

# Images

Processing

# Images - PImage

An image is a collection of data (numbers) that indicate the amount of red, green and blue on each pixel.

PImage is a class in processing that loads and display loading and displaying an image as well as looking at its pixels.

```
PImage img = loadImage("my_pic.jpg"); // load image  
image(img,0,0); // display image at the top=let corner
```

Processing supports .gif, .jpg, .tga, and .png

The following methods will help us deal with individual colors

**set():** Changes the color of any pixel or writes an image directly into the display window.

**get():** Reads the color of any pixel or grabs a section of an image

For more info about PImage check the documentation [PImage](#)



# Kernel (image processing)

It is a small 2D matrix of numbers (generally 3 x 3)

It applies effects to an image (blurring, sharpening, outlining, embossing) like the ones you might find in Photoshop or Gimp,

```
{  
    {1, 1, 1},  
    {1, 1, 1},  
    {1, 1, 1}  
}
```



# Kernels and images - Convolution

If images are 2D and kernels are 2D, then we can use a kernel to modify an image, through a process called convolution.

**Convolution** is a process that adds each element of the image to its local neighbors, weighted by the kernel.



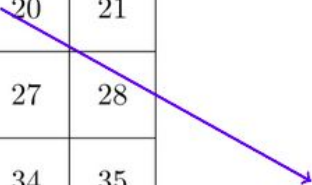
# Convolution Steps

For each pixel:

1. Enter the kernel over the pixel
2. Multiply the kernel values times the corresponding pixel values
3. Add the result - this final value is the new value of the current pixel.

1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30	31	32	33	34	35
36	37	38	39	40	41	42
43	44	45	46	47	48	49

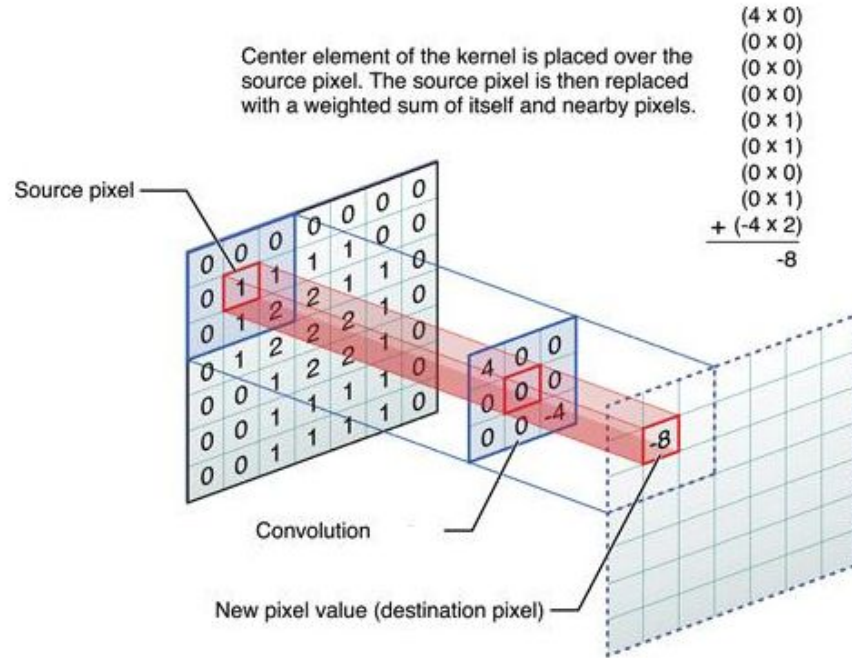
0.1	0.2	0.3
0.4	0.5	0.6
0.7	0.8	0.9


$$\begin{aligned} &= 0.1 \times 10 + 0.2 \times 11 + 0.3 \times 12 \\ &+ 0.4 \times 17 + 0.5 \times 18 + 0.6 \times 19 \\ &+ 0.7 \times 24 + 0.8 \times 25 + 0.9 \times 26 \\ &= 94.2 \end{aligned}$$

In this example, the new value for 18 would be 94.2

See some demos [here](#)

# Convolution example



Convolution Operation on a 7×7 matrix with a 3×3 kernel

Having a source image, we move the kernel over each pixel to calculate the new pixel value.

# Edge cases

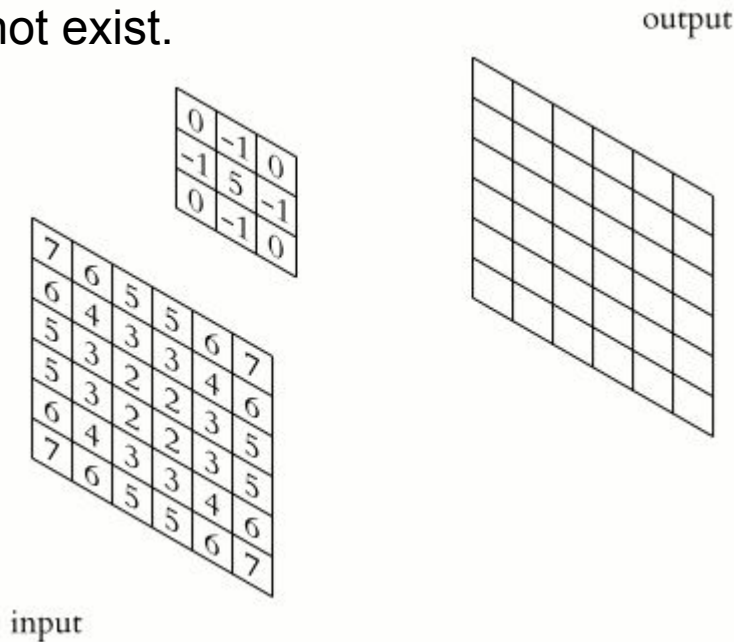
The convolution needs data from pixels that do not exist.

## Possible solutions

**Cropping:** Ignores the top/bottom/side pixels to avoid kernel going out of bounds. Simple solution.

**Duplicating:** Assumes the values off of the array, match the closest values on the array. See animation.

**Wrapping:** Takes any non-existent data from the opposite edge.



# Let's practice!!!

- Load an image
- Apply a kernel
  - Crop edges (required)
  - Duplicate edges (optional)
- Display original and modified images
- Upload your files to your repo in the folder processing/RedCar/





# You should know...

## Create colors


```
color firstColor = #FF66B2;  
color secondColor = color(255, 102, 178);
```

firstColor and secondColor are the same color so,

```
red(firstColor) == red(secondColor)
```

## Change colors

```
color myColor = color(200, 102, 178)  
int r = red(myColor);  
int g = green(myColor);  
int b = blue(myColor);  
color myOtherColor = color(r+1,g-1,b+2);
```



# You should know...

Processing provides an array of pixels for images.

We have the idea that pixels on the the screen have an X and Y coordinates.

But.... that is not true. The array has only one dimension. Colors are store in a linear sequence.

How the pixels look:

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19
20	21	22	23	24

How the pixels are stored:

0	1	2	3	4	5	6	7	8	9	.	.	.		
---	---	---	---	---	---	---	---	---	---	---	---	---	--	--

