Lecture 02
Digital Image Basics

2024-08-21

Sébastien Valade

Computer Vision for Geosciences (CV4GS)

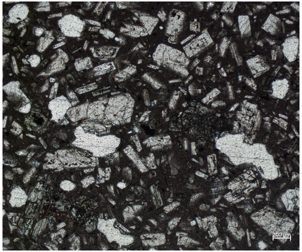
Computer Vision for Geosciences (CV4GS)

- **What is Computer Vision?**
  - ⇒ discipline focused on enabling computers to acquire, process, and interpret visual data, primarily from digital images or video

Computer Vision for Geosciences (CV4GS)

- **Sources of images for geosciences applications?**
  $\Rightarrow$ images can be derived from different imaging techniques at different scales

microscope ($\sim$ μm-scale)

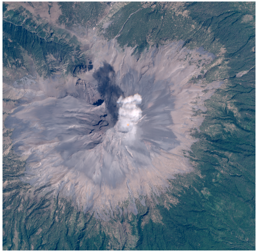


→

camera ($\sim$ m-scale)

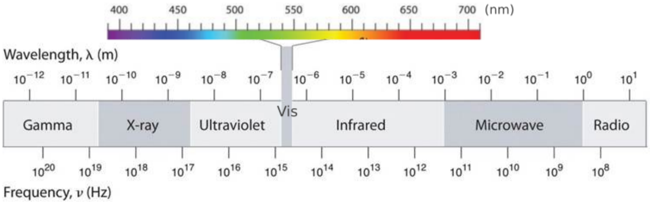
→

satellite ($\sim$ km-scale)

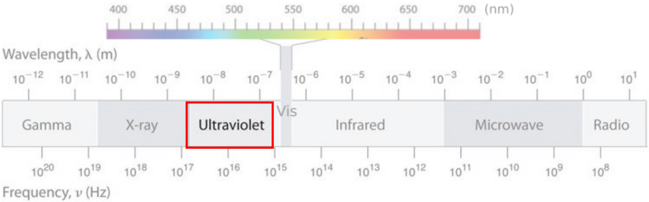


Computer Vision for Geosciences (CV4GS)

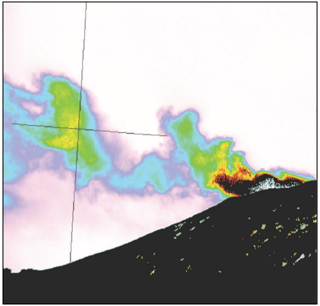
- **Sources of images for geosciences applications?**
  - ⇒ images can be constructed using different wavelengths spanning the entire electromagnetic spectra



1.1. Why Computer Vision for Geosciences?

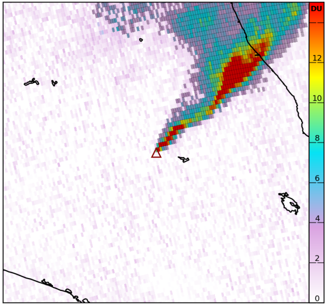


UV camera



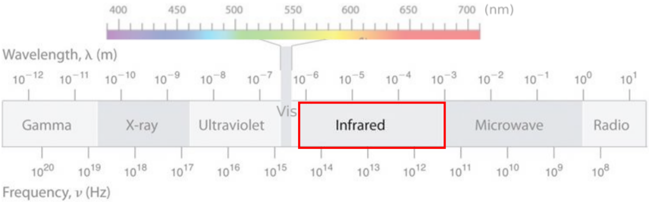
Popocatépetl 2013-01-29 (UV camera, Campion et al. 2018)

UV satellite

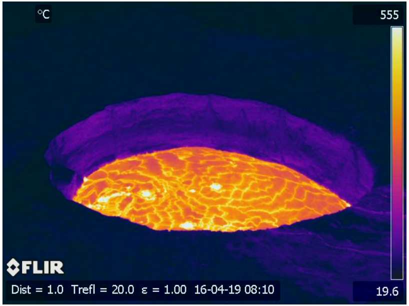


Popocatépetl 2019-02-17 (Sentinel-5P, MOUNTS)

1.1. Why Computer Vision for Geosciences?

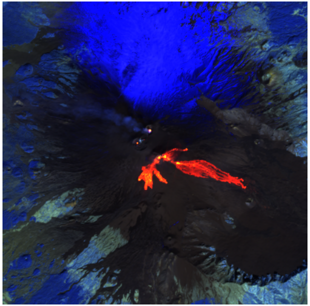


IR camera



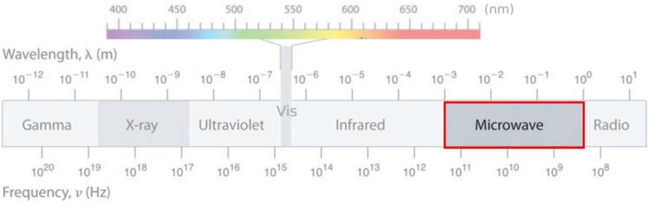
Nyiragongo 2016-04-16 (FLIR image, Valade et al. 2018)

IR satellite

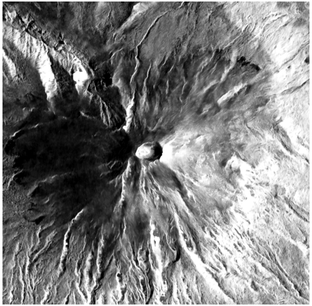


Etna 2021-02-23 (Sentinel-2 image, MOUNTS)

1.1. Why Computer Vision for Geosciences?

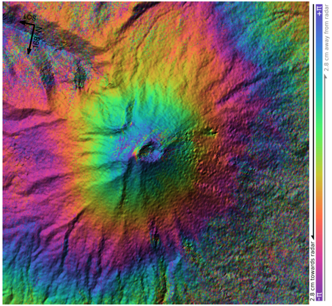


SAR satellite



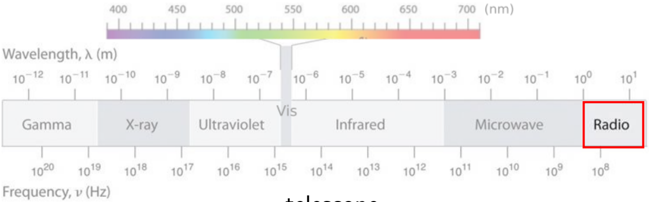
Popocatépetl SAR 2021-12-25 (Sentinel-1, MOUNTS)

InSAR satellite

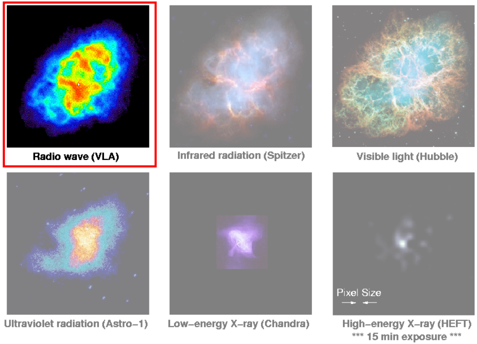


Popocatépetl InSAR interferogram 2021-12-25 dt=6 days (MOUNTS)

1.1. Why Computer Vision for Geosciences?

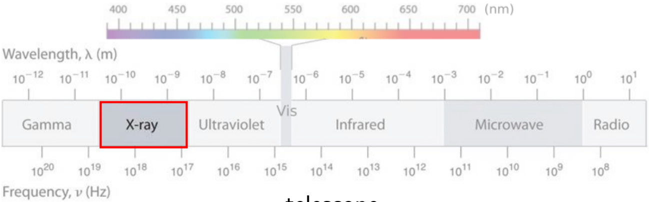


telescope

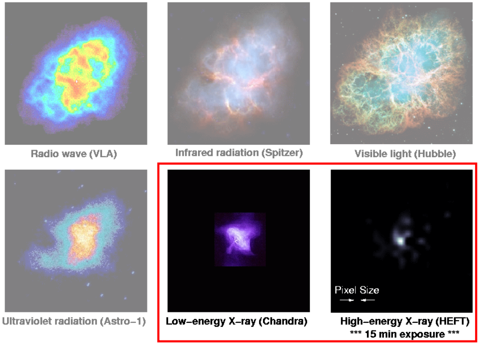


Crab Nebula - remanent of an exploded star (supernova)

1.1. Why Computer Vision for Geosciences?

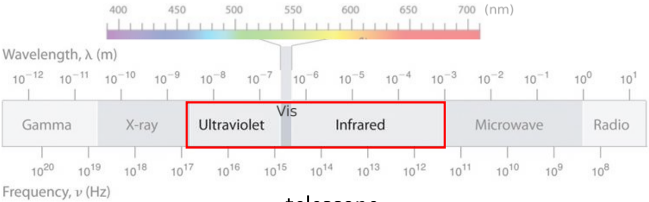


telescope

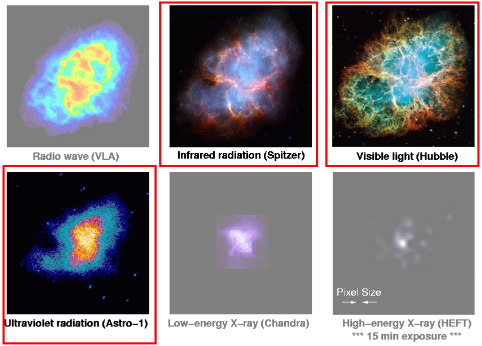


Crab Nebula - remanent of an exploded star (supernova)

1.1. Why Computer Vision for Geosciences?

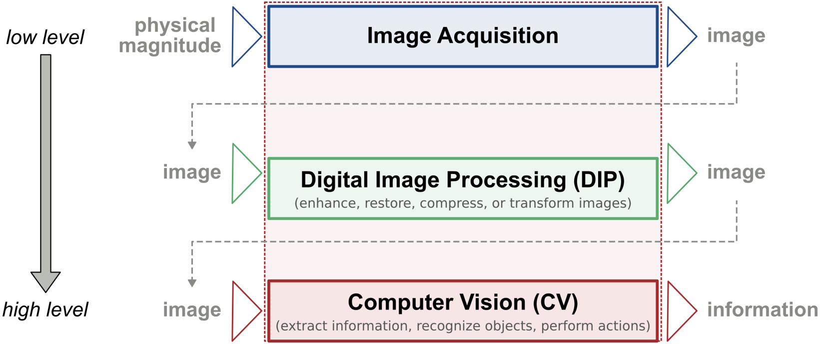


telescope



Crab Nebula - remanent of an exploded star (supernova)

From image acquisition to image processing:



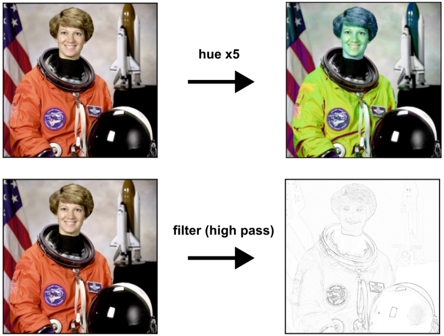
*"Computer Vision tasks include methods for **acquiring**, **processing**, **analyzing** and understanding digital images, and extraction of high-dimensional data from the real world in order to produce numerical or symbolic information, e.g., in the forms of decisions".*

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*

Examples of processing levels:

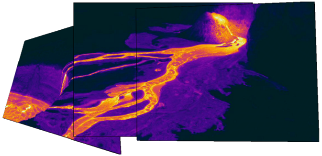
- Low-level processing
  - image manipulation ⇒ *resizing, color adjustments, filtering, etc.*
  - feature extraction ⇒ *edges, gradients, etc.*
- Mid-level processing
  - panorama stitching
  - Structure from Motion (SfM) ⇒ 2D to 3D
  - Optical Flow ⇒ velocities
- High-level processing
  - classification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



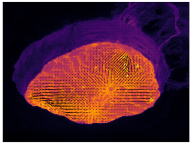
Examples of processing levels:

- 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*

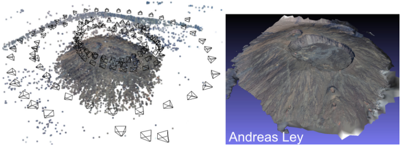
panorama stiching



Optical Flow (Farneback)

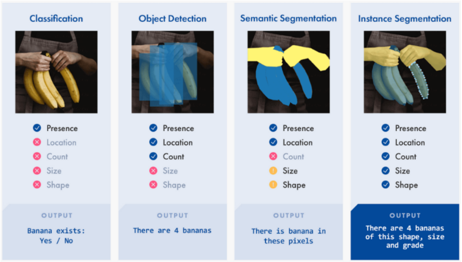


3D reconstruction



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*



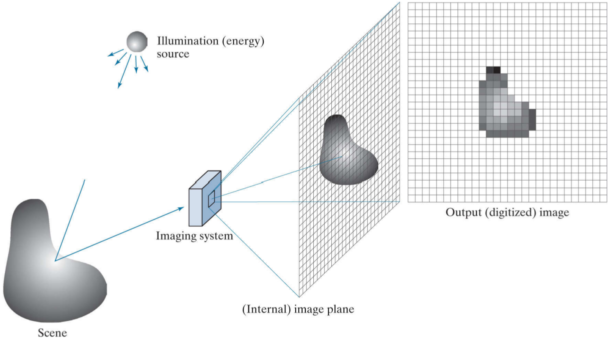
Credit: cloudfactory

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*

2.1. 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**



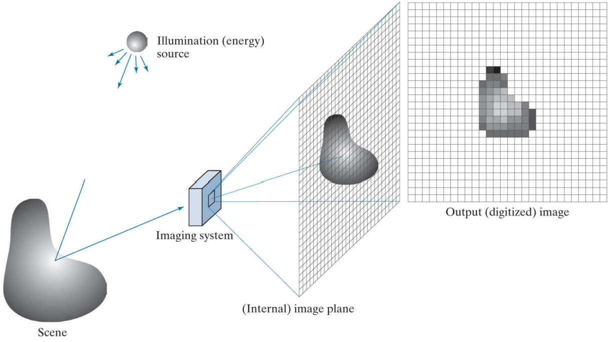
Credit: Gonzalez & Woods 2018

2.1. 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**

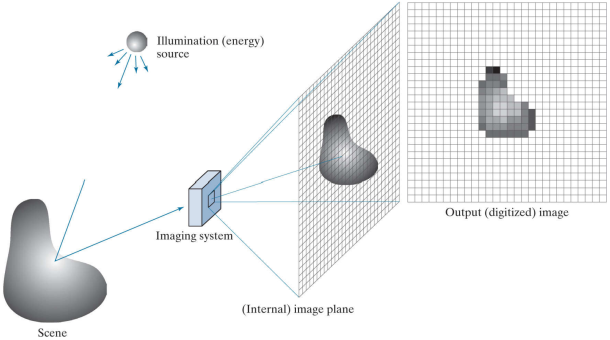


Credit: Gonzalez & Woods 2018

2.1. 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**

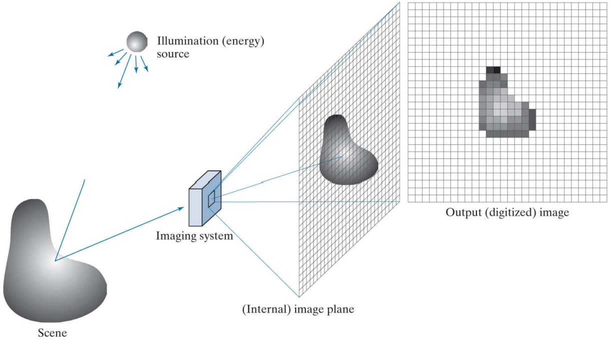


Credit: Gonzalez & Woods 2018

2.1. 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**

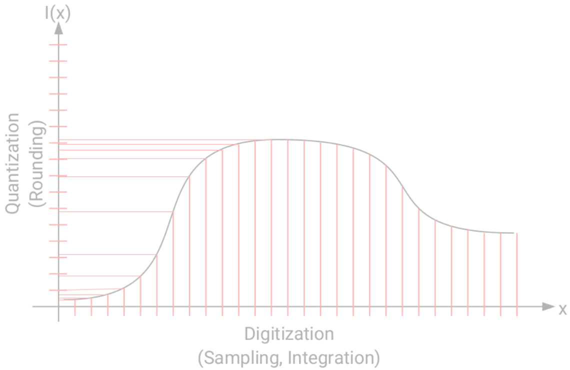
   NB: light-sensing instruments typically use 2-D arrays of photosensors to record incoming light intensity $I(x)$: the CCD (Charge-Coupled Device)
3. the image plane is sampled and quantized to produce a **digital image**



Credit: Gonzalez & Woods 2018

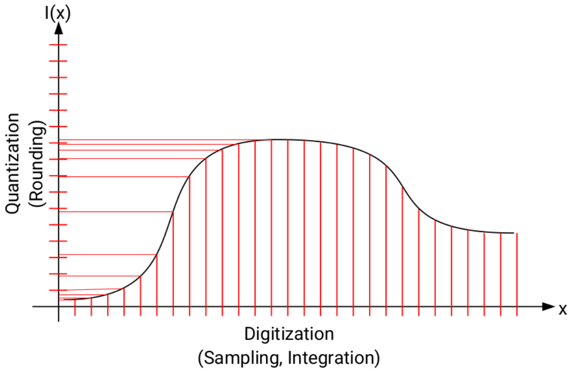
2.2. sampling and quantization

- 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)



2.2. 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)

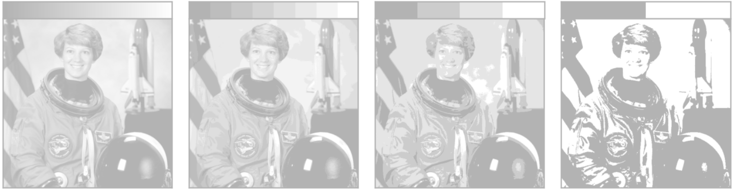
⇒ smallest element resulting from the discretization of the space is called a pixel (=picture element)

(512, 512) (128, 128) (64, 64) (32, 32)

intensity quantization (= discretization of light intensity signal)

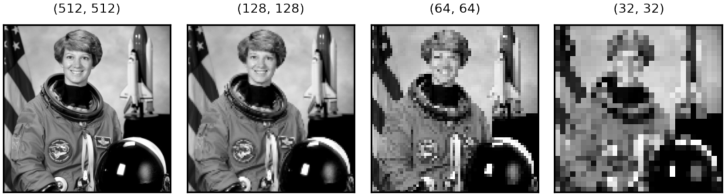
⇒ typically, 256 levels (8 bits/pixel = 2^8 values) suffices to represent the intensity

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



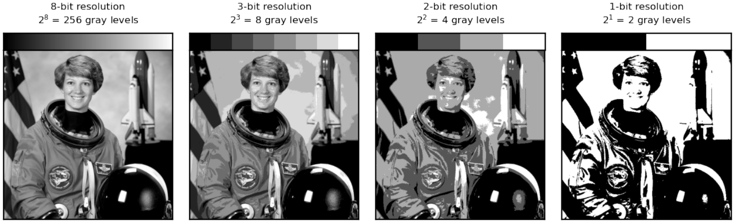
spatial sampling (= discretization of space domain)

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

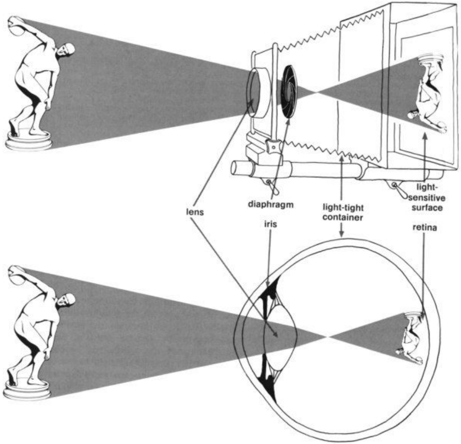


intensity quantization (= discretization of light intensity signal)

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

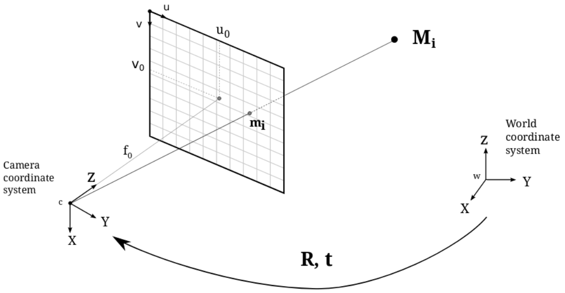


But how is the 3D world projected on a 2D plane?
$\Rightarrow$ comparison between human eye and pinhole camera:



In 1514, Leonardo da Vinci explained: "By letting the images of illuminated objects penetrate through a small hole into a very dark room, you will then intercept these images on a white sheet of paper placed in this room. [...] but they will be smaller and reversed".

Image = 3D world projection on 2D
$\Rightarrow$ projection using the **pinhole camera** model:



(from PyTorch Geometry)

Perspective transformation:

$$s\ m' = K[R|t]M' \quad (1)$$

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

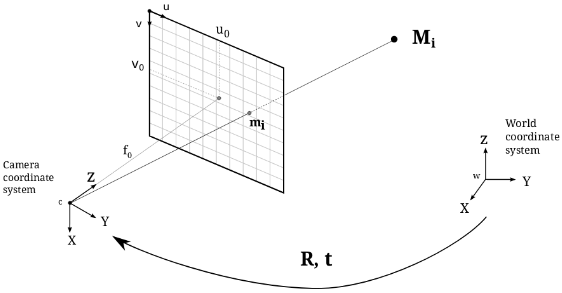
where:

- 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

2.3. 3D projection on 2D plane

Image = 3D world projection on 2D

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



(from PyTorch Geometry)

Perspective transformation:

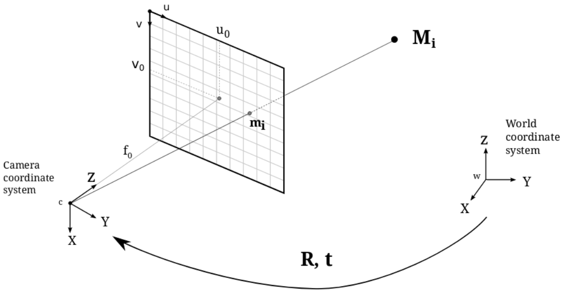
$$s\ m' = K[R|t]M' \tag{1}$$

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

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

Image = 3D world projection on 2D
$\Rightarrow$ projection using the **pinhole camera** model:



(from PyTorch Geometry)

Perspective transformation:

$$s\ m' = K[R|t]M' \qquad (1)$$

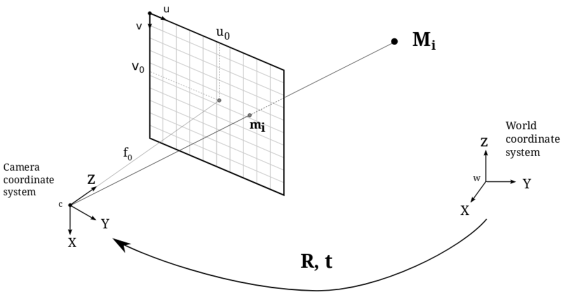
$$s \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & u_0 \\ 0 & f_y & v_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \qquad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

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

2.3. 3D projection on 2D plane

Image = 3D world projection on 2D
$\Rightarrow$ projection using the **pinhole camera** model:



(from PyTorch Geometry)

Perspective transformation:

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

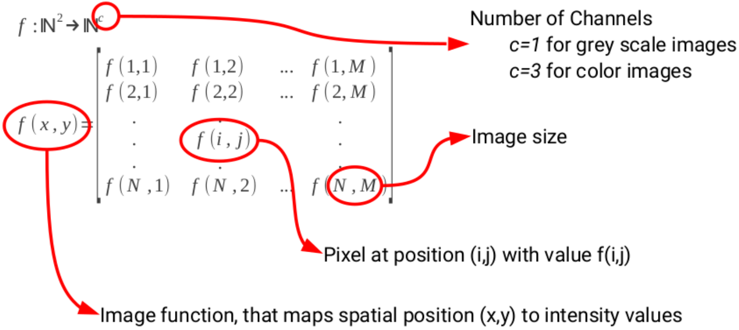
$$s \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & u_0 \\ 0 & f_y & v_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \tag{2}$$

where:

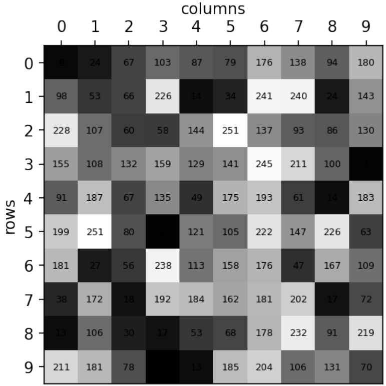
- M' = 3D point in space with coordinates $[X, Y, Z]^T$ expressed in Euclidean coordinates
- m' = projection of the 3D point M' onto the image plane with coordinates $[u, v]^T$ expressed in pixel units
- K = camera calibration matrix (a.k.a instrinsics parameters matrix)
  - fx, fy = focal lengths expressed 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

2.3. 3D projection on 2D plane

$\Rightarrow$ digital image function $f(x, y)$



⇒ digital image function $f(x, y)$

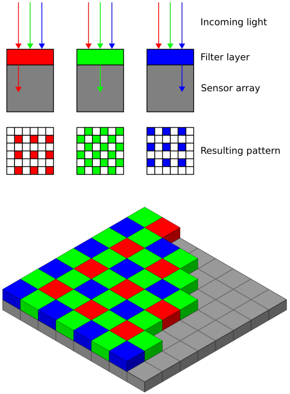


Typical ranges:

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

How do we record colors?

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

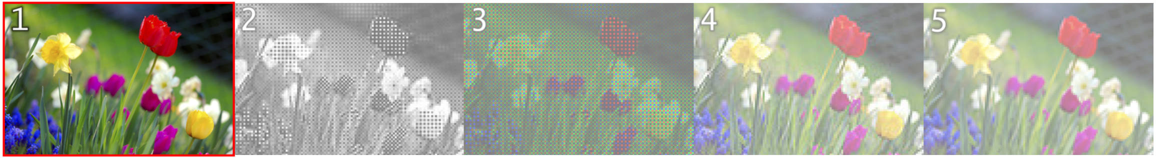


(source wikipedia)

<u>NB</u>: notice the filter pattern is 1/2 green, 1/4 red and 1/4 blue ⇒ more green photosensors are used in order to mimic the physiology of the human eye, which is more sensitive to green light.

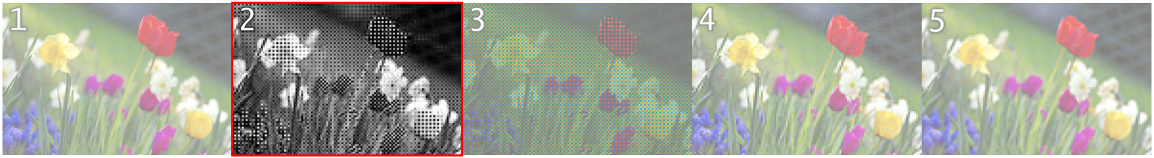
How do we record colors?

⇒ **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

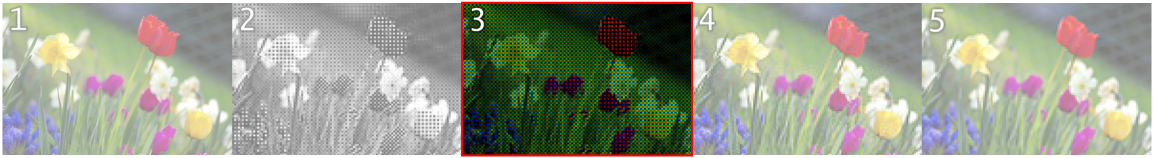
How do we record colors?
⇒ **Bayer Filter**: color filter array for arranging RGB color filters on a square grid of photosensors



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?

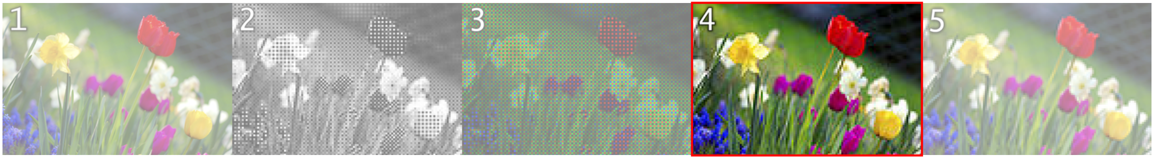
⇒ **Bayer Filter**: color filter array for arranging RGB color filters on a square grid of photosensors



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

How do we record colors?

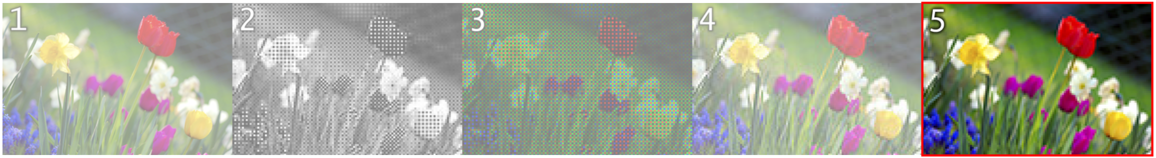
⇒ **Bayer Filter**: color filter array for arranging RGB color filters on a square grid of photosensors



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?

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



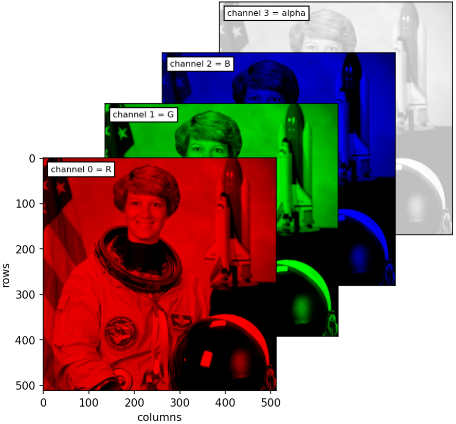
1. Original scene
2. Output of a 120$\times$80-pixel sensor with a Bayer filter
3. Output color-coded with Bayer filter colors
4. Reconstructed image after interpolating missing color information (a.k.a. demosaicing)
5. Full RGB version at 120$\times$80-pixels for comparison

2.4. color image

⇒ color image = 3D tensor in colorspace

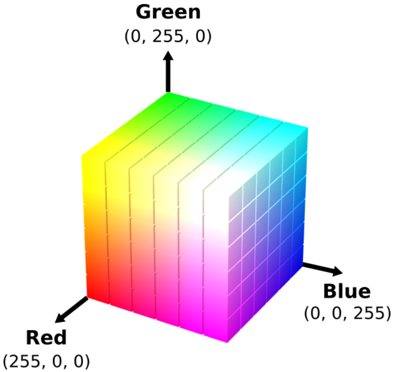
- **RGB** = Red + Green + Blue bands (.JPEG)
- **RGBA** = Red + Green + Blue + Alpha bands (.PNG, .GIF, .BMP, TIFF, .JPEG 2000)

$\longrightarrow$

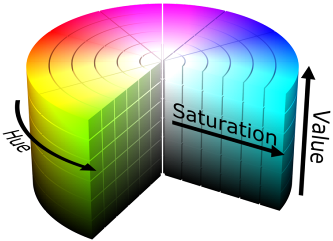


Other ways to represent the color information?

RGB colorspace



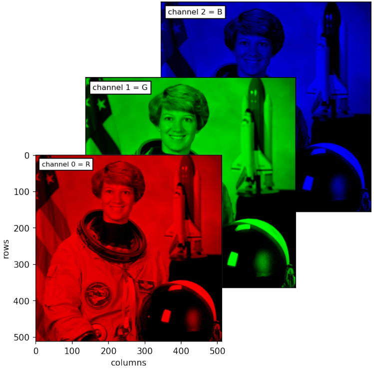
HSV colorspace



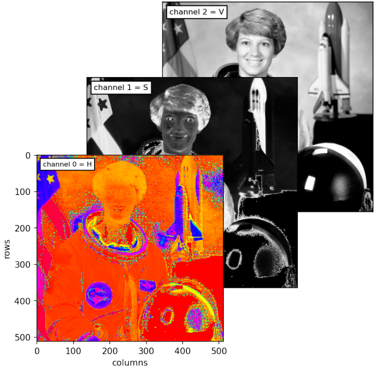
- Hue (H) = [0-360] $\Rightarrow$ shift color
- Saturation (S) = [0-1] $\Rightarrow$ shift intensity
- Value (V) = [0-1] $\Rightarrow$ shift brightness

3D tensor with different information:

RGB colorspace



HSV colorspace



HSV allows for more intuitive color adjustments:

- more saturation S

$\Rightarrow$ more intense colors

original

saturation x2

- more value V

$\Rightarrow$ brighter colors

- shift hue H

$\Rightarrow$ shift color

HSV allows for more intuitive color adjustments:

- more saturation S

$\Rightarrow$ more intense colors

- more value V

$\Rightarrow$ brighter colors

- shift hue H

$\Rightarrow$ shift color

HSV allows for more intuitive color adjustments:

- more saturation S

$\Rightarrow$ more intense colors

- more value V

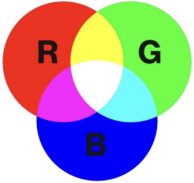
$\Rightarrow$ brighter colors

- shift hue H

$\Rightarrow$ shift color

Additive vs. Subtractive Color Models:

RGB (Additive Model)



⇒ used for devices that emit light (monitors, TVs, smartphones)

⇒ additive model: colors are created by combining different intensities of red, green, and blue light
→ combining all three colors at full intensity results in *white* light, absence of all results in *black*

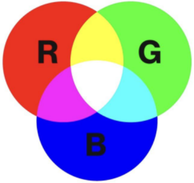
CMYK (Subtractive Model)

⇒ used for printing on paper and other physical media

⇒ subtractive color model: colors are created by subtracting light reflected off the paper
→ combining all three colors ideally absorb all light, resulting in *black* (NB: in printers black ink is added to achieve deeper blacks and reduce usage of the other inks)

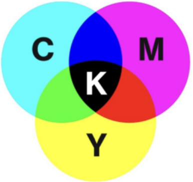
Additive vs. Subtractive Color Models:

RGB (Additive Model)



- ⇒ used for devices that emit light (monitors, TVs, smartphones)
- ⇒ additive model: colors are created by combining different intensities of red, green, and blue light
  → combining all three colors at full intensity results in *white* light, absence of all results in *black*

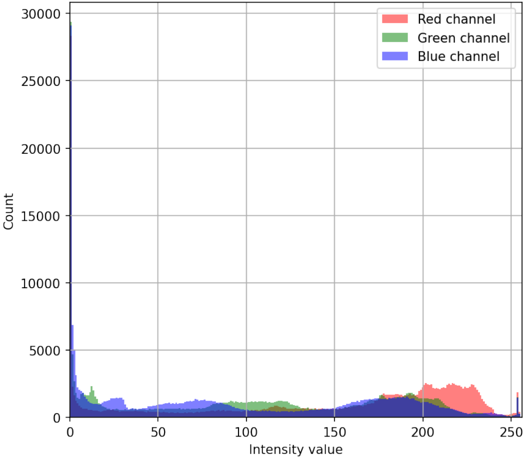
CMYK (Subtractive Model)



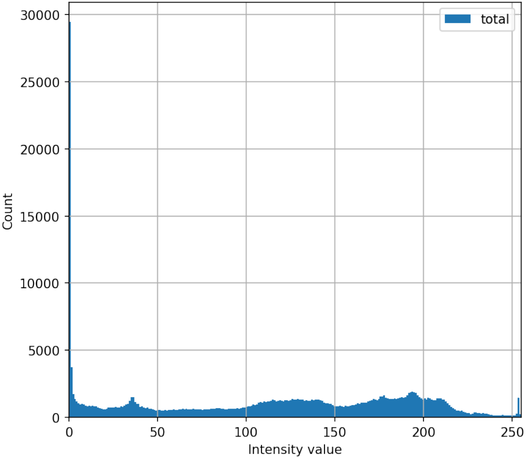
- ⇒ used for printing on paper and other physical media
- ⇒ subtractive color model: colors are created by subtracting light reflected off the paper
  → combining all three colors ideally absorb all light, resulting in *black* (NB: in printers black ink is added to achieve deeper blacks and reduce usage of the other inks)

Histogram of pixel values in each band:

original (uint8)

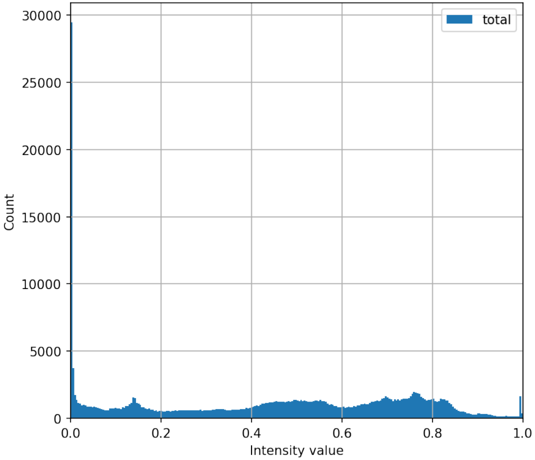


Histogram of pixel values after conversion from RGB (3-bands) to gray-scale (1-band):



NB: weights are chosen to mimic human perception of red, green and blue: the weight on the green band is larger because the human eye has greater sensitivity to green (the retina contains more photoreceptor cells (cones) that are tuned to detect green light)

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



Point operations

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

Value of original image at position (x,y)

Operator function

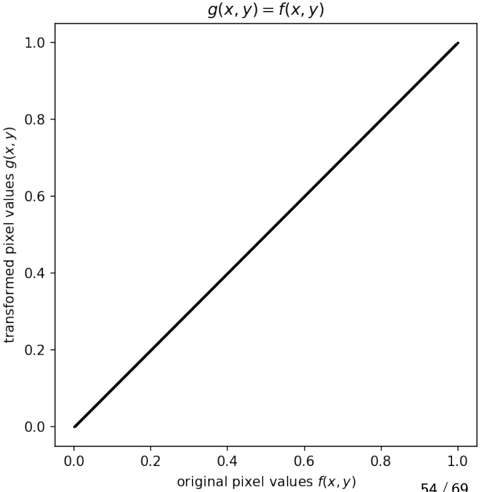
Value of output image

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)

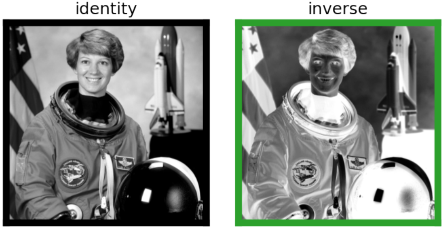
1. image intensity transformation using standard mathematical operations ($\Rightarrow$ adjust pixel color 0=black / 1=white)

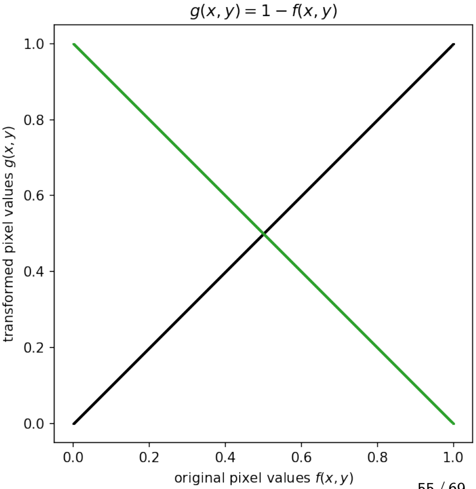
identity



Homogeneous Point Operations (does not depend on pixel position)

1. image intensity transformation using standard mathematical operations (⇒ adjust pixel color 0=black / 1=white)





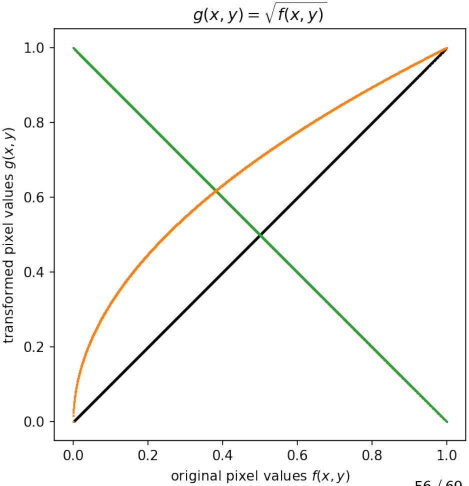
Homogeneous Point Operations (does not depend on pixel position)

1. image intensity transformation using standard mathematical operations (⇒ adjust pixel color 0=black / 1=white)

identity

inverse

square root



Homogeneous Point Operations (does not depend on pixel position)

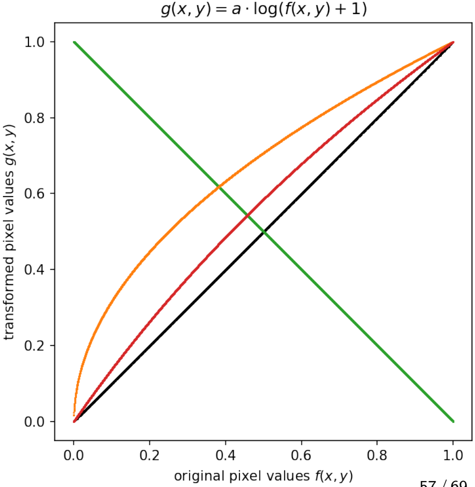
1. image intensity transformation using standard mathematical operations (⇒ adjust pixel color 0=black / 1=white)

identity

inverse

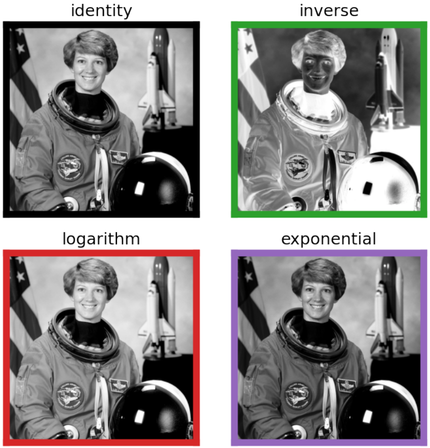
square root

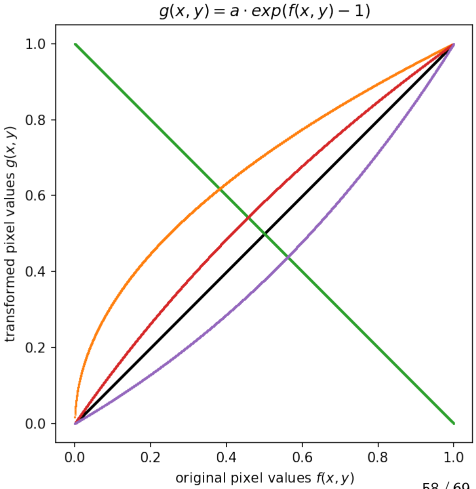
logarithm



Homogeneous Point Operations (does not depend on pixel position)

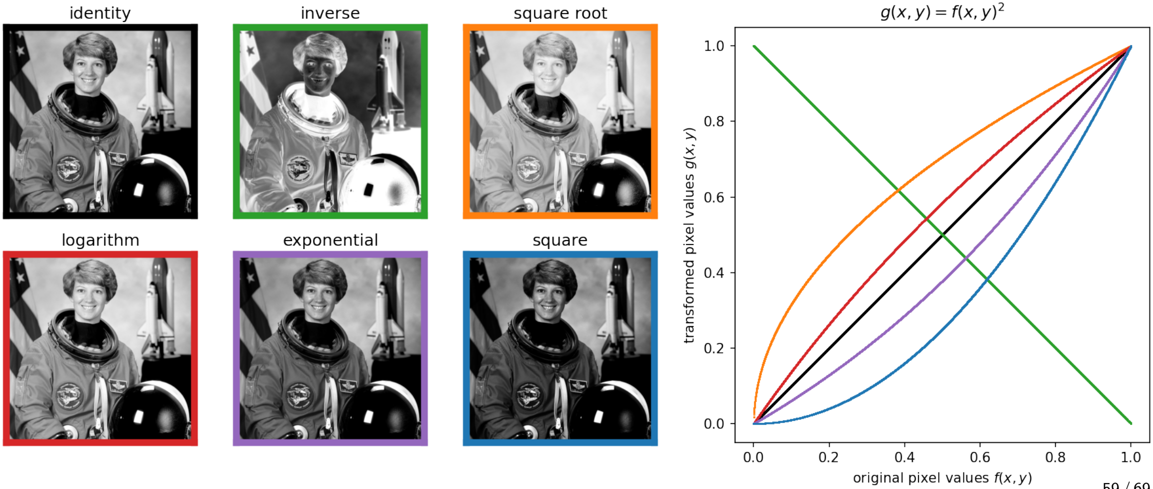
1. **image intensity transformation** using standard mathematical operations (⇒ adjust pixel color 0=black / 1=white)





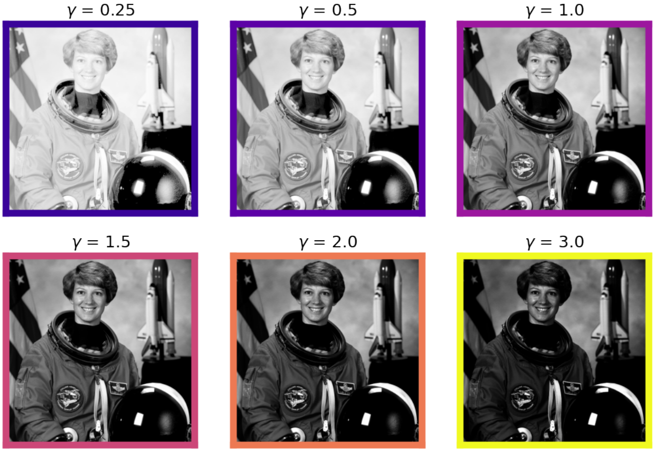
Homogeneous Point Operations (does not depend on pixel position)

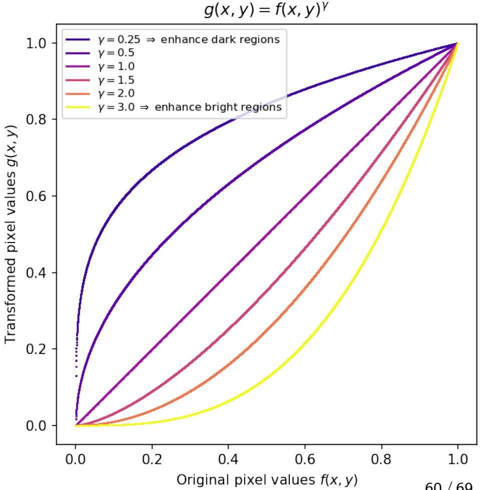
1. **image intensity transformation** using standard mathematical operations (⇒ adjust pixel color 0=black / 1=white)



Homogeneous Point Operations (does not depend on pixel position)

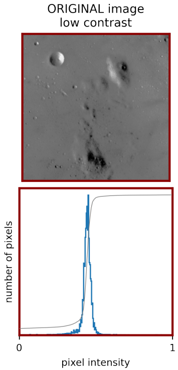
1. **image intensity transformation** using Gamma correction ($\Rightarrow$ power-law transformation)





Homogeneous Point Operations (does not depend on pixel position)

2. **image contrast adjustment** (⇒ adjust image histogram)



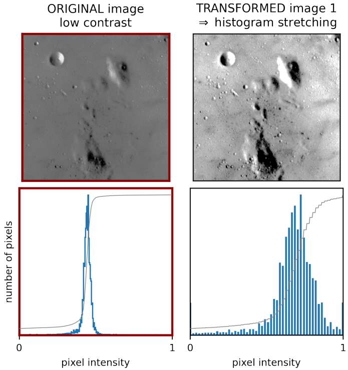
Modified after: skimage-tutorial

Original image (no stretch)

- ⇒ image pixel intensity values are limited to a narrow range
- ⇒ without stretch only a small portion of the full range of possible display levels is used
- ⇒ results in a low contrast image

Homogeneous Point Operations (does not depend on pixel position)

2. image contrast adjustment (⇒ adjust image histogram)



Modified after: skimage-tutorial

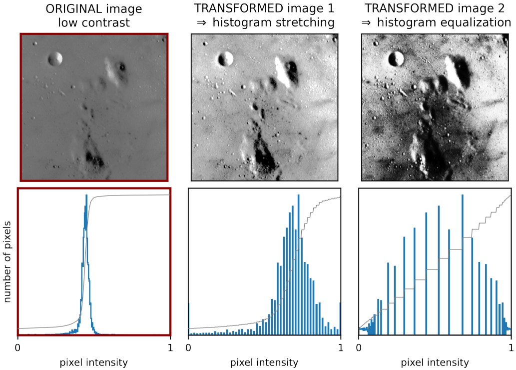
Transformed image #1: linear histogram stretching

⇒ expand range of pixel intensities to stretch across full range of possible values

⇒ rescale pixel values to a specific range
EX: rescale pixel intensities between 2nd and 98th percentiles to occupy full 0-1 range

Homogeneous Point Operations (does not depend on pixel position)

2. image contrast adjustment ($\Rightarrow$ adjust image histogram)



Modified after: skimage-tutorial

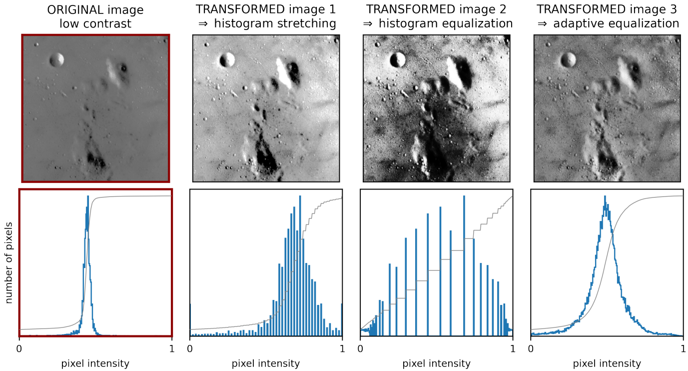
Transformed image #2: histogram equalization

$\Rightarrow$ expand image pixel values on the basis of their frequency of occurrence (i.e. spreads out the most frequent intensity values)

$\Rightarrow$ equalized image has a roughly linear cumulative distribution function

Homogeneous Point Operations (does not depend on pixel position)

2. image contrast adjustment ($\Rightarrow$ adjust image histogram)



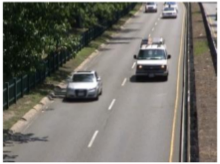
Modified after: skimage-tutorial

Transformed image #3:
adaptive histogram equalization

- $\Rightarrow$ algorithm “Contrast Limited Adaptive Histogram Equalization” (CLAHE)
- $\Rightarrow$ computes histograms over different regions of the image for local contrast enhancement
- $\Rightarrow$ local details can be enhanced even in regions that are darker or lighter than most of the image

Inhomogeneous Point Operations (depends on pixel position)

EX: background detection / change detection



f_1

f_i

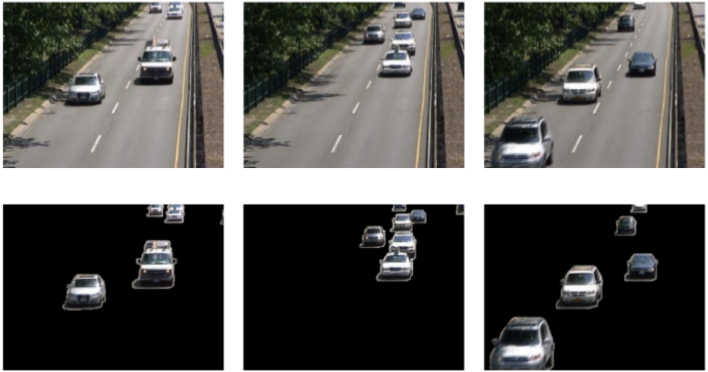
f_N

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

$a(x,y)$

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

Inhomogeneous Point Operations (depends on pixel position)



4.1. numpy tutorial

Numpy tutorial:

$\Rightarrow$ Open CV4GS_02_image-basics/CV4GS_02_numpy-tutorial.ipynb

4.2. exercises

Exercices:

⇒ Open CV4GS_02_image-basics/CV4GS_02_exercices.ipynb