ELEC 421

Digital Signal and Image Processing

Siamak Najarian, Ph.D., P.Eng.,

Professor of Biomedical Engineering (retired),

Electrical and Computer Engineering Department,

University of British Columbia

Course Roadmap for DIP

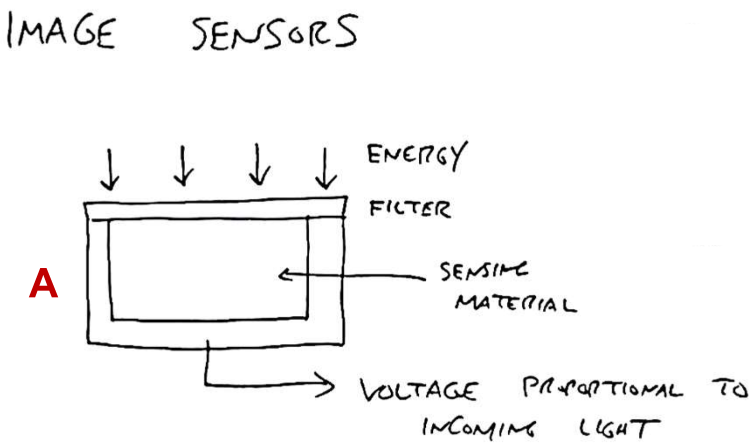
Lecture	Title
Lecture 1	Digital Image Modalities and Processing
Lecture 2	The Human Visual System, Perception, and Color
Lecture 3	Image Acquisition and Sensing
Lecture 4	Histograms and Point Operations
Lecture 5	Geometric Operations
Lecture 6	Spatial Filters

Lecture 3:
Image Acquisition and Sensing

Table of Contents

- Image sensors
- Perspective projection
- CCD array sizes and pixels
- The Bayer array; color sensing
- Illumination model
- Sampling and quantization
- MATLAB demo
- Image coordinate systems
- Useful MATLAB commands
- Pixel neighbors and distances

Image sensors



- **Image Sensors:** Last time, we talked about the human visual system and how people actually perceive images of the natural world. We discussed how the retina works. Here, we want to know how digital images work in computers. What we want to talk about in this lecture is mainly an introduction to how a digital camera/scanner works. Then, we will discuss the fundamental camera/scanner parameters that we should be aware of when taking a digital image.

- **Diagram A:** Typically, the first thing we need to do is to have a **CCD** or a **light aware sensor**. As shown in A, the idea is that we have a little window through which light energy is collected. Here, energy comes down and hits a **filter** that selects a certain color. We talked about how our eyes are sensitive to roughly red, green, and blue light, although we saw from eye responses that is not actually 100% correct. But, fundamentally, in that range, the filter acts to pass certain wavelengths of light through. Next in A, we have a **cavity** with a **sensing material** and then what comes out of these little sensor elements is basically a **voltage** that is proportional to the incoming light. A common way of doing this is to think about it as a **photodiode**. That is, we put a filter on top of photodiode, we stack these together into an array, and that is what gives us our **pixels**.

Image sensors

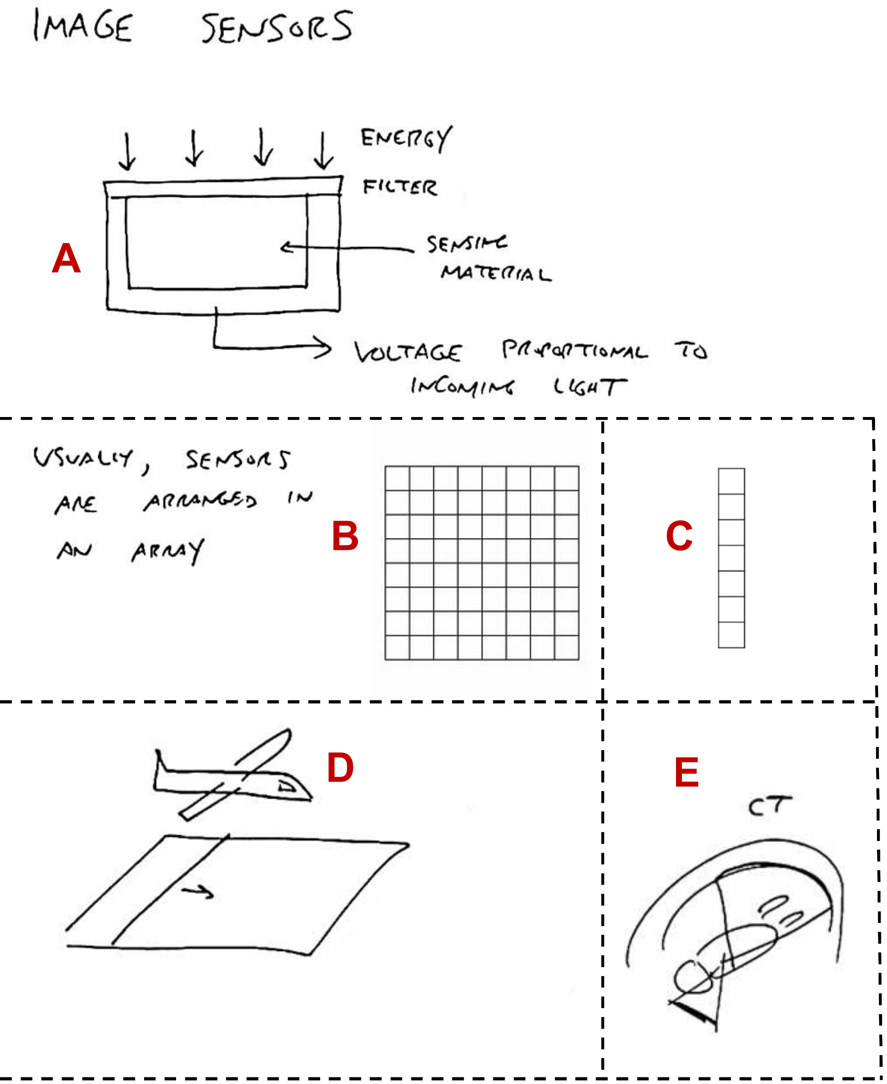
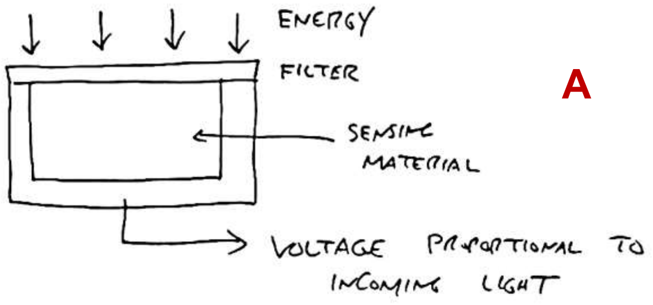
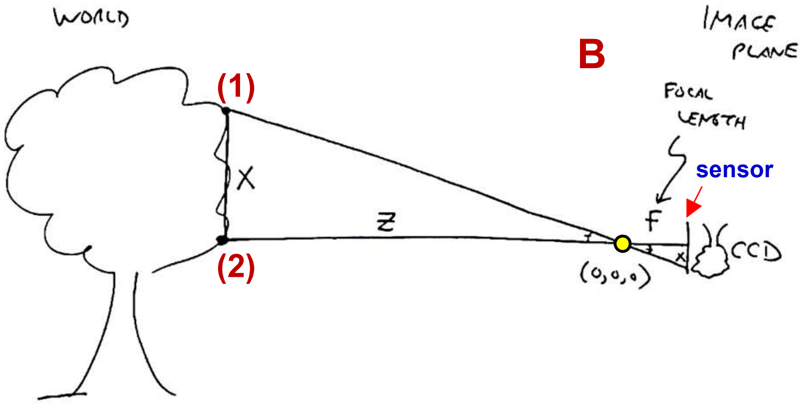


IMAGE SENSORS
ENERGY
FILTER
A
SENSING MATERIAL
VOLTAGE PROPORTIONAL TO INCOMING LIGHT
USUALLY, SENSORS ARE ARRANGED IN AN ARRAY
B
C
D
E
CT

- **Diagrams B and C:** We usually have many sensors in a camera or a scanner, and usually, these sensors are arranged in an **array**, B. In almost every instance, we have a **rectangular array** of these sensors, which form pixels. But, it is also common to have just a **single strip of sensors**, C. These kinds of systems are very common in scanning machines like **flatbed scanners**. Here, we put a page on the glass plate and then there is a scanner head that runs across the width of the page. There is really only one line of sensors that goes across the page. Other examples are a photocopier or a fax machine.
- **Diagram D:** Another setup for a single strip of sensors is in **airborne scanning**. Here, we have an aircraft that is surveying some terrain. It may be doing so with a **moving stripe** that is picking up responses as the stripe moves forward.
- **Diagram E:** In a CT-scanning scenario, we have a patient who is surrounded by the CT ring. What is happening here is that we are getting a strip of responses from rays that pass through the patient. In that case, we are not acquiring the whole image at once. Instead, we are acquiring one stripe and as the patient has moved laterally through the ring, we acquire information from the whole body.
- In this course, we are going to assume that most of the time our pixels are arranged in a grid.

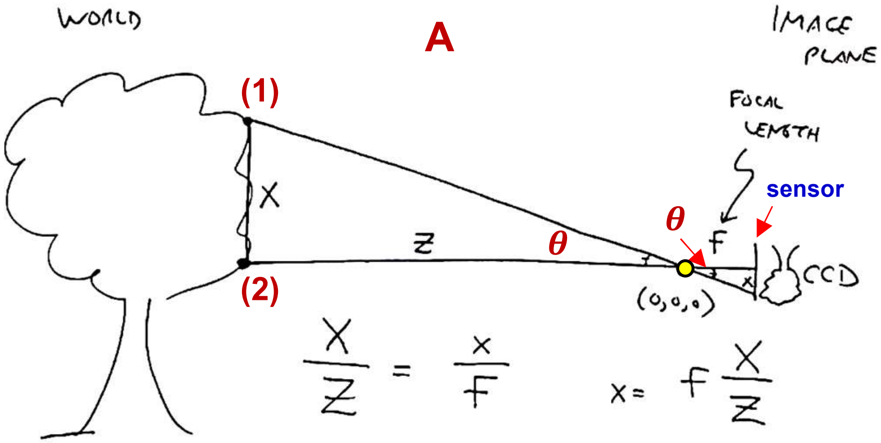
Perspective projection





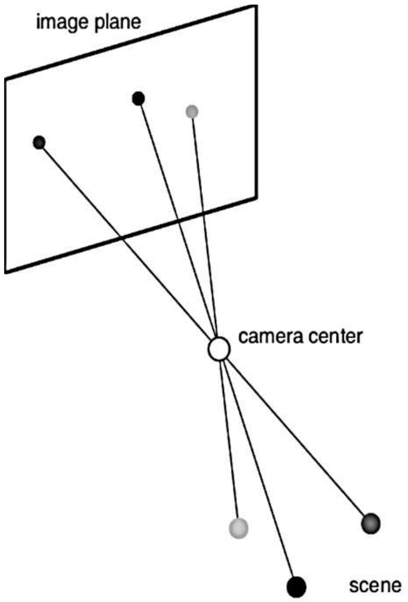
- **Perspective projection** refers to the mapping of three-dimensional (3D) points in the world onto a two-dimensional (2D) image plane.
- **How do we go from Diagram A setup to the actual image?** Let us start by looking at the setup in B. Here, we have some object, a tree, in the **outside world** (or the **real world**) and we have our camera on the right, in the **image plane world**, with its CCD. A model that we can use for examining how the image actually gets formed on the sensor is called **pinhole image projection**. If we imagine that there is an infinitely small hole, the yellow circle, which is basically like the camera aperture, the rays of light are passing through this small hole. From a point on the tree, **(1)**, rays of light pass through this pinhole and impinge on the **sensor**. Consequently, what appears on the sensor is a **tiny upside-down image** of the tree.
- If we have got an object in the world that is a certain size, **how big does the object appear on the CCD? How many pixels will this image take up?** The answer to this question is related to the **similar triangles**. As shown in B, a camera has a **focal length**, which we denote by length **f**. The focal length is the distance between the pinhole that lets the light into the camera and the location of CCD. This length is a very small number, certainly measured in the millimeters. Let us now imagine that we are looking at point **(2)** on the tree. That is basically straight through the middle of the sensor. Let us call this distance **Z**. Now, we want to know how high the tree appears to be on the physical CCD sensor. Let us call the tree height **X**. The height of the tree length on the CCD is represented by little **x**.

Perspective projection

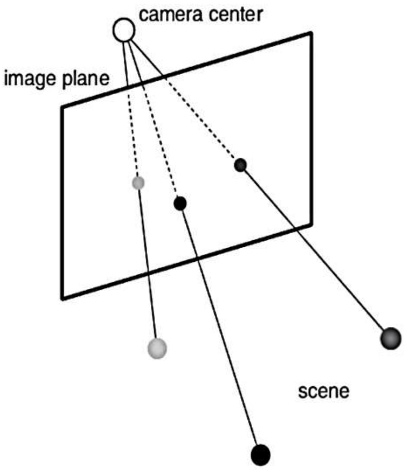


- By similar triangles, we can say that both angles in **A** are equal, so **X/Z = x/f**. That means that if we know the focal length and we also know how big the object is in the real world, we can get the size of image on CCD, i.e., **x = fX/Z**.
- Note that, here, we have two worlds, one is the world in which the object is (the real world) and another in which the image is (the image plane world). The idea is that in terms of measuring things, we assume that the pinhole of the camera is located in the image plane world at (**0,0,0**) coordinates.
- **Conclusion:** To compute **x**, given that we have **f**, we can measure and use the distance from the point (**0,0,0**), shown as **Z**, and the up and down distance from **(1)** to **(2)**, shown as **X**.

Perspective projection



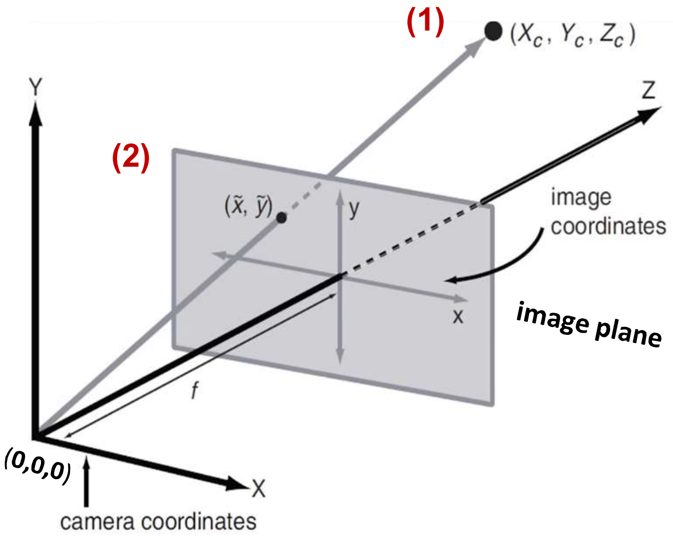
A



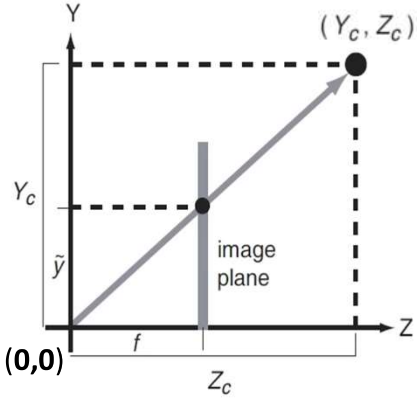
B

- The process we just discussed also works in three dimensions. Diagram **A** shows a better picture of how points in the scene pass through the **camera center** (also known as **optical center** or **aperture** or **pinhole**) and impinge on the **image plane**. We can see from this picture that everything on the image plane, just like in our retina, is upside down. So, the camera is going to turn it upside down before it passes it to us as an image.
- Diagram **B** shows another way of thinking about this problem. This diagram is mainly used for modeling. To do the modeling, we can imagine that the image plane is between the aperture (the **camera center**) and the world (the **scene**) and that way ***things seem to be right-side up***.

Perspective projection



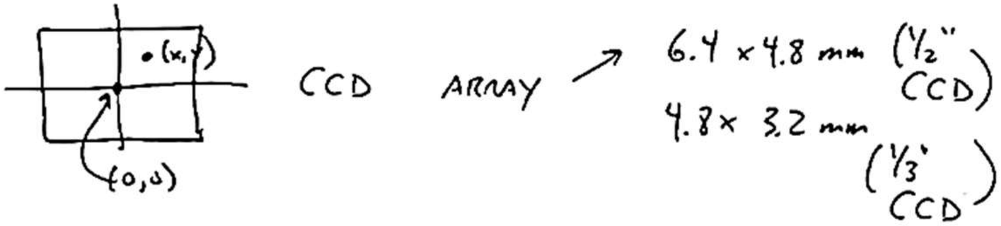
A



B

- **How the geometry works?** Here, we have got the image plane in gray, that is like the camera sensor. We can imagine that we are pushing through the image plane, from the origin, out into the world. Now, we take a coordinate out into the world that has three dimensions, point (**X_c, Y_c, Z_c**) and we project it onto the image plane, which has two dimensions, point $(\tilde{\mathbf{x}}, \tilde{\mathbf{y}})$. That is, we go from **(1)** to **(2)**, via the similar triangles Illustrated in **B**. Here, **$f/Z_c = \tilde{\mathbf{y}}\ /Y_c$**.
- **Conclusion:** Based on this model, we have a point (**X_c, Y_c, Z_c**) in the world coordinates by which we mean this is a 3D pointed world. We assume that the camera pinhole is at (**0,0,0**) in **XYZ** coordinates and then the projection of point (**X_c, Y_c, Z_c**) onto the image plane is $(\tilde{\mathbf{x}}, \tilde{\mathbf{y}})$.

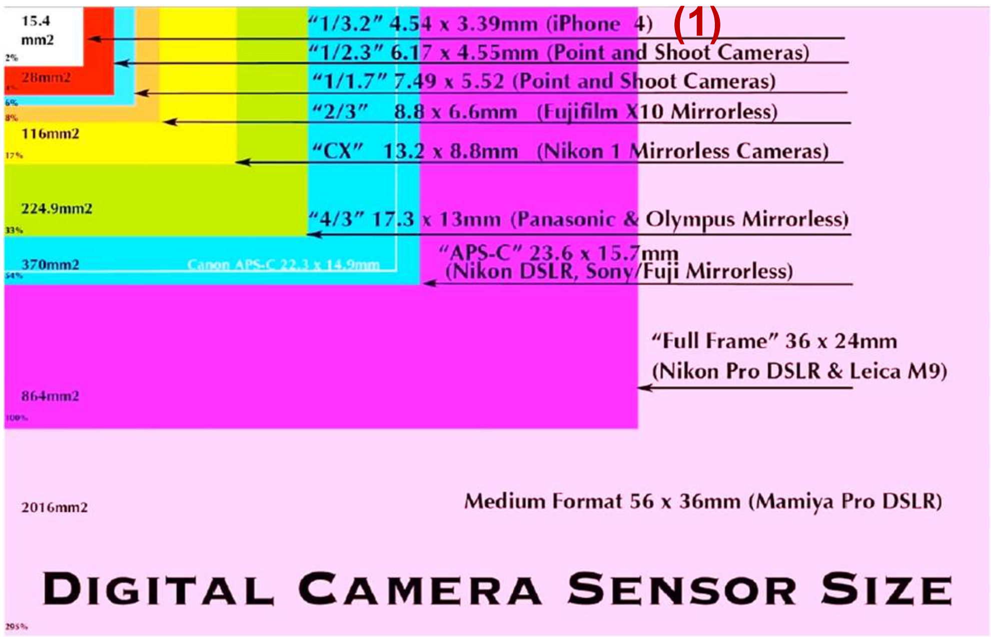
CCD array sizes and pixels



- On the CCD array, if we imagine that the point shown by (**0, 0**) is in the middle, then this will tell us where (**x, y**) is. We have to still think about what the units of everything are. Using the first model, the world is measured in say meters and when we do the ratios of **X/Z** or **Y/Z**, the units cancel out. Here, **f** is usually measured in millimeters. So, that means when we finish this computation, we are going to get an **x** and a **y**, measured in millimeters.

- Note that this (**x, y**) coordinate is not the same as pixels. In order to convert this to the pixel grid that we have in MATLAB or in Photoshop, we need to do one more conversion. This conversion will basically tell us how the pixels fit onto the CCD array. Typically, a CCD array is pretty small. For example, they are **6.4** by **4.8** mm (so-called **1/2** inch CCD) and **4.8** by **3.2** mm (so-called **1/3** inch CCD). Typical pixel sizes range from **1.4** µm to **5** µm.

CCD array sizes and pixels


15.4
mm2
2%
28mm2
4%
6%
8%
116mm2
17%
224.9mm2
33%
370mm2
54%
864mm2
100%
2016mm2
295%
"1/3.2" 4.54 x 3.39mm (iPhone 4)
(1)
"1/2.3" 6.17 x 4.55mm (Point and Shoot Cameras)
"1/1.7" 7.49 x 5.52 (Point and Shoot Cameras)
"2/3" 8.8 x 6.6mm (Fujifilm X10 Mirrorless)
"CX" 13.2 x 8.8mm (Nikon 1 Mirrorless Cameras)
"4/3" 17.3 x 13mm (Panasonic & Olympus Mirrorless)
Canon APS-C 22.3 x 14.9mm
"APS-C" 23.6 x 15.7mm
(Nikon DSLR, Sony/Fuji Mirrorless)
"Full Frame" 36 x 24mm
(Nikon Pro DSLR & Leica M9)
Medium Format 56 x 36mm (Mamiya Pro DSLR)
DIGITAL CAMERA SENSOR SIZE

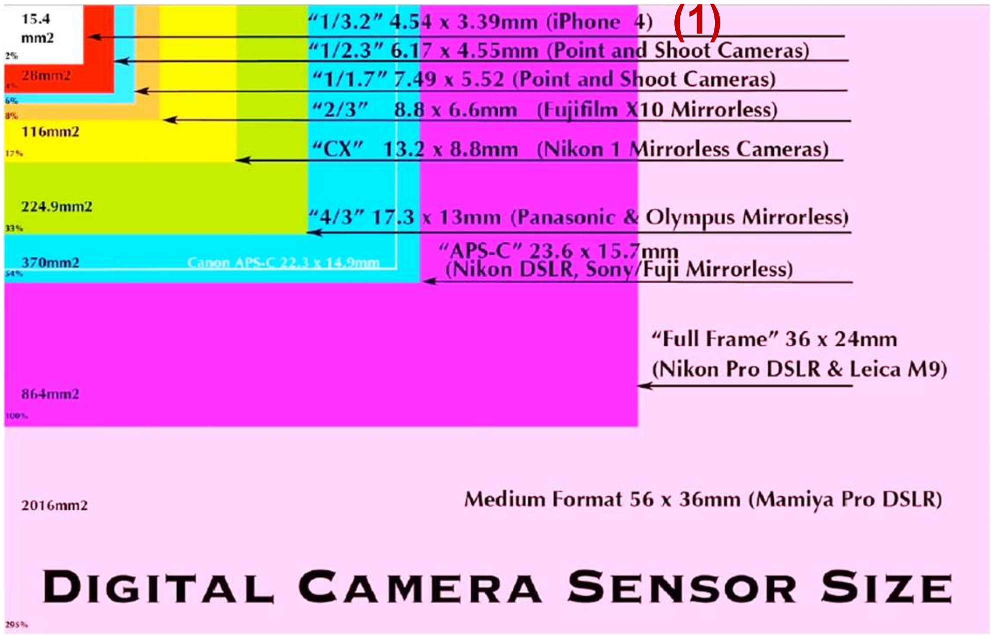
A

- **Digital Camera Sensor Size:** Picture **A** shows us how big the CCD array is on different systems. An old-school 35 millimeter SLR analog camera (SLR stands for Single-Lens Reflex) was pretty heavy and big. Comparing to digital cameras, such as the ones in iPhone, the analog cameras were very bulky. In digital systems, usually the sensor is much smaller, as shown in **(1)**. For iPhone 4, it is **4.54** mm by **3.39** mm. The latest iPhone model as of 2024 is the iPhone 16 Pro, released in September 2024 (**1/1.28** inch, **48** MP).

- In some sense, when we are talking about photography/imaging, one generally wants to have as much light onto the CCD array as possible. That is, we want to have a lot of light gathering ability and we also want the effective pixels to be getting as much light as possible. So, in general, if one is a photography person, they will think that having a bigger sensor is better. And, that is usually the first thing they look at when they are buying digital cameras, like a DSLR camera (Digital Single-Lens Reflex).

- Note that CCD (Charge-Coupled Device) sensors were widely used in cameras, especially during the early to mid-2000s. However, the industry has since shifted toward CMOS (Complementary Metal-Oxide-Semiconductor) sensors.

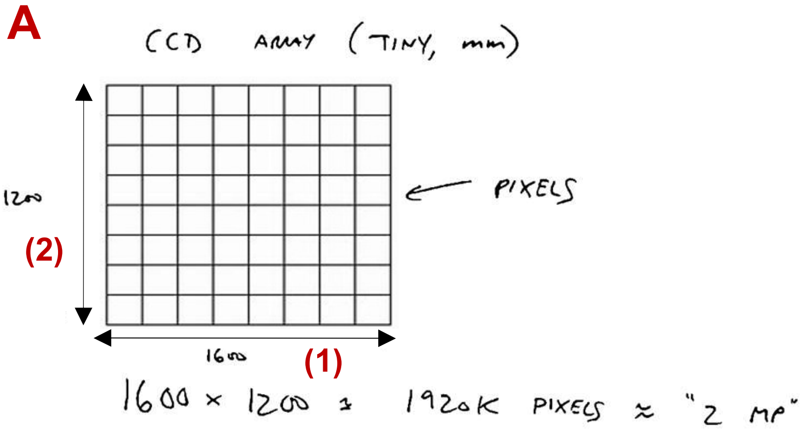
CCD array sizes and pixels


15.4
mm2
2%
28mm2
6%
8%
116mm2
17%
224.9mm2
33%
370mm2
54%
864mm2
100%
2016mm2
295%
"1/3.2" 4.54 x 3.39mm (iPhone 4)
(1)
"1/2.3" 6.17 x 4.55mm (Point and Shoot Cameras)
"1/1.7" 7.49 x 5.52 (Point and Shoot Cameras)
"2/3" 8.8 x 6.6mm (Fujifilm X10 Mirrorless)
"CX" 13.2 x 8.8mm (Nikon 1 Mirrorless Cameras)
"4/3" 17.3 x 13mm (Panasonic & Olympus Mirrorless)
Canon APS-C 22.3 x 14.9mm
"APS-C" 23.6 x 15.7mm
(Nikon DSLR, Sony/Fuji Mirrorless)
"Full Frame" 36 x 24mm
(Nikon Pro DSLR & Leica M9)
Medium Format 56 x 36mm (Mamiya Pro DSLR)
DIGITAL CAMERA SENSOR SIZE

A

- **Depth of Field:** For professional photographers or medical scanner operators, the notion of **depth of field** is also important. Depth of field (**DOF**) refers to the distance between the nearest and farthest objects in a scene that appear acceptably sharp in an image. The idea is that when we take a portrait, we want the object or person that is close to the camera (the object or the person that we are shooting) to be highly in-focus. And also, it strikes us as a pleasing picture if the background is out-of-focus. Here, it seems to us that the object or the person pops if it is sharper than the background.
- It is hard to get a nice depth of field with a tiny sensor and a tiny focal length. We need a really good camera to be able to do this so that we could hallucinate that depth of field. One way of doing this is to incorporate a **range image sensor** into our camera that bounces infrared light off the subject and figures out how far away they are. This way we can automatically keep some far-away objects in-focus and blur the background.

CCD array sizes and pixels



- **Image Resolution:** In **A**, we have a CCD array which is really fundamentally some tiny thing measured in units of millimeters. Then, this array is split up into **pixels** (picture elements). That is, every digital image is composed of these tiny squares or pixels. Each pixel holds color information and contributes to the overall image. This is where we get notion of image resolution and megapixels. Generally, **image resolution** refers to the level of detail an image contains. So, in digital imaging, image resolution is directly related to the number of pixels present in the image.

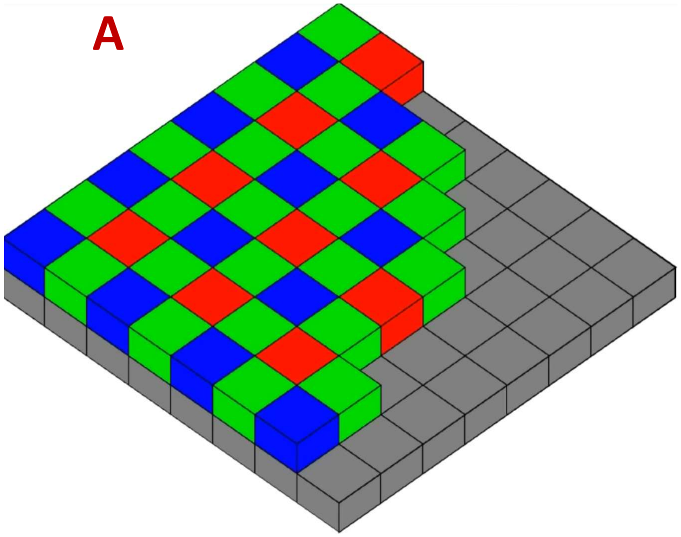
- For example, an image might be **1600** by **1200** pixels, which means that there are **1600** little sensors crammed into the CCD array, shown by **(1)**, and **1200** little sensors crammed into the CCD array, shown by **(2)**. If we multiply these two numbers together, we get **1,920,000** pixels, which is roughly a **2** MP (megapixel) image.

- Usually, we do not want to just go for the camera or a scanner that has the largest number of megapixels. This is because if the image sensor is not very big on the camera, then every one of those pixels is effectively gathering less light. Instead, what we want to look for first is ***the size of the physical sensor*** and ***then*** we worry about ***how that sensor has been divided into pixels***. Typical pixel resolution changes day to day. We can easily record four thousand pixels wide (**4K** wide). One thing to remember though is that megapixels is not exactly the best or the most obvious measure of the quality of an image.

Bayer array; color sensing

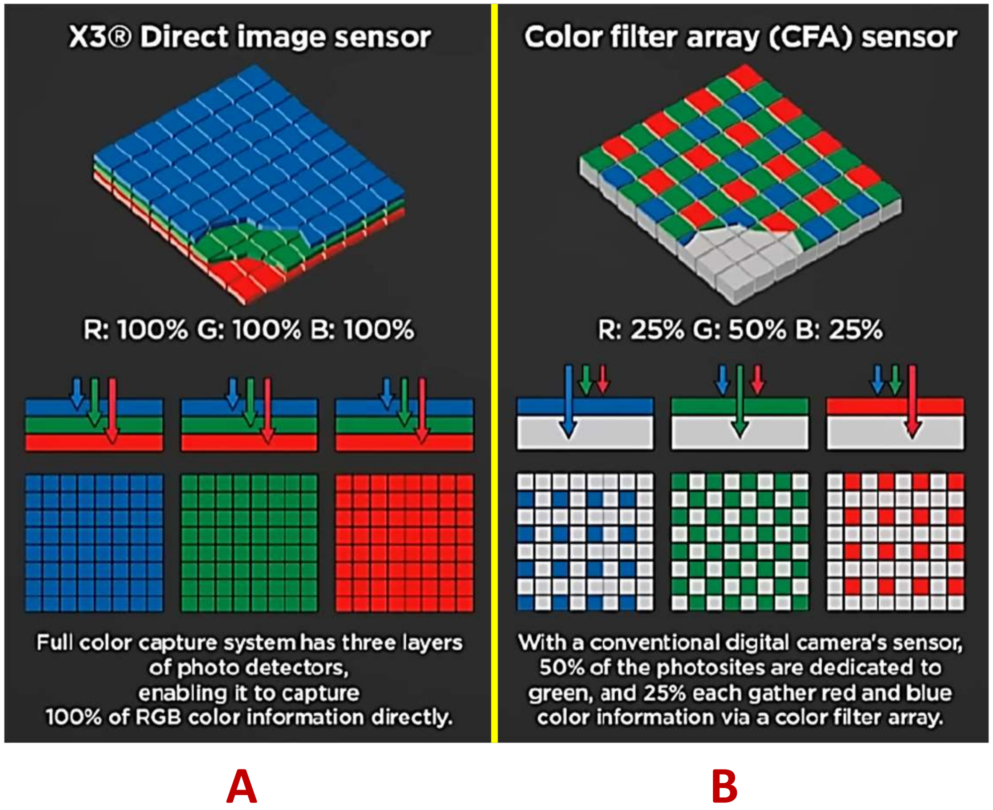
PIXEL COLOR RESPONSES ARE USUALLY ARRANGED IN A "BAYER PATTERN"

A



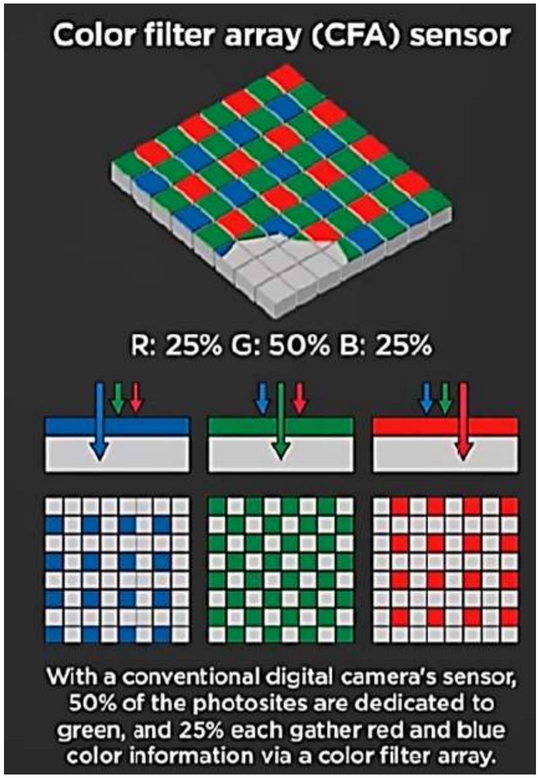
- **Bayer Pattern:** We want to acquire the red, green, and blue values for all these points at every pixel. When we open up an image in Photoshop, we can get RGB at each pixel. The pixels in most digital cameras are the pixel color responses that are usually arranged in what is called **Bayer Pattern**. That means the responses of the pixels are not like we get a raw reading of red-green-blue at every pixel. Here, every pixel in the digital camera has only one actual color reading, either red, or green, or blue and typically they are arranged in the way shown in **A**.
- We talked about how the ***human eye is most sensitive to green*** and so that is why half of the pixels in this **Bayer pattern** are **green**. A quarter of them are **red** pixels and a quarter of them are **blue** pixels. That means that when our digital camera gives us back RGB, it is doing what is called **demosaicing**. This means it is kind of filling in or it is ***hallucinating*** (a better term is ***educated- guessing***) the other two channels that it did not actually physically acquire at every pixel. Demosaicing (or **demosaicking**), also known as **color reconstruction**, is a digital image processing algorithm used to reconstruct a full color image from the ***incomplete color samples*** output from an image sensor overlaid with a color filter array (CFA) such as a **Bayer filter**. It is also known as **CFA interpolation** or **debayering**. That is, demosaicing algorithms analyze the surrounding pixels and their colors to estimate the missing color information for each pixel.

Bayer array; color sensing


X3® Direct image sensor
R: 100% G: 100% B: 100%
Full color capture system has three layers of photo detectors, enabling it to capture 100% of RGB color information directly.
Color filter array (CFA) sensor
R: 25% G: 50% B: 25%
With a conventional digital camera's sensor, 50% of the photosites are dedicated to green, and 25% each gather red and blue color information via a color filter array.
A
B

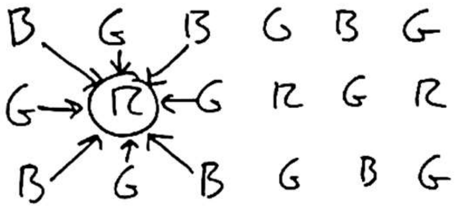
- **Direct Image Sensor (Foveon Sensor) vs. CFA Sensor:** In direct image sensor, shown in A, every pixel is actually acquiring red, green, and blue. However, this is typically not the way that most current digital cameras work. That is, they usually use the Bayer array, shown in B. Foveon sensor is used for high-end cameras and uses a ***layered approach***. Here, we have a vertically stacked layer design, unlike the Bayer pattern with a single layer. Each layer captures light with varying sensitivity to different wavelengths (colors). The top layer is most sensitive to blue light, the middle layer to green, and the bottom layer to red.

Bayer array; color sensing



With a conventional digital camera's sensor, 50% of the photosites are dedicated to green, and 25% each gather red and blue color information via a color filter array.

A

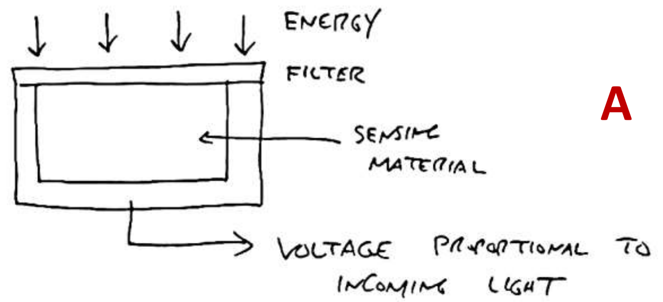


B

"DEMOSAICING"

- **Color Filter Array (CFA) Sensor:** In **A**, we have this so-called Bayer array. Here, we can see the pattern of the blue responses, the green responses, and the red responses that we get.
- Let us start with a Bayer pattern, as shown in **B**, and take a look at the letter **R** which has been encircled. Here, we know the **R** exactly because the sensor captured that. If we want to infer the green ***at the same pixel***, what we would do is to average the four surrounding green pixels together to get the green at that point. And then, to get the blue, we would average the four diagonal blue channels. This process is called **demosaicing algorithm**.
- **Conclusion:** The Bayer pattern with demosaicing offers a clever solution to capture a good balance of light and color information with a limited number of sensor pixels. It prioritizes green sensitivity to align with human perception and uses intelligent algorithms to reconstruct a full-color image despite each pixel not capturing all color data directly.

Illumination model



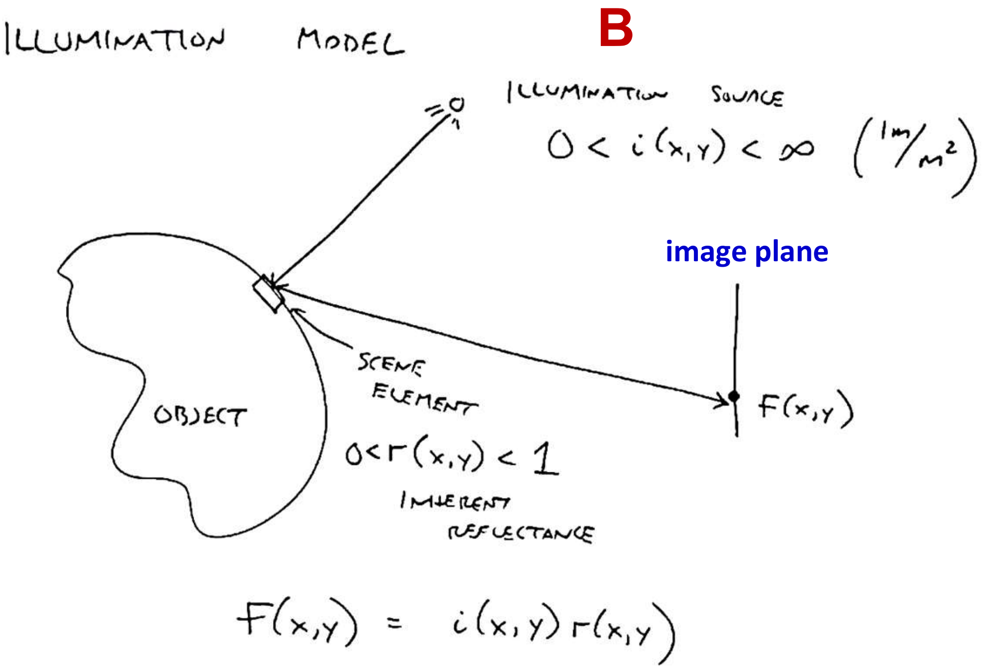
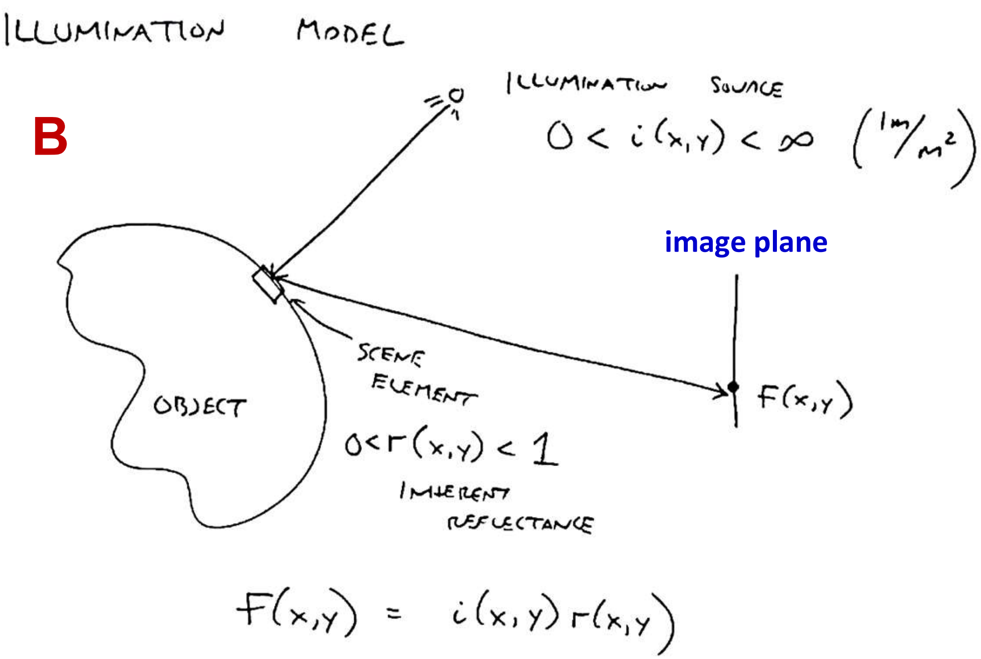


image plane

- **Illumination Model:** Let us now talk a little bit about the ***light response*** and the ***illumination model***. Picture **A** shows the light energy that is coming onto the CCD array. In this picture, the question is **how much light energy is actually reaching this array?** The answer is that it depends on what is going on in the scene, shown in **B**. Here, the little region of the object, shown in **B**, is called the **scene element**. Let us say we have some object in the scene and we want to know how much light is arriving at point **(x,y)** on the image plane, i.e., **F(x,y)**. On this plane, we have our image sensor. Usually, we have some sort of an **illumination source** and this could be the Sun, it could be a spotlight, or it could be a glowing object emitting light. Light from all the sources of the scene is bouncing off the object and then some of that light is bouncing into the **image sensor**.
- The amount of light energy that gets recorded on the sensor is a combination of both ***the strength of the illumination source***, **i(x,y)**, and ***the inherent color of the object***. For example, the illumination source can have infinite brightness, i.e., **0 < i(x,y) <** ∞, where, **i(x,y)** has the units of **lm/m²** or **lumens/m²**. Here, lumens per meter squared (lm/m²) represent the **illuminance**, which is the amount of visible light falling on a surface per unit area. In simpler terms, it tells us how bright a surface appears due to the light source.

Illumination model



- In **B**, the scene element has some sort of **inherent reflectance** that is basically a measure of how shiny the object is. Fundamentally, what we receive at point **(x,y)** in the sensor is the product of the illumination source ***intensity function***, **i(x,y)** and the ***reflectance***, **r(x,y)**. Here, we have **0 < r(x,y) < 1**, and so, **F(x,y) = i(x,y).r(x,y)**.
- This is a simplified model because it is assuming that the color or the intensity at one pixel is only coming from one point in scene. That does not take into account things like shadows, multiple illumination sources, and transparency. All that stuff is required to accurately build a model. In reality, by using this simple model, we might realize that the image looks unnatural unless we take into account, e.g., multiple bounces of light. So, this is a crude model for what happens ***along each light ray***.

Illumination model

	i(x,y)	
CLEAR SUNNY DAY	90K	lm/m^2
CLOUDY DAY	10K	lm/m^2
INDOORS	1K	lm/m^2
FULL MOON	0.1	lm/m^2

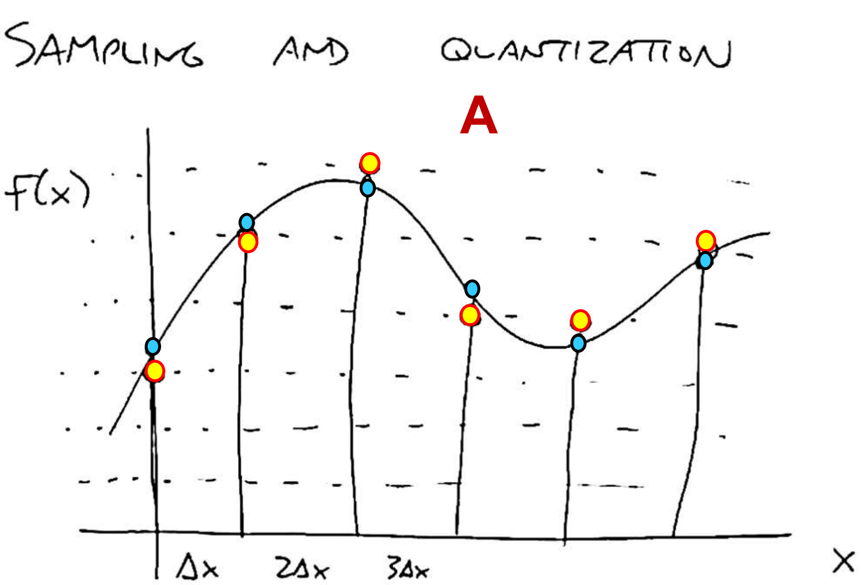
	r(x,y)
SNOW	0.93
FLAT WHITE WALL	0.80
STAINLESS STEEL	0.65
BLACK VELVET	0.01

- **How bright can illumination sources be?** There is a pretty wide range for **i(x,y)**. If we have a clear sunny day, that is about as much light as we can imagine naturally coming into the world, and that turns out to be about **90,000** lumens per meter squared or **90 klux** (where **lm/m^2** or **lux** or **lx**). A cloudy day is a little bit dimmer, about **10** times less, i.e., **10 klux**, and indoors is another order of magnitude less, about **1 klux**. And about as dark as we could imagine seeing stuff, like the full moon, is another order of magnitude lower, i.e., **0.1 klux**. So, we could imagine that in our natural world, we have a very wide range between **90 klux** for the brightest possible situation, to the full moon, which is about **0.1 klux**.

- **How reflective objects can be?** We know that **r(x,y)** is a number between **0** and **1**. That basically says how much of the incoming light is bounced back to our eye. For example, snow is about as reflective as we can get, i.e., about **0.93**. This is because so much light is reflected back from the snow into our eyes. A flat white wall, like the ones in a typical classroom have a reflectance about **0.80**. Stainless steel is about **0.65** and black velvet, which seems to suck all the light out of it, has a very low reflectance of **0.01**.

- **Conclusion:** This above concepts are telling us how much light actually shows up on the CCD sensor. It also shows that there is a complicated combination of properties at work, such as the brightness of the illumination source coming from the world, the inherent properties of the objects that are being imaged, and the sensitivity of the sensor.

Sampling and quantization

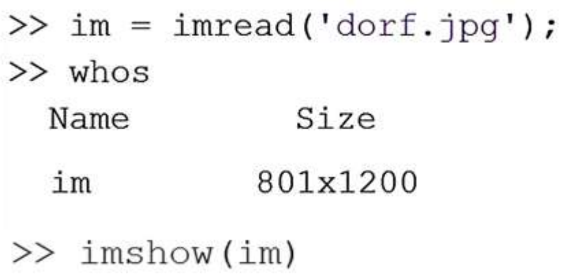


- **Quantization** in image processing is a technique used to reduce the amount of data required to represent a digital image. It involves simplifying the image data by reducing the number of possible values each pixel can hold.

- **Sampling and Quantization:** In **A**, and along the **x**-axis, we do not get continuous samples of the continuous function **F(x)**. Instead, what we get are discrete samples. This is called **sampling**. Here, on the **x**-axis, we will have **Δx**, **2Δx**, **3Δx**, and so on. We know from the Nyquist theory, when we sample this function, if the functional samples are close together, we can reconstruct the continuous function more closely. The same idea applies to our digital image. Because, here, what we really get is a series of discrete pixels. We really do not have any sort of continuous world when it comes to real-world image processing. It would have been true if we were talking about the kind of images that are produced on film (here, there really is no sense of discretization). ***But in the world of digital images, we have these rigidly boxed-in pixels.***

- The other thing that is important is **quantization**. Quantization in image processing is done on the **y**-axis, which represents the amplitude or intensity values of an image. In **A**, we have **F(x)** as the **y**-axis. For **F(x)**, we may not actually be able to represent the intensity or the color of a pixel with what we would call **floating-point precision** (shown by **blue** circles). So, instead, the samples that we get are actually ***rounded*** to the nearest quantization level (shown by **yellow** circles). That is, we get the yellow points instead of the blue points in **A**. Alternatively, this means that the measured light intensity on the pixel is not exactly the energy that is arriving in sensor, i.e., it has been quantized.

MATLAB demo

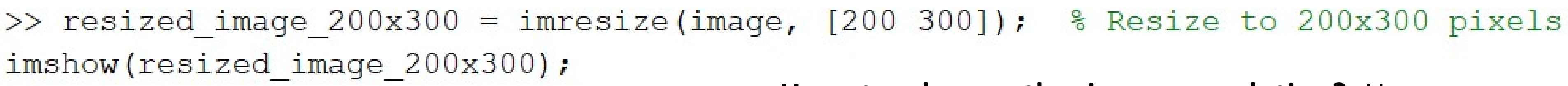


```
>> im = imread('dorf.jpg');
>> whos
  Name          Size

  im           801x1200

>> imshow(im)
```

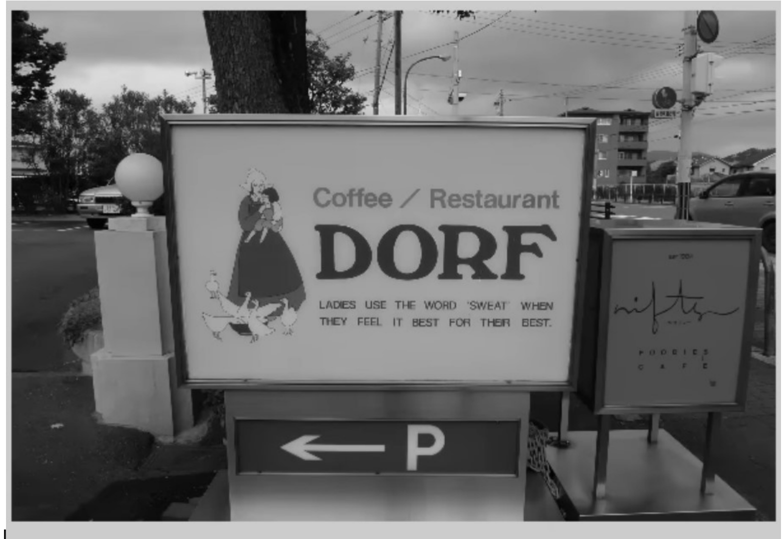
Coffee / Restaurant
DORF
LADIES USE THE WORD 'SWEAT' WHEN
THEY FEEL IT BEST FOR THEIR BEST.
← P

- Let us take a look at an example and see how to change the sampling (pixel numbers) and quantization using MATLAB tools. This image is already a grayscale image, but if the image was originally in RGB, we can use the following command to change it to grayscale: **grayscale_image = rgb2gray(rgb_image)**.
- By typing **whos**, we can see the innate original resolution of the image, which is **801** (height) by **1200** (width).
- **Note:** This difference in convention is important and is why there can be some confusion when switching between contexts. **In MATLAB (and other programming contexts where matrices are described), we use Rows × Columns (height × width).** However, in image dimensions notation (such as in Photoshop, image processing, or general use), we use Columns × Rows (width × height).
- Here, **imshow** is a specific function in MATLAB used for ***displaying an image***.
- **Conclusion:** The command **imread** is like ***putting*** an image file (like a photograph) into a box. The command **imshow** is like ***opening*** the box and looking at the image inside.

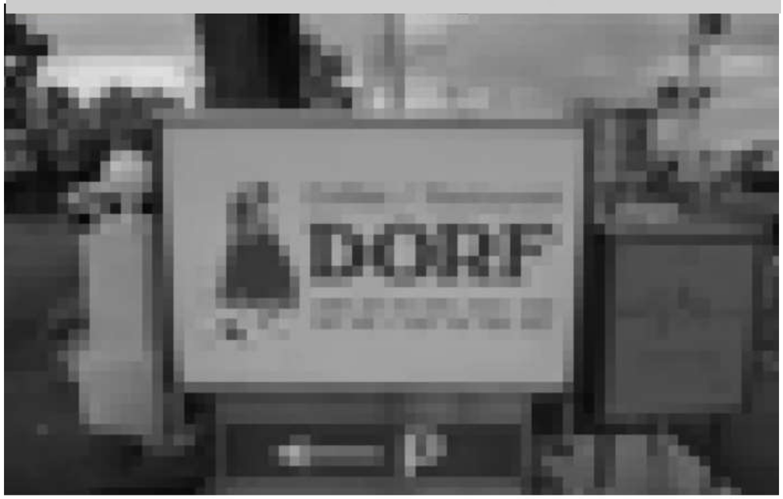
MATLAB demo



```
>> resized_image_200x300 = imresize(image, [200 300]);  % Resize to 200x300 pixels
imshow(resized_image_200x300);
```


Coffee / Restaurant
DORF
LADIES USE THE WORD 'SWEAT' WHEN THEY FEEL IT BEST FOR THEIR BEST.
P

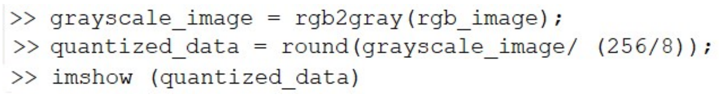
A



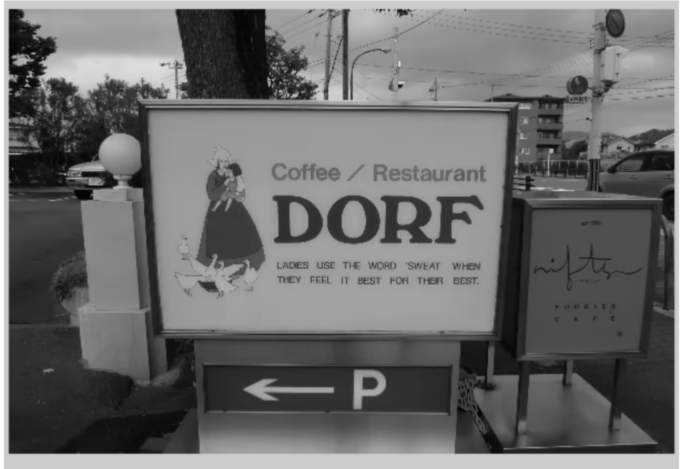
B

- **How to change the image resolution?** Here, we are basically making the pixels larger or making things ***blockier***. We can use the command **imresize (image, [a b])** to change the resolution in MATLAB. The arguments **a** (represents the new number of rows or height, and **b** (represents the new number of columns or width) are the number of pixels. Here, we are specifying the desired new **height** and **width** of the image as a two-element vector. For example, **[200 300]** will resize the image to **200 rows** and **300 columns**. Now, ***in MATLAB***, we have a **200**×**300** pixel image.
- Changing the resolution is like the effect of saying that if we had a lower quality image sensor and the pixels on the physical CCD were bigger, what would the resulting image look like? So, the manifestation of it is that as we make the pixels larger. That is, things seem to look blockier and blockier to the point that we cannot even tell what is going on in the image. In **B**, we have a lower resolution, i.e., bigger pixels, compared to **A**. We might argue that we could go one unit down and still be able to read the text, but another unit down and we may not be able to see what is going on.
- **Conclusion:** The image resolution, which is determined by the number of pixels, certainly has a huge effect on the usefulness of an image.

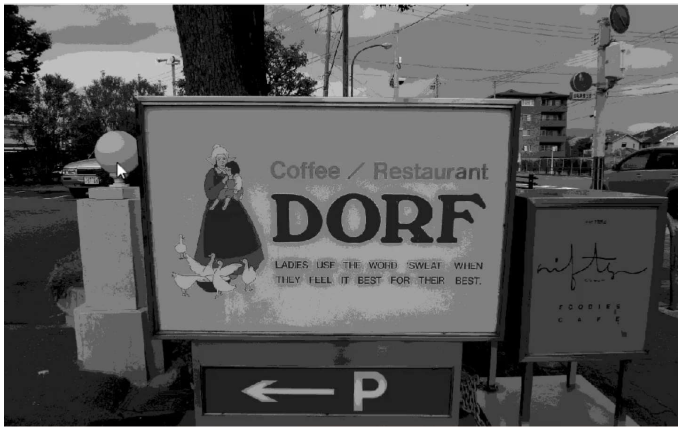
MATLAB demo



```
>> grayscale_image = rgb2gray(rgb_image);
>> quantized_data = round(grayscale_image/ (256/8));
>> imshow (quantized_data)
```


Coffee / Restaurant
DORF
LADIES USE THE WORD 'SWEAT' WHEN THEY FEEL IT BEST FOR THEIR BEST.
FOODIES CAFE
← P

A


Coffee / Restaurant
DORF
LADIES USE THE WORD 'SWEAT' WHEN THEY FEEL IT BEST FOR THEIR BEST.
FOODIES CAFE
← P

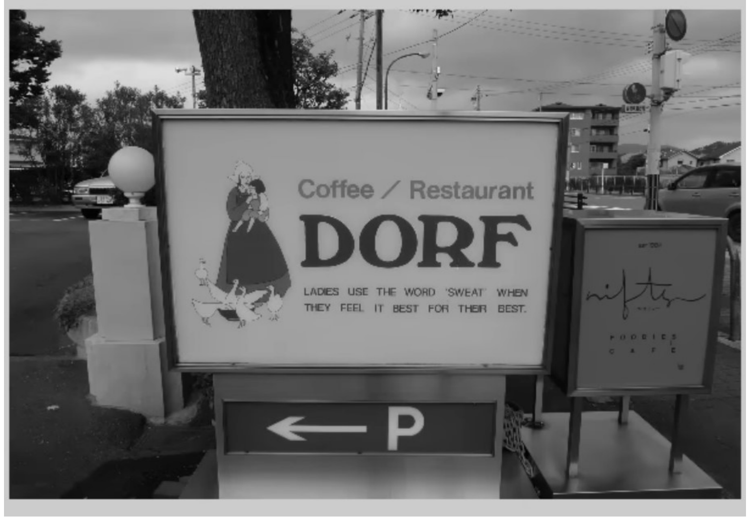
B

- **How to change the quantization levels?** This basically means how many **gray levels** we are using to represent the image. In **A**, we have an image with **256** gray levels, which is pretty standard. We call this an **8-bit image**. We can change the number of gray levels, which means we are kind of ***rounding*** the pixels to fewer and fewer levels.

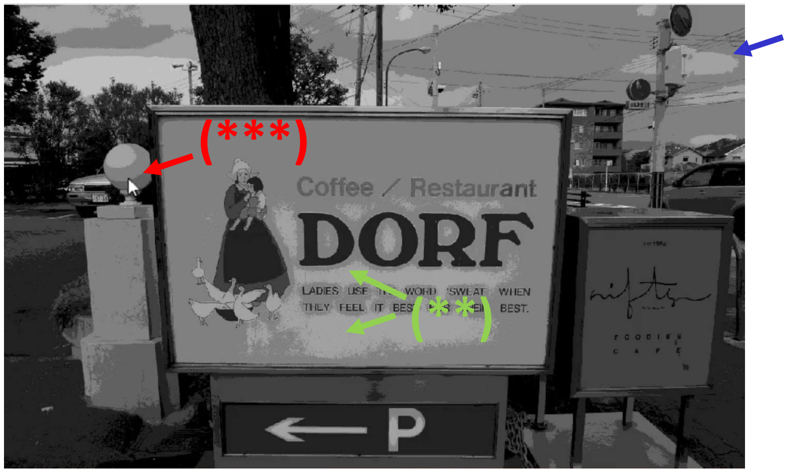
- The command **round(image_data / (256/n))** is used to change the **number of grayscale levels** in an image, and hence, to quantize the image to a different number of grayscale levels. Here, **n** is **the number of desired intensity levels** (e.g., **16**, **8**, or **1**). In **B**, the command **round(image_data / (256/8))**, first, divides **256**/**8**, which is equal to **32** (or **Δ = 32**). This effectively reduces the possible intensity values to approximately **n = 8** levels (**0**, **32**, **64**, **96**, **128**, **160**, **192**, **224**). There might be some rounding up or down depending on the original values. If **n = 1**, i.e., **256**/**1** = **256** (or **Δ = 256**), we will have **2** shades of gray (**0**, **255**). The resulting image will appear highly ***contrasted*** and ***blocky***, as most pixels will become either pure black (**0**) or pure white (**255**). When **n = 256**, we have the full range of grayscale levels for an **8**-bit image, which is the maximum resolution for typical grayscale images.

- While technically possible, using only **2** grayscale levels is not common in digital image processing ***due to the loss of detail***. In practice, quantization often aims for a balance between reducing data and preserving a reasonable amount of image quality. Common choices might be **n = 8**, **16**, **32**, or **64** levels, which offer more nuanced grayscale representations.

MATLAB demo


Coffee / Restaurant
DORF
LADIES USE THE WORD 'SWEAT' WHEN THEY FEEL IT BEST FOR THEIR BEST.
P

A



B

- **False Contouring:** In B, let us look at the upper right hand corner, shown by (*). This is called **false contouring**. Originally, as shown in A, the cloudy area of the upper right hand sky is a nice continuously changing gray level. But, as we change the quantization, we start to see here that there is some false contouring happening. Now, it looks like there is a band of a circular dot of white pixels. And, if we pick it down even more, we start to see that what used to look like smooth continuous changes in grayscale now look like blotchy rings of false contouring.
- We see the same thing on the sign. Even though the original sign in A, looked like maybe it was roughly the same color, as we kick down the number of desired intensity levels (decreasing n), we can see that the area above and below the bottom text, (**), becomes a little lighter than the background area that contains the text. Now, we start to get this kind of false contouring on the sign too. Additionally, we can also see this false contouring on the corner of the post, shown in B by (***).

MATLAB demo

WE OFTEN USE 8 BITS = 256 LEVELS PER COLOR CHANNEL

$$\underset{R}{2^8} \times \underset{G}{2^8} \times \underset{B}{2^8}$$

- **Color Depth in DIP:** In essence, **color depth** is a measure of the number of colors an image can display, impacting the richness and detail of the colors we see. In this context, color depth is a special case of **bit depth** that is used in digital imaging. In DIP, we often use **8** bits (**8 bit per pixel** or **bpp**), which is the same as **256** levels per color channel. That means that if we have a **red-green-blue** image, we have **2** to the **8**th **red**, **2** to the **8**th **green**, and **2** to the **8**th **blue** colors. That would still be called an **8-bit color image**. In Microsoft Windows, there is a little drop-down box that would say how many colors do we want for our display to be? There is **16** colors, **256** colors, and there is one that is called **millions of colors** (also called **True Color** or **24-bit depth**). The last one is usually about as good as we could get.

- The **8**-bit color image is the most common option, especially for web graphics and applications where file size is a concern. It offers a good balance between **color representation** and **file size**. Here, each **color channel** (red, green, and blue) has **2^8 = 256** possible values, resulting in a total of **256^3** (over **16 million**) possible color combinations.

- Modern displays generally support True Color (**24**-bit depth) or even higher color depths (**32**-bit or more). These options provide a vast spectrum of colors, significantly exceeding the limitations of older systems.

- When we look at image file formats, it is true that there are some ways of storing images that use more than **8**-bit. For example, the TIFF standard has an option to have **8**, **12**, or **16** bits of color. But for the most part, if we download an image off the web, a PNG or a JPEG, most probably it is going to have **8** bits of color in it.

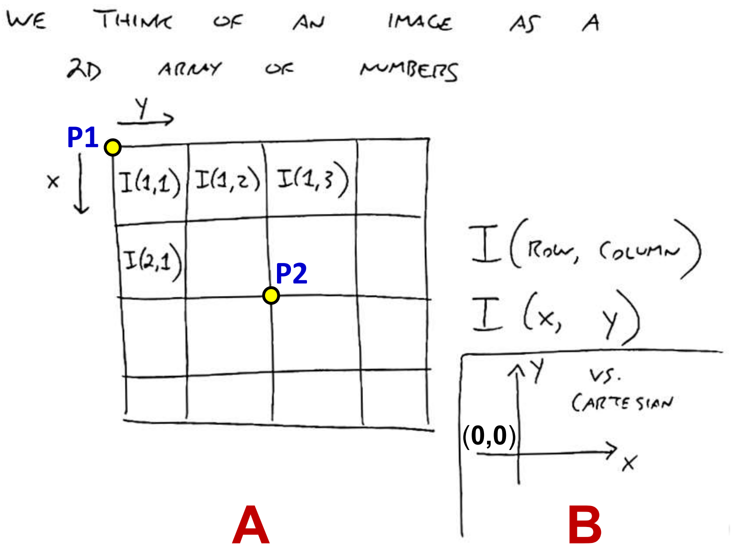
MATLAB demo

LOWER SAMPLING RATE (RESOLUTION)
→ BLOCKINESS

LOWER # OF INTENSITY LEVELS
→ FALSE CONTOURING (LOW DETAIL)

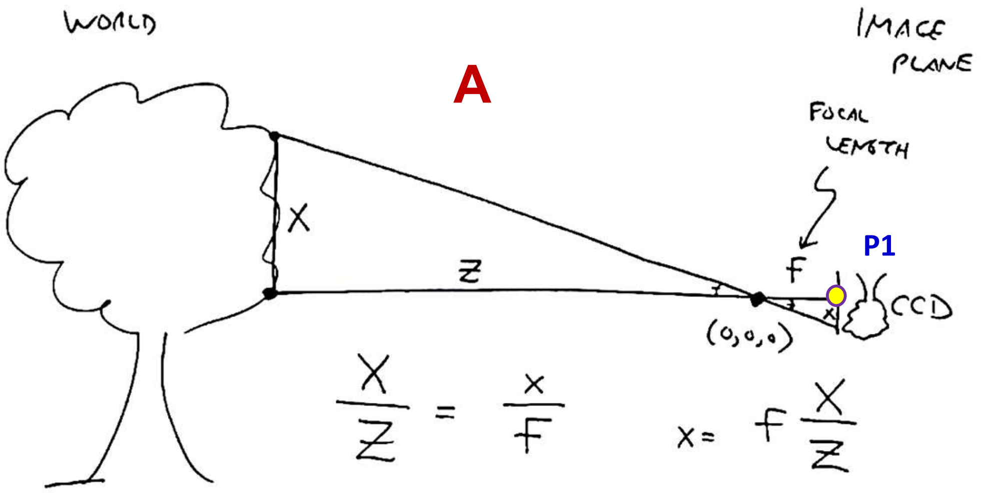
- **Image Blockiness:** In DIP, **Image Blockiness** is mainly due to **low resolution** and **downsampling** (reducing the image size). Here, low resolution is the primary culprit for blockiness. When an image has a low resolution (fewer pixels), it captures less detail and sharp edges can appear ***stair-stepped*** or ***pixelated***, leading to a blocky appearance. Another culprit is downsampling. Here, reducing the image size (downsampling) without proper filtering can cause blockiness. Downsampling discards pixels, and if not done carefully, it can lead to sharp transitions becoming blocky.
- **Image False Contouring:** Image false contouring is mainly due to **low color depth** (or low bit depth). Low color depth, specifically a low bit depth per color channel (red, green, blue), limits the image's ability to represent smooth color variations. This can lead to false contouring, where gradual changes in color appear as ***abrupt bands*** or steps. Another culprit in false contouring is **quantization**. As discussed before, this is a process used in image compression where color values are rounded or mapped to a limited set of discrete values. While necessary for compression, aggressive quantization can introduce false contouring artifacts if the bit depth is insufficient.
- **Conclusion:** A lower sampling rate (e.g., low resolution or fewer pixels) primarily leads to blockiness, where sharp edges appear jagged or pixelated. A lower bit depth and aggressive quantization mainly result in false contouring, where smooth color transitions appear as distinct bands, and there is a general loss of detail.

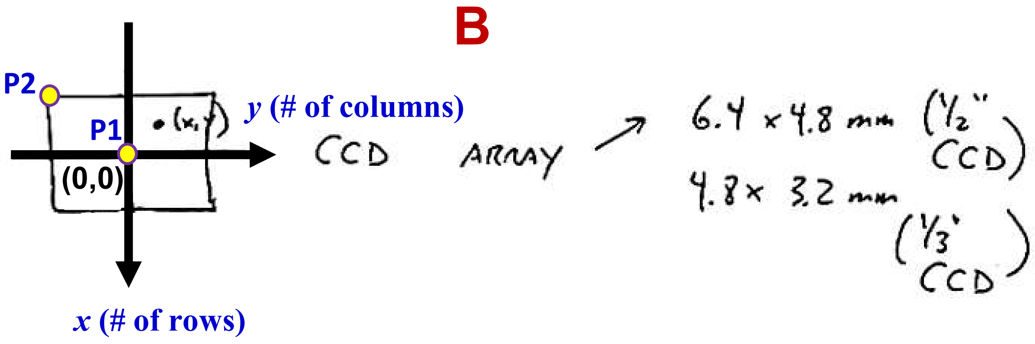
Image coordinate systems



- We usually think of an image as a **2D array of numbers**, as shown in **A**. Here, we will be mostly using grayscale images since it is easier to explain things as a function of the one number in the 2D array, which is the grayscale level. If we want to make it work on RGB images, it is relatively straightforward to do so.
- There is sometimes a little bit of disconnect when we are thinking about ***naming matrix elements***. The format in **A** is the standard way of talking about the entries of the matrix. Usually, **x** refers to the row and **y** refers to the column. The problem with this arrangement is that when we look at the direction of increasing **x**, and the direction of increasing **y**, we can run into problems when we start to talk about things in **Cartesian coordinates**. When plotting a graph of a calculus function, we show **x** and **y** as represented in **B**.
- Another confusion is again how MATLAB presents these numbers. Engineers will probably want to call **I(1,1)** the (**0,0**) element of image, but we are going to stick with calling up that the (**1,1**) element of the image because we will mostly use the MATLAB way of thinking.
- A bigger problem along those same lines is that there are some cases where it is more convenient to think about the center of the image, point **P2**, being (**0,0**) and some cases where it is more convenient to think about the (**0,0**) being at the upper left, point **P1**. So, we need to be clear about when we need to use each one of those things.

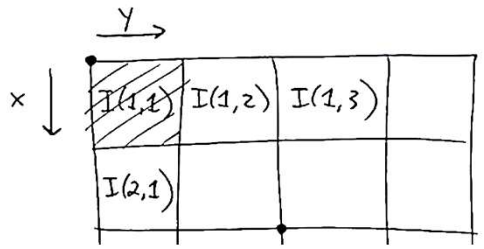
Image coordinate systems



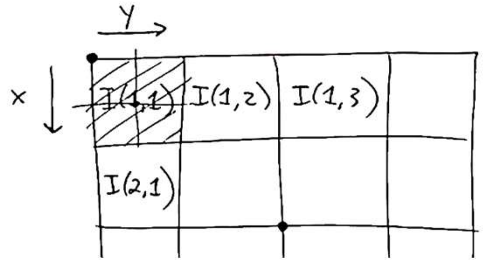


- **Example for Coordinates Conversion:** For instance, in **A**, we are implicitly assuming that the middle point of the CCD, i.e., point **P1**, is (**0,0**). This means that our coordinates in **B** are going to count from (**0,0**) in the middle. So, if we had a **600** by **800** image, point **P2** in **B** would be (**x = -300, y = -400**) instead of being (**0,0**). Note that an **600×800** image ***in MATLAB*** means that we have **600** rows (height) and **800** columns (width).
- In Photoshop and many other image editing programs, by default, the origin (**0,0**) is typically located at the top-left corner of the image. So, if we want to convert image **B** to Photoshop pixels, we would also need to add half the width and half the height (also sometimes called **the half length**) to make sure we do not have negative **xy** locations.
- The above approach is just a convention. When we start to program image processing algorithms, we want to make sure that we are doing the above conversion in the right way and just following the convention.

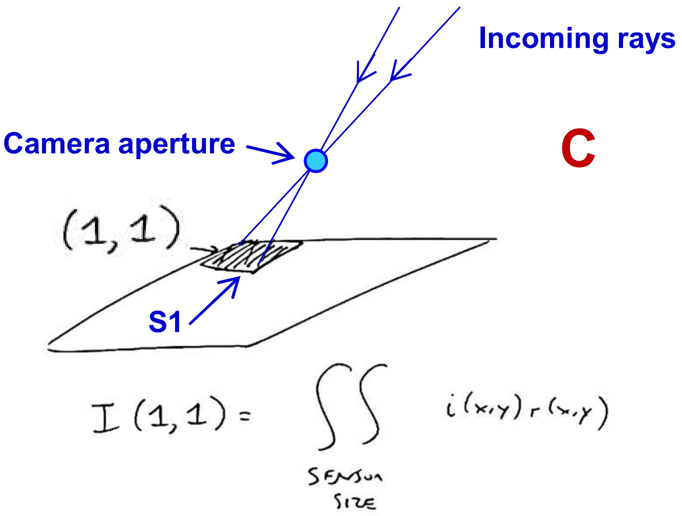
Image coordinate systems



A



B



C

- In **A**, when dealing with **(1,1)** coordinate, we are basically looking at this whole pixel as the (**1,1**) pixel, which is shaded for more clarification. Another way of thinking about it, is to consider the (**1,1**) as the middle of the pixel (shown in **B**) or to consider it is as a corner pixel. For the purpose of this class, it will not matter and we can also think of the (**1,1**) being at the center of the pixel.
- In **C**, the way to think about this is that we have an array of sensors. Let us call the (**1,1**) sensor **S1**. The camera aperture and the rays of light coming through it are also depicted. This physical sensor has some finite extent. So, the (**1,1**) value, **I(1,1)**, is the integral over the sensor size of all of this illumination and reflection.
- In summary, when thinking about this (**1,1**), we consider it as one block, and we do not have to worry about being centered or not.

Useful MATLAB commands

imread

imshow

imshow (a , [ ])

↳ SCALE SMALLEST VALUE TO BLACK

SCALE LARGEST VALUE TO WHITE.

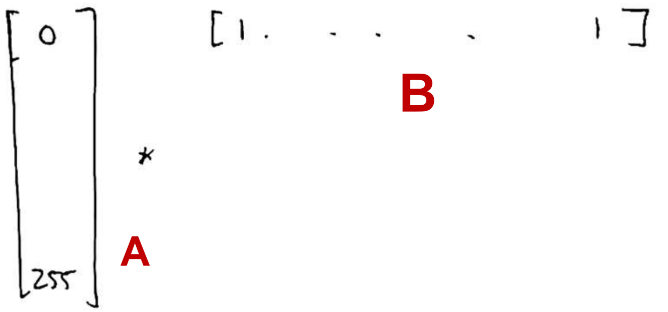
rgb2gray

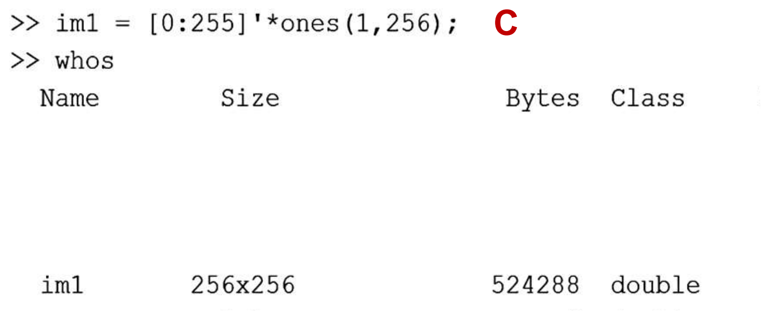
$A + B$, $A - B$, $\frac{1}{2}A + \frac{1}{2}B$

uint8 vs. double

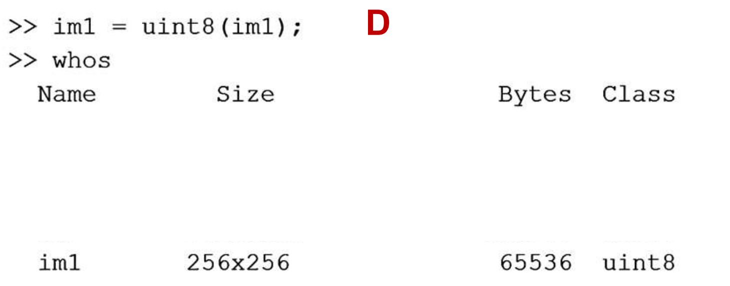
- **Some Useful MATLAB Commands:** We already talked about “**imread**” and “**imshow**”. In **imshow (a, [ ])**, the pair of brackets basically says “scale the smallest value to black and the largest value to white”. Here, **a** is the first argument and represents the variable containing the image data we want to display.
- A lot of times we want to do some image processing in which we need to ***read in*** a color image and then we try to apply some sort of grayscale operation to it. But, MATLAB might get back to us with an error saying this is actually a color image. In that case, we can convert the color image to grayscale with command “**rgb2gray**”.
- In some cases, we may have an image that looks grayscale to us but stored as a file with color image format, such as with three color channels. Here, we might also need to get rid of the three color channels and convert that to one, using **rgb2gray**.
- We could definitely do various other image processing operations, like adding two images, **A + B**, subtracting two images, **A – B**, and averaging two images, **1/2A + 1/2B**.
- Another subject that is important to remember is the difference between a MATLAB **unsigned 8-bit integer** (**uint8**) and a **double**, which will be discussed next.

Useful MATLAB commands





```
>> im1 = [0:255]'*ones(1,256);    C
>> whos
  Name         Size                 Bytes   Class

  im1        256x256               524288   double
```

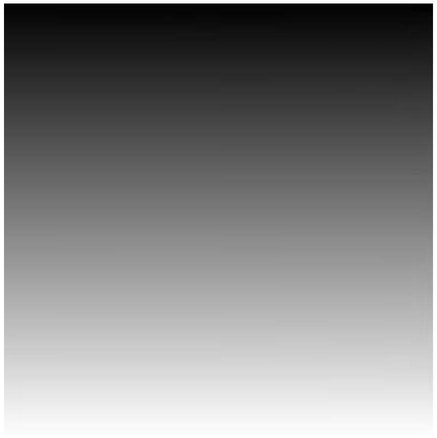


```
>> im1 = uint8(im1);    D
>> whos
  Name         Size                 Bytes   Class

  im1        256x256                65536   uint8
```

- **Difference between unsigned 8-bit integer (uint8) and double:** For data representation, we can either use uint8 or double. For uint8, this data type can store non-negative whole numbers within the range of **0** to **255**. It uses **8** bits (binary digits) to represent these values. For double (double-precision floating-point), this data type can represent a much wider range of numbers, including positive and negative numbers, very small values close to **0**, and very large numbers. It typically uses **64** bits for storage, providing much higher precision compared to uint8.
- **How to make a ramp Image (*revisited*)?** Let us make a ramp image, where we are going from black to white, shown by matrix **A**, and we are multiplying that by a vector of **1**'s, shown as the row matrix of **B**. We have very briefly discussed this before.
- In **C**, that is like saying we have this row vector, [**0:255**], we transpose it, " **'** ", and then we multiply it,"*****", by the row vector of **1**'s.
- If we do **whos** (read ***whose***), we will see that "**im1**" is **256** by **256**, and also it is **524,288** bytes (with double class). If we want "**im1**" to be smaller, we can use the command in **D**. This will use **uint8( )** for "**im1**" and the size will be reduced to **65,536** bytes. Now, we see that the same variable, "**im1**", would take up a lot less space.

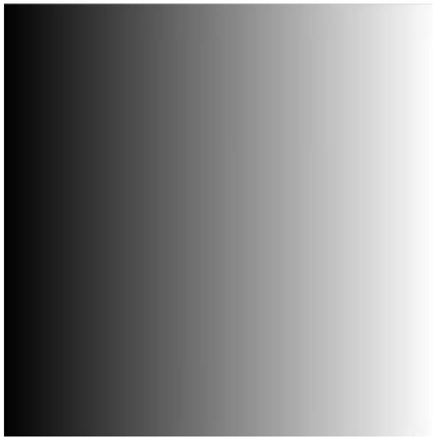
Useful MATLAB commands

```
>> imshow(im1)
```



A

```
>> im2 = im1';
>> imshow(im2)
```

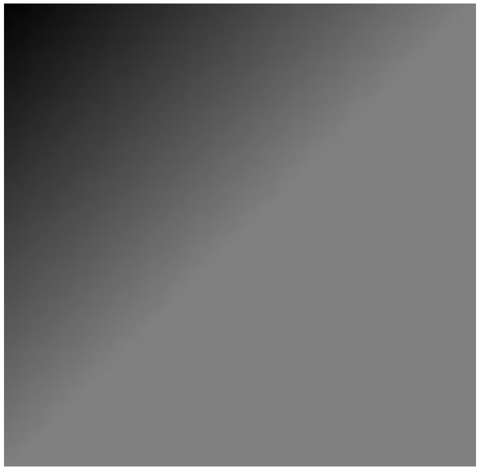


B

- If we were to look at that image in MATLAB, then we get this smooth ramp, as shown in A.
- Now, let us make another image that is just the sideways version of A. To do that, we just use **im2** = **im1'**, as shown in B.

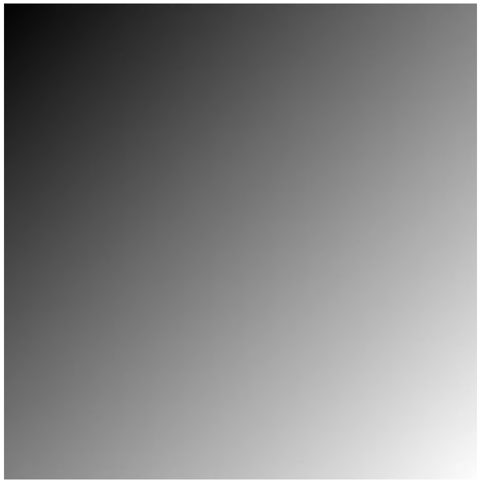
Useful MATLAB commands

```
>> av1 = (im1+im2)/2;
>> imshow(av1)
```



A

```
>> av2 = im1/2 + im2/2;
>> imshow(av2)
```

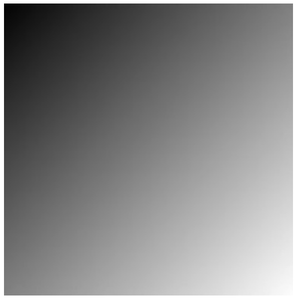


B

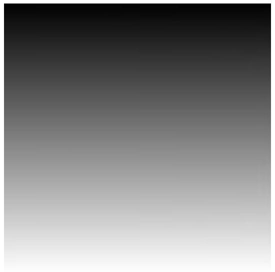
- **Averaging two images:** We can do things like averaging, where we might run into some trouble. Suppose we want to take the average of the previous two images, **im1** and **im2**, i.e., the original ramp and its sideways version.
- There are a few ways we could do it. And, many of these ways are not going to work for us! The way we might expect is just to look at **image 1** plus **image 2** over **2**, **av1 = (im1 + im2)/2**, as shown in **A**. Now, let us try **av2 = im1/2 + im2/2**, as shown in **B**. In theory, these two averaging should give us exactly the same thing. But, when we look at them side-by-side, they definitely look different! For example, if we look at **A** and focus our attention on the whole lower triangle, it seems that it is actually just a flat gray color, and that does not seem right. **Why do they look different?**

A

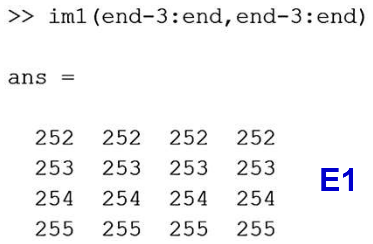
B



Useful MATLAB commands



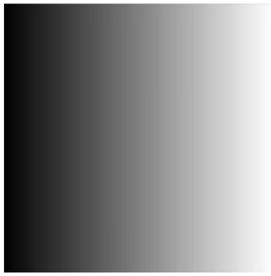
A



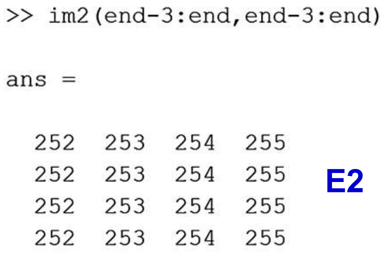
```
>> im1(end-3:end,end-3:end)

ans =

   252   252   252   252
   253   253   253   253
   254   254   254   254
   255   255   255   255
```

C1

E1



B



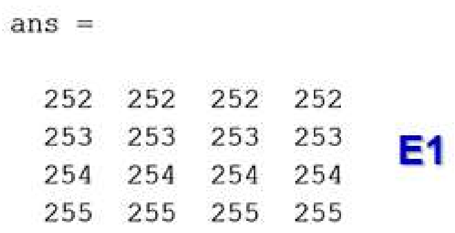
```
>> im2(end-3:end,end-3:end)

ans =

   252   253   254   255
   252   253   254   255
   252   253   254   255
   252   253   254   255
```

C2

E2

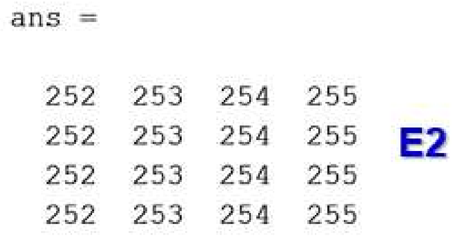
- The reason for this discrepancy is that we cannot do operations on **8**-bit integer in the same way that we would with a floating point variable. This is because any time we complete an operation on a uint8, it has to get turned back into a uint8, which means it has to be in the range from **0** to **255**. Anything that is below **0** is going to get turned into **0**, and anything that is above **0** is going to turn into **255**.
- Let us see what happened in **A**, which is **im1**. Suppose we are looking at the lower right-hand corner of the image. Using **C1**, we are asking MATLAB to show us the image for the **4x4** corner at the bottom of the image, shown by **E1**. These are all are pretty close to white (around **255**). The same thing is true for the other image in **B**, using **C2** gives us **E2**.
- This MATLAB code basically extracts a square sub-region of **4** rows and **4** columns from the bottom-right corner of the images **im1** and **im2**. Here, "**end**" refers to the last row index of the image. In a matrix, indexing starts from **1**, so the last row would have an index of the total number of rows in the image. "**-3**" indicates we are going **3** rows backward from the last row (**inclusive**). So, **end-3** refers to the third row from the end. The colon "**:**" specifies a range starting from **end-3** and ending at **end**. Therefore, this part extracts the last **4** rows of the image (from the third row from the end to the last row itself). "**,end-3:end**" defines the column range for the sub-region. Similar to the row range, it extracts the last **4** columns of the images (from the third column from the end to the last column).

Useful MATLAB commands



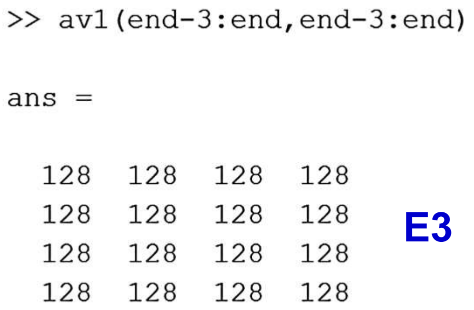
```
ans =

   252   252   252   252
   253   253   253   253
   254   254   254   254
   255   255   255   255
```

E1



```
ans =

   252   253   254   255
   252   253   254   255
   252   253   254   255
   252   253   254   255
```

E2

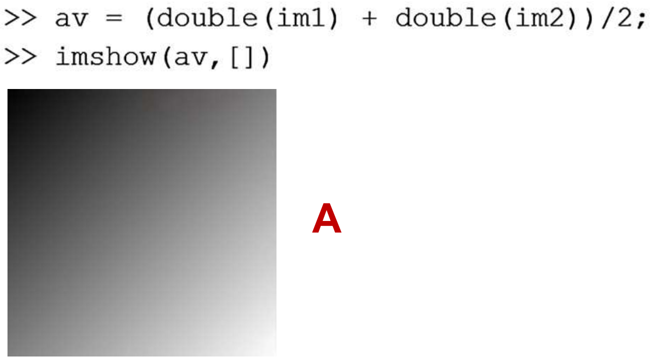


```
>> av1(end-3:end,end-3:end)

ans =

  128  128  128  128
  128  128  128  128
  128  128  128  128
  128  128  128  128
```

E3

- If we take the average of E1 and E2, using **av1 = (im1 + im2)/2**, first, it is immediately going to add those two numbers and it is going to get some bigger number than **255**. For example, for the top left corner of both matrices in E1 and E2, we get **252 + 252 = 504**. The software is going to first clip that to **255** and then divide it by **2** to give us roughly **128**, as seen in E3. We get the same pixel value for other entries. So, even though we know that there should be a series of numbers here, by virtue of the fact that everything has been done in this quantized world, we have got some problems.

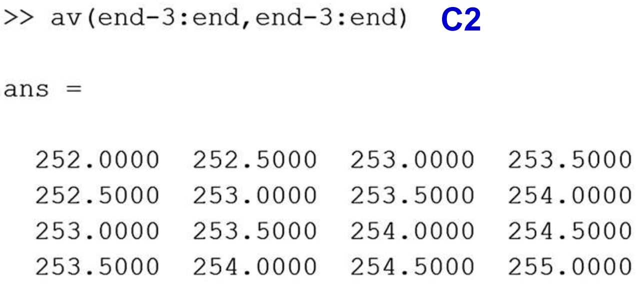
Useful MATLAB commands



```
>> av = (double(im1) + double(im2))/2;
>> imshow(av,[])
```

C1

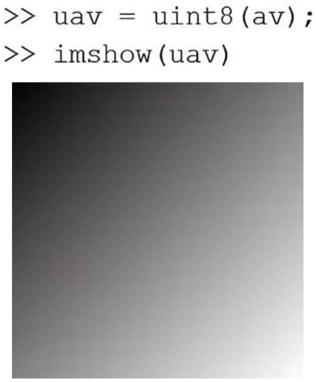
A



```
>> av(end-3:end,end-3:end)

ans =

  252.0000   252.5000   253.0000   253.5000
  252.5000   253.0000   253.5000   254.0000
  253.0000   253.5000   254.0000   254.5000
  253.5000   254.0000   254.5000   255.0000
```

C2

E2



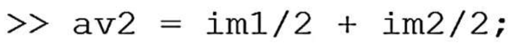
```
>> uav = uint8(av);
>> imshow(uav)
```

C3

B

- So, the safest thing to do in order to correct the problem that we just faced, which is also a little bit clumsy, would be to do something like as follows. As shown in C1, we say that the average is equal to (**double** (**im1**) + **double** (**im2**))/**2**. Now, this is going to be a double and will give us the output image that looks like our continuous ramp, shown in A.
- Now, if we were to look at the lower right-hand corner of the image, we would see that since this is a double, we are going to have **continuous values**, some of which are fractional, as shown in E2.
- If we wanted to, we could turn the image **av** back into a uint8, using C3. The output image will be B.

Useful MATLAB commands



```
>> av2 = im1/2 + im2/2;
```

- **Potential Problem with av2 = im1/2 + im2/2:** Here, **av2** is saying give us half the original image plus half the other image. The problem of this is that when we divide those integers individually by **2**, we are losing all our fractions. When we do that, we cannot have an image pixel that is, e.g., **67.5**. That means that all pixel values get rounded and we lose these little half pixel intensities.
- **Conclusion:** The moral of the story is that when we are doing image operations like combining images, we need to be careful with making sure that we are not losing information.

Useful MATLAB commands

- There are some built-in MATLAB commands that take care of this topic for us. In **Image Arithmetic** , we can find various built-in commands for dividing, combinations, multiplication, etc.

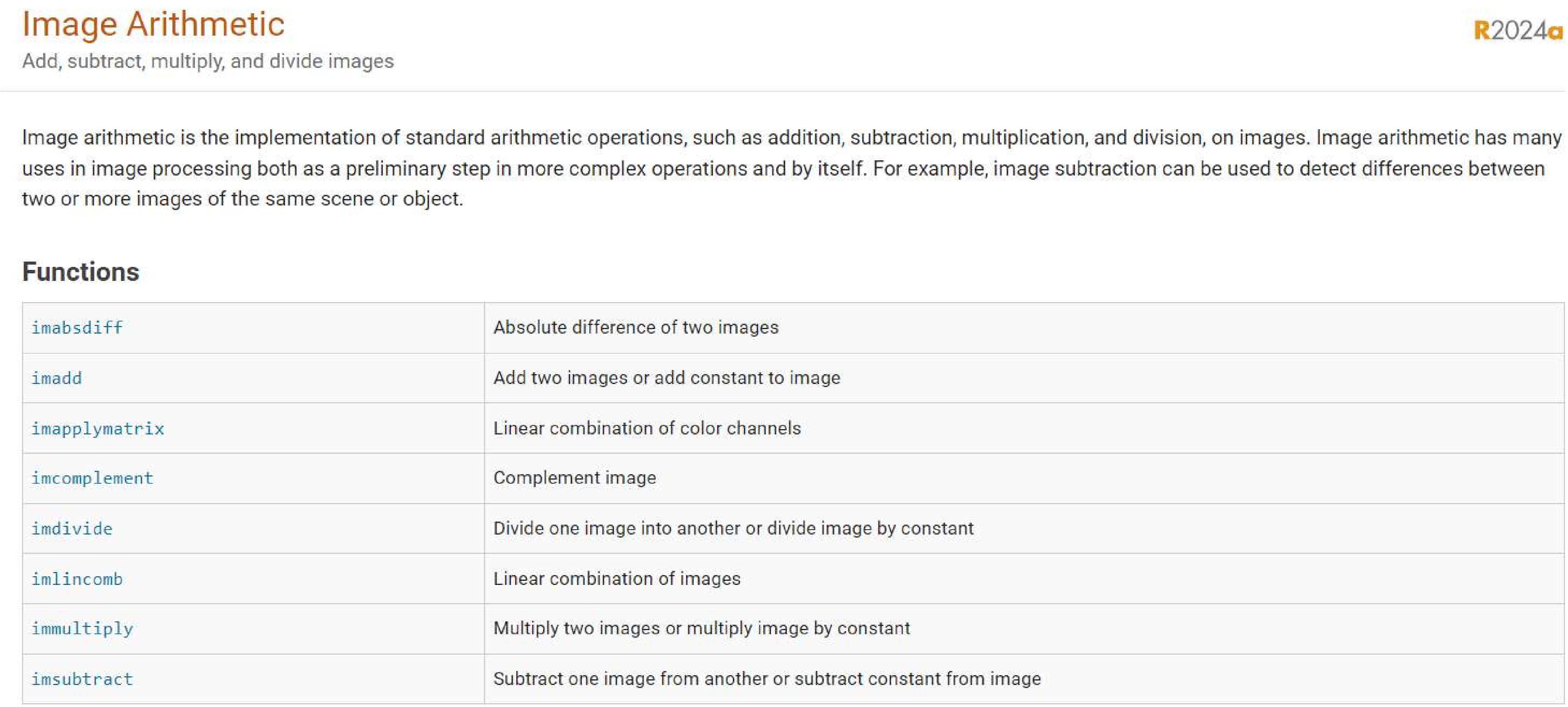


Image Arithmetic

R2024a

Add, subtract, multiply, and divide images

Image arithmetic is the implementation of standard arithmetic operations, such as addition, subtraction, multiplication, and division, on images. Image arithmetic has many uses in image processing both as a preliminary step in more complex operations and by itself. For example, image subtraction can be used to detect differences between two or more images of the same scene or object.

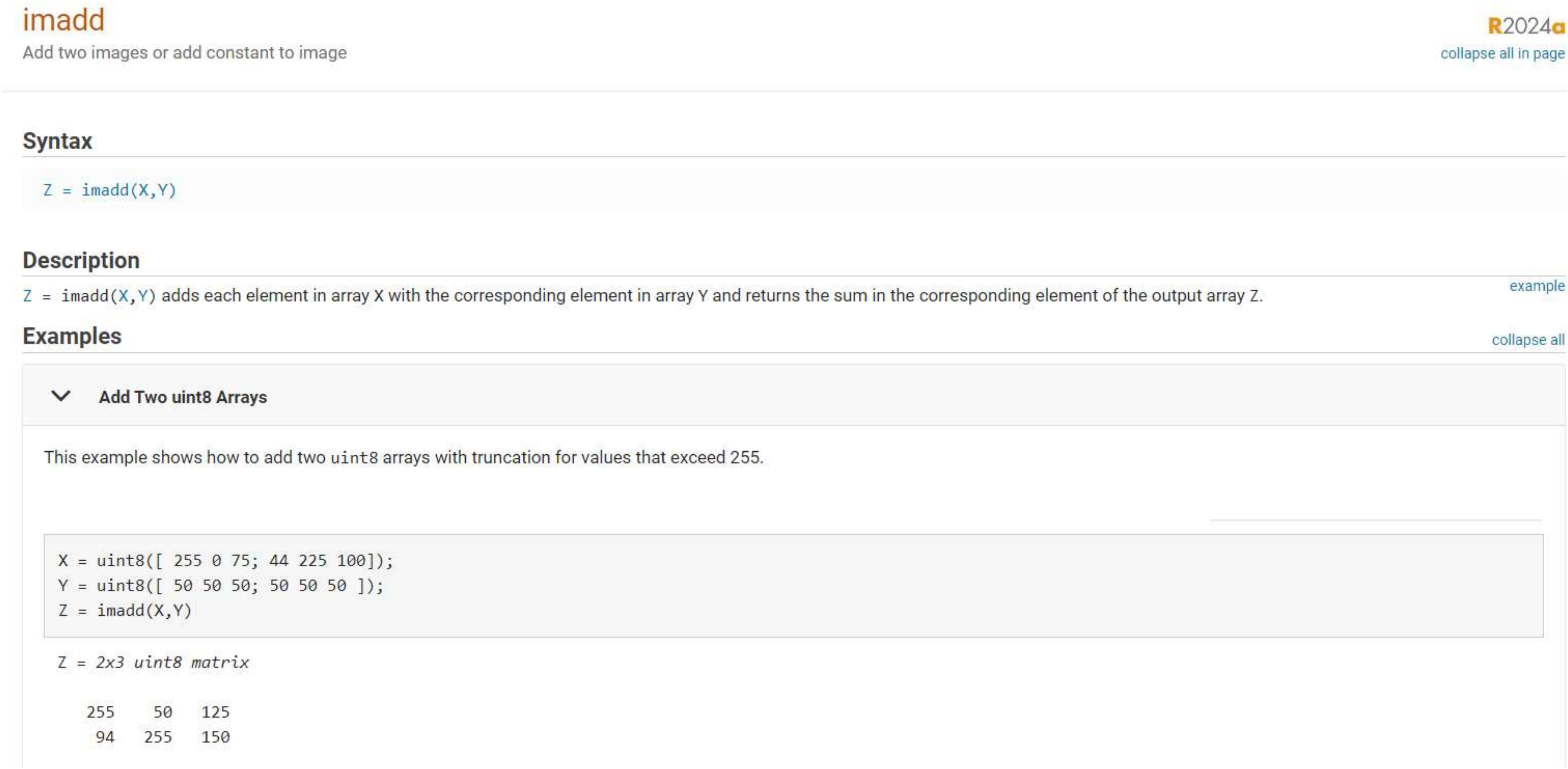
Functions

`imabsdiff`	Absolute difference of two images
`imadd`	Add two images or add constant to image
`imapplymatrix`	Linear combination of color channels
`imcomplement`	Complement image
`imdivide`	Divide one image into another or divide image by constant
`imlincomb`	Linear combination of images
`immultiply`	Multiply two images or multiply image by constant
`imsubtract`	Subtract one image from another or subtract constant from image

Useful MATLAB commands

- A useful built-in command is **imadd**.

```
>> doc imadd
```



imadd

Add two images or add constant to image

R2024a

collapse all in page

Syntax

```
Z = imadd(X,Y)
```

Description

`Z = imadd(X,Y)` adds each element in array X with the corresponding element in array Y and returns the sum in the corresponding element of the output array Z.

example

Examples

collapse all

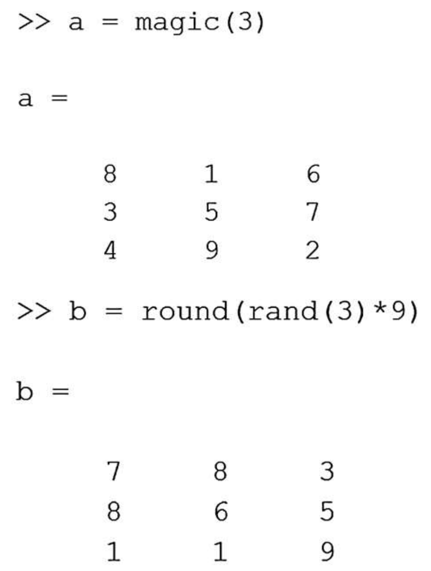
Add Two uint8 Arrays

This example shows how to add two `uint8` arrays with truncation for values that exceed 255.

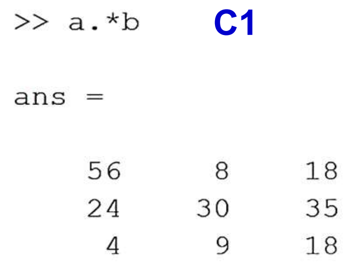
```
X = uint8([ 255 0 75; 44 225 100]);
Y = uint8([ 50 50 50; 50 50 50 ]);
Z = imadd(X,Y)
```

```
Z = 2x3 uint8 matrix

   255    50   125
    94   255   150
```

Useful MATLAB commands

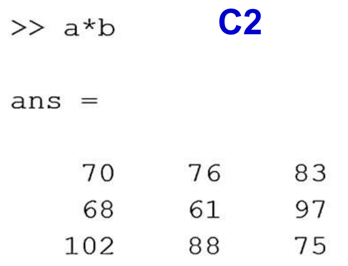


```
>> a = magic(3)

a =

     8     1     6
     3     5     7
     4     9     2

>> b = round(rand(3)*9)

b =

     7     8     3
     8     6     5
     1     1     9
```



C1

```
>> a.*b

ans =

    56     8    18
    24    30    35
     4     9    18
```

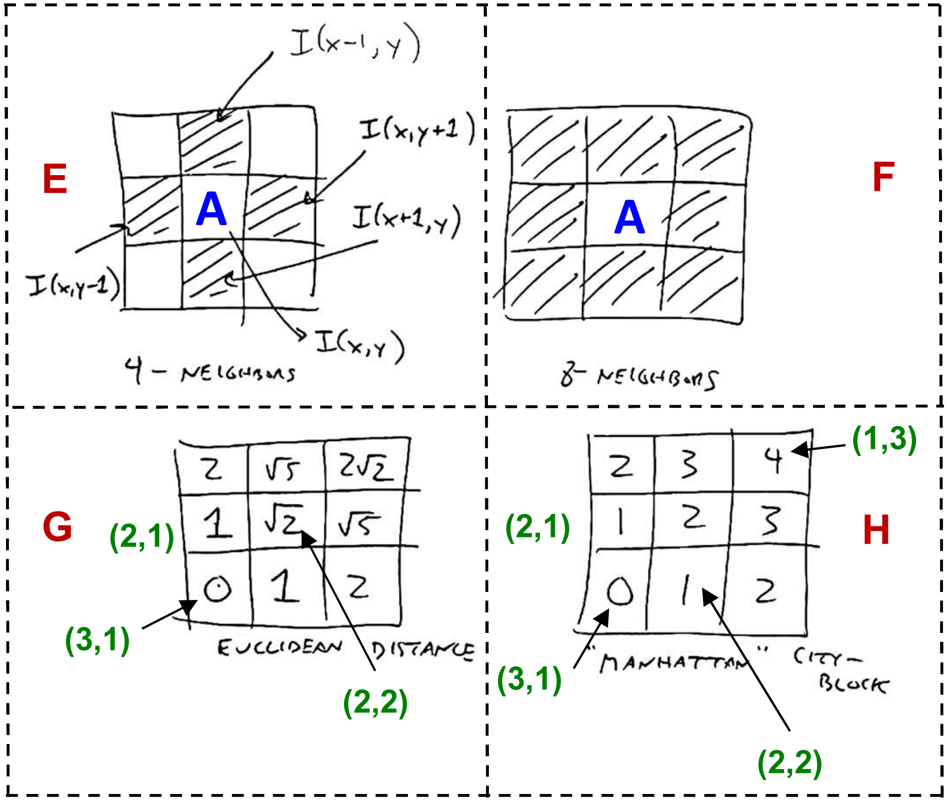


C2

```
>> a*b

ans =

    70    76    83
    68    61    97
   102    88    75
```

- Suppose that we have two **3** by **3** matrices. There is a difference between an **asterisk b** (or **times b**), ***b**, which gives us some **3x3** matrix, and a **dot asterisk b**, **.*b**. These give us two different results.
- The **.*b**, read ***dot times***, shown in **C1**, is basically saying **element by element multiply the entries together**. For example, here, what we want to get is **8** times **7**, which is **56**, **1** times **8**, which is **8**, **6** times **3**, which is **18**. That is often what we want in image processing.
- If we want to **multiply point by point two images together**, we need to use **asterisk b**, ***b**. This is shown in **C2**. This regular asterisk is going to be like regular MATLAB ***matrix operation***. This is the kind of operation that we usually need to do in a lot of other classes in computer and electrical engineering.

Pixel neighbors and distances



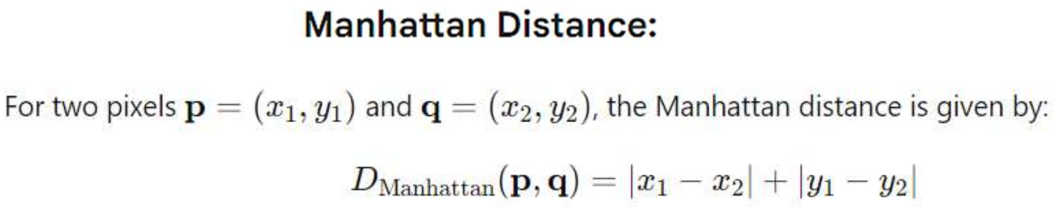
2	√5	2√2
1	√2	√5
0	1	2

2	3	4
1	2	3
0	1	2

- **Pixel Notation:** In the context of **pixel processing**, we sometimes use the notation shown in Diagram **E**. This is called the **4-neighbors of pixel A**. If pixel **A** is right in the middle, then the pixels that are directly adjacent to it are called the **4-neighbors**. These **4-neighbors** of **A** are shown by shaded area. In terms of notation, if pixel **A** is at an intensity **I(x,y)**, then our notation tells us that the pixel right above pixel **A** is **1** less **x** value, **I(x-1,y)**, and the one right below is going to be **1** more **x** value **I(x+1,y)**, etc. In Diagram **F**, we have **8-neighbors of pixel A**. That means that we look at the pixels on the off-diagonal as well.

- **Pixel Distances:** Another topic is related to **pixel distance measures**. The first type is called **Cartesian distance** or **Euclidean distance** (or **L2 norm**). Let us say we want to know how far pixel **(2,1)** in Diagram **G** is away from pixel **(3,1)**. Pixel **(2,1)** is **1** away in the vertical direction. Pixel **(2,2)** will be square root of **2** away, because it is on the diagonal, etc. The second type of pixel distance measure is called the **Manhattan** or **City-block distance** (or **L1 norm**). Sometimes, in a discrete world, we may also need to talk about how many **steps** it takes us to get to another pixel. For example, in Diagram **H**, we have to walk **1** block from pixel **(3,1)** to get to pixel **(2,1)**, or **4** blocks to get to pixel **(1,3)**.

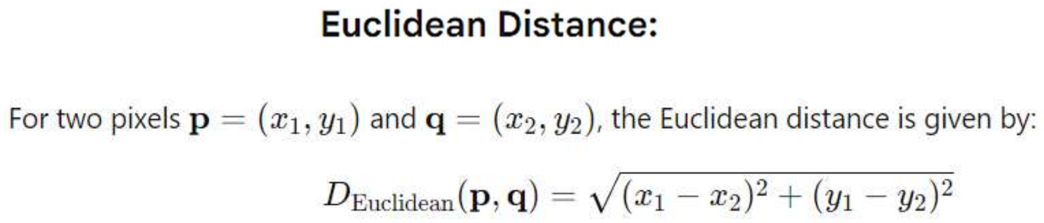
Pixel neighbors and distances



Manhattan Distance:

For two pixels $\mathbf{p} = (x_1, y_1)$ and $\mathbf{q} = (x_2, y_2)$, the Manhattan distance is given by:

$$D_{\text{Manhattan}}(\mathbf{p}, \mathbf{q}) = |x_1 - x_2| + |y_1 - y_2|$$



Euclidean Distance:

For two pixels $\mathbf{p} = (x_1, y_1)$ and $\mathbf{q} = (x_2, y_2)$, the Euclidean distance is given by:

$$D_{\text{Euclidean}}(\mathbf{p}, \mathbf{q}) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

- **General Formulas for Manhattan and Euclidean Distances:** The **Manhattan distance**, also called city-block distance or **L1 norm**, calculates the sum of the absolute values of the differences between corresponding coordinates in two vectors. The **Euclidean distance** or **L2 norm** calculates the straight-line distance between two points in **n**-dimensional space. It involves squaring the differences between corresponding coordinates, summing them, and then taking the square root.

End of Lecture 3