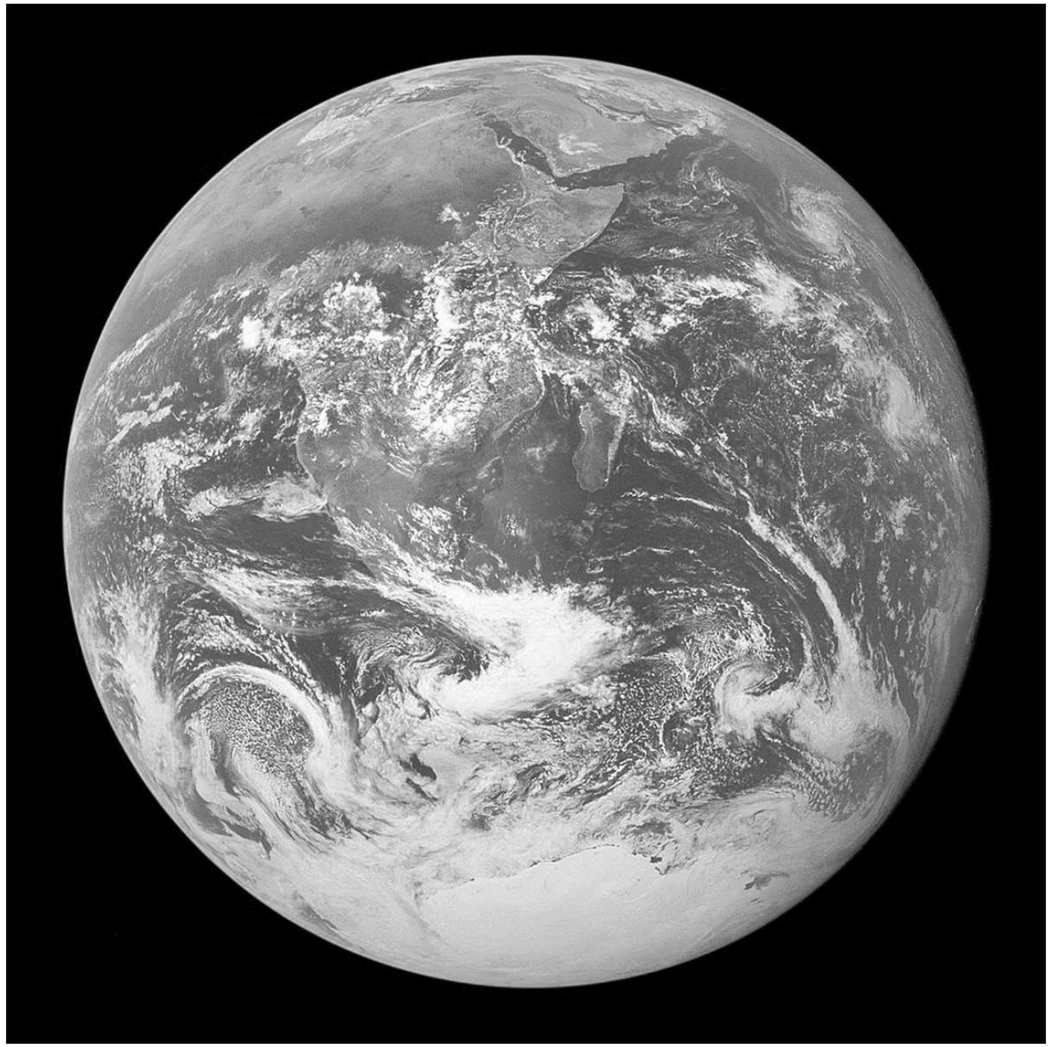
Lecture 02 — Image Formation and Digital Images

Lecture 2

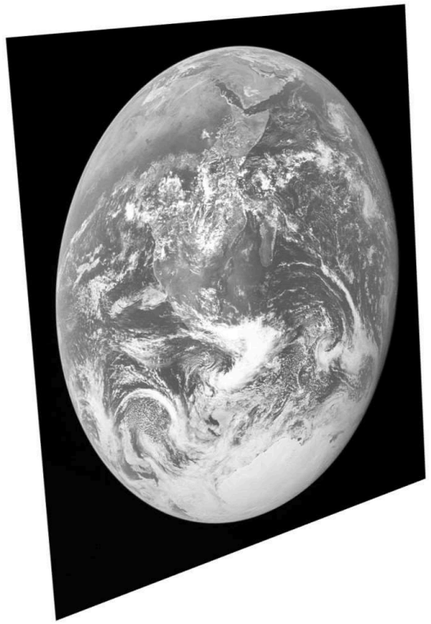
Kaveh Fathian
Computer Science Department, Colorado School of Mines

What is an Image and how is it formed?

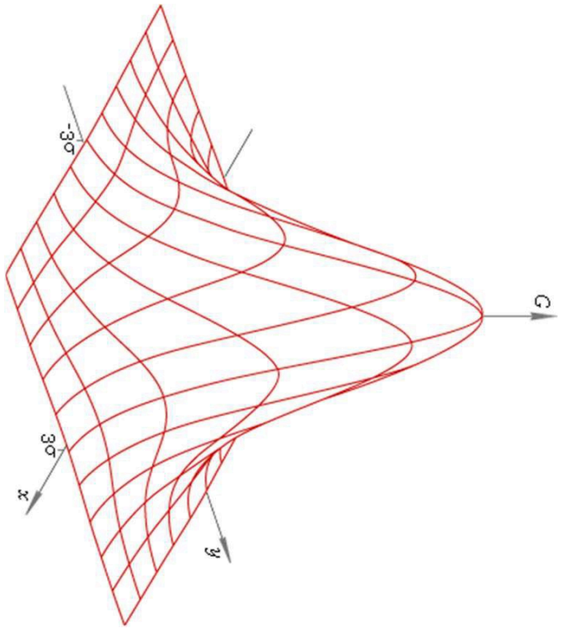


Blue Marble, NASA / Apollo 17

Image Formation



Scene / object

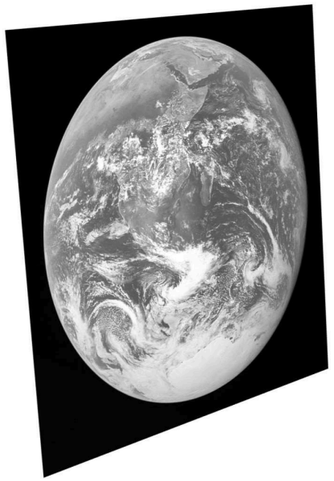


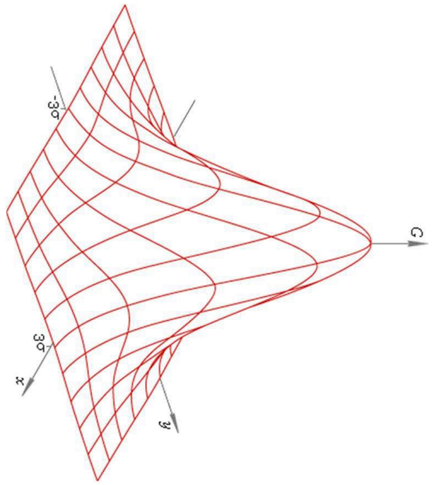
Light as a signal

Camera / sensor

A camera measures light from the world and turns it into a digital image.

Signal

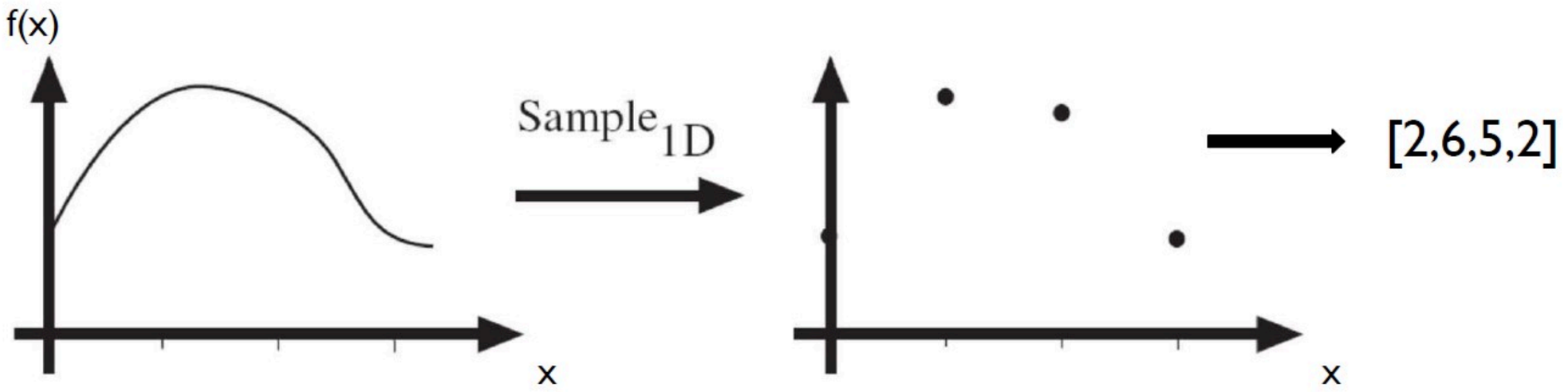




Signal: A (multi-dimensional) function that contains information about a phenomenon (e.g., light, heat, gravity, sound, pressure, motion).

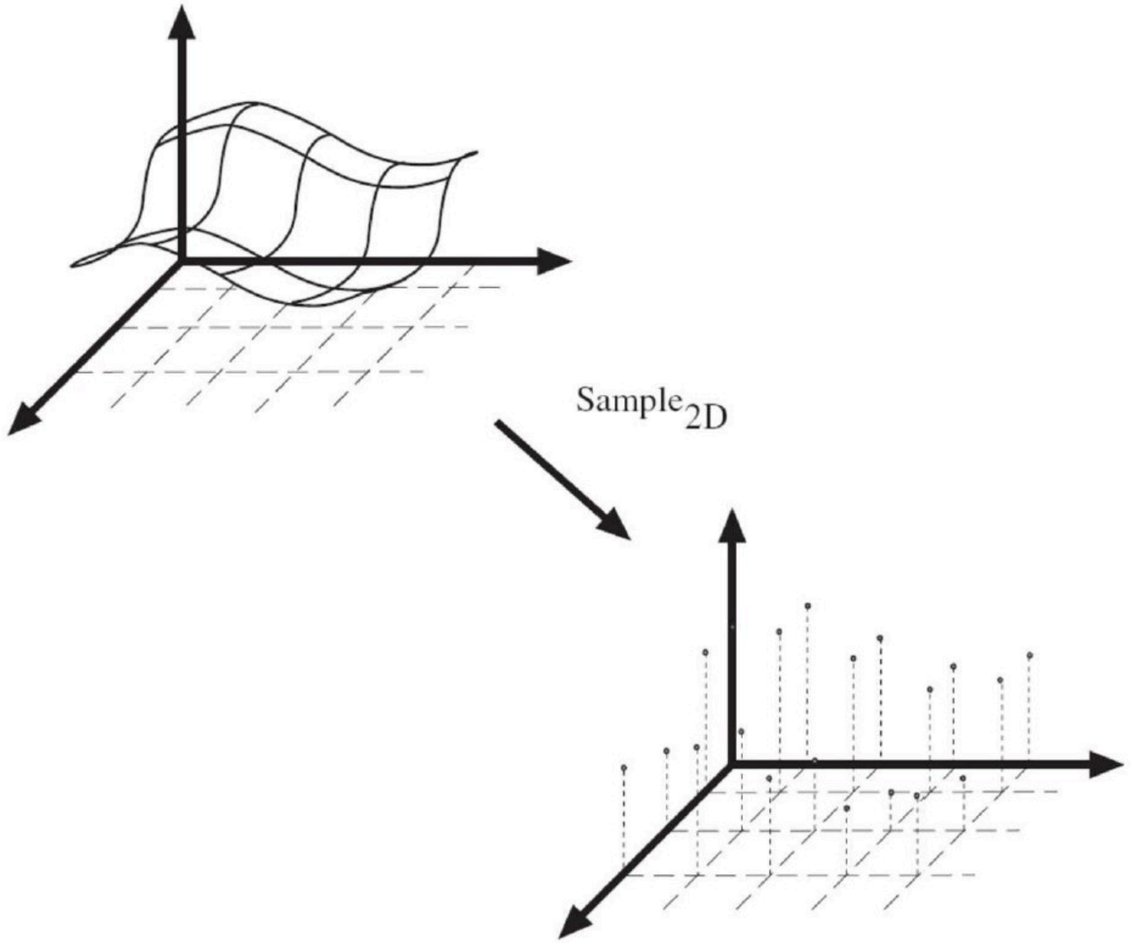
- Signals can be **continuous**, such as light in the physical world.
- Measurements are often **discrete** samples of that continuous signal.
- Sampling reduces a continuous signal to a finite set of values.

Sampling in 1D



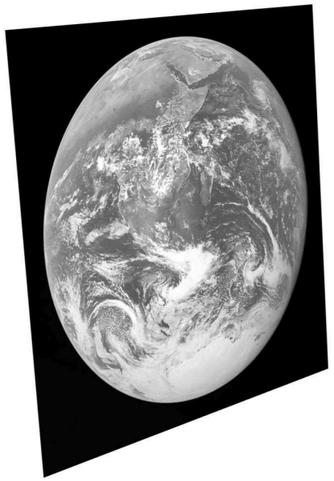
- Sampling a 1D continuous function returns a vector whose elements are values of that function at selected sample points.
- Instead of the full function, we store values at selected locations.

Sampling in 2D



Sampling a 2D function returns a matrix.

2D Image



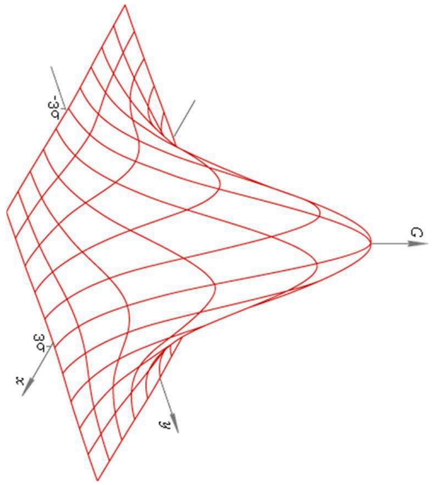


Image: A sampling of a function that contains information about a 2D signal.

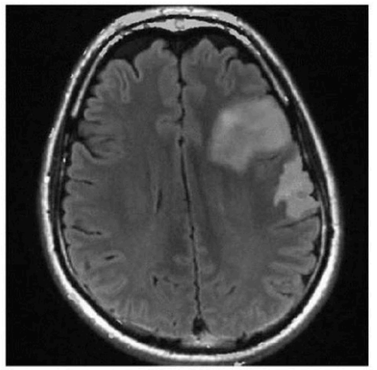
- The function stores **brightness** or **intensity**.
- The image samples brightness along the x and y dimensions.
- A video can be viewed as a time-varying 2D image signal: (x, y, t).

> **Note**
>
> Cameras observe a 2D projection of the 3D world.

Examples of 2D Images

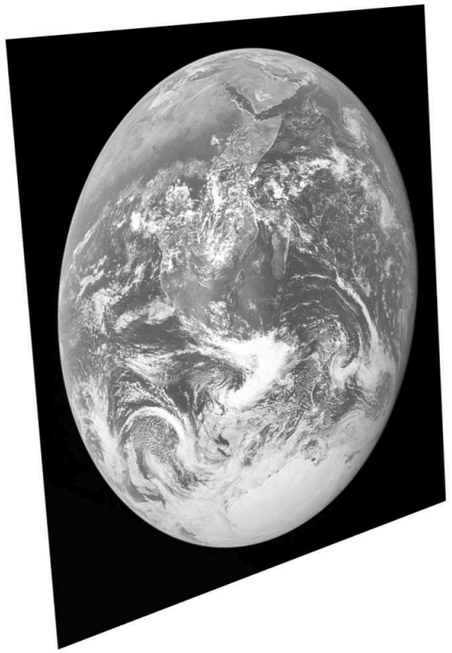
2D images appear in many sensing modalities:

- Natural images
- Thermal images
- Satellite/aerial images
- Microscopy
- Ultrasound
- MRI / CT slices
- Radar images
- Depth images

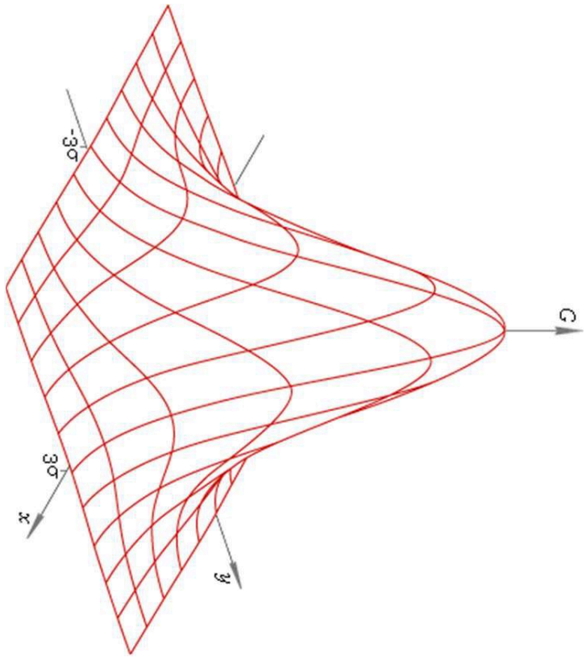


> **Note**
>
> The signal may **not** always be visible light. An image arranges what we have measured over a 2D domain.

Sampling in Practice



World / scene

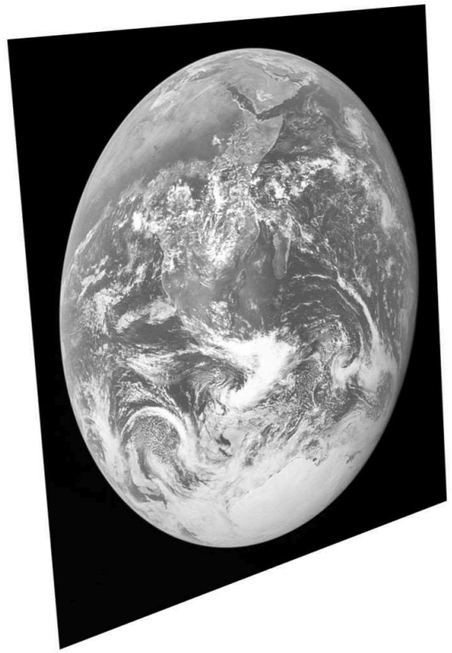


Continuous signal

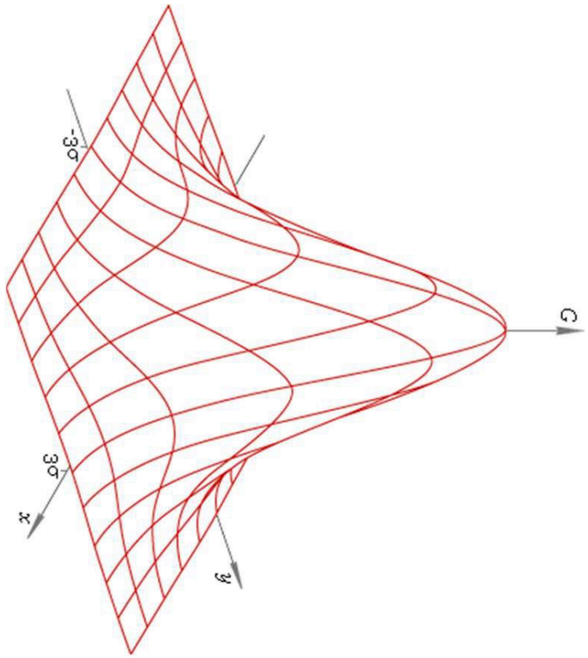
Camera

The imaging pipeline converts a continuous physical signal into discrete sensor measurements.

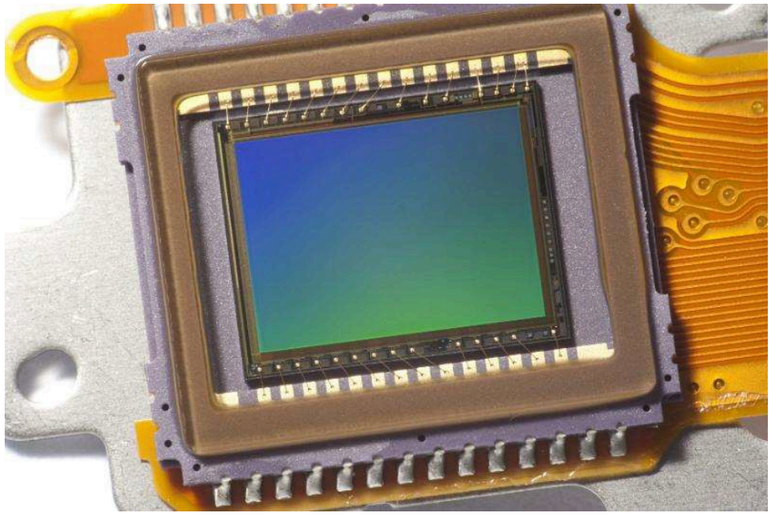
Sampling in Practice: Digital Image



Scene / image plane



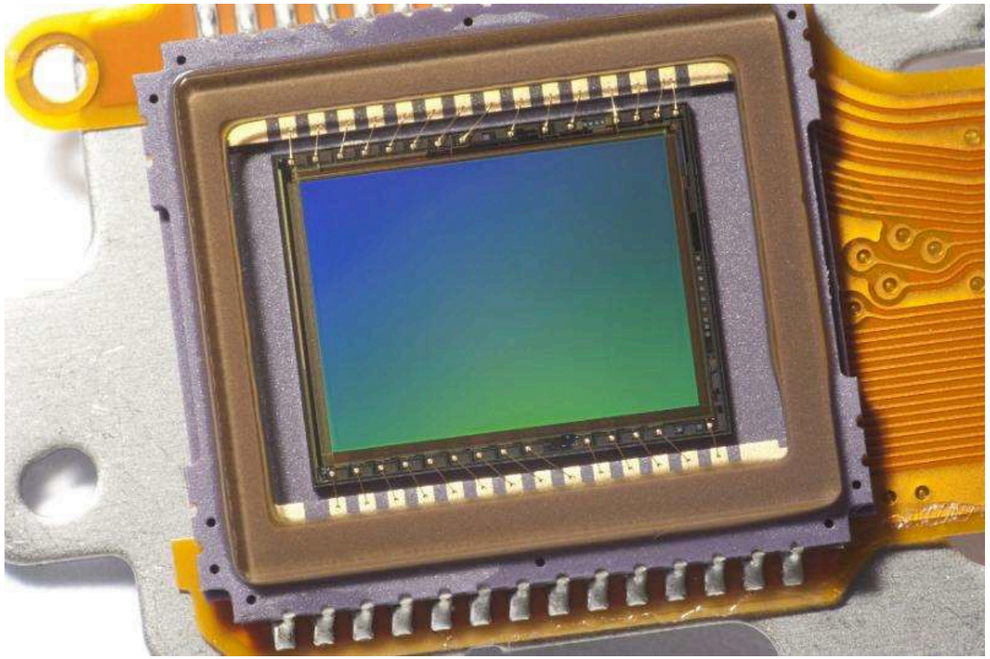
Light signal



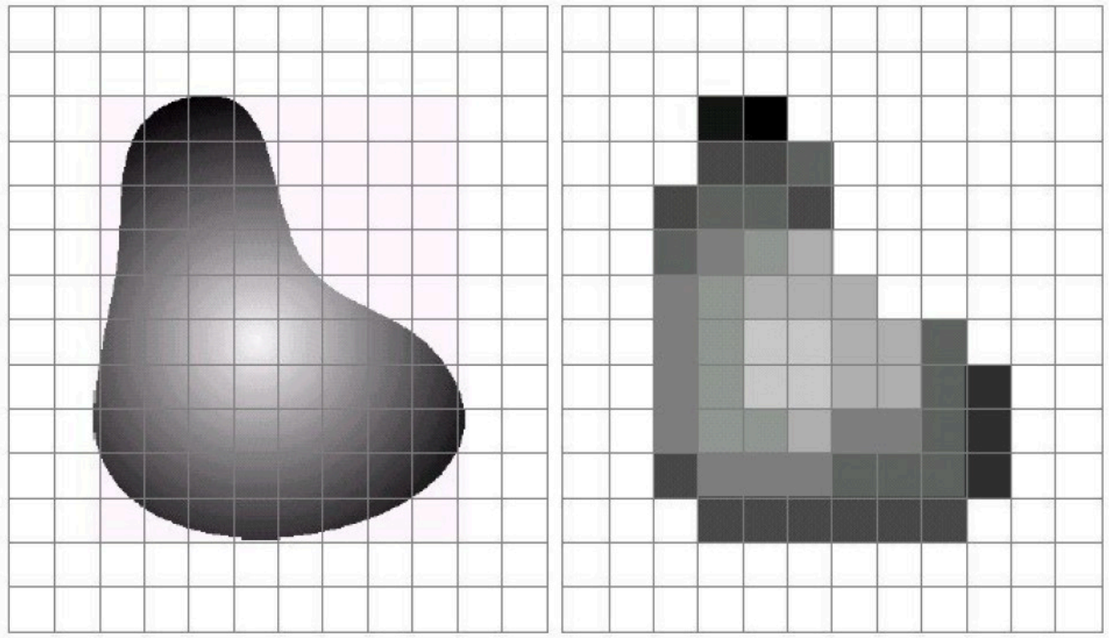
CMOS / CCD sensor

Digital cameras use sensor arrays to sample light and produce a grid of pixel values.

Sensor Array



Camera / sensor hardware



CMOS sensor array

Continuous light is **sampled** and **quantized** by the 2D sensor array (pixels) to produce a digital image.

Elements of a Digital Image

A pixel is a **picture element**.

For a grayscale image:

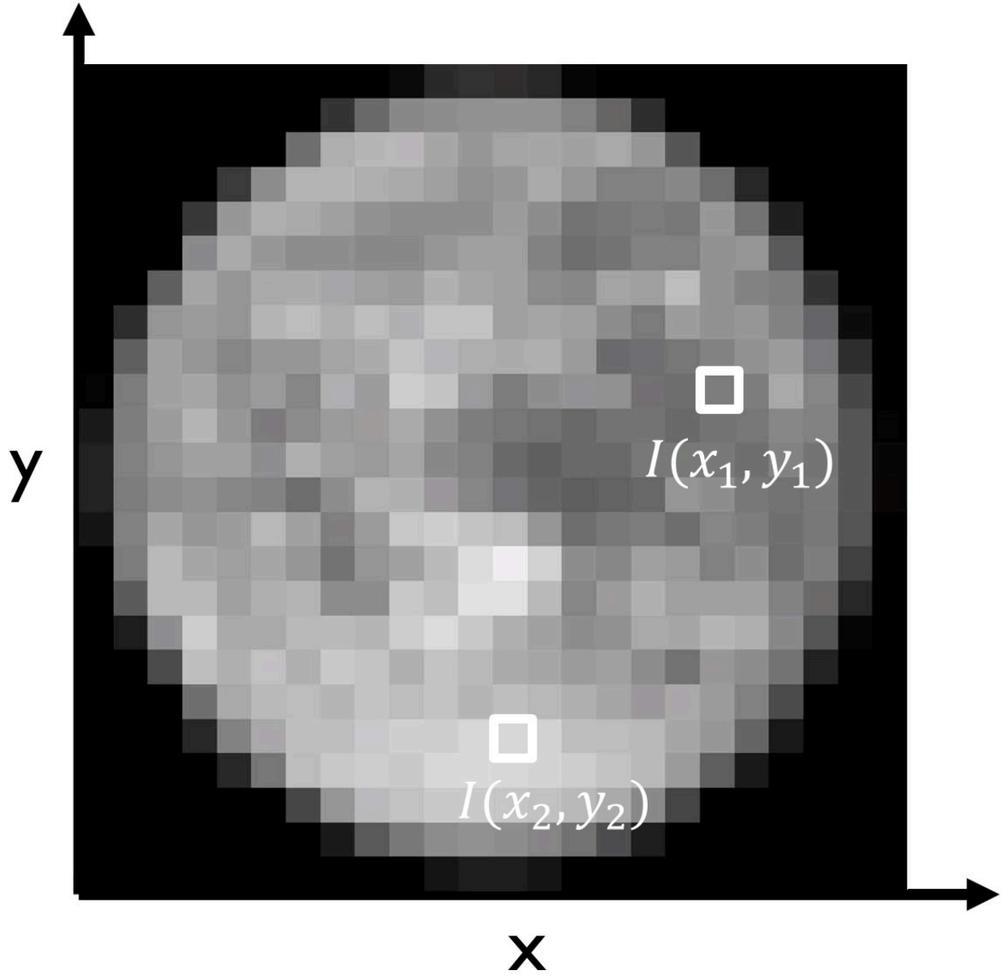
$$I(x, y) = \text{intensity at image location}(x, y)$$

- x: horizontal coordinate
- y: vertical coordinate

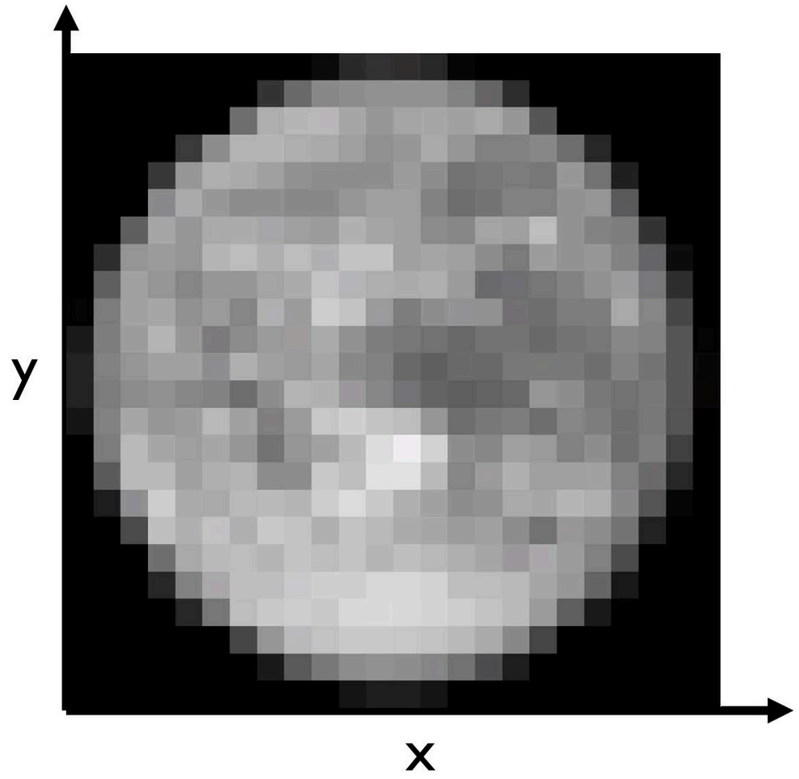
Example:

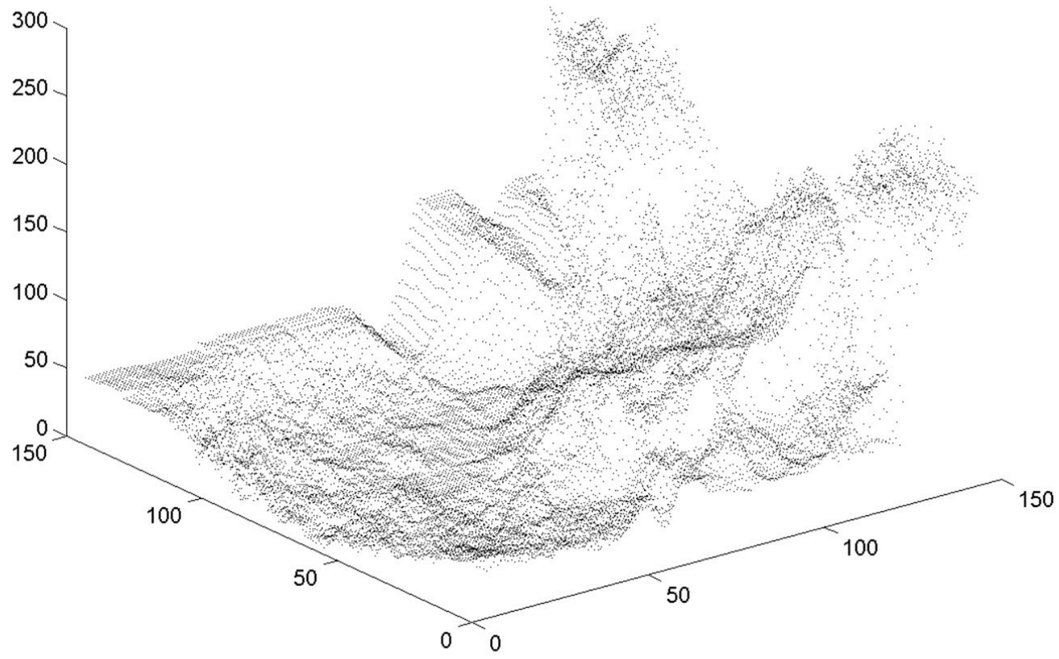
```
I(x1, y1) = 101
I(x2, y2) = 191
```

Different pixel locations can have different intensity values.



Digital Image is a 2D Signal

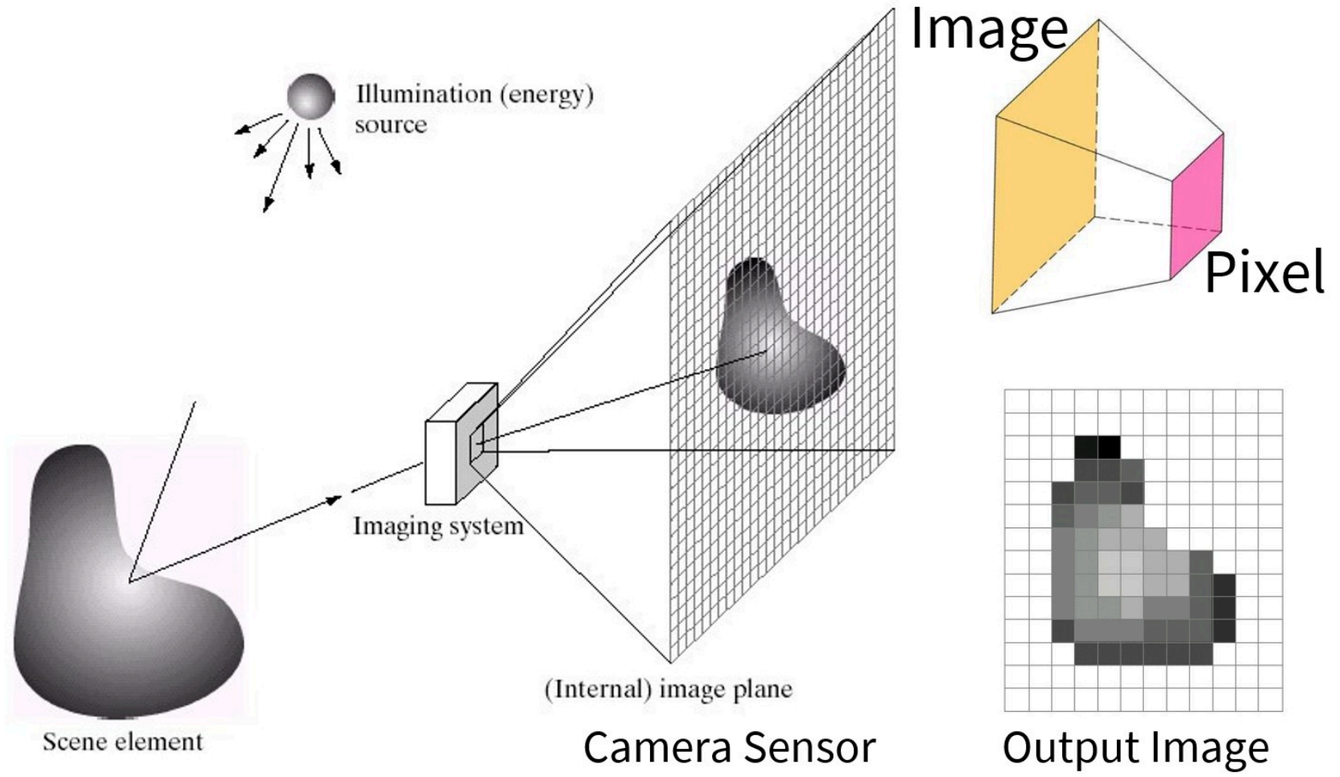




A digital image can be interpreted as a 2D function whose value is **brightness** or **intensity**.

Light Integration Over a Pixel Region

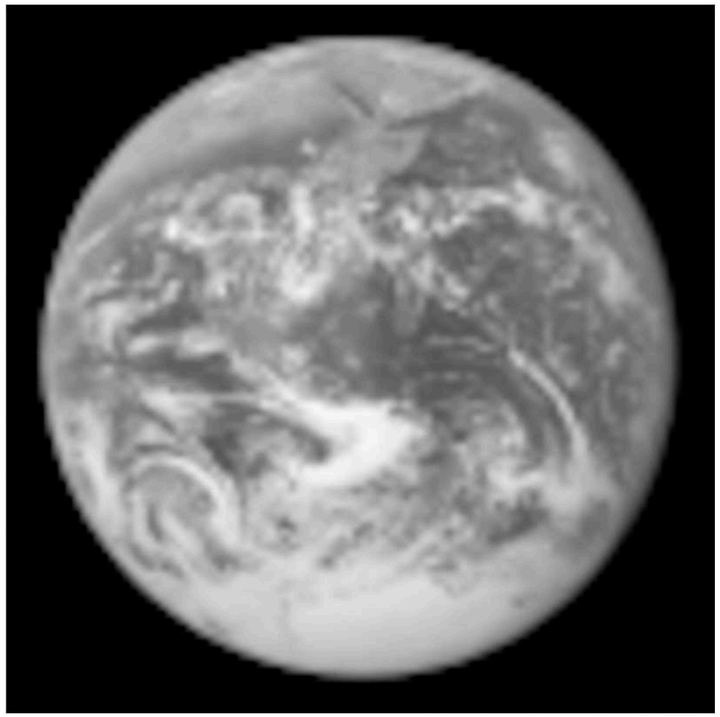
- A single pixel does not measure an infinitely thin ray.
- It measures light integrated over a small region of the sensor and the corresponding cone/frustum of incoming light.



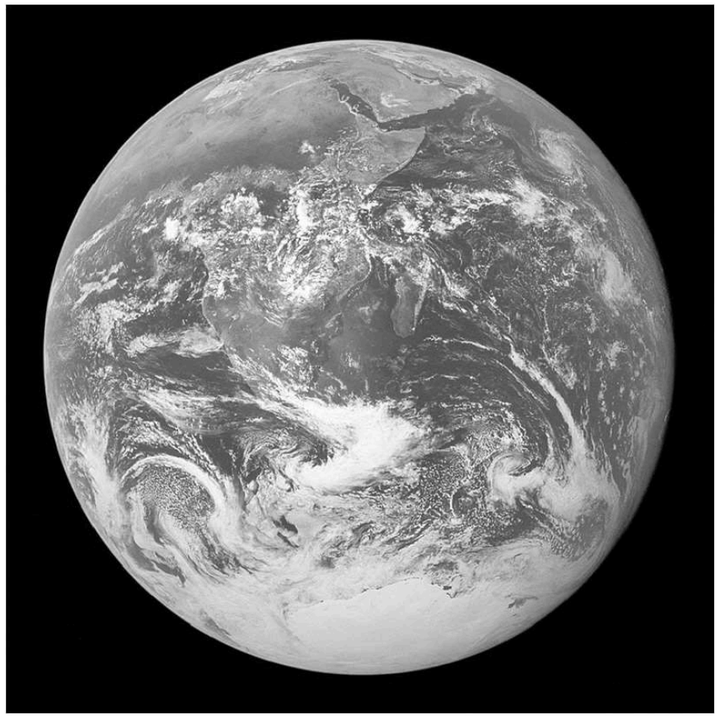
> **Note**
>
> This is one reason why cameras blur, average, and alias details that are smaller than a pixel can represent.

Resolution: Geometric vs. Spatial

Both images can have the same number of pixels, but the amount of **scene detail** represented can be different.



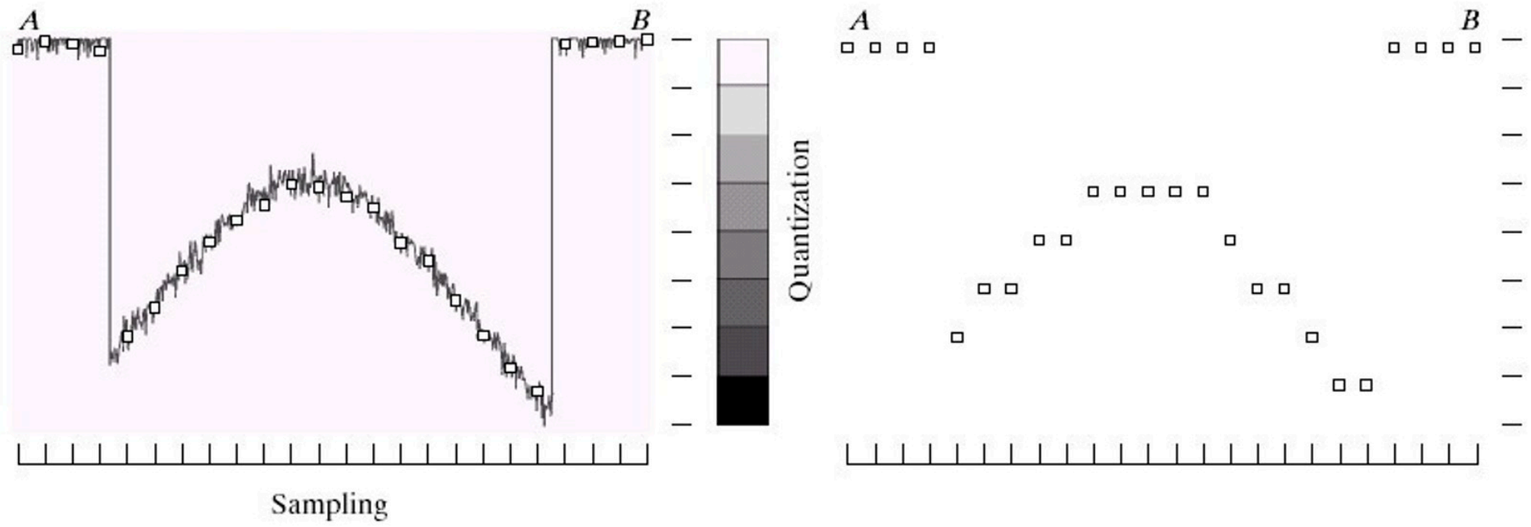
Low geometric resolution



High geometric resolution

> **Note**
>
> Spatial resolution counts pixels. Geometric resolution is how accurately those pixels represent the positions of objects in the real world; or how sharp the image is. Geometric resolution is limited by physics, optics, and the environment (see Rayleigh criterion).

Quantization



- Sampling chooses **where** to measure a signal.
- Quantization chooses **which values** can be represented.
- For example, an 3 bits can represent:

$$2^3 = 8$$

possible intensity levels in a grayscale image.

Quantization Effects — Radiometric Resolution

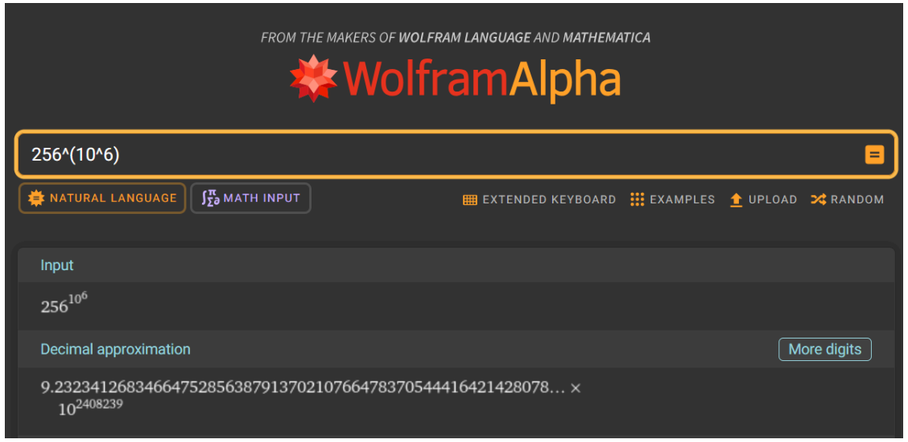
8 bit – 256 levels | 4 bit – 16 levels | 2 bit – 4 levels | 1 bit – 2 levels

- Radiometric resolution is the number of distinct intensity levels available.

Bit depth	Number of levels
1 bit	2 levels
2 bit	4 levels
4 bit	16 levels
8 bit	256 levels

- Higher bit depth can represent more subtle intensity differences.
- In photography, this is closely related to dynamic range.

Dimensionality of an Image



- An image of size 1000×1000 with 8-bit quantization per pixel has:

$$256^{1000 \times 1000} \approx 10^{2,408,239}$$

  possible intensity configurations.

- This is an enormous high-dimensional space – the known universe has only about 10^{80} atoms!
- Computer vision works in this extremely high dimensional space – but real images are not arbitrary arrays and they occupy a small, structured subset of this space.

Images in Python: Grayscale Arrays

For an $N \times M$ grayscale image `im`:

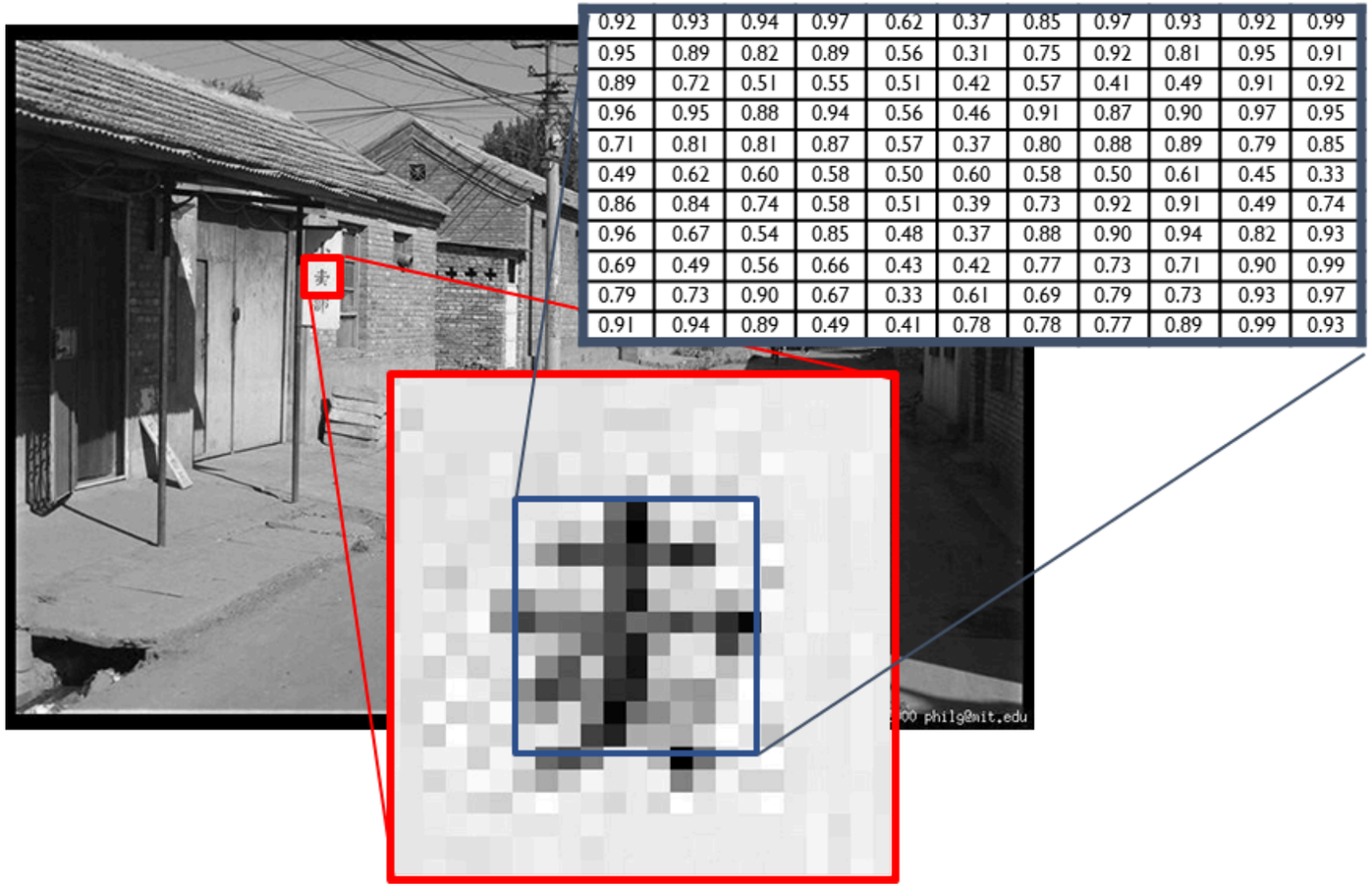
```
im[0, 0]        # top-left pixel value
im[y, x]        # y rows down, x columns right
im[N-1, M-1]    # bottom-right pixel value
```

0.92	0.93	0.94	0.97	0.62	0.37	0.85	0.97	0.93	0.92	0.99
0.95	0.89	0.82	0.89	0.56	0.31	0.75	0.92	0.81	0.95	0.91
0.89	0.72	0.51	0.55	0.51	0.42	0.57	0.41	0.49	0.91	0.92
0.96	0.95	0.88	0.94	0.56	0.46	0.91	0.87	0.90	0.97	0.95
0.71	0.81	0.81	0.87	0.57	0.37	0.80	0.88	0.89	0.79	0.85
0.49	0.62	0.60	0.58	0.50	0.60	0.58	0.50	0.61	0.45	0.33
0.86	0.84	0.74	0.58	0.51	0.39	0.73	0.92	0.91	0.49	0.74
0.96	0.67	0.54	0.85	0.48	0.37	0.88	0.90	0.94	0.82	0.93
0.69	0.49	0.56	0.66	0.43	0.42	0.77	0.73	0.71	0.90	0.99
0.79	0.73	0.90	0.67	0.33	0.61	0.69	0.79	0.73	0.93	0.97
0.91	0.94	0.89	0.49	0.41	0.78	0.78	0.77	0.89	0.99	0.93

Important

In NumPy, image indexing is usually **row first, column second**: `im[y, x]`, not `im[x, y]`.

Grayscale Intensity



A grayscale image is a **matrix** that stores one intensity value per pixel.

Color Images

- A color image stores multiple intensity values per pixel.
- For RGB images, each pixel has three channels:
  - Red intensity
  - Green intensity
  - Blue intensity

```
image shape: height × width × 3
```

> **Note**
>
> Different libraries may use different channel order conventions. OpenCV commonly uses BGR; Matplotlib commonly expects RGB.

Images in Python: Color Arrays

For an $N \times M$ color image `im` with three channels:

```
im[0, 0, 0]           # top-left pixel, first channel
im[y, x, 1]           # pixel at row y, column x, second channel
im[N-1, M-1, 2]       # bottom-right pixel, third channel
```

Typical shape:

```
im.shape  # (height, width, channels)
```

For an RGB image:

```
channel 0 = red
channel 1 = green
channel 2 = blue
```

For OpenCV-loaded images:

```
channel 0 = blue
channel 1 = green
channel 2 = red
```

Images in Python: Data Types

Take care of image data types.

```python
# uint8 image: values 0 to 255
im = cv2.imread("file.jpg")

# convert to float32, still values 0 to 255
im_float = im.astype(np.float32)

# normalize to 0 to 1
im_norm = im.astype(np.float32) / 255.0
```

Important

Many image-processing bugs come from mixing `uint8`, `float32`, and different value ranges.

Random Images in Python

```
1 from numpy import random as r
2
3 I = r.rand(256, 256)
```

`r.rand(256, 256)` creates a 256 by 256 array of random floating-point values in the range $[0, 1)$.

Questions:

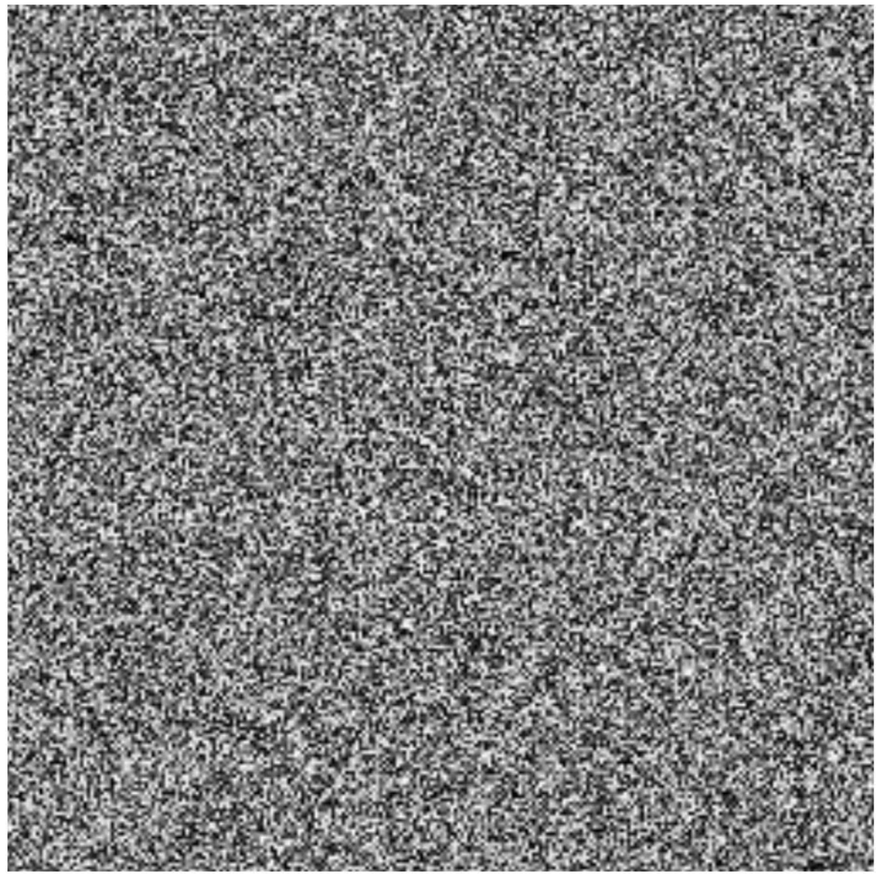
- What is `I`?
- What does it look like?
- Which values does it contain?
- How many values are there?

Displaying an Image in Python

```python
from matplotlib import pyplot as plt
from numpy import random as r

I = r.rand(256, 256)

plt.imshow(I, cmap="gray")
plt.axis("off")
plt.show()
```



Is this an image?

Coding in This Course

We will use:

- **Python** for programming
- **Conda** for managing the course environment
- **NumPy** for numerical arrays
- **OpenCV** for image processing and computer vision
- **PyTorch** for deep learning topics
- **VS Code** for editing, debugging, and notebooks

> **Important**
>
> Do not use GenAI to solve homework for you. If you copy-paste code without understanding it, you will not learn how to code.

Be kind and supportive: students come from different backgrounds and have different levels of Python experience.

Python / Tool Refresher

Course tutorials are available here:

https://ariarobotics.github.io/cv/tutorials/

Review as needed:

- **WSL + Ubuntu** — recommended Linux environment for Windows users
- **Git & GitHub** — cloning, committing, pulling, pushing, and GitHub Classroom
- **Python & Conda** — installing Conda and creating the course environment
- **Visual Studio Code** — Python interpreter, notebooks, and debugging
- **NumPy** — arrays, indexing, vectorization, matrices, and image-related operations

> **Tip**
>
> These tutorials are reference material. Use them when you need help with the tools so class time can focus on computer vision.

Recommended Development Setup

Windows

Recommended workflow:

```
Windows
   ↓
WSL 2 + Ubuntu
   ↓
Miniforge / Conda
   ↓
cv environment
   ↓
VS Code + WSL extension
```

Keep course Python environments and projects **inside WSL**.

macOS / Linux

Recommended workflow:

```
macOS / Linux
      ↓
Terminal
      ↓
Miniforge / Conda
      ↓
cv environment
      ↓
VS Code
```

No WSL is needed.

Before We Create the Environment

Open a terminal and check the tools you already have.

Git

```
1 git --version
```

Conda

```
1 conda --version
```

Existing Conda environments

```
1 conda env list
```

Windows users working in WSL can also check:

```
1 uname -a
```

> **Tip**
>
> If `conda` is not installed yet, open the **Python & Conda** tutorial and follow the Miniforge installation section.

The Course Environment

We use a dedicated Conda environment named:

```
cv
```

The environment file is maintained in the course repository:

https://github.com/ariarobotics/cv/blob/main/code/cv-environment.yml

It defines the software stack used by the course, including:

- Python
- NumPy / SciPy
- OpenCV
- PyTorch / TorchVision
- Matplotlib
- scikit-image / scikit-learn
- Jupyter and VS Code notebook support

Do not manually assemble your own version of the environment.

Get the Environment File

Option 1 — Clone the course repository

```
cd ~/cv_projects
git clone https://github.com/ariarobotics/cv.git
cd cv/code
```

You should now see:

```
ls
```

including:

```
cv-environment.yml
```

Option 2 — Download only the environment file

```
curl -L -o cv-environment.yml \
https://raw.githubusercontent.com/ariarobotics/cv/main/code/cv-environment.yml
```

Create the `cv` Environment

From the directory containing `cv-environment.yml`:

```
conda env create -f cv-environment.yml
```

This may take several minutes.

The environment file already specifies the name:

```
cv
```

When installation finishes:

```
conda activate cv
```

Your terminal prompt should now begin with something similar to:

```
(cv)
```

> **Important**
>
> Whenever you run course Python code, make sure the `cv` environment is active.

Verify the Environment

With `cv` activated:

```
python --version
```

Check which Python is being used:

```
which python
```

Then verify the main packages:

```
python -c "import numpy; print('NumPy:', numpy.__version__)"
python -c "import cv2; print('OpenCV:', cv2.__version__)"
python -c "import torch; print('PyTorch:', torch.__version__)"
python -c "import torchvision; print('TorchVision:', torchvision.__version__)"
```

If all four commands work, the core environment is ready.

Configure VS Code

Open the project folder from your terminal:

```
code .
```

Windows + WSL users should see **WSL: Ubuntu** in the VS Code status area.

Then:

1. Open the Command Palette: `Ctrl+Shift+P`
2. Search for **Python: Select Interpreter**
3. Select the interpreter associated with **`cv`**

For notebooks, use **Select Kernel** and choose the same `cv` environment.

Verify from Python:

```
import sys
print(sys.executable)
```

The path should point into the `cv` Conda environment.

Our First Computer Vision Program

Create a file named:

```
first_cv.py
```

Start by importing the libraries:

```
from pathlib import Path

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

Create a folder for output files:

```
output_dir = Path("outputs")
output_dir.mkdir(exist_ok=True)
```

Create and Save a Simple Image

For today's example, we will create a small synthetic image so everyone has the same input.

```
height = 360
width = 640

image = np.full(
    (height, width, 3),
    235,
    dtype=np.uint8,
)

# OpenCV uses BGR color ordering.
cv2.rectangle(
    image, (70, 80), (260, 280),
    (255, 100, 0), thickness=-1
)

cv2.circle(
    image, (470, 180), 90,
```

We now have a real image file on disk that we can load with OpenCV.

Load and Inspect the Image

Load the image:

```
image_bgr = cv2.imread(
    str(output_dir / "sample_image.png")
)
```

Inspect it:

```
print("Shape:", image_bgr.shape)
print("Data type:", image_bgr.dtype)
```

You should see something similar to:

```
Shape: (360, 640, 3)
Data type: uint8
```

An image is fundamentally a **NumPy array**.

Display the Image

OpenCV loads color images in **BGR** order.

Matplotlib expects **RGB**, so convert before displaying:

```
image_rgb = cv2.cvtColor(
    image_bgr,
    cv2.COLOR_BGR2RGB
)

plt.imshow(image_rgb)
plt.axis("off")
plt.show()
```

A useful rule:

```
OpenCV processing → usually BGR
Matplotlib display → RGB
```

> **Note**
>
> Using Matplotlib is also convenient in notebooks and avoids problems that can occur with `cv2.imshow()` in some WSL setups.

A First Image-Processing Operation

Convert the image to grayscale:

```
gray = cv2.cvtColor(
    image_bgr,
    cv2.COLOR_BGR2GRAY
)
```

Detect edges:

```
edges = cv2.Canny(
    gray,
    100,
    200
)
```

Display the result:

```
plt.imshow(edges, cmap="gray")
plt.axis("off")
plt.show()
```

What Just Happened?

In only a few lines, we used several ideas that will appear throughout the course:

- An image is represented as a **NumPy array**
- Image dimensions are described by `.shape`
- Pixel values have a data type such as `uint8`
- OpenCV can **load, save, transform, and analyze images**
- Color-channel conventions matter: **BGR vs. RGB**
- Computer vision pipelines are often sequences of simple operations

```
image
  ↓
load
  ↓
represent as an array
  ↓
transform / analyze
  ↓
display or use the result
```

Common Setup Problems

`conda: command not found`

Restart the terminal or reload your shell:

```
source ~/.bashrc
```

Python imports fail

First check:

```
conda activate cv
which python
```

VS Code runs a different Python

Use:

```
Python: Select Interpreter
```

and select **`cv`**.

Windows paths and Linux paths are getting mixed

If you are using WSL, keep the course environment and projects inside the WSL/Linux filesystem.

After Class

Please go through any tutorials that cover tools you are not yet comfortable with:

https://ariarobotics.github.io/cv/tutorials/

In particular, make sure you can:

- Activate the `cv` environment
- Use basic Git commands
- Open a project in VS Code
- Run Python scripts and Jupyter notebooks
- Work with basic NumPy arrays

You do **not** need to memorize every Git, Conda, or NumPy command. The tutorials are there as references throughout the semester.