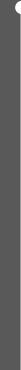


# **Mastering Computer Vision**

with **OpenCV** in **Python**

Learn all core basics, do almost coding 50 exercises & completing 12 fun mini projects!





**Computer Vision** is a cutting edge field of Computer Science that aims to enable computers to understand what is being seen in an image



# The Artificial Intelligence Dream

Machines that can see and understand the world around them!



*Source: Terminator 2: Judgement Day*

However, Computer Vision is a **Challenging!**

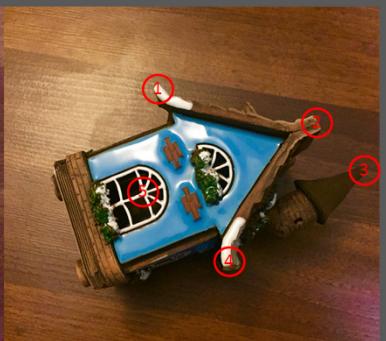
# Why is it so hard? So Many Reasons!

- Camera sensor & lens limitations



# Why is it so hard? So Many Reasons!

- Viewpoint variations



# Why is it so hard? So Many Reasons!

- Changing Lighting

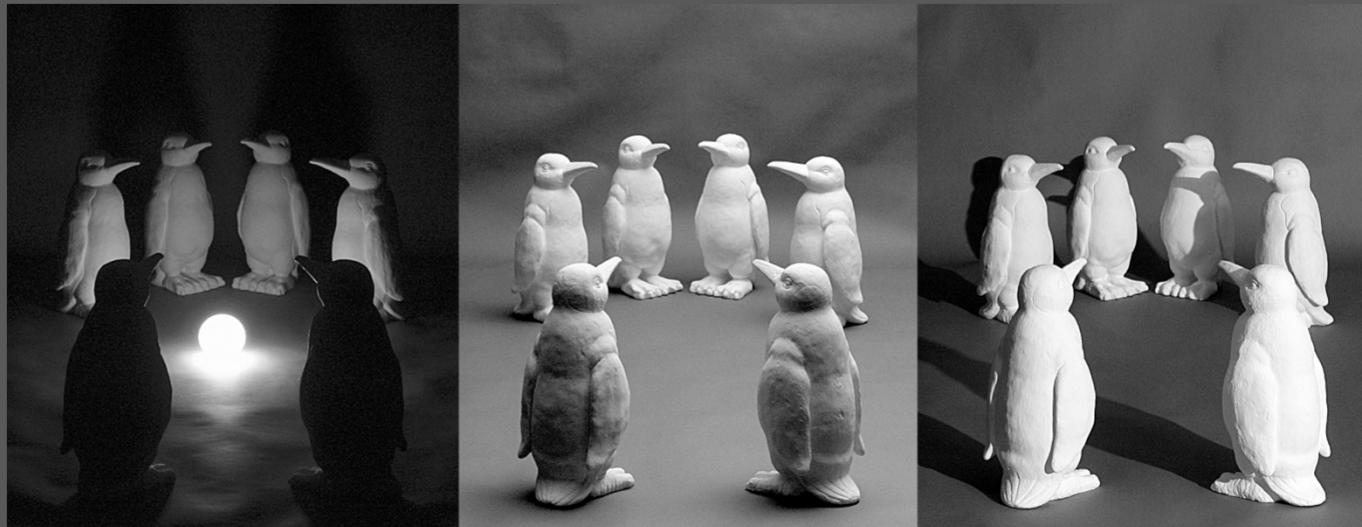


image credit: J. Koenderink

# Why is it so hard? So Many Reasons!

- Scaling



# Why is it so hard? So Many Reasons!

- Non-rigid deformations



image credit: Xu, Beihong 1943

# Why is it so hard? So Many Reasons!

- Occlusion

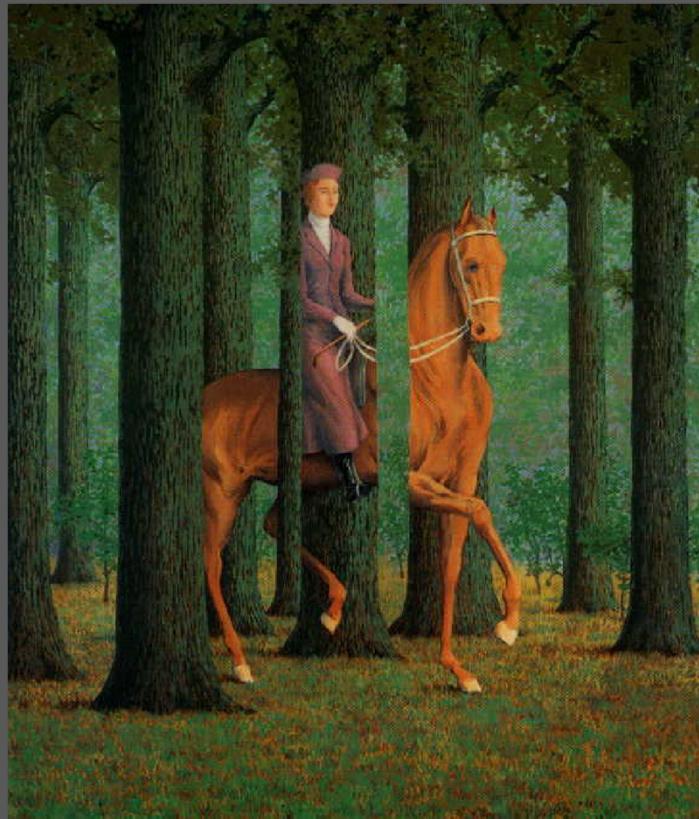


image credit: Magritte, 1957

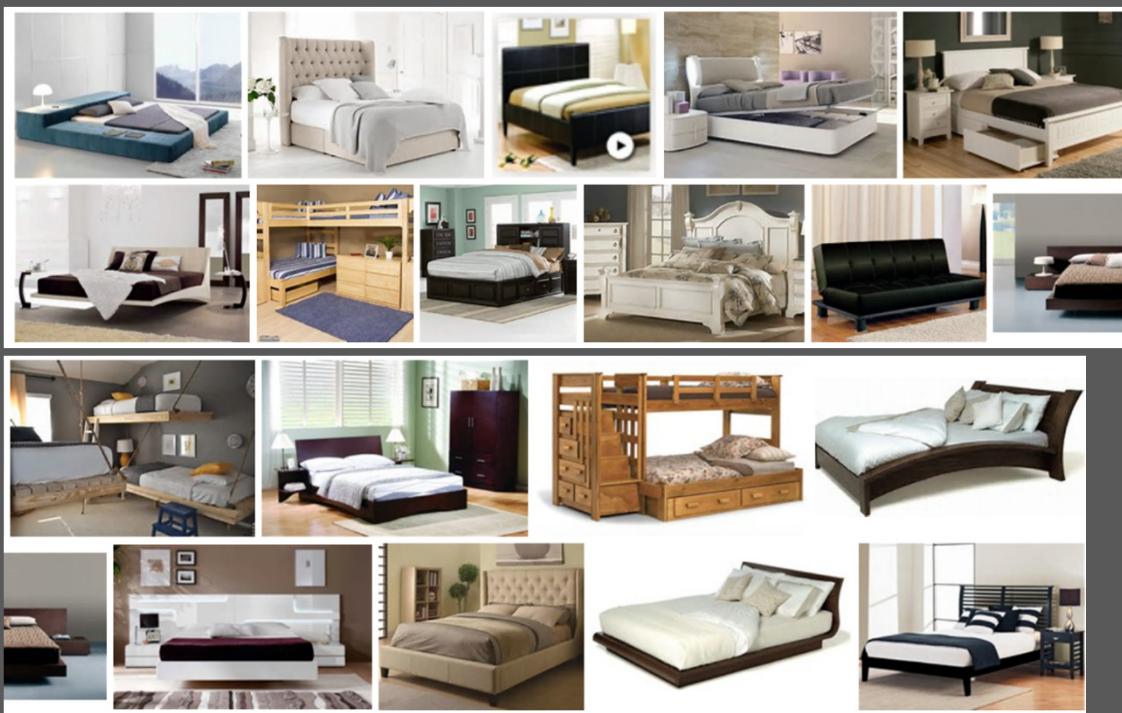
# Why is it so hard? So Many Reasons!

- Clutter



# Why is it so hard? So Many Reasons!

- Object class variations



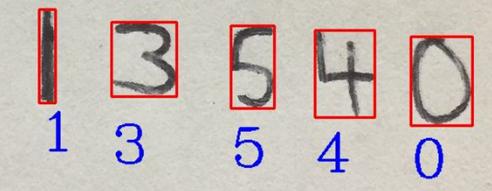
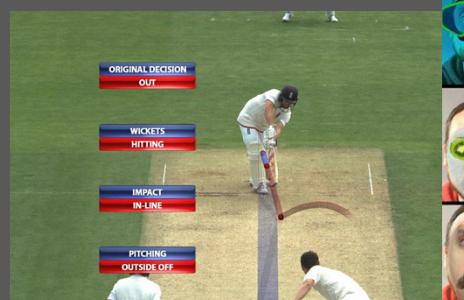
# Why is it so hard? So Many Reasons!

- Ambiguous Images/Optical Illusions



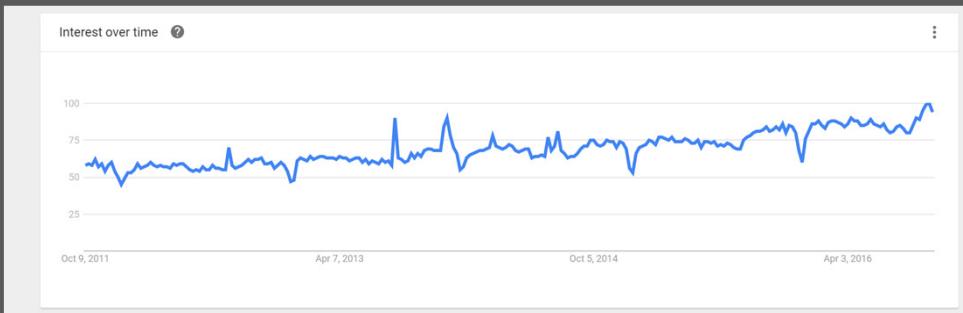
# Despite the **difficulty**, Computer Vision scientists have had many success stories!

- Robotic Navigation – Self Driving Cars
- Face Detection & Recognition
- Search Engine Image Search
- License Plate Reading
- Handwriting Recognition
- Snapchat & MSQRD Face Filters
- Object Recognition
- Ball & Player Tracking in Sports
- And many more!



# So why OpenCV in Python?

- Python is one of the **easiest** languages for beginners



- It is **extremely powerful** for data science and machine learning applications
- It stores images in **numpy arrays** which allows us to do some very powerful operations quite easily

# What You'll be Learning!

- Introduction to all key areas of Computer Vision Theory
  - Course Introduction & Setup
  - Basics of Computer Vision & OpenCV
  - Image Manipulations
  - Image Segmentation
  - Object Detection
  - Face, People & Car Detection
  - Face Analysis and Filters
  - Machine Learning in Computer Vision
  - Motion Analysis & Object Tracking
  - Bonus - Computational Photography
  - Course Summary
    - Become an expert!
    - Explore latest research
    - Startup Ideas
- Coding along with almost 50 exercises
- Develop 12 Fun Mini Projects!

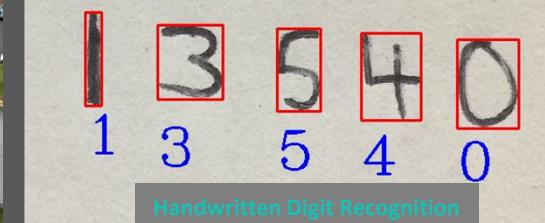
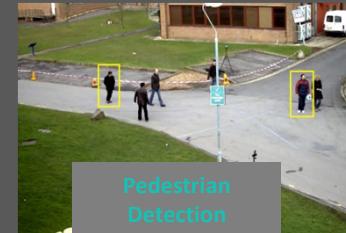
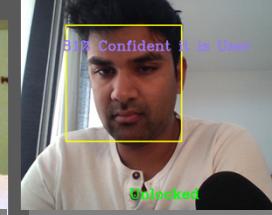
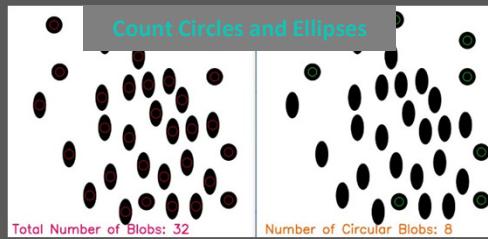
# How this course is taught?

- More than half your time will be spent **coding**
- **Overview** of the theory first then we **code**
- Almost all **code is explained line by line**

# **12 Amazing Mini Projects**

# What are these Mini Projects?

1. Making a Live Drawing Sketch of yourself
2. Shape Matching
3. Counting Circles and Ellipses
4. Finding Waldo
5. Object Identification
6. Face, Pedestrian & Car Detection
7. Live Face Swaps - Become Donald Trump or Kim Kardashian (just like MSQRD & Snapchat)
8. Face Reader - Detect and count yawns
9. Handwritten Digit Recognition
10. Facial Recognition
11. Ball Tracking
12. Photo Restoration



# Requirements?

- Basic programming is useful, but not needed I'll walk you through most of the code. Exposure to Numpy would be helpful.
- High School Level Math
- A Webcam
- Python 2.7 or 3.6
  - Anaconda Package is preferred but not required. I use Python 2.7, but all code works in 3.6
  - I use Python Notebooks which are excellent for teaching
- OpenCV 2.4.6 & or 3.4.X
  - Ideally we'd like to use the latest version OpenCV 3.4.6, however it is missing some important functions (SIFT, SURF etc.) that are included in 2.4.16. There are ways around it however, so don't despair.

# Who is this course for?

- **Beginners** interest in Computer Vision
- **Software developers & engineers** looking strengthen their skills for job promotion
- **College/University students** looking to get a head start in computer vision projects and research are ideal.
- **Startup founders** who wish to utilize computer vision
- **Hobbyist** who want to build a fun computer vision project. E.g. Raspberry Pi projects.

• . . .

**Let's begin our exciting journey into  
The world of Computer Vision using  
OpenCV in Python!**

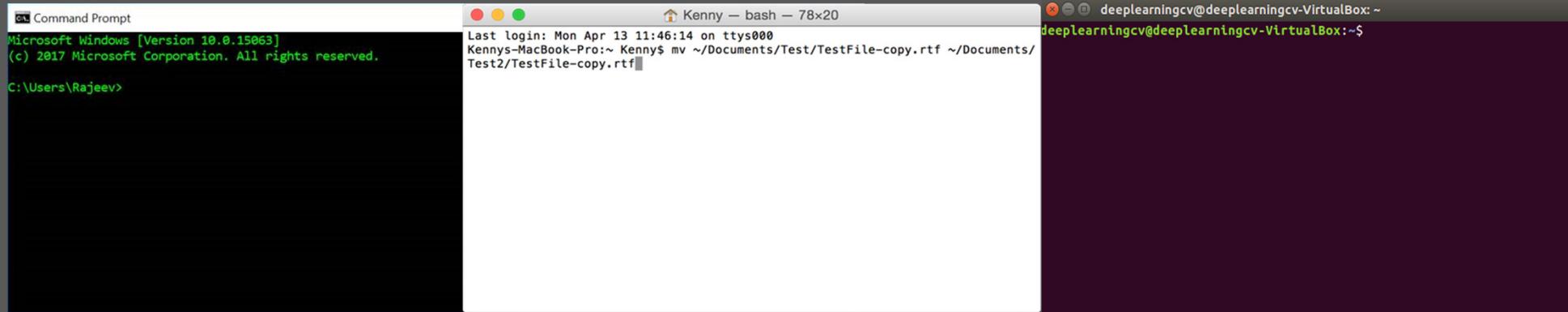
•

# Python & OpenCV Installation

## STEP 1 – Download & Install Anaconda Python Package

- Go to: <https://www.anaconda.com/download>
- Select appropriate version - either Python 2.7 or 3.7
- **To Test** – Go to windows command prompt and type:
  - jupyter notebook
  - This should launch the JuPyter Notebook server →

## STEP 2 – Open Command Prompt or Terminal (Mac & Linux)



# OpenCV & Dlib Installation

## STEP 3

- Use pip or pip3 to install the following packages:
  - pip install opencv-contrib-python
  - pip install dlib
- If not installed (should come with Anaconda by default, but if not using Anaconda you will need to install the following prior to the above)
  - pip install numpy
  - pip install matplotlib

## STEP 4

- Test by opening a new notebook and running the following lines (bottom right)

The image shows two screenshots of the Jupyter Notebook interface. The left screenshot shows the file browser with a new notebook named 'Python 3' selected. The right screenshot shows the notebook interface with the code cell 'In [1]: import cv2\nimport dlib' executed.

jupyter

Files Running Clusters

Select items to perform actions on them.

0 /

anacoda3  
bin  
ckpt  
cocoapi  
darkflow  
darknet  
DeepLearningCV

Name: Python 3

Notebook: Python 3

Other:

Text File  
Folder  
Terminal

3 months ago  
5 months ago  
14 days ago

jupyter Test

Logout

File Edit View Insert Cell Kernel Help Trusted Python 3

In [1]: import cv2  
import dlib

## Section 2

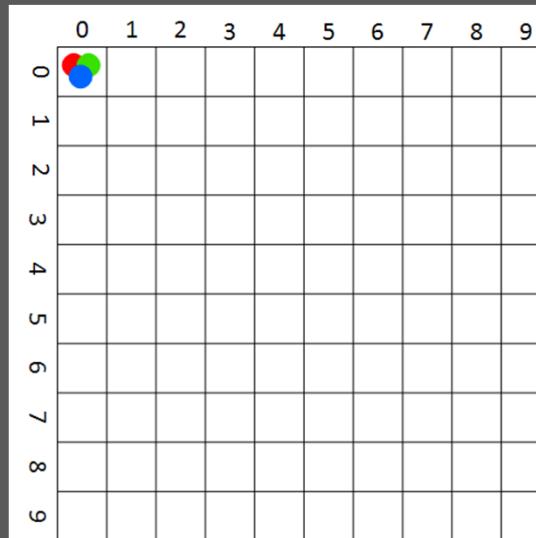
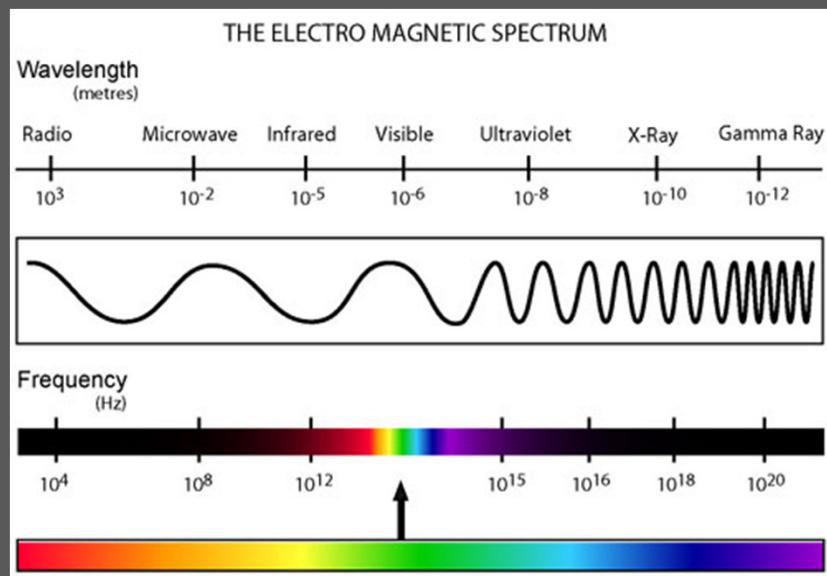
# Basics of Computer Vision & OpenCV

# Basics of Computer Vision & OpenCV

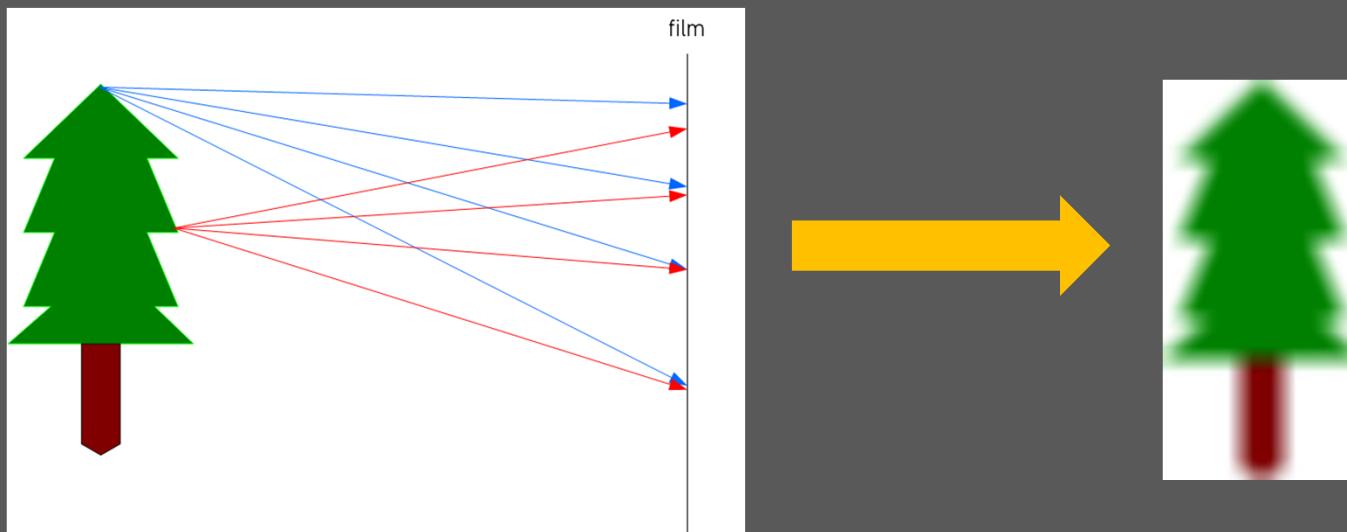
1. What are Images?
2. Image Formation
3. Storing images on computers
4. Getting Started with OpenCV: reading, writing and displaying images
5. Gray scaling
6. Color Spaces
7. Histograms
8. Drawing images

# What are Images?

- 2-Dimensional representation of the [visible light spectrum](#)

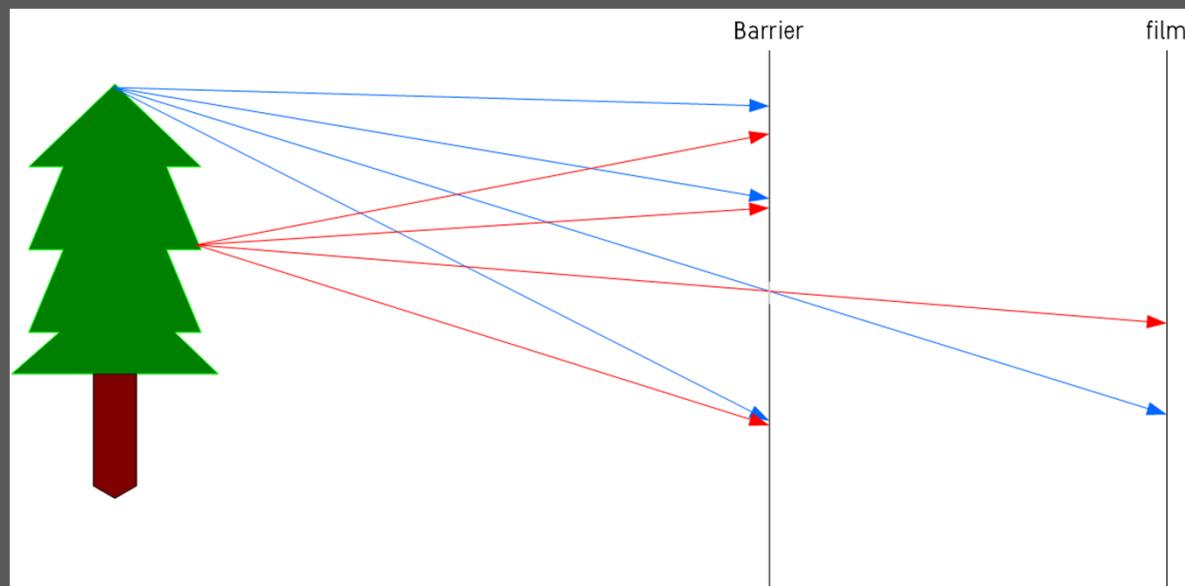


# How are images formed



- When light reflects off an object onto a film, sensor or retina
- In the above example the image here will be blurred

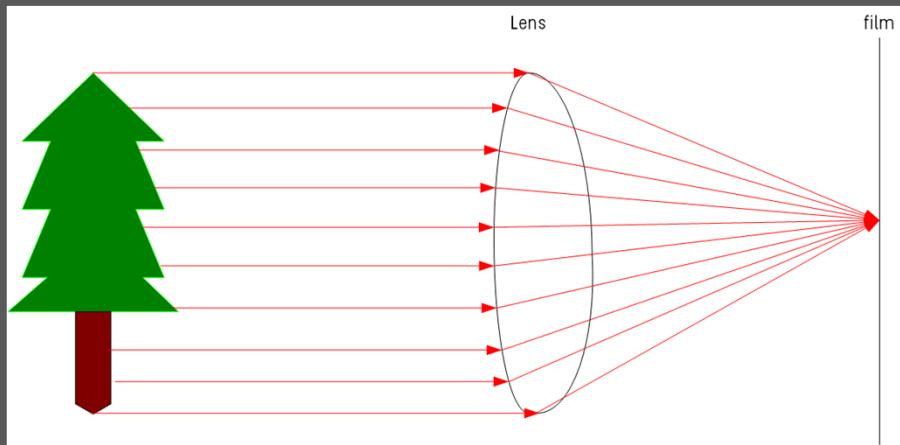
# Image Formation



- Using a small opening in the barrier (called aperture), we block off most of the rays of light reducing blurring on the film or sensor
- This is the pinhole camera model

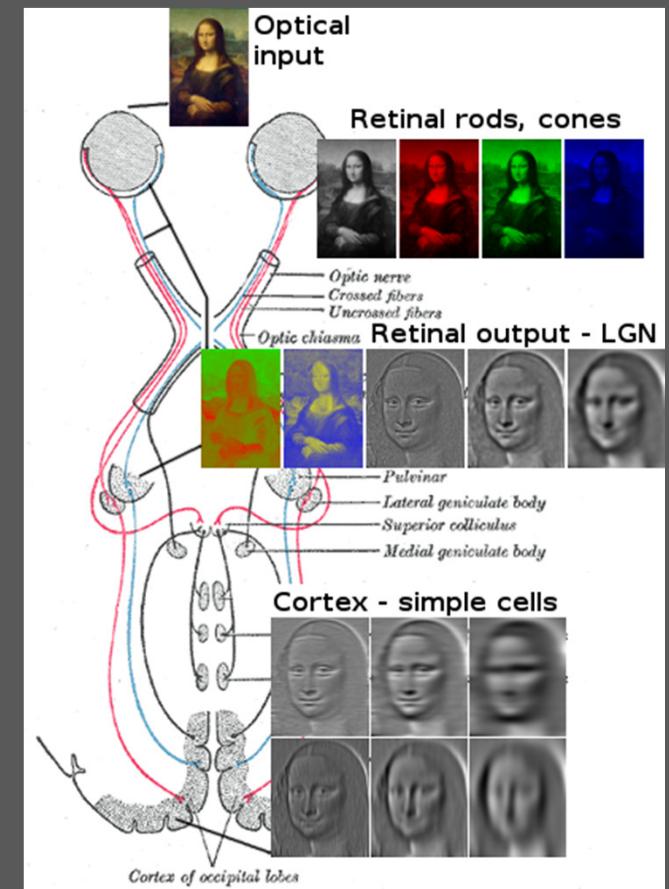
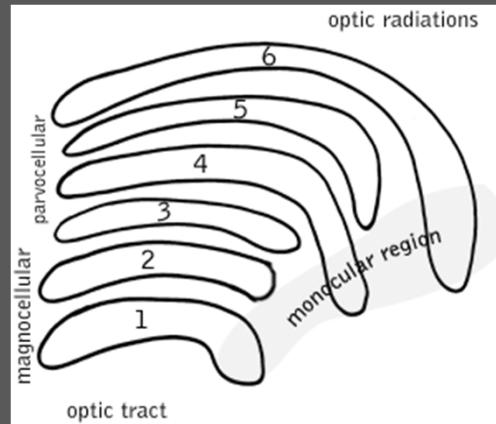
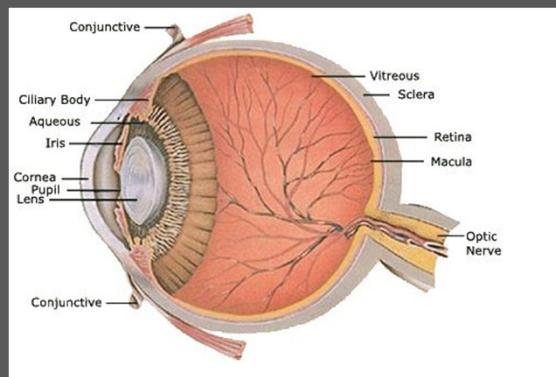
# Controlling Image Formation with a Lens

- Both our **eyes** and **cameras** use an **adaptive lens** to control many aspects of image formation such as:
  - **Aperture Size**
    - Controls the amount of light allowed through (f-stops in cameras)
    - Depth of Field (Bokeh)
  - **Lens width** - Adjusts focus distance (near or far)



# How Humans See

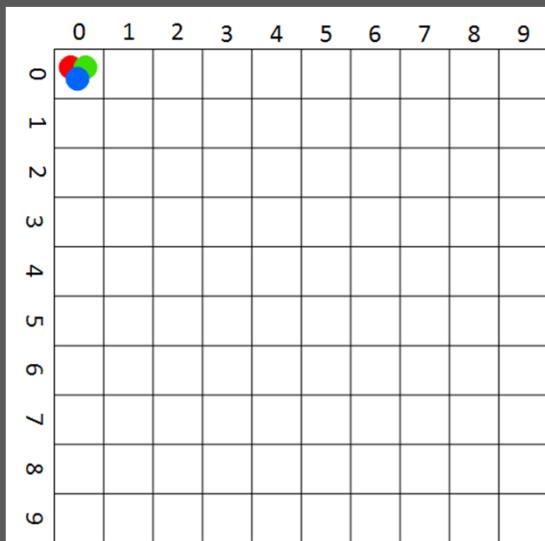
- The human visual system (eye & visual cortex) is incredibly good at image processing



Check out - [https://en.wikipedia.org/wiki/Visual\\_system](https://en.wikipedia.org/wiki/Visual_system)

# How do Computers store images?

- OpenCV uses **RGB** color space by default.

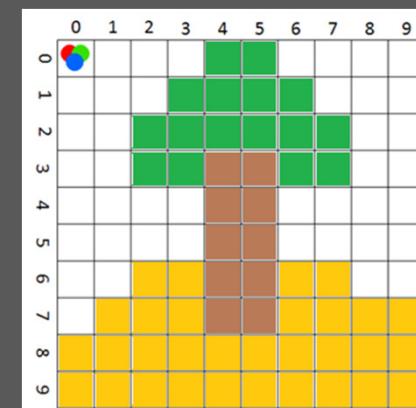


Each pixel coordinate  $(x, y)$  contains **3** values ranging for intensities of 0 to 255 (8-bit).

- Red
- Green
- Blue

Mixing different intensities of each color gives us the full color spectrum, example **Yellow**:

- Red – 255
- Green – 255
- Blue - 0





# Images are stored in multi-dimensional arrays

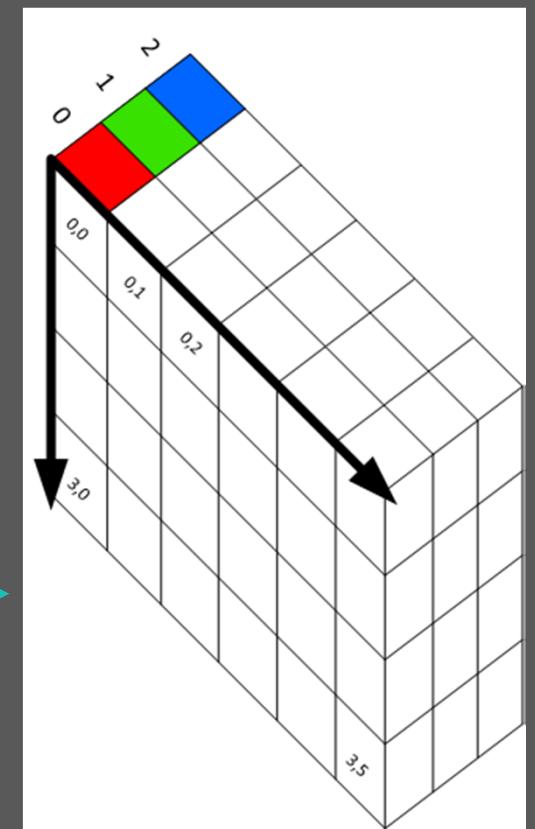
- Think of an array as table. A **1-dimensional** arrays looks like this:

0	1	2	3	4	5	6	7

- A **2-dimensional** array looks like this:

0	1	2	3	4	5	6	7

- **3-dimensional** array



# Black and White or Greyscale

- Black and White images are stored in 2-Dimensional arrays
  - There are two types of B&W images
    - **Grayscale** – Ranges of shades of grey
    - **Binary** – Pixels are either black or white

# What exactly is OpenCV

- Officially launched in 1999, OpenCV ([Open Source Computer Vision](#)) from an Intel initiative.
- OpenCV is written in C++.
- First major release 1.0 was in 2006, second in 2009 and third in 2015.
- OpenCV [2.4.13](#) is latest stable release for OpenCV 2.X.
- Open 3.X is very similar, and has some benefits and new functions, however, it also removed some of the important algorithms (due to patents) such as SIFT & SURF.
- Visit - [opencv.org](#) or [wikipedia.org/wiki/OpenCV](#) to learn more

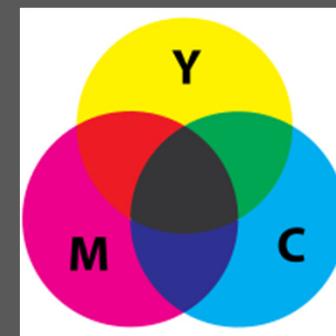
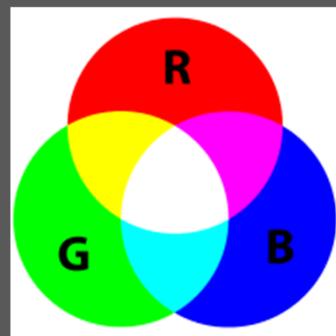
• . . .  
**Let's now get started with  
using OpenCV in Python.**



# Color Spaces

Ever heard of the terms **RGB**, **HSV** or **CMYK**?

These are color spaces, **which is simply a way to represent color.**



# RGB....wait BGR Color Space

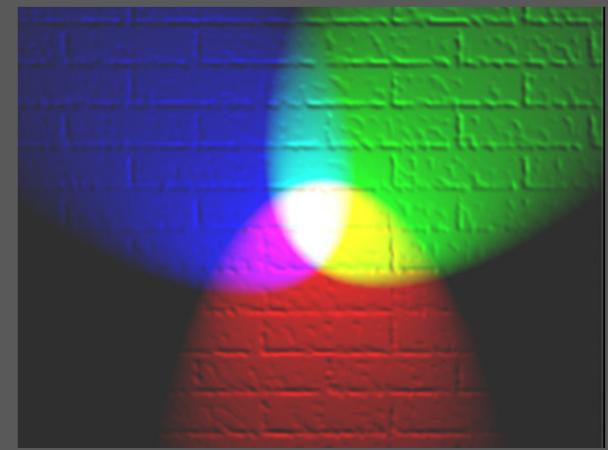
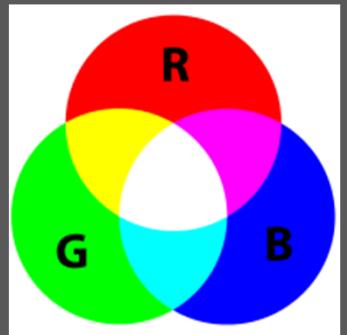
OpenCV's default color space is **RGB**.

RGB is an **additive color** model that generates colors by combining blue, green and red and different intensities/brightness. In OpenCV we 8-bit color depths

- Red
- Green
- Blue

However, OpenCV actually stores color in the **BGR format**.

**COOL FACT** - We use BGR order on computers due to how unsigned 32-bit integers are stored in memory, it still ends up being stored as RGB. The integer representing a color e.g. 0x00BBGGRR will be stored as 0xRRGGBB00.



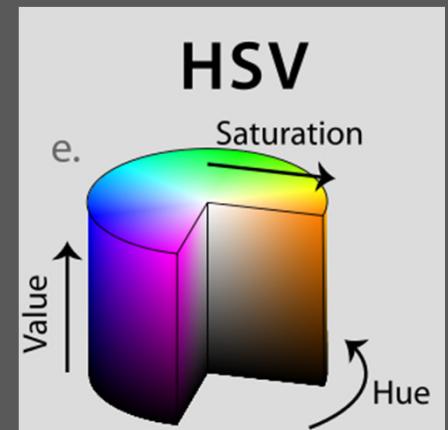
# HSV Color Space

**HSV** (Hue, Saturation & Value/Brightness) is a color space that attempts to represent colors the way humans perceive it. It stores color information in a cylindrical representation of RGB color points.

Hue – Color Value (0 – 179)

Saturation – Vibrancy of color (0-255)

Value – Brightness or intensity (0-255)



It's useful in computer vision for color segmentation. In RGB, filtering specific colors isn't easy, however, HSV makes it much easier to set color ranges to filter specific colors as we perceive them.

Visit these links to learn more:

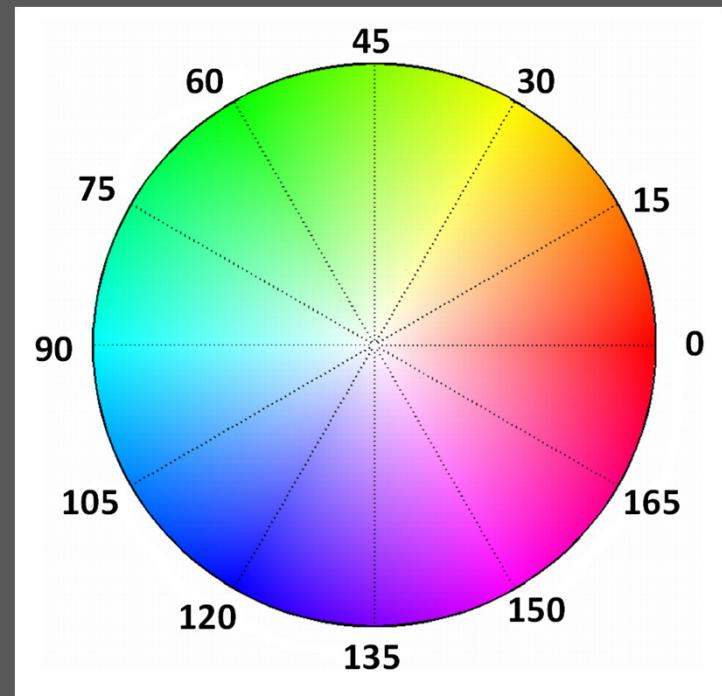
- [wikipedia.org/wiki/HSL\\_and\\_HSV](https://en.wikipedia.org/wiki/HSL_and_HSV)
- [http://coecsl.ece.illinois.edu/ge423/spring05/group8/FinalProject/HSV\\_writeup.pdf](http://coecsl.ece.illinois.edu/ge423/spring05/group8/FinalProject/HSV_writeup.pdf)

# Color Filtering

The Hue (Hue color range, goes from 0 to 180, not 360) and is mapped differently than standard

## Color Range Filters:

- Red – 165 to 15
- Green – 45 to 75
- Blue – 90 to 120



## Section 3

# Image Manipulations

# Image Manipulations

1. Transformations, affine and non affine
2. Translations
3. Rotations
4. Scaling, re-sizing and interpolations
5. Image Pyramids
6. Cropping
7. Arithmetic Operations
8. Bitwise Operations and Masking
9. Convolutions & Blurring
10. Sharpening
11. Thresholding and Binarization
12. Dilation, erosion, opening and closing
13. Edge Detection & Image Gradients
14. Perspective & Affine Transforms
15. **Mini Project # 1 – Make a Live Sketch of Yourself!**

# Transformations

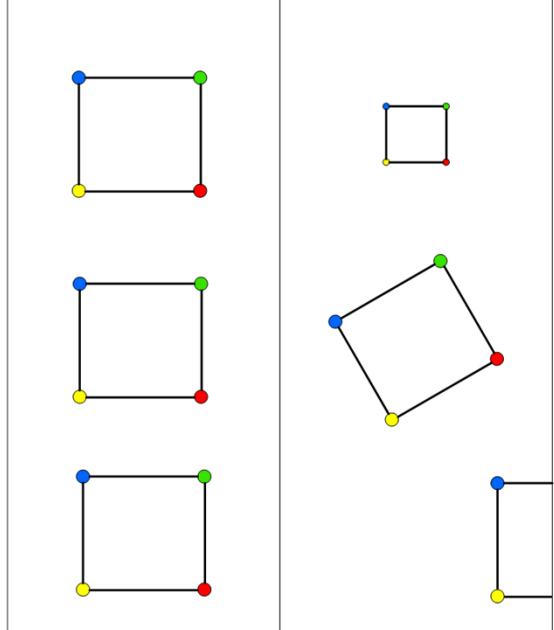
Transformations – are geometric distortions enacted upon an image.

We use transformations to correct distortions or perspective issues from arising from the point of view an image was captured.

Types:

- Affine
- Non-Affine

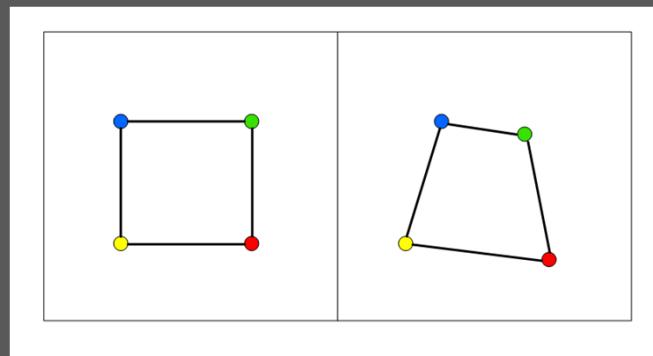
# Affine vs Non Affine Theory



Scaling

Rotation

Translation



Non-Affine or  
Projective Transform and  
also called Homography

The non-affine or projective transformation does not preserve parallelism, length, and angle. It does however preserve collinearity and incidence.

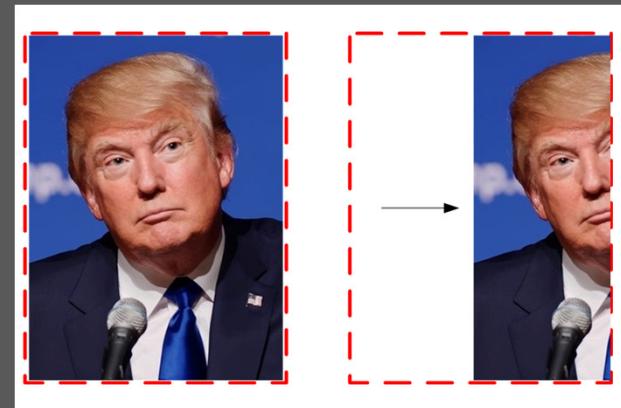
# Translations

Translation Matrix

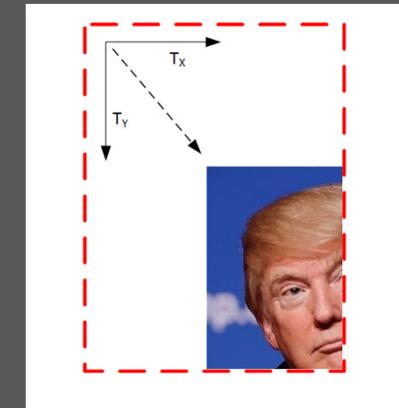
$$T = \begin{bmatrix} 1 & 0 & T_x \\ 0 & 1 & T_y \end{bmatrix}$$

$T_x$  - Represents the shift along the x-axis (horizontal)

$T_y$  - Represents the shift along the y-axis (vertical)



We use the OpenCV function cv2.warpAffine to implement these translations

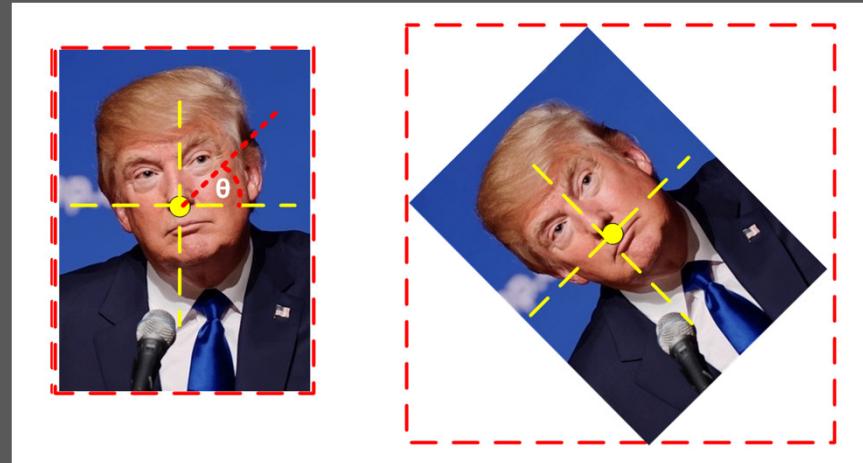


# Rotations

Rotation Matrix

$$M = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix}$$

$\theta$  – the angle of rotation



OpenCV allows you to scale and rotate at the same time using the function:

```
cv2.getRotationMatrix2D(rotation_center_x, rotation_center_y, angle of rotation, scale)
```

# Re-sizing, Scaling and Interpolation

What is interpolation?

**Interpolation** is a method of constructing new data points within the range of a discrete set of known data points

`cv2.INTER_AREA` - Good for shrinking or down sampling

`cv2.INTER_NEAREST` - Fastest

`cv2.INTER_LINEAR` - Good for zooming or up sampling (default)

`cv2.INTER_CUBIC` - Better

`cv2.INTER_LANCZOS4` - Best

Good comparison of interpolation methods:

- <http://tanbakuchi.com/posts/comparison-of-opencv-interpolation-algorithms/>



# Image Pyramids

Pyramiding image refers to either upscaling (enlarging) and downscaling (shrinking images).

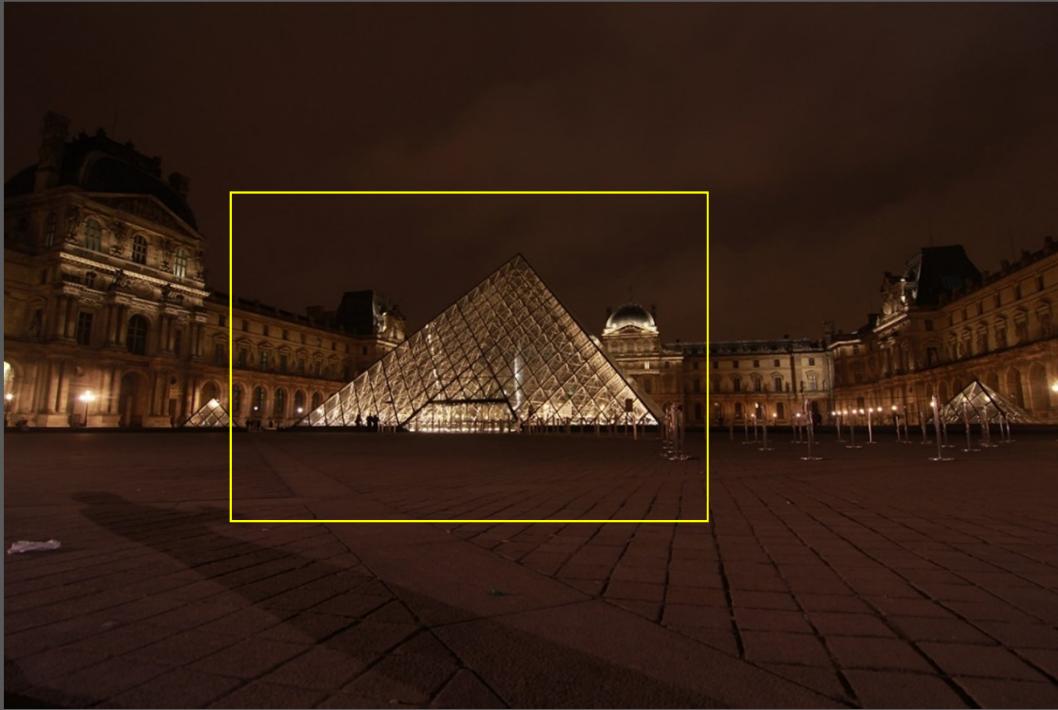
It's simply a different way of re-sizing that allows us to easily and quickly scale images. Scaling down reduces the height and width of the new image by half.

This comes in useful when making object detectors that scales images each time it looks for an object.



# Cropping Images

Cropping images refers to extracting a segment a of that image.

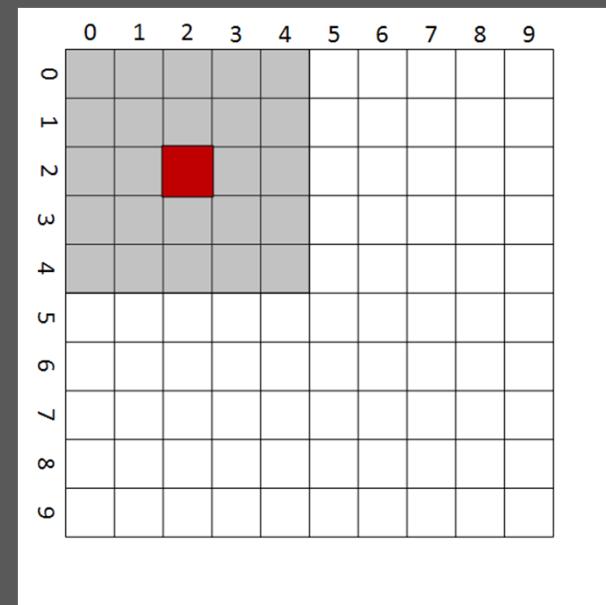


# Convolutions & Blurring

A **Convolution** is a mathematical operation performed on two functions producing a third function which is typically a modified version of one of the original functions.

$$\text{Output Image} = \text{Image} \otimes \text{Function}_{\text{Kernel Size}}$$

In Computer Vision we use **kernel's** to specify the size over which we run our manipulating function over our image.



5 x 5 Kernel over our image

# Blurring

Blurring is an operation where we average the pixels within a region (kernel).

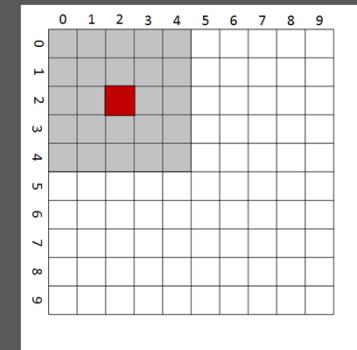
$$Kernel = \frac{1}{25} \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$



The above is a 5 x 5 kernel.

We multiply by 1/25 to normalize i.e. sum to 1, otherwise we'd be increasing intensity.

```
cv2.filter2D(image, -1, kernel)
```

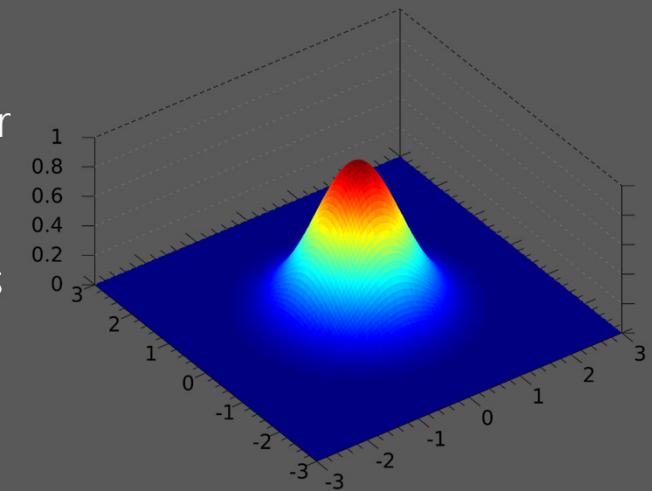


5 x 5 Kernel over our image

# Other Types of Blurring

- `cv2.blur` – Averages values over a specified window
- `cv2.GaussianBlur` – Similar, but uses a Gaussian window (more emphasis or weighting on points around the center)
- `cv2.medianBlur` – Uses median of all elements in the window
- `cv2.bilateralFilter` – Blur while keeping edges sharp (slower). It also takes a Gaussian filter in space, but one more Gaussian filter which is a function of pixel difference. The pixel difference function makes sure only those pixels with similar intensity to central pixel is considered for blurring. So it preserves the edges since pixels at edges will have large intensity variation.

$\frac{1}{273}$	1	4	7	4	1
	4	16	26	16	4
	7	26	41	26	7
	4	16	26	16	4
	1	4	7	4	1



# Sharpening

Sharpening is the opposite of blurring, it strengthens or emphasizes edges in an image

$$\text{Kernel}_\circ = \begin{bmatrix} -1 & -1 & -1 \\ -1 & 9 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

Our kernel matrix sums to one, so there is no need to normalize (i.e. multiply by a factor to retain the same brightness of the original)



# Thresholding, Binarization & Adaptive Thresholding

Thresholding is act of converting an image to a binary form.

```
cv2.threshold(image, Threshold Value, Max Value, Threshold Type)
```

Threshold Types:

- cv2.THRESH\_BINARY – Most common
- cv2.THRESH\_BINARY\_INV – Most common
- cv2.THRESH\_TRUNC
- cv2.THRESH\_TOZERO
- cv2.THRESH\_TOZERO\_INV



**NOTE:** Image need to be converted to greyscale before thresholding.

# Adaptive Thresholding

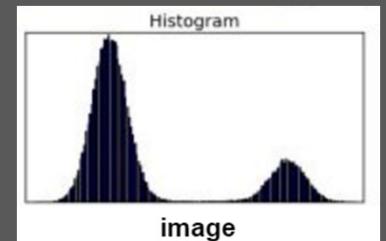
- Simple threshold methods require us to provide the threshold value.
- Adaptive threshold methods take that uncertainty away

`cv2.adaptiveThreshold(image, Max Value, Adaptive type, Threshold Type, Block size, Constant that is subtracted from mean)`

**NOTE:** Block sizes need to be odd numbers!

## Adaptive Threshold Types:

- `ADAPTIVE_THRESH_MEAN_C` – based on mean of the neighborhood of pixels
- `ADAPTIVE_THRESH_GAUSSIAN_C` – weighted sum of neighborhood pixels under the Gaussian window
- `THRESH_OTSU` ([uses cv2.threshold function](#)) – Clever algorithm assumes there are two peaks in the gray scale histogram of the image and then tries to find an optimal value to separate these two peaks to find T.



# Dilation and Erosion

These are operations in the field of mathematical morphology:

**Dilation** – Adds pixels to the boundaries of objects in an image

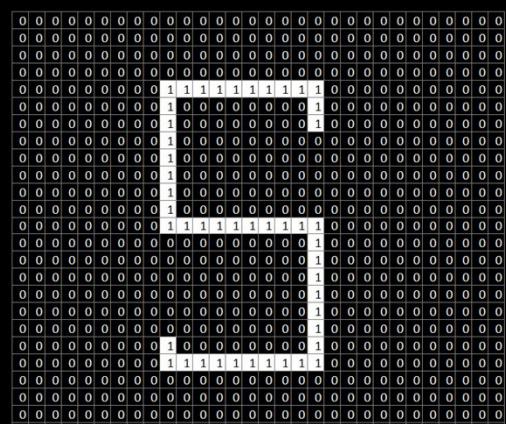
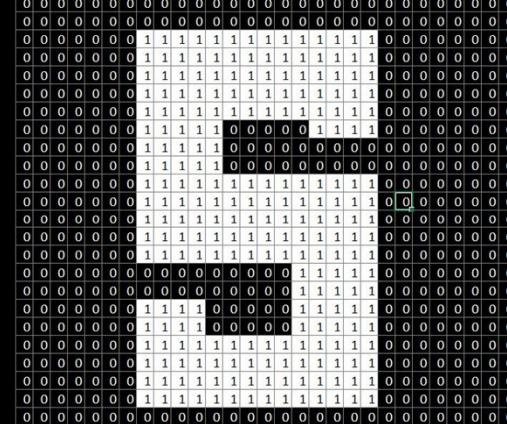
**Erosion** – Removes pixels at the boundaries of objects in an image

**Opening** - Erosion followed by dilation

**Closing** - Dilation followed by erosion

Erosion

Dilation

Erosion	Dilation
	

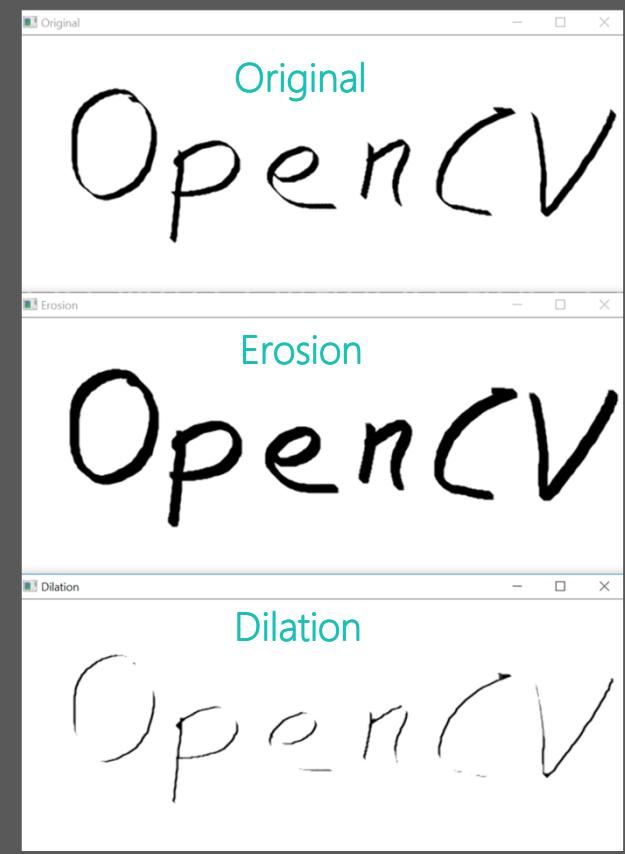
# Confusion with Dilation and Erosion

Common StackOverflow question:

*"Why is dilation and erosion doing the reverse of what I expect?"*

Remember:

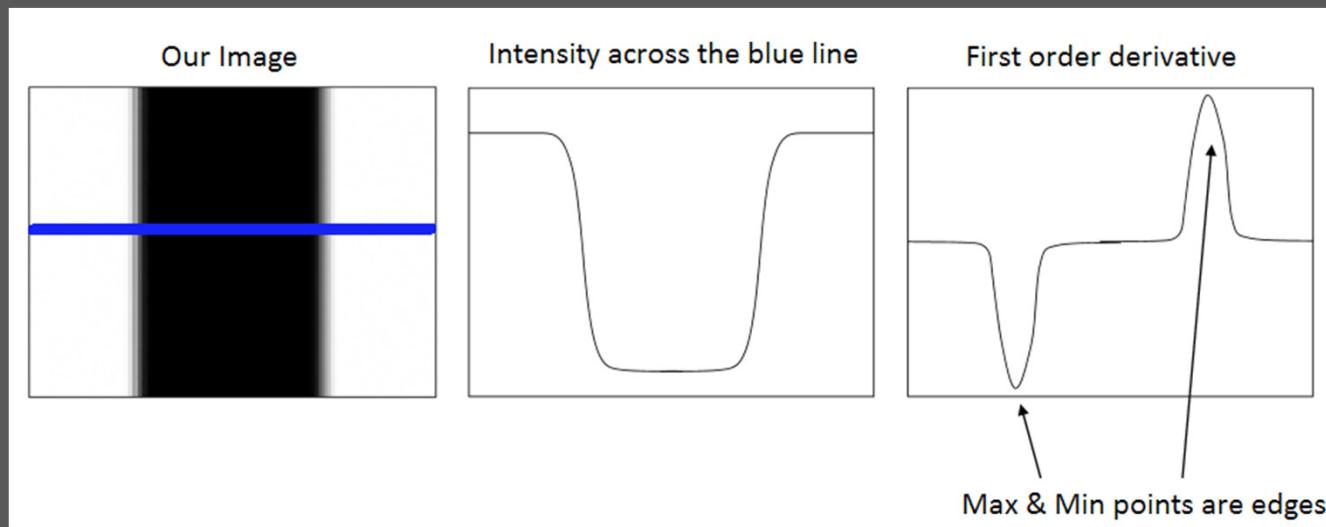
- **Dilation** – Adds pixels to the boundaries of objects in an image
- **Erosion** – Removes pixels at the boundaries of objects in an image



# Edge Detection & Image Gradients

Edge Detection is a very important area in Computer Vision, especially when dealing with contours (you'll learn this later soon).

Edges can be defined as sudden changes (discontinuities) in an image and they can encode just as much information as pixels.



# Edge Detection Algorithms

There are three main types of Edge Detection:

- **Sobel** – to emphasize vertical or horizontal edges
- **Laplacian** – Gets all orientations
- **Canny** – Optimal due to low error rate, well defined edges and accurate detection.

**Canny Edge Detection Algorithm** (developed by John F. Canny in 1986)

1. Applies Gaussian blurring
2. Finds intensity gradient of the image
3. Applied non-maximum suppression (i.e. removes pixels that are not edges)
4. Hysteresis – Applies thresholds (i.e. if pixel is within the upper and lower thresholds, it is considered an edge)

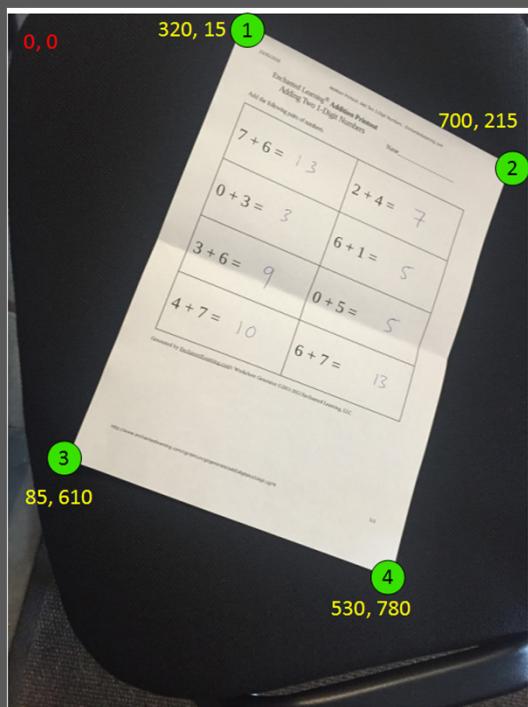


Learn more here – [https://en.wikipedia.org/wiki/Canny\\_edge\\_detector](https://en.wikipedia.org/wiki/Canny_edge_detector)

Comparison of Edge Detection Techniques - <http://www.ijcsi.org/papers/IJCSI-9-5-1-269-276.pdf>

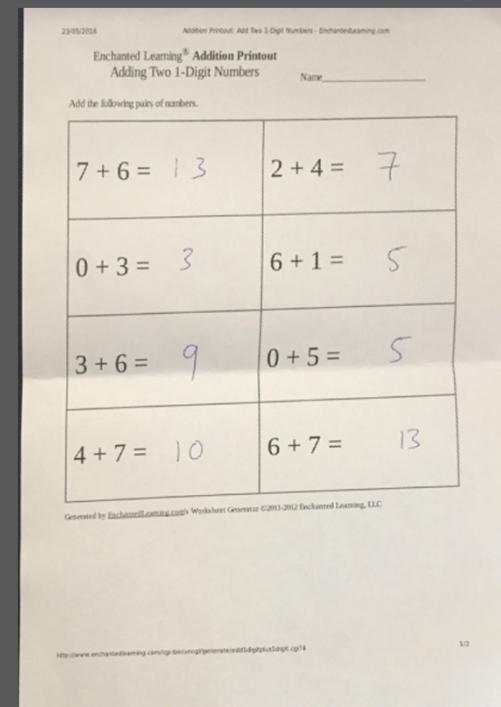
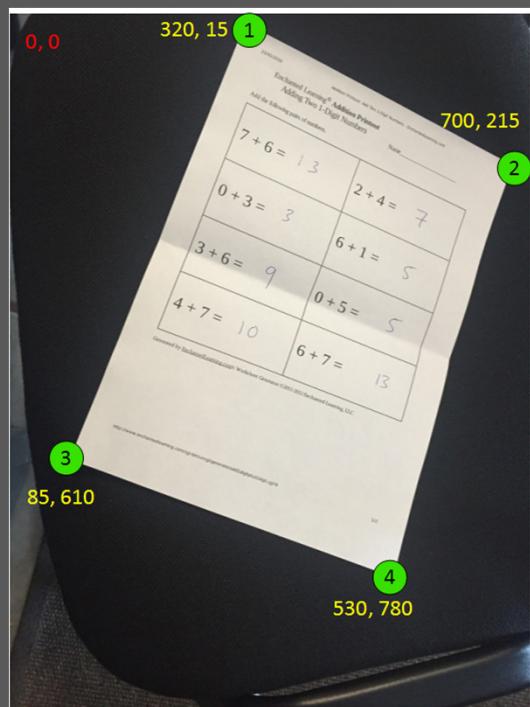
# Obtaining the Perspective of Non-Affine Transforms

What if we had a mapping of points between images?

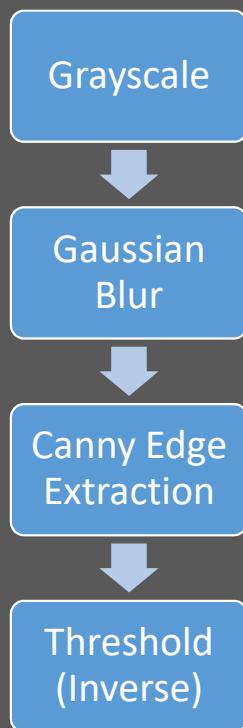


# Obtaining the Perspective of Affine Transforms

What if we had a mapping of points between images?



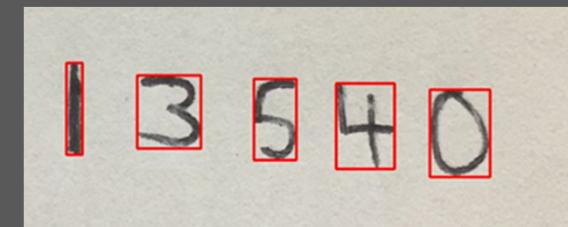
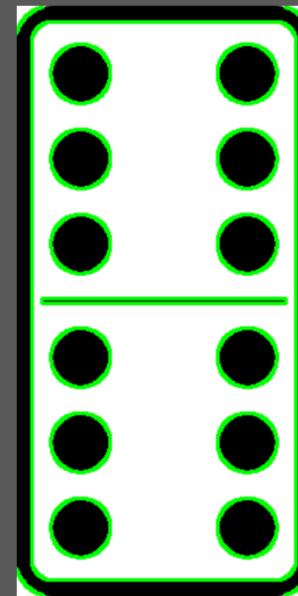
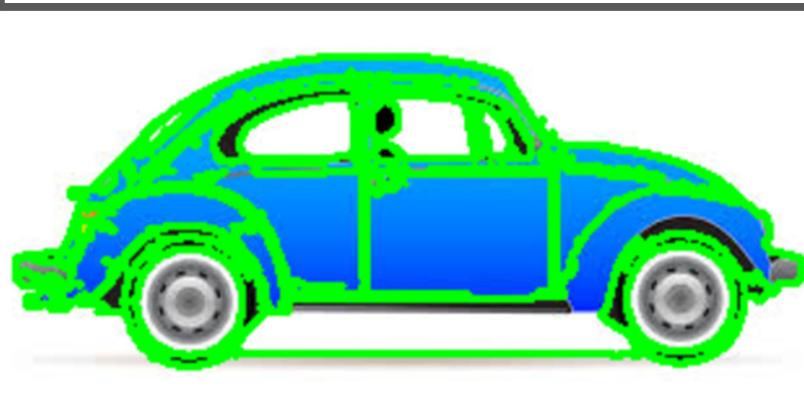
# Mini Project # 1 – Live Sketch Using Webcam



## Section 4

# Image Segmentation

## Segmentation - Partitioning images into different regions

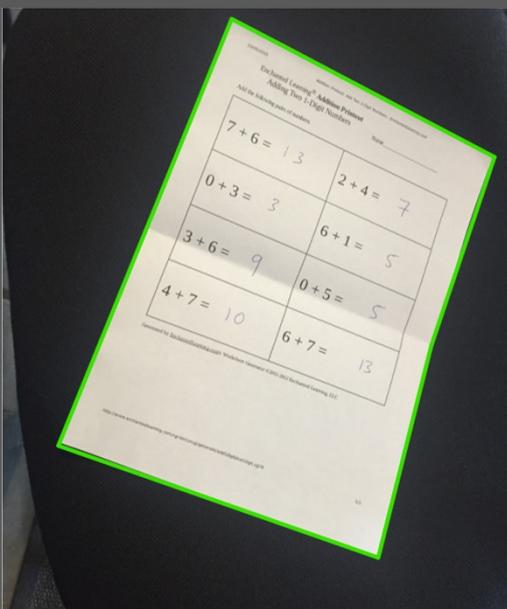


# Image Segmentation

1. Understanding contours
2. Sorting contours by size or left to right
3. Approximating contours & finding their convex hull
4. Matching Contour Shapes
5. **Mini Project # 2** – Identifying Shapes
6. Line Detection
7. Circle Detection
8. Blob Detection
9. **Mini Project # 3** – Counting Circles and Ellipses

# Contours

- Contours are continuous lines or curves that bound or cover the full boundary of an object in an image.



Contours are very important in:

- Object Detection
- Shape Analysis

# Contours

- OpenCV stores Contours in a list of lists.

# Contours in OpenCV

- `cv2.findContours(image, Retrieval Mode, Approximation Method)`
  - **Returns** -> contours, hierarchy
  - **Contours** are stored as a numpy array of (x,y) points that form the contour
  - **Hierarchy** describes the child-parent relationships between contours (i.e. contours within contours)
- Retrieval Mode:
  - `cv2.CHAIN_APPROX_NONE` – Stores all the points along the line (inefficient!)
  - `cv2.CHAIN_APPROX_SIMPLE` – Stores the end points of each line

**NOTE:** Differences between OpenCV 2.4 and OpenCV 3.X:

- `cv2.findContours` – returns 3 arguments in OpenCV3, which is the modified image with contours overlaid

# Drawing Contours

```
cv2.drawContours(image, contours, specific contour, color,  
thickness)
```

- The 'contours' parameter is the output from findContours
- Specific contour relates to which contour you wish to draw e.g. Contour[0], Contour[1].
- If you wish to draw all contours, use '-1'

# Hierarchy in Contours

Hierarchy Types (the first two are the most useful)

- cv2.RETR\_LIST – Retrieves all contours
- cv2.RETR\_EXTERNAL - Retrieves external or outer contours only
- cv2.RETR\_COMP - Retrieves all in a 2-level hierarchy
- cv2.RETR\_TREE - Retrieves all in full hierarchy

Hierarchy is stored in the following format: [Next, Previous, First Child, Parent]

NOTE - Contour Hierarchy is a quite lengthy to explain, if you're interested read here:

[http://opencv-python-tutorials.readthedocs.io/en/latest/py\\_tutorials/py\\_imgproc/py\\_contours/py\\_contours\\_hierarchy/py\\_contours\\_hierarchy.html](http://opencv-python-tutorials.readthedocs.io/en/latest/py_tutorials/py_imgproc/py_contours/py_contours_hierarchy/py_contours_hierarchy.html)

# Sorting Contours

- Sorting contours is quite useful when doing image processing.
- Sorting by Area can assist in Object Recognition (using contour area)
  - Eliminate small contours that may be noise
  - Extract the largest contour
- Sorting by spatial position (using the contour centroid)
  - Sort characters left to right
  - Process images in specific order

# Contour Moments

- Moments are a set of scalers that are an aggregate of set of vectors
- Used in Mechanics and Statistics quite frequently and has now been adopted in Computer Vision
- Without getting to heavy into the mathematics behind it think of it as a measure of image intensities
- An image with pixel intensities  $I(x,y)$ , moments are given by:

$$M_{ij} = \sum_x \sum_y I(x, y)$$

# Approximating Contours

- Approximating contours is useful when correcting slight distortions in your contour.
- We can use approxPolyDP to achieve this:

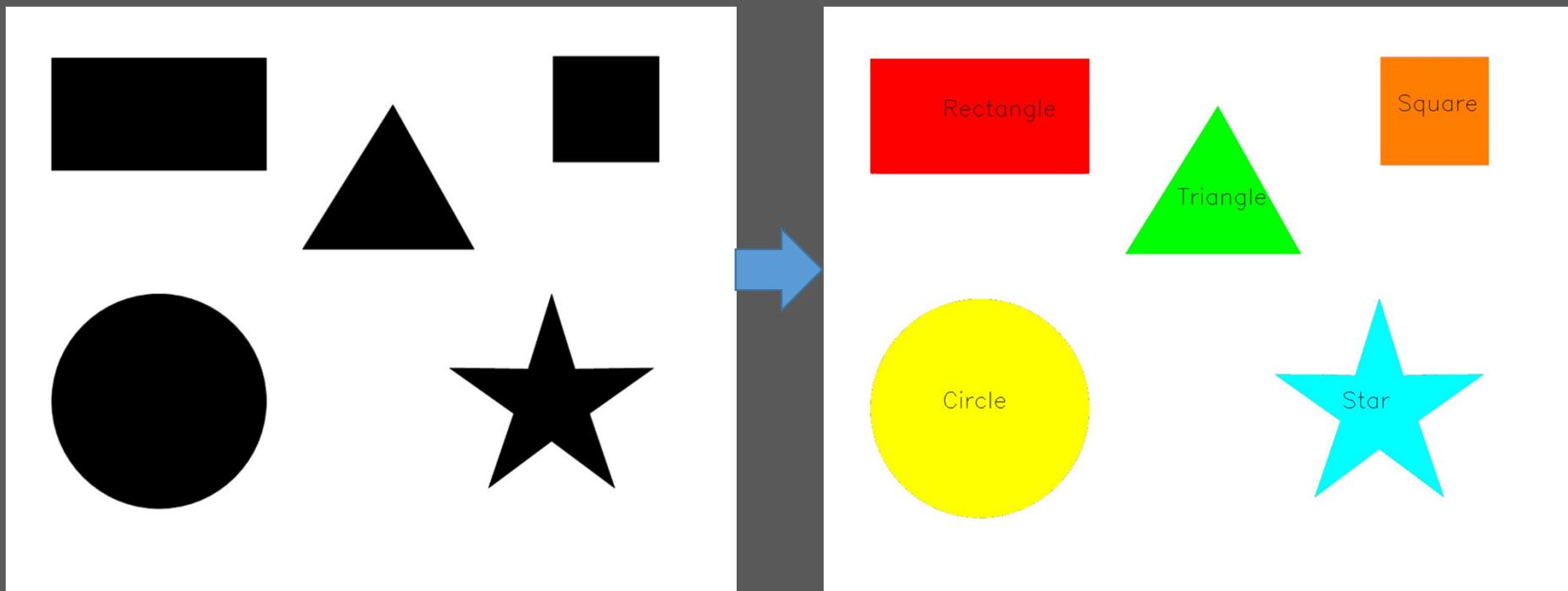
## **cv2.approxPolyDP (contour, Approximation Accuracy, Closed)**

- Contour – is the individual contour we wish to approximate
- Approximation Accuracy – Important parameter is determining the accuracy of the approximation. Small values give precise approximations, large values give more generic approximation. A good rule of thumb is less than 5% of the contour perimeter
- Closed – a Boolean value that states whether the approximate contour should be open or closed

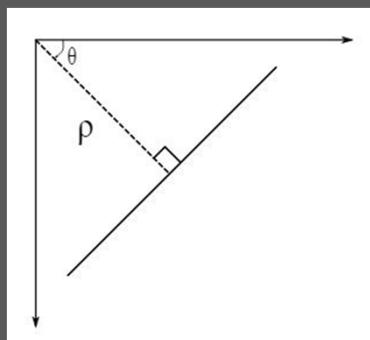
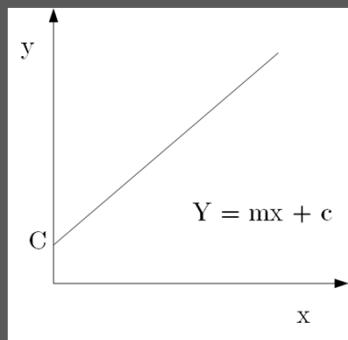
# Using OpenCV's matchShapes function

- **cv2.matchShapes (contour template, contour, method, method parameter)**
  - Output – match value (lower values means a closer match)
- Contour Template – This is our reference contour that we're trying to find in the new image
- Contour – The individual contour we are checking against
- Method – Type of contour matching (1, 2, 3)
- Method Parameter – leave alone as 0.0 (not fully utilized in python OpenCV)

# Mini Project # 2 – Shape Matching

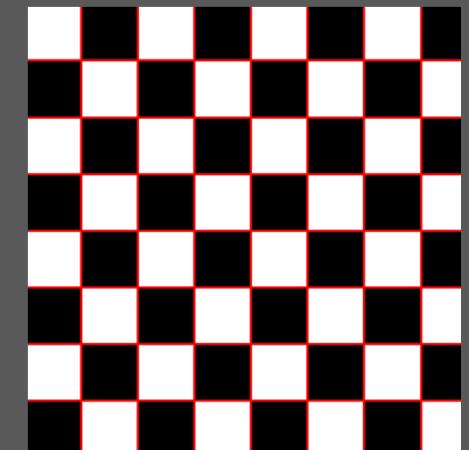


# Line Detection – Hough Lines & Probabilistic Hough Lines



$$\rho = x\cos\theta + y\sin\theta$$

$\rho$  is the perpendicular distance from origin  
 $\theta$  is the angle formed by the normal of this line to the origin (measured in radians)



`cv2.HoughLines(binarized image, rho accuracy, theta accuracy, threshold)`

- Threshold here is the minimum vote for it to be considered a line

Probabilistic Hough Lines (<http://cmp.felk.cvut.cz/~matas/papers/matas-bmvc98.pdf>)

- Idea is that it takes only a random subset of points sufficient enough for line detection
- Also returns the start and end points of the line unlike the previous function

`cv2.HoughLinesP(binarized image, rho accuracy, theta accuracy, threshold, minLineLength, maxLineGap)`

# Circle Detection

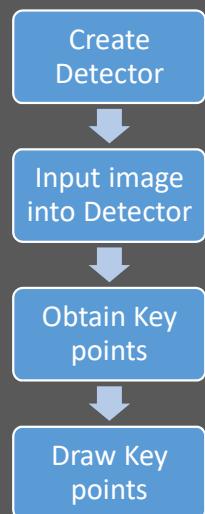
- `cv2.HoughCircles(image, method, dp, MinDist, param1, param2, minRadius, MaxRadius)`

- Method - currently only cv2.HOUGH\_GRADIENT available
- dp - Inverse ratio of accumulator resolution
- MinDist - the minimum distance between the center of detected circles
- param1 - Gradient value used in the edge detection
- param2 - Accumulator threshold for the HOUGH\_GRADIENT method,
  - Lower allows more circles to be detected (false positives)
- minRadius - limits the smallest circle to this size (via radius)
- MaxRadius - similarly sets the limit for the largest circles

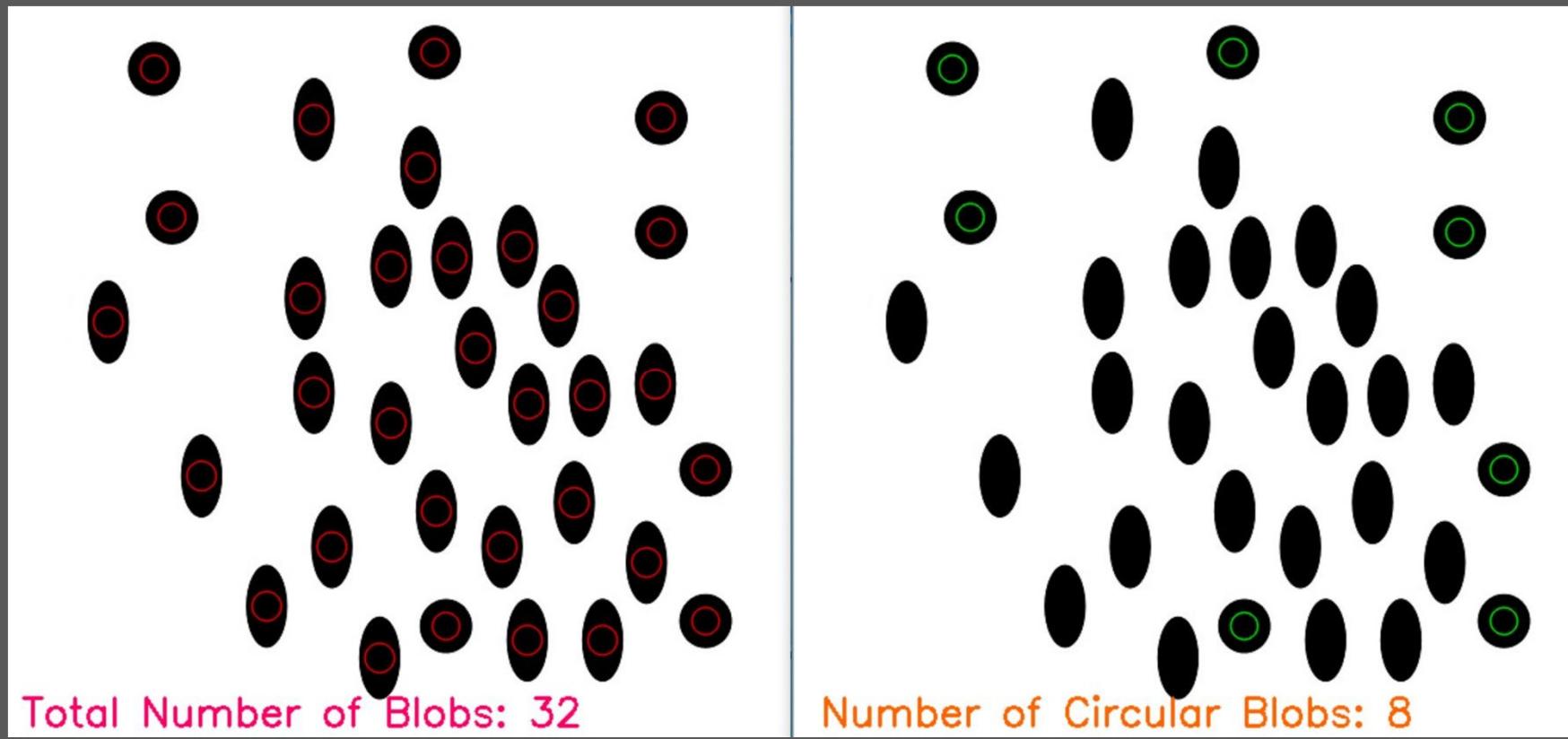


# Blob Detection

- Blobs can be described as groups of connected pixels that all share a common property.
- How to use OpenCV's simpleBlobDetector?



# Mini Project # 3 – Counting Circles and Ellipses



# Blob Filtering – Shape & Size

`cv2.SimpleBlobDetector_Params()`

## Area

- `params.filterByArea = True/False`
- `params.minArea = pixels`
- `params.maxArea = pixels`

## Circularity

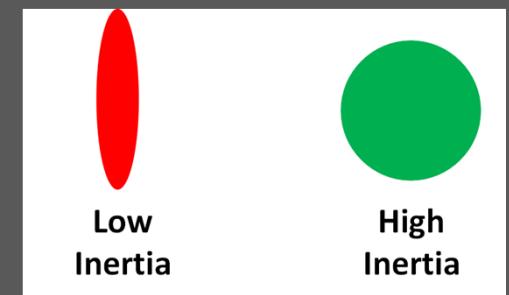
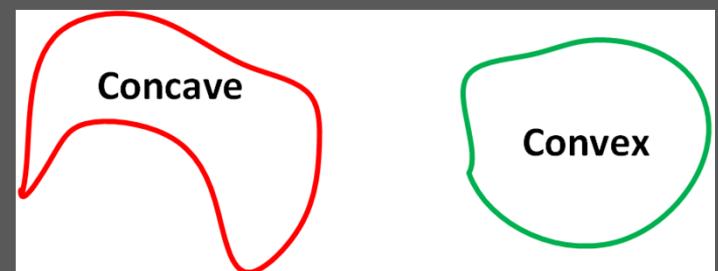
- `params.filterByCircularity = True/False`
- `params.minCircularity = 1 being perfect circle, 0 the opposite`

## Convexity – Area of blob / Area of Convex Hull

- `params.filterByConvexity = True/False`
- `params.minConvexity = 0 to 1`

## Inertia – Measure of ellipticalness (low being more elliptical, high being more circular)

- `params.filterByInertia = True/False`
- `params.minInertiaRatio = 0.01`



# Section 5

## Object Detection

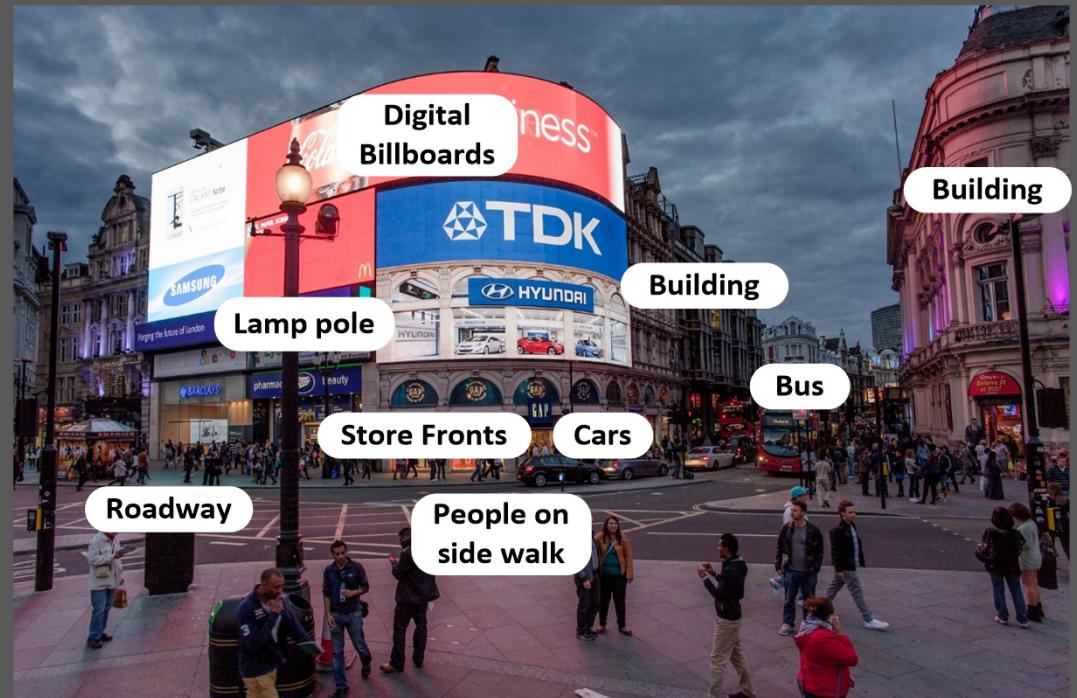
# Object Detection

1. Object Detection using Template Matching
2. Mini Project – Finding Waldo
3. Feature Description Theory
4. Finding Corners
5. SIFT, SURF, FAST, BREIF & ORB
6. Mini Project – Object Detection using Features
7. Histogram of Gradients (HoG) as a Descriptor

# Object Detection

Why do we need **detect objects** in image?

- Labeling Scenes
- Robot Navigation
- Self Driving Cars
- Body Recognition (Microsoft Kinect)
- Disease & Cancer Detection
- Facial Recognition
- Handwriting Recognition
- Identifying objects in satellite images
- Etc.

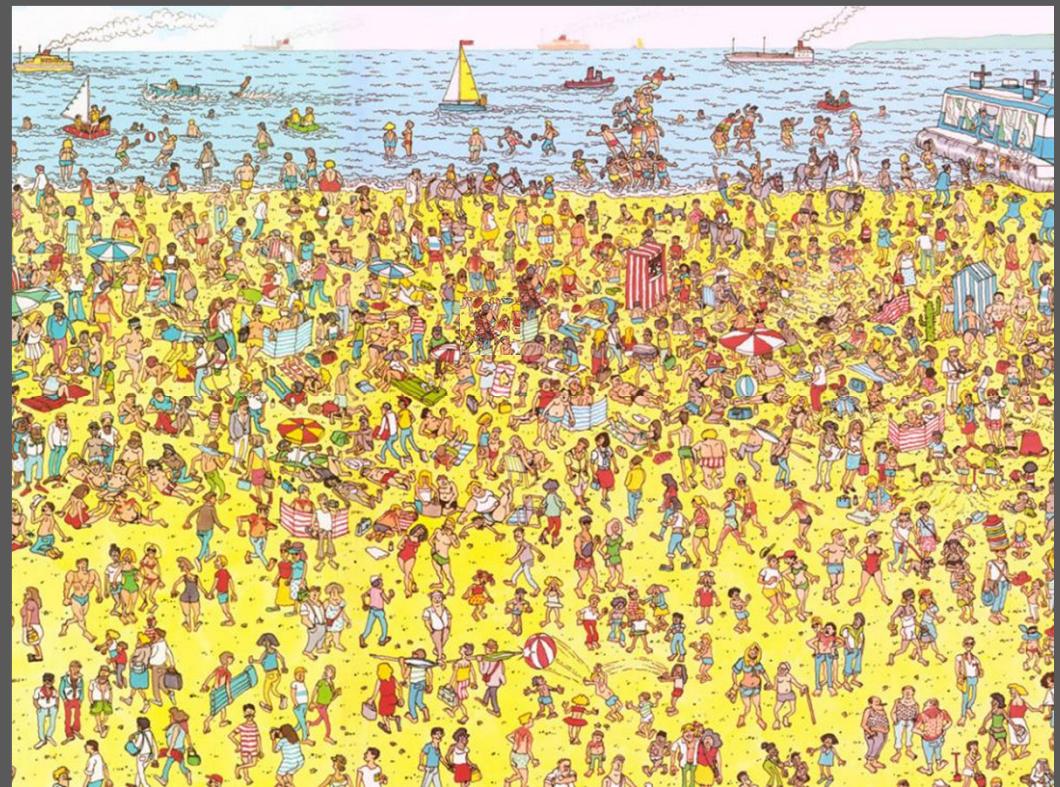


# Object Detection vs Recognition

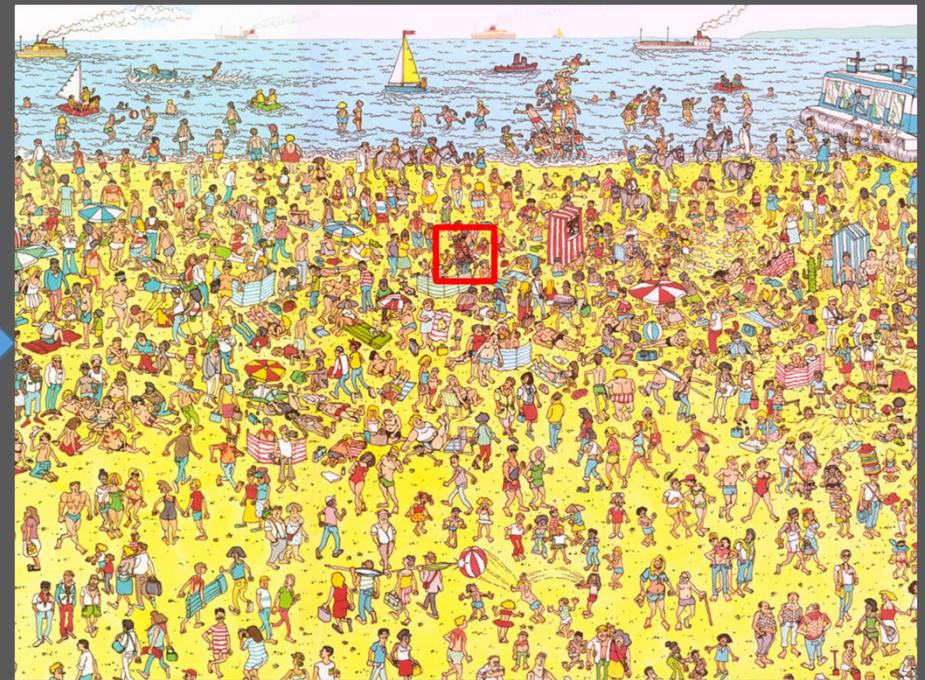
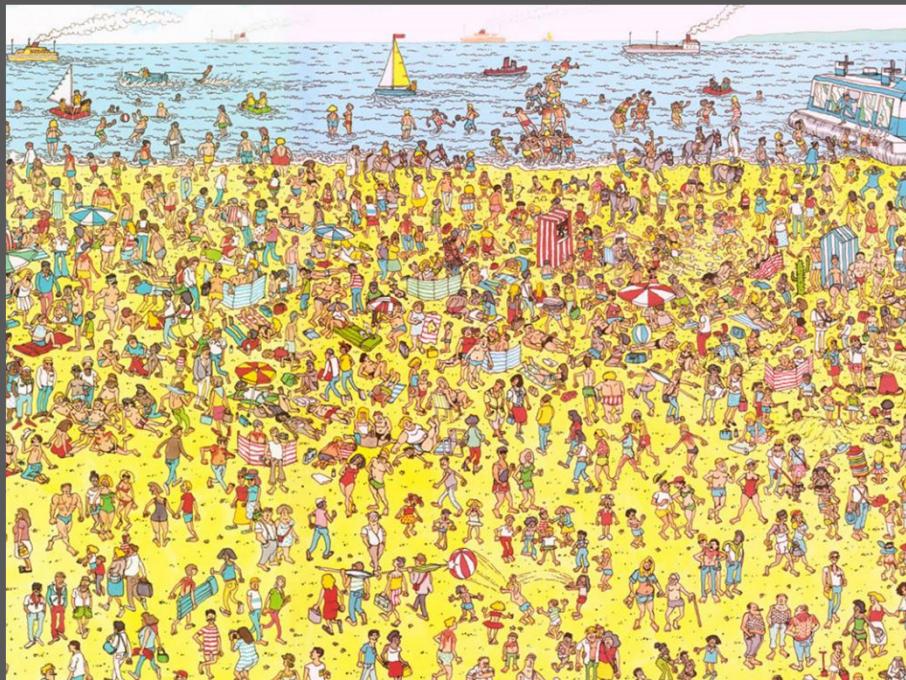


# Template Matching

- What if we slide a template image across a source image until the a match is found.



# Mini Project # 4 – Finding Waldo



# This method isn't very resilient

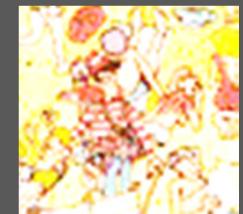
- Rotation renders this method ineffective



- Size (known as scaling) affects this as well



- Photometric changes (e.g. brightness, contrast, hue etc.)

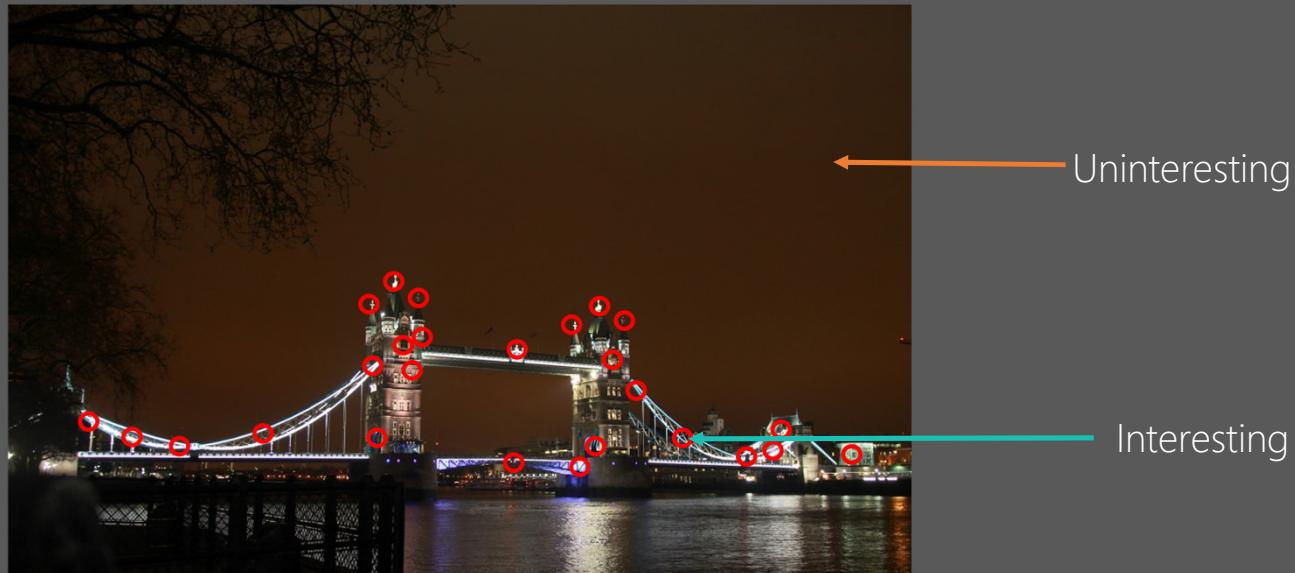


- Distortions from view point changes (Affine)



# Image Features

- Image Features are **interesting** areas in an image that are somewhat **unique** to that specific image. They are also popularly called **key point features** or **interest points**.

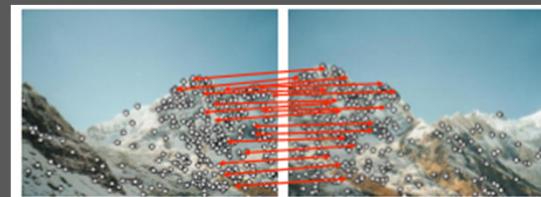
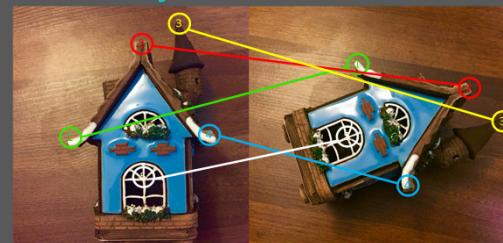


# Why is this important?

- Features are important as they can be used to analyze, describe and match images.

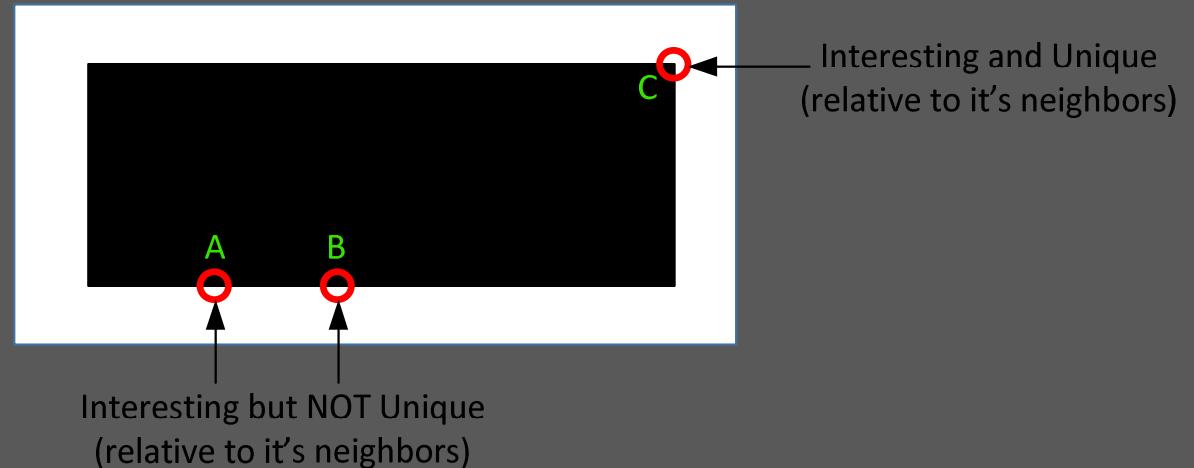
- They are used in:

- **Image Alignment** – e.g. Panorama Stitching (finding corresponding matches so we can stitch images together)
- **3D Reconstruction**
- **Robot Navigation**
- **Object Recognition**
- **Motion Tracking**
- **And more!**



# What Defines Interesting?

- Interesting areas carry a lot of **distinct** and **unique** information at that point.
  - High change of intensity
  - Corners or edges
  - And more!



- Be careful that noise can appear "informative" when it is not!

# Characteristics of Good or Interesting Features

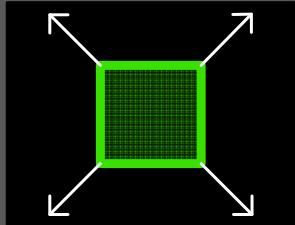
- **Repeatable** – they can be found in multiple pictures of the same scene
- **Distinctive** – Each feature is somewhat unique and different to other features of the same scene
- **Compactness/Efficiency** – Significantly less features than pixels in the image
- **Locality** – Feature occupies a small area of the image and is robust to clutter and occlusion



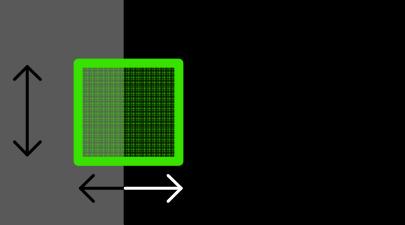
# Corners as Features

- Corners are identified when shifting a window in any direction over that point gives a large change in intensity.

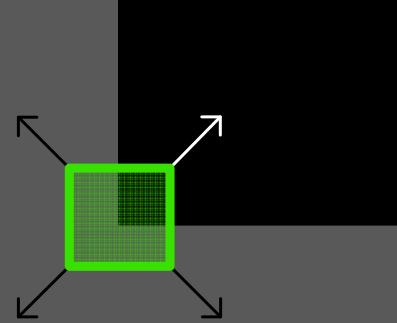
**Flat** – No change in any direction



**Edge** – Change in one direction



**Corner** – Change in all directions



# Corner Detection Methods – Harris Corners

- Harris Corner Detection is an algorithm developed in 1998 for corner detection (<http://www.bmva.org/bmvc/1988/avc-88-023.pdf>) and works fairly well.
  - `cv2.cornerHarris (input image, block size, ksize, k)`
    - Input image - should be grayscale and float32 type.
    - blockSize - the size of neighborhood considered for corner detection
    - ksize - aperture parameter of Sobel derivative used.
    - k - harris detector free parameter in the equation
    - Output – array of corner locations (x,y)
- It was improved in 1994 by improving the scoring function when determining corners locations.
  - `cv2.goodFeaturesToTrack`

## Corner Detection Methods – 'good Features To Track'

- It was improved in 1994 by improving the scoring function when determining corners locations.
  - `cv2.goodFeaturesToTrack (input image, maxCorners, qualityLevel, minDistance)`
- **Input Image** - 8-bit or floating-point 32-bit, single-channel image.
- **maxCorners** – Maximum number of corners to return. If there are more corners than are found, the strongest of them is returned.
- **qualityLevel** – Parameter characterizing the minimal accepted quality of image corners. The parameter value is multiplied by the best corner quality measure (smallest eigenvalue). The corners with the quality measure less than the product are rejected. For example, if the best corner has the quality measure = 1500, and the qualityLevel=0.01 , then all the corners with the quality measure less than 15 are rejected.
- **minDistance** – Minimum possible Euclidean distance between the returned corners.

# Problems with corners as features

Corner matching in images is **tolerant** of:

- Rotations
- Translations (i.e. shifts in image)
- Slight photometric changes e.g. brightness or affine intensity

However, it is **intolerant** of:

- Large changes in intensity or photometric changes)
- Scaling (i.e. enlarging or shrinking)

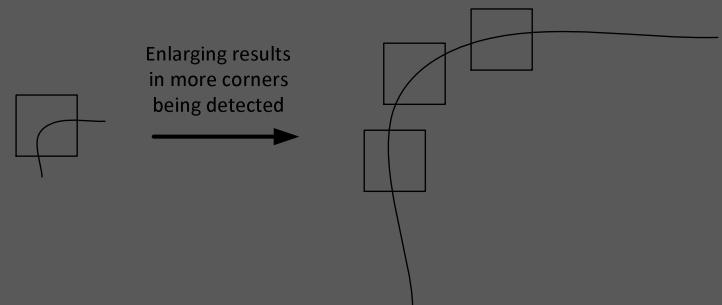
Intensity Issues



Scaling Issues



Enlarging results  
in more corners  
being detected



# Introducing SIFT (Scale Invariant Feature Transform)

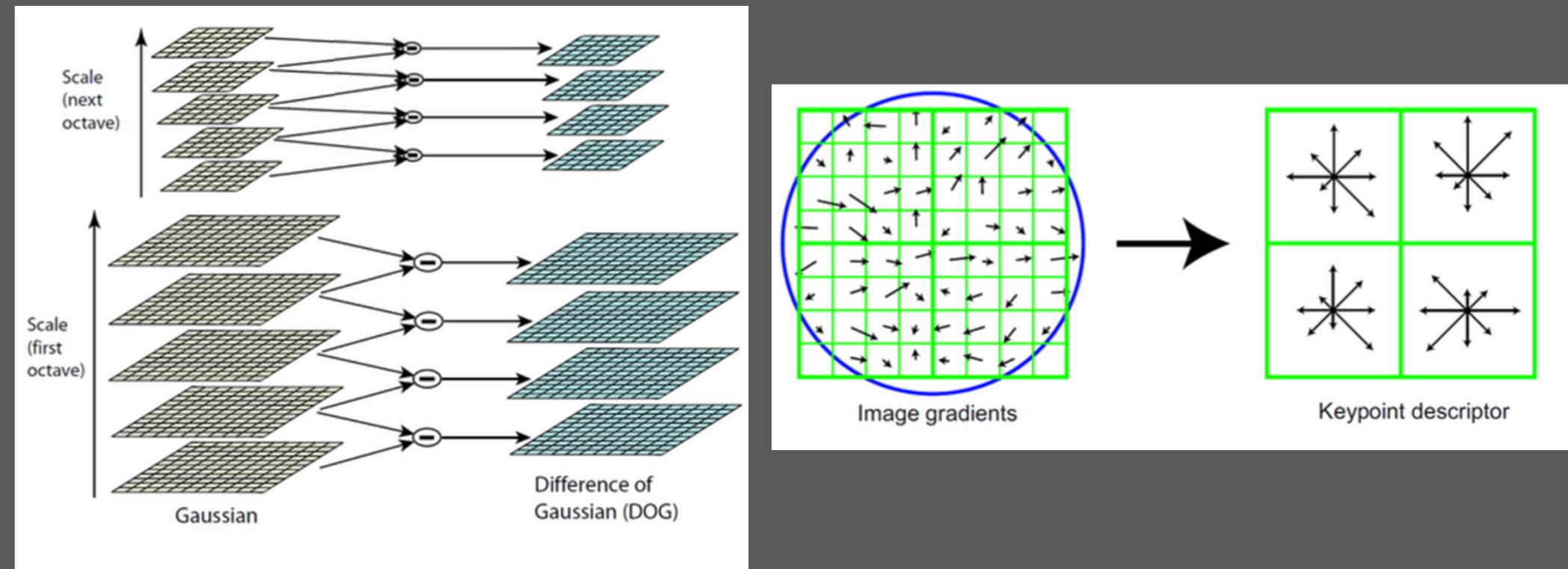
- Needed tolerance to **scaling** (*known as scale invariance*)
- SIFT is widely used (although patented) in computer vision as it very successfully dealt with the scale invariance issue
- Patented and no longer freely available with OpenCV 3.0+



# SIFT in a nutshell

1. We firstly detect **interesting key points** in an image using the Difference of Gaussian method. These are areas of the image where variation exceeds a certain threshold and are better than edge descriptors.
2. We then **create vector descriptor** for these interesting areas. Scale invariance is achieved via the following process:
  - i. Interest points are scanned at **several different scales**
  - ii. The scale at which we meet a **specific stability criteria**, is then selected and is encoded by the vector descriptor. Therefore, regardless of the initial size, the more stable scale is found which allows us to be scale invariant.
3. Rotation invariance is achieved by obtaining the **Orientation Assignment** of the key point using image gradient magnitudes. Once we know the 2D direction, we can normalize this direction.
4. The full paper on SIFT can be read here:
  - <http://www.cs.ubc.ca/~lowe/papers/ijcv04.pdf>
5. An excellent tutorial on SIFT also available here
  - [http://opencv-python-tutorials.readthedocs.io/en/latest/py\\_tutorials/py\\_feature2d/py\\_sift\\_intro/py\\_sift\\_intro.html](http://opencv-python-tutorials.readthedocs.io/en/latest/py_tutorials/py_feature2d/py_sift_intro/py_sift_intro.html)

# SIFT (Scale Invariant Feature Transform)



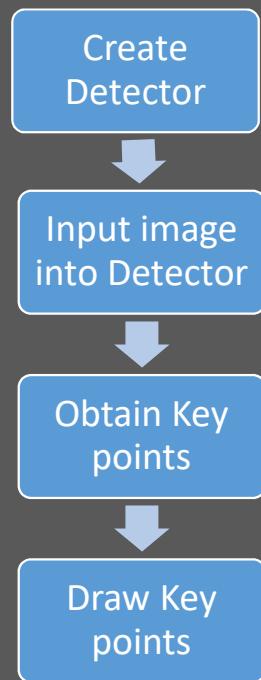
# Speeded Up Robust Features (SURF)

- SIFT is quite effective but computationally expensive
- SURF was developed to improve the speed of a scale invariant feature detector
- Instead of using the Difference of Gaussian approach, SURF uses Hessian matrix approximation to detect interesting points and use the sum of Haar wavelet responses for orientation assignment.

# Alternatives to SIFT and SURF

- **Features from Accelerated Segment Test (FAST)**
  - Key point detection only (no descriptor, we can use SIFT or SURF to computer that)
  - Used in real time applications
- **Binary Robust Independent Elementary Features (BRIEF)**
  - Computers descriptors quickly (instead of using SIFT or SURF)
  - Fast
- **Oriented FAST and Rotated BRIEF (ORB)** – Developed out of OpenCV Labs (not patented so free to use!)
  - Combines both Fast and Brief
  - [http://www.willowgarage.com/sites/default/files/orb\\_final.pdf](http://www.willowgarage.com/sites/default/files/orb_final.pdf)

# Using SIFT, SURF, FAST, BRIEF & ORB in OpenCV



# Mini Project # 5 – Object Detection



# Histogram of Oriented Gradients (HOGs)

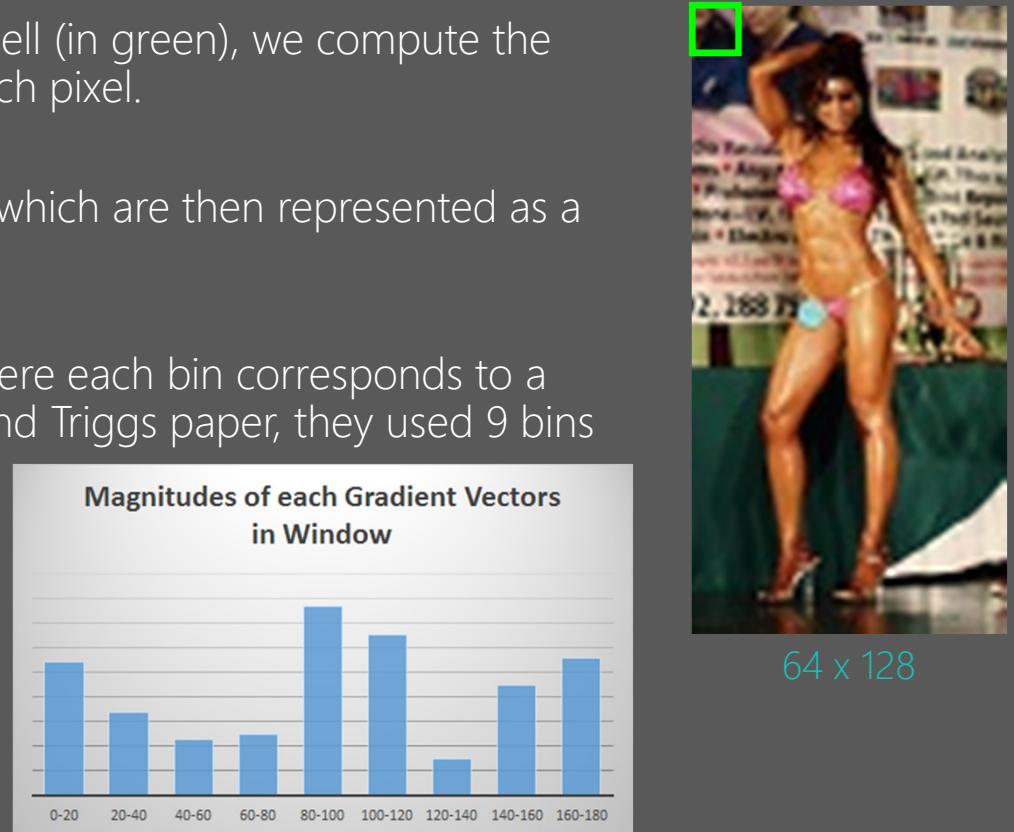
- HOGs are a **feature descriptor** that has been widely and successfully used for object detection.
- It represents objects as a **single feature vector** as opposed to a set of feature vectors where each represents a segment of the image.
- It's computed **by sliding window detector** over an image, where a HOG descriptor is a computed for each position. Like SIFT the scale of the image is adjusted (pyramiding).
- HOGs are often used with **SVM** (support vector machine) classifiers. Each HOG descriptor that is computed is fed to a SVM classifier to determine if the object was found or not).

Great Paper by Dalal & Triggs on using **HOGs for Human Detection**:

- <https://lear.inrialpes.fr/people/triggs/pubs/Dalal-cvpr05.pdf>

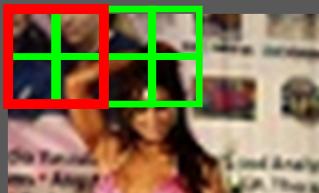
# Histogram of Gradients (HOGs) Step by Step

1. Using an  $8 \times 8$  pixel detection window or cell (in green), we compute the gradient vector or edge orientations at each pixel.
2. This generates 64 ( $8 \times 8$ ) gradient vectors which are then represented as a histogram.
3. Each cell is then split into angular bins, where each bin corresponds to a gradient direction (e.g. x, y). In the Dalal and Triggs paper, they used 9 bins  $0-180^\circ$  ( $20^\circ$  each bin).
4. This effectively reduces 64 vectors to just 9 values.
5. As it stores gradients magnitudes, it's relatively immune to deformations.

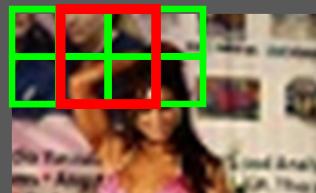


# Histogram of Gradients (HoG) Step by Step

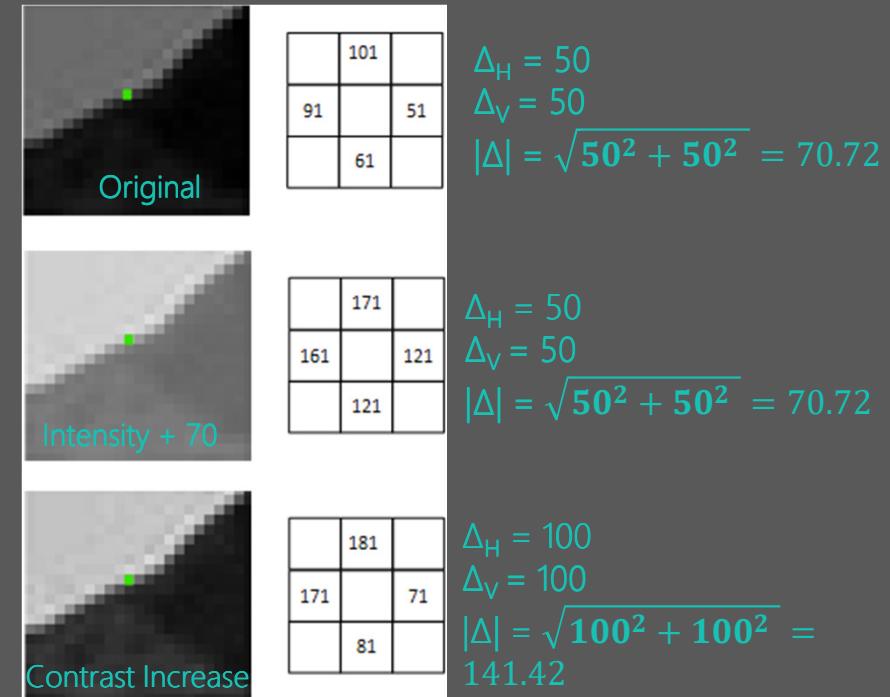
6. We then **Normalize** the gradients to ensure invariance to illumination changes i.e. Brightness and Contrast. E.g. in the images on the right, if we **divide the vectors by the gradient magnitudes** we get 0.707 for all, this is normalization.
7. Instead of individual window cell normalization, a method called Block Normalization is used. This takes into account neighboring blocks so we normalize taking into consideration larger segments of the image.



Block 1



Block 2



# **Section 6**

## **Face, People & Car Detection**

# Face, People & Car Detection

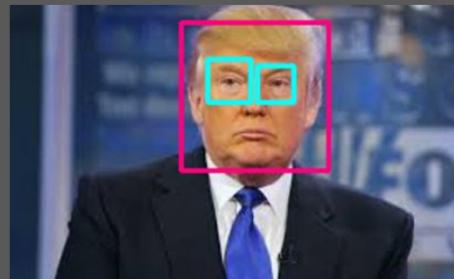
1. Understanding HAAR Cascade Classifiers
2. Face & Eye Detection
3. **Mini Project # 6 - Car Detection & Pedestrian (Body) Detection**

# HAAR Cascade Classifiers

- As we saw in the previous section, we can extract features from an image and use those features to classify objects.

## What are HAAR Cascade Classifiers?

An object detection method that inputs Haar features into a series of classifiers (cascade) to identify objects in an image. They are trained to identify one type of object, however, we can use several of them in parallel e.g. detecting eyes and faces together.

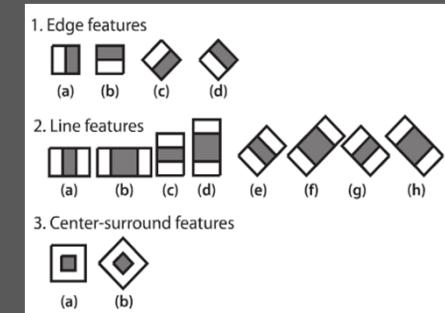
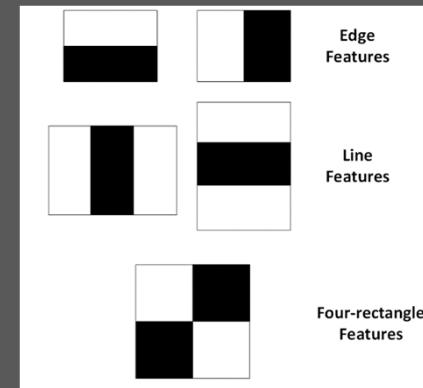


# HAAR Classifiers Explained

1. HAAR Classifiers are trained using lots of **positive images** (i.e. images with the object present) and **negative images** (i.e. images without the object present).

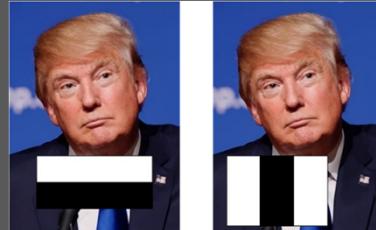
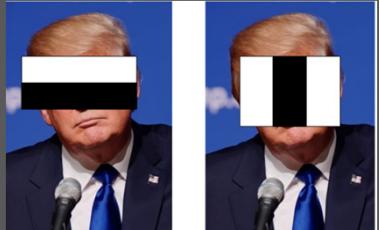


2. We then extract features using **sliding windows of rectangular blocks**. These features are single valued and are calculated by subtracting the sum of pixel intensities under the white rectangles from the black rectangles. However, this is a ridiculous number of calculations, even for a base window of 24 x 24 pixels (180,000 features generated). So the researchers devised a method called **Integral Images** that computed this with four array references.



# HAAR Classifiers Explained

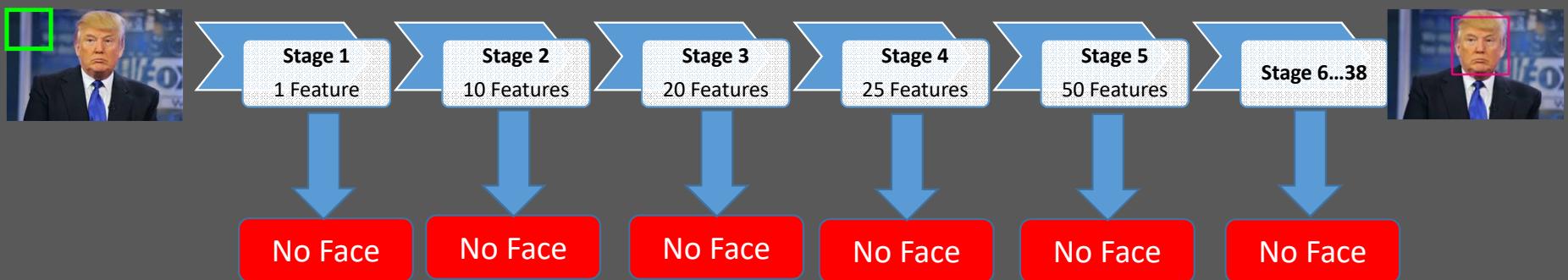
3. However, they still had 180,000 features and the majority of them added no real value.



4. Boosting was then used to determine the most informative features, with Freund & Schapire's AdaBoost the algorithm of choice due to its ease of implementation. Boosting is the process by which we use weak classifiers to build strong classifiers, simply by assigning heavier weighted penalties on incorrect classifications. Reducing the 180,000 features to 6000, which is still quite a bit features.

# HAAR Classifiers Explained

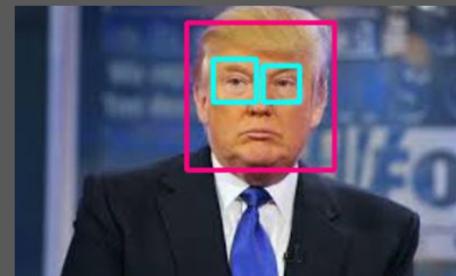
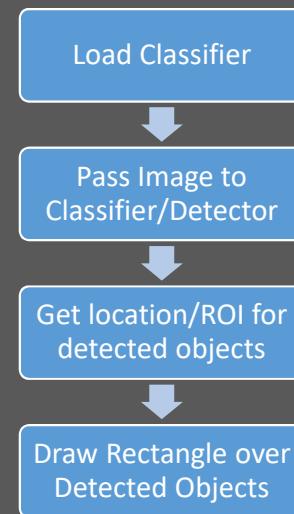
5. Think about this intuitively, if of those 6000 features, some will be more informative than others. What if we used the most informative features to first check whether the region can potentially have a face (false positives will be no big deal). Doing so eliminates the need for calculating all 6000 features at once.
6. This concept is called the Cascade of Classifiers - for face detection, the Viola Jones method used 38 stages.



# Face & Eye Detection

1. We use some pre-trained classifiers that have been provided by OpenCV
  - Stored as .XML files
2. These pre-trained classifiers can be found here:
  - <https://github.com/opencv/opencv/tree/master/data/haarcascades>

Cascade Classifiers Flow



# Parameters for detectMultiScale

```
ourClassifier.detectMultiScale(input image, Scale Factor , Min Neighbors)
```

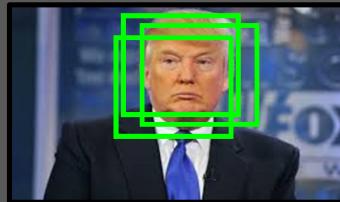
## Scale Factor

- Specifies how much we reduce the image size each time we scale. E.g. in face detection we typically use 1.3. This means we reduce the image by 30% each time it's scaled. Smaller values, like 1.05 will take longer to compute, but will increase the rate of detection.



## Min Neighbors

- Specifies the number of neighbors each potential window should have in order to consider it a positive detection. Typically set between 3-6.
  - It acts as sensitivity setting, low values will sometimes detect multiples faces over a single face. High values will ensure less false positives, but you may miss some faces.



# List of OpenCV Pre-Trained Cascade Classifiers

Found Here - <https://github.com/opencv/opencv/tree/master/data/haarcascades>

<a href="#">haarcascade_eye.xml</a>	some attempts to tune the performance
<a href="#">haarcascade_eye_tree_eyeglasses.xml</a>	some attempts to tune the performance
<a href="#">haarcascade_frontalcatface.xml</a>	Removing whitespace to appease doc builder
<a href="#">haarcascade_frontalcatface_extende...</a>	Removing whitespace to appease doc builder
<a href="#">haarcascade_frontalface_alt.xml</a>	some attempts to tune the performance
<a href="#">haarcascade_frontalface_alt2.xml</a>	some attempts to tune the performance
<a href="#">haarcascade_frontalface_alt_tree.xml</a>	some attempts to tune the performance
<a href="#">haarcascade_frontalface_default.xml</a>	some attempts to tune the performance
<a href="#">haarcascade_fullbody.xml</a>	fixing wrong model sizes
<a href="#">haarcascade_lefteye_2splits.xml</a>	some attempts to tune the performance
<a href="#">haarcascade_licence_plate_rus_16sta...</a>	Added Haar cascade for russian cars licence plate detection, 16 stage...
<a href="#">haarcascade_lowerbody.xml</a>	fixing wrong model sizes
<a href="#">haarcascade_profileface.xml</a>	some attempts to tune the performance
<a href="#">haarcascade_righteye_2splits.xml</a>	some attempts to tune the performance
<a href="#">haarcascade_russian_plate_number....</a>	Create haarcascade_russian_plate_number.xml
<a href="#">haarcascade_smile.xml</a>	fixing wrong model sizes
<a href="#">haarcascade_upperbody.xml</a>	fixing wrong model sizes

# Mini Project # 6- Car and Pedestrian Detection



## Section 7

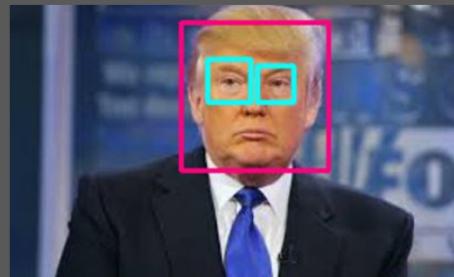
# Face Analysis & Filters

# HAAR Cascade Classifiers

- As we saw in the previous section, we can extract features from an image and use those features to classify objects.

## What are HAAR Cascade Classifiers?

An object detection method that inputs Haar features into a series of classifiers (cascade) to identify objects in an image. They are trained to identify one type of object, however, we can use several of them in parallel e.g. detecting eyes and faces together.

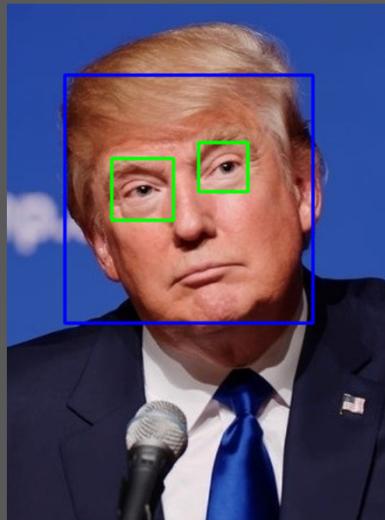


# Face Analysis & Filters

1. Finding Facial Landmarks using Dlib
2. Merging faces
3. [Mini Project # 7](#) – Live Face Swaps – Become Donald Trump or Kim Kardashian
4. [Mini Project # 8](#) – Face Reader, detect and count Yawns!

# Why is Advanced Face Morphing Hard?

- Previously we saw that HAAR Cascade Classifiers provide excellent results in Face Detection
- However, we can't simply cut someone's face out of one picture and place it onto another and expect realistic results



Face morphing isn't simply cutting and sticking a face on another image!

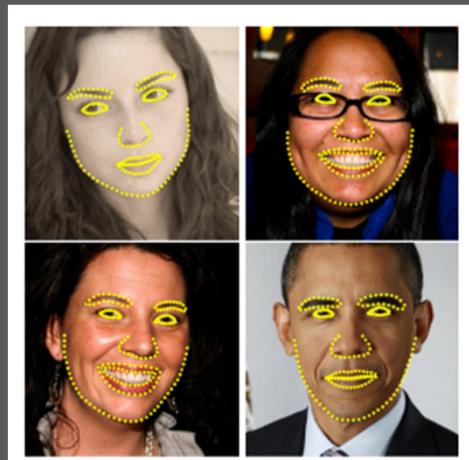


However, the latest Apps like MSQRD and Snapchat are doing this well! How?



## Researchers are getting better at detecting facial landmarks

- In 2014, Swedish Computer Vision researchers Kazemi and Sullivan, created the One Millisecond Face Alignment with Ensemble of Regression Trees.



- They developed a method to quickly determine facial landmarks in almost real-time! This was a major milestone in Face Swaps!

# Better Morphing Methods Produce Much Better (and scarier!) Results



# Better Morphing Methods Produce Much Better (and scarier!) Results



# What was so hard about that?

1. Identifying Facial Features
  2. Warping the image to fit the new and different facial expression
  3. Color Matching
  4. Creating seamless borders on the edges of the new swapped face
- 
- It is hard, but we can now do it relatively easy in Python using dlib and OpenCV

# Finding Facial Landmarks Using DLIB

- DLIB is a fantastic Machine Learning C++ library that can be utilized in python to quickly identify facial landmarks.
- Let's take a look and test it out!

# Setting up DLIB in Python

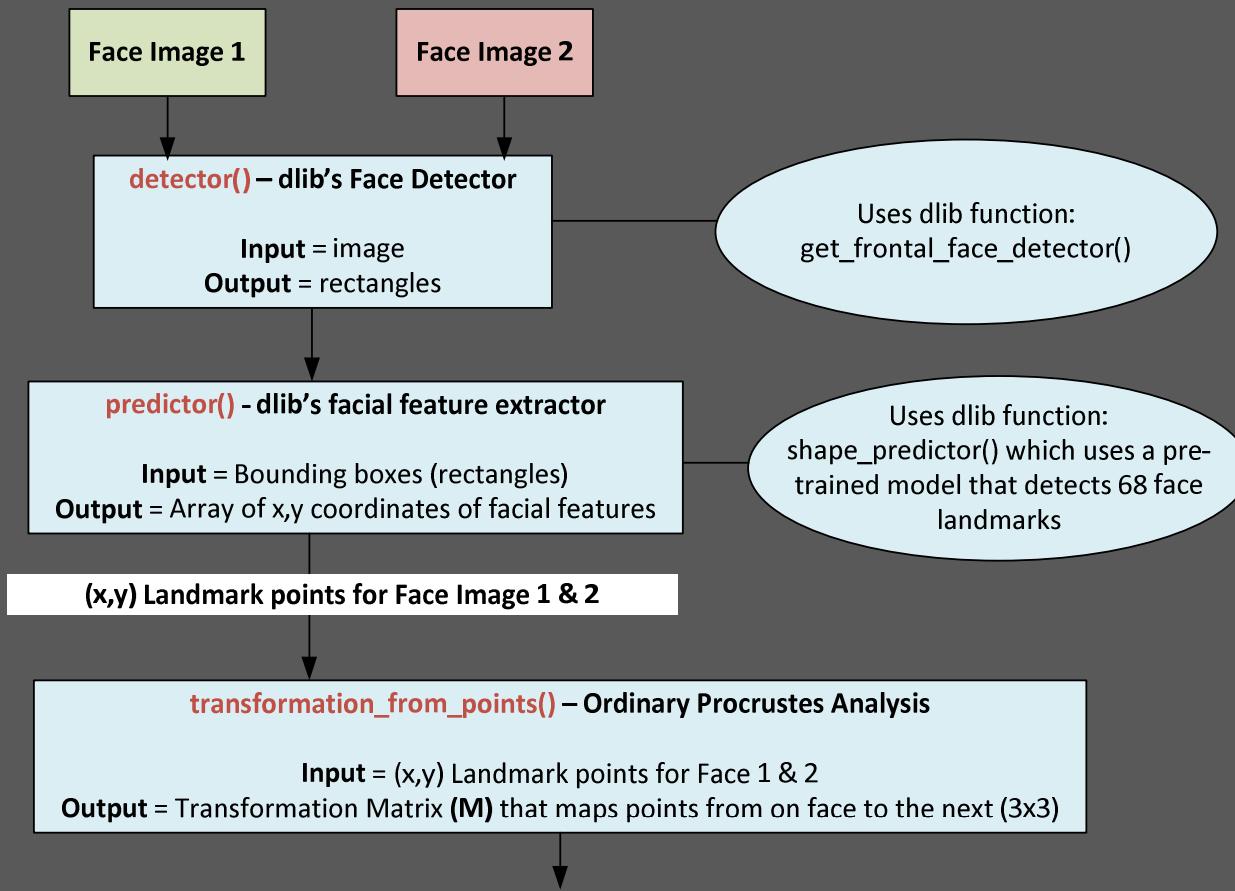
- Download and Install Dlib
  - <https://sourceforge.net/projects/dclib/>
  - Extract files in C:/dlib
  - Use command prompt to Cd to folder and run "python setup.py install"
- Download the pre-trained model here
  - [http://dlib.net/files/shape\\_predictor\\_68\\_face\\_landmarks.dat.bz2](http://dlib.net/files/shape_predictor_68_face_landmarks.dat.bz2)
  - Place this file in your default ipython notebook folder
- Run a simple example to show facial landmarks on face ----->
- Facial Landmarks Number Order
  - MOUTH\_POINTS = 48 to 61
  - RIGHT\_BROW\_POINTS = 17 to 21
  - LEFT\_BROW\_POINTS = 22 to 27
  - RIGHT\_EYE\_POINTS = 36 to 42
  - LEFT\_EYE\_POINTS = 42 to 48
  - NOSE\_POINTS = 27 to 35
  - JAW\_POINTS = 0 to 17



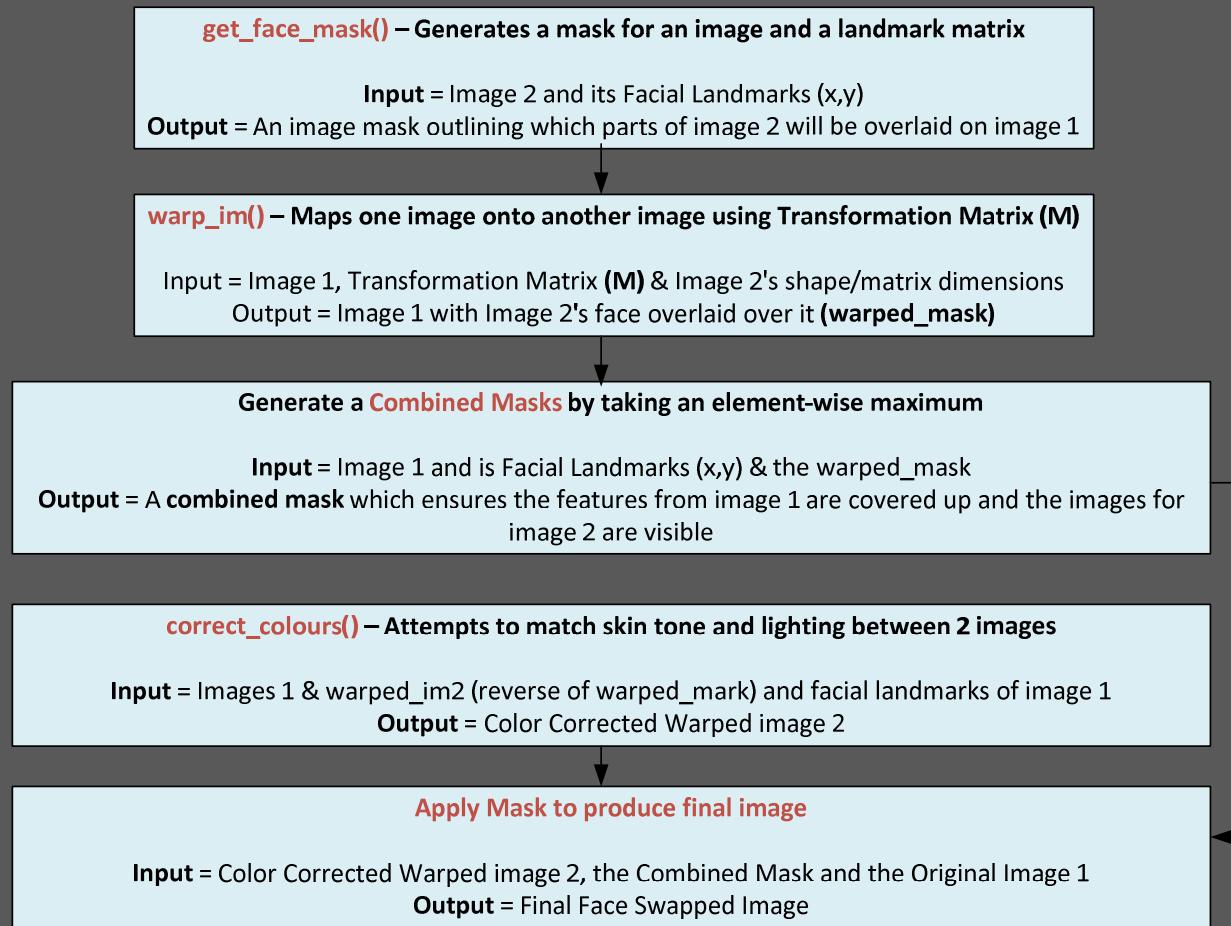
# Making Our Own Face Swapping App

- This involves some complicated math and advanced computer vision techniques
- We will not go through the code in detail, however we will attempt to understand at a high level, what each function does and the steps we need to take to create our swapped face.

# First Half of our Face Swapper



# Continued..



Let's do a quick run through of the code  
and have some fun!

It's sort of impressive for still images, but  
can we turn this into a live face swapper?

YES!

Let's now edit the code to turn this into our own  
version the popular and \$35M app MSQRD!

# Mini Project # 7 – Live Face Swapping



# Mini Project # 8 – Yawn Detector and Counter



## Section 8

# Machine Learning in Computer Vision

# Machine Learning

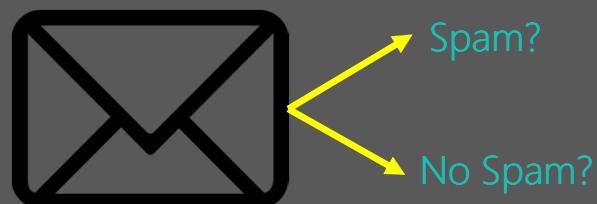
1. Overview of Machine Learning in Computer Vision and KNNs
2. **Mini Project # 9** – Handwritten Digit Recognition
3. **Mini Project # 10** – Face Recognition, Unlock Your Computer With Your Face!

# What is Machine Learning?

Machine Learning gives computers the ability to make predictions or learn from data.

Examples:

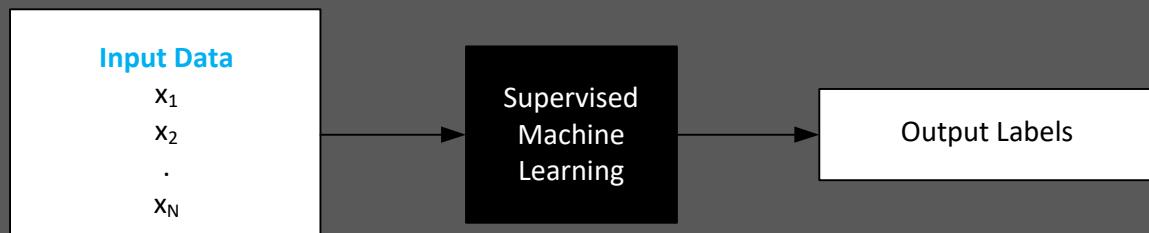
1. Using today's news articles to predict tomorrow's stock price
2. Using words in a Tweet to predict the sentiment expressed (e.g. anger, sadness, joy, political views)
3. Predicting illnesses based on medical data
4. Using your past shopping data to predict future purchases'
5. And the most popular example, filtering email spam.



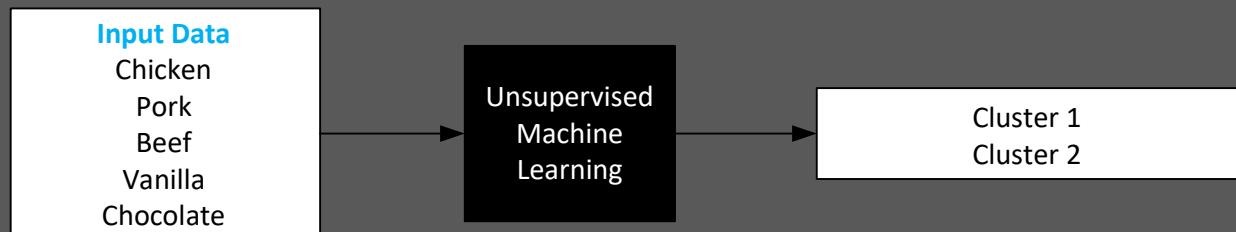
# Two main types of Machine Learning

## Supervised & Unsupervised Learning

**Supervised Learning** – We feed an algorithm some ground truth examples. It then formulates a model mapping inputs to outputs (Training Process). We then use this model to predict the outputs of new inputs.

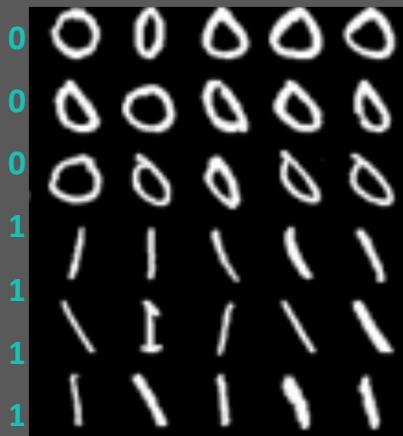


**Unsupervised Learning** – We similarly try to predict the output from input data, however no ground truth examples are given. Think clustering!



# Examples - Supervised & Unsupervised Learning

Supervised Learning



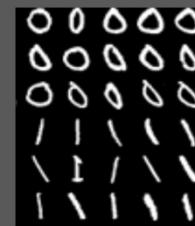
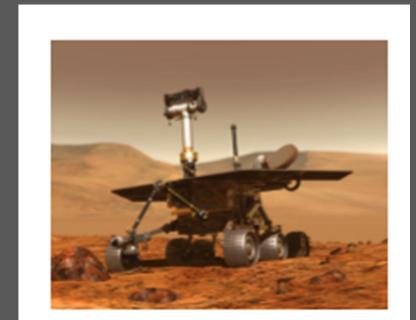
Unsupervised Learning



# Machine Learning in Computer Vision

## What is it good for?

- Predicting if a face is in image (Face Detection)
- Recognizing Faces in Images ([Mini Project # 10](#))
- Classifying the type of car in an image
- Road Sign Detection
- License Plate Reading
- Signature Verification
- Finger Print Reading
- Xbox Kinect
- Object Recognition
- Robotic Navigation
- Understanding Handwriting ([Mini Project # 9](#))



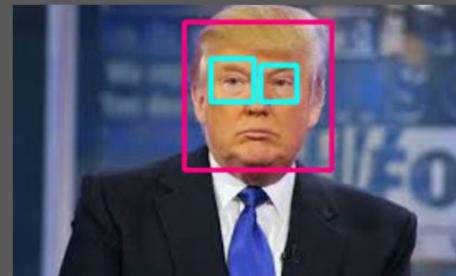
# How is typically used?

- Looking at raw pixel intensities or colors
  - Key point Descriptors
  - Histograms of Image Gradients (HOGs)
  - Pixel statistics (such as dimensions, mean, variance and many more)

# Common Machine Learning Algorithms used in Computer Vision

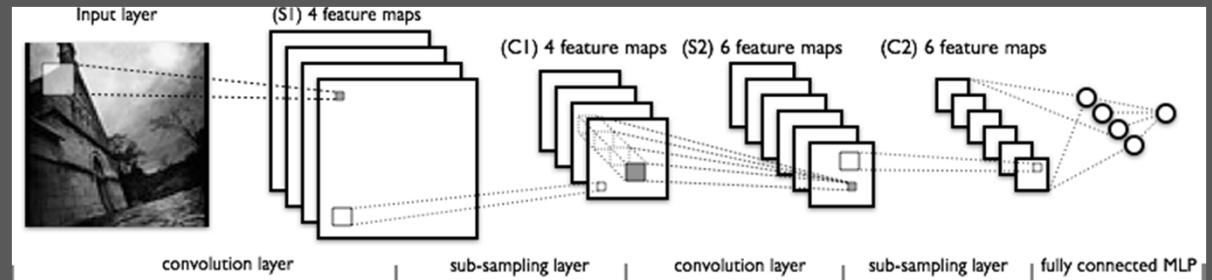
## Traditional ML Algorithms

- KNN
- SVMs
- Haar Cascade Classifiers



## Neural Networks

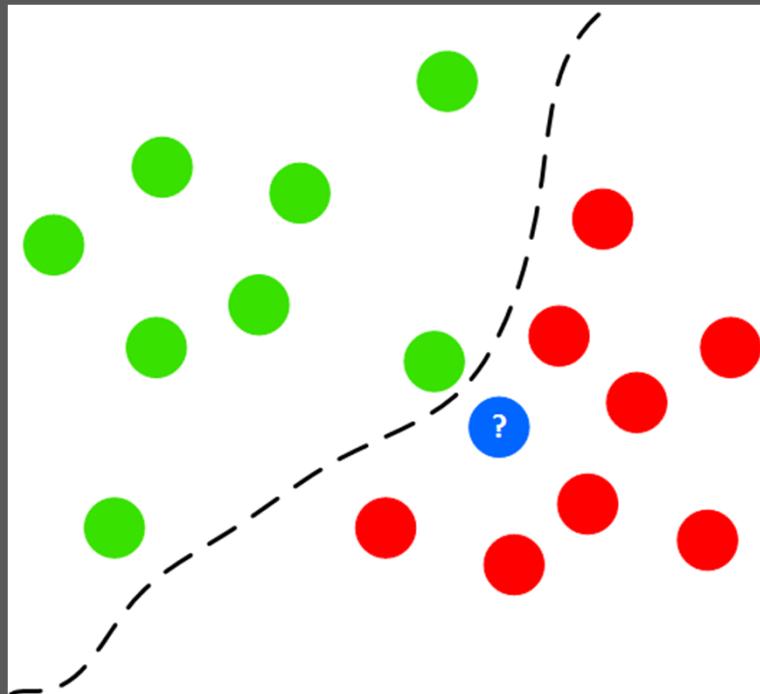
- Convolutional Neural Networks (CNN) – are able to beat humans in object recognition tasks
  - Café
  - TensorFlow
  - Theano & Keras
- Deep Belief Networks



# K-Nearest Neighbors Algorithm

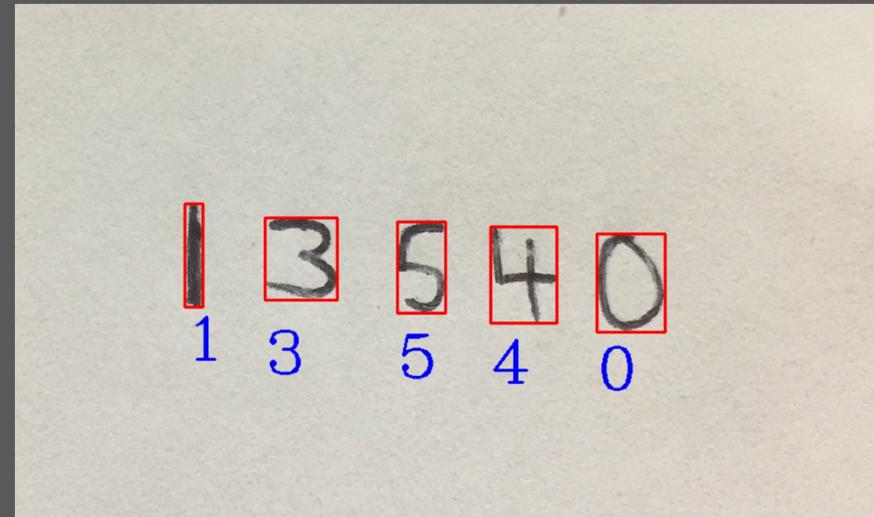
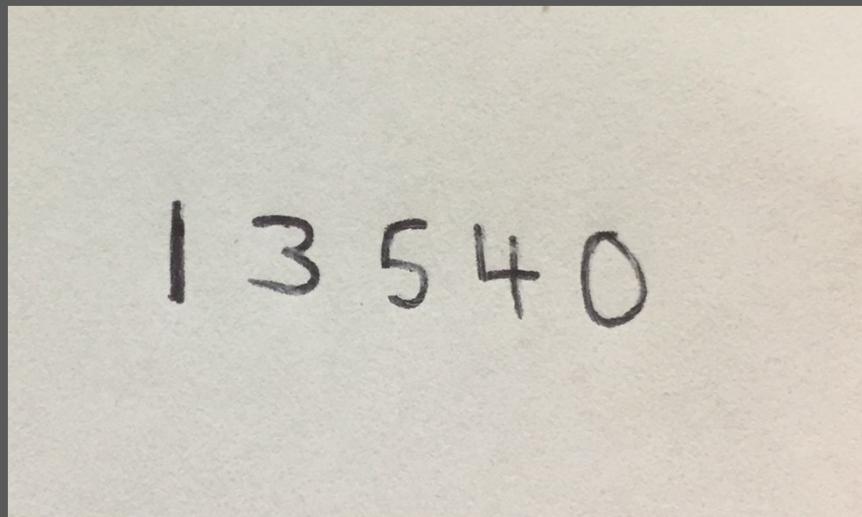
- KNN is a simple machine learning classifier that classifies a new input based on the closest examples in the feature space.
- It's based on the weighted Euclidean distance of the k-nearest neighbors.

# K-Nearest Neighbors Algorithm



- The new blue dot is closest to a green dot
- However, it's obvious that the blue dot belongs to the red class
- Let's look at the blue dot's 3 nearest neighbors:
  - 1 Green
  - 2 Red
- Using this logic, the blue should be in the red class now.
- What if k was even, or we had 3 class categories?
- To avoid ties in the above scenario, we use the Euclidean distance function:
  - $distance = \sum_{i=1}^N (a_i - b_i)^2$
- Each neighbor is assigned a distance weight which is used to determine its class e.g.
  - $d(\text{blue}, \text{green}) = 2$
  - $d(\text{blue}, \text{red1}) = 3$
  - $d(\text{blue}, \text{red2}) = 3$
- Overall Distance from Red =  $3/2 = 1.5$
- Overall Distance from Green = 2

## Mini Project # 9 – Handwritten Digits Classification

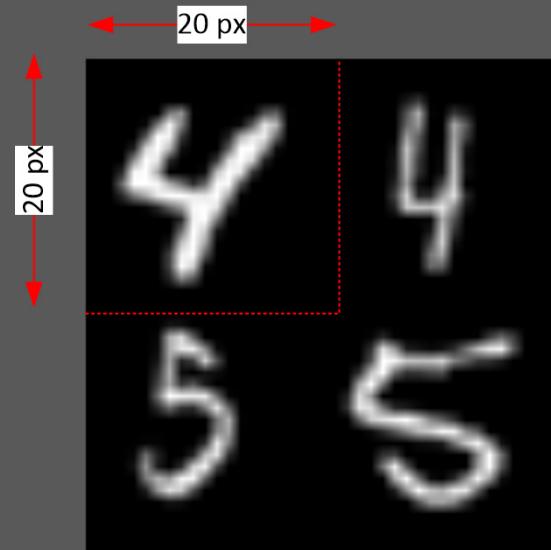


50 x 100 Images, 500 samples of each number



# About the Digits Data Set

- The digits.png image contains 500 samples of each numeral (0-9)
- Total of 5000 samples of data
- Each individual character has dimensions: 20 x 20 pixels
- Is grayscale with black backgrounds



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

# Preparing our Dataset

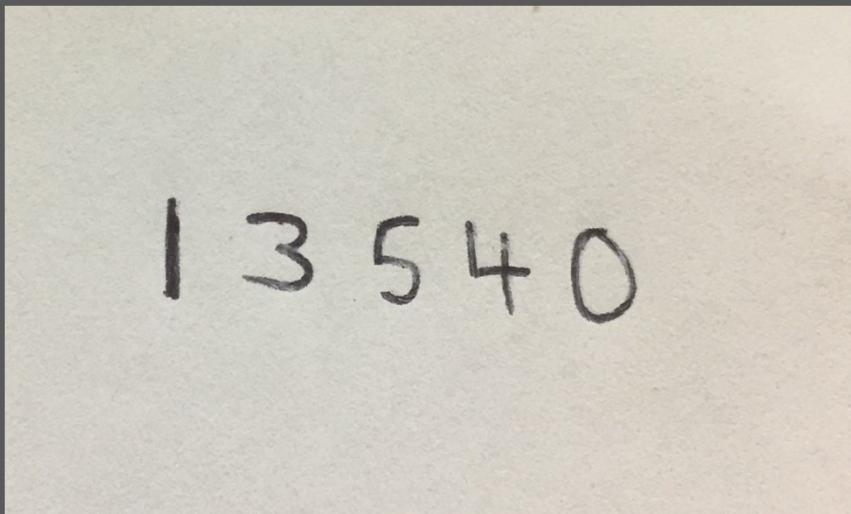
- How we prepare the data?
  - 500 samples of each digit with 5 rows of 100 samples
  - Each character is a grayscale 20 x 20 pixels
  - We use numpy to arrange the data in this format:
    - $50 \times 100 \times 20 \times 20$
  - We then split the training dataset into 2 segments and flatten our 20x20 array.
    - Training Set - 70% of the data
    - Test Set - 30% of the data - we use a test set to evaluate our model
    - Each dataset is then flattened, meaning we turn the 20 x 20 pixel array into a flat 1 x 400. Each row of 20 pixels is simply appended into one long column.
  - We then assign labels to both training & test datasets (i.e. 0,1,2,3,4,5,6,7,9)

# Using OpenCV's KNN

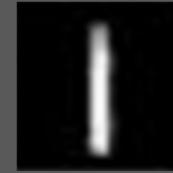
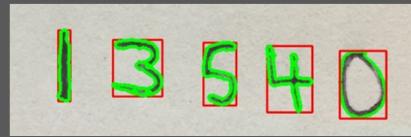
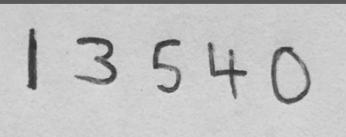
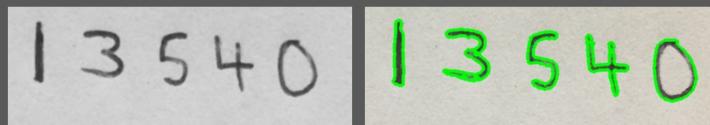
- Using OpenCV's KNN functions are very simple
1. Initialize our KNN Classifier
    - `knn = cv2.KNearest()`
  2. Train Model using the training dataset and correct labels
    - `knn.train(train, train_labels)`
  3. Evaluate Model with the `test` dataset
    - `ret, result, neighbors, distance = knn.find_nearest(test, k)`
      - ret – true or false Boolean indicating if operation was successful
      - Result – Accuracy i.e. (number correctly classified) / total number of inputs
      - Neighbors – the class of the nearest k neighbors
      - Distance – Euclidean distance from each point neighbor

# Testing Our Classifier

- So we got an accuracy of 93.47%
- How do we test this on a new image



# Processing a New Image

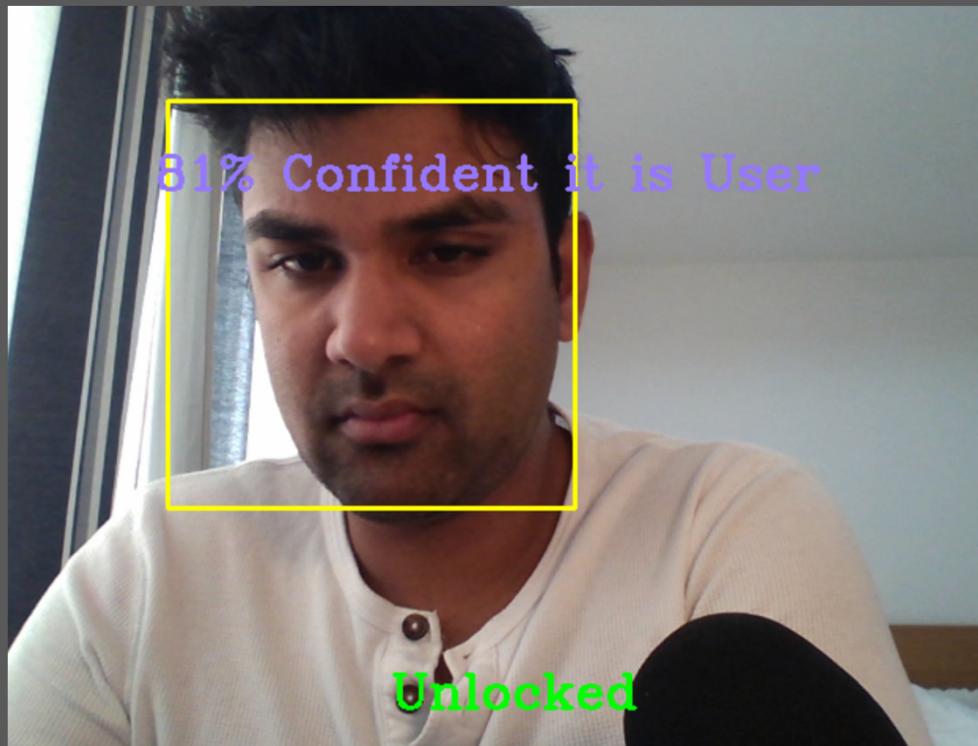


# Can we Improve this?

- Get larger Dataset ([MNIST](#) is excellent with 6000 examples per class)
- Add [perturbations](#) in data to account for variances (neat way of expanding training dataset)
- Use different classifiers, [Neural Networks](#) are excellent at getting 99%+ accuracy

Check out the current accuracy levels achieved in classifying the MNIST dataset: <http://yann.lecun.com/exdb/mnist/>

# Mini Project # 10 – Facial Recognition

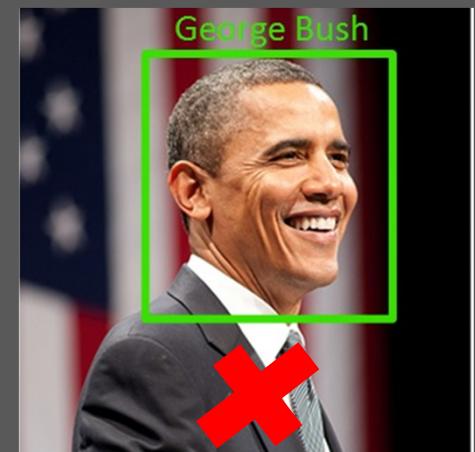


# Facial Recognition

- Facial Recognition is a task that human abilities are absolutely brilliant, and even some animals, dogs, crows, sheep can do it.

General overview of facial recognition systems steps:

1. Obtain faces & normalize images (face alignment etc.)
  - Build a dataset of face images
2. Detect and record features of face
  - General global features
  - Geometric features such as spatial relations of eyes, nose, mouth etc.
  - PCA or LDA representations
  - Local feature extraction
3. Use features to then classify face or return a confidence/probability value



# Facial Recognition with OpenCV

OpenCV comes with [3 facial recognition libraries](#), all of which operate similarly where they take our dataset of labelled faces, and compute features to represent the images. Their classifiers then utilize these features to classify.

1. [Eigenfaces](#) - `createEigenFaceRecognizer()`
2. [Fisherfaces](#) - `createFisherFaceRecognizer()`
3. [Local Binary Patterns Histograms](#) - `createLBPHFaceRecognizer()`

Learn more here - [http://docs.opencv.org/2.4/modules/contrib/doc/facerec/facerec\\_tutorial.html](http://docs.opencv.org/2.4/modules/contrib/doc/facerec/facerec_tutorial.html)

# Facial Recognition with OpenCV

1. **Eigenfaces** - Uses Principle Component Analysis to reduce the dimensionality of the face. However, this neglects the class label into account and can represent variance from changes in illumination.
2. **Fisherfaces** – Solves this by using LDA (Linear Discriminant Analysis) that is a class-specific projection, which means it attempts to minimize variance within a class, while maximizing variance between classes.
3. **Local Binary Patterns Histograms** - Uses local feature extraction while persevering spatial relationships. It divides faces into cells and then compares each cell to face being classified. It then produces a histogram showing the matching values of an area.



# Mini Project # 10 – Face Recognition

## Create our Training Data

Record 100 Images of your face using the HAAR Cascade Face Detector



Normalize by gray scaling and re-sizing to 200 x 200 pixels



Create an array of labels for recorded images (will you only one, yourself in this example)

## Classify New Face

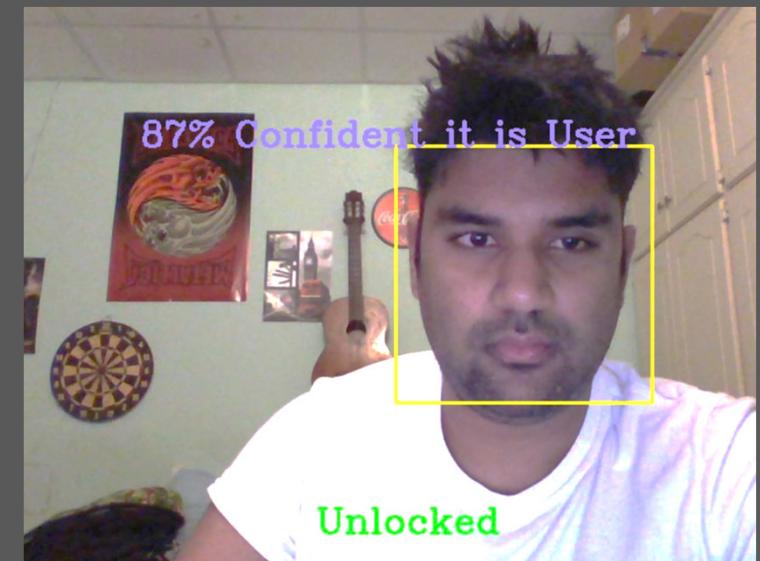
Extract Face from Webcam using the HAAR Cascade Face Detector



Normalize by gray scaling and re-sizing to 200 x 200 pixels



Pass face to our model predictor to get label and confidence value



## Section 9

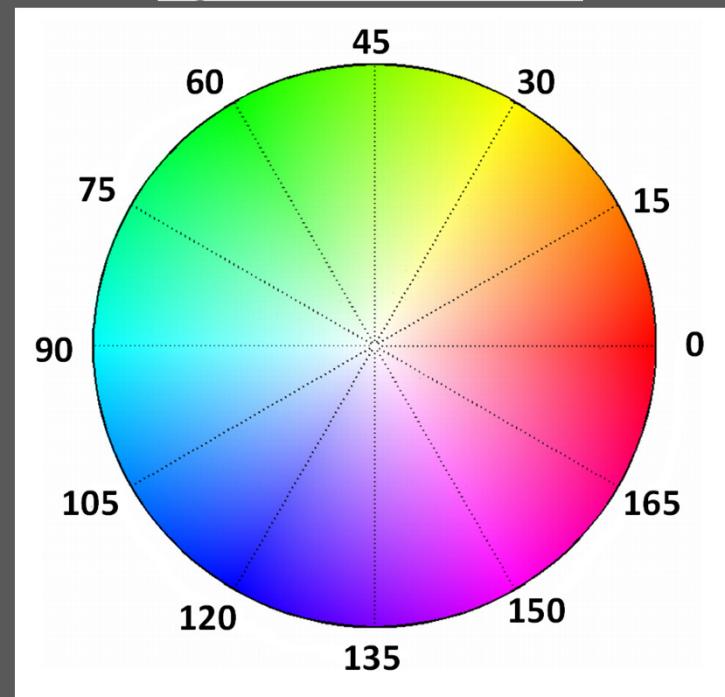
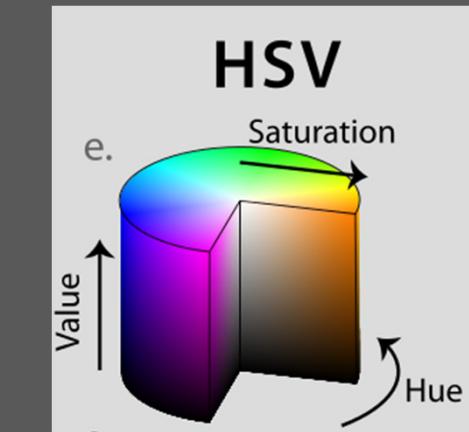
# Motion Analysis and Object Tracking

# Motion Analysis and Object Tracking

1. Filtering by Color
2. Background Subtraction
3. Meanshift
4. Camshift
5. Optical Flow
6. Mini Project # 11 – Ball Tracking

# Color Filtering

- Filter by Hue
  - Hue color range, goes from 0 to 180
  - Color Range Filters:
    - Red – 165 to 15
    - Green – 45 to 75
    - Blue – 90 to 120
- Filter by Saturation & Value/Brightness
  - Typically we set the filter range from 50 to 255 for both saturation and value as this captures the actual colors
  - 0 to 60 in Saturation is close to white
  - 0 to 60 in Value is close to black



# Implementing a Color Filter

1. Define upper and lower range of color filter.
2. Create a binary (thresholded) mask showing only the desired colors in white
  - `mask = cv2.inRange(hsv_img, lower_color_range, upper_color_range)`
3. Perform a `bitwise_and` operation on the original image and the `mask`.



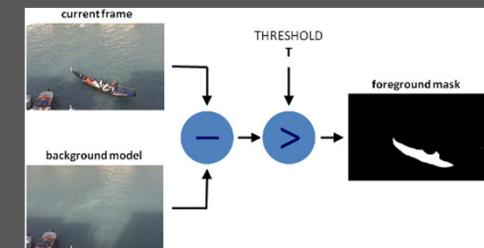
# Background Subtraction

This is a very useful computer vision technique which allows us to [separate foregrounds from the backgrounds](#) in a video stream.

These algorithms essentially learn about the frame in view (video stream) and are able to accurately “learn” and identify the foreground mask. What results is a binary segmentation of the image which highlights regions of non-stationary objects.

There are several Background subtraction algorithms in OpenCV specifically for video analysis:

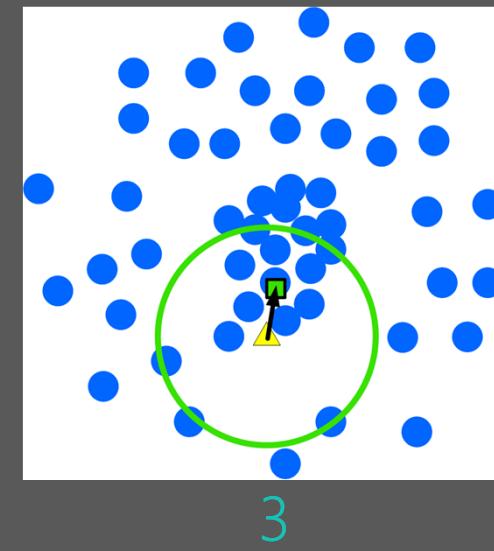
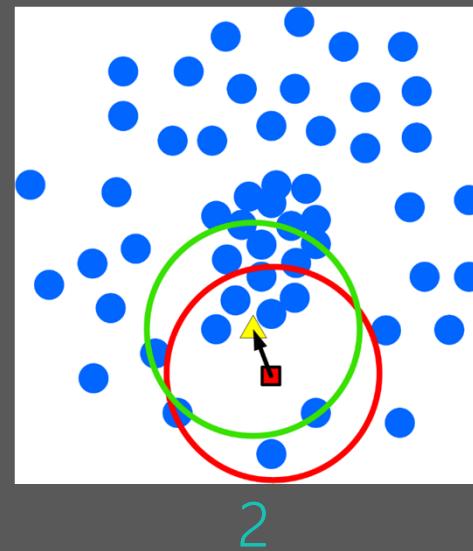
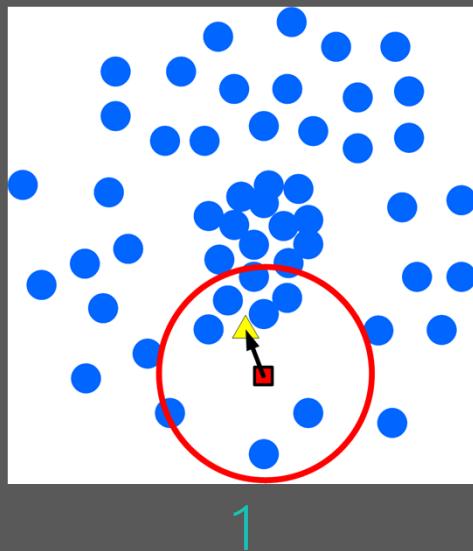
- [BackgroundSubtractorMOG](#) - Gaussian Mixture-based background/foreground Segmentation Algorithm.
- [BackgroundSubtractorMOG2](#) – Another Gaussian Mixture-based background subtraction method, however with better adaptability to illumination changes and with better ability to detect shadows!
- [Geometric Multigrid \(GMG\)](#) - This method combines statistical background image estimation and per-pixel Bayesian segmentation



# Meanshift – An Object Tracking Algorithm

Its premise is simple, it tracks objects by finding the [maximum density of a discrete sample of points](#) and then recalculates it at the next frame. This effectively, moves our observation window in the direction the object has moved.

In OpenCV we typically use the histogram back projected image and initial target location.



# Camshift – An Object Tracking Algorithm

Camshift is very similar to Meanshift, however you may have noted the [window in Meanshift is of a fixed size](#). That is problematic since movement in images can be small or large. If the window is too large, you can miss the object when tracking.

Camshift (Continuously Adaptive Meanshift) uses an adaptive window size that changes both size and orientation (i.e. rotates). We'll simply its steps here:

1. Applies Meanshift till it converges
2. Calculates the size of the window
3. Calculates the orientation by using the best fitting ellipse

# When and how to use Meanshift or Camshift?

- If you have some prior knowledge of the object being tracked (e.g. size wrt to camera point of view) then Meanshift would work well.
- Employ Camshift when the object being track is changing shape wrt to the camera perspective. Generally more versatile, but also more sensitive.

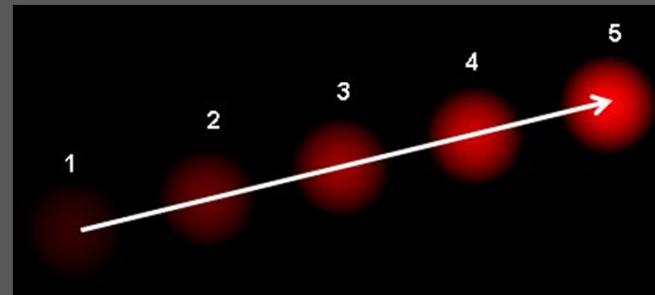
**Tip:** Beware of the starting location of the window, you can get stuck in a local minima!



# Optical Flow

Seeks to get the pattern of apparent motion of objects in an image between two consecutive frames.

Shows the distribution of the apparent velocities of objects in an image.



# Optical Flow Implementations in OpenCV

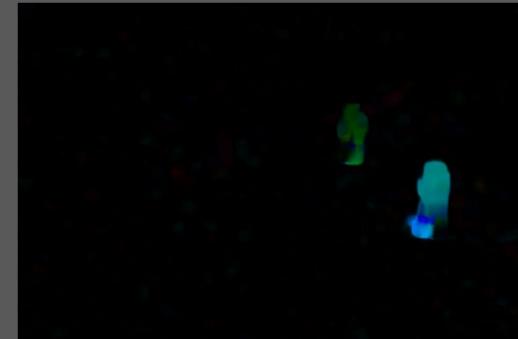
OpenCV has two implementations of Optical Flow.

- **Lucas-Kanade Differential Method** – Tracks some keypoints in the video, good for corner-like features (tracking cars from drones)
  - <http://www.cse.psu.edu/~rtc12/CSE486/lecture30.pdf>
  - [https://www.cs.cmu.edu/afs/cs/academic/class/15385-s12/www/lec\\_slides/Baker&Matthews.pdf](https://www.cs.cmu.edu/afs/cs/academic/class/15385-s12/www/lec_slides/Baker&Matthews.pdf)
- **Dense Optical Flow** – Slower, but computes the optical flow for all points in a frame, unlike Lucas-Kanade which uses corner features (sparse dataset). Colors are used to reflect movement with Hue being direction and value (brightness/intensity) being speed.

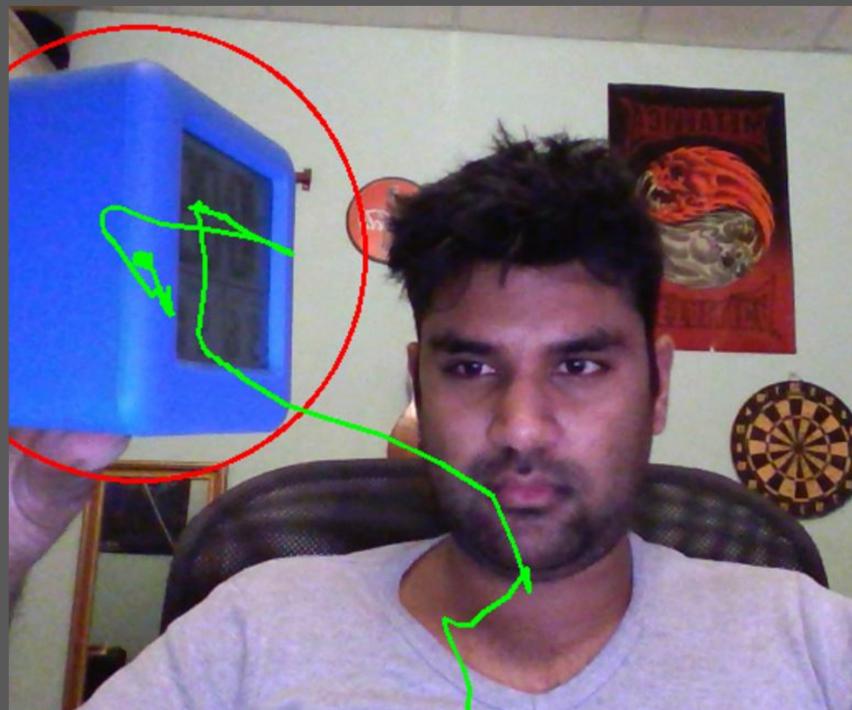
Lucas-Kanade Method



Dense Optical Flow



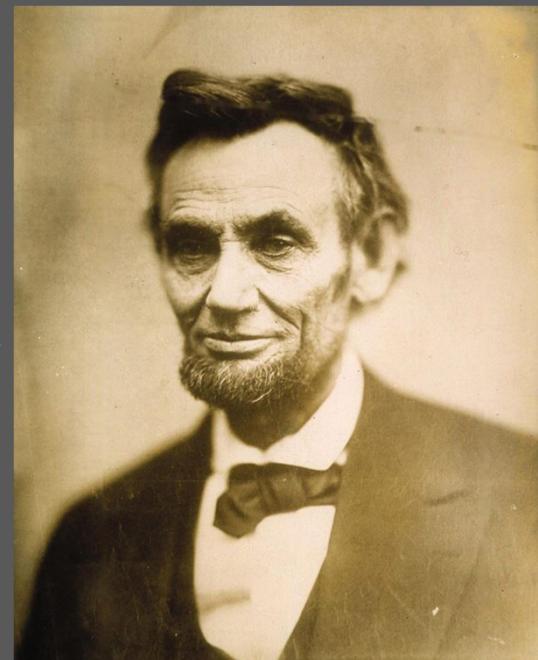
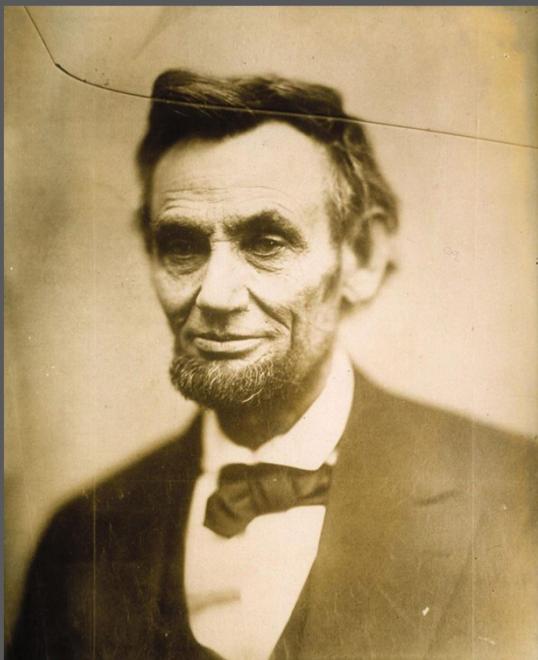
# Mini Project # 11 – Object Tracking



# Computational Photography

1. What is Computational Photography
2. Noise Reduction
3. Mini Project # 11 – Photo Restoration (remove strokes, scratches, bends in old photos)

# Mini Project # 12 – Photo-restoration



# What is Computational Photography?

- These are digital image processing techniques used on images produced by cameras.
- They seek to enhance images via computational processing rather than use expensive optical processes (cost more and are bulky).
- Used significantly in all cameras, especially smartphones:
  - Noise Reduction
  - High Dynamic Range
  - Image stabilization
  - Panoramas
  - [Inpainting](#) (removal of small noises, strokes etc.)



# Noise Reduction

- These are digital image processing techniques used on images produced by cameras.
- They seek to enhance images via computational processing rather than use expensive optical processes (cost more and are bulky).
- Used significantly in all cameras, especially smartphones:
  - Noise Reduction
  - High Dynamic Range
  - Inpainting (removal of small noises, strokes etc.)
  - Image stabilization
  - Panoramas

# Advanced Topics

1. Other Computer Vision Python Libraries & Tools
  - PIL
  - imutils
  - skimage
  - Matlab Labview
2. OCR (Tesseract) & OMR
  - QR Code & Barcode Reading
3. Other Advanced Algorithms
  - Watershed
  - Connected Components
  - 3D Reconstruction
  - Augmented Reality
1. Bag of Words Model
2. SVM & HOGs
3. Deep Learning & Convolutional Neural Networks (TensorFlow, theano, lasagna, café)
4. C++ vs Python OpenCV
5. OpenCV 3.1 vs 2.4
6. Mobile Apps Computer Visions

# **Section 10**

**Course Wrap Up**

# Course Wrap Up

1. Where do you go from here?
2. Computer Vision Research Areas & Startup Ideas and Mobile Computer Vision

# Consolidating What You've Learnt

- Familiarity with Python and Numpy
- Computer Vision:
  - Image Manipulation and Segmentation
  - Object Recognition Methods
  - Machine Learning in Computer Vision
  - Facial Feature Analysis and Recognition
  - Motion Analysis and Object Tracking

# Becoming an Expert



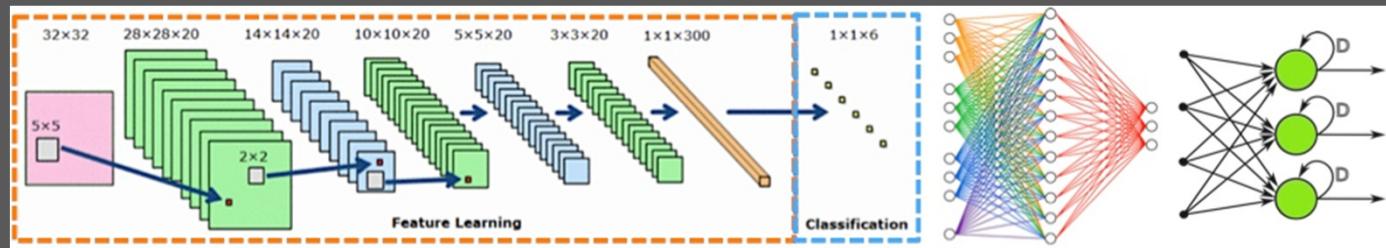
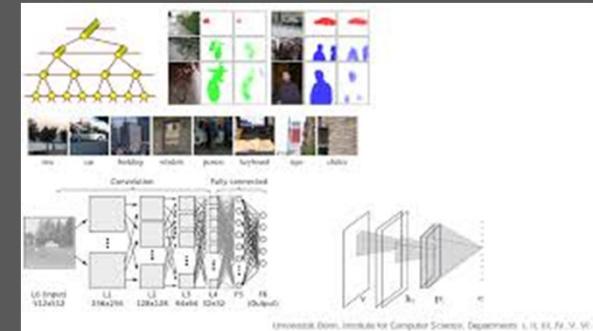
1. Create your own project
2. Read and implement research papers
  - See my document titled "[ResearchPapersandURLS.pdf](#)"
3. Explore Computer Vision Blogs
  - [pyimagesearch.com](#) (Outstanding!)
  - [learnopencv.com](#) (Outstanding!)
  - [aishack.in](#) (lots of projects)
  - [opencv.org/category/news](#)

# Current State of Computer Vision?

Object Recognition - Deep Belief and Convolution Neural Networks are changing everything!

Where to learn more?

- Stanford CS231 class - cs231n.github.io
- Café - caffe.berkeleyvision.org/tutorial
- Deep Learning Tutorial - ufldl.stanford.edu/tutorial
- Deep Learning Tutorial - deeplearning.net/reading-list/tutorials
- Deep Learning Tutorial - www.deeplearningbook.org
- Tensor Flow - www.tensorflow.org/versions/r0.11/tutorials/index.html





# 12 Awesome Startup Ideas



1. Fall Detection for elderly people
2. Searching for missing people using drones (e.g. Hikers)
3. Using drones for people counting and/or tailing
4. Using CCTV footage for Smart surveillance systems (e.g. detect when children are unsupervised near a swimming pools or balconies)
5. Identifying the state of crops (e.g. checking for diseases)
6. Understanding medical images (x-rays, MRI scans etc.)
7. Identify swimmers who may be in trouble
8. Identify counterfeit money or products
9. Novelty apps (e.g. upload a photo of chess or checkers being played and it then gives you the next best move)
10. Take a picture of a product and it brings reviews or prices of that item online
11. Identifying calories and macros from a picture of food
12. Analyze faces to detect emotions and wellbeing
13. [Want More?](http://cs229.stanford.edu/projects2015.html#vision) - cs229.stanford.edu/projects2015.html#vision



# OpenCV in Mobile Apps & Web APIs?

- **API** - Create a RESTful API using DJANGO or FLASK to bundle your OpenCV code
  - **PROS**
    - Can be accessed on any mobile phone app or on a PC/MAC
    - Can be updated remotely (cloud based)
    - Powerful as processing is off loaded
  - **CONS**
    - Requires internet connection to transfer images
    - Will need to pay API cloud hosting
    - Can't do real time video based apps
- **Native Mobile** – Use OpenCV natively (using wrappers for iOS and Android)
  - **PROS**
    - Can do real time video based apps
    - No need to pay for cloud based storage
  - **CONS**
    - Can't do heavy processing on mobile device
    - More difficult than using OpenCV in python
- **iOS** - [http://docs.opencv.org/2.4/doc/tutorials/introduction/ios\\_install/ios\\_install.html](http://docs.opencv.org/2.4/doc/tutorials/introduction/ios_install/ios_install.html)
- **Android** - <http://blog.codeonion.com/2015/11/17/learning-the-packages-of-opencv-sdk-for-android/>

