2^8 = 256 gray levels

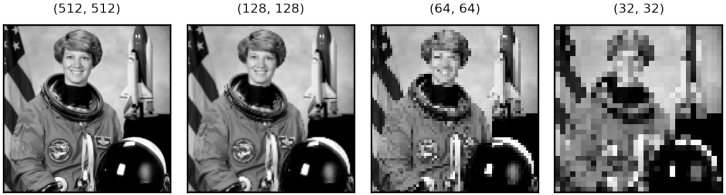
3-bit resolution
2^3 = 8 gray levels

2-bit resolution
2^2 = 4 gray levels

1-bit resolution
2^1 = 2 gray levels

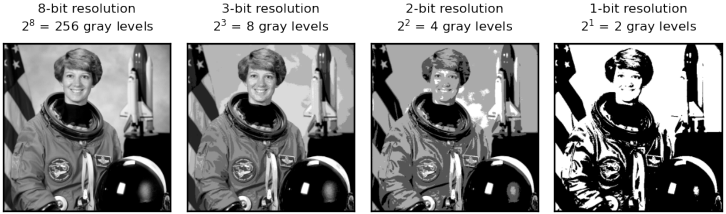
spatial sampling (= discretization of space domain)

⇒ smallest element resulting from the discretization of the space is called a pixel (=picture element)



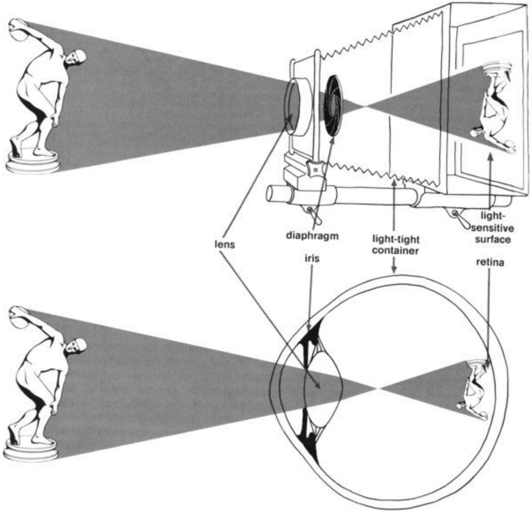
intensity quantization (= discretization of light intensity signal)

⇒ typically, 256 levels (8 bits/pixel = 2^8 values) suffices to represent the intensity



1.3. 3D projection on 2D plane

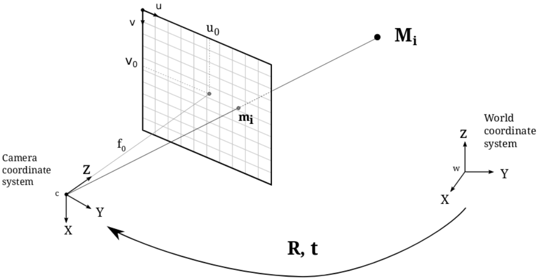
But how is the 3D world projected on a 2D plane?
⇒ comparison between human eye and pinhole camera:



1.3. 3D projection on 2D plane

Image = 3D world projection on 2D

⇒ projection using the **pinhole camera** model:



(from PyTorch Geometry)

Perspective transformation:

$$s\ m' = K[R|t]M' \tag{1}$$

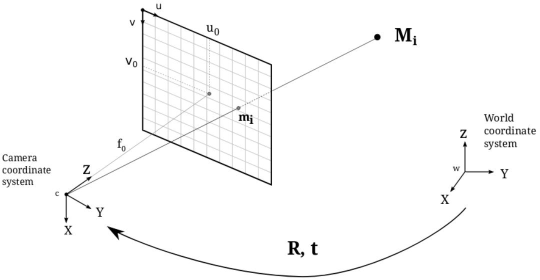
$$s \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & u_0 \\ 0 & f_y & v_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \tag{2}$$

where:

- M' = 3D point in space with coordinates $[X, Y, Z]^T$ expressed in Euclidean coordinates
- m' = projection of the 3D point M' onto the image plane with coordinates $[u, v]^T$ expressed in pixel units
- K = camera calibration matrix (a.k.a instrinsics parameters matrix)
  - fx, fy = focal lengths expressed in pixel units
  - u_0, v_0 = coordinates of the optical center (aka principal point), origin in the image plane
- $[R|t]$ = joint rotation-translation matrix (a.k.a. extrinsics parameters matrix), describing the camera pose, and translating from world coordinates to camera coordinates

Image = 3D world projection on 2D

⇒ projection using the **pinhole camera** model:



(from PyTorch Geometry)

Perspective transformation:

$$s\ m' = K[R|t]M' \tag{1}$$

$$s\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & u_0 \\ 0 & f_y & v_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \tag{2}$$

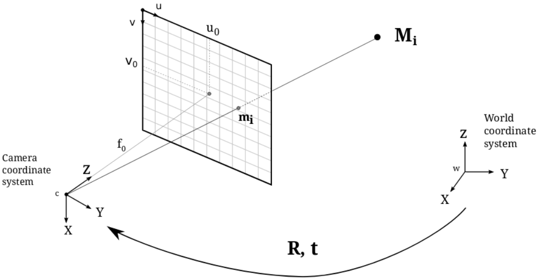
where:

- M' = 3D point in space with coordinates $[X, Y, Z]^T$ expressed in Euclidean coordinates
- m' = projection of the 3D point M' onto the image plane with coordinates $[u, v]^T$ expressed in pixel units
- K = camera calibration matrix (a.k.a instrinsics parameters matrix)
  - fx, fy = focal lengths expressed in pixel units
  - u_0, v_0 = coordinates of the optical center (aka principal point), origin in the image plane
- $[R|t]$ = joint rotation-translation matrix (a.k.a. extrinsics parameters matrix), describing the camera pose, and translating from world coordinates to camera coordinates

1.3. 3D projection on 2D plane

Image = 3D world projection on 2D

⇒ projection using the **pinhole camera** model:



(from PyTorch Geometry)

Perspective transformation:

$$s\ m' = K[R|t]M' \tag{1}$$

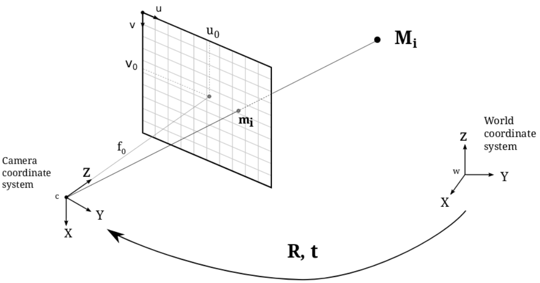
$$s\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & u_0 \\ 0 & f_y & v_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \tag{2}$$

where:

- M' = 3D point in space with coordinates $[X, Y, Z]^T$ expressed in Euclidean coordinates
- m' = projection of the 3D point M' onto the image plane with coordinates $[u, v]^T$ expressed in pixel units
- K = camera calibration matrix (a.k.a instrinsics parameters matrix)
  - fx, fy = focal lengths expressed in pixel units
  - u_0, v_0 = coordinates of the optical center (aka principal point), origin in the image plane
- $[R|t]$ = joint rotation-translation matrix (a.k.a. extrinsics parameters matrix), describing the camera pose, and translating from world coordinates to camera coordinates

Image = 3D world projection on 2D

⇒ projection using the **pinhole camera** model:



(from PyTorch Geometry)

Perspective transformation:

$$s\ m' = K[R|t]M' \tag{1}$$

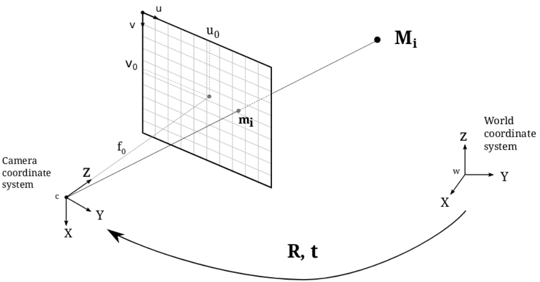
$$s\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & u_0 \\ 0 & f_y & v_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \tag{2}$$

where:

- M' = 3D point in space with coordinates $[X, Y, Z]^T$ expressed in Euclidean coordinates
- m' = projection of the 3D point M' onto the image plane with coordinates $[u, v]^T$ expressed in pixel units
- K = camera calibration matrix (a.k.a instrinsics parameters matrix)
  - fx, fy = focal lengths expressed in pixel units
  - u_0, v_0 = coordinates of the optical center (aka principal point), origin in the image plane
- $[R|t]$ = joint rotation-translation matrix (a.k.a. extrinsics parameters matrix), describing the camera pose, and translating from world coordinates to camera coordinates

Image $=$ 3D world projection on 2D

$\Rightarrow$ projection using the **pinhole camera** model:



(from PyTorch Geometry)

Perspective transformation:

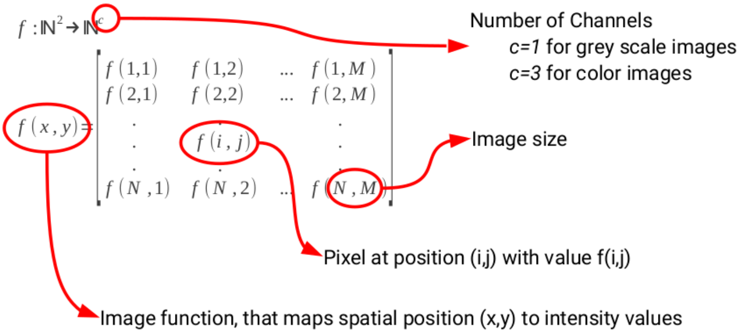
$$s\ m' = K[R|t]M' \tag{1}$$

$$s\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & u_0 \\ 0 & f_y & v_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \tag{2}$$

where:

- M' $=$ 3D point in space with coordinates $[X, Y, Z]^T$ expressed in Euclidean coordinates
- m' $=$ projection of the 3D point M' onto the image plane with coordinates $[u, v]^T$ expressed in pixel units
- K $=$ camera calibration matrix (a.k.a instrinsics parameters matrix)
    - fx, fy $=$ focal lengths expressed in pixel units
    - u_0, v_0 $=$ coordinates of the optical center (aka principal point), origin in the image plane
- $[R|t]$ $=$ joint rotation-translation matrix (a.k.a. extrinsics parameters matrix), describing the camera pose, and translating from world coordinates to camera coordinates

⇒ digital image function $f(x, y)$



$$f : \mathbb{N}^2 \rightarrow \mathbb{N}^c$$

$$f(x,y) = \begin{bmatrix} f(1,1) & f(1,2) & \dots & f(1,M) \\ f(2,1) & f(2,2) & \dots & f(2,M) \\ \vdots & f(i,j) & & \vdots \\ f(N,1) & f(N,2) & \dots & f(N,M) \end{bmatrix}$$

Number of Channels

c=1 for grey scale images

c=3 for color images

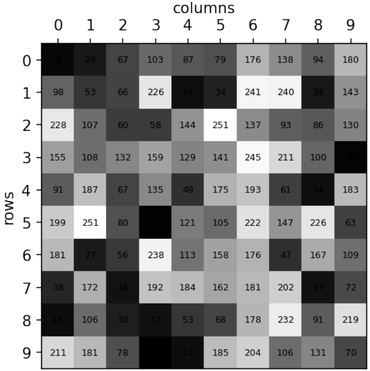
Image size

Pixel at position (i,j) with value f(i,j)

Image function, that maps spatial position (x,y) to intensity values

1.3. 3D projection on 2D plane

⇒ digital image function $f(x, y)$



	0	1	2	3	4	5	6	7	8	9
0	8	24	67	103	87	79	176	138	94	180
1	98	53	66	226	14	34	241	240	24	143
2	228	107	60	58	144	251	137	93	86	130
3	155	108	132	159	129	141	245	211	100	4
4	91	187	67	135	49	175	193	61	14	183
5	199	251	80	2	121	105	222	147	226	63
6	181	27	56	238	113	158	176	47	167	109
7	38	172	18	192	184	162	181	202	17	72
8	13	106	30	17	53	68	178	232	91	219
9	211	181	78		13	185	204	106	131	70

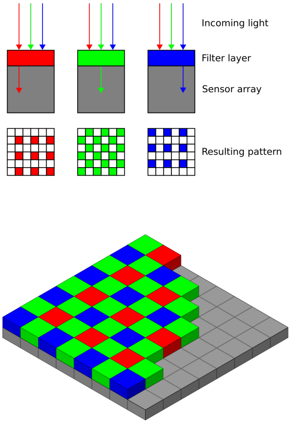
Typical ranges:

- uint8 = [0-255]
  (8 bits = 1 byte = 2^8 = 256 values per pixel)
- float32 = [0-1]
  (32 bits = 4 bytes = 4.3e9 values per pixel)

1.4. color image

How do we record colors?

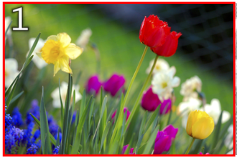
⇒ **Bayer Filter**: color filter array for arranging RGB color filters on a square grid of photosensors

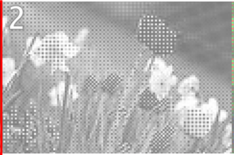


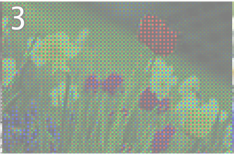
(source wikipedia)

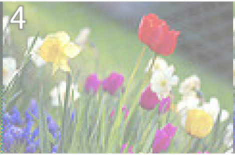
How do we record colors?

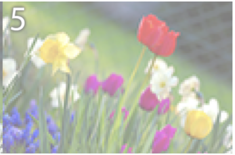
⇒ **Bayer Filter**: color filter array for arranging RGB color filters on a square grid of photosensors









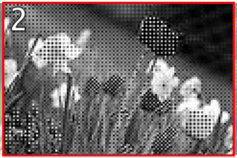


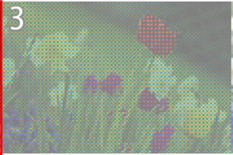
1. Original scene
2. Output of a 120×80-pixel sensor with a Bayer filter
3. Output color-coded with Bayer filter colors
4. Reconstructed image after interpolating missing color information (a.k.a. demosaicing)
5. Full RGB version at 120×80-pixels for comparison

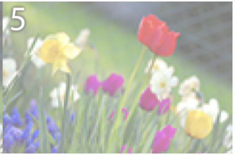
1.4. color image

How do we record colors?

⇒ **Bayer Filter**: color filter array for arranging RGB color filters on a square grid of photosensors



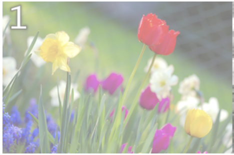


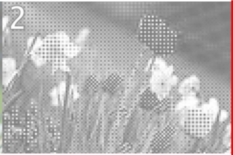


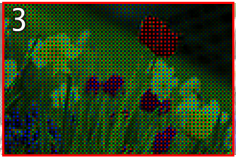
1. Original scene
2. Output of a 120×80-pixel sensor with a Bayer filter
3. Output color-coded with Bayer filter colors
4. Reconstructed image after interpolating missing color information (a.k.a. demosaicing)
5. Full RGB version at 120×80-pixels for comparison

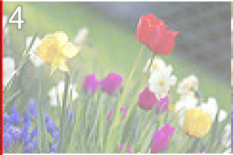
How do we record colors?

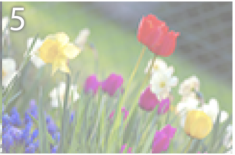
⇒ **Bayer Filter**: color filter array for arranging RGB color filters on a square grid of photosensors







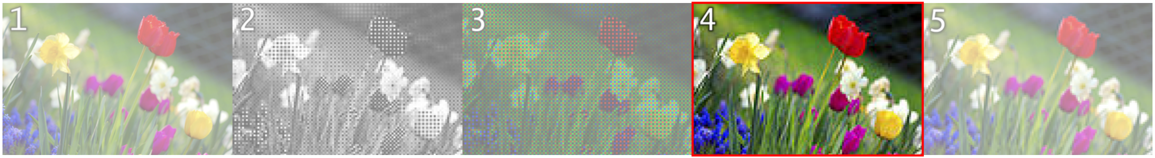




1. Original scene
2. Output of a 120×80-pixel sensor with a Bayer filter
3. Output color-coded with Bayer filter colors
4. Reconstructed image after interpolating missing color information (a.k.a. demosaicing)
5. Full RGB version at 120×80-pixels for comparison

How do we record colors?

⇒ **Bayer Filter**: color filter array for arranging RGB color filters on a square grid of photosensors

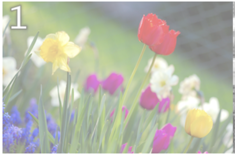


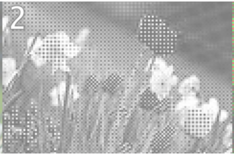
1. Original scene
2. Output of a 120×80-pixel sensor with a Bayer filter
3. Output color-coded with Bayer filter colors
4. Reconstructed image after interpolating missing color information (a.k.a. demosaicing)
5. Full RGB version at 120×80-pixels for comparison

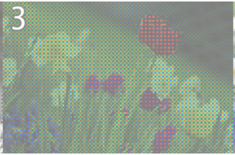
1.4. color image

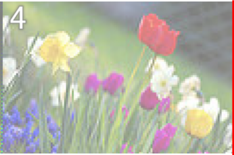
How do we record colors?

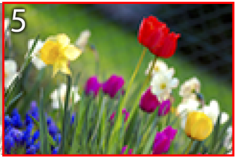
⇒ **Bayer Filter**: color filter array for arranging RGB color filters on a square grid of photosensors









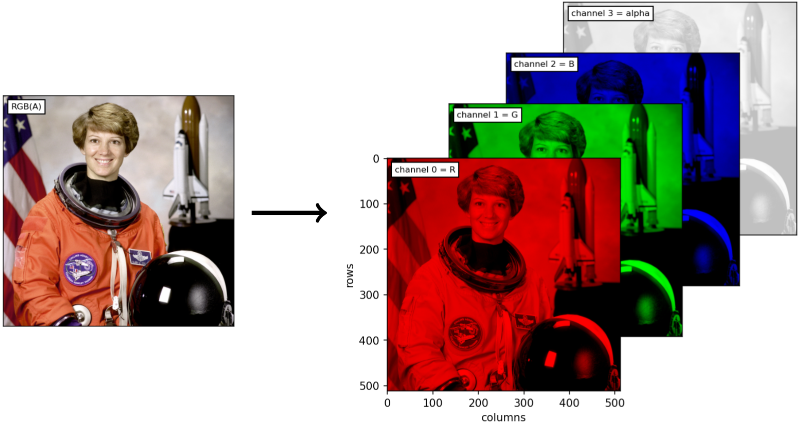


1. Original scene
2. Output of a 120×80-pixel sensor with a Bayer filter
3. Output color-coded with Bayer filter colors
4. Reconstructed image after interpolating missing color information (a.k.a. demosaicing)
5. Full RGB version at 120×80-pixels for comparison

1.4. color image

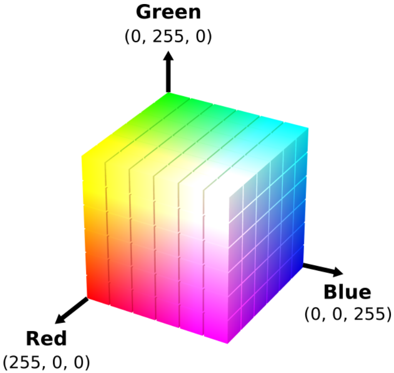
$\Rightarrow$ color image = 3D tensor in colorspace

- **RGB** = Red + Green + Blue bands (.JPEG)
- **RGBA** = Red + Green + Blue + Alpha bands (.PNG, .GIF, .BMP, TIFF, .JPEG 2000)

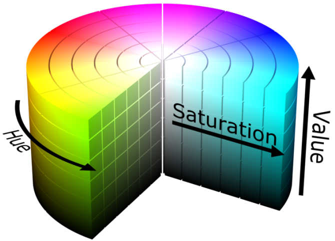


Other ways to represent the color information?

RGB colorspace



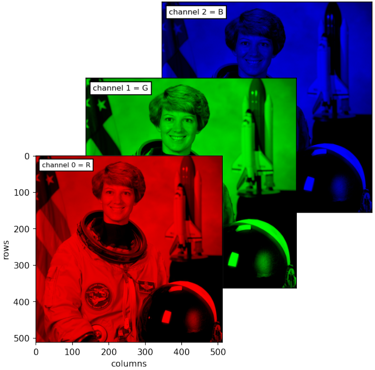
HSV colorspace



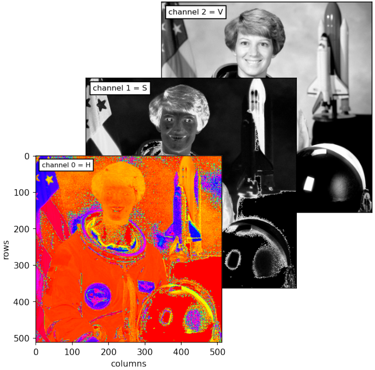
- Hue (H) = [0-360] ⇒ shift color
- Saturation (S) = [0-1] ⇒ shift intensity
- Value (V) = [0-1] ⇒ shift brightness

3D tensor with different information

RGB colorspace



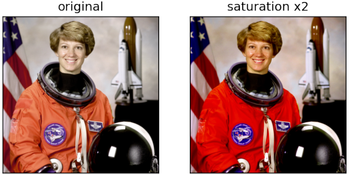
HSV colorspace



1.5. color spaces

- more saturation S

⇒ more intense colors



- more value V

⇒ brighter colors

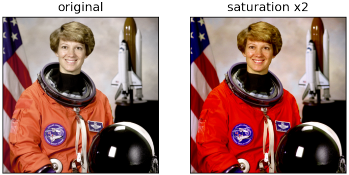
- shift hue H

⇒ shift color

1.5. color spaces

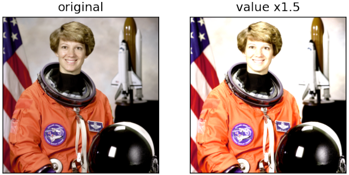
- more saturation S

⇒ more intense colors



- more value V

⇒ brighter colors



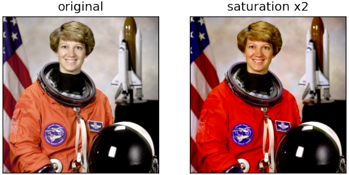
- shift hue H

⇒ shift color

1.5. color spaces

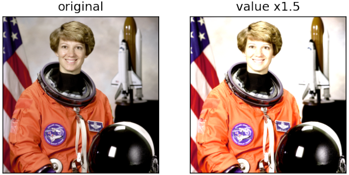
- more saturation S

⇒ more intense colors



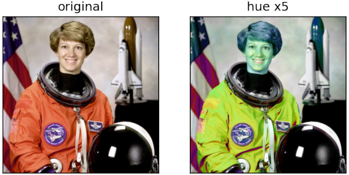
- more value V

⇒ brighter colors



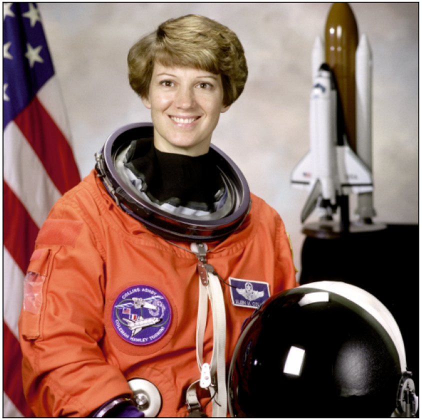
- shift hue H

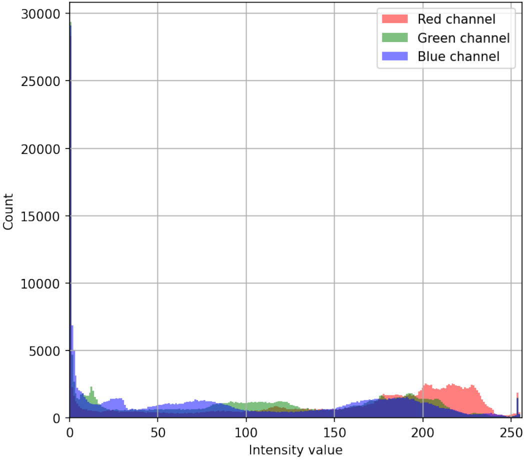
⇒ shift color



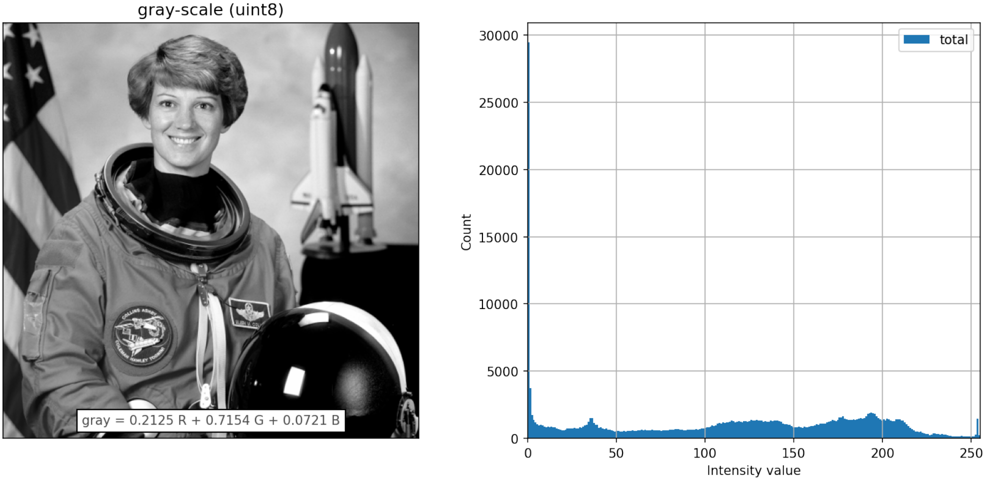
Histogram of pixel values in each band:

original (uint8)

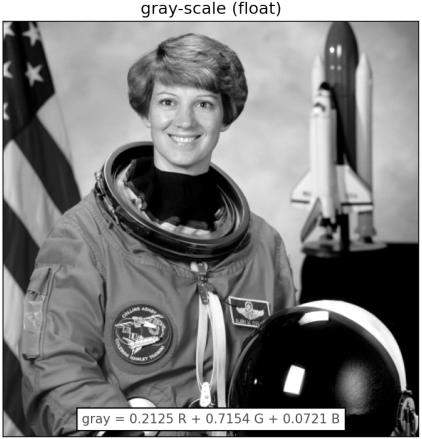


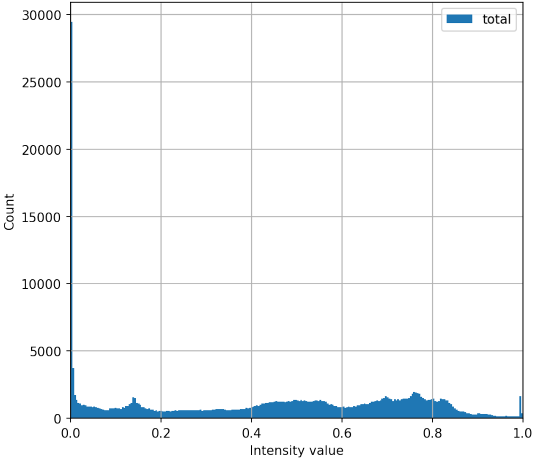


Histogram of pixel values after conversion from RGB (3-bands) to gray-scale (1-band):

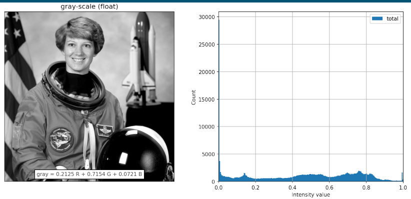


Histogram of pixel values after conversion to float values (range [0-1])





1.6. image histogram



- original gray-scale

- histogram rescale to 10-90 percentiles

⇒ contrast stretching

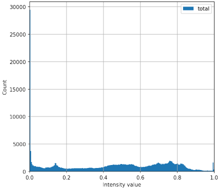
- histogram equalize

⇒ spread out the most frequent intensity values

1.6. image histogram

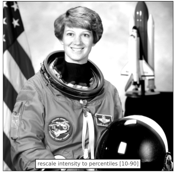
- original gray-scale

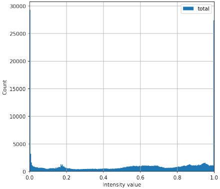
gray-scale (float)



- histogram rescale to 10-90 percentiles
⇒ contrast stretching

contrast stretching

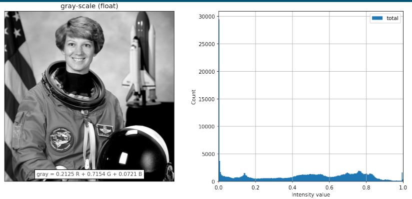




- histogram equalize
⇒ spread out the most frequent intensity values

1.6. image histogram

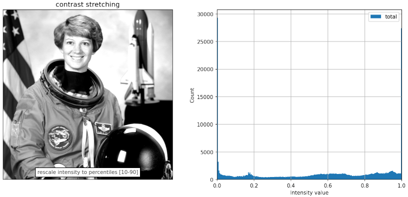
- original gray-scale



gray-scale (float)

- histogram rescale to 10-90 percentiles

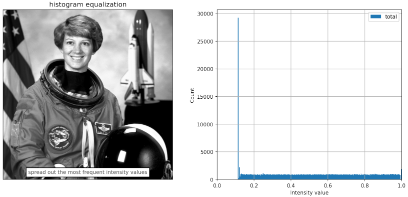
⇒ contrast stretching



contrast stretching

- histogram equalize

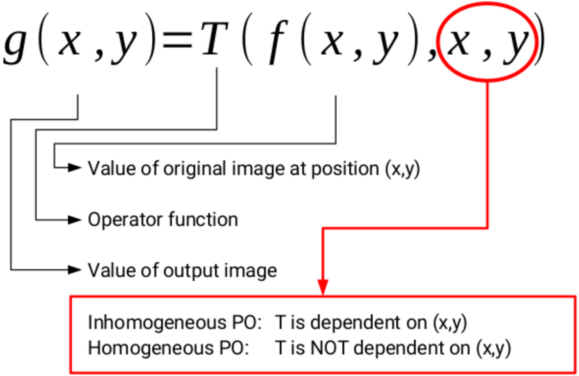
⇒ spread out the most frequent intensity values



histogram equalization

1. What is a digital image?
2. Point operations
    1. homogeneous point operations
    2. inhomogeneous Point Operations
3. Image processing levels
4. Image manipulation with Python

Point operations



$$g(x,y)=T(f(x,y),x,y)$$

Value of original image at position (x,y)

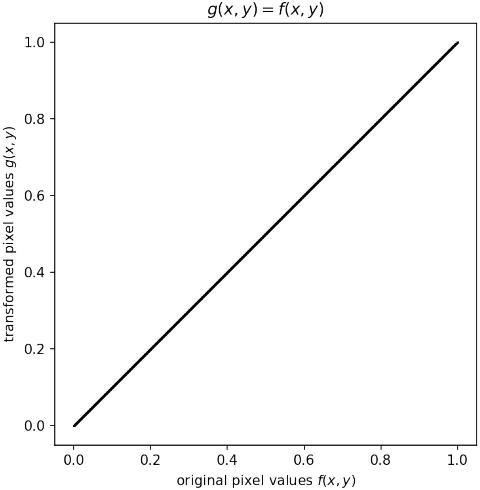
Operator function

Value of output image

Inhomogeneous PO: T is dependent on (x,y)
Homogeneous PO: T is NOT dependent on (x,y)

Homogeneous Point Operations (does not depend on pixel position)

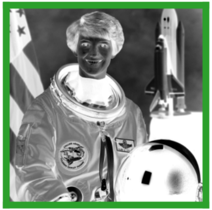
identity

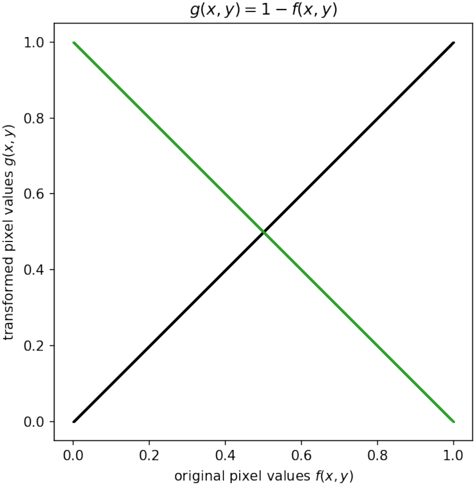

$g(x, y) = f(x, y)$
transformed pixel values $g(x, y)$
original pixel values $f(x, y)$
0.0
0.2
0.4
0.6
0.8
1.0
0.0
0.2
0.4
0.6
0.8
1.0

Homogeneous Point Operations (does not depend on pixel position)

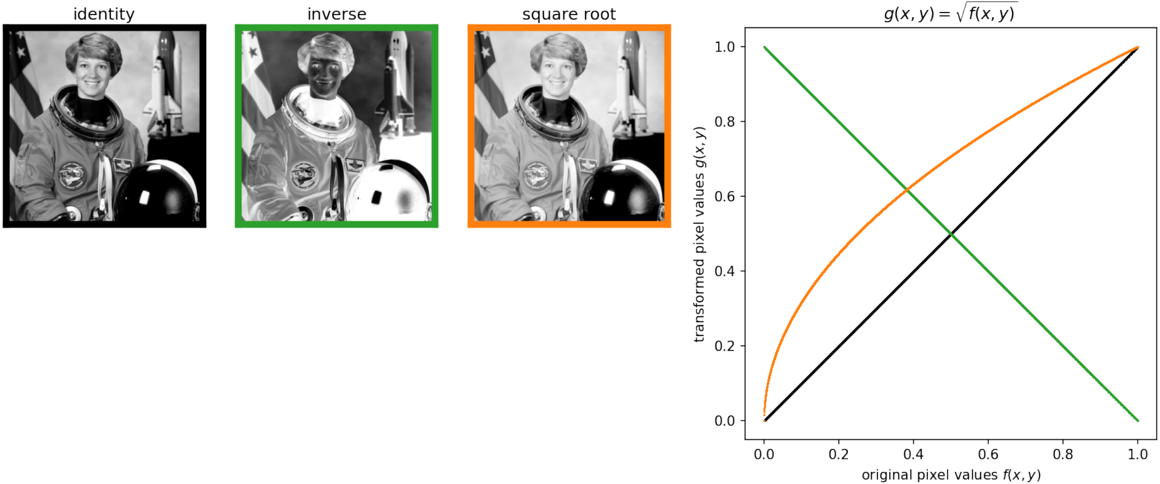
identity

inverse




$g(x, y) = 1 - f(x, y)$
transformed pixel values $g(x, y)$
original pixel values $f(x, y)$
0.0
0.2
0.4
0.6
0.8
1.0
0.0
0.2
0.4
0.6
0.8
1.0

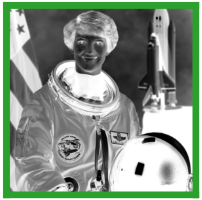
Homogeneous Point Operations (does not depend on pixel position)



Homogeneous Point Operations (does not depend on pixel position)

identity

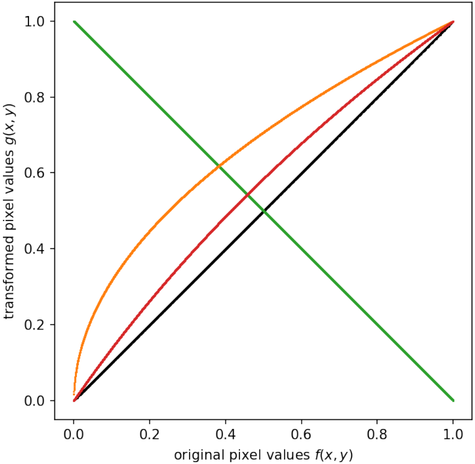
inverse



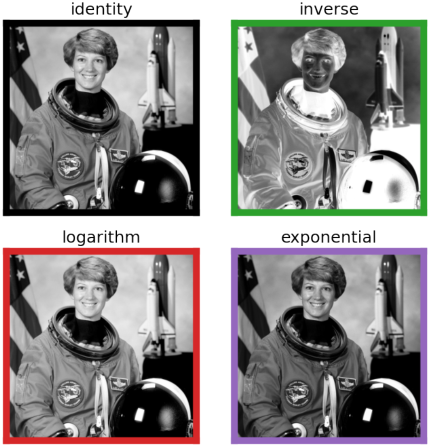
square root

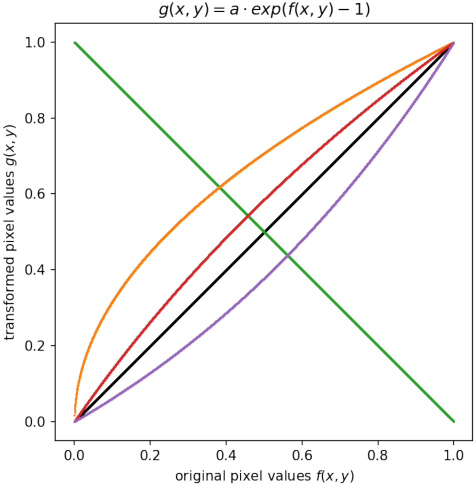
logarithm

$g(x, y) = a \cdot \log(f(x, y) + 1)$

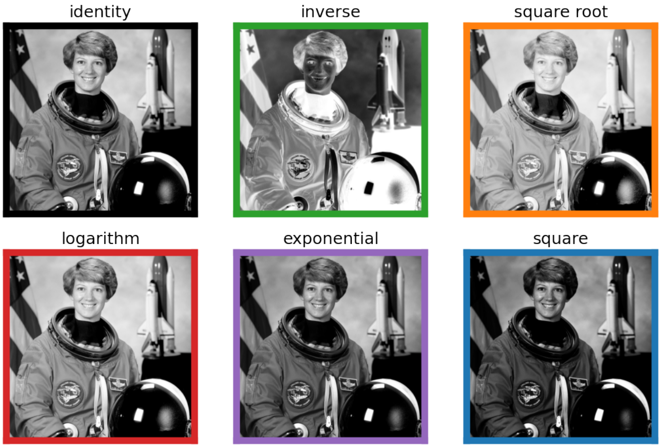


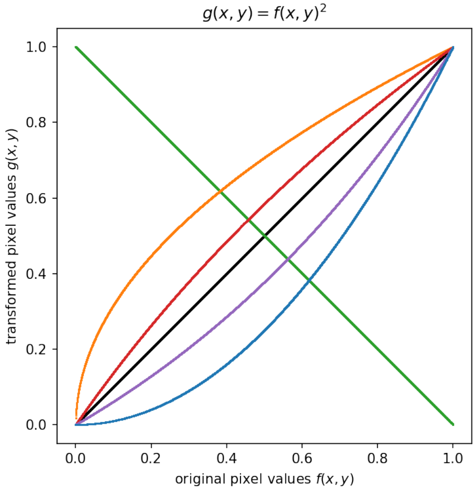
Homogeneous Point Operations (does not depend on pixel position)





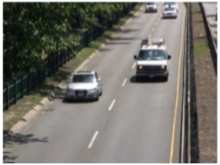
Homogeneous Point Operations (does not depend on pixel position)





Inhomogeneous Point Operations (depends on pixel position)

EX: background detection / change detection



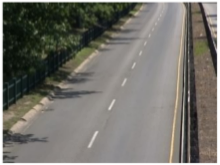
f_1

f_i

f_N

$$a(x,y)=\frac{1}{N}\sum_{i=0}^{N} f_i(x,y)$$

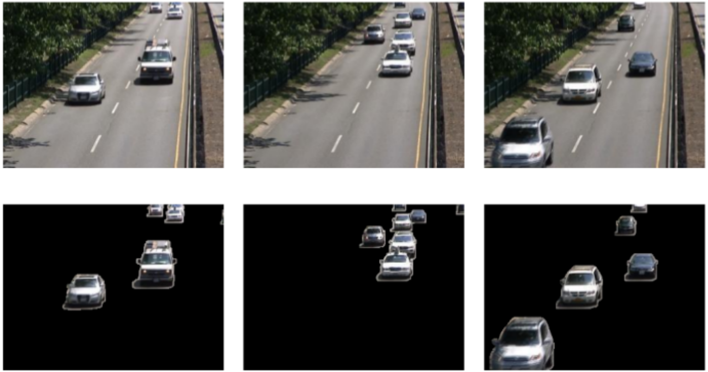
$a(x,y)$



$$\begin{aligned} g_i(x,y) &= T(f(x,y),x,y) \\ &= f_i(x,y)-a(x,y) \end{aligned}$$

Inhomogeneous Point Operations (depends on pixel position)

EX: background detection / change detection



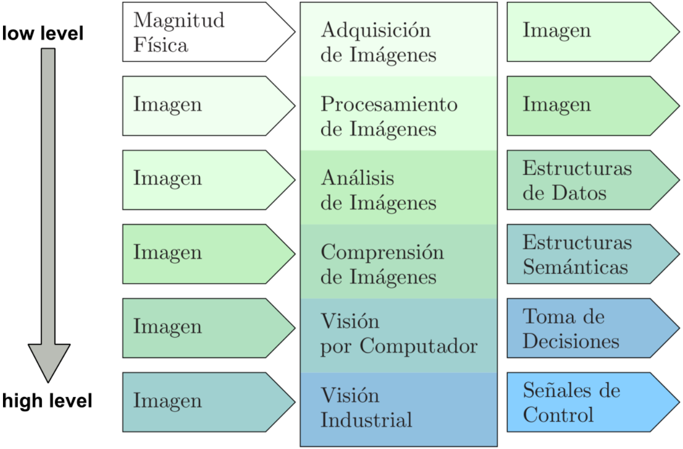
1. What is a digital image?

2. Point operations

3. Image processing levels

4. Image manipulation with Python

Image processing levels: inhomogeneous Point Operations



low level

high level

Magnitud Física	Adquisición de Imágenes	Imagen
Imagen	Procesamiento de Imágenes	Imagen
Imagen	Análisis de Imágenes	Estructuras de Datos
Imagen	Comprensión de Imágenes	Estructuras Semánticas
Imagen	Visión por Computador	Toma de Decisiones
Imagen	Visión Industrial	Señales de Control

Credit: Pablo Alvarado 2012

Image processing levels

Examples of processing levels:

- Low-level processing
  - image manipulation ⇒ *resizing, color adjustments, filtering, etc.*
  - feature extraction ⇒ *edges, gradients, etc.*
- Mid-level processing
  - panorama stitching
  - Structure from Motion (SfM) ⇒ 2D to 3D
  - Optical Flow ⇒ velocities
- High-level processing
  - classification ⇒ *what is in the image?*
  - detection ⇒ *where are they?*
  - segmentation (semantic or instance) ⇒ *segment image and give names*

Image processing levels

Examples of processing levels:

- Low-level processing
  - image manipulation ⇒ *resizing, color adjustments, filtering, etc.*
  - feature extraction ⇒ *edges, gradients, etc.*
- Mid-level processing
  - panorama stitching
  - Structure from Motion (SfM) ⇒ 2D to 3D
  - Optical Flow ⇒ velocities
- High-level processing
  - classification ⇒ *what is in the image?*
  - detection ⇒ *where are they?*
  - segmentation (semantic or instance) ⇒ *segment image and give names*

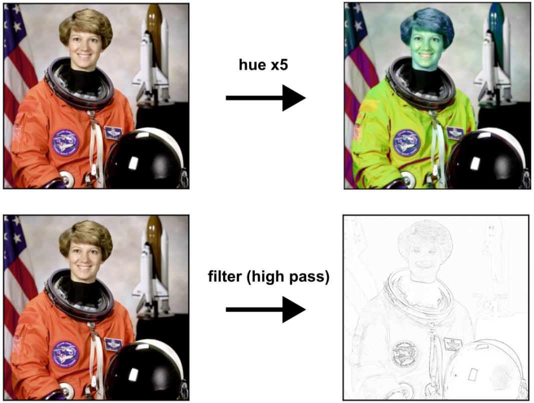
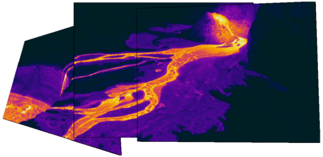


Image processing levels

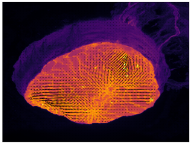
Examples of processing levels:

- Low-level processing
  - image manipulation ⇒ *resizing, color adjustments, filtering, etc.*
  - feature extraction ⇒ *edges, gradients, etc.*
- Mid-level processing
  - panorama stitching
  - Structure from Motion (SfM) ⇒ 2D to 3D
  - Optical Flow ⇒ velocities
- High-level processing
  - classification ⇒ *what is in the image?*
  - detection ⇒ *where are they?*
  - segmentation (semantic or instance) ⇒ *segment image and give names*

panorama stiching



Optical Flow (Farneback)



3D reconstruction

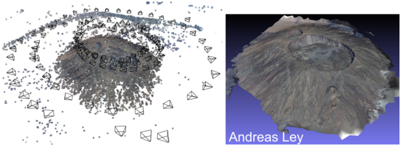
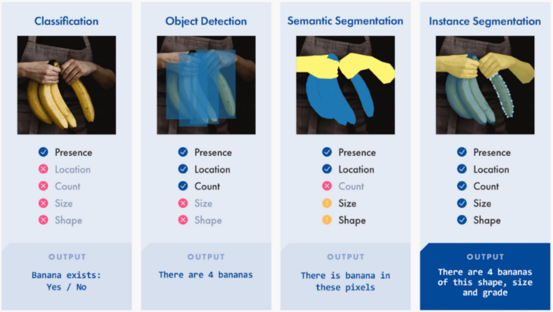


Image processing levels

Examples of processing levels:

- Low-level processing
  - image manipulation ⇒ *resizing, color adjustments, filtering, etc.*
  - feature extraction ⇒ *edges, gradients, etc.*
- Mid-level processing
  - panorama stitching
  - Structure from Motion (SfM) ⇒ 2D to 3D
  - Optical Flow ⇒ velocities
- High-level processing
  - classification ⇒ *what is in the image?*
  - detection ⇒ *where are they?*
  - segmentation (semantic or instance) ⇒ *segment image and give names*



Credit: cloudfactory

Image processing levels

Examples of processing levels:

- Low-level processing
  - image manipulation ⇒ *resizing, color adjustments, filtering, etc.*
  - feature extraction ⇒ *edges, gradients, etc.*
- Mid-level processing
  - panorama stitching
  - Structure from Motion (SfM) ⇒ 2D to 3D
  - Optical Flow ⇒ velocities
- High-level processing
  - classification ⇒ *what is in the image?*
  - detection ⇒ *where are they?*
  - segmentation (semantic or instance) ⇒ *segment image and give names*

1. What is a digital image?

2. Point operations

3. Image processing levels

4. Image manipulation with Python
    1. numpy tutorial
    2. exercises

4.1. numpy tutorial

Numpy tutorial:

⇒ Open DIP4RS_02_imagebasics/DIP4RS_02_numpy-tutorial.ipynb

4.2. exercises

Exercices:

⇒ Open DIP4RS_02_imagebasics/DIP4RS_02_exercices.ipynb