PERIYAR UNIVERSITY

(NAAC 'A++' Grade with CGPA 3.61 (Cycle - 3)

State University - NIRF Rank 56 - State Public University Rank 25

SALEM - 636 011

CENTRE FOR DISTANCE AND ONLINE EDUCATION

(CDOE)

MASTER OF COMPUTER APPLICATIONS

SEMESTER - II

ELECTIVE – III: COMPUTER VISION

(Candidates admitted from 2024 onwards)

PERIYAR UNIVERSITY

CENTRE FOR DISTANCE AND ONLINE EDUCATION (CDOE)

MCA 2024 admission onwards

Elective Course – III

COMPUTER VISION

Prepared by:

Centre for Distance and Online Education (CDOE)
Periyar University
Salem - 636011

23PCAE06 - Computer Vision

Course Objectives:

- To get understanding about Computer vision techniques behind a wide variety of real- world applications.
- To get familiar with various Computer Vision fundamental algorithms and how to implement and apply.
- To get an idea of how to build a computer vision application with Python language.
- To understand various machine learning techniques that are used in computer vision tasks.

Unit-I

Basic Image Handling and Processing: PIL – the Python Imaging Library-Matplotlib- NumPy-SciPy-Advanced example: Image de-noising. **Local Image Descriptors:** Harris corner detector-SIFT - Scale-Invariant Feature Transform-Matching Geotagged Images.

Unit-II

Image to Image Mappings: Homographies-Warping images-Creating Panoramas. **Camera Models and Augmented Reality:** The Pin-hole Camera Model-Camera Calibration-Pose Estimation from Planes and Markers-Augmented Reality.

Unit-III

Multiple View Geometry: Epipolar Geometry-Computing with Cameras and 3D Structure- Multiple View Reconstruction-Stereo Images. **Clustering Images:** K-means Clustering- Hierarchical Clustering-Spectral Clustering.

Unit-IV

Searching Images: Content based Image Retrieval-Visual Words-Indexing Images-Searching the Database for Images-Ranking Results using Geometry-Building Demos and Web Applications. **Classifying Image Content:** K-Nearest Neighbors-Bayes Classifier- Support Vector Machines-Optical Character Recognition.

Unit-V

Image Segmentation: Graph Cuts-Segmentation using Clustering-Variational Methods.

OpenCV: Python Interface-OpenCV Basics-Processing Video-Tracking.

Text Book:

1**.** Programming Computer Vision with Python – Jan Erik Solem.

Reference Books:

1**.** Mastering OpenCV 4 with Python : A practical guide – Alberto Fernandez Villan.

LIST OF CONTENTS

UNIT – I

1.1 Basic Image Handling and Processing

Image processing in Python is a rapidly growing field with a wide range of applications. It is used in a variety of industries, including Computer vision, medical imaging, security, etc. Image processing is the field of study and application that deals with modifying and analyzing digital images using computer algorithms. While taking photographs is as simple as pressing a button, processing and improving those images sometimes takes more than a few lines of code. That"s where image processing libraries like OpenCV come into play. OpenCV is a popular open-source package that covers a wide range of image processing and computer vision capabilities and methods. It supports multiple programming languages including Python, C++, and Java.

The basic steps involved in digital image processing are:

1. **Image acquisition:** This involves capturing an image using a digital camera or scanner, or importing an existing image into a computer.
2. **Image enhancement:** This involves improving the visual quality of an image, such as increasing contrast, reducing noise, and removing artifacts.
3. **Image restoration:** This involves removing degradation from an image, such as blurring, noise, and distortion.
4. **Image segmentation:** This involves dividing an image into regions or segments, each of which corresponds to a specific object or feature in the image.
5. **Image representation and description:** This involves representing an image in a way that can be analyzed and manipulated by a computer, and describing the features of an image in a compact and meaningful way.
6. **Image analysis:** This involves using algorithms and mathematical models to extract information from an image, such as recognizing objects, detecting patterns, and quantifying features.
7. **Image synthesis and compression:** This involves generating new images or compressing existing images to reduce storage and transmission requirements.
8. Digital image processing is widely used in a variety of applications, including medical imaging, remote sensing, computer vision, and multimedia.

1.1.1 Python: Pillow (a fork of PIL)

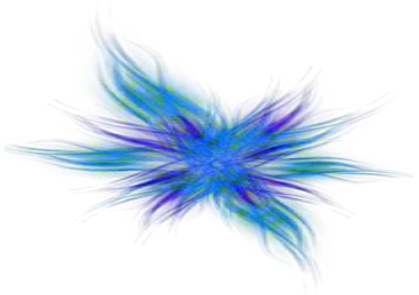
Python Imaging Library (expansion of PIL) is the de facto image processing package for Python language. It incorporates lightweight image processing tools that aids in editing, creating and saving images. Pillow was announced as a replacement for PIL for future usage. Pillow supports a large number of image file formats including BMP, PNG, JPEG, and TIFF. The library encourages adding support for newer formats in the library by creating new file decoders.

Beginning with Pillow

- **Opening an image using open():** The PIL Image. Image class represents the image object. This class provides the open() method that is used to open the image.

Example

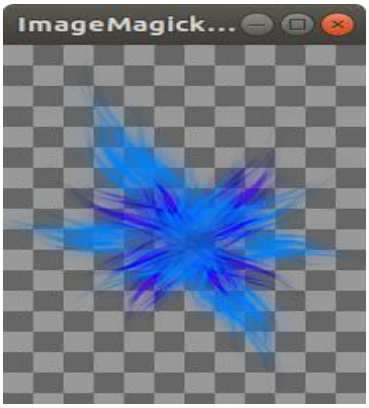
```
from PIL import Image
# test.png =>
location_of_image img =
Image.open(r"test.png")
```



- **Displaying the image using show():** This method is used to display the image. For displaying the image Pillow first converts the image to a .png format (on Windows OS) and stores it in a temporary buffer and then displays it. Therefore, due to the conversion of the image format to .png some properties of the original image file format might be lost (like animation). Therefore, it is advised to use this method only for test purposes.

Example

```
from PIL import Image
img =
Image.open(r"test.png")
img.show()
```



- **Obtaining information about the opened image**

Getting the mode (color mode) of the image: The mode attribute of the image tells the

A) type and depth of the pixel in the image. A 1-bit pixel has a range of 0-1, and an 8-bit pixel has a range of 0-255. There are different modes provided by this module. Few of them are:

Mode	Description
1	1-bit pixels, black and white
L	8-bit pixels, Greyscale
P	8-bit pixels, mapped to any other mode using a color palette
RGB	3×8-bit pixels, true color
RGBA	4×8-bit pixels, true color with transparency mask

Example

```
from PIL import Image
img =
Image.open(r"test.png")
print(img.mode)
```

Output

RBGA

B) Getting the size of the image: This attribute provides the size of the image. It returns a tuple that contains width and height.

Example

```
from PIL import Image
img =
Image.open(r"test.png")
print(img.size)
```

Output

(180, 263)

C) **Getting the format of the image:** This method returns the format of the image file.

Example

```
from PIL import Image
img =
Image.open(r"test.png")
print(img.format)
```

Output

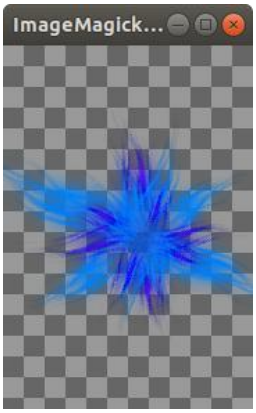
```
PNG
```

- **Rotating an image using rotate():** After rotating the image, the sections of the image having no pixel values are filled with black (for non-alpha images) and with completely transparent pixels.

Example

```
from PIL import
Image angle = 40
img =
Image.open(r"test.png")
r_img = img.rotate(angle)
```

Output



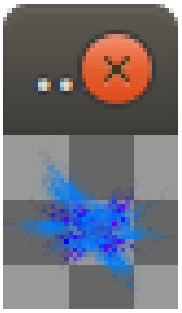
- **Resizing an image using resize():** Interpolation happens during the resize process, due to which the quality of image changes whether it is being upscaled (resized to a higher dimension than original) or downscaled (resized to a lower Image then original). Therefore resize() should be used cautiously and while providing suitable value for resampling argument.

Example

```
from PIL import
Image size = (40, 40)
```

```
img = Image.open(r"test.png")
r_img =
img.resize(size)
r_img.show(
```

Output



- **Saving an image using save():** While using the save() method Destination_path must have the image filename and extension as well. The extension could be omitted in Destination_path if the extension is specified in the format argument.

Example

```
from PIL import Image
size = (40, 40)
img = Image.open(r"test.png")
r_img = img.resize(size, resample =
Image.BILINEAR) # resized_test.png =>
Destination_path r_img.save("resized_test.png")
# Opening the new image
img =
Image.open(r"resized_test.png")
print(img.size)
```

Output

```
(40, 40)
```

1.1.2 Matplotlib, NumPy, SciPy

Matplotlib is an amazing visualization library in Python for 2D plots of arrays. Matplotlib is a multi-platform data visualization library built on NumPy arrays and designed to work with the broader SciPy stack. It was introduced by John Hunter in the year 2002. One of the greatest benefits of visualization is that it allows us visual access to huge amounts of data in easily digestible visuals. Matplotlib consists of several plots like line, bar,

scatter, histogram, etc.

Types of Matplotlib

Matplotlib comes with a wide variety of plots. Plots help to understand trends, and patterns, and to make correlations. They"re typically instruments for reasoning about quantitative information. Some of the sample plots are covered here.

- Matplotlib Line Plot
- Matplotlib Bar Plot
- Matplotlib Histograms Plot
- Matplotlib Scatter Plot
- Matplotlib Pie Charts
- Matplotlib Area Plot

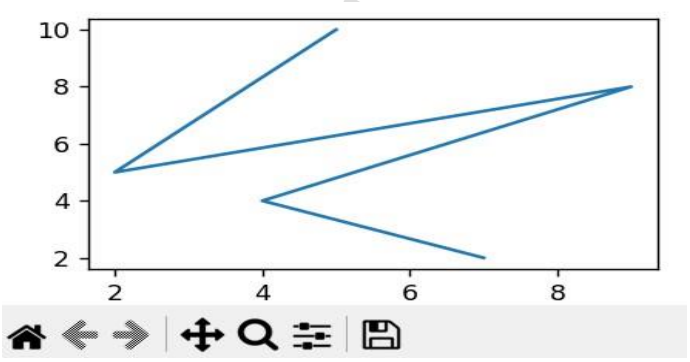
Matplotlib Line Plot

By importing the matplotlib module, defines x and y values for a plotPython, plots the data using the plot() function and it helps to display the plot by using the show() function . The plot() creates a line plot by connecting the points defined by x and y values.

Example

```
# importing matplotlib module
from matplotlib import pyplot as
plt # x-axis values
x = [5, 2, 9, 4, 7]
# Y-axis values
y = [10, 5, 8, 4, 2]
# Function to
plot plt.plot(x, y)
# function to show the
plot plt.show()
```

Output



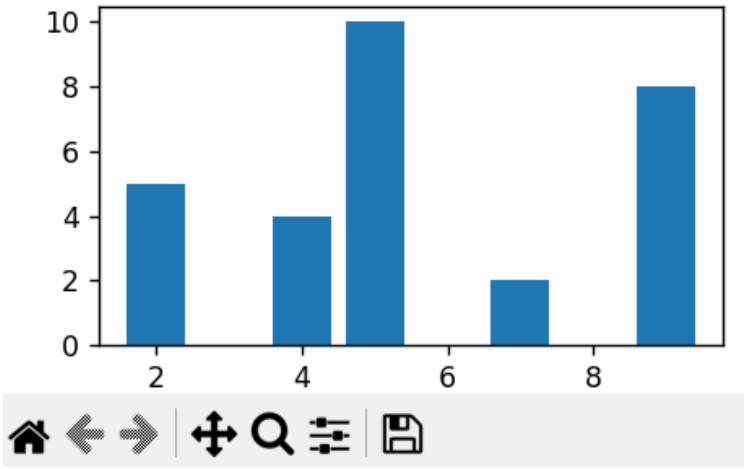
Matplotlib Bar Plot

By using matplotlib library in Pythontwo, it allows us to access the functions and classes provided by the library for plotting. There are tow lists x and y are defined . This function creates a bar plot by taking x-axis and y-axis values as arguments and generates the bar plot based on those values.

Example

```
# importing matplotlib module
from matplotlib import pyplot as
plt # x-axis values
x = [5, 2, 9, 4, 7]
# Y-axis values
y = [10, 5, 8, 4, 2]
# Function to plot the
bar plt.bar(x, y)
# function to show the
plot plt.show()
```

Output



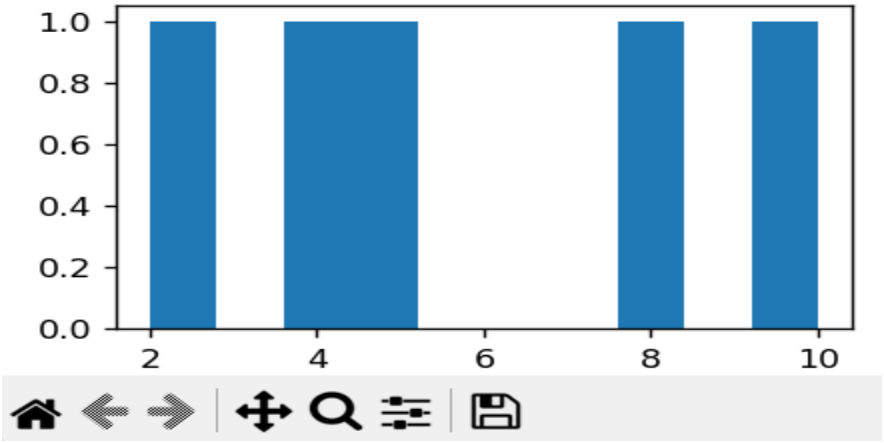
Matplotlib Histograms Plot

By using the matplotlib module defines the y-axis values for a histogram plot. Plots in the ,histogram using the hist() function and displays the plot using the show() function. The hist() function creates a histogram plot based on the values in the y-axis list.

Example

```
# importing matplotlib module
from matplotlib import pyplot as
plt # Y-axis values
y = [10, 5, 8, 4, 2]
# Function to plot
histogram plt.hist(y)
# Function to show the
plot plt.show()
```

Output



Matplotlib Scatter Plot

By imports,plot the matplotlib module, defines x and y values for a scatter plot, plots the data using the scatter() function, and displays the plot using the show() function. The scatter() function creates a scatter plot by plotting individual data points defined by the x and y values.

Example

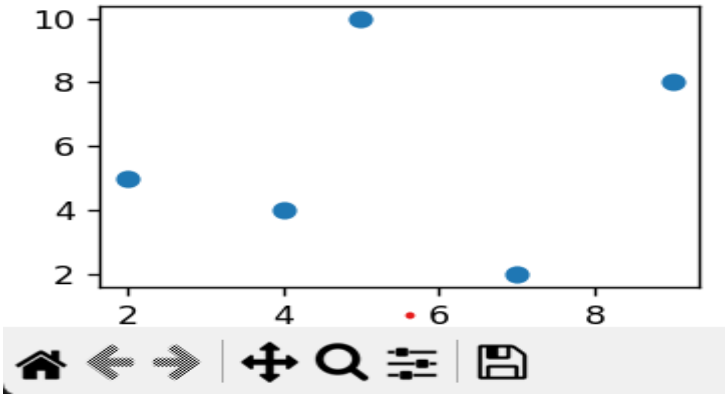
```
# importing matplotlib module
from matplotlib import pyplot as
plt # x-axis values
x = [5, 2, 9, 4, 7]
# Y-axis values
y = [10, 5, 8, 4, 2]
# Function to plot
```

scatter plt.scatter(x, y)

function to show the

plot plt.show()

Output



Matplotlib Pie Charts

By importing the module Matplotlib in Python to create a pie chart with three categories and respective sizes. The plot .pie() function is used to generate the chart, including labels, percentage formatting, and a starting angle.

Example

import matplotlib.pyplot as

plt # Data for the pie chart

labels = ['Maths', 'Physics',

'Chemistry'] sizes = [35, 35, 30]

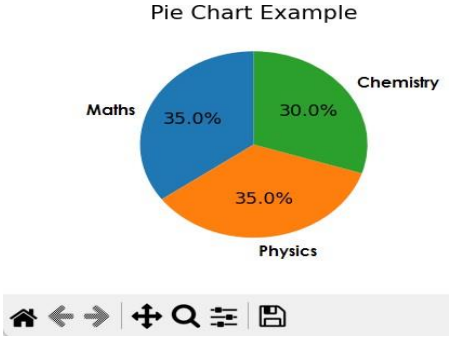
Plotting the pie chart

plt.pie(sizes, labels=labels, autopct='%1.1f%%',

startangle=90) plt.title('Pie Chart Example')

plt.show()

Output



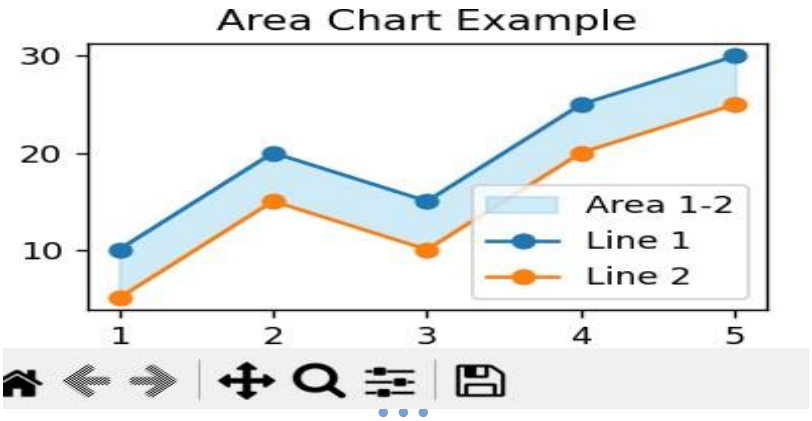
Matplotlib Area Plot

By importing Matplotlib we sky bluegenerated an area chart with two lines („Line 1" and „Line 2"). The area between the lines is shaded in a skyblue color with 40% transparency. The x-axis values are in the list „x", and the corresponding y-axis values for each line are in „y1" and „y2". Labels, titles legends, and legend are added, and the resulting area chart is displayed.

Example

```
import matplotlib.pyplot as
plt # Data
x = [1, 2, 3, 4, 5]
y1, y2 = [10, 20, 15, 25, 30], [5, 15, 10, 20, 25]
# Area Chart
plt.fill_between(x, y1, y2, color='skyblue', alpha=0.4, label='Area
1-2') plt.plot(x, y1, label='Line 1', marker='o')
plt.plot(x, y2, label='Line 2',
marker='o') # Labels and Title
plt.xlabel('X-axis'), plt.ylabel('Y-axis'), plt.title('Area Chart
Example') # Legend and Display
plt.legend(), plt.show()
```

Output



NumPy

Numpy is a general-purpose array-processing package. It provides a high-performance multidimensional array object, and tools for working with these arrays. It is the fundamental package for scientific computing with Python.

Features of NumPy

NumPy has various features including these important ones:

- A powerful N-dimensional array object
- Sophisticated (broadcasting) functions
- Tools for integrating C/C++ and Fortran code
- Useful linear algebra, Fourier transform, and random number capabilities

NumPy Array Creation

There are various ways of Numpy array creation in Python.

If we can create an array from a regular Python list or tuple using the array() function. The type of the resulting array is deduced from the type of the elements in the sequences.

Example

```
import numpy as np
# Creating array from list with type float
a = np.array([[1, 2, 4], [5, 8, 7]], dtype =
'float') print ("Array created using passed
list:\n", a)
# Creating array from
tuple b = np.array((1 , 3,
2))
print ("\nArray created using passed tuple:\n", b)
```

Output

```
Array created using passed
 list: [[1. 2. 4.]
 [5. 8. 7.]]
Array created using passed
 tuple: [1 3 2]
```

Arrange: This function returns evenly spaced values within a given interval. Step size is specified.

Example

```
# Create a sequence of
integers # from 0 to 30 with
steps of 5
```

```
f = np.arange(0, 30, 5)
print ("A sequential array with steps of 5:\n", f)
```

Output

```
A sequential array with steps of 5:
[ 0 5 10 15 20 25]
```

linspace: It returns evenly spaced values within a given interval.

Example

```
# Create a sequence of 10 values in range
0 to 5 g = np.linspace(0, 5, 10)
print ("A sequential array with 10 values between" "0 and 5:\n", g)
```

Output

```
A sequential array with 10 values between0 and 5:
[0.         0.55555556 1.11111111 1.66666667 2.22222222 2.77777778
 3.33333333 3.88888889 4.44444444 5. ]
```

Reshaping array: We can use **reshape** method to reshape an array. Consider an array with shape (a1, a2, a3, &#x2026, aN). We can reshape and convert it into another array with shape (b1, b2, b3, &#x2026, bM). The only required condition is a1 x a2 x a3 &#x2026 x aN = b1 x b2 x b3 &#x2026 x bM. (i.e. the original size of the array remains unchanged.)

Example

```
# Reshaping 3X4 array to 2X2X3
array arr = np.array([[1, 2, 3, 4],
                      [5, 2, 4, 2],
                      [1, 2, 0, 1]])
newarr = arr.reshape(2, 2,
3) print ("Original array:\n",
arr) print("..........")
print ("Reshaped array:\n", newarr)
```

Output

```
Original
array: [[1 2 3
4]
 [5 2 4 2]
```

[1 2 0 1]]

Reshaped array:

[[[1 2 3]

[4 5 2]]

[[4 2 1]

[2 0 1]]]

Flatten array: We can use **flatten** method to get a copy of the array collapsed into **one dimension**. It accepts *order* argument. The default value is &#x2018C&#x2019 (for row-major order). Use &#x2018F&#x2019 for column-major order.

Example

```
# Flatten array
arr = np.array([[1, 2, 3], [4, 5,
6]]) flat_arr = arr.flatten()
print ("Original array:\n", arr)
print ("Fattened array:\n",
flat_arr)
```

Output

Original

array: [[1 2 3]

[4 5 6]]

Fattened

array: [1 2 3

4 5 6]

Both NumPy and SciPy are Python libraries used for used mathematical and numerical analysis. NumPy contains array data and basic operations such as sorting, indexing, etc whereas, SciPy consists of all the numerical code. Though NumPy provides a number of functions that can help resolve linear algebra, Fourier transforms, etc, SciPy is the library that actually contains fully- featured versions of these functions along with many others. However, if you are doing scientific analysis using Python, you will need to install both NumPy and SciPy since SciPy builds on NumPy.

NumPy vs SciPy

Types of Differences	NumPy	SciPy
Primary Focus	NumPy primarily focuses on providing efficient array manipulation and fundamental numerical operations.	On the other hand, SciPy contains all the functions that are present in NumPy to some extent.
Use Cases	NumPy is often used when you need to work with arrays, and matrices, or perform basic numerical operations. It is commonly used in tasks like data manipulation, linear algebra, and basic mathematical computations.	SciPy becomes essential for tasks like solving complex differential equations, optimizing functions, conducting statistical analysis, and working with specialized mathematical functions.
Module Structure	NumPy provides a single, comprehensive library for array manipulation and basic numerical operations. It doesn"t have a modular structure like SciPy.	SciPy is organized into submodules, each catering to a specific scientific discipline. This modular structure makes it easier to find and use functions relevant to your specific scientific domain.
Capabilities	* Efficient storage of data	* Multidimensional image processing.
	* Vectorization arithmetic	*Advanced optimization routines using “optimize”.

	* Broadcasting mechanisms to handle arrays of different shapes during mathematical operations.	* Special functions through its “special module.
Domain	Elementary linear algebra.	Spatial data structure and algorithm
	Basic statistical functions.	Interpolation functions with interpolate.
	Fourier analysis.	Eigenvalue problems and matrix functions.
	Random number capabilities.	Sparse matrix computations.
Evolution	NumPy is originated from the older Numeric and Numarray libraries. It was designed to provide an efficient array computing utility for Python.	Scipy is started with Travis Oliphant wanting to combine the functionalities of Numeric and another library called “scipy.base”. The result was the more comprehensive and integrated library we know today.

Subpackages in SciPy:

SciPy has a number of subpackages for various scientific computations which are shown in the following table:

Name	Description
cluster	Clustering algorithms
constants	Physical and mathematical constants
fftpack	Fast Fourier Transform routines
integrate	Integration and ordinary differential equation solvers
interpolate	Interpolation and smoothing splines
io	Input and Output
linalg	Linear algebra
ndimage	N-dimensional image processing
odr	Orthogonal distance regression
optimize	Optimization and root-finding routines
signal	Signal processing
sparse	Sparse matrices and associated routines
spatial	Spatial data structures and algorithms
special	Special functions
stats	Statistical distributions and functions

Interaction with NumPy:

SciPy builds on NumPy and therefore you can make use of NumPy functions itself to handle arrays. To know in-depth about these functions, you can simply make use of help(), info() or source() functions.

help():

To get information about any function, you can make use of the ***help()*** function. There are two ways in which this function can be used:

- without any parameters
- using parameters

info():

This function returns information about the desired functions, modules, etc.

Example: scipy.info(cluster)

source():

The source code is returned only for objects written in Python. This function does not return useful information in case the methods or objects are written in any other language such as C.

Syntax: scipy.source(cluster)

Example	**Output**
from scipy import special	1000.0
a = special.exp10(3)	8.0
print(a)	1.0
b = special.exp2(3)	0.7071067811865475
print(b)	
c = special.sindg(90)	
print(c)	
d = special.cosdg(45)	
print(d)	

Integration Functions:

SciPy provides a number of functions to solve integrals. Ranging from ordinary differential integrator to using trapezoidal rules to compute integrals, SciPy is a storehouse of functions to solve all types of integrals problems.

General Integration:

SiPy provides a function named ***quad*** to calculate the integral of a function which has one variable. The limits can be ±∞ (± inf) to indicate infinite limits.

Example	**Output**
from scipy import special a= lambda x:special.exp10(x) b = scipy.integrate.quad(a, 0, 1) print(b)	(3.9086503371292665, 4.3394735994897923e-14) from scipy import integrate

Double Integral Function:

SciPy provides ***dblquad*** that can be used to calculate double integrals. A double integral, as many of us know, consists of two real variables. The dblquad() function will take

the function to be integrated as its parameter along with 4 other variables which define the limits and the functions *dy* and *dx.*

Example **Output**

```
from scipy import integrate                    -1.3333333333333335,
1.4802973661668755e-14) a = lambda y, x: x*y**2
b = lambda x:
1 c = lambda
x: -1
integrate.dblquad(a, 0, 2, b, c)
```

Interpolation Functions:

In the field of numerical analysis, interpolation refers to constructing new data points within a set of known data points. The SciPy library consists of a subpackage named scipy.interpolate that consists of spline functions and classes, one-dimensional and multi-dimensional (univariate and multivariate) interpolation classes, etc.

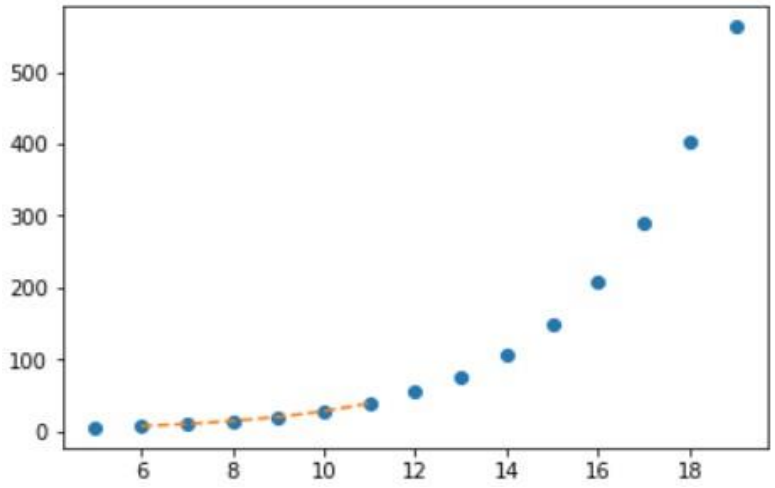
Univariate Interpolation:

Univariate interpolation is basically an area of curve-fitting which finds the curve that provides an exact fit to a series of two-dimensional data points. SciPy provides *interp1d* function that can be utilized to produce univariate interpolation.

Example

```
import matplotlib.pyplot as
plt from scipy import
interpolate x =
np.arange(5, 20)
y = np.exp(x/3.0)
f = interpolate.interp1d(x, y)x1 = np.arange(6, 12)
y1 = f(x1) # use interpolation function returned by
`interp1d` plt.plot(x, y, 'o', x1, y1, '--')
plt.show()
```

Output



Multivariate Interpolation:

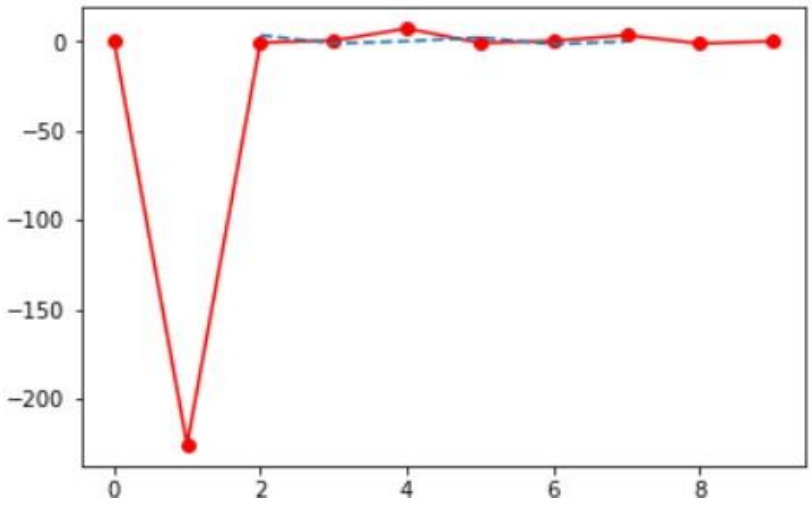
Multivariate interpolation (spatial interpolation) is a kind interpolation on functions that consist of more than one variables. The following example demonstrates an example of the *interp2d* function.

Interpolating over a 2-D grid using the interp2d(x, y, z) function basically will use x, y, z arrays to approximate some function *f: "z = f(x, y)"* and returns a function whose call method uses *spline interpolation* to find the value of new points.

Example

```
from scipy import
interpolate import
matplotlib.pyplot as plt x =
np.arange(0,10)
y = np.arange(10,25)
x1, y1 = np.meshgrid(x,
y) z = np.tan(xx+yy)
f = interpolate.interp2d(x, y, z,
kind='cubic') x2 = np.arange(2,8)
y2 =
np.arange(15,20)
z2 = f(xnew, ynew)
plt.plot(x, z[0, :], 'ro-', x2, z2[0, :], '-
-') plt.show()
```

Output



1.1.3 Denoising techniques

Denoising is the process of removing or reducing the noise or artifacts from the image. Denoising makes the image more clear and enables us to see finer details in the image clearly. It does not change the brightness or contrast of the image directly, but due to the removal of artifacts, the final image may look brighter.

In this denoising process, we choose a 2-D box and slide it over the image. The intensity of each and every pixel of the original image is recalculated using the box.

Box averaging technique:

Box averaging can be defined as the intensity of the corresponding pixel would be replaced with the average of all intensities of its neighbour pixels spanned by the box. This is a point operator.

Example

MATLAB

```
% MATLAB code for Box averaging
% Read the cameraman
image.
k1=imread("cameraman.jpg
");
% create the noise of standard
deviation 25 n=25*randn(size(k1));
%add the noise to the
image=noisy_image k2=double(k1)+n;
```

```
%display the noisy
image. imtool(k2,[]);
%averaging using [5 5] sliding box.
k3=uint8(colfilt(k2,[5 5], 'sliding',
@mean));
%display the denoised
image. imtool(k3,[]);
%averaging using [9 9] sliding box.
k4=uint8(colfilt(k2, [9 9], 'sliding', @mean));
%display the denoised
image. imtool(k4,[]);
```

Output

Gaussian Filter:

This kernel or filter has more weightage for the central pixel. While averaging at the edges, more weightage is given to the edged pixel and thus it gives us the pixel value close to the actual one, therefore, reduces the blurriness at the edges.

Example

MATLAB

```
% MATLAB code for denoised using
% Gaussian Filter:
```

```
k1=imread("cameraman.jpg
");
% create the noise.
n=25*randn(size(k1)
);
% add the noise to the image =
noisy_image k2=double(k1)+n;
%create and print the kernel of size
[3 3] h1=fspecial('gaussian',3,1);
h1
% convulse the image with the
kernel. k3=uint8(conv2(k2,
h1,'same'));
% display the denoised
image. imtool(k3,[]);
% create and print the kernel of size [20
20] h2=fspecial('gaussian',20,1);
h2
% convulse the image with the kernel. k4=uint8(conv2(k2,h2,'same'));
% display the denoised
image. imtool(k4,[]);
```

Output

Denoising by averaging noisy images:

This is a very simple and interesting technique of denoising. The requirement for using this technique is that:

- We should have 2 or more images of the same scene or object.
- The noise of the image capturing device should be fixed. For example, the camera has a noise of a standard deviation of 20.

Working:

Collect the multiple images captured by the same device and of the same object. Just take the average of all the images to get the resultant image. The intensity of every pixel will be replaced by the average of the intensities of the corresponding pixel in all those collected images. This technique will reduce the noise and also there would not be any blurriness in the final image.

Example

MATLAB

```
% MATLAB code for denoising by averaging
% Read the cameraman image: original
image. I=imread("cameraman.jpg");
% Create noise-1 of
std=40
n140*randn(size(I));
% Create first noisy_image by adding the noise to orig
image. I1=double(I)+n1;
% Create noise-2 of
std=40
n2=40*randn(size(I));
% Create 2nd noisy_image by adding the noise to orig
image. I2=double(I)+n2;
% Create noise-3 of
std=40
n3=40*randn(size(I));
% Create 3rd noisy_image by adding the noise to orig
```

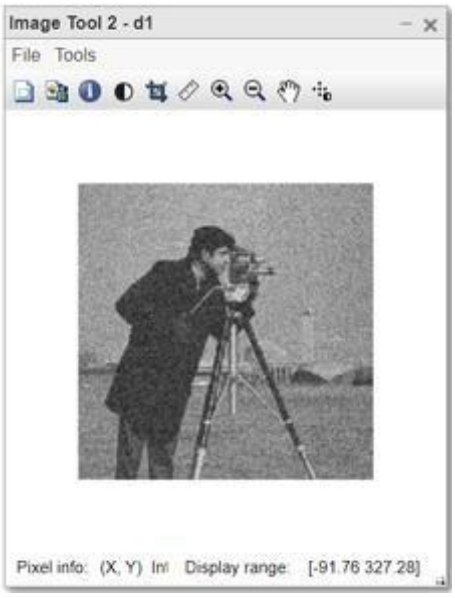
• • •

```
image. I3=double(I)+n3;
```

```
% Create noise-4 of
std=40
n4=40*randn(size(I));
% Create 4th noisy_image by adding the noise to orig
image. I4=double(I)+n4;
% Create noise-5 of
std=40
n5=40*randn(size(I));
image. I5=double(I)+n5;
% Now lets see
denoising. d1=(I1+I2)/2;
d2=(I1+I2+I3)/3;
d3=(I1+I2+I3+I4)/4;
d4=(I1+I2+I3+I4+I5)/5;
%display each denoised image with original noisy
image. imtool(I1,[]);
imtool(d1,[]);
imtool(d2,[]);
imtool(d3,[]);
imtool(d4,[]);
```

Output

Noisy_image and Denoised-1



Noisy_image and Denoised-2

1.2 Image Descriptors

1.2.1 Harris corner detector

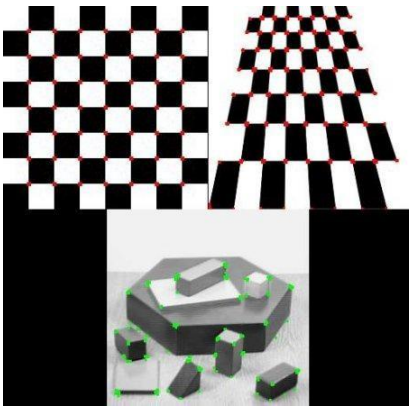
Harris Corner detection algorithm was developed to identify the internal corners of an image. The corners of an image are basically identified as the regions in which there are variations in large intensity of the gradient in all possible dimensions and directions. Corners extracted can be a part of the image features, which can be matched with features of other images, and can be used to extract accurate information. Harris Corner Detection is a method to extract the corners from the input image and to extract features from the input image.

Example

```
import numpy as
np import cv2 as
cv
filename =
'chessboard.png' img =
cv.imread(filename)
gray =
cv.cvtColor(img,cv.COLOR_BGR2GRAY)
```

```
gray = np.float32(gray)
dst = cv.cornerHarris(gray,2,3,0.04)

dst = cv.dilate(dst,None)
# Threshold for an optimal value, it may vary depending on the
image. img[dst>0.01*dst.max()]=[0,0,255]
cv.imshow('dst',img)

if cv.waitKey(0) & 0xff ==
27: cv.destroyAllWindows()
```

Output:

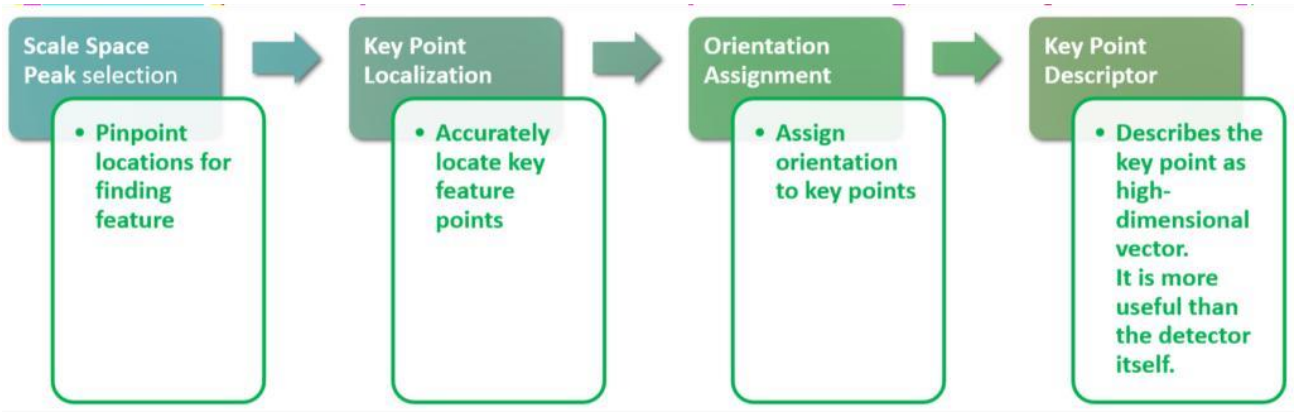


1.2.2 Scale Invariant Feature Transform

SIFT (Scale Invariant Feature Transform) Detector is used in the detection of interest points on an input image. It allows the identification of localized features in images which is essential in applications such as:

- Object Recognition in Images
- Path detection and obstacle avoidance algorithms
- Gesture recognition, Mosaic generation, etc.

Fig : Sequence of steps followed in SIFT Detector



Phase I: Scale Space Peak Selection

The concept of Scale Space deals with the application of a continuous range of Gaussian Filters to the target image such that the chosen Gaussian have differing values of the sigma parameter. The plot thus obtained is called the Scale Space. Scale Space Peak Selection depends on the Spatial Coincidence Assumption. According to this, if an edge is detected at the same location in multiple scales (indicated by zero crossings in the scale space) then we classify it as an actual edge.

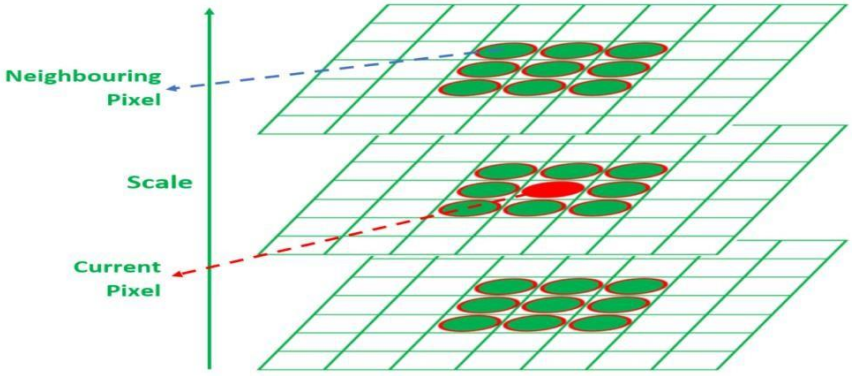


Fig: Peaks are selected across Scales.

In 2D images, we can detect the Interest Points using the local maxima/minima in **Scale Space of Laplacian of Gaussian.** A potential SIFT interest point is determined for a given sigma value by picking the potential interest point and considering the pixels in the level above (with higher sigma), the same level, and the level below (with lower sigma than current sigma level). If the point is maxima/minima of all these 26 neighboring points, it is a potential SIFT interest point – and it acts as a starting point for interest point detection.

Phase II: Key Point Localization

Key point localization involves the refinement of keypoints selected in the previous stage. Low contrast key-points, unstable key points, and keypoints lying on edges are eliminated. This is achieved by calculating the Laplacian of the keypoints found in the previous stage. The extrema values are computed as follows:

$$z = -\frac{\partial^2 D^{-1}}{\partial x^2}\frac{\partial D}{\partial x}$$

In the above expression, D represents the Difference of Gaussian. To remove the unstable key points, the value of *z* is calculated and if the function value at z is below a threshold value then the point is excluded.

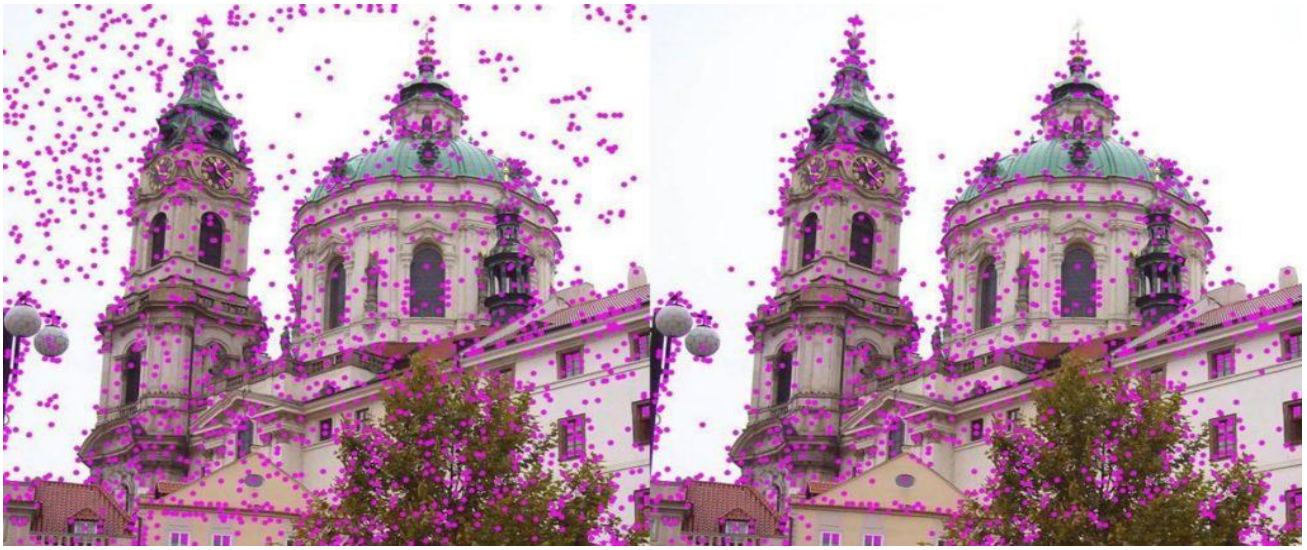


Fig: Refinement of Keypoints after Keypoint Localization

Phase III: Assigning Orientation to Keypoints

To achieve detection which is invariant with respect to the rotation of the image, orientation needs to be calculated for the key-points. This is done by considering the neighborhood of the keypoint and calculating the magnitude and direction of gradients of the neighborhood. Based on the values obtained, a histogram is constructed with 36 bins to represent 360 degrees of orientation(10 degrees per bin). Thus, if the gradient direction of a

certain point is, say, 67.8 degrees, a value, proportional to the gradient magnitude of this point, is added to the bin representing 60-70 degrees. Histogram peaks above 80% are converted into a new keypoint are used to decide the orientation of the original keypoint.

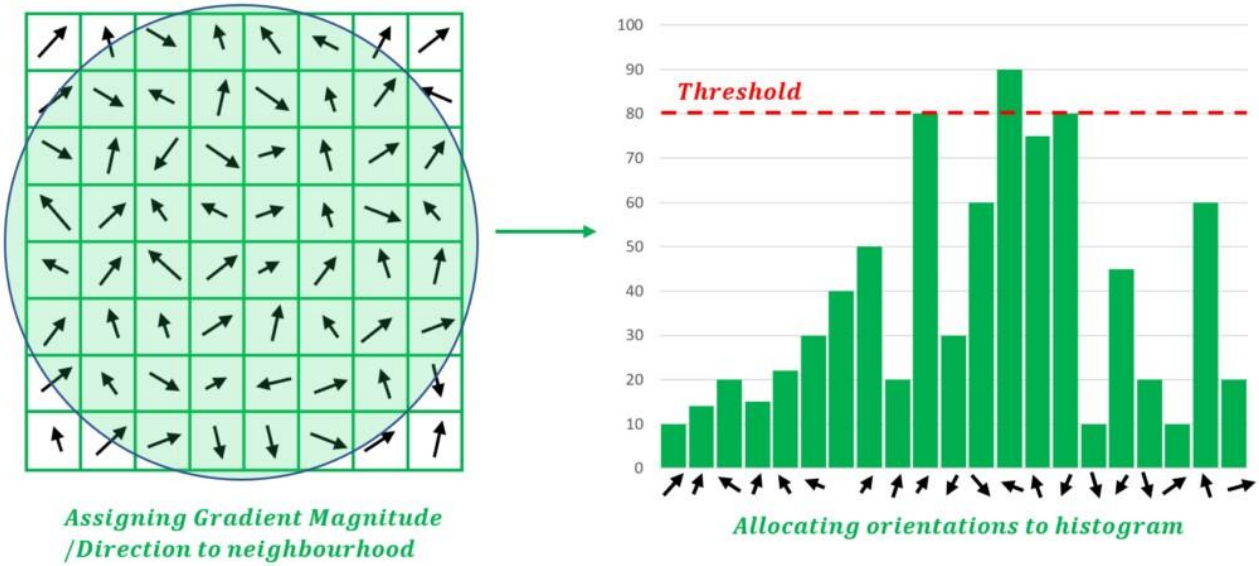


Fig: Assigning Orientation to Neighborhood and creating Orientation Histogram

Phase IV: Key Point Descriptor

Finally, for each keypoint, a descriptor is created using the keypoints neighborhood. These descriptors are used for matching keypoints across images. A 16×16 neighborhood of the keypoint is used for defining the descriptor of that key-point. This 16×16 neighborhood is divided into sub- block. Each such sub-block is a non-overlapping, contiguous, 4×4 neighborhood. Subsequently, for each sub-block, an 8 bin orientation is created similarly as discussed in Orientation Assignment. These 128 bin values (16 sub-blocks * 8 bins per block) are represented as a vector to generate the keypoint descriptor.

Example: SIFT detector in Python

```
# Important NOTE: Use opencv >=4.4 import cv2
# Loading the image
img = cv2.imread('geeks.jpg')
# Converting image to grayscale
gray=cv2.cvtColor(img,cv2.COLOR_BGR2GRAY) # Applying SIFT detector
sift = cv.SIFT_create()
kp = sift.detect(gray, None)
# Marking the keypoint on the image using circles img=cv2.drawKeypoints(gray ,
```

kp , img ,

flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)

cv2.imwrite('image-with-keypoints.jpg', img)

Output

Fig: The image on left is the original, the image on right shows the various highlighted interest points on the image

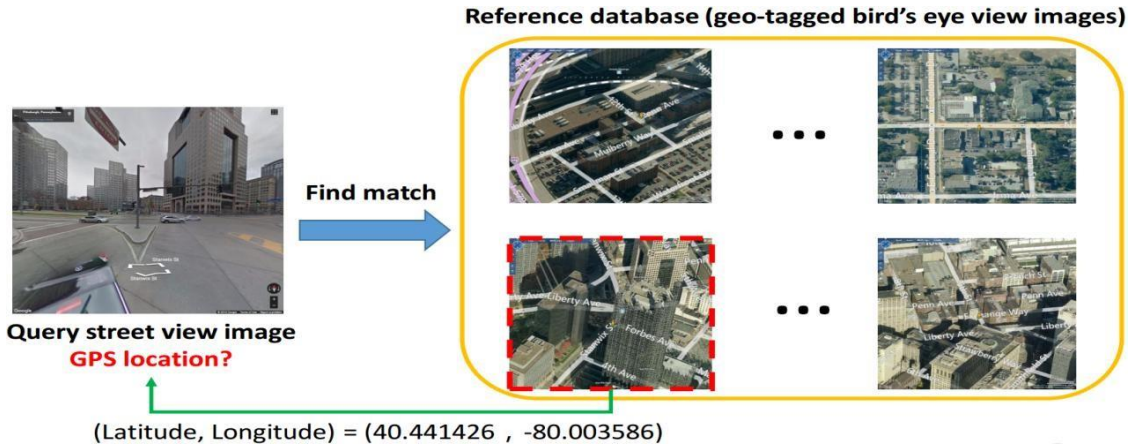
1.2.3 Matching Geotagged Images

Geotagged images are images with GPS coordinates either added manually by the photographer or automatically by the camera.

Problem & Motivation

The goal of this effort is to develop a novel method which automatically finds the geo- location of an image with an accuracy comparable to GPS devices. In most image matching based geo-localization methods, the geo-location of a query image is obtained by finding its matching reference images from the same view (e.g. street view images), assuming that a reference dataset consisting of geo-tagged images is available. However, since only small number of cities in the world are covered by ground-level imagery, it has not been feasible to scale up ground-level image- to-image matching approaches to global level.

On the other hand, a more complete coverage for overhead reference data such as satellite/areial imagery and digital elevation model (DEM) is available. Therefore, an alternative is to predict the geo-location of a query image by finding its matching reference images from some other views. For example, predict the geo-location of a query street view image based on a reference database of bird"s eye view images, or vice versa.



Method

To present a new framework for cross-view image geolocalization. First, we employ the Faster R-CNN [1] to detect buildings in the query and reference images. Next, for each building in the query image, we retrieve the k nearest neighbors from the reference buildings using a Siamese network trained on both positive matching image pairs and negative pairs. To find the correct NN for each query building, we develop an efficient multiple nearest neighbors matching method based on dominant sets. The final geo-localization result is obtained by taking the mean GPS location of selected reference buildings in the dominant set.

Figure: The pipeline of the proposed cross-view geo-localization method.

Building Detection

To find the matching image or images in the reference database for a query image, we resort to match buildings between cross-view images since the semantic information of images is more robust to viewpoint variations than appearance features. Therefore, the first step is to detect buildings in images. We employ the Faster R-CNN [1] to achieve this goal due to its state-of-the-art performance for object detection and real-time execution. In our application, the detected buildings in a query image serve as query buildings for retrieving the matching buildings in the reference images. Figure 3 shows examples of the building detection results in both street view and bird"s eye view images. Each detected bounding box is assigned a score.

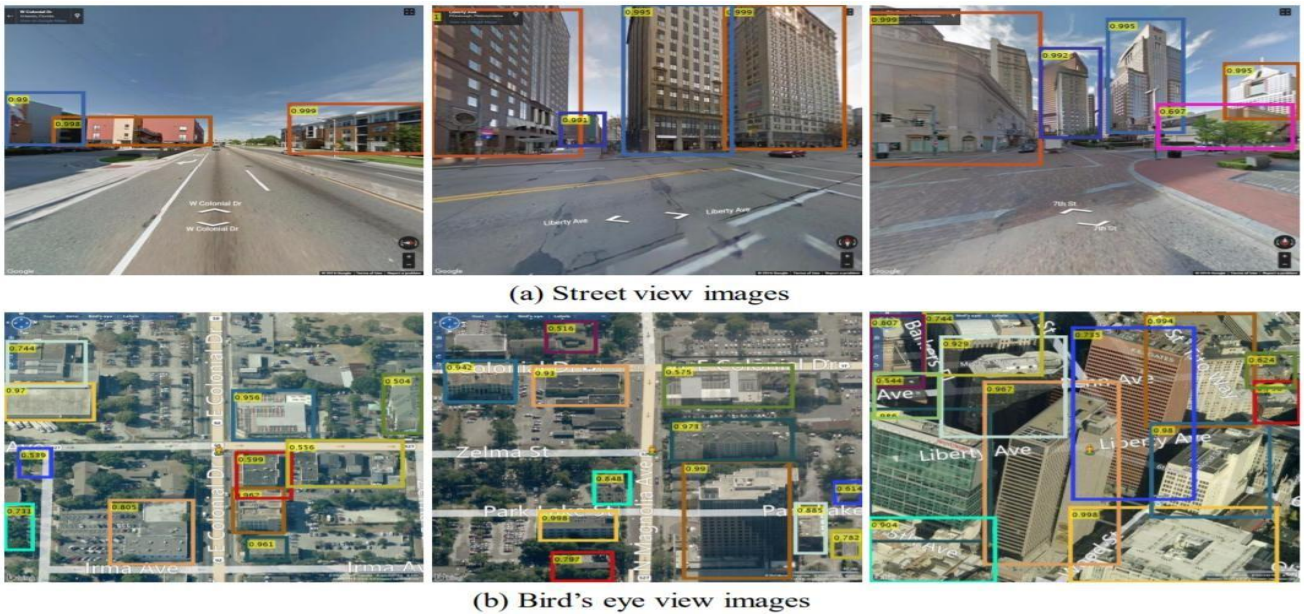
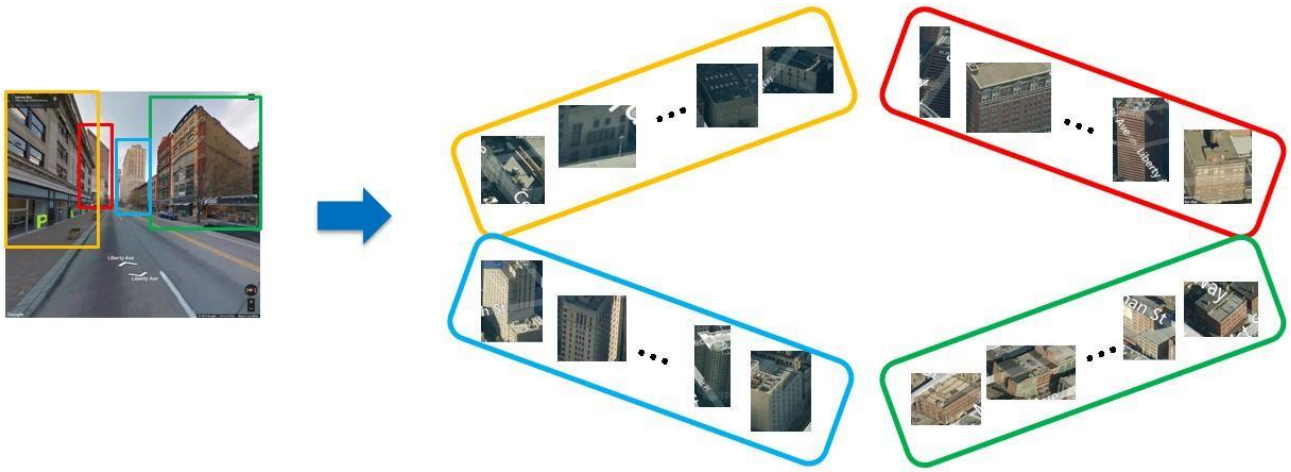


Figure: Building detection examples using Faster

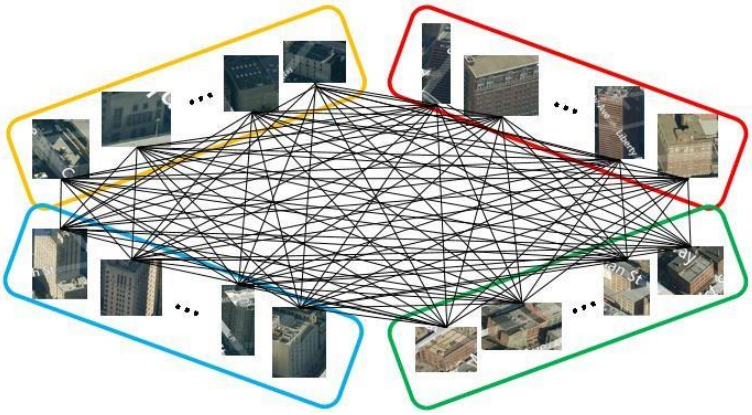
R-CNN. Geo-localization Using Dominant Sets

A simple approach for geo-localization will be, for each detected building in the query image, take the GPS location of its nearest neighbor (NN) in reference images, according to building matching. However, this will not be optimal. In fact, in most cases the nearest neighbor does not correspond to the correct match. Therefore, besides local matching (matching individual buildings), we introduce a global constraint to help make better geo-localization decision. In a given query image, typically there are multiple buildings and their GPS locations should be close. Therefore, the GPS locations of their matched buildings should be close as well.

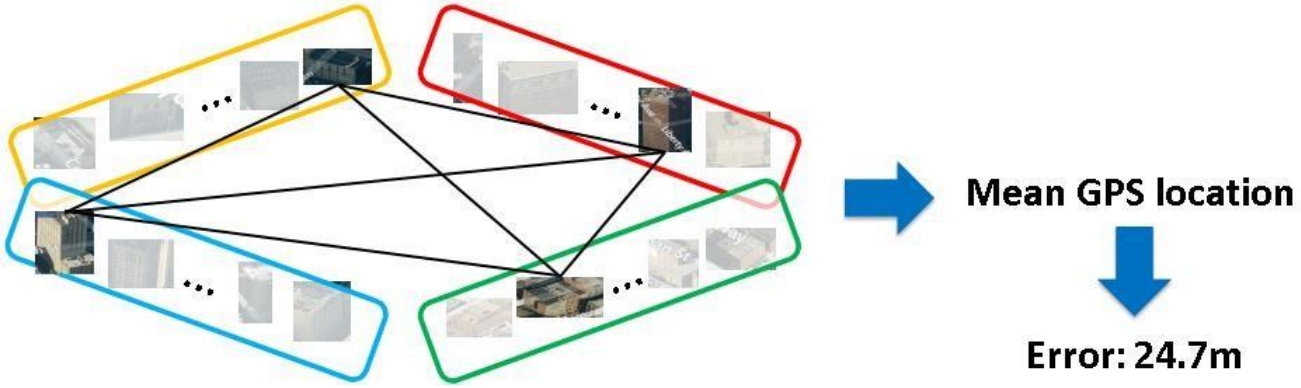


For each building in a query image, select k NNs from reference images

- Build a graph $G = (V, E, w)$



$$w_{ij} = \frac{1}{2}(e^{-\frac{d_{ij}^2}{2\sigma^2}} + \alpha(s_i + s_j))$$



- Dominant set [3] selection

Dataset

To explore the geo-localization task using cross-view image matching, we have collected a new dataset of street view and bird"s eye view image pairs around downtown Pittsburg, Orlando and part of Manhattan. For this dataset we use the list of GPS coordinates from Google Street View Dataset [4]. There are $1,586$, $1,324$ and $5,941$ GPS locations in Pittsburg, Orlando and Manhattan, respectively. We utilize DualMaps to generate side-by-side street view and bird"s eye view images at each GPS location with the same heading direction. The street view images are from Google and the overhead 45° bird"s eye view images are from Bing. For each GPS location, four image pairs are generated with camera heading directions of 0°, 90°, 180° and 270°. In order to learn the deep network for building matching, we annotate corresponding buildings in every street view and bird"s eye view image pair.

Fig: Sampled GPS locations in Pittsburg, Orlando and part of Manhattan.

Unit Summary

Basic image handling and processing involves manipulating and analyzing digital images using various techniques and tools. At its core, image handling encompasses tasks such as loading, displaying, and saving images in different formats. Formats like JPEG, PNG, and GIF each have distinct properties affecting image quality and file size, influencing their suitability for various applications. Image processing techniques are applied to enhance or extract information from images. Common operations include adjusting brightness and contrast, resizing, cropping, and rotating images to meet specific requirements or improve visual appeal. These adjustments can be crucial for preparing images for publication, analysis, or presentation.

Filtering is a fundamental processing technique used to modify or enhance image features. Filters such as blurring, sharpening, and edge detection help in reducing noise, highlighting important details, or preparing images for further analysis. For example, blurring filters can smooth out details to reduce noise, while sharpening filters enhance edge clarity and contrast.

Color space conversion is another important aspect of image processing, involving the transformation of images from one color model to another, such as RGB (Red, Green, Blue) to grayscale or HSV (Hue, Saturation, Value). This conversion can be useful for various applications, including image analysis and compression. Histogram equalization is a technique used to improve the contrast of an image by adjusting its intensity distribution. By spreading out the most frequent intensity values, this method enhances the visibility of features that might be less distinguishable in the original image.

Self assessment Questions:

1. Explain PIL (Python Imaging Library).
2. Briefly Explain Matplotlib.
3. Illustration NumPy.
4. Explain SciPy.
5. Illustration The Concept of Image De-Noising.
6. Briefly Explain Harris Corner Detector.
7. Explain SIFT (Scale-Invariant Feature Transform).
8. Briefly Explain Matching Geotagged Images.

Glossary

Resolution: The amount of detail an image holds, typically measured in pixels per inch (PPI) or dots per inch (DPI). Higher resolution images have more pixels and detail.

Brightness: The overall lightness or darkness of an image. Adjusting brightness changes the intensity of all pixels uniformly.

Contrast: The difference in brightness between the lightest and darkest areas of an image. Increasing contrast makes light areas lighter and dark areas darker.

Cropping: The process of removing unwanted outer areas from an image, focusing on a specific portion of the image.

Resizing: Adjusting the dimensions of an image, either enlarging or reducing its size. Resizing can affect image quality and detail.

Rotation: The process of turning an image around a fixed point, often used to correct orientation or apply artistic effects.

Filtering: Applying algorithms to an image to enhance or alter its appearance. Common filters include blur, sharpen, and edge detection.

Blurring: A filter that smooths an image by reducing sharpness and detail, often used to remove noise or create a soft effect.

Sharpening: A filter that enhances the edges and details in an image, making objects appear more defined.

Edge Detection: A technique used to identify and highlight the boundaries between different regions in an image. Common algorithms include the Sobel and Canny edge detectors.

Histogram: A graphical representation of the distribution of pixel intensity values in an image. It shows the frequency of different brightness levels.

Histogram Equalization: A technique to improve image contrast by spreading out the most frequent intensity values, making details more visible.

Color Space: A specific organization of colors, which helps in color representation and manipulation. Common color spaces include RGB (Red, Green, Blue) and HSV (Hue, Saturation, Value).
Grayscale: An image mode that uses shades of gray to represent an image, removing color information and simplifying the image to variations in intensity.

Color Conversion: The process of changing an image from one color space to another, such as from RGB to grayscale or HSV. Affine Transformation: A geometric transformation that preserves points, straight lines, and planes. Common operations include scaling, rotation, and translation.

Check your progress

1. Which of the following techniques is commonly used to remove Gaussian noise from an image?

A) Median Filtering
B) Gaussian Filtering
C) Bilateral Filtering
D) Edge Detection

Answer: B) Gaussian Filtering
Explanation: Gaussian filtering is specifically designed to remove Gaussian noise by smoothing the image, using a Gaussian kernel.

Question 2

What is the primary purpose of median filtering in image processing?

A) To enhance edges
B) To reduce Gaussian noise
C) To remove salt-and-pepper noise
D) To perform image segmentation

Answer: C) To remove salt-and-pepper noise

Explanation: Median filtering is effective at removing salt-and-pepper noise by replacing each pixel value with the median value of the pixels in its neighborhood.

Question 3

Which of the following is a non-linear filtering technique used for image denoising?

A) Mean Filter
B) Gaussian Filter
C) Wiener Filter
D) Bilateral Filter

Answer: D) Bilateral Filter

Explanation: Bilateral filtering is a non-linear method that smooths images while preserving edges, making it effective for denoising while maintaining sharpness.

Question 4

In image processing, what does the term —denoising‖ refer to?

A) Enhancing the contrast of an image
B) Removing unwanted noise from an image
C) Detecting edges in an image
D) Increasing the brightness of an image

Answer: B) Removing unwanted noise from an image

Explanation: Denoising refers to the process of removing noise (unwanted variations in pixel values) from an image to improve its quality.

Question 5

Which image processing technique is used to detect edges in an image?

A) Gaussian Blur
B) Histogram Equalization
C) Sobel Operator
D) Bilateral Filtering

Answer: C) Sobel Operator

Explanation: The Sobel operator is used to detect edges by computing the gradient of the image intensity at each pixel, highlighting areas of high spatial frequency.

Question 6

What type of noise is most effectively reduced by wavelet-based denoising techniques?

A) Gaussian Noise
B) Poisson Noise
C) Salt-and-Pepper Noise
D) Uniform Noise

Answer: A) Gaussian Noise

Explanation: Wavelet-based denoising is particularly effective for Gaussian noise because it leverages multi-resolution analysis to distinguish between noise and image details.

Question 7

In image processing, what does the term —blurringǁ generally refer to?

A) Enhancing fine details in an image
B) Adding noise to an image
C) Smoothing out the image to reduce detail and noise
D) Sharpening edges in an image

Answer: C) Smoothing out the image to reduce detail and noise
Explanation: Blurring is a technique used to smooth out the image, reducing noise and detail, which can help in various image processing tasks such as denoising.

Question 8
Which of the following filters is designed to preserve edges while smoothing an image?

A) Mean Filter
B) Gaussian Filter
C) Median Filter
D) Bilateral Filter

Answer: D) Bilateral Filter
Explanation: The bilateral filter smooths the image while preserving edges by considering both the spatial distance and pixel intensity differences.

Question 9
Which function in OpenCV is used to apply a Gaussian filter to an image?

A) cv2.medianBlur()
B) cv2.filter2D()
C) cv2.GaussianBlur()
D) cv2.bilateralFilter()

Answer: C) cv2.GaussianBlur()
Explanation: cv2.GaussianBlur() is the OpenCV function specifically used to apply a Gaussian filter to an image.

Question 10
What is the main advantage of using non-local means denoising over traditional methods?
A) It is computationally less expensive
B) It can better preserve fine details and textures

C) It removes more types of noise

D) It is simpler to implement

Answer: B) It can better preserve fine details and textures

Explanation: Non-local means denoising is advantageous because it leverages the similarity of patches in the image to preserve fine details and textures better than many traditional denoising methods.

Books

1. "Digital Image Processing", Author: Rafael C. Gonzalez, Richard E. Woods
2. "Image Processing: The Fundamentals", Author: Maria Petrou, Kosmas Petrou
3. "Digital Image Processing: A Practical Introduction", Author: William K. Pratt

Open source e-content link

https://uwaterloo.ca/vision-image-processing-lab/research-demos/image-denoising#:~:text=One%20of%20the%20fundamental%20challenges,contaminated%20version%20 of%20the%20image.

https://www.math.ucdavis.edu/~saito/data/acha.read.s11/buades-coll-morel-siamrev.pdf

UNIT – I END

UNIT – II

2.1 Image to Image Mapping

To describes transformations between images and some practical methods for computing them. These transformations are used for warping, image registration and finally we look at an example of automatically creating panoramas.

A homography is a 2D projective transformation that maps points in one plane to another. In our case, the planes are images or planar surfaces in 3D. Homographies have many practical uses, such as registering images, rectifying images, texture warping, and creating panoramas. We will make frequent use of them. In essence, a homography H maps 2D points (in homogeneous coordinates) according to

$$\begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} = \begin{bmatrix} h_1 & h_2 & h_3 \\ h_4 & h_5 & h_6 \\ h_7 & h_8 & h_9 \end{bmatrix} \begin{bmatrix} x \\ y \\ w \end{bmatrix} \quad \text{or} \quad \mathbf{x}' = H\mathbf{x}.$$

Homogeneous coordinates are a useful representation for points in image planes (and in 3D, as we will see later). Points in homogeneous coordinates are only defined up to scale so that **x** = [*x*, *y*, *w*] = [*αx*, *αy*, *αw*] = [*x/w*, *y/w*, 1] all refer to the same 2D point. As a consequence, the homography *H* is also only defined up to scale and has eight independent degrees of freedom. Often points are normalized with *w* = 1to have a unique identification of the image coordinates *x*, *y*. The extra coordinate makes it easy to represent transformations with a single matrix.

Create a file homography.py and add the following functions to normalize and convert to homogeneous coordinates.

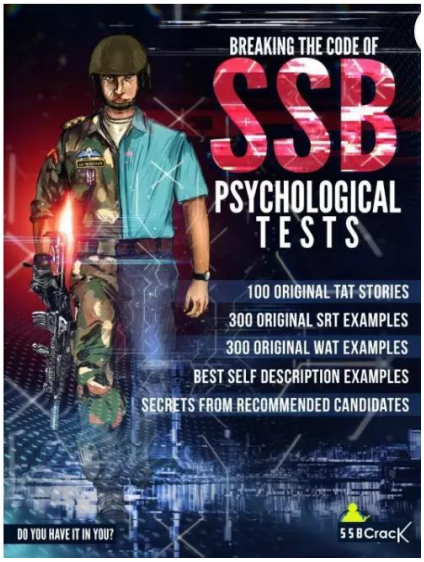
```
def normalize(points):
""" Normalize a collection of points in homogeneous coordinates so that last row
= 1. """ for row in points:
row /= points[-
1] return points
def make_homog(points):
""" Convert a set of points (dim*n array) to homogeneous
coordinates. """ return vstack((points,ones((1,points.shape[1]))))
```

2.1.1 Homographies

Homography is a transformation that maps the points in one point to the corresponding point in another image. The homography is a 3×3 matrix:

$$\mathbf{H} = \begin{vmatrix} h_{00} & h_{01} & h_{02} \\ h_{10} & h_{11} & h_{12} \\ h_{20} & h_{21} & h_{22} \end{vmatrix}$$

Importing Image Data: We will be reading the following image:



Above image is the cover page of book and it is stored as 'img.jpg'.

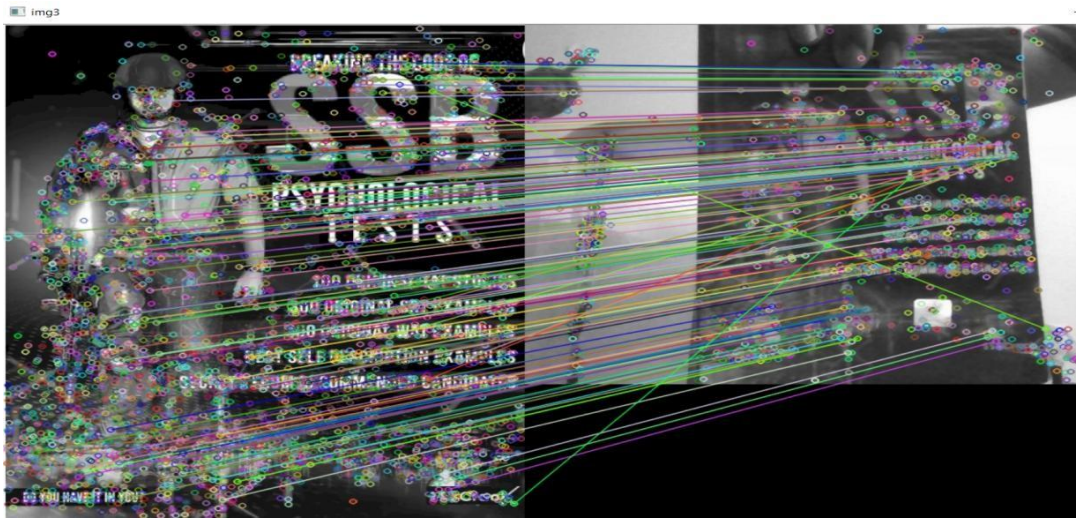
Feature Matching: Feature matching means finding corresponding features from two similar datasets based on a search distance. Now will be using sift algorithm and flann type feature matching.

Example

```
# creating the SIFT algorithm
sift = cv2.xfeatures2d.SIFT_create()
# find the keypoints and descriptors with SIFT
kp_image, desc_image =sift.detectAndCompute(img,
None) # initializing the dictionary
index_params = dict(algorithm = 0, trees
= 5) search_params = dict()
# by using Flann Matcher
flann = cv2.FlannBasedMatcher(index_params, search_params)
```

```
# reading the frame
_, frame = cap.read()

# converting the frame into grayscale
grayframe = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

# find the keypoints and descriptors with SIFT
kp_grayframe, desc_grayframe = sift.detectAndCompute(grayframe, None)

# finding nearest match with KNN algorithm
matches= flann.knnMatch(desc_image, desc_grayframe, k=2)

# initialize list to keep track of only good
points good_points=[]
for m, n in matches:
#append the points
according #to distance of
descriptors if(m.distance <
0.6*n.distance):
good_points.append(m)
```

Output



Homography: To detect the homography of the object we have to obtain the matrix and use function findHomography() to obtain the homograph of the object.

Example

```
# maintaining list of index of
descriptors # in query descriptors
query_pts = np.float32([kp_image[m.queryIdx]
.pt for m in good_points]).reshape(-1, 1, 2)


# maintaining list of index of
descriptors # in train descriptors
train_pts = np.float32([kp_grayframe[m.trainIdx]
.pt for m in good_points]).reshape(-1, 1, 2)


# finding perspective
transformation # between two
planes
matrix, mask = cv2.findHomography(query_pts, train_pts, cv2.RANSAC, 5.0)


# ravel function returns
# contiguous flattened array
matches_mask =
mask.ravel().tolist()
```

Everything is done till now, but when we try to change or move the object in another direction then the computer cannot able to find its homograph to deal with this we have to use perspective transform. For example, humans can see near objects larger than far objects, here perspective is changing. This is called Perspective transform.

Perspective transform

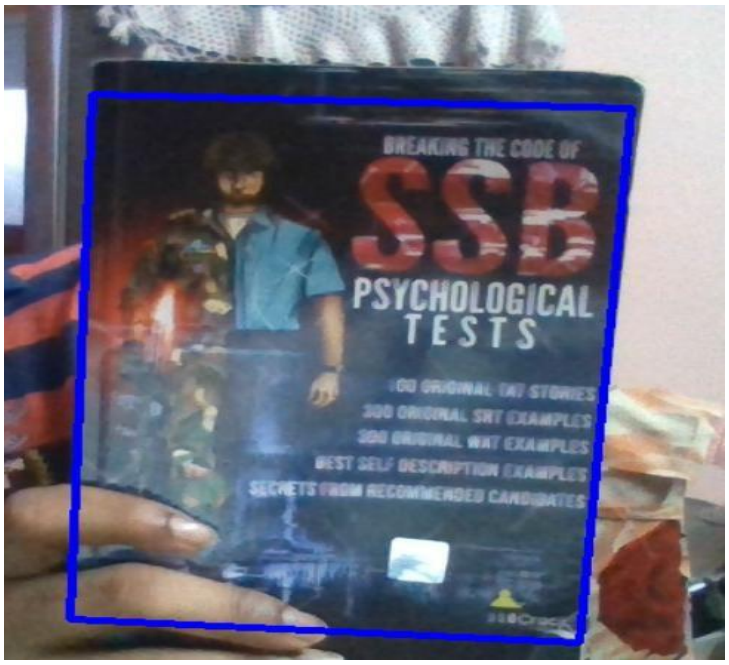
```
# initializing height and width of the
image h, w = img.shape
# saving all points in pts
pts = np.float32([[0, 0], [0, h], [w, h], [w, 0]]) .reshape(-1, 1, 2)
```

```
# applying perspective algorithm
dst = cv2.perspectiveTransform(pts, matrix)
```

At the end, let's see the output

```
# using drawing function for the frame
homography = cv2.polylines(frame, [np.int32(dst)], True, (255, 0,
0), 3) # showing the final output
# with homography
cv2.imshow("Homography",
homography)
```

Output



2.1.2 Warping images

The image warping process means that it is a process of distorting an image using various transformations like scaling, rotation, translation, and more. This is also known as geometric transformation. Using this method we can see the image from another angle.

Example: If you have an image of a rectangle, and you want to rotate it by 45 degrees. You would create a rotation matrix, apply it to the image, and obtain its rotated version. This is an example of simple warping.

Step 1: Importing libraries

Now as we have our library installed, we have to import the libraries we are going to use. We are going to import **open-cv** as we are doing these tasks in python programming language. We would also need **numpy** library for numerical purposes.

The code is as follows:

```
import cv2
import numpy as np
```

Step 2: Image Loading

We have to then load the image into the programming environment to work with it. The function used here is **'cv2.imread'** function. The argument passed must be the path of the image present on your desktop.

```
image = cv2.imread("pathofimage")
```

Step 3: Choose your source and destination points

Now that we have our image we need to choose the source points and destination points, which means the points of image and where they should be after the transformation.

```
src_points = np.float32([[0,0],[100,0],[0,100],[100,100]])
dst_points = np.float32([[60, 60],[40,60],[0,40],[40,40]])
```

Step 4: Calculating Transformation Matrix

Based on the chosen transformation, we need to construct a transformation matrix. For that we would use a pre-defined function called '**getPerspectiveTransform**'.

```
matrix = cv2.getPerspectiveTransform(src_points, dst_points)
```

Step 5: Apply Transformation

Now using this matrix, we can apply transformation to the image.

```
warped_image = cv2.warpPerspective(image, matrix, (image.shape[1], image.shape[0]))
```

The ‘**cv2.warpPerspective**‘ function is used to apply the **perspective warp** to the image using the transformation ‘**matrix**‘. The image.shape[1] and image.shape[0] are the width and height of the image respectively. These are nothing but the dimensions of the image.

Step 6: Display Result

Now display the original and warped image to see the effects of the transformation using ‘**imshow**‘ function.

cv2.imshow(„Original Image‟, image)

cv2.imshow(„Warped Image‟,

warped_image) **Step 7: Closing windows**

We can press ‘0’ and close the image windows, which is done by the two functions given below, where first, the script waits till we press ‘0’ key on keyboard, and then closes the window when pressed. The functions used are ‘**waitKey()**‘ and ‘**destroyAllWindows()**‘.

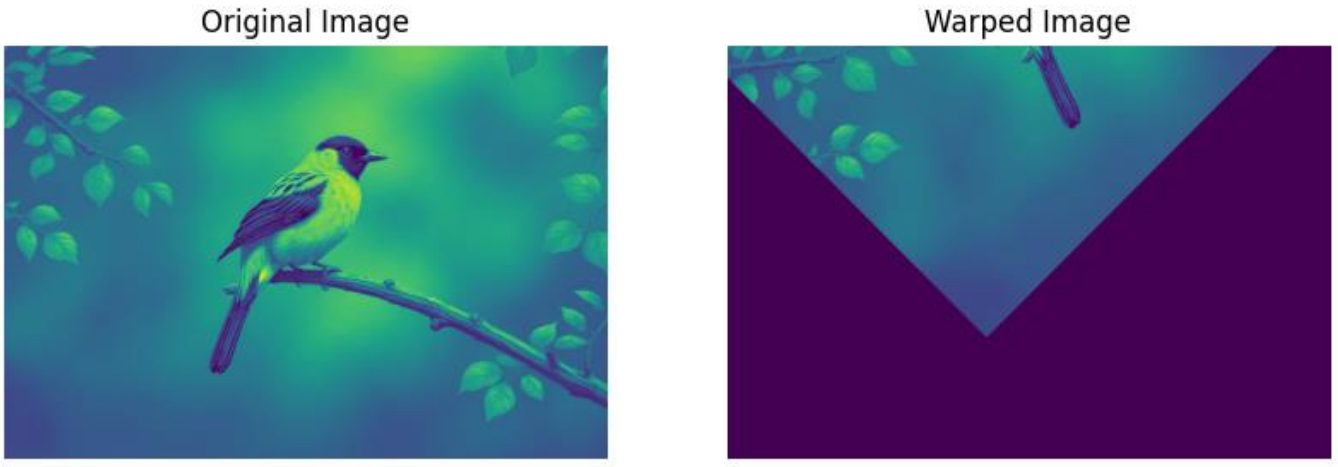
cv2.waitKey(0)

cv2.destroyAllWindos()

Example

```
#import
libraries import
cv2
import numpy as np
image = cv2.imread("C:/Users/preet/Desktop/warp.png")
src_points = np.float32([[0, 0], [image.shape[1] - 1, 0], [0, image.shape[0] - 1],
[image.shape[1] - 1,
image.shape[0] - 1]])
dst_points = np.float32([[100, 100], [image.shape[1] - 100, 100], [0, image.shape[0] - 1],
[image.shape[1] - 1, image.shape[0] - 1]])
matrix = cv2.getPerspectiveTransform(src_points, dst_points)
warped_image = cv2.warpPerspective(image, matrix, (image.shape[1],
image.shape[0])) cv2.imshow('Original Image', image)
cv2.imshow('Warped Image',
warped_image) cv2.waitKey(0)
cv2.destroyAllWindows()
```

Output



2.1.3 Creating Panoramas

Two (or more) images that are taken at the same location (that is, the camera position is the same for the images) are homographically related. This is frequently used for creating panoramic images where several images are stitched together into one big mosaic. In this section we will explore how this is done.

Comparing mean images. (left) without alignment. (right) with three-point rigid alignment.

RANSAC

RANSAC, short for "RANdom SAmple Consensus", is an iterative method to fit models to data that can contain outliers. Given a model, for example a homography between sets of points, the basic idea is that the data contains inliers, the data points that can be described by the model, and outliers, those that do not fit the model. The standard example is the case of fitting a line to a set of points that contains outliers. Simple least squares fitting will fail but RANSAC can hopefully single out the inliers and obtain the correct fit.

An example of running ransac.test(). As you can see, the algorithm selects only points consistent with a line model and correctly finds the right solution. RANSAC is a very useful algorithm which we will use in the next section for homography estimation and again for other examples.

Robust homography estimation

We can use this RANSAC module for any model. All that is needed is a Python class with fit() and get_error() methods, the rest is taken care of by ransac.py. Here we are interested in automatically finding a homography for the panorama images using a set of possible correspondences.

Example

```
class RansacModel(object):
""" Class for testing homography fit with ransac.py
from http://www.scipy.org/Cookbook/RANSAC"""
def_init_(self,debug=False):
self.debug = debug def fit(self, data):
""" Fit homography to four selected
correspondences. """ # transpose to fit
H_from_points()
data =
data.T #
from points
fp = data[:3,:4] # target points tp = data[3:,:4]
```

```
# fit homography and
return
return
H_from_points(fp,tp) def
get_error( self, data, H):
""" Apply homography to all
correspondences, return error for each
transformed point. """ data = data.T
# from
points fp =
data[:3] #
target points
tp = data[3:]
# transform fp
fp_transformed = dot(H,fp)
# normalize hom.
coordinates for i in
range(3):
fp_transformed[i] /=
fp_transformed[2] # return error per
point
return sqrt( sum((tp-fp_transformed)**2,axis=0) )
```

Comparing PCA-modes of unregistered and registered images. (top) the mean image and the first nine principal components without registering the images beforehand. (bottom) the same with the registered images.

To fit a homography using RANSAC we first need to add the following model class to homography. This class contains a fit() method which just takes the four correspondences selected by ransac.py (they are the first four in data) and fits a homogra phy. Remember, four points are the minimal number to compute a homography. The method get_error() applies the homography and returns the sum of squared distance for each correspondence pair so that RANSAC can chose which points to keep as inliers and outliers. This is done with a threshold on this distance. For ease of use, add the following function to homography.py.

Stitching the images together

With the homographies between the images estimated (using RANSAC) we now need to warp all images to a common image plane. It makes most sense to use the plane of the center image (otherwise the distortions will be huge). One way to do this is to create a very large image, for example filled with zeros, parallel to the central image and warp all the images to it. Since all our images are taken with a horizontal rotation of the camera we can use a simpler procedure, we just pad the central image with zeros to the left or right to make room for the warped images. Add the following function which handles this to warp.py.

Example

```
def
panorama(H,fromim,toim,padding=2400,delta=2400)
: """ Create horizontal panorama by blending two
images
using a homography H (preferably estimated using
RANSAC). The result is an image with the same height
as toim. 'padding'
specifies number of fill pixels and 'delta' additional
translation. """ # check if images are grayscale or color
is_color = len(fromim.shape) == 3
# homography transformation for
geometric_transform() def transf(p):
p2 = dot(H,[p[0],p[1],1])
return (p2[0]/p2[2],p2[1]/p2[2])
if H[1,2]<0: # fromim is to the
right print 'warp - right'
# transform
fromim if is_color:
# pad the destination image with zeros to the right
toim_t =
hstack((toim,zeros((toim.shape[0],padding,3))))
fromim_t =
zeros((toim.shape[0],toim.shape[1]+padding,toim.shape[2])) for
col in range(3):
```

```
fromim_t[:,:,col] =
ndimage.geometric_transform(fromim[:,:,col],
transf,(toim.shape[0],toim.shape[1]+padding))
else:
# pad the destination image with zeros to the right
toim_t =
hstack((toim,zeros((toim.shape[0],padding))))
fromim_t =
ndimage.geometric_transform(fromim,transf,
(toim.shape[0],toim.shape[1]+padding))
else:
print 'warp - left'
# add translation to compensate for padding to
the left H_delta = array([[1,0,0],[0,1,-
delta],[0,0,1]])
H =
dot(H,H_delta) #
transform fromim
if is_color:
# pad the destination image with zeros to the left
toim_t = hstack((zeros((toim.shape[0],padding,3)),toim))
fromim_t =
zeros((toim.shape[0],toim.shape[1]+padding,toim.shape[2])) for
col in range(3):
fromim_t[:,:,col] =
ndimage.geometric_transform(fromim[:,:,col],
transf,(toim.shape[0],toim.shape[1]+padding))
else:
# pad the destination image with zeros to the left
toim_t =
hstack((zeros((toim.shape[0],padding)),toim))
fromim_t = ndimage.geometric_transform(fromim,
transf,(toim.shape[0],toim.shape[1]+padding))
# blend and return (put fromim above
toim) if is_color:
```

```
# all non black pixels
alpha = ((fromim_t[:,:,0] * fromim_t[:,:,1] * fromim_t[:,:,2] ) >
0) for col in range(3):
toim_t[:,:,col] = fromim_t[:,:,col]*alpha + toim_t[:,:,col]*(1-
alpha) else:
alpha = (fromim_t > 0)
toim_t = fromim_t*alpha + toim_t*(1-
alpha) return toim_t
```

Output

2.2.1 Pin-Hole Camera Model

The pin-hole camera model (or sometimes projective camera model) is a widely used camera model in computer vision. It is simple and accurate enough for most applications. The name comes from the type of camera, like a camera obscura, that collects light through a small hole to the inside of a dark box or room. In the pin-hole camera model, light passes through a single point, the camera center, C, before it is projected onto an image plane. The image plane in an actual camera would be upside down behind the camera center but the model is the same. The projection properties of a pin- hole camera can be derived from this illustration and the assumption that the image axis are aligned with the x and y axis of a 3D coordinate system. The optical axis of the camera then coincides with the z axis and the projection follows from similar triangles. By adding rotation and translation to

put a 3D point in this coordinate system before projecting, the complete projection transform follows. The interested reader can find the details in [13] and [25, 26]. With a pin-hole camera, a 3D point X is projected to an image point x (both expressed in homogeneous coordinates) as

$$!x = PX .$$

Here the 3 ⇥ 4 matrix P is called the camera matrix (or projection matrix). Note that the 3D point X has four elements in homogeneous coordinates, X = [X, Y,Z,W]. The scalar ! is the inverse depth of the 3D point and is needed if we want all coordinates to be homogeneous with the last value normalized to one.

The camera matrix

The camera matrix can be decomposed as

$$P = K [R | t] ,$$

Where R is a rotation matrix describing the orientation of the camera, t a 3D translation vector describing the position of the camera center, and the intrinsic calibration matrix K describing the projection properties of the camera. The calibration matrix depends only on the camera properties and is in a general form written as

$$K = \begin{bmatrix} \alpha f & s & c_x \\ 0 & f & c_y \\ 0 & 0 & 1 \end{bmatrix}$$

The focal length, f, is the distance between the image plane and the camera center. The skew, s, is only used if the pixel array in the sensor is skewed and can in most cases safely be set to zero.

This gives

$$K = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$$

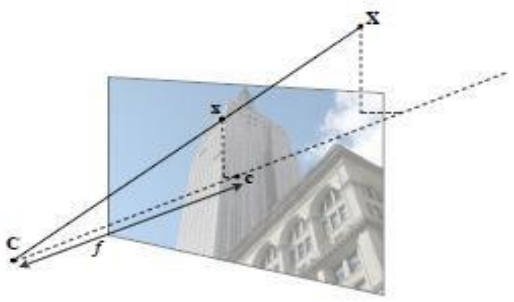
where we used the alternative notation fx and fy, with fx = ↲fy.

The aspect ratio, ↲ is used for non-square pixel elements. It is often safe to assume ↲ = 1.

$$K = \begin{bmatrix} f & 0 & c_x \\ 0 & f & c_y \\ 0 & 0 & 1 \end{bmatrix}$$

With this assumption the matrix becomes

the focal length, the only remaining parameters are the coordinates of the optical center (sometimes called the principal point), the image point c = [cx, cy] where



The pin-hole camera model. The image point x is at the intersection of the image plane and the line joining the 3D point X and the camera center C. The dashed line is the optical axis of the camera. The optical axis intersects the image plane. Since this is usually in the center of the image and image coordinates are measured from the top left corner, these values are often well approximated with half the width and height of the image. It is worth noting that in this last case the only unknown variable is the focal length f.

2.2.2 Camera Calibration

Calibrating a camera means determining the internal camera parameters, in our case the matrix K. It is possible to extend this camera model to include radial distortion and other artifacts if your application needs precise measurements. For most applications however, the simple model in equation . The standard way to calibrate cameras is to take lots of pictures of a flat checkerboard pattern.

A simple calibration method

Here we will look at a simple calibration method. Since most of the parameters can be set using basic assumptions (square straight pixels, optical center at the center of the image) the tricky part is getting the focal length right. For this calibration method you need a flat rectangular calibration object (a book will do), measuring tape or a ruler and preferable a flat surface.

- Measure the sides of your rectangular calibration object. Let's call these dX and dY .
- Place the camera and the calibration object on a flat surface so that the camera back and calibration object are parallel and the object is roughly in the center of the camera's view. You might have to raise the camera or object to get a nice alignment.
- Measure the distance from the camera to the calibration object. Let's call this dZ.
- Take a picture and check that the setup is straight, meaning that the sides of the

calibration object align with the rows and columns of the image.

- Measure the width and height of the object in pixels. Let's call these dx and dy.

Using similar triangles (look at Figure 4.1 to convince yourself that) the following relation gives the focal lengths:

$$f_x = \frac{dx}{dX}dZ \quad , \quad f_y = \frac{dy}{dY}dZ$$

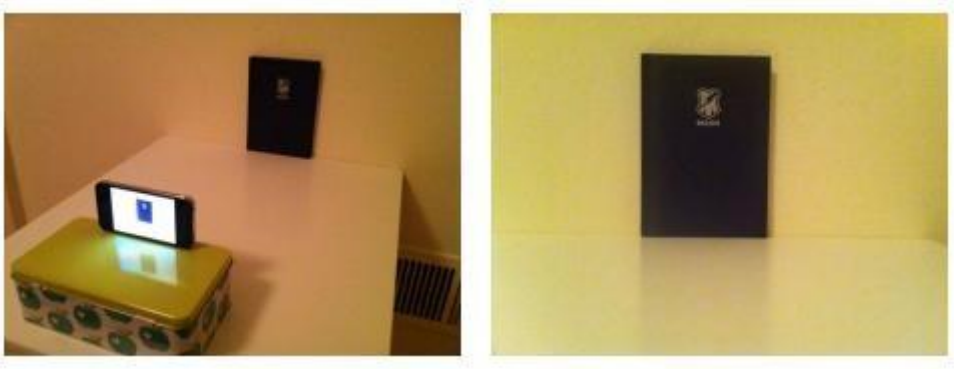


Figure 4.3 A simple camera calibration setup: an image of the setup used (left); the image used for the calibration (right). Measuring the width and height of the calibration object in the image and the physical dimensions of the setup is enough to determine the focal length.

For the particular setup in Figure 4.3, the object was measured to be 130 by 185 mm, so dX = 130 and dY = 185. The distance from camera to object was 460 mm, so dZ = 460. If we can use any unit of measurement, it doesn't matter, only the ratios of the measurements matter. Using ginput() to select four points in the image, the width and height in pixels was 722 and 1040. This means that dx
= 722 and dy = 1040. Putting these values in the relationship above gives.

$$f_x = 2555 \quad , \quad f_y = 2586$$

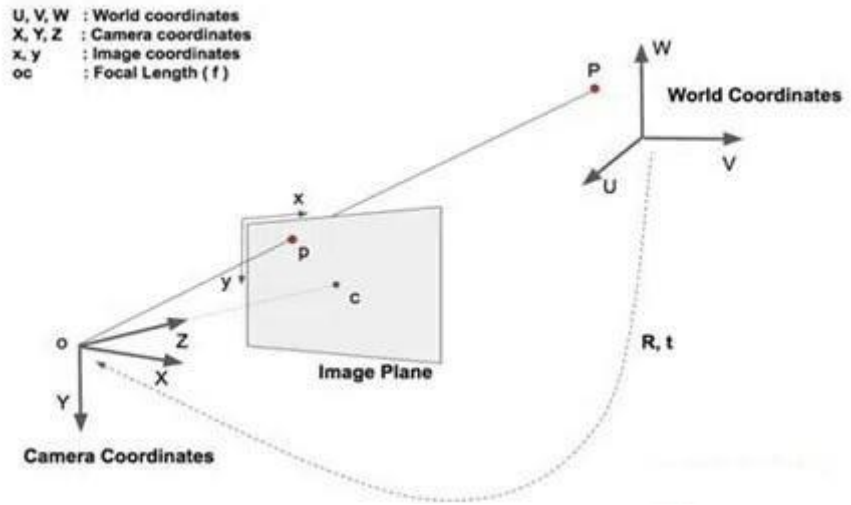
It is important to note that this is for a particular image resolution. In this case the image was 2592 ⇥ 1936 pixels. Remember that the focal length and the optical center are measured in pixels and scale with the image resolution.

2.2.3 Pose Estimation from Planes and Markers

We saw how to estimate homographies between planes. Combining this with a calibrated camera makes it possible to compute the camera's pose (rotation and translation) if the image contains a planar marker object. This is marker object can be almost any flat object.

Let's illustrate with an example. Consider the two top images in Figure 4.3. The following code will extract SIFT features in both images and robustly estimate a homography using RANSAC:

Figure 4.4 Example of computing the projection matrix for a new view using a planar object as marker. Matching image features to an aligned marker gives a homography that can be used to compute the pose of the camera. Template image with a gray square (top left); an image taken from an unknown viewpoint with the same square transformed with the estimated homography (top right); a cube transformed using the estimated camera matrix (bottom). Now we have a homography that maps points on the marker (in this case the book) in one image to their corresponding locations in the other image. Let's define our 3D coordinate system so that the marker lies in the X-Y plane ($Z = 0$) with the origin somewhere on the marker.



2.2.4 Augmented Reality

Augmented reality (*AR*) is a collective term for placing objects and information on top of image data. The classic example is placing a 3D computer graphics model so that it looks like it belongs in the scene, and moves naturally with the camera motion in the case of video. Given an image with a marker plane as in the section above, we can compute the camera's position and pose and use that to place computer graphics models so that they are rendered correctly.

PyGame and PyOpenGL

PyGame is a popular package for game development that easily handles display windows, input devices, events, and much more. PyGame is open source and available from. It is actually a Python binding for the SDL game engine. For installation instructions, see Appendix A.

PyOpenGL is the Python binding to the OpenGL graphics programming interface. OpenGL comes pre-installed on almost all systems and is a crucial part for graphics performance. OpenGL is cross platform and works the same across operating systems.

There is no way we can cover any significant portion of OpenGL programming. We will instead just show the important parts, for example how to use camera matrices in OpenGL and setting up a basic 3D model. Some good examples and demos are available in the PyOpenGL-Demo package. This is a good place to start if you are new to PyOpenGL.

We want to place a 3D model in a scene using OpenGL. To use PyGame and PyOpenGL for this application, we need to import the following at the top of our scripts:

```
from OpenGL.GL import *
from OpenGL.GLU import *
import pygame,
pygame.image from
pygame.locals import *
```

The two main components of setting up an OpenGL scene are the projection and model view matrices. Let's get started and see how to create these matrices from our pin-hole cameras.

From Camera Matrix to OpenGL Format

OpenGL uses 4 × 4 matrices to represent transforms (both 3D transforms and projections). This is only slightly different from our use of 3 × 4 camera matrices. However, the camera-scene

transformations are separated in two matrices, the *GL_PROJECTION* matrix and the *GL_MODELVIEW* matrix. GL_PROJECTION handles the image formation properties and is the equivalent of our internal calibration matrix *K*. GL_MODELVIEW handles the 3D transformation of the relation between the objects and the camera. This corresponds roughly to the *R* and *t* part of our camera matrix. One difference is that the coordinate system is assumed to be centered at the camera so the GL_MODELVIEW matrix actually contains the transformation that places the objects in front of the camera. There are many peculiarities with working in OpenGL; we will comment on them as they are encountered in the examples below.

Given that we have a camera calibrated so that the calibration matrix *K* is known, the following function translates the camera properties to an OpenGL projection matrix:

Example

```
def set_projection_from_camera(K):
""" Set view from a camera calibration matrix.
"""                glMatrixMode(GL_PROJECTION)
glLoadIdentity()
fx = K[0,0]
fy = K[1,1]
fovy = 2*arctan(0.5*height/fy)*180/pi aspect =
(width*fy)/(height*fx)
# define the near and far clipping
planes near = 0.1
far = 100.0
#          set          perspective
gluPerspective(fovy,aspect,near,
far) glViewport(0,0,width,height)
```

We assume the calibration to be of the simpler form with the optical center at the image center. The first function glMatrixMode() sets the working matrix to GL_PROJECTION and subsequent commands will modify this matrix. Then glLoadIdentity() sets the matrix to the identity matrix, basically reseting any prior changes. We then calculate the vertical field of view in degrees with the help of the image height and the camera's focal length as well as the aspect ratio. An OpenGL projection also has a

near and far clipping plane to limit the depth range of what is rendered. We just set

the near depth to be small enough to contain the nearest object and the far depth to some large number. We use the GLU utility function gluPerspective() to set the projection matrix and define the whole image to be the view port (essentially what is to be shown). There is also an option to load a full projection matrix with glLoadMatrixf() similar to the model view function below. This is useful when the simple version of the calibration matrix is not good enough. The model view matrix should encode the relative rotation and translation that brings the object in front of the camera (as if the camera was at the origin). It is a 4 × 4 matrix that typically looks like this:

where *R* is a rotation matrix with columns equal to the direction of the three coordinate axis and **t** is a translation vector. When creating a model view matrix, the rotation part will need to hold all rotations (object and coordinate system) by multiplying together the individual components.

Placing Virtual Objects in the Image

The first thing we need to do is to add the image (the one we want to place virtual objects in) as a background. In OpenGL this is done by creating a quadrilateral, a *quad*, that fills the whole view. The easiest way to do this is to draw the quad with the projection and model view matrices reset so that the coordinates go from – 1 to 1 in each dimension. This function loads an image, converts it to an OpenGL texture, and places that texture on the quad:

Script1

```
def draw_background(imname):
""" Draw background image using a quad. """
# load background image (should be .bmp) to OpenGL
texture bg_image =
pygame.image.load(imname).convert()
bg_data = pygame.image.tostring(bg_image,"RGBX",1)


glMatrixMode(GL_MODELVIEW
) glLoadIdentity()
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)


# bind the texture
glEnable(GL_TEXTURE_D)
```

```
glBindTexture(GL_TEXTURE_2D,glGenTextures(1))
glTexImage2D(GL_TEXTURE_2D,0,GL_RGBA,width,height,0,GL_RGBA,GL_UNSIGNED
_BY TE,bg_data)
glTexParameterf(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_NEAREST)
glTexParameterf(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_NEAREST)

# create quad to fill the whole
window glBegin(GL_QUADS)
glTexCoord2f(0.0,0.0); glVertex3f(-1.0,-1.0,-1.0)
glTexCoord2f(1.0,0.0); glVertex3f( 1.0,-1.0,-1.0)
glTexCoord2f(1.0,1.0); glVertex3f( 1.0, 1.0,-1.0)
glTexCoord2f(0.0,1.0); glVertex3f(-1.0, 1.0,-
1.0) glEnd()

# clear the texture
glDeleteTextures1)
```

The full script for generating an image like the one in Figure 4-5 looks like this (assuming that you also have the functions introduced above in the same file):

Script2

```
from OpenGL.GL import *
from OpenGL.GLU import *
from OpenGL.GLUT import
* import pygame,
pygame.image from
pygame.locals import *
import pickle
width,height =
1000,747 def setup():
""" Setup window and pygame environment. """
pygame.init()
pygame.display.set_mode((width,height),OPENGL |
DOUBLEBUF) pygame.display.set_caption('OpenGL AR
```

```
demo')

# load camera data
with open('ar_camera.pkl','r')
as f: K = pickle.load(f)
Rt = pickle.load(f)

setup()
draw_background('book_perspective.bmp')
set_projection_from_camera(K)
set_modelview_from_camera(Rt)
draw_teapot(0.02)

while True:
event = pygame.event.poll()
if event.type in
(QUIT,KEYDOWN): break
pygame.display.flip()
```

Figure 4.5 Augmented reality. Placing a computer graphics model on a book in a scene using camera parameters computed from feature matches: the Utah teapot rendered in place aligned with the coordinate axis (top); sanity check to see the position of the origin (bottom).

Unit Summary

Image mapping and augmented reality (AR) are two interconnected technologies that enhance the way we interact with digital and physical environments. Image mapping involves the technique of applying a digital image or texture to a 3D surface, creating a more realistic or visually appealing representation. This process is commonly used in 3D graphics and modeling to give surfaces textures and details, such as applying a brick texture to a 3D wall in a video game or simulation. Image mapping helps in adding depth and realism to virtual objects, improving user experience in digital environments.

Augmented reality (AR), on the other hand, overlays digital information onto the physical world in real-time, enhancing the user's perception and interaction with their surroundings. AR technology uses devices such as smartphones, tablets, or AR glasses to display digital content, such as images, text, or 3D models, superimposed on the real world. This can range from simple overlays, like displaying navigation directions on a windshield, to more complex applications, such as interactive educational tools or immersive gaming experiences where virtual elements interact with real-world objects.

In essence, while image mapping focuses on applying textures and details to digital models, augmented reality expands the scope by integrating digital elements into the physical environment, creating a blended experience where users can interact with both real and virtual objects seamlessly. Both technologies leverage advanced graphics and processing techniques to enhance visual experiences and provide users with more immersive and interactive interactions.

Let us sum up

Image Mapping: The technique of applying a digital image or texture to a 3D surface to create a realistic or detailed appearance. It‘s often used in 3D modeling and computer graphics.

Texture Mapping: A method in 3D graphics where a 2D image (texture) is wrapped around a 3D model to give it a detailed appearance. This can include patterns, colors, and visual details.

UV Mapping: The process of mapping a 2D image (texture) onto a 3D model by unwrapping the 3D surface into a 2D plane. This allows textures to be applied accurately to complex shapes.

Self Assessment Questions:

1. Briefly Explain the Concept of Homographies.
2. Explain the Direct Linear Transformation Algorithm.
3. Explain the Warping Images.
4. Explain the Creating Panoramas in detail.
5. Briefly Explain the Pin-Hole Camera Model with diagram.
6. Explain the Camera Calibration in detail.
7. Pose Estimation from Planes and Markers in detail.
8. Augmented Reality in detail and discuss the advantages of AR.

Books

1. "3D Game Programming with DirectX 9.0: A Tutorial Approach", Author: Eric Lengyel
2. "Computer Graphics: Principles and Practice", Authors: John F. Hughes, Andries van Dam, Morgan McGuire, David Sklar, James A. Fogarty, Kurt Akeley
3. "Augmented Reality: Principles and Practice", Authors: Dieter Schmalstieg, Tobias Hollerer

Glossary

Bump Mapping: A technique used to simulate surface detail by altering the way light interacts with the surface, creating the illusion of bumps and wrinkles without changing the model's geometry.

Normal Mapping: A technique similar to bump mapping but uses a normal map to provide more detailed surface textures by modifying the way light reflects off the surface, creating more realistic visual effects.

Diffuse Map: A texture map that defines the color and texture of a surface. It is the primary texture that determines the base color of the 3D object.

Specular Map: A texture map that controls the shininess and reflection characteristics of a surface. It defines how reflective and shiny different parts of the surface are.

Environment Mapping: A technique used to simulate reflective surfaces by mapping an environment image onto a 3D object, creating the illusion of reflections from the surrounding environment.

Augmented Reality (AR): A technology that overlays digital content (such as images, text, or 3D models) onto the real world in real-time, enhancing the user's perception of their physical surroundings.

AR Headset: A wearable device that displays augmented reality content in the user's field of view. Examples include Microsoft HoloLens and Magic Leap.

AR Glasses: Lightweight, glasses-like devices that project digital information onto the real world, offering hands-free AR experiences. Examples include Google Glass.

Marker-Based AR: An AR approach that uses visual markers or patterns (such as QR codes) to trigger the display of digital content when the marker is recognized by the AR system.

Markerless AR: AR that does not rely on visual markers. Instead, it uses features like GPS, accelerometers, or computer vision algorithms to place and track digital content in the real world.

Haptic Feedback: Sensory feedback provided through touch or vibration to enhance the interaction with AR elements. Haptic feedback can make virtual interactions feel more real and immersive.

AR Application (App): A software application designed to deliver augmented reality experiences. These apps utilize AR technology to blend digital content with the real world for various purposes, such as navigation, gaming, or education.

Virtual Object: A digital entity created within an AR environment that interacts with or enhances the physical world. Virtual objects can include 3D models, animations, or interactive elements.

Check your progress

Question 1

What is the primary goal of image mapping in computer graphics?

A) To convert a 3D model into a 2D image

B) To apply textures to a 3D model

C) To enhance the color of an image

D) To detect edges in an image

Answer: B) To apply textures to a 3D model

Explanation: Image mapping (or texture mapping) involves applying a 2D image (texture) to the surface of a 3D model to give it a more realistic appearance.

Question 2

Which technique is commonly used to align a virtual object with the real world in augmented reality?

A) Depth Sensing

B) Image Recognition

C) Feature Matching

D) All of the Above

Answer: D) All of the Above

Explanation: AR systems often use a combination of depth sensing, image recognition, and feature matching to align virtual objects with the real world accurately.

Question 3

What is the main purpose of a "marker" in augmented reality applications?

A) To increase image resolution

B) To provide a reference point for aligning virtual content

C) To apply filters to images

D) To enhance the brightness of images

Answer: B) To provide a reference point for aligning virtual content

Explanation: Markers are used in AR to serve as reference points for tracking and aligning virtual content with the real world.

Question 4

Which OpenCV function is used to perform feature detection and matching, which is crucial for AR applications?

A) cv2.GaussianBlur()

B) cv2.findContours()

C) cv2.SIFT()

D) cv2.threshold()

Answer: C) cv2.SIFT()

Explanation: The cv2.SIFT() function (Scale-Invariant Feature Transform) detects and describes local features in images, which is essential for matching and tracking features in AR applications.

Question 5

What is "homography" in the context of image mapping and augmented reality?

A) The process of merging multiple images into one

B) A transformation that maps points from one plane to another

C) A method for color correction

D) A technique for edge detection

Answer: B) A transformation that maps points from one plane to another

Explanation: Homography is a transformation used to map points between two planes, which is fundamental in tasks such as aligning textures to surfaces and tracking objects in AR.

Question 6

Which of the following is a common technique for tracking and overlaying virtual content on physical objects in AR?

A) Particle Filtering
B) Optical Flow
C) Marker-based Tracking
D) Histogram Matching

Answer: C) Marker-based Tracking

Explanation: Marker-based tracking uses visual markers (like QR codes) to detect and track the position and orientation of objects, allowing for accurate overlay of virtual content.

Question 7

In augmented reality, what does "SLAM" stand for and what is its purpose?

A) Simultaneous Localization and Mapping; to track the position of a device and map its environment
B) Single Lens Alignment and Mapping; to correct image distortions
C) Spatial Localization and Augmented Mapping; to enhance image resolution
D) Stereoscopic Lens Alignment and Measurement; to measure depth

Answer: A) Simultaneous Localization and Mapping; to track the position of a device and map its environment

Explanation: SLAM is used in AR to simultaneously track the position of the device and build a map of the environment, allowing virtual content to be accurately placed and moved in relation to the real world.

Question 8

What role does "depth sensing" play in augmented reality?

A) To enhance the color of the image

B) To measure the distance between objects in the real world

C) To apply textures to 3D models

D) To detect edges in the image

Answer: B) To measure the distance between objects in the real world

Explanation: Depth sensing measures the distance between objects in the real world, which is crucial for accurately placing and interacting with virtual content in AR environments.

Question 9

Which of the following is an example of a common marker-based AR toolkit?

A) OpenCV

B) Vuforia

C) TensorFlow

D) Keras

Answer: B) Vuforia

Explanation: Vuforia is a popular AR toolkit that supports marker-based tracking and is widely used for developing AR applications.

Question 10

In augmented reality, what is the purpose of "registration"?

A) To correct image distortions

B) To align virtual objects with real-world features

C) To enhance image contrast

D) To apply color maps to images

Answer: B) To align virtual objects with real-world features

Explanation: Registration is the process of aligning virtual objects with real-world features to ensure that the virtual content appears correctly positioned and oriented in the AR environment.

Open source e-content links

https://provenreality.com/augmented-reality-mapping-systems/

https://www.blippar.com/blog/2017/11/06/welcome-ar-city-future-maps-and-navigation

UNIT – II END

UNIT – III

3.1 Multiple View Geometry

This chapter will show you how to handle multiple views and how to use the geometric relationships between them to recover camera positions and 3D structure. With images taken at different viewpoints it is possible to compute 3D scene points as well as camera locations from feature matches. We introduce the necessary tools and show a complete 3D reconstruction example. The last part of the chapter shows how to compute dense depth reconstructions from stereo images.

3.1.1 Epipolar Geometry

Multiple view geometry is the field studying the relationship between cameras and features when there are correspondences between many images that are taken from varying viewpoints. The image features are usually interest points and we will focus on that case throughout this chapter. The most important constellation is two-view geometry.

With two views of a scene and corresponding points in these views there are geometric constraints on the image points as a result of the relative orientation of the cameras, the properties of the cameras, and the position of the 3D points. These geometric relationships are described by what is called epipolar geometry.

Fig: A single picture such as this picture of a man holding up the Leaning Tower of Pisa can result in ambiguous scenarios. Multiple views of the same scene help us resolve these

potential ambiguities.

Above Figure, we may be initially fooled to believe that the man is holding up the Leaning Tower of Pisa. Only by careful inspection can we tell that this is not the case and merely an illusion based on the projection of di_erent depths onto the image plane. However, if we were able to view this scene from a completely di_erent angle, this illusion immediately disappears and we would instantly _gure out the correct scene layout.

Without any prior knowledge of the cameras, there is an inherent ambiguity in that a 3D point, **X**, transformed with an arbitrary (4⇥4) homography H as H**X** will have the same image point in a camera P H1 as the original point in the camera P. Expressed with the camera equation, this is

x = PX = P H1HX = P^X

Because of this ambiguity, when analyzing two view geometry we can always transform the cameras with a homography to simplify matters. Often this homography is just a rigid transformation to change the coordinate system. A good choice is to set the origin and coordinate axis to align with the first camera so that

P1 = K1[I | 0] and P2 = K2[R | t]

Here we use the same notation as in Chapter 4; K1 and K2 are the calibration matrices, R is the rotation of the second camera, and t is the translation of the second camera. Using these camera matrices one can derive a condition for the projection of a point X to image points x1 and x2 (with P1 and P2 respectively). This condition is what makes it possible to recover the camera matrices from corresponding image points.

$$\mathbf{x}_2^T F \mathbf{x}_1 = 0 \ ,$$

where

$$F = K_2^{-T} S_t R K_1^{-1}$$

The following equation must be satisfied **(5.1)**

and the matrix St is the skew symmetric matrix **(5.2)**

$$S_t = \begin{bmatrix} 0 & -t_3 & t_2 \\ t_3 & 0 & -t_1 \\ -t_2 & t_1 & 0 \end{bmatrix}$$

Equation (5.1) is called the epipolar constraint . The matrix F in the epipolar constraint is called the fundamental matrix and as you can see, it is expressed in components of the two camera matrices (their relative rotation R and translation t). The fundamental matrix has rank 2 and det(F)=0. This will be used in algorithms for estimating F. The fundamental matrix makes it possible to compute the camera matrices and then a 3D reconstruction.

The equations above mean that the camera matrices can be recovered from F, which in turn can be computed from point correspondences as we will see later. Without knowing the internal calibration (K1 and K2) the camera matrices are only recoverable up to a projective transformation. With known calibration, the reconstruction will be metric. A metric reconstruction is a 3D reconstruction that correctly represents distances and angles. There is one final piece of geometry needed before we can proceed to actually using this theory on some image data.

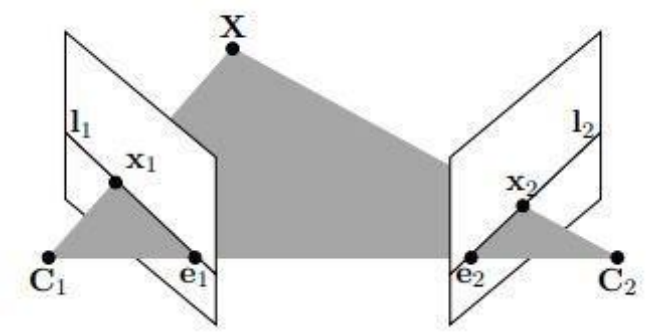


Figure5.1 Epipolar geometry

Figure 5.1: An illustration of epipolar geometry. A 3D point X is projected to x1 and x2, in the two views respectively. The baseline between the two camera centers, C1 and C2, intersect the image planes in the epipoles, e1 and e2. The lines l1 and l2 are called epipolar lines.

The epipolar lines all meet in a point, e, called the epipole. The epipole is actually the image point corresponding to the projection of the other camera center. This point can be outside the actual image, depending on the relative orientation of the cameras. Since the epipole lies on all epipolar lines it must satisfy Fe1 = 0. It can therefore be computed as the null vector of F as we will see later. The other epipole can be computed from the relation eT 2 F = 0.

The first five of the epipolar lines are shown in the first view and the corresponding matching points in view 2.

Here we used the helper plot function.

```
def plot_epipolar_line(im,F,x,epipole=None,show_epipole=True):
""" Plot the epipole and epipolar line
F*x=0 in an image. F is the
fundamental matrix and x a point in the
other image."""
m,n =
im.shape[:2] line
= dot(F,x)
# epipolar line parameter and
values t = linspace(0,n,100)
lt = array([(line[2]+line[0]*tt)/(-line[1]) for tt in
t]) # take only line points inside the image
ndx = (lt>=0) & (lt <m)
plot(t[ndx],lt[ndx],linewidth=2
) if show_epipole:
if epipole is None:
epipole = compute_epipole(F)
plot(epipole[0]/epipole[2],epipole[1]/epipole[2],'r
*')
```

This function parameterizes the line with the range of the x axis and removes parts of lines above and below the image border. If the last parameter show_epipole is true,

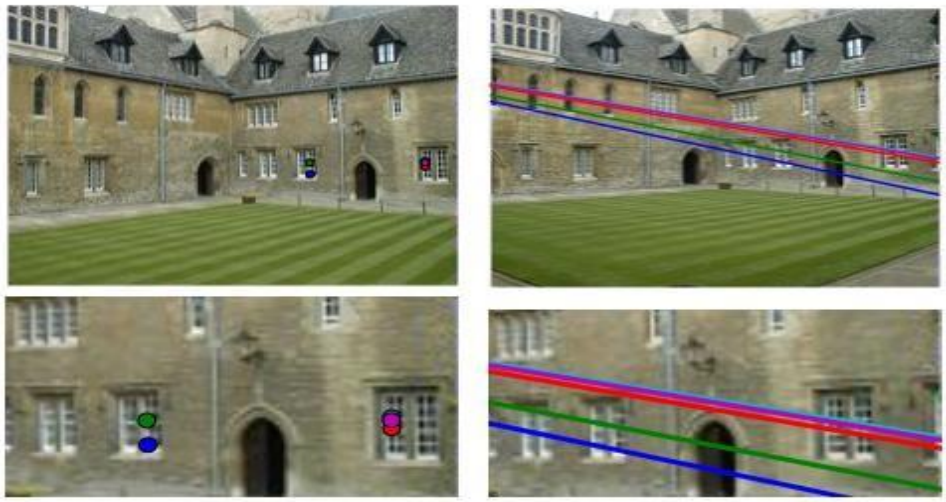
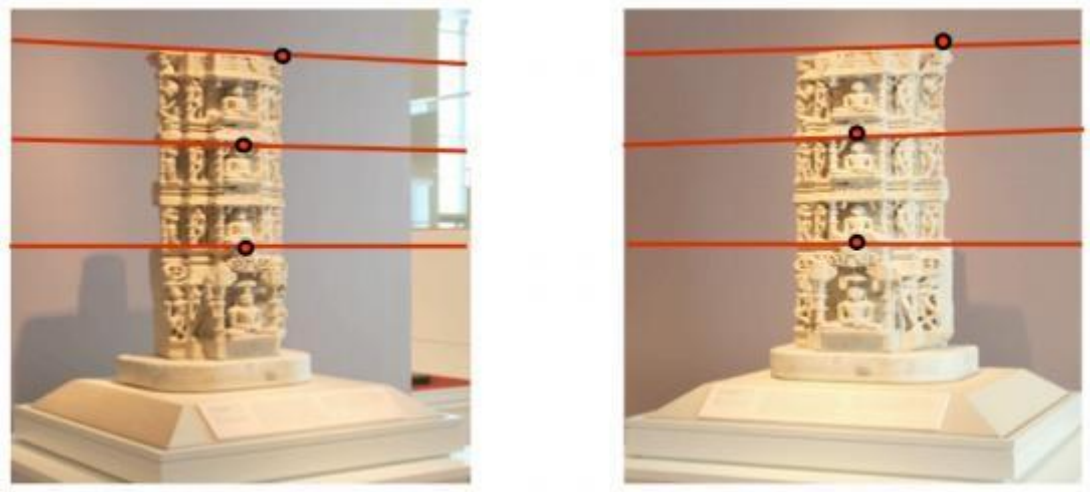


Figure 5.4: Epipolar lines in view 1 shown for five points in view 2 of the Merton1 data. The bottom row shows a close up of the area around the points. The lines can be seen to converge on a point outside the image to the left. The lines show where point correspondences can be found in the other image (the color coding matches between lines and points).



An example of epipolar lines and their corresponding points drawn on an image pair.

3.1.2 Computing with Cameras and 3D Structure

The previous section covered relationships between views and how to compute the fundamental matrix and epipolar lines. Here we briefly explain the tools we need for computing with cameras and 3D structure.

Triangulation

Given known camera matrices, a set of point correspondences can be triangulated to recover the 3D positions of these points. The basic algorithm is fairly simple.

For two views with camera matrices P1 and P2, each with a projection x1 and x2 of the same 3D point X (all in homogeneous coordinates), the camera equation (4.1) gives the following relation.

$$\begin{bmatrix} P_1 & -\mathbf{x}_1 & 0 \\ P_2 & 0 & -\mathbf{x}_2 \end{bmatrix} \begin{bmatrix} \mathbf{X} \\ \lambda_1 \\ \lambda_2 \end{bmatrix} = 0$$

There might not be an exact solution to these equations due to image noise, errors in the camera matrices or other sources of errors. Using SVD, we can get a least squares estimate of the 3D point.

```
def triangulate_point(x1,x2,P1,P2):
  """ Point pair triangulation from
    least squares solution. """

  M = zeros((6,6))
  M[:3,:4] = P1
  M[3:,:4] = P2
  M[:3,4] = -x1
  M[3:,5] = -x2

  U,S,V = linalg.svd(M)
  X = V[-1,:4]

  return X / X[3]
```

Add the following function that computes the least squares triangulation of a point pair to sfm.py.

```
def triangulate(x1,x2,P1,P2):
  """ Two-view triangulation of points in
    x1,x2 (3*n homog. coordinates). """

  n = x1.shape[1]
  if x2.shape[1] != n:
    raise ValueError("Number of points don't match.")

  X = [ triangulate_point(x1[:,i],x2[:,i],P1,P2) for i in range(n)]
  return array(X).T
```

```
import sfm

# index for points in first two views
ndx = (corr[:,0]>=0) & (corr[:,1]>=0)

# get coordinates and make homogeneous
x1 = points2D[0][:,corr[ndx,0]]
x1 = vstack( (x1,ones(x1.shape[1])) )
x2 = points2D[1][:,corr[ndx,1]]
x2 = vstack( (x2,ones(x2.shape[1])) )

Xtrue = points3D[:,ndx]
Xtrue = vstack( (Xtrue,ones(Xtrue.shape[1])) )

# check first 3 points
Xest = sfm.triangulate(x1,x2,P[0].P,P[1].P)
print Xest[:,:3]
print Xtrue[:,:3]

# plotting
from mpl_toolkits.mplot3d import axes3d
fig = figure()
ax = fig.gca(projection='3d')
ax.plot(Xest[0],Xest[1],Xest[2],'ko')
ax.plot(Xtrue[0],Xtrue[1],Xtrue[2],'r.')
axis('equal')

show()
```

This will triangulate the points in correspondence from the first two views and print out the coordinates of the first three points to the console before plotting the recovered 3D points next to the true values. The printout looks like this:

```
[[ 1.03743725  1.56125273  1.40720017]
 [-0.57574987 -0.55504127 -0.46523952]
 [ 3.44173797  3.44249282  7.53176488]
 [ 1.          1.          1.        ]]
[[ 1.0378863   1.5606923   1.4071907 ]
 [-0.54627892 -0.5211711  -0.46371818]
 [ 3.4601538   3.4636809   7.5323397 ]
 [ 1.          1.          1.        ]]
```

The estimated points are close enough.

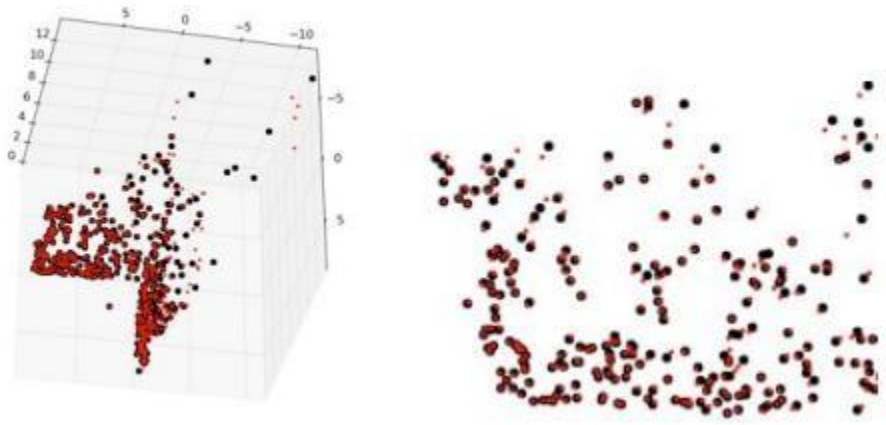


Figure 5.5: Triangulated points using camera matrices and point correspondences. The estimated points are shown with black circles and the true points with red dots. (left) view from above and to the side. (right) close up of the points from one of the building walls.

3.1.3 Multiple View Reconstruction

To compute an actual 3D reconstruction from a pair of images. Computing a 3D reconstruction like this is usually referred to as structure from motion (SfM) since the motion of a camera (or cameras) give you 3D structure.

Assuming the camera has been calibrated, the steps are as follows:

- Detect feature points and match them between the two images.
- Compute the fundamental matrix from the matches.
- Compute the camera matrices from the fundamental matrix.

- Triangulate the 3D points.

The goal of multiview 3D reconstruction is to infer geometrical structure of a scene captured by a collection of images. Usually the camera position and internal parameters are assumed to be known or they can be estimated from the set of images. By using multiple images, 3D information can be (partially) recovered by solving a pixel-wise correspondence problem. Since automatic correspondence estimation is usually ambiguous and incomplete further knowledge (prior knowledge) about the object is necessary. A typical prior is assume that the object surface is smooth.

Fig: 5.1 Multiview 3D reconstruction

```
def compute_fundamental_normalized(x1,x2):
  """ Computes the fundamental matrix from corresponding points
    (x1,x2 3*n arrays) using the normalized 8 point algorithm. """

  n = x1.shape[1]
  if x2.shape[1] != n:
    raise ValueError("Number of points don't match.")

  # normalize image coordinates
  x1 = x1 / x1[2]
  mean_1 = mean(x1[:2],axis=1)
  S1 = sqrt(2) / std(x1[:2])
  T1 = array([[S1,0,-S1*mean_1[0]],[0,S1,-S1*mean_1[1]],[0,0,1]])
  x1 = dot(T1,x1)

  x2 = x2 / x2[2]
  mean_2 = mean(x2[:2],axis=1)
  S2 = sqrt(2) / std(x2[:2])
  T2 = array([[S2,0,-S2*mean_2[0]],[0,S2,-S2*mean_2[1]],[0,0,1]])
  x2 = dot(T2,x2)

  # compute F with the normalized coordinates
  F = compute_fundamental(x1,x2)

  # reverse normalization
  F = dot(T1.T,dot(F,T2))

  return F/F[2,2]
```

The fit() method now selects eight points and uses a normalized version of the eight point algorithm.

3D reconstruction example

In this section we will see a complete example of reconstructing a 3D scene from start to finish. We will use two images taken with a camera with known calibration.

Let's split the code up in a few chunks so that it is easier to follow. First we extract features, match them and estimate a fundamental matrix and camera matrices.

```
import homography
import sfm
import sift

# calibration
K = array([[2394,0,932],[0,2398,628],[0,0,1]])
```

Figure 5.7: Example image pair of a scene where the images are taken at different viewpoints. This function normalizes the image points to zero mean and fixed variance.

```
# load images and compute features
im1 = array(Image.open('alcatraz1.jpg'))
sift.process_image('alcatraz1.jpg','im1.sift')
l1,d1 = sift.read_features_from_file('im1.sift')

im2 = array(Image.open('alcatraz2.jpg'))
sift.process_image('alcatraz2.jpg','im2.sift')
l2,d2 = sift.read_features_from_file('im2.sift')


# match features
matches = sift.match_twosided(d1,d2)
ndx = matches.nonzero()[0]

# make homogeneous and normalize with inv(K)
x1 = homography.make_homog(l1[ndx,:2].T)
ndx2 = [int(matches[i]) for i in ndx]
x2 = homography.make_homog(l2[ndx2,:2].T)

x1n = dot(inv(K),x1)
x2n = dot(inv(K),x2)


# estimate E with RANSAC
model = sfm.RansacModel()
E,inliers = sfm.F_from_ransac(x1n,x2n,model)

# compute camera matrices (P2 will be list of four solutions)
P1 = array([[1,0,0,0],[0,1,0,0],[0,0,1,0]])
P2 = sfm.compute_P_from_essential(E)
```

The calibration is known so here we just hardcode the K matrix at the beginning. As in earlier examples, we pick out the points that belong to matches. After that we normalize them with K1 and run the RANSAC estimation with the normalized eight point algorithm. Since the points are normalized, this gives us an essential matrix. We make sure to keep the index of the inliers, we will need them. From the essential matrix we compute the four possible solutions of the second camera matrix.

From the list of camera matrices, we pick the one that has the most scene points in front of both cameras after triangulation.

```
# pick the solution with points in front of cameras
ind = 0
maxres = 0
for i in range(4):
  # triangulate inliers and compute depth for each camera
  X = sfm.triangulate(x1n[:,inliers],x2n[:,inliers],P1,P2[i])
  d1 = dot(P1,X)[2]
  d2 = dot(P2[i],X)[2]
  if sum(d1>0)+sum(d2>0) > maxres:
    maxres = sum(d1>0)+sum(d2>0)
    ind = i
    infront = (d1>0) & (d2>0)


# triangulate inliers and remove points not in front of both cameras
X = sfm.triangulate(x1n[:,inliers],x2n[:,inliers],P1,P2[ind])
X = X[:,infront]
```

We loop through the four solutions and each time triangulate the 3D points corresponding to the inliers. The sign of the depth is given by the third value of each image point after projecting the triangulated X back to the images. We keep the index with the most positive depths and also store a boolean for each point in the best solution so that we can pick only the ones that actually are in front. Due to noise and errors in all of the estimations done, there is a risk that some points still are behind one camera, even with the correct camera matrices. Once we have the right solution, we triangulate the inliers and keep the points in front of the cameras.

```
# plot the projection of X
import camera

# project 3D points
cam1 = camera.Camera(P1)
cam2 = camera.Camera(P2[ind])
x1p = cam1.project(X)
x2p = cam2.project(X)

# reverse K normalization
x1p = dot(K,x1p)
x2p = dot(K,x2p)


figure()
imshow(im1)
gray()
plot(x1p[0],x1p[1],'o')
plot(x1[0],x1[1],'r.')
axis('off')

figure()
imshow(im2)
gray()
plot(x2p[0],x2p[1],'o')
plot(x2[0],x2[1],'r.')
axis('off')
show()
```

Now we can plot the reconstruction.

The result looks like Figure: 5.8. As you can see, the reprojected points (blue) don't exactly match the original feature locations (red) but they are reasonably close. It is possible to further refine the camera matrices to improve the reconstruction and reprojection but that is outside the scope of this simple example.

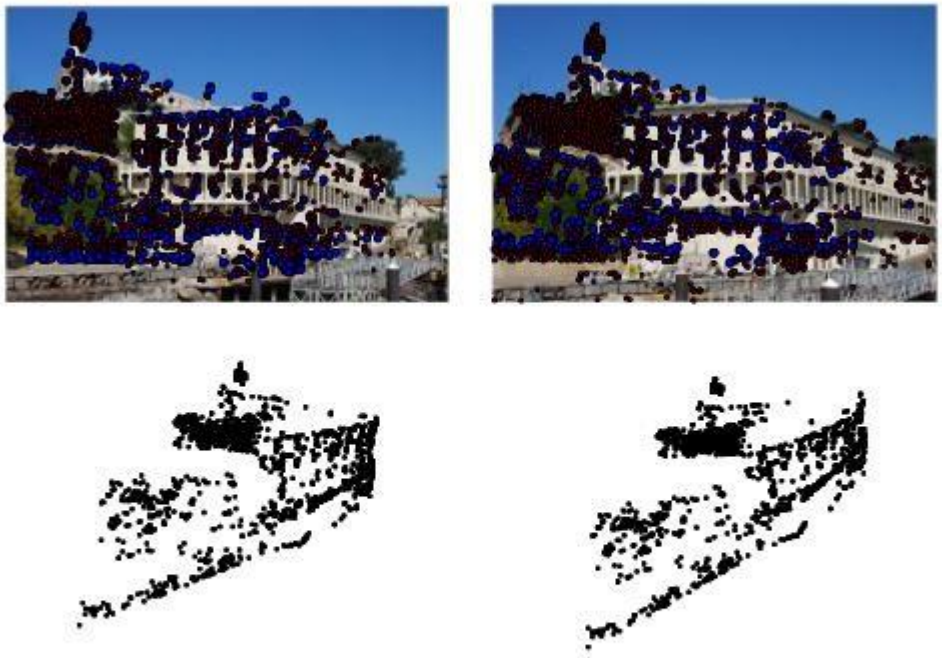
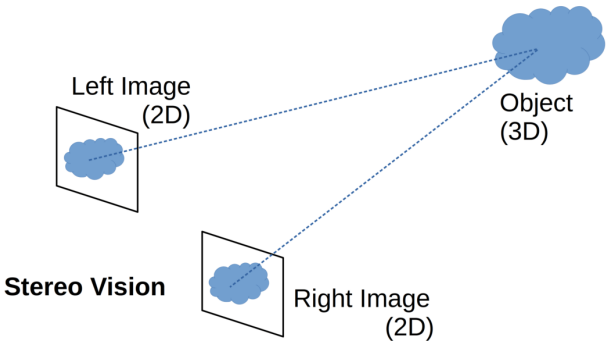


Figure 5.8: Example of computing a 3D reconstruction from a pair of images using image matches. (top) the two images with feature points shown in red and reprojected reconstructed 3D points shown in blue. (bottom) the 3D reconstruction.

3.1.4 Stereo Images

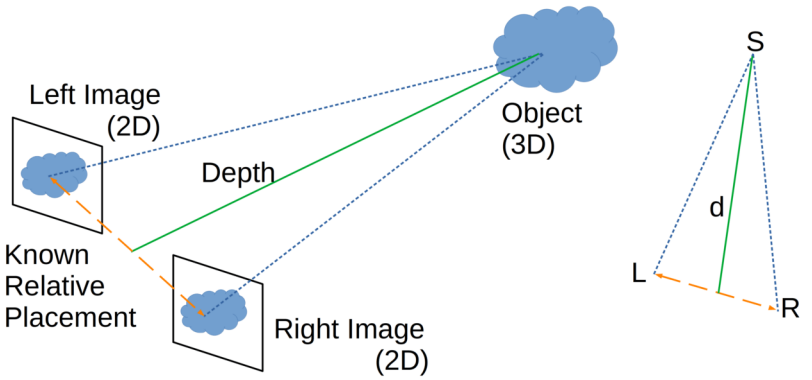
A special case of multi-view imaging is stereo vision (or stereo imaging) where two cameras are observing the same scene with only a horizontal (sideways) displacement between the cameras. When the cameras are configured so that the two images have the same image plane with the image rows vertically aligned, the image pair is said to be rectified. This is common in robotics and such a setup is often called a stereo rig.

Computer stereo vision is the extraction of 3D information from 2D images, such as those produced by a CCD camera. It compares data from multiple perspectives and combines the relative positions of things in each view. As such, we use stereo vision in applications like advanced driver assistance systems and robot navigation.



Perceiving Depth

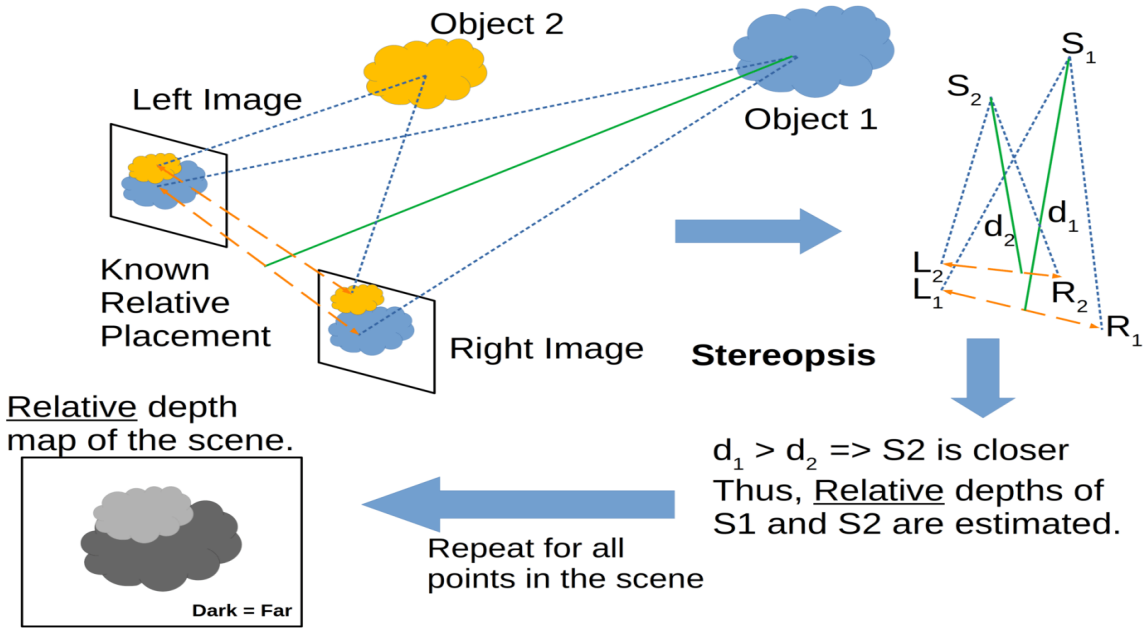
Let's suppose there are left and right cameras, both producing a 2D image of a scene. Let S be a point on a real-world (3D) object in the scene:



To determine the depth of S in the composite 3D image, we first find two pixels *L* and *R* in the left and right 2D images that correspond to it. We can assume that we know the relative positioning of the two cameras. **The computing system estimates the depth *d* by triangulation using the prior knowledge of the relative distance between the cameras.**

Computer Systems Achieve Stereo Vision

We need to estimate each point's depth to produce a 3D image from two-dimensional ones.



From there, we can determine the points' relative depths and get a depth map:

A depth map is an image (or image channel) that contains the data on the separation between the surfaces of scene objects from a viewpoint. This is a common way to represent scene depths in 3D computer graphics and computer vision.

The Geometrical Basis of Stereo Vision

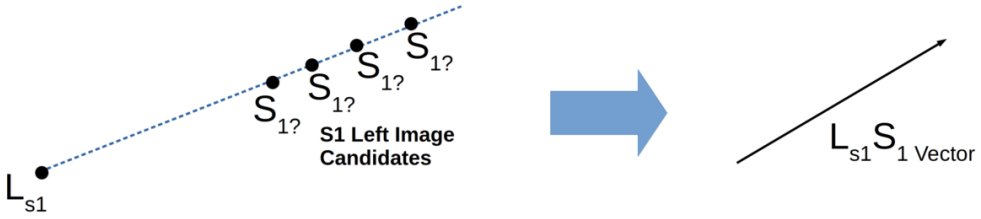
Epipolar geometry is the geometry of stereo vision. There are a variety of geometric relationships between the 3D points and their projections onto the 2D images. These relationships have been developed for the pinhole camera model. We assume that we can represent normal using these relationships.

A 3D item is projected into a 2D (planar) projective space when captured (projected) in an image. The issue with this so-called "planar projection" is that it causes the loss of depth.

The disparity between the two stereo pictures is the apparent motion of things. If we close one eye and open it quickly while keeping the other closed, we'll observe that objects near us move quite a bit, whereas those farther away move barely at all. We refer to this phenomenon as "discrepancy."

The Direction Vector

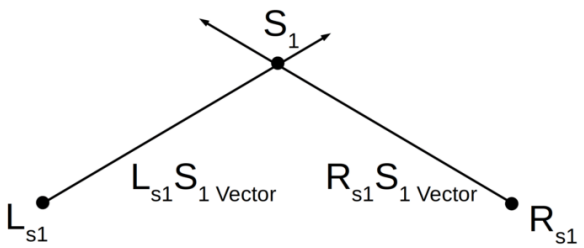
In epipolar geometry, a direction vector is a vector in three dimensions emanating from a pixel in the image:



The direction vector, as the name suggests, is the direction from where the light ray arrives at the pixel sensor. This line thus carries all the 3D points that could be candidate sources for the 2D pixels in the image. In the above figure, the direction vector Ls1S1 originates from the point Ls1, which is the "left" 2D pixel corresponding to the 3D point $_{S1}$ in the scene.

Direction Vector Intersection

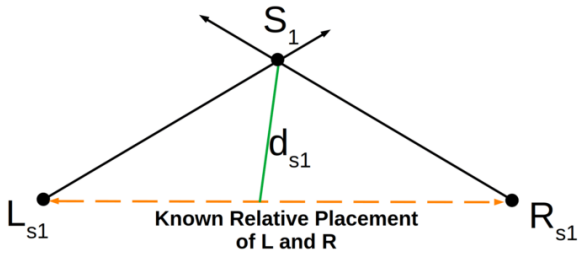
Direction vectors for a 3D point in the scene will cast corresponding 2D points in the images taken from different views. **A stereo pair of images will thus have direction vectors emanating from the 2D pixels representing a common 3D point in the 3D scene.** All points on a direction vector are candidate sources. **Since two vectors can intersect at only one unique point, we take the intersection point as the source:**



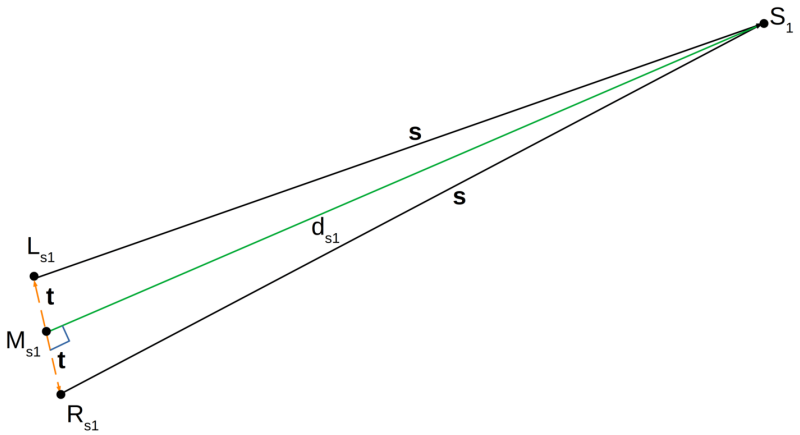
In the above figure, the direction vectors from the left and right images (Ls1S1 and Rs1S1, respectively) intersect at the single source S1. This 3D source point in the scene is the point from where light rays cast image pixels Ls1 and Rs1 in the left and right images.

Depth Calculation

We assume that we know the distance between cameras and that it's very small compared to the distance between the object and the cameras. Under that assumption, we can determine the location of the 3D point in space by triangulation. The depth is a perpendicular cast on the line joining the two cameras:



The above image shows the actual depth ds1for the point from the line joining the two cameras. Let's note that the angle between the line d_{s1} and the line Ls1Rs1is not exactly 90 degrees. In reality, however, the distance Ls1Rs1, is very small compared to ds1. This results in the angle between the line ds1and the line Ls1Rs1being approximately 90 degrees. Since we determined the location of S1 by triangulation, and we know the relative distance Ls1Rs1, we can calculate the depth ds1 using the Pythagorean theorem:



Since s is very large compared to t, the angle \angle S1 Ms1Rs1approaches 90^{\circ}. Lengths Ls1Ms1and Ms1Rs1 are almost the same (denoted by t). Also, lengths Ls1S1 and Rs1S1 are almost the same (denoted by s). Applying the Pythagorean theorem,

$$d_{s1} = \sqrt{s^2 - t^2}$$

we get s^2 = {d{s1}}^2 + t^2. Solving for the depth of point S1 we get:

3.2 Clustering Images

This chapter introduces several clustering methods and shows how to use them for clustering images for finding groups of similar images. Clustering can be used for recognition, for dividing data sets of images and for organization and navigation. We also look at using clustering for visualizing similarity between images.

3.2.1 K-means Clustering

K-means is a very simple clustering algorithm that tries to partition the input data in k clusters. K-means works by iteratively refining an initial estimate of class centroids as follows:

- Initialize centroids μi, i = 1 ...k, randomly or with some guess.
- Assign each data point to the class ci of its nearest centroid.
- Update the centroids as the average of all data points assigned to that class.
- Repeat 2 & 3 until convergence.

$$V = \sum_{i=1}^{k} \sum_{\mathbf{x}_j \in c_i} (\mathbf{x}_j - \boldsymbol{\mu}_i)^2$$

K-means tries to minimize the total within-class variance

where xj are the data vectors. The algorithm above is a heuristic refinement algorithm that works fine for most cases but does not guarantee that the best solution is found. To avoid the effects of choosing a bad centroid initialization, the algorithm is often run several times with different initialization centroids. Then the solution with lowest variance V is selected.

The main drawback of this algorithm is that the number of clusters needs to be decided beforehand and an inappropriate choice will give poor clustering results. The benefits are that it is simple to implement, it is parallelizable and works well for a large range of problems without any need for tuning.

To define a target number k, which refers to the number of centroids you need in the dataset. A centroid is the imaginary or real location representing the center of the cluster. This algorithm will allow us to group our feature vectors into k clusters. Each cluster should contain images that are visually similar.

```
import imtools
import pickle
from scipy.cluster.vq import *

# get list of images
imlist = imtools.get_imlist('selected_fontimages/')
imnbr = len(imlist)

# load model file
with open('a_pca_modes.pkl','rb') as f:
  immean = pickle.load(f)
  V = pickle.load(f)

# create matrix to store all flattened images
immatrix = array([array(Image.open(im)).flatten()
                  for im in imlist],'f')

# project on the 40 first PCs
immean = immean.flatten()
projected = array([dot(V[:40],immatrix[i]-immean) for i in range(imnbr)])

# k-means
projected = whiten(projected)
centroids,distortion = kmeans(projected,4)

code,distance = vq(projected,centroids)
```

projected = array([dot(V[[0,2]],immatrix[i]-immean) for i in range(imnbr)])

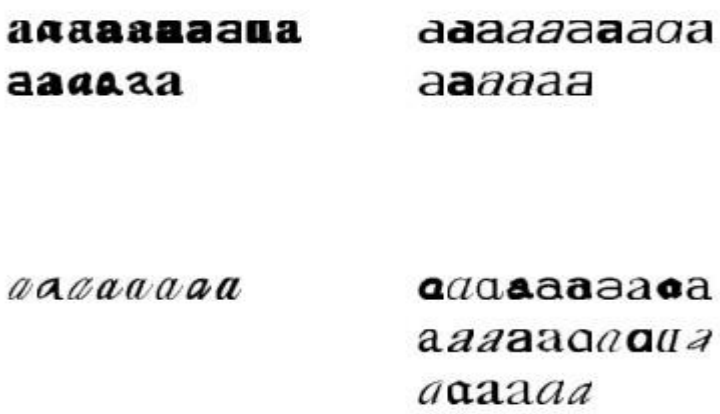
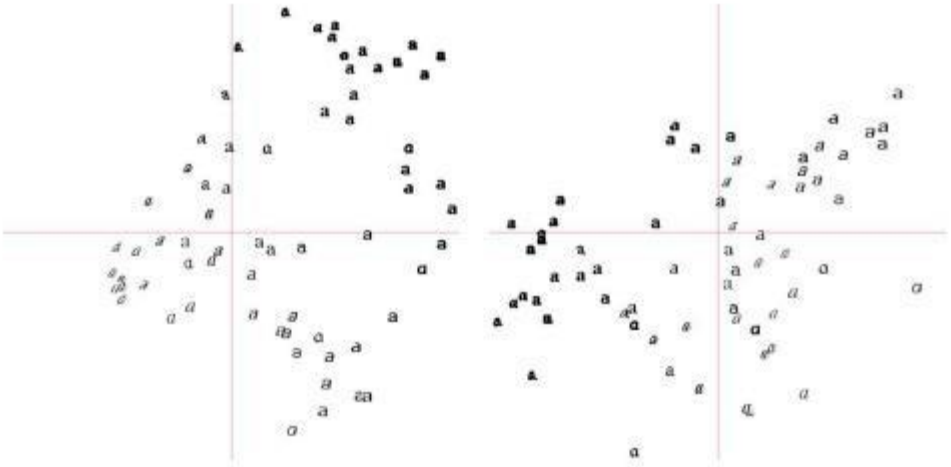


Figure: 6.2

An example of k-means clustering with k = 4 of the font images using 40 principal components.

For the visualization we will use the ImageDraw module in PIL. Assuming that you have the projected images and image list as above, the following short script will generate a plot like the one.

```
from PIL import Image, ImageDraw

# height and width
h,w = 1200,1200

# create a new image with a white background
img = Image.new('RGB',(w,h),(255,255,255))
draw = ImageDraw.Draw(img)

# draw axis
draw.line((0,h/2,w,h/2),fill=(255,0,0))
draw.line((w/2,0,w/2,h),fill=(255,0,0))

# scale coordinates to fit
scale = abs(projected).max(0)
scaled = floor(array([ (p / scale) * (w/2-20,h/2-20) +
                       (w/2,h/2) for p in projected]))

# paste thumbnail of each image
```



3.2.2 Hierarchical Clustering

Hierarchical clustering (or agglomerative clustering) is another simple but powerful clustering algorithm. The idea is to build a similarity tree based on pairwise distances. The algorithm starts with grouping the two closest objects (based on the distance between feature vectors) and creates an "average" node in a tree with the two objects as children. Then the next closest pair is found among the remaining objects but also including any average nodes, and so on. At each node the distance between the two children is also stored. Clusters can then be extracted by traversing this tree and stopping at nodes with distance smaller some threshold that then determines the cluster size. Hierarchical clustering has several benefits. For example, the tree structure can be used to visualize relationships and show how clusters are related. A good feature vector will give a nice separation in the tree. Another benefit is that the tree can be reused with different cluster

thresholds without having to recompute the tree. The drawback is that one needs to choose a threshold if the
actual clusters are needed.

Types of Hierarchical Clustering

Basically, there are two types of hierarchical Clustering:

1. Agglomerative Clustering
2. Divisive clustering

Hierarchical Agglomerative Clustering

It is also known as the bottom-up approach or hierarchical agglomerative clustering (HAC). A structure that is more informative than the unstructured set of clusters returned by flat clustering. This clustering algorithm does not require us to prespecify the number of clusters. Bottom-up algorithms treat each data as a singleton cluster at the outset and then successively agglomerate pairs of clusters until all clusters have been merged into a single cluster that contains all data.

Algorithm:

given a dataset (d1, d2, d3, dN) of size N

compute the distance
matrix for i=1 to N:
as the distance matrix is symmetric about
the primary diagonal so we compute only
lower # part of the primary diagonal
for j=1 to i:
dis_mat[i][j] = distance[di, dj]
each data point is a singleton
cluster repeat
merge the two cluster having minimum
distance update the distance matrix
until only a single cluster remains

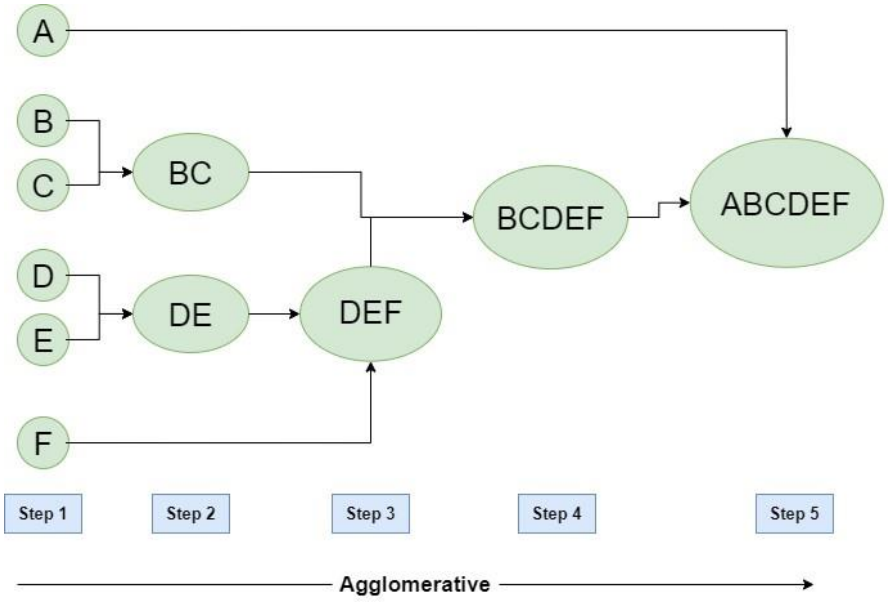


Fig: Hierarchical Agglomerative Clustering

- Consider each alphabet as a single cluster and calculate the distance of one cluster from all the other clusters.
- In the second step, comparable clusters are merged together to form a single cluster. Let's say cluster (B) and cluster (C) are very similar to each other therefore we merge them in the second step similarly to cluster (D) and (E) and at last, we get the clusters [(A), (BC), (DE), (F)]
- We recalculate the proximity according to the algorithm and merge the two nearest clusters([(DE), (F)]) together to form new clusters as [(A), (BC), (DEF)]
- Repeating the same process; The clusters DEF and BC are comparable and merged together to form a new cluster. We're now left with clusters [(A), (BCDEF)].
- At last, the two remaining clusters are merged together to form a single cluster [(ABCDEF)].

Example Program

```
from sklearn.cluster import
AgglomerativeClustering import numpy as np
# randomly chosen dataset
X = np.array([[1, 2], [1, 4], [1, 0],[4, 2], [4, 4], [4, 0]])
# here we need to mention the number of
clusters # otherwise the result will be a
single cluster
# containing all the data
clustering =
AgglomerativeClustering(n_clusters=2).fit(X) # print
the class labels
print(clustering.labels_)
```

Output

[1, 1, 1, 0, 0, 0]

Hierarchical Divisive clustering

It is also known as a top-down approach. This algorithm also does not require to prespecify the number of clusters. Top-down clustering requires a method for splitting a cluster that contains the whole data and proceeds by splitting clusters recursively until individual data have been split into singleton clusters.

Algorithm:

given a dataset (d_1, d_2, d_3, d_N) of size N

at the top we have all data in one cluster

the cluster is split using a flat clustering method eg. K-Means etc

repeat

choose the best cluster among all the clusters

to split split that cluster by the flat clustering

algorithm **until** each data is in its own singleton

cluster

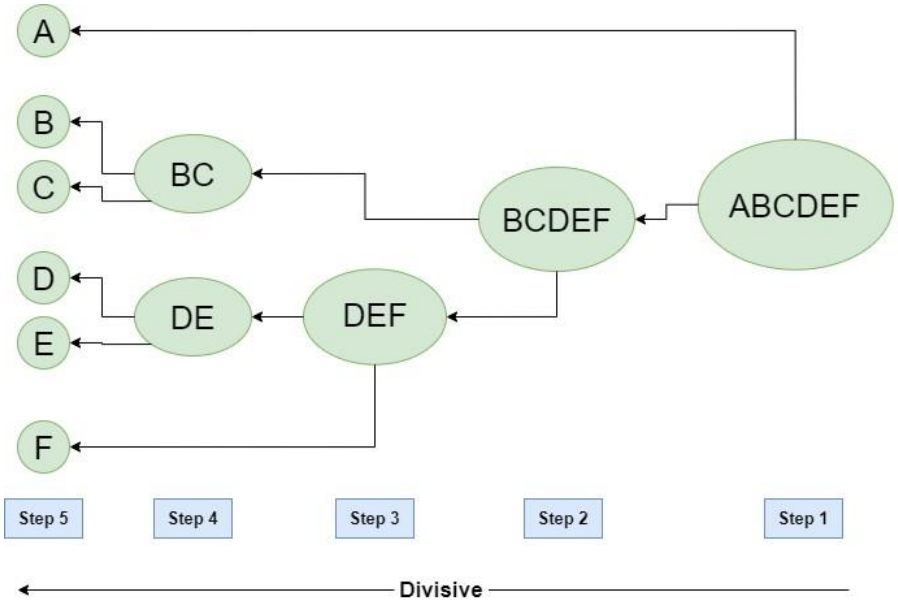
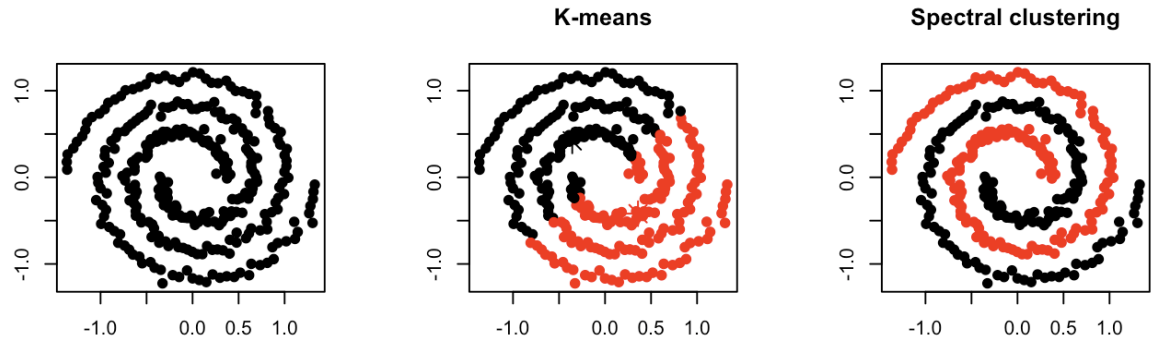


Fig: Hierarchical Divisive clustering

3.2.3 Spectral Clustering

Spectral Clustering is a variant of the clustering algorithm that uses the connectivity between the data points to form the clustering. It uses eigenvalues and eigenvectors of the data matrix to forecast the data into lower dimensions space to cluster the data points. It is based on the idea of a graph representation of data where the data point are represented as nodes and the similarity between the data points are represented by an edge.



Building the Similarity Graph Of The Data: This step builds the Similarity Graph in the form of an adjacency matrix which is represented by A. The adjacency matrix can be built in the following manners:

- **Epsilon-neighbourhood Graph:** A parameter epsilon is fixed beforehand. Then, each point is connected to all the points which lie in its epsilon-radius. If all the distances between any two points are similar in scale then typically the weights of the edges ie the distance between the two points are not stored since they do not provide any additional information. Thus, in this case, the graph built is an undirected and unweighted graph.
- **K-Nearest Neighbours:** A parameter k is fixed beforehand. Then, for two vertices u and v, an edge is directed from u to v only if v is among the k-nearest neighbours of u. Note that this leads to the formation of a weighted and directed graph because it is not always the case that for each u having v as one of the k-nearest neighbours, it will be the same case for v having u among its k-nearest neighbours. To make this graph undirected, one of the following approaches is followed:-
- Direct an edge from u to v and from v to u if either v is among the k-nearest neighbours of u **OR** u is among the k-nearest neighbours of v.
- Direct an edge from u to v and from v to u if v is among the k-nearest neighbours of u **AND** u is among the k-nearest neighbours of v.
- **Fully-Connected Graph:** To build this graph, each point is connected with an undirected edge-weighted by the distance between the two points to every other point. Since this approach is used to model the local neighbourhood relationships thus typically the Gaussian similarity metric is used to calculate the distance.

Projecting the data onto a lower Dimensional Space: This step is done to account for the possibility that members of the same cluster may be far away in the given dimensional space. Thus the dimensional space is reduced so that those points are closer in the

reduced dimensional space and thus can be clustered together by a traditional clustering algorithm. It is done by computing the **Graph Laplacian Matrix**.

Python Code For Graph Laplacian Matrix

To compute it though first, the degree of a node needs to be defined. The degree of the ith node is given by

Note that wij is the edge between the nodes i and j as defined in the adjacency matrix above.

Clustering the Data: This process mainly involves clustering the reduced data by using any traditional clustering technique – typically K-Means Clustering. First, each node is assigned a row of the normalized of the Graph Laplacian Matrix. Then this data is clustered using any traditional technique. To transform the clustering result, the node identifier is retained.

Properties:

1. **Assumption-Less:** This clustering technique, unlike other traditional techniques do not assume the data to follow some property. Thus this makes this technique to answer a more- generic class of clustering problems.
2. **Ease of implementation and Speed:** This algorithm is easier to implement than other clustering algorithms and is also very fast as it mainly consists of mathematical computations.
3. **Not-Scalable:** Since it involves the building of matrices and computation of eigenvalues and eigenvectors it is time-consuming for dense datasets.
4. **Dimensionality Reduction**: The algorithm uses eigenvalue decomposition to reduce the dimensionality of the data, making it easier to visualize and analyze.
5. **Cluster Shape**: This technique can handle non-linear cluster shapes, making it suitable for a wide range of applications.
6. **Noise Sensitivity**: It is sensitive to noise and outliers, which may affect the quality of the resulting clusters.
7. **Number of Clusters:** The algorithm requires the user to specify the number of clusters beforehand, which can be challenging in some cases.
8. **Memory Requirements:** The algorithm requires significant memory to store the similarity matrix, which can be a limitation for large datasets.

Advantages of Spectral Clustering

1. Scalability: Spectral clustering can handle large datasets and high-dimensional data, as it reduces the dimensionality of the data before clustering.
2. Flexibility: Spectral clustering can be applied to non-linearly separable data, as it does not rely on traditional distance-based clustering methods.
3. Robustness: Spectral clustering can be more robust to noise and outliers in the data, as it considers the global structure of the data, rather than just local distances between data points.

Disadvantages of Spectral Clustering

1. Complexity: Spectral clustering can be computationally expensive, especially for large datasets, as it requires the calculation of eigenvectors and eigenvalues.
2. Model selection: Choosing the right number of clusters and the right similarity matrix can be challenging and may require expert knowledge or trial and error.

```
from scipy.cluster.vq import *

n = len(projected)

# compute distance matrix
S = array([[ sqrt(sum((projected[i]-projected[j])**2))
          for i in range(n) ] for j in range(n)], 'f')

# create Laplacian matrix
rowsum = sum(S,axis=0)
D = diag(1 / sqrt(rowsum))
I = identity(n)
L = I - dot(D,dot(S,D))

# compute eigenvectors of L
U,sigma,V = linalg.svd(L)

k = 5
# create feature vector from k first eigenvectors
# by stacking eigenvectors as columns
features = array(V[:k]).T
```

Output

Figure: Spectral clustering of font images using the eigenvectors of the Laplacian matrix.

Unit Summary

Multiple View Geometry focuses on understanding and computing the 3D structure from multiple 2D images, with critical concepts like epipolar geometry and stereo imaging playing central roles. Clustering techniques such as K-Means, Hierarchical, and Spectral Clustering are used to group images based on their feature similarity, helping in tasks like image segmentation and classification.

Let us sum up

Multiple View Geometry: The study of how multiple images of a scene, taken from different viewpoints, can be used to reconstruct 3D structures and understand spatial relationships in the scene.

Camera Calibration: The process of determining the internal parameters of a camera, such as focal length and distortion coefficients, necessary for accurate 3D reconstruction and image analysis.

Epipolar Geometry: The geometric relationship between two images of the same scene, where corresponding points in each image lie on epipolar lines. It is fundamental for stereo vision and 3D reconstruction.

Stereo Vision: A technique that uses two or more cameras to capture images of the same

scene from different angles to compute depth information and create a 3D representation of the scene.

Self Assesment Questions:

1. Briefly Explain the Concept of Epipolar Geometry with diagram.
2. Computing with Cameras and 3D Structure in detail.
3. Explain the Multiple View Reconstruction in detail.
4. Briefly Explain the Concept of Stereo Images with diagram.
5. Explain the K-Means Clustering algorithm.
6. Briefly Explain the Hierarchical Clustering in detail.
7. Explain the concept of Spectral Clustering.

Check your progress

Question 1

Which image augmentation technique involves rotating an image by a certain angle?

A) Translation
B) Scaling
C) Rotation
D) Flipping

Answer: C) Rotation

Explanation: Rotation involves turning the image around its center by a specified angle, which helps the model learn to recognize objects from different orientations.

Question 2

What is the primary purpose of image flipping in augmentation?

A) To change the color balance of the image
B) To adjust the brightness of the image
C) To create a mirror image of the original image
D) To increase the resolution of the image

Answer: C) To create a mirror image of the original image

Explanation: Image flipping (either horizontally or vertically) creates a mirror image of the

original, which helps the model become invariant to such transformations.

Question 3

Which of the following techniques is used to adjust the size of an image?

A) Scaling
B) Translation
C) Rotation
D) Cropping

Answer: A) Scaling

Explanation: Scaling changes the size of the image by a specified factor, allowing the model to handle objects of different sizes.

Question 4

What does the term "translation" refer to in the context of image augmentation?

A) Changing the image‘s color scheme
B) Shifting the image in the x or y direction
C) Rotating the image
D) Flipping the image horizontally

Answer: B) Shifting the image in the x or y direction

Explanation: Translation involves moving the image along the x or y axis, which helps the model learn to recognize objects even when they are not centered.

Question 5

Which augmentation technique is particularly useful for handling variations in object size and maintaining object position?

A) Cropping
B) Normalization
C) Rotation
D) Scaling

Answer: D) Scaling

Explanation: Scaling adjusts the size of the image, which is crucial for learning about objects at different scales and maintaining positional context.

Question 6

In image augmentation, what does the term "crop" refer to?

A) Rotating a part of the image
B) Changing the image‘s contrast
C) Removing a portion of the image
D) Adding noise to the image

Answer: C) Removing a portion of the image

Explanation: Cropping involves cutting out a portion of the image, which can help the model focus on specific regions and learn object details better.

Question 7

What is the purpose of applying random brightness adjustment during image augmentation?

A) To increase the image‘s contrast
B) To simulate varying lighting conditions
C) To reduce the image‘s noise
D) To resize the image

Answer: B) To simulate varying lighting conditions

Explanation: Random brightness adjustment helps the model learn to recognize objects under different lighting conditions by varying the image‘s brightness.

Question 8

Which augmentation technique would be best for introducing variations in the sharpness of images?

A) Gaussian Blur

B) Edge Detection

C) Histogram Equalization

D) Rotation

Answer: A) Gaussian Blur

Explanation: Gaussian blur is used to adjust the sharpness of an image, creating variations that help the model generalize better to images with different levels of sharpness.

Question 9

What effect does the —shearingǁ technique have on an image during augmentation?

A) It changes the image's color balance

B) It distorts the image by shifting the pixels along a particular axis

C) It enhances the edges in the image

D) It smooths out the image

Answer: B) It distorts the image by shifting the pixels along a particular axis

Explanation: Shearing distorts the image by slanting it along a specific direction, which helps the model learn to recognize objects even when they are skewed or distorted.

Question 10

How does image augmentation benefit machine learning models?

A) By increasing the model's size

B) By reducing the computational complexity

C) By expanding the training dataset with diverse variations

D) By simplifying the model architecture

Answer: C) By expanding the training dataset with diverse variations

Explanation: Image augmentation helps by creating variations of the original images, effectively expanding the training dataset and improving the model‘s robustness and generalization ability.

Glossary

Structure-from-Motion (SfM): A method for reconstructing 3D structures from a series of 2D images taken as the camera moves through space. SfM estimates both camera motion and 3D scene structure simultaneously.

Fundamental Matrix: A matrix that describes the epipolar geometry between two views, representing the intrinsic projective geometry of the image pair.

Essential Matrix: A matrix that captures the relative rotation and translation between two camera views, essential for reconstructing 3D points from corresponding image points.

Homography: A transformation matrix that relates the coordinates of points in one image to their corresponding coordinates in another image, assuming both images are of the same planar surface.

Triangulation: The process of determining the 3D position of a point by intersecting the lines of sight from multiple images taken from different viewpoints.
Depth Map: A representation of the distance of surfaces within a scene from the camera, often generated from stereo vision or structured light systems.

Multi-View Stereo (MVS): Techniques that use multiple images from different viewpoints to reconstruct dense 3D models of a scene by estimating the depth of each point.
Image Clustering: The process of grouping a set of images into clusters based on their visual similarity, using features extracted from the images.
Feature Extraction: The process of identifying and quantifying important aspects of images, such as color, texture, and shape, which are used for clustering.

K-Means Clustering: A popular clustering algorithm that partitions a set of images into K clusters by minimizing the variance within each cluster.

Hierarchical Clustering: A clustering method that builds a hierarchy of clusters, either by iteratively merging smaller clusters into larger ones (agglomerative) or by splitting larger clusters into smaller ones (divisive).
Spectral Clustering: A technique that uses eigenvalues of similarity matrices to reduce

dimensionality before applying clustering algorithms, useful for handling complex cluster shapes.

DBSCAN (Density-Based Spatial Clustering of Applications with Noise): A clustering algorithm that groups together points that are closely packed while marking points in low-density regions as outliers.

Fuzzy Clustering: A method where each image can belong to multiple clusters with varying degrees of membership, as opposed to hard clustering where each image belongs to only one cluster.

Dimensionality Reduction: Techniques such as Principal Component Analysis (PCA) or t-Distributed Stochastic Neighbor Embedding (t-SNE) used to reduce the number of features or dimensions in the data, making clustering more effective.

Books

1."Multiple View Geometry in Computer Vision‖, Authors: Richard Hartley, Andrew Zisserman
2."Computer Vision: Algorithms and Applications", Author: Richard Szeliski
3."Pattern Recognition and Machine Learning", Author: Christopher M. Bishop

Open source e-content links

https://discuss.pytorch.org/t/data-agumentation-in-3d-images/86289

https://www.mathworks.com/matlabcentral/answers/519306-augmentation-of-data-in-image-processing

UNIT – III END

4.1 Searching Images

UNIT – IV

This chapter shows how to use text mining techniques to search for images based on their visual content. The basic ideas of using visual words are presented and the details of a complete setup are explained and tested on an example image data set.

4.1.1 Content-based Image Retrieval

Content-based image retrieval (CBIR) deals with the problem of retrieving visually similar images from a (large) database of images. This can be images with similar color, similar textures or similar objects or scenes, basically any information contained in the images themselves.

For high-level queries, like finding similar objects, it is not feasible to do a full comparison (for example using feature matching) between a query image and all images in the database. It would simply take too much time to return any results if the database is large. In the last couple of years, researchers have successfully introduced techniques from the world of text mining for CBIR problems making it possible to search millions of images for similar content.

Inspiration from text mining

The vector space model The vector space model is a model for representing and searching text documents. As we will see, it can be applied to essentially any kind of objects, including images. The name comes from the fact that text documents are represented with vectors that are histograms of the word frequencies in the text1. In other words, the vector will contain the number of occurrences of every word (at the position corresponding to that word) and zeros everywhere else. This model is also called a bag-of-word representation since order and location of words is ignored. Documents are indexed by doing a word count to construct the document histogram vector v, usually with common words like "the", "and", "is" etc. ignored. These common words are called stop words. To compensate for document length, the vectors can be normalized to unit length by dividing with the total histogram sum. The individual components of the histogram vector are usually weighted according to the importance of each word. Usually, the importance of a word increases proportional to how often it appears in the document but decreases if the word is common in all documents in a data set (or "corpus") The most common weighting

is tf-idf weighting (term frequency - inverse document frequency) where the term frequency of a word w in document d, is

where n_w is the number of occurrences of w in d. To normalize, this is divided by the total

$$\mathrm{tf}_{w,d} = \frac{n_w}{\sum_j n_j}$$

number of occurrences of all words in the document.

$$\mathrm{idf}_{w,d} = \log \frac{|D|}{|\{d : w \in d\}|}$$

The inverse document frequency is

where |D| is the number of documents in the corpus D and the denominator the number of documents d in D containing w. Multiplying the two gives the tf-idf weight which is then the elements in v.

4.1.2 Visual Words

To apply text mining techniques to images, we first need to create the visual equivalent of words. This is usually done using local descriptors like the SIFT descriptor. The idea is to quantize the descriptor space into a number of typical examples and assign each descriptor in the image to one of those examples. These typical examples are determined by analyzing a training set of images and can be considered as visual words and the set of all words is then a visual vocabulary (sometimes called a visual codebook). This vocabulary can be created specifically for a given problem or type of images or just try to represent visual content in general.

The visual words are constructed using some clustering algorithm applied to the feature descriptors extracted from a (large) training set of images. The most common choice is k-means2, which is what we will use here. Visual words are nothing but a collection of vectors in the given feature descriptor space, in the case of k-means they are the cluster centroids. Representing an image with a histogram of visual words is then called a bag of visual words model.

Creating a vocabulary

To create a vocabulary of visual words we first need to extract descriptors. Here we will use the SIFT descriptor. Running the following lines of code, with imlist, as usual, containing the filenames of the images, will give you descriptor files for each image. Create a file vocabulary.py and add the following code for a vocabulary class and a method for training a vocabulary on some training image data.

```
    nbr_images = len(imlist)
    featlist = [ imlist[i][:-3]+'sift' for i in range(nbr_images)]

    for i in range(nbr_images):
      sift.process_image(imlist[i],featlist[i])
    from scipy.cluster.vq import *
    import vlfeat as sift


    class Vocabulary(object):

      def __init__(self,name):
        self.name = name
        self.voc = []
        self.idf = []
        self.trainingdata = []
def train(self,featurefiles,k=100,subsampling=10):
  """ Train a vocabulary from features in files listed
    in featurefiles using k-means with k number of words.
    Subsampling of training data can be used for speedup. """

  nbr_images = len(featurefiles)
  # read the features from file
  descr = []
  descr.append(sift.read_features_from_file(featurefiles[0])[1])
  descriptors = descr[0] #stack all features for k-means
  for i in arange(1,nbr_images):
    descr.append(sift.read_features_from_file(featurefiles[i])[1])
    descriptors = vstack((descriptors,descr[i]))

  # k-means: last number determines number of runs
  self.voc,distortion = kmeans(descriptors[::subsampling,:],k,1)
  self.nbr_words = self.voc.shape[0]

  # go through all training images and project on vocabulary
  imwords = zeros((nbr_images,self.nbr_words))
  for i in range( nbr_images ):
    imwords[i] = self.project(descr[i])

  nbr_occurences = sum( (imwords > 0)*1 ,axis=0)

  self.idf = log( (1.0*nbr_images) / (1.0*nbr_occurences+1) )
  self.trainingdata = featurefiles


def project(self,descriptors):
  """ Project descriptors on the vocabulary
    to create a histogram of words. """

  # histogram of image words
  imhist = zeros((self.nbr_words))
  words,distance = vq(descriptors,self.voc)
  for w in words:
    imhist[w] += 1
```

Output for crating Vocabulary

4.1.3 Indexing Images

To start indexing images we first need to set up a database. Indexing images in this context means extracting descriptors from the images, converting them to visual words using a vocabulary and storing the visual words and word histograms with information about which image they belong to. This will make it possible to query the database using an image and get the most similar images back as search result.

Here we will use SQLite as database. SQLite is a database which stores everything in a single file and is very easy to set up and use. We are using it here since it is the easiest way to get started without having to go into database and server configurations and other details way outside the scope of this book. SQLite uses the SQL query language so the transition should be easy if you want to use another database. To get started we need to create tables and indexes and an indexer class to write image data to the database. First, create a file imagesearch.py and add the following code:

imlist
rowid
filename

imwords
imid
wordid
vocname

imhistograms
imid
histogram
vocname

Table: A simple database schema for storing images and visual words.

```
import pickle
from pysqlite2 import dbapi2 as sqlite

class Indexer(object):

  def __init__(self,db,voc):
    """ Initialize with the name of the database
      and a vocabulary object. """

    self.con = sqlite.connect(db)
    self.voc = voc


  def __del__(self):
    self.con.close()


  def db_commit(self):
    self.con.commit()
```

First of all, we need pickle for encoding and decoding these arrays to and from strings. SQLite is imported from the pysqlite2 module (see appendix for installation details). The Indexer class connects to to a database and stores a vocabulary object upon creation (where the_init_() method is called). The_del_() method makes sure to close the database connection and db_commit() writes the changes to the database file. We only need a very simple database schema of three tables. The table imlist contains the filenames of all indexed images, imwords contains a word index of the words, which vocabulary was used, and which images the words appear in. Finally, imhistograms contains the full word histograms for each image. We need those to compare images according to our vector space model. The following method for the Indexer class creates the tables and some useful indexes to make searching faster.

Adding images

With the database tables in place, we can start adding images to the index. To do this, we need a method add_to_index() for our Indexer class. Add this method to imagesearch.py.

```
def create_tables(self):
  """ Create the database tables. """

  self.con.execute('create table imlist(filename)')
  self.con.execute('create table imwords(imid,wordid,vocname)')
  self.con.execute('create table imhistograms(imid,histogram,vocname)')
  self.con.execute('create index im_idx on imlist(filename)')
  self.con.execute('create index wordid_idx on imwords(wordid)')
  self.con.execute('create index imid_idx on imwords(imid)')
  self.con.execute('create index imidhist_idx on imhistograms(imid)')
  self.db_commit()
```

```
def add_to_index(self,imname,descr):
  """ Take an image with feature descriptors,
    project on vocabulary and add to database. """

  if self.is_indexed(imname): return
  print 'indexing', imname

  # get the imid
  imid = self.get_id(imname)

  # get the words
  imwords = self.voc.project(descr)
  nbr_words = imwords.shape[0]

  # link each word to image
  for i in range(nbr_words):
    word = imwords[i]
    # wordid is the word number itself
    self.con.execute("insert into imwords(imid,wordid,vocname)
          values (?,?,?)", (imid,word,self.voc.name))

  # store word histogram for image
  # use pickle to encode NumPy arrays as strings
  self.con.execute("insert into imhistograms(imid,histogram,vocname)
        values (?,?,?)", (imid,pickle.dumps(imwords),self.voc.name))
```

This method takes the image filename and a NumPy array with the descriptors found in the image. The descriptors are projected on the vocabulary and inserted in imwords (word by word) and imhistograms. We used two helper functions, is_indexed() which checks if the image has been indexed already, and get_id() which gives the image id for an image filename. Add these to imagesearch.py.

```
def is_indexed(self,imname):
  """ Returns True if imname has been indexed. """

  im = self.con.execute("select rowid from imlist where
        filename='%s'" % imname).fetchone()
  return im != None


def get_id(self,imname):
  """ Get an entry id and add if not present. """

  cur = self.con.execute(
  "select rowid from imlist where filename='%s'" % imname)
  res=cur.fetchone()
  if res==None:
    cur = self.con.execute(
    "insert into imlist(filename) values ('%s')" % imname)
    return cur.lastrowid
  else:
    return res[0]
```

4.1.4 Searching the Database for Images

With a set of images indexed we can search the database for similar images. Here we have used a bag-of-word representation for the whole image but the procedure explained here is generic and can be used to find similar objects, similar faces, similar colors etc. It all depends on the images and descriptors used. To handle searches we introduce a Searcher class to imagesearch.py

```
class Searcher(object):

  def __init__(self,db,voc):
    """ Initialize with the name of the database. """
    self.con = sqlite.connect(db)
    self.voc = voc


  def __del__(self):
    self.con.close()
```

A new Searcher object connects to the database and closes the connection upon deletion, same as for the Indexer class before.

If the number of images is large, it is not feasible to do a full histogram comparison across all images in the database. We need a way to find a reasonably sized set of candidates (where "reasonable" can be determined by search response time, memory requirements etc.). This is where the word index comes into play. Using the index we can get a set of candidates and then do the full comparison against that set.

Using the index to get candidates

We can use our index to find all images that contain a particular word. This is just a

```
def candidates_from_word(self,imword):
  """ Get list of images containing imword. """

  im_ids = self.con.execute(
      "select distinct imid from imwords where wordid=%d" % imword).fetchall()
  return [i[0] for i in im_ids]
```

This gives the image ids for all images containing the word. To get candidates for more than one word, for example all the nonzero entries in a word histogram, we can loop over each word, get images with that word and aggregate the lists. Here we should also keep track of how many times each image id appears in the aggregate list since this shows how many words that matches the ones in the word histogram. This can be done with the following Searcher method:

```
def candidates_from_histogram(self,imwords):
  """ Get list of images with similar words. """

  # get the word ids
  words = imwords.nonzero()[0]

  # find candidates
  candidates = []
  for word in words:
    c = self.candidates_from_word(word)
    candidates+=c

  # take all unique words and reverse sort on occurrence
  tmp = [(w,candidates.count(w)) for w in set(candidates)]
  tmp.sort(cmp=lambda x,y:cmp(x[1],y[1]))
  tmp.reverse()

  # return sorted list, best matches first
  return [w[0] for w in tmp]
```

This method creates a list of word ids from the nonzero entries in a word histogram of an image. Candidates for each word are retrieved and aggregated in the list candidates. Then we create a list of tuples (word id, count) with the number of occurrences of each word in the candidate list and sort this list (in place for efficiency) using sort() with a custom comparison function that compares the second element in the tuple. The comparison function is declared inline using lambda functions, convenient oneline function declarations. The result is returned as a list of image ids with the best matching image first.

```
src = imagesearch.Searcher('test.db')
locs,descr = sift.read_features_from_file(featlist[0])
iw = voc.project(descr)

print 'ask using a histogram...'
print src.candidates_from_histogram(iw)[:10]
```

Output

ask using a histogram... [655, 656, 654, 44, 9, 653, 42, 43, 41, 12]

None of the top 10 candidates are correct. Don't worry, we can now take any number of elements from this list and compare histograms. As you will see, this improves things considerably.

Querying with an image

There is not much more needed to do a full search using an image as query. To do word histogram comparisons a Searcher object needs to be able to read the image word histograms from the database. Add this method to the Searcher class. Again we use pickle to convert between string and NumPy arrays, this time with loads().

```
def query(self,imname):
    """ Find a list of matching images for imname"""

    h = self.get_imhistogram(imname)
    candidates = self.candidates_from_histogram(h)

    matchscores = []
    for imid in candidates:
        # get the name
        cand_name = self.con.execute(
            "select filename from imlist where rowid=%d" % imid).fetchone()
        cand_h = self.get_imhistogram(cand_name)
        cand_dist = sqrt( sum( (h-cand_h)**2 ) ) #use L2 distance
        matchscores.append( (cand_dist,imid) )

    # return a sorted list of distances and database ids
    matchscores.sort()
    return matchscores
```

Now we can combine everything into a query method:

This Searcher method takes the filename of an image, retrieves the word histogram and a list of candidates (which should be limited to some maximum number if you have a large data set). For each candidate, we compare histograms using standard Euclidean distance and return a sorted list of tuples containing distance and image id.

```
src = imagesearch.Searcher('test.db')
print 'try a query...'
print src.query(imlist[0])[:10]
```

Let's try a query for the same image as in the previous section:

This will again print the top 10 results, including the distance, and should look something like this:

```
try a query...
[(0.0, 1), (100.03999200319841, 2), (105.45141061171255, 3), (129.47200469599596, 708),
(129.73819792181484, 707), (132.68006632497588, 4), (139.89639023220005, 10),
(142.31654858097141, 706), (148.1924424523734, 716), (148.22955170950223, 663)]
```

Benchmarking and plotting the results

To get a feel for how good the search results are, we can compute the number of correct images on the top four positions. This is the measure used to report performance for the ukbench image set. Here's a function that computes this score. Add it to imagesearch.py and you can start optimizing your queries.

```
def compute_ukbench_score(src,imlist):
    """ Returns the average number of correct
        images on the top four results of queries."""

    nbr_images = len(imlist)
    pos = zeros((nbr_images,4))
    # get first four results for each image
    for i in range(nbr_images):
        pos[i] = [w[1]-1 for w in src.query(imlist[i])[:4]]

    # compute score and return average
    score = array([ (pos[i]//4)==(i//4) for i in range(nbr_images)])*1.0
    return sum(score) / (nbr_images)
```

This function gets the top four results and subtracts one from the index returned by query() since the database index starts at one and the list of images at zero. Then we compute the score using integer division, using the fact that the correct images are consecutive in groups of four. A perfect result gives a score of 4, nothing right gives a score of 0 and only retrieving the identical images gives a score of 1. Finding the identical image together with two of the three other images gives a score of 3.

4.1.5 Ranking Results using Geometry

Let's briefly look at a common way of improving results obtained using a bag of visual words model. One of the drawbacks if the model is that the visual words representation of an image does not contain the positions of the image features. This was the price paid to get speed and scalability.

One way to have the feature points improve results is to re-rank the top results using some criteria that takes the features geometric relationships into account. The most common approach is to fit homographies between the feature locations in the query image and the top result images.

To make this efficient the feature locations can be stored in the database and correspondences determined by the word id of the features (this only works if the vocabulary is large enough so that the word id matches contain mostly correct matches). This would require a major rewrite of our database and code above and complicate the presentation. To illustrate we will just reload the features for the top images and match them.

Here is what a complete example of loading all the model files and re-ranking the top results using homographies looks like.

```
import pickle
import sift
import imagesearch
import homography

# load image list and vocabulary
```

```
with open('ukbench_imlist.pkl','rb') as f:
  imlist = pickle.load(f)
  featlist = pickle.load(f)

nbr_images = len(imlist)

with open('vocabulary.pkl', 'rb') as f:
  voc = pickle.load(f)


src = imagesearch.Searcher('test.db',voc)

# index of query image and number of results to return
q_ind = 50
nbr_results = 20

# regular query
res_reg = [w[1] for w in src.query(imlist[q_ind])[:nbr_results]]
print 'top matches (regular):', res_reg


# load image features for query image
q_locs,q_descr = sift.read_features_from_file(featlist[q_ind])
fp = homography.make_homog(q_locs[:,:2].T)

# RANSAC model for homography fitting
model = homography.RansacModel()

rank = {}
# load image features for result
for ndx in res_reg[1:]:
  locs,descr = sift.read_features_from_file(featlist[ndx])

  # get matches
  matches = sift.match(q_descr,descr)
  ind = matches.nonzero()[0]
  ind2 = matches[ind]
  tp = homography.make_homog(locs[:,:2].T)

# sort dictionary to get the most inliers first
sorted_rank = sorted(rank.items(), key=lambda t: t[1], reverse=True)
res_geom = [res_reg[0]]+[s[0] for s in sorted_rank]
print 'top matches (homography):', res_geom


# plot the top results
imagesearch.plot_results(src,res_reg[:8])
imagesearch.plot_results(src,res_geom[:8])
 # compute homography, count inliers. if not enough matches return empty list
 try:
   H,inliers = homography.H_from_ransac(fp[:,ind],tp[:,ind2],model,match_theshold=4)
 except:
   inliers = []

 # store inlier count
 rank[ndx] = len(inliers)
```

Some example search results on the ukbench data set. The query image is shown on the far left followed by the top five retrieved images

Fig: Shows some sample results with the regular and re-ranked top images.

First the image list, feature list (containing the filenames of the images and SIFT feature files respectively) and the vocabulary is loaded. Then a Searcher object is created and a regular query is performed and stored in the list res_reg. The features for the query image are loaded. Then for each image in the result list, the features are loaded and matched against the query image. Homographies are computed from the matches and the number of inliers counted. If the homography fitting fails we set the inlier list to an empty list. Finally we sort the dictionary rank that contains image index and inlier count according to decreasing number of inliers. The result lists are printed to the console and the top images visualized.

The output looks like:

top matches (regular): [39, 22, 74, 82, 50, 37, 38, 17, 29, 68, 52, 91, 15, 90, 31, ...]

top matches (homography): [39, 38, 37, 45, 67, 68, 74, 82, 15, 17, 50, 52, 85, 22, 87, ...]

4.1.6 Building Demos and Web Applications

In this last section on searching we'll take a look at a simple way of building demos and web applications with Python. By making demos as web pages, you automatically get cross platform support and an easy way to show and share your project with minimal

requirements. In the sections below we will go through an example of a simple image search engine.

Creating web applications with CherryPy

To build these demos we will use the CherryPy package, CherryPy is a pure Python lightweight web server that uses an object oriented model. See the appendix for more details on how to install and configure CherryPy. Assuming that you have studied to have an initial idea of how CherryPy works, let's build an image search web demo:

Fig: Some example search results with re-ranking based on geometric consistency using homographies. For each example, the top row is the regular result and the bottom row the re-ranked result.

Image search demo

First we need to initialize with a few html tags and load the data using Pickle. We need the vocabulary for the Searcher object that interfaces with the database. Create a file searchdemo.py and add the following class with two methods.

```
import cherrypy, os, urllib, pickle
import imagesearch

class SearchDemo(object):

  def __init__(self):
    # load list of images
    with open('webimlist.txt') as f:
      self.imlist = f.readlines()

    self.nbr_images = len(self.imlist)
    self.ndx = range(self.nbr_images)

    # load vocabulary
    with open('vocabulary.pkl', 'rb') as f:
      self.voc = pickle.load(f)

    # set max number of results to show
    self.maxres = 15

    # header and footer html
    self.header = """
      <!doctype html>
      <head>
      <title>Image search example</title>
      </head>
      <body>
      """
    self.footer = """
      </body>
      </html>
      """

  def index(self,query=None):
    self.src = imagesearch.Searcher('web.db',self.voc)

    html = self.header
    html += """
      <br />
      Click an image to search. <a href='?query='>Random selection</a> of images.
      <br /><br />


     if query:
       # query the database and get top images
       res = self.src.query(query)[:self.maxres]
       for dist,ndx in res:
         imname = self.src.get_filename(ndx)
         html += "<a href='?query="+imname+"'>"
         html += "<img src='"+imname+"' width='100' />"
         html += "</a>"
     else:
       # show random selection if no query
       random.shuffle(self.ndx)
       for i in self.ndx[:self.maxres]:
         imname = self.imlist[i]
         html += "<a href='?query="+imname+"'>"
         html += "<img src='"+imname+"' width='100' />"
         html += "</a>"

     html += self.footer
     return html

   index.exposed = True

cherrypy.quickstart(SearchDemo(), '/',
          config=os.path.join(os.path.dirname(__file__), 'service.conf'))
```

As you can see, this simple demo consists of a single class with one method for initialization and one for the "index" page (the only page in this case). Methods are automatically mapped to URLs and arguments to the methods can be passed directly in the URL. The index method has a query parameter which in this case is the query image to sort the others agains. If it is empty, a random selection of images is shown instead. The line makes the index URL accessible and the last line starts the CherryPy web server with configurations read from service.conf. Our configuration file for this example has the following lines.

```
[global]
server.socket_host = "127.0.0.1"
server.socket_port = 8080
server.thread_pool = 50
tools.sessions.on = True

[/]
tools.staticdir.root = "tmp/"
tools.staticdir.on = True
tools.staticdir.dir = ""
```

The first part specifies which IP address and port to use. The second part enables a local folder for reading (in this case "tmp/"). This should be set to the folder containing your images.

4.2 Classifying Image Content

This chapter introduces algorithms for classifying images and image content. We look at some simple but effective methods as well as state of the art classifiers and apply them to two-class and multi-class problems. We show examples with applications in gesture recognition and object recognition.

4.2.1 K-Nearest Neighbors

Supervised Learning

It is the learning where the value or result that we want to predict is within the training data (labeled data) and the value which is in data that we want to study is known as Target or Dependent Variable or Response Variable. All the other columns in the dataset are known as the Feature or Predictor Variable or Independent Variable.

Supervised Learning is classified into two categories:

1. **Classification**: Here our target variable consists of the categories.
2. **Regression**: Here our target variable is continuous and we usually try to find out the line of the curve.

There are various ways to get labeled data:

1. Historical labeled Data
2. Experiment to get data: We can perform experiments to generate labeled data like A/B Testing.
3. Crowd-sourcing

k-nearest neighbor algorithm

This algorithm is used to solve the classification model problems. K-nearest neighbor or K- NN algorithm basically creates an imaginary boundary to classify the data. When new data points come in, the algorithm will try to predict that to the nearest of the boundary line.

Therefore, larger k value means smother curves of separation resulting in less complex models. Whereas, smaller k value tends to overfit the data and resulting in complex models. Using the k-nearest neighbor algorithm we fit the historical data (or train the model) and predict the future.

Example of the k-nearest neighbor algorithm

```
# Import necessary modules
from sklearn.neighbors import
KNeighborsClassifier from
sklearn.model_selection import train_test_split
from sklearn.datasets import load_iris
# Loading data
irisData =
load_iris()
# Create feature and target
arrays X = irisData.data
y = irisData.target
# Split into training and test set
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2,
random_state=42) knn = KNeighborsClassifier(n_neighbors=7)
knn.fit(X_train, y_train)
```

Predict on dataset which model has not seen

before print(knn.predict(X_test))

Steps are performed:

1. The k-nearest neighbor algorithm is imported from the scikit-learn package.
2. Create feature and target variables.
3. Split data into training and test data.
4. Generate a k-NN model using neighbors value.
5. Train or fit the data into the model.
6. Predict the future.

Model Accuracy

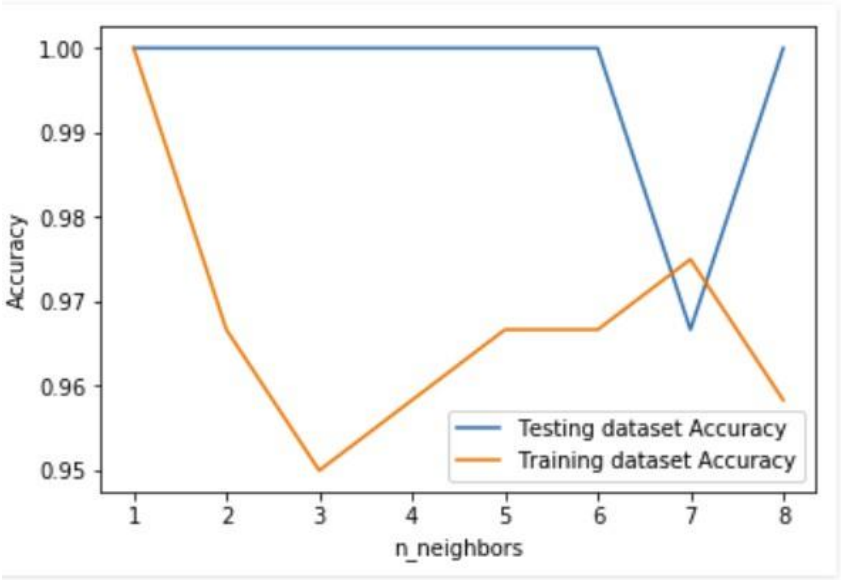
How to decide the right k-value for the dataset? We need to be familiar to data to get the range of expected k-value, but to get the exact k-value we need to test the model for each and every expected k-value.

Example

```
# Import necessary modules
from sklearn.neighbors import
KNeighborsClassifier from
sklearn.model_selection import train_test_split
from sklearn.datasets import load_iris
import numpy as np
import matplotlib.pyplot as
plt irisData = load_iris()
# Create feature and target
arrays X = irisData.data
y = irisData.target


# Split into training and test set
X_train, X_test, y_train, y_test = train_test_split( X, y, test_size = 0.2, random_state=42)


neighbors = np.arange(1, 9)
train_accuracy =
np.empty(len(neighbors))
test_accuracy =
np.empty(len(neighbors))


# Loop over K values
for i, k in enumerate(neighbors): knn =
KNeighborsClassifier(n_neighbors=k) knn.fit(X_train, y_train)


# Compute training and test data accuracy
train_accuracy[i] = knn.score(X_train,
y_train) test_accuracy[i] =
knn.score(X_test, y_test)


# Generate plot
```

```
plt.plot(neighbors, test_accuracy, label = 'Testing dataset
Accuracy') plt.plot(neighbors, train_accuracy, label = 'Training
dataset Accuracy')
plt.legend()
plt.xlabel('n_neighbos')
plt.ylabel('Accuracy') plt.show()
```

Output



4.2.2 Baye's

Bayes' theorem (alternatively Bayes' law or Bayes' rule) describes the probability of an event, based on prior knowledge of conditions that might be related to the event. For example, if a disease is related to age, then, using Bayes' theorem, a person's age can be used to more accurately assess the probability that they have the disease, compared to the assessment of the probability of disease made without knowledge of the person's age.

$$P(A|B) = \frac{P(B|A)\,P(A)}{P(B)}$$

Bayes' theorem is stated mathematically as the following equation:
where A and B are events and $P(B) \neq 0$.

- P(A|B) is a conditional probability: the likelihood of event A occurring given that B is true.
- P(B|A) is also a conditional probability: the likelihood of event B occurring given that A is true.
- P(A) and P(B) are the probabilities of observing A and B independently of each other; this is known as the marginal probability.
-

For example, suppose the probability of the weather being cloudy is 40%. Also suppose the probability of rain on a given day is 20%. Also suppose the probability of clouds on a rainy day is 85%.

1. If it's cloudy outside on a given day, what is the probability that it will rain that day?

Solution:

- P(cloudy) = 0.40
- P(rain) = 0.20

- P(cloudy | rain) = 0.85

  we can calculate:

- P(rain | cloudy) = P(rain) * P(cloudy | rain) / P(cloudy)
- P(rain | cloudy) = 0.20 * 0.85 / 0.40
- P(rain | cloudy) = 0.425

If it's cloudy outside on a given day, the probability that it will rain that day is **42.5%.**

2. A person has undertaken a job. The probabilities of completion of the job on time with and without rain are 0.44 and 0.95 respectively. If the probability that it will rain is 0.45, then determine the probability that the job will be completed on time.

Solution:

Let E_1 be the event that the mining job will be completed on time and E_2 be the event that it rains. We have,

P(A) = 0.45,

P(no rain) = P(B) = 1 − P(A) = 1 − 0.45 = 0.55

By multiplication law of probability,

$P(E_1) = 0.44$

$P(E_2) = 0.95$

Since, events A and B form partitions of the sample space S, by total probability theorem, we have $P(E) = P(A)\,P(E_1) + P(B)\,P(E_2)$

= 0.45 × 0.44 + 0.55 × 0.95

= 0.198 + 0.5225 = 0.7205

So, the probability that the job will be completed on time is 0.684.

3. There are three urns containing 3 white and 2 black balls; 2 white and 3 black balls; 1 black and 4 white balls respectively. There is an equal probability of each urn being chosen. One ball is equal probability chosen at random. what is the probability that a white ball is drawn?

Solution:

Let E_1, E_2, and E_3 be the events of choosing the first, second, and third urn respectively. Then, $P(E_1) = P(E_2) = P(E_3) = 1/3$

Let E be the event that a white ball is drawn.

Then, $P(E/E_1) = 3/5$, $P(E/E_2) = 2/5$, $P(E/E_3) = 4/5$

By theorem of total probability, we have

$P(E) = P(E/E_1) \cdot P(E_1) + P(E/E_2) \cdot P(E_2) + P(E/E_3) \cdot P(E_3)$

$= (3/5 \times 1/3) + (2/5 \times 1/3) + (4/5 \times 1/3)$

$= 9/15 = 3/5$

4.2.3 Support Vector Machines

An SVM model is a representation of the examples as points in space, mapped so that the examples of the separate categories are divided by a clear gap that is as wide as possible. In addition to performing linear classification, SVMs can efficiently perform a non-linear classification, implicitly mapping their inputs into high-dimensional feature spaces.

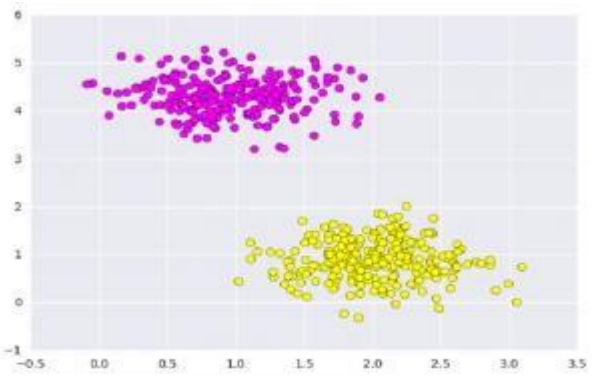
classification and regression analysis. A Support Vector Machine (SVM) is a discriminative classifier formally defined by a separating hyperplane. In other words, given labeled training data (supervised learning), the algorithm outputs an optimal hyperplane which categorizes new example.

Example

```
# importing scikit learn with
make_blobs from sklearn.datasets
import make_blobs
# creating datasets X containing
n_samples # Y containing two classes
X, Y = make_blobs(n_samples=500, centers=2, random_state=0,
cluster_std=0.40) import matplotlib.pyplot as plt
# plotting scatters
plt.scatter(X[:, 0], X[:, 1], c=Y, s=50,
cmap='spring'); plt.show()
```

Output

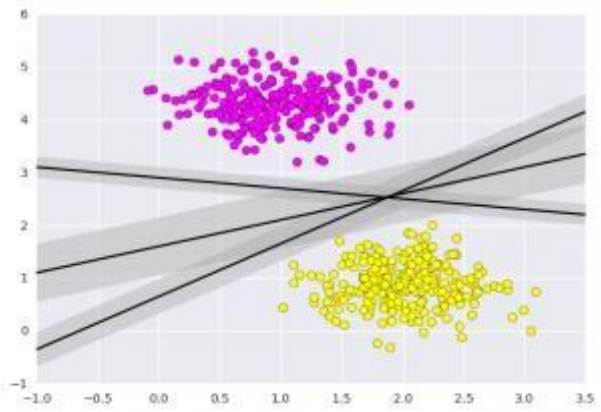
Support vector machines do, is to not only draw a line between two classes here, but consider a region about the line of some given width.



Example

```
# creating linspace between -1 to
3.5 xfit = np.linspace(-1, 3.5)

# plotting scatter
plt.scatter(X[:, 0], X[:, 1], c=Y, s=50, cmap='spring')

# plot a line between the different sets of data
for m, b, d in [(1, 0.65, 0.33), (0.5, 1.6, 0.55), (-0.2, 2.9, 0.2)]:
yfit = m * xfit + b
plt.plot(xfit, yfit, '-k')
plt.fill_between(xfit, yfit - d, yfit + d, edgecolor='none', color='#AAAAAA',
alpha=0.4) plt.xlim(-1, 3.5);
plt.show()
```

Output



Importing datasets

This is the intuition of support vector machines, which optimize a linear discriminant model representing the perpendicular distance between the datasets, we need to import cancer datasets as csv file where we will train two features out of all features.

Example

```
# importing required libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
# reading csv file and extracting class column
to y. x = pd.read_csv("C:\...\cancer.csv")
a = np.array(x)
y = a[:,30] # classes having 0
and 1 # extracting two features
x =
np.column_stack((x.malignant,x.benign))
# 569 samples and 2 features
x.shape
print (x),(y)
```

Output

```
[[  122.    1001.  ]
   8
```

```
[  132.9  1326. ]
[  130.   1203. ]
...,
[  108.3   858.1 ]
[  140.1  1265. ]
[   47.9    181.  ]]
   2

array([ 0., 0., 0., 0., 0., 0., 0., 0., 0.,  0.,    0.
                                                     ,

       0., 0., 0., 0., 0., 0., 1., 1., 1., 0., 0., 0.,

       0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1.,

       0., 0., 0., 0., 0., 0., 0., 1., 0., 1., 1., 1.,
```

Fitting a Support Vector Machine

Support Vector Machine Classifier to these points. While the mathematical details of the likelihood model are interesting, we'll let read about those elsewhere. Instead, we'll just treat the scikit-learn algorithm as a black box which accomplishes the above task.

Example

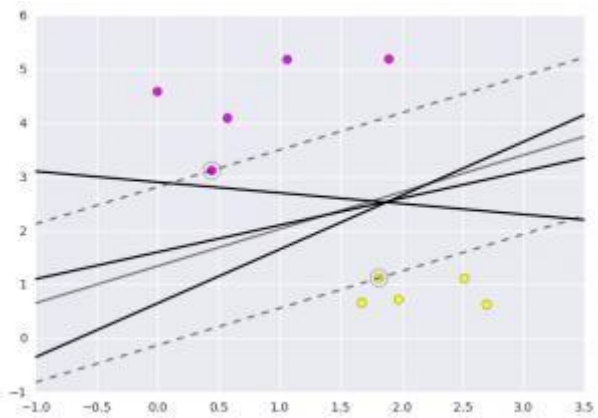
```
# import support vector
classifier # "Support Vector
Classifier" from sklearn.svm
import SVC clf =
SVC(kernel='linear')
# fitting x samples and y
classes clf.fit(x, y)
```

After being fitted, the model can then be used to predict new values:

```
clf.predict([[120, 990]])
clf.predict([[85, 550]])
```

Output

```
array([ 0.])
array([ 1.])
```



The graph how does this show.

4.2.4 Optical Character Recognition

Python is widely used for analyzing the data but the data need not be in the required format always. In such cases, we convert that format (like PDF or JPG, etc.) to the text format, in order to analyze the data in a better way. Python offers many libraries to do this task. There are several ways of doing this, including using libraries like PyPDF2 in Python.

The major disadvantage of using these libraries is the encoding scheme. PDF documents can come in a variety of encodings including UTF-8, ASCII, Unicode, etc. So, converting the PDF to text might result in the loss of data due to the encoding scheme. Let's see how to read all the contents of a PDF file and store it in a text document using OCR. Firstly, we need to convert the pages of the PDF to images and then, use OCR (Optical Character Recognition) to read the content from the image and store it in a text file.

There are two parts to the program as follows:

Part #1 deals with converting the PDF into image files. Each page of the PDF is stored as an image file. The names of the images stored are: PDF page 1 -> page_1.jpg PDF page 2 -> page_2.jpg PDF page 3 -> page_3.jpg …. PDF page n -> page_n.jpg.

Part #2 deals with recognizing text from the image files and storing it into a text file. Here, we process the images and convert it into text. Once we have the text as a string variable, we can do any processing on the text. For example, in many PDFs, when a line is completed, but a particular word cannot be written entirely in the same line, a hyphen ('-') is added, and the word is continued on the next line.

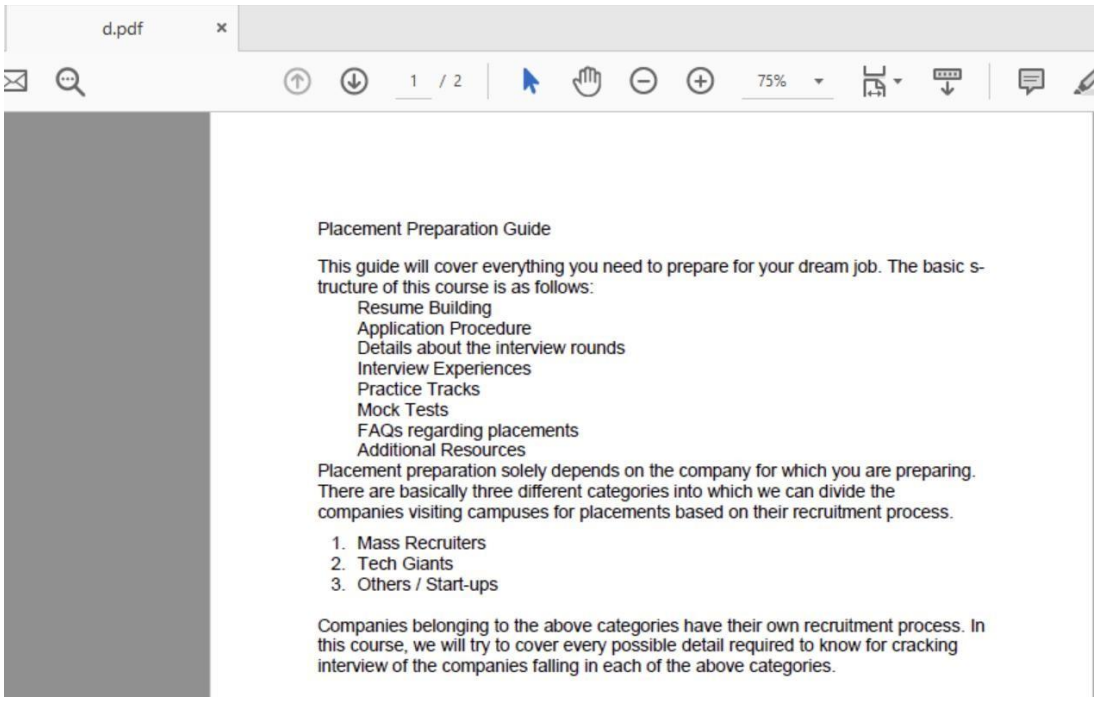
Example Program for PDF to Text file

```
# Requires Python 3.6 or higher due to f-strings
# Import
libraries import
platform
from tempfile import
TemporaryDirectory from pathlib
import Path
import pytesseract
from pdf2image import
convert_from_path from PIL import
Image
if platform.system() == "Windows":
# We may need to do some additional downloading and
setup... # Windows needs a PyTesseract Download
# https://github.com/UB-Mannheim/tesseract/wiki/Downloading-Tesseract-OCR-
Engine pytesseract.pytesseract.tesseract_cmd = ( r"C:\Program Files\Tesseract-
OCR\tesseract.exe")

# Windows also needs
poppler_exe path_to_poppler_exe
= Path(r"C:\..... ")

# Put our output files in a sane place...
out_directory =
Path(r"~\Desktop").expanduser() else:
out_directory = Path("~").expanduser()

# Path of the Input pdf
PDF_file =
Path(r"d.pdf")

# Store all the pages of the PDF in a
```

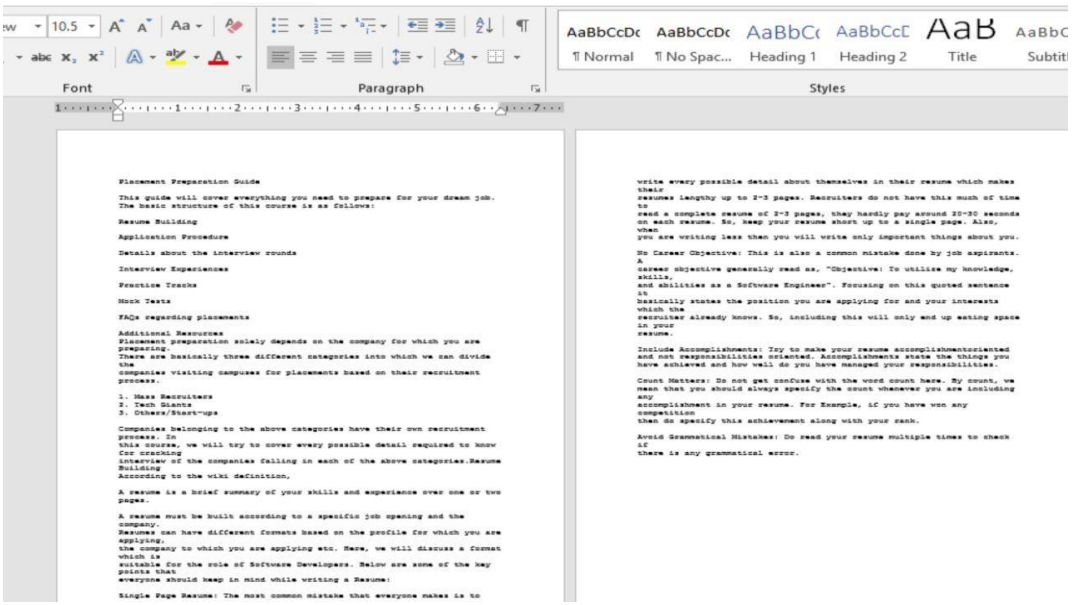
```
variable image_file_list = []
text_file = out_directory / Path("out_text.txt")
def main():
''' Main execution point of the
program''' with TemporaryDirectory()
as tmpdir:
text = str(((pytesseract.image_to_string(Image.open(image_file)))))
        # Create a temporary directory to hold our
temporary images. """Part #1 : Converting PDF to
images"""
if platform.system() == "Windows":pdf_pages = convert_from_path (
PDF_file, 500, poppler_path=path_to_poppler_exe  )
else:
pdf_pages = convert_from_path(PDF_file,
500) # Read in the PDF file at 500 DPI
# Iterate through all the pages stored above
for page_enumeration, page in enumerate(pdf_pages,
start=1): # enumerate() "counts" the pages for us.

# Create a file name to store the image
filename = f"{tempdir}\page_{page_enumeration:03}.jpg"

# Declaring filename for each page of PDF as
JPG # For each page, filename will be:
#     PDF     page     1     ->
page_001.jpg  #  PDF  page
2  ->  page_002.jpg  #  PDF
page  3  ->  page_003.jpg  #
....
# PDF page n -> page_00n.jpg

# Save the image of the page in system page.save(filename,
"JPEG") image_file_list.append(filename)
```

```
""" Part #2 - Recognizing text from the images using
OCR """ with open(text_file, "a") as output_file:
# Open the file in append mode so that
# All contents of all images are added to the same file
# Iterate from 1 to total number of pages for image_file in
image_file_list: # Set filename to recognize text from
# Again, these files will be:
#
page_1.jpg
#
page_2.jpg
# ....
# page_n.jpg

# The recognized text is stored in variable
text # Any string processing may be
applied on text # Here, basic formatting
has been done:
# In many PDFs, at line ending, if a word
can't # be written fully, a 'hyphen' is
added.
# The rest of the word is written in the next line
# Eg: This is a sample text this word here GeeksF- or Geeks is half on first line, remaining
on next. # To remove this, we replace every '-\n' to "text = text.replace("-\n", "")
# Finally, write the processed text to the
file. output_file.write(text)
# At the end of the with .. output_file
block # the file is closed after writing all
the text. # At the end of the with ..
tempdir block, the
# TemporaryDirectory() we're using gets
removed! # End of main function!
```

```
if__name__== "_main_":
# We only want to run this if it's directly
executed! main()
```



Output: Input PDF file



Advantages of this method include:

1. Avoiding text-based conversion because of the encoding scheme resulting in loss of data.
2. Even handwritten content in PDF can be recognized due to the usage of OCR.
3. Recognizing only particular pages of the PDF is also possible.
4. Getting the text as a variable so that any amount of required pre-processing can be done.

Disadvantages of this method include:

1. Disk storage is used to store the images in the local system. Although these images are tiny in size.
2. Using OCR cannot guarantee 100% accuracy. Given a computer-typed PDF document results in very high accuracy.
3. Handwritten PDFs are still recognized, but the accuracy depends on various factors like handwriting, page color, etc.

Unit Summary:

Searching Images and Classifying Image Content-This unit focuses on two main areas: Searching Images through Content-Based Image Retrieval (CBIR) and Classifying Image Content using machine learning techniques. Searching Images (Content-Based Image Retrieval - CBIR)-Content-Based Image Retrieval (CBIR) is a process that retrieves images based on their visual content rather than relying on text-based tags or metadata Classifying Image Content-Classifying images involves assigning labels or categories to images based on their content using machine learning algorithms.

Let us sum up

Content-Based Image Retrieval (CBIR) refers to a technique used to search, find, and retrieve images from large databases based on the visual content of the images, such as color, texture, shape, or any other visual features. Unlike traditional methods, which rely on metadata (keywords, tags, or descriptions), CBIR extracts information directly from the image data itself.
Feature Extraction:Images are analyzed to extract visual features like color, texture, shape, or patterns. These features represent the image as numerical data, often in the form of vectors.
Color Histograms: Represent the color distribution within an image.

Edge Detection: Captures shape information.

Texture Descriptors: Quantify surface properties or repeating patterns.

Visual Words and Bag of Visual Words (BoVW):

Inspired by text retrieval, images are represented by "visual words" based on extracted key points, similar to how text documents are represented by words.

The Bag of Visual Words (BoVW) model clusters key points (features) into visual words, forming a "visual dictionary" for image retrieval.

Image Indexing:
To enable fast retrieval, the extracted features are indexed. This can involve methods like hash tables, k-d trees, or other efficient data structures that allow the system to store and quickly

access the image feature vectors.

Query Processing:

When a user submits a query image, its visual features are extracted in the same way as the images in the database. The system compares these features with those in the database to find similar images.

Similarity Measurement:

Various techniques are used to compare the feature vectors of the query image with those in the database. Common methods include:

Euclidean distance: Measures the distance between feature vectors in a multi-dimensional space.

Cosine similarity: Measures the cosine of the angle between two feature vectors.

Ranking and Retrieval:

After comparing the query image with the images in the database, the system ranks the images based on their similarity scores, showing the most similar ones first.

Advanced systems may also use spatial verification (checking geometric consistency) to refine the ranking.

Check your progress

1. What does Content-Based Image Retrieval (CBIR) use to find images?

A) Metadata

B) Image file names

C) Visual content like color, texture, shape

D) URL of the image

Answer: C

Explanation: CBIR retrieves images based on their visual content, such as color, texture, and shape, instead of relying on metadata like file names or descriptions.

2. The concept of "Visual Words" in CBIR is similar to which model in text retrieval?

A) Bag of Visual Words

B) Bag of Words

C) Word2Vec

D) One-Hot Encoding

Answer: B

Explanation: "Visual Words" is inspired by the "Bag of Words" model in text retrieval, where images are represented as collections of visual features, just like documents are represented by word counts.

3.Which of the following methods is commonly used for indexing images in CBIR?

A) Hash Tables

B) Linked Lists

C) K-d Trees

D) All of the above

Answer: D

Explanation: Hash tables, k-d trees, and similar data structures are used to efficiently index and organize images based on their feature vectors to support fast retrieval.

4. What is the primary goal of ranking results using geometry in CBIR?

A) To improve the resolution of images

B) To verify spatial relationships between features in images

C) To enhance image color representation

D) To improve the metadata of images

Answer: B

Explanation: After retrieving visually similar images, spatial verification is done to ensure the geometric consistency of features, such as key points or objects, within the images.

5. In a CBIR system, which method is commonly used to compare feature vectors of images?

A) Mean Squared Error

B) Cosine Similarity

C) Euclidean Distance

D) Pearson Correlation

Answer: C

Explanation: Euclidean distance is widely used to measure the similarity between feature vectors, as it calculates the straight-line distance between two points in multi-dimensional

space.

6.Which algorithm classifies data based on the majority label of its nearest neighbors?

A) K-Means Clustering

B) Support Vector Machines

C) K-Nearest Neighbors (K-NN)

D) Decision Trees

Answer: C

Explanation: K-Nearest Neighbors (K-NN) is a simple classification algorithm that assigns the label of the majority class among the k nearest neighbors in the dataset.

7. The Bayes Classifier is based on which of the following principles?

A) Law of Large Numbers

B) Bayes' Theorem

C) Central Limit Theorem

D) Markov Chains

Answer: B

Explanation: The Bayes Classifier applies Bayes' Theorem, which calculates the probability of an event based on prior knowledge or evidence, often assuming conditional independence between features.

8. Which classifier finds the hyperplane that best separates classes in a feature space?

A) K-Nearest Neighbors

B) Support Vector Machines (SVM)

C) Naive Bayes Classifier

D) Decision Trees

Answer: B

Explanation: Support Vector Machines (SVM) work by finding the hyperplane that maximizes the margin between different classes, providing the best separation between them in the feature space.

9. Optical Character Recognition (OCR) is used to:

A) Convert text to speech

B) Convert handwritten or printed text in images into machine-readable text

C) Enhance the resolution of text in images

D) Identify objects in images

Answer: B

Explanation: OCR technology is used to detect and convert textual information in scanned documents or images into machine-readable text that can be edited and searched.

10.Which method is most effective for classifying images with a known boundary between classes?

A) K-Nearest Neighbors

B) Support Vector Machines (SVM)

C) Random Forest

D) Principal Component Analysis

Answer: B

Explanation: Support Vector Machines are highly effective when there is a clear, known boundary between classes, as the algorithm maximizes the margin between classes in the feature space.

Glossary

Content-Based Image Retrieval (CBIR): A technique to search and retrieve images from a database based on their content rather than metadata or keywords. Visual features like color, texture, and shape are used to index and find images.

Visual Words: A concept derived from the Bag of Words (BoW) model used in text retrieval, where images are represented as a collection of visual features. These features are often based on key points or regions of interest in the image.

Indexing Images: Creating an index or structured data that allows fast retrieval of images based on their visual features. It can involve hashing, tree-based structures, or quantization techniques.

Searching the Database for Images: Involves querying a database using extracted features from an image and finding similar images by comparing feature vectors.

Building Demos and Web Applications: Demonstrating CBIR systems via web apps, allowing users to upload or select an image and retrieve similar ones from a database.

K-Nearest Neighbors (K-NN): A simple classification algorithm that assigns a class to an input based on the majority class of its k nearest neighbors in feature space.

Bayes Classifier: A probabilistic classifier based on Bayes' theorem, often used when the features are conditionally independent. It's typically used in text classification but can also be applied to image data.

Support Vector Machines (SVM): A supervised learning algorithm used for classification tasks. It attempts to find the hyperplane that best separates different classes in the feature space.

Optical Character Recognition (OCR): A technique used to convert scanned images or photos of text into machine-readable text. Common in document processing applications.

Books

1. Computer Vision: Algorithms and Applications" by Richard Szeliski
ISBN: 978-1848000659 Link: Springer
2. Content-Based Image Retrieval: A Comprehensive Review" by M. A. M. Ali and F .D. D. F. W. A. Rehman N: 978-3319752904

Web Resources

1. "Introduction to Content-Based Image Retrieval" - Tutorialspoint
2. "Understanding Support Vector Machines (SVM)" - Towards Data Science
3. "K-Nearest Neighbors Algorithm" - GeeksforGeeks

UNIT – IV END

UNIT – V

5.1 Image Segmentation

The process of splitting images into multiple layers, represented by a smart, pixel-wise mask is known as Image Segmentation. It involves merging, blocking, and separating an image from its integration level. Splitting a picture into a collection of Image Objects with comparable properties is the first stage in image processing. Scikit-Image is the most popular tool/module for image processing in Python.

Installation

To install this module type the below command in the terminal.

pip install scikit-image

Converting Image Format RGB to Grayscale

rgb2gray module of skimage package is used to convert a 3-channel RGB Image to one channel monochrome image. In order to apply filters and other processing techniques, the expected input is a two-dimensional vector i.e. a monochrome image.

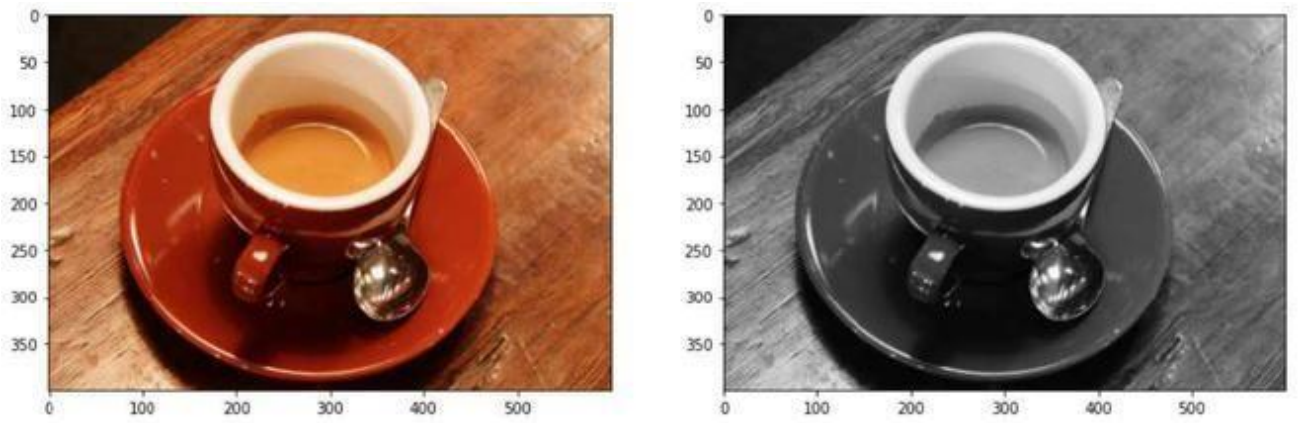
Syntax : skimage.color.rgb2gray(image)

Parameters : image : An image – RGB format **Return :** The image – Grayscale format **Example**

```
# Importing Necessary
Libraries from skimage
import data
from skimage.color import
rgb2gray import matplotlib.pyplot
as plt


# Setting the plot size to
15,15 plt.figure(figsize=(15,
15))
```

```
# Sample Image of scikit-image
package coffee = data.coffee()
plt.subplot(1, 2, 1)

# Displaying the sample
image plt.imshow(coffee)

# Converting RGB image to
Monochrome gray_coffee =
rgb2gray(coffee) plt.subplot(1, 2, 2)

# Displaying the sample image -
Monochrome # Format
plt.imshow(gray_coffee, cmap="gray")
```

Output



Converting 3-channel image data to 1-channel image data

Explanation: By using rgb2gray() function, the 3-channel RGB image of shape (400, 600, 3) is converted to a single-channel monochromatic image of shape (400, 300). We will be using grayscale images for the proper implementation of thresholding functions. The average of the red, green, and blue pixel values for each pixel to get the grayscale value is a simple approach to convert a color picture 3D array to a grayscale 2D array. This creates an acceptable gray approximation by combining the lightness or brightness contributions of each color band.

RGB to HSV

The HSV (Hue, Saturation, Value) color model remaps the RGB basic colors into dimensions that are simpler to comprehend for humans. The RGB color space describes the proportions of red, green, and blue in a colour. In the HSV color system, colors are defined in terms of Hue, Saturation, and Value.

Syntax : skimage.color.rgb2hsv(image)

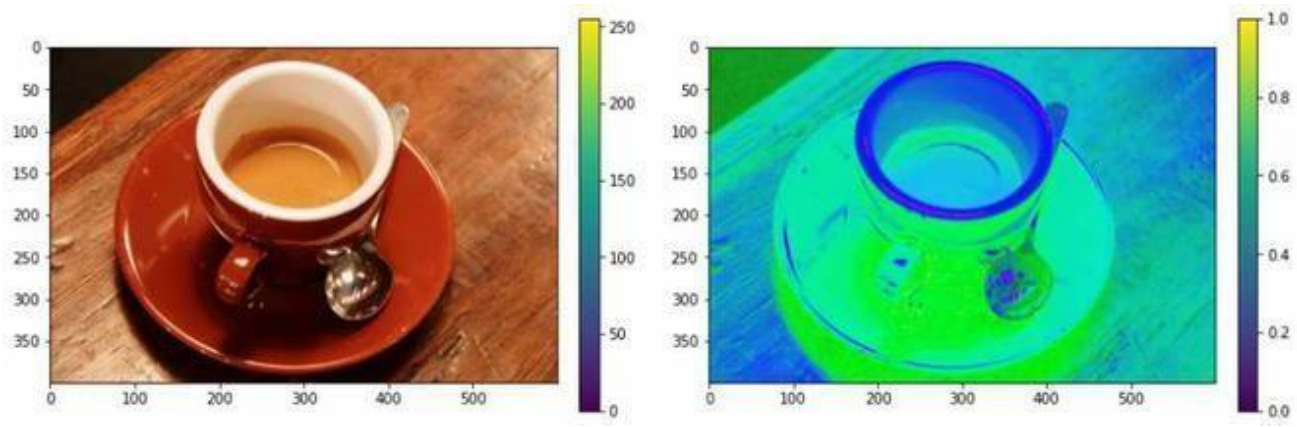
Parameters : image : An image – RGB format **Return :** The image – HSV format

Example

```
# Importing Necessary
Libraries from skimage
import data
from skimage.color import
rgb2hsv import matplotlib.pyplot
as plt


# Setting the plot size to
15,15 plt.figure(figsize=(15,
15))


# Sample Image of scikit-image
package coffee = data.coffee()
plt.subplot(1, 2, 1)


# Displaying the sample
image plt.imshow(coffee)
```

```
# Converting RGB Image to HSV Image
hsv_coffee =
rgb2hsv(coffee)
plt.subplot(1, 2, 2)

# Displaying the sample image - HSV
Format hsv_coffee_colorbar =
plt.imshow(hsv_coffee)

# Adjusting colorbar to fit the size of the image
plt.colorbar(hsv_coffee_colorbar, fraction=0.046,
pad=0.04)
```

Output



Converting the RGB color format to HSV color format

Supervised Segmentation

For this type of segmentation to proceed, it requires external input. This includes things like setting a threshold, converting formats, and correcting external biases.

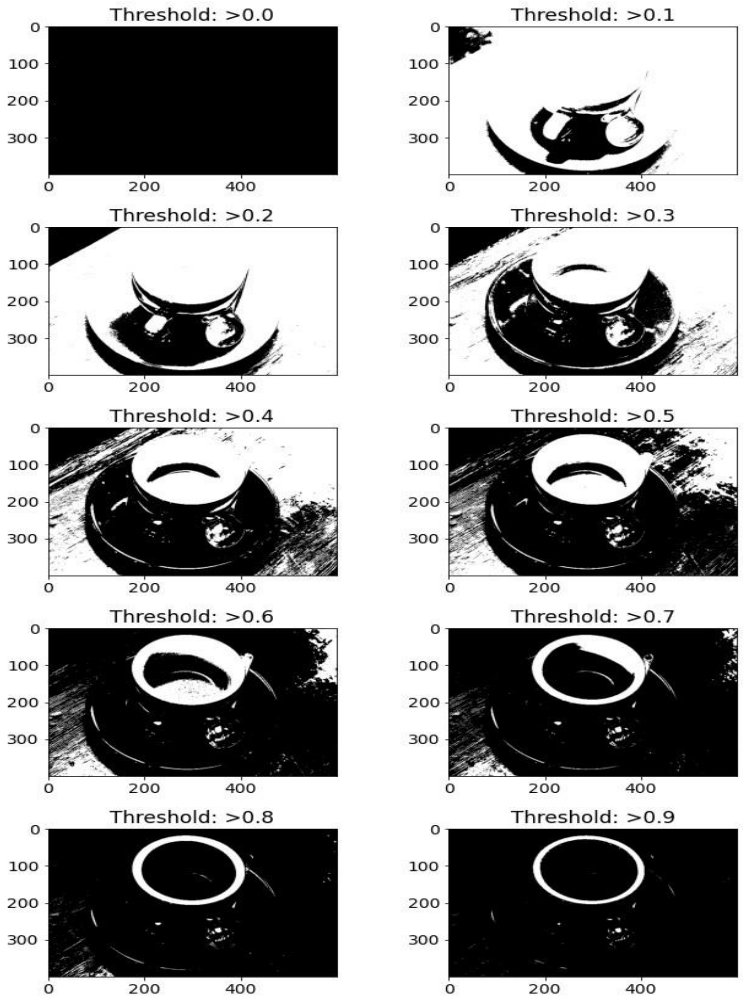
Segmentation by Thresholding – Manual Input

An external pixel value ranging from 0 to 255 is used to separate the picture from the background. This results in a modified picture that is larger or less than the specified threshold.

Example

```
# Importing Necessary Libraries
# Displaying the sample image - Monochrome
Format from skimage import data
from skimage import filters
from skimage.color import
rgb2gray import matplotlib.pyplot
as plt

# Sample Image of scikit-image
package coffee = data.coffee()
gray_coffee = rgb2gray(coffee)

# Setting the plot size to
15,15 plt.figure(figsize=(15,
15))
for i in range(10):
# Iterating different thresholds
```

```
binarized_gray = (gray_coffee >
i*0.1)*1 plt.subplot(5,2,i+1)

# Rounding of the
threshold # value to 1
decimal point
plt.title("Threshold: >"+str(round(i*0.1,1)))

# Displaying the binarized
image # of various thresholds
plt.imshow(binarized_gray, cmap =
'gray') plt.tight_layout()
```



Output

Explanation: The first step in this thresholding is implemented by normalizing an image

from 0 – 255 to 0 – 1. A threshold value is fixed and on the comparison, if evaluated to be true, then we store the result as 1, otherwise 0. This globally binarized image can be used to detect edges as well as analyze contrast and color difference.

Active Contour Segmentation

The concept of energy functional reduction underpins the active contour method. An active contour is a segmentation approach that uses energy forces and restrictions to separate the pixels of interest from the remainder of the picture for further processing and analysis. The term “active contour” refers to a model in the segmentation process.

Syntax : skimage.segmentation.active_contour(image, snake)

Parameters :

- **image :** An image
- **snake :** Initial snake coordinates – for bounding the feature
- **alpha :** Snake length shape
- **beta :** Snake smoothness shape
- **w_line :** Controls attraction – Brightness
- **w_edge :** Controls attraction – Edges
- **gamma :** Explicit time step

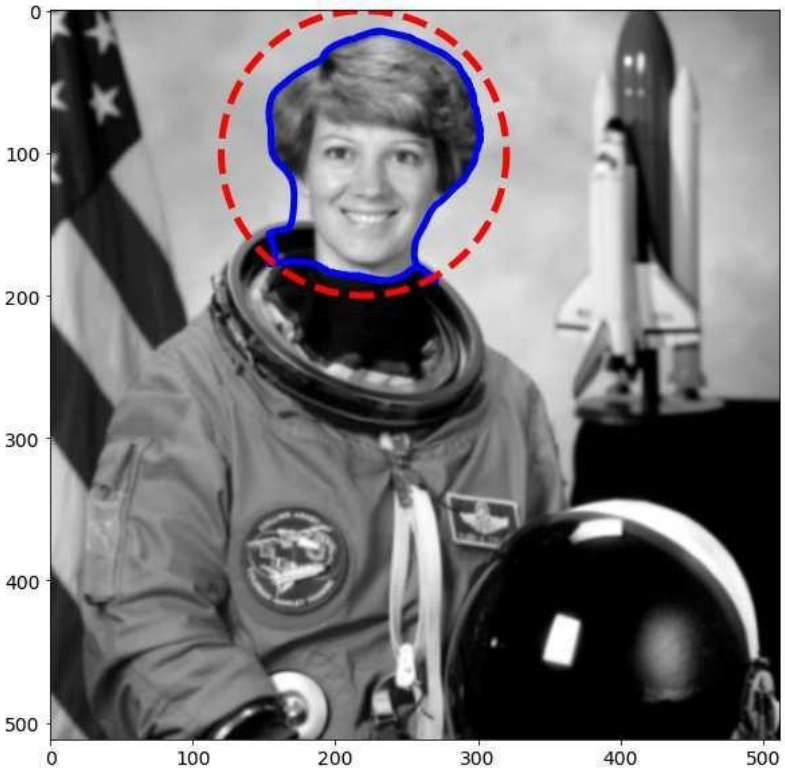
Return : snake : Optimised snake with input parameter’s size

Example

```
# Importing necessary
libraries import numpy as
np
import matplotlib.pyplot as plt
from skimage.color import
rgb2gray from skimage import
data
from skimage.filters import gaussian
from skimage.segmentation import active_contour


# Sample Image of scikit-image
package astronaut =
data.astronaut() gray_astronaut =
rgb2gray(astronaut)

# Applying Gaussian Filter to remove noise
gray_astronaut_noiseless =
gaussian(gray_astronaut, 1)

# Localising the circle's center at 220, 110
x1 = 220 + 100*np.cos(np.linspace(0, 2*np.pi,
500)) x2 = 100 + 100*np.sin(np.linspace(0,
2*np.pi, 500))

# Generating a circle based on
x1, x2 snake = np.array([x1,
x2]).T

# Computing the Active Contour for the given image
astronaut_snake =
```

```
active_contour(gray_astronaut_noiseless,snake) fig =
plt.figure(figsize=(10, 10))

# Adding subplots to display the
markers ax = fig.add_subplot(111)

# Plotting sample image
ax.imshow(gray_astronaut_noisel
ess) # Plotting the face boundary
marker
ax.plot(astronaut_snake[:, 0], astronaut_snake[:, 1], '-b', lw=5)

# Plotting the circle around face
ax.plot(snake[:, 0], snake[:, 1], '--r', lw=5)
```

Output



Explanation: The active contour model is among the dynamic approaches in image segmentation that uses the image's energy restrictions and pressures to separate regions of interest. For segmentation, an active contour establishes a different border or curvature for each section of the target object. The active contour model is a technique for minimizing the energy function resulting from external and internal forces. An exterior force is specified as curves or surfaces, while an interior force is defined as picture data. The external force is a force that allows initial outlines to automatically transform into the forms of objects in pictures.

Unsupervised Segmentation

Mark Boundaries

This technique produces an image with highlighted borders between labeled areas, where the pictures were segmented using the SLIC method.

skimage.segmentation.mark_boundaries() function is to return image with boundaries between labeled regions.

Parameters:

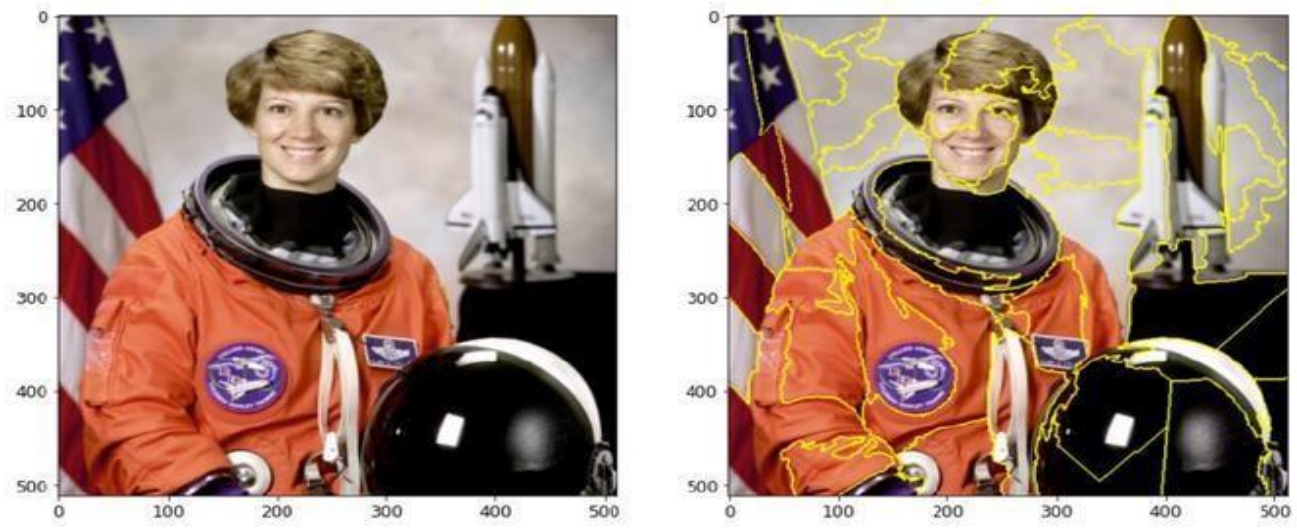
- **image :** An image
- **label_img :** Label array with marked regions
- **color :** RGB color of boundaries
- **outline_color :** RGB color of surrounding

boundaries **Return: marked:** An image with

boundaries are marked **Example**

```
# Importing required boundaries
from skimage.segmentation import slic,
mark_boundaries from skimage.data import
astronaut

# Setting the plot figure as 15,
15 plt.figure(figsize=(15, 15))

# Sample Image of scikit-image
package astronaut = astronaut()
```

```
# Applying SLIC
segmentation # for the edges
to be drawn over
astronaut_segments = slic(astronaut, n_segments=100,
compactness=1) plt.subplot(1, 2, 1)

# Plotting the original
image
plt.imshow(astronaut)

# Detecting boundaries for
labels plt.subplot(1, 2, 2)

# Plotting the outpu t of marked_boundaries

# function i.e. the image with segmented boundaries
plt.imshow(mark_boundaries(astronaut,
astronaut_segments))
```



Explanation: We cluster the image into 100 segments with compactness = 1 and this segmented image will act as a labeled array for the mark_boundaries() function. Each segment of the clustered image is differentiated by an integer value and the result of mark_boundaries is the superimposed boundaries between the labels.

Simple Linear Iterative Clustering

By combining pixels in the image plane based on their color similarity and proximity, this method generates superpixels. Simple Linear Iterative Clustering is the most up-to-date approach for segmenting superpixels, and it takes very little computing power. In a nutshell, the technique clusters pixels in a five-dimensional color and picture plane space to create small, nearly uniform super pixels.

skimage.segmentation.slic() function is used to segment image using k-means clustering.

Syntax : skimage.segmentation.slic(image)

Parameters:

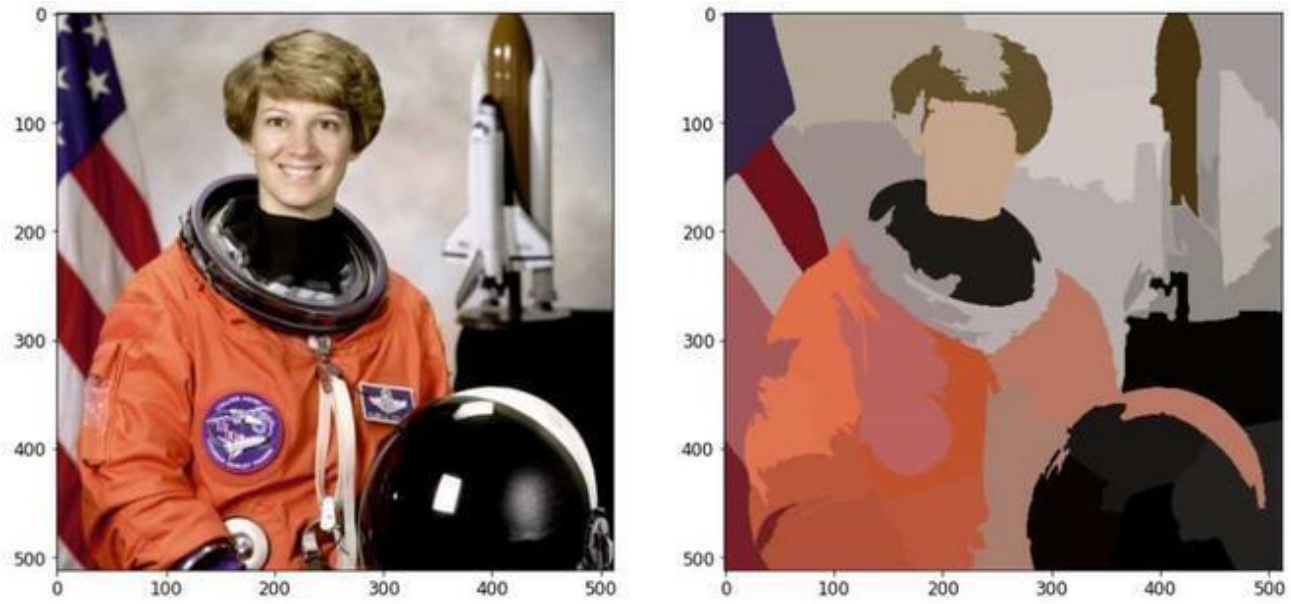
- **image :** An image
- **n_segments :** Number of labels
- **compactness :** Balances color and space proximity.
- **max_num_iter :** Maximum number of iterations

Return: labels: Integer mask indicating segment labels.

Example

```
# Importing required libraries
from skimage.segmentation import
slic from skimage.data import
astronaut from skimage.color
import label2rgb

# Setting the plot size as 15,
15 plt.figure(figsize=(15,15))

# Sample Image of scikit-image
package astronaut = astronaut()
```

```
# Applying Simple Linear
Iterative # Clustering on the
image
# - 50 segments & compactness = 10
astronaut_segments = slic(astronaut, n_segments=50,
compactness=10) plt.subplot(1,2,1)

# Plotting the original image
plt.imshow(astronaut)
plt.subplot(1,2,2)

# Converts a label image into
# an RGB color image for visualizing
# the labeled regions.
plt.imshow(label2rgb(astronaut_segments, astronaut, kind = 'avg'))
```



Explanation: This technique creates superpixels by grouping pixels in the picture plane based on their color similarity and closeness. This is done in 5-D space, where XY is the pixel location. Because the greatest possible distance between two colors in CIELAB space is restricted, but the spatial distance on the XY plane is dependent on the picture size, we must normalize the spatial distances in order to apply the Euclidean distance in this 5D

space. As a result, a new distance measure that takes superpixel size into account was created to cluster pixels in this 5D space.

There are many other supervised and unsupervised image segmentation techniques. This can be useful in confining individual features, foreground isolation, noise reduction, and can be useful to analyze an image more intuitively. It is a good practice for images to be segmented before building a neural network model in order to yield effective results.

5.1.1 Graph Cut

A graph cut is the partitioning of a directed graph into two disjoint sets. Graph cuts can be used for solving many different computer vision problems like stereo depth reconstruction, image stitching and image segmentation. By creating a graph from image pixels and their neighbors and introducing an energy or a "cost" it is possible to use a graph cut process to segment an image in two or more regions. The basic idea is that similar pixels that are also close to each other should belong to the same partition.

Here's how to build the graph:

- Every pixel node has an incoming edge from the source node.
- Every pixel node has an outgoing edge to the sink node.
- Every pixel node has one incoming and one outgoing edge to each of its neighbors.

To determine the weights on these edges, you need a segmentation model that determines the edge weights (representing the maximum flow allowed for that edge) between pixels and between pixels and the source and sink. As before we call the edge weight between pixel i and pixel j, wij . Let's call the weight from the source to pixel i, wsi, and from pixel i to the sink, wit.

Let's look at using a naive Bayesian classifier from Section 8.2 on the color values of the pixels. Given that we have trained a Bayes classifier on foreground and background pixels (from the same image or from other images), we can compute the probabilities pF (Ii) and pB(Ii) for the foreground and background. Here Ii is the color vector of pixel i.

We can now create a model for the edge weights as follows:

$$w_{si} = \frac{p_F(I_i)}{p_F(I_i) + p_B(I_i)}$$

$$w_{it} = \frac{p_B(I_i)}{p_F(I_i) + p_B(I_i)}$$

$$w_{ij} = \kappa\, e^{-|I_i - I_j|^2/\sigma} \quad .$$

With this model, each pixel is connected to the foreground and background (source and sink) with weights equal to a normalized probability of belonging to that class. The wij describe the pixel similarity between neighbors, similar pixels have weight close to, dissimilar close to 0. The parameter determines how fast the values decay towards zero with increasing dissimilarity.

Create a file graphcut.py and add the following function that creates this graph from an image.

```
from pygraph.classes.digraph import digraph
from pygraph.algorithms.minmax import maximum_flow

import bayes

def build_bayes_graph(im,labels,sigma=1e2,kappa=2):
  """ Build a graph from 4-neighborhood of pixels.
    Foreground and background is determined from
    labels (1 for foreground, -1 for background, 0 otherwise)
    and is modeled with naive Bayes classifiers."""

  m,n = im.shape[:2]
```

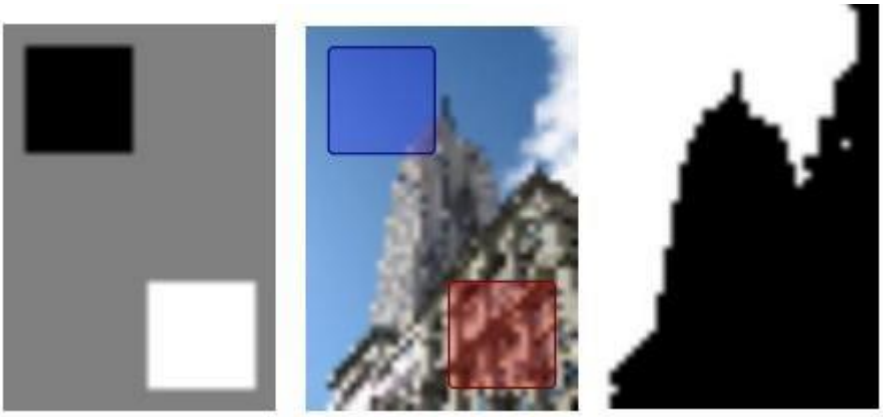
```
# RGB for foreground and background
foreground = im[labels==1].reshape((-1,3))
background = im[labels==-1].reshape((-1,3))
train_data = [foreground,background]

# train naive Bayes classifier
bc = bayes.BayesClassifier()
bc.train(train_data)

# get probabilities for all pixels
bc_lables,prob = bc.classify(vim)
prob_fg = prob[0]
prob_bg = prob[1]

# create graph with m*n+2 nodes
gr = digraph()
gr.add_nodes(range(m*n+2))

source = m*n # second to last is source
sink = m*n+1 # last node is sink

# normalize
for i in range(vim.shape[0]):
  vim[i] = vim[i] / linalg.norm(vim[i])

# go through all nodes and add edges
for i in range(m*n):
  # add edge from source
  gr.add_edge((source,i), wt=(prob_fg[i]/(prob_fg[i]+prob_bg[i])))

  # add edge to sink
  gr.add_edge((i,sink), wt=(prob_bg[i]/(prob_fg[i]+prob_bg[i])))

  # add edges to neighbors
  if i%n != 0: # left exists
    edge_wt = kappa*exp(-1.0*sum((vim[i]-vim[i-1])**2)/sigma)
    gr.add_edge((i,i-1), wt=edge_wt)
  if (i+1)%n != 0: # right exists
    edge_wt = kappa*exp(-1.0*sum((vim[i]-vim[i+1])**2)/sigma)
    gr.add_edge((i,i+1), wt=edge_wt)
  if i//n != 0: # up exists
    edge_wt = kappa*exp(-1.0*sum((vim[i]-vim[i-n])**2)/sigma)
  gr.add_edge((i,i-n), wt=edge_wt)
if i//n != m-1: # down exists
  edge_wt = kappa*exp(-1.0*sum((vim[i]-vim[i+n])**2)/sigma)

    gr.add_edge((i,i+n), wt=edge_wt)

return gr
```

Here we used a label image with values 1 for foreground training data and -1 for background training data. Based on this labeling, a Bayes classifier is trained on the RGB values. Then classification probabilities are computed for each pixel. These are then used as edge weights for the edges going from the source and to the sink. A graph with n ⇤ m + 2 nodes is created. Note the index of the source and sink, we choose them as the last two to simplify the indexing of the pixels. To visualize the labeling overlaid on the image we can use the function contourf() which fills the regions between contour levels of an image (in this case the label image). The alpha variable sets the transparency. Add the following function to graphcut.py.

```
def show_labeling(im,labels):
  """ Show image with foreground and background areas.
    labels = 1 for foreground, -1 for background, 0 otherwise."""

  imshow(im)
  contour(labels,[-0.5,0.5])
  contourf(labels,[-1,-0.5],colors='b',alpha=0.25)
  contourf(labels,[0.5,1],colors='r',alpha=0.25)
  axis('off')
```

Once the graph is built it needs to be cut at the optimal location. The following function computes the min cut and reformats the output to a binary image of pixel labels.

```
def cut_graph(gr,imsize):
  """ Solve max flow of graph gr and return binary
    labels of the resulting segmentation."""

  m,n = imsize
  source = m*n # second to last is source
  sink = m*n+1 # last is sink

  # cut the graph
  flows,cuts = maximum_flow(gr,source,sink)

  # convert graph to image with labels
  res = zeros(m*n)
  for pos,label in cuts.items()[:-2]: #don't add source/sink
    res[pos] = label

  return res.reshape((m,n))
```



Output

An example of graph cut segmentation using a Bayesian probability model. Image is down sampled to size 54*38. (left) label image for model training. (center) training regions shown on the image. (right) segmentation.

Segmentation with user input

Graph cut segmentation can be combined with user input in a number of ways. For example, a user can supply markers for foreground and background by drawing on an image. Another way is to select a region that contains the foreground with a bounding box

or using a "lasso" tool.

(a) (b) (c) (d)

These images come with ground truth labels for measuring segmentation performance. They also come with annotations simulating a user selecting a rectangular image region or drawing on the image with a "lasso" type tool to mark foreground and background. We can use these user inputs to get training data and apply graph cuts to segment the image guided by the user input.

The user input is encoded in bitmap images with the following meaning.

Figure: The effect of changing the relative weighting between pixel similarity and class probability. The same segmentation as in Figure 9.2 with: (a) k = 1, (b) k = 2, (c) k = 5 and (d) k = 10.

pixel value	meaning
0, 64	background
128	unknown
255	foreground

Example of loading an image and annotations and passing that to our graph cut segmentation routine.

```
from scipy.misc import imresize
import graphcut

def create_msr_labels(m,lasso=False):
  """ Create label matrix for training from
    user annotations. """

  labels = zeros(im.shape[:2])

  # background
  labels[m==0] = -1
  labels[m==64] = -1

  # foreground
  if lasso:
    labels[m==255] = 1
  else:
    labels[m==128] = 1

  return labels
```

```
# load image and annotation map
im = array(Image.open('376043.jpg'))
m = array(Image.open('376043.bmp'))

# resize
scale = 0.1
im = imresize(im,scale,interp='bilinear')
m = imresize(m,scale,interp='nearest')

# create training labels
labels = create_msr_labels(m,False)

# build graph using annotations
g = graphcut.build_bayes_graph(im,labels,kappa=2)

# cut graph
res = graphcut.cut_graph(g,im.shape[:2])

# remove parts in background
res[m==0] = 1
res[m==64] = 1


# plot the result
figure()
imshow(res)
gray()
xticks([])
yticks([])
savefig('labelplot.pdf')
```

First we define a helper function to read the annotation images and format them so we can pass them to our function for training background and foreground models. The bounding rectangles contain only background labels. In this case we set the foreground training region to the whole "unknown" region (the inside of the rectangle). Next we build the graph and cut it. Since we have user input we remove results that have any foreground in the marked background area. Last, we plot the resulting segmentation and remove the tick markers by setting them to an empty list. That way we get a nice bounding box (otherwise the boundaries of the image will be hard to see in this black and white plot).

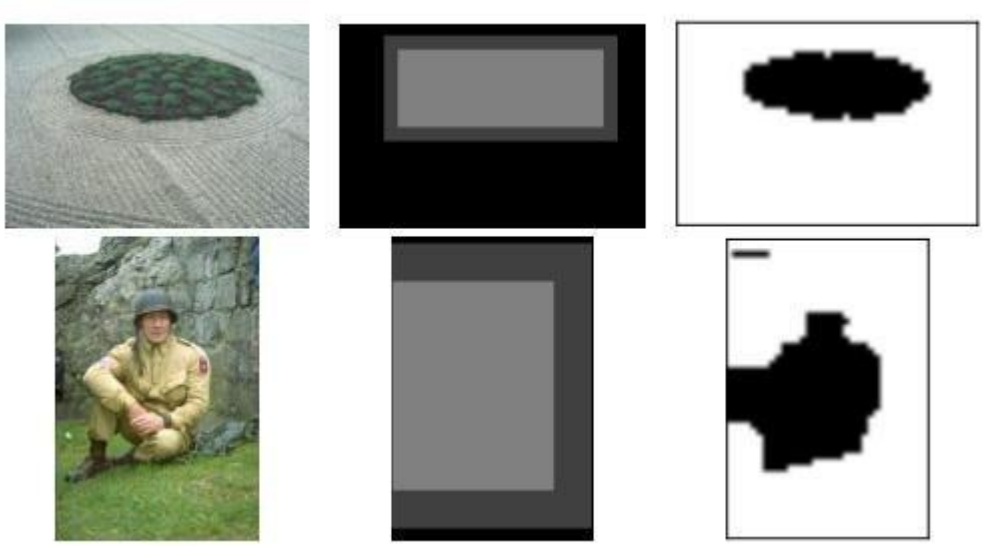


Figure: Sample graph cut segmentation results using images from the Grab Cut data set. (left) original image, downsampled. (middle) mask used for training. (right) resulting segmentation using RGB values as feature vectors.

5.1.2 Segmentation using Clustering

It is a method to perform Image Segmentation of pixel-wise segmentation. In this type of segmentation, we try to cluster the pixels that are together. There are two approaches for performing the Segmentation by clustering.

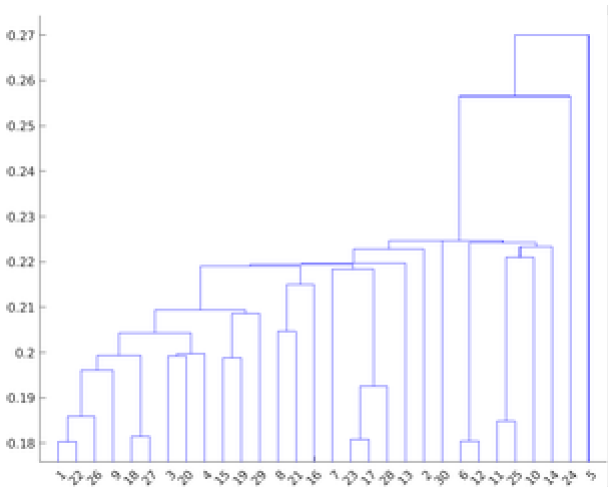
- Clustering by Merging
- Clustering by Divisive

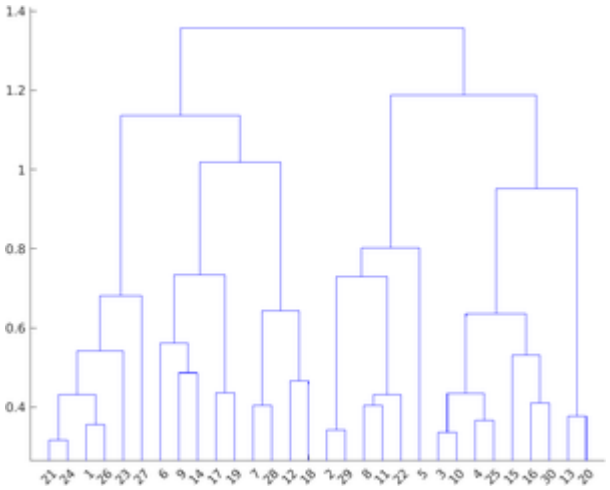
Clustering by merging or Agglomerative Clustering

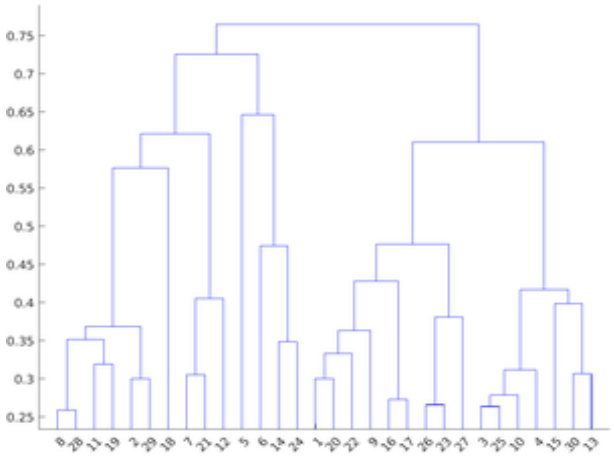
In this approach, we follow the bottom-up approach, which means we assign the pixel closest to the cluster. The algorithm for performing the agglomerative clustering as follows:

- Take each point as a separate cluster.
- For a given number of epochs or until clustering is satisfactory.
- Merge two clusters with the smallest inter-cluster distance (WCSS).
- Repeat the above step

The agglomerative clustering is represented by Dendrogram. It can be performed in 3 methods: by selecting the closest pair for merging, by selecting the farthest pair for merging, or by selecting the pair which is at an average distance (neither closest nor farthest).







Nearest clustering Average Clustering

Farthest Clustering

• • •

Clustering by division or Divisive splitting

In this approach, we follow the top-down approach, which means we assign the pixel closest to the cluster. The algorithm for performing the agglomerative clustering as follows:

- Construct a single cluster containing all points.
- For a given number of epochs or until clustering is satisfactory.
- Split the cluster into two clusters with the largest inter-cluster distance.
- Repeat the above steps.

K-Means Clustering

K-means clustering is a very popular clustering algorithm which applied when we have a dataset with labels unknown. The goal is to find certain groups based on some kind of similarity in the data with the number of groups represented by K. This algorithm is

generally used in areas like market segmentation, customer segmentation, etc.

The algorithm for image segmentation works as follows:

1. First, we need to select the value of K in K-means clustering.
2. Select a feature vector for every pixel (color values such as RGB value, texture etc.).
3. Define a similarity measure b/w feature vectors such as Euclidean distance to measure the similarity b/w any two points/pixel.
4. Apply K-means algorithm to the cluster centers
5. Apply connected component's algorithm.
6. Combine any component of size less than the threshold to an adjacent component that is similar to it until you can't combine more.

The steps for applying the K-means clustering algorithm:

- Select K points and assign them one cluster center each.
- Until the cluster center won't change, perform the following steps:
- Allocate each point to the nearest cluster center and ensure that each cluster center has one point.
- Replace the cluster center with the mean of the points assigned to it.
- End

Example # imports

```
import numpy as
np import cv2 as
cv
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] =
(12,50)

# load image
img =
cv.imread('road.jpg') Z =
img.reshape((-1,3))
# convert to
```

```
np.float32 Z =
np.float32(Z)

# define stopping criteria, number of clusters(K) and apply
kmeans() # TERM_CRITERIA_EPS : stop when the epsilon
value is reached
# TERM_CRITERIA_MAX_ITER: stop when Max iteration is reached
criteria = (cv.TERM_CRITERIA_EPS + cv.TERM_CRITERIA_MAX_ITER, 10, 1.0)

fig, ax = plt.subplots(10,2,
sharey=True) for i in range(10):
K = i+3
# apply K-means algorithm
ret,label,center=cv.kmeans(Z,K,None,criteria,attempts = 10,
cv.KMEANS_RANDOM_CENTERS) # Now convert back into uint8, and make original
image
center =
np.uint8(center) res =
center[label.flatten()]
res2 = res.reshape((img.shape))
# plot the original image and K-means
image ax[i, 1].imshow(res2)
ax[i,1].set_title('K = %s Image'%K)
ax[i, 0].imshow(img)
ax[i,0].set_title('Original
Image')
```

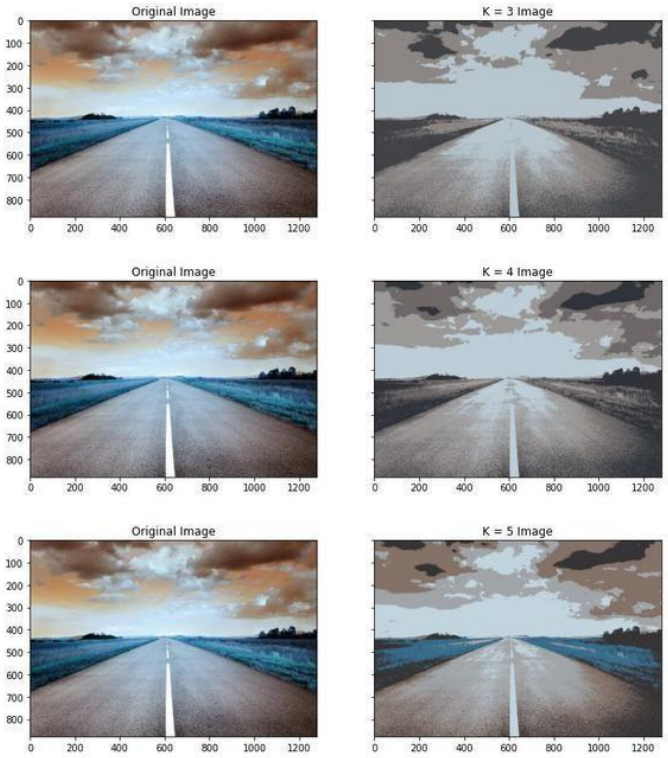


Image Segmentation for K=3,4,5

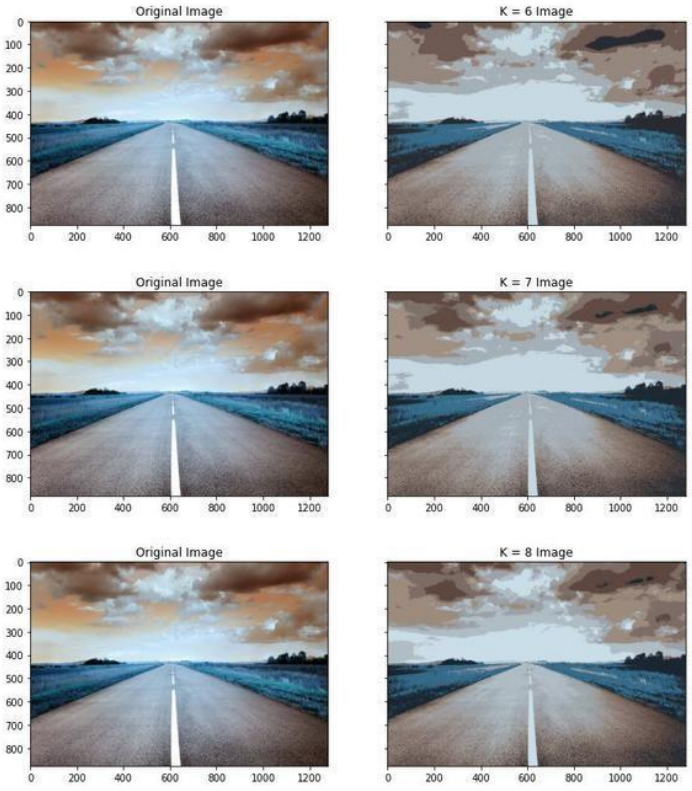
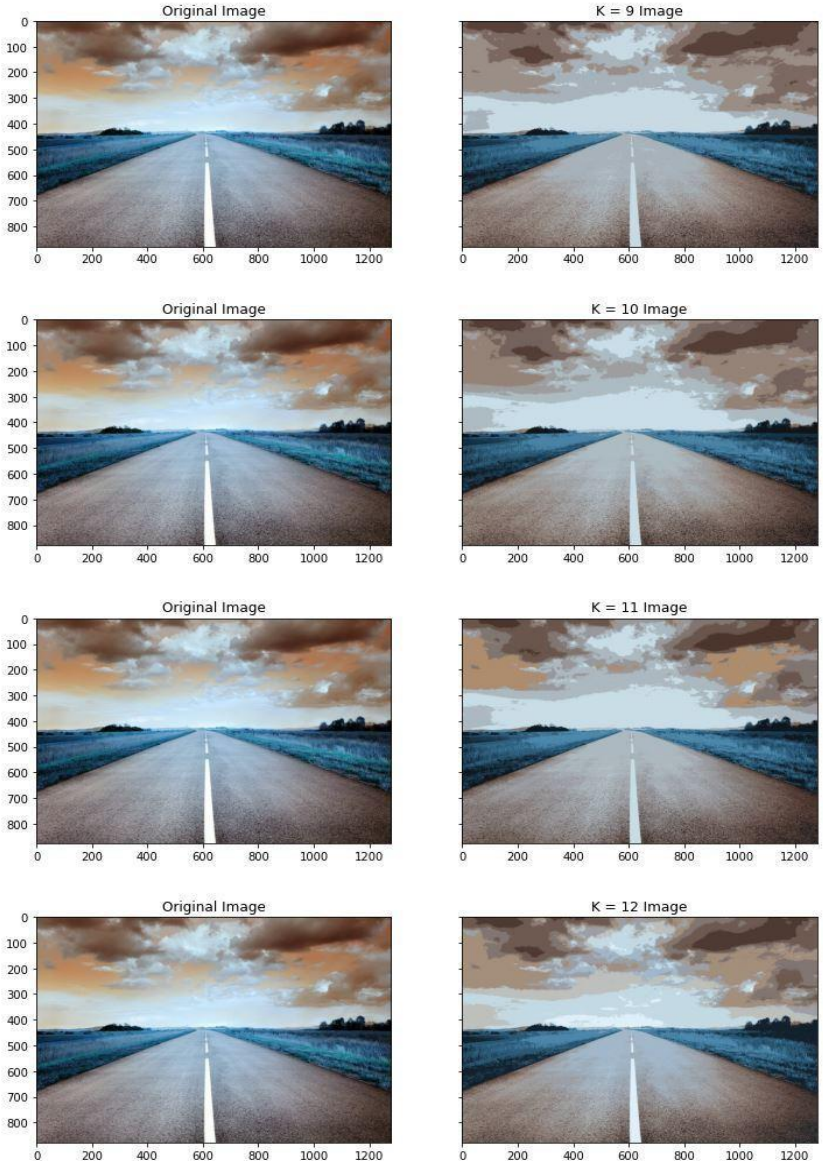


Image Segmentation for K=6,7,8



5.1.3 Variational Methods

In the previous sections it was minimizing the cut in a graph but we also saw examples like the ROF de-noising, k-means and support vector machines. These are examples of optimization problems. When the optimization is taken over functions, the problems are called variational problems and algorithms for solving such problems are called variational methods. Let's look at a simple and effective variational model.

The Chan-Vese segmentation model assumes a piece-wise constant image model for the image regions to be segmented. Here we will focus on the case of two regions, for example foreground and background, but the model extends to multiple regions as well. The model can be described as follows. If we let a collection of curves separate the image into two regions Ω1 and Ω2 as the segmentation model energy.

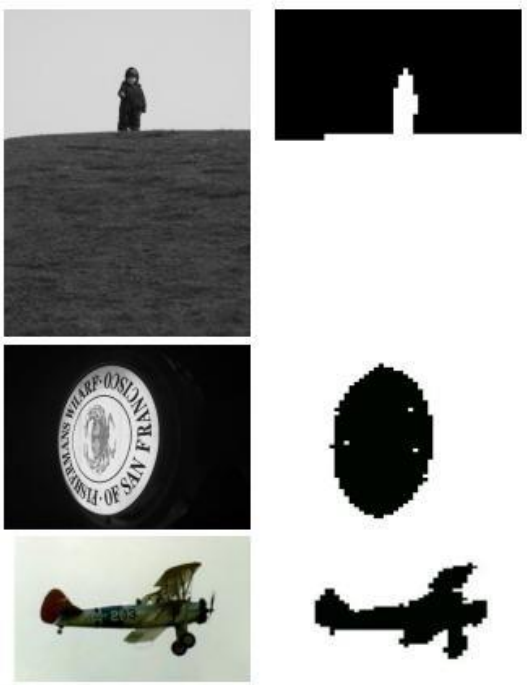


Fig: Examples of two-class image segmentation using the normalized cuts algorithm. (left) original image. (right) segmentation result.

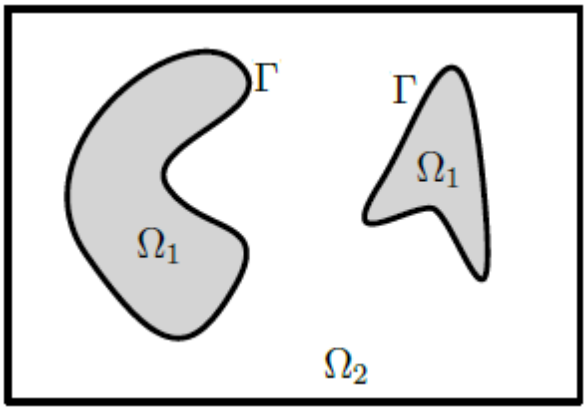


Fig: The piece-wise constant Chan-Vese segmentation model.

The segmentation is given by minima of the Chan-Vese model energy which measures the deviation from the constant graylevels in each region, c1 and c2.
Here the integrals are taken over each region and the length of the
separating curves are there to prefer smoother solutions.

$$E(\Gamma) = \lambda \,\text{length}(\Gamma) + \int_{\Omega_1} (I - c_1)^2 dx + \int_{\Omega_2} (I - c_2)^2 dx$$

With a piece-wise constant image U = (1c1 + (2c2 this can we re-written as

$$E(\Gamma) = \lambda \frac{|c_1 - c_2|}{2} \int |\nabla U| dx + ||I - U||^2$$

Where χ1 and χ2 are characteristic (indicator) functions for the two regions5. This transformation is non-trivial and requires some heavy mathematics that are not needed for understanding and are well outside the scope of this book.

The point is that this equation is now the same as the ROF equation (1.1) with λ replaced by λ |c1 c2|. The only difference is that in the Chan-Vese case we are looking for an image U which is piece-wise constant. It can be shown that thresholding the ROF solution will give a good minimizer. Minimizing the Chan-Vese model now becomes a ROF de-noising followed by thresholding

(a) (b) (c)

Examples image segmentation by minimizing the Chan-Vese model using ROF de-noising.

(a) original image, (b) image after ROF de-noising. (c) final segmentation.

```
import rof

im = array(Image.open('ceramic-houses_t0.png').convert("L"))
U,T = rof.denoise(im,im,tolerance=0.001)
t = 0.4 #threshold

import scipy.misc
scipy.misc.imsave('result.pdf',U < t*U.max())
```

In this case we turn down the tolerance threshold for stopping the ROF iterations to make sure we get enough iterations. Above Examples shows the result on two rather difficult images.

5.2. OpenCV

OpenCV is the huge open-source library for the computer vision, machine learning, and image processing and now it plays a major role in real-time operation which is very important in today's systems. By using it, one can process images and videos to identify objects, faces, or even handwriting of a human. When it integrated with various libraries, such as NumPy, python is capable of processing the OpenCV array structure for analysis. To Identify image pattern and its various features we use vector space and perform mathematical operations on these features.

History of Open CV

The first OpenCV version was 1.0. OpenCV is released under a BSD license and hence it's free for both **academic** and **commercial** use. It has C++, C, Python and Java interfaces and supports Windows, Linux, Mac OS, iOS and Android. When OpenCV was designed the main focus was real- time applications for computational efficiency. All things are written in optimized C/C++ to take advantage of multi-core processing.

Applications of OpenCV

There are lots of applications which are solved using OpenCV, some of them are listed below

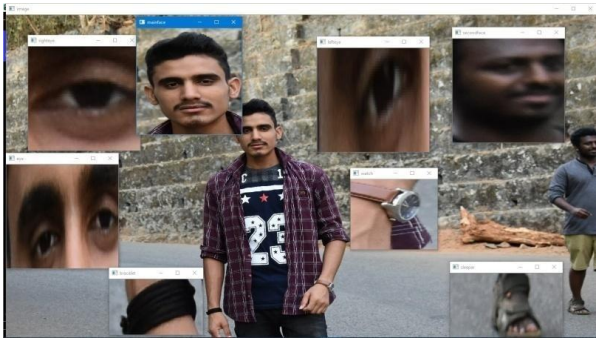
- face recognition
- Automated inspection and surveillance
- number of people – count (foot traffic in a mall, etc)

- Vehicle counting on highways along with their speeds
- Interactive art installations
- Anomaly (defect) detection in the manufacturing process (the odd defective products)
- Street view image stitching
- Video/image search and retrieval
- Robot and driver-less car navigation and control
- object recognition
- Medical image analysis
- Movies – 3D structure from motion
- TV Channels advertisement recognition

OpenCV Functionality

Image/video I/O, processing, display (core, imgproc, highgui)

- Object/feature detection (objdetect, features2d, nonfree)
- Geometry-based monocular or stereo computer vision (calib3d, stitching, videostab)
- Computational photography (photo, video, superres)
- Machine learning & clustering (ml, flann)
- CUDA acceleration (gpu)



The following images from OpenCV

from the above original image, lots of pieces of information that are present in the original image can be obtained. Like in the above image there are two faces available and the person(I) in the images wearing a bracelet, watch, etc so by the help of OpenCV we can get all these types of information from the original image.

5.2.1 Python Interface

OpenCV is a C++ library with modules that cover many areas of computer vision. Besides C++ (and C) there is growing support for Python as a simpler scripting language through a Python interface on top of the C++ code base. The Python interface is still under development and not all parts of OpenCV are exposed and many functions are undocumented. This is likely to change as there is an active community behind this interface.

The old cv module uses internal OpenCV datatypes and can be a little tricky to use from NumPy. The new cv2 module uses NumPy arrays and is much more intuitive to use. The module is available as and the old module can be accessed as We will focus on the cv2 module in this chapter. Look out for future name changes, as well as changes in function names and definitions in future versions. OpenCV and the Python interface is under rapid development.

import cv2 import cv2.cv

5.2.2 OpenCV Basics

OpenCV comes with functions for reading and writing images as well as matrix operations and math libraries.

Reading and writing images

For reading an image, use the imread() function in OpenCV. imread(filename, flags)

Two arguments:

1. The first argument is the image name, which requires a fully qualified **pathname** to the file.
2. The second argument is an optional flag that lets you specify how the image should be represented. OpenCV offers several options for this flag, but those that are most common include:

- cv2.IMREAD_UNCHANGED or -1
- cv2.IMREAD_GRAYSCALE or 0
- cv2.IMREAD_COLOR or 1

flag values

```
img_color = cv2.imread('test.jpg',1)
img_grayscale =
cv2.imread('test.jpg',0)
----------
img_unchanged =
cv2.imread('test.jpg',-1)
```

Color spaces

In OpenCV images are not stored using the conventional RGB color channels, they are stored in BGR order (the reverse order). When reading an image the default is BGR, however there are several conversions available. Color space conversion are done using the function cvtColor().

```
im                          =
cv2.imread('empire.jpg')    #
create a grayscale version
gray = cv2.cvtColor(im,cv2.COLOR_BGR2GRAY)
```

After the source image there is an OpenCV color conversion code. Some of the most useful conversion codes are:

- cv2.COLOR_BGR2GRAY
- cv2.COLOR_BGR2RGB
- cv2.COLOR_GRAY2BGR

In each of these, the number of color channels for resulting images will match the conversion code (single channel for gray and three channels for RGB and BGR).

Displaying an Image

In OpenCV, you display an image using the imshow() function. imshow(window_name, image)

This function also takes two arguments:

1. The first argument is the window name that will be displayed on the window.
2. The second argument is the image that you want to display.

To display multiple images at once, specify a new window name for every image you want to display.
The imshow() function is designed to be used along with the waitKey() and destroyAllWindows() / destroyWindow() functions.

- It takes a single argument, which is the time (in milliseconds), for which the window will be displayed.
- If the user presses any key within this time period, the program continues.
- If 0 is passed, the program waits indefinitely for a keystroke.
- You can also set the function to detect specific keystrokes like the Q key or the ESC key on the keyboard, thereby telling more explicitly which key shall trigger which behavior.

The code examples below show how the imshow() function is used to display the images you read in.

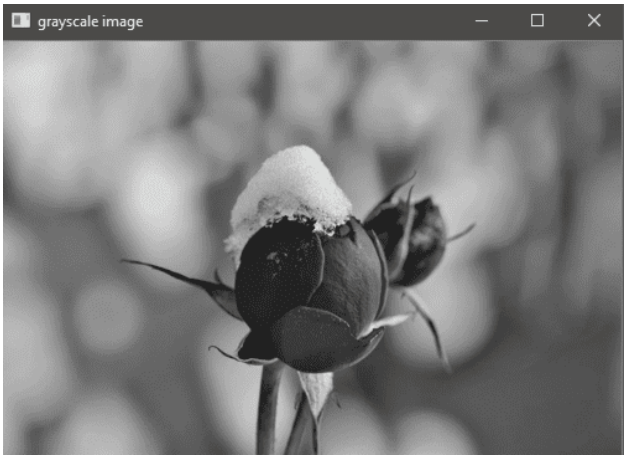
```
#Displays image inside a window
cv2.imshow('color image',img_color)
cv2.imshow('grayscale
image',img_grayscale)
cv2.imshow('unchanged
image',img_unchanged)

# Waits for a
keystroke
cv2.waitKey(0)

# Destroys all the windows
created
cv2.destroyAllwindows()
```

The three output screens shown below, you can see:

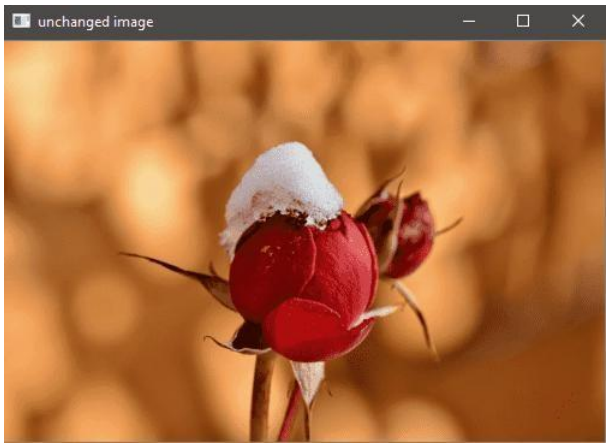
1. The first image is displayed in color
2. The next as grayscale
3. The third is again in color, as this was the original format of the image (which was

read using cv2.IMREAD_UNCHANGED)

Displaying an image in color using the using imshow() function



Displaying an image in grayscale the imshow() function.



Displaying an unchanged image using the imshow() function.

5.2.3 Processing Video

Video processing consists in signal processing employing statistical analysis and video filters to extract information or perform video manipulation. Basic video processing techniques include trimming, image resizing, brightness and contrast adjustment, fade in and fade out, amongst others. More complex video processing techniques, also known as Computer Vision Techniques, are based on image recognition and statistical analysis to perform tasks such as face recognition, detection of certain image patterns, and computer-human interaction.

Video files can be converted, compressed or decompressed using particular software devices. Usually, compression involves a reduction of the bitrate (the number of bits processed per time unit), which makes it possible to store the video digitally and stream it over the network. Uncompressed audio or video usually are called RAW streams, and although different formats and codecs for raw data exist, they appear to be too heavy (in bitrate terms) to be stored or streamed over the network in these formats.

Video input

Reading video from a camera is very well supported in OpenCV. A basic complete example that captures frames and shows them in an OpenCV window looks like this.

Example

```
import cv2
# setup video capture
cap =
cv2.VideoCapture(0)
while True:
ret,im = cap.read()
cv2.imshow('video test',im) key
= cv2.waitKey(10)
if key == 27:
break
if key == ord(' '):
cv2.imwrite('vid_result.jpg',im)
```

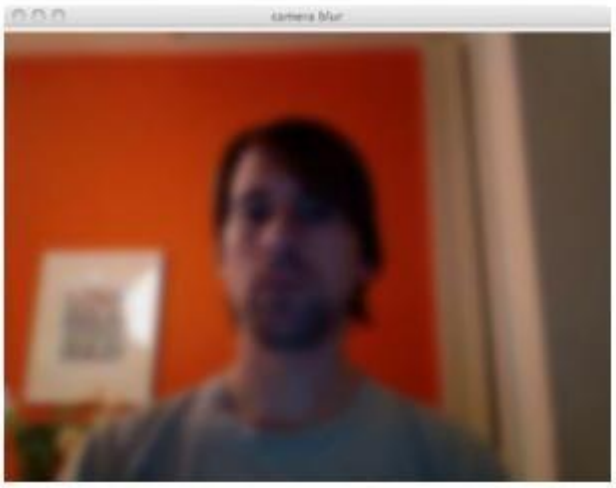
The capture object VideoCapture captures video from cameras or files. Here w pass an integer at initialization. This is the id of the video device, with a single camera connected this is 0. The method read() decodes and returns the next video frame. The first value is a success flag and the second the actual image array. The waitKey() function waits for a key to be pressed and quit the application if the 'esc' key (Ascii number 27) is pressed or saves the frame if the 'space' key is pressed.

The camera input and show a blurred (color) version of the input in an OpenCV window

```
import cv2
# setup video capture
cap = cv2.VideoCapture(0)
# get frame, apply Gaussian smoothing, show
result while True:
ret,im = cap.read()
blur =
cv2.GaussianBlur(im,(0,0),5)
cv2.imshow('camera blur',blur)
if cv2.waitKey(10) == 27:
break
```

Each frame is passed to the function GaussianBlur() which applies a Gaussian filter to the image. In this case we are passing a color image so each color channel is blurred separately. The function takes a tuple for filter size and the standard deviation for the Gaussian function (in this case 5). If the filter size is set to zero, it will automatically be determined from the standard deviation.

Reading video from files works the same way but with the call to VideoCapture() taking the video filename as input.



capture = cv2.VideoCapture('filename')

Screenshot of a blurred video of the author as he's writing this chapter.

5.2.4 Video Tracking

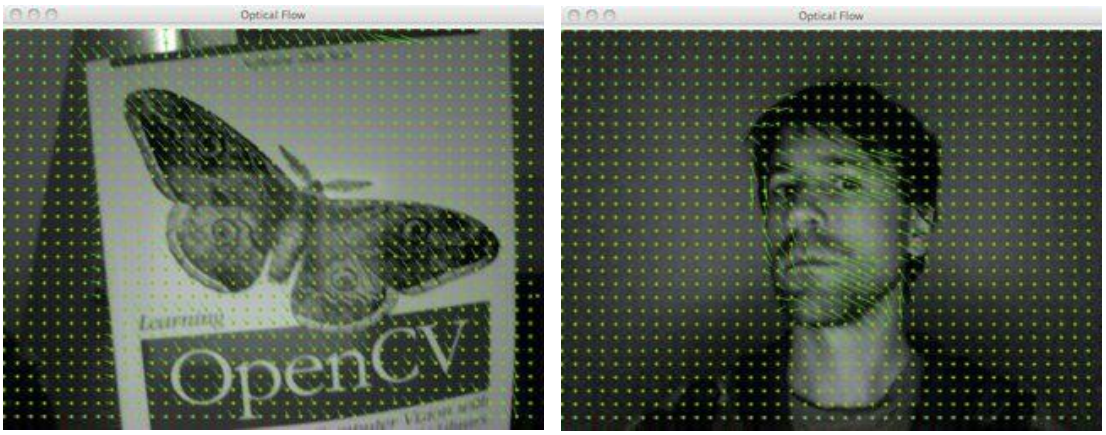
Video tracking is an application of object tracking where moving objects are located within video information. Hence, video tracking systems are able to process live, real-time footage and also recorded video files. The processes used to execute video tracking tasks differ based on which type of video input is targeted. Different videotracking applications play an important role in video analytics, in scene understanding for security and surveillance, military, transportation, and other industries. Today, a wide range of real-time computer vision and deep learning applications use videotracking methods.

Optical flow Optical flow (sometimes called optic flow) is the image motion of objects as the objects, scene or camera moves between two consecutive images. It is a 2D vector field of within- image translation. Is is a classic and well studied field in computer vision with many successful applications in for example video compression, motion estimation, object tracking and image segmentation.

Optical flow relies on three major assumptions.

1. Brightness constancy: The pixel intensities of an object in an image does not change between consecutive images.
2. Temporal regularity: The between-frame time is short enough to consider the motion change between images using differentials (used to derive the central equation below).
3. Spatial consistency: Neighboring pixels have similar motion.

```
import cv2

def draw_flow(im,flow,step=16):
  """ Plot optical flow at sample points
    spaced step pixels apart. """

  h,w = im.shape[:2]
  y,x = mgrid[step/2:h:step,step/2:w:step].reshape(2,-1)
  fx,fy = flow[y,x].T

  # create line endpoints
  lines = vstack([x,y,x+fx,y+fy]).T.reshape(-1,2,2)
  lines = int32(lines)

  # create image and draw
  vis = cv2.cvtColor(im,cv2.COLOR_GRAY2BGR)
  for (x1,y1),(x2,y2) in lines:
    cv2.line(vis,(x1,y1),(x2,y2),(0,255,0),1)
    cv2.circle(vis,(x1,y1),1,(0,255,0), -1)
  return vis


# setup video capture
cap = cv2.VideoCapture(0)

ret,im = cap.read()
prev_gray = cv2.cvtColor(im,cv2.COLOR_BGR2GRAY)

while True:
  # get grayscale image
  ret,im = cap.read()
  gray = cv2.cvtColor(im,cv2.COLOR_BGR2GRAY)

  # compute flow
  flow = cv2.calcOpticalFlowFarneback(prev_gray,gray,None,0.5,3,15,3,5,1.2,0)
prev_gray = gray

# plot the flow vectors
cv2.imshow('Optical flow',draw_flow(gray,flow))
if cv2.waitKey(10) == 27:
  break
```



Optical flow vectors (sampled at every 16th pixel) shown on video of a translating book and a turning head.

Lucas-Kanade method

Tracking is the process of following objects through a sequence of images or video. The most basic form of tracking is to follow interest points such as corners. A popular algorithm for this is the Lucas-Kanade tracking algorithm which uses a sparse optical flow algorithm.

OpenCV is a huge open-source library for computer vision, machine learning, and image processing. OpenCV supports a wide variety of programming languages like Python, C++, Java, etc. It can process images and videos to identify objects, faces, or even the handwriting of a human. When it is integrated with various libraries, such as Numpy which is a highly optimized library for numerical operations, then the number of weapons increases in your Arsenal i.e whatever operations one can do in Numpy can be combined with OpenCV.

- Points will be tracked using the Lucas-Kanade Algorithm provided by OpenCV, i.e, cv2.calcOpticalFlowPyrLK().

Syntax: cv2.calcOpticalFlowPyrLK(prevImg, nextImg, prevPts, nextPts[, winSize[, maxLevel[, criteria]]])

Parameters:

prevImg – first 8-bit input image

nextImg – second input image

prevPts – vector of 2D points for which the flow needs to be found.

winSize – size of the search window at each pyramid level.

maxLevel – 0-based maximal pyramid level number; if set to 0, pyramids are not used (single level), if set to 1, two levels are used, and so on.

criteria – parameter, specifying the termination criteria of the iterative search algorithm.

Return:

nextPts – output vector of 2D points (with single-precision floating-point coordinates) containing the calculated new positions of input features in the second image; when OPTFLOW_USE_INITIAL_FLOW flag is passed, the vector must have the same size as in the input.

status – output status vector (of unsigned chars); each element of the vector is set to 1 if the flow for the corresponding features has been found, otherwise, it is set to 0.

err – output vector of errors; each element of the vector is set to an error for the corresponding feature, type of the error measure can be set in flags parameter; if the flow wasn't found then the error is not defined (use the status parameter to find such cases).

Example

```
import numpy as np
import cv2
cap =
cv2.VideoCapture('sample.mp4') #
params for corner detection
feature_params = dict( maxCorners = 100, qualityLevel = 0.3, minDistance = 7, blockSize = 7
)

# Parameters for lucas kanade optical flow
lk_params = dict( winSize = (15, 15),          maxLevel = 2, criteria =
(cv2.TERM_CRITERIA_EPS | cv2.TERM_CRITERIA_COUNT, 10, 0.03))

# Create some random colors
color = np.random.randint(0, 255, (100,
3)) # Take first frame and find corners
in it
ret, old_frame = cap.read()
old_gray = cv2.cvtColor(old_frame, cv2.COLOR_BGR2GRAY)
p0 = cv2.goodFeaturesToTrack(old_gray, mask = None, **feature_params)

# Create a mask image for drawing
purposes mask =
np.zeros_like(old_frame)
while(1):

ret, frame = cap.read()
frame_gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

# calculate optical flow
```

```
p1, st, err = cv2.calcOpticalFlowPyrLK(old_gray,       frame_gray, p0, None, **lk_params)

# Select good points
good_new = p1[st ==
1] good_old = p0[st
== 1]

# draw the tracks
for i, (new, old) in enumerate(zip(good_new,
good_old)): a, b = new.ravel()
c, d = old.ravel()
mask = cv2.line(mask, (a, b), (c,
d), color[i].tolist(), 2)

frame = cv2.circle(frame, (a, b),
5, color[i].tolist(), -1)

img = cv2.add(frame,
mask)
cv2.imshow('frame', img)
k =
cv2.waitKey(25) if
k == 27:
break
# Updating Previous frame and
points old_gray =
frame_gray.copy()
p0 = good_new.reshape(-1,
1, 2)
cv2.destroyAllWindows()
cap.release()
```

Output

Unit Summary:

Image Segmentation is the process of partitioning an image into distinct regions or segments, often to simplify or change the representation of an image into something more meaningful and easier to analyze. OpenCV offers a comprehensive suite of tools for image and video processing, with a user-friendly Python interface that facilitates rapid development and integration of computer vision solutions.

Let us sum up:

Image Segmentation is the process of dividing an image into distinct, meaningful regions to simplify analysis. Key methods include:

Graph Cuts:Models the segmentation problem as a graph where nodes represent pixels or superpixels and edges represent similarity between them. The goal is to find a cut that minimizes a cost function balancing the segmentation's smoothness and fidelity.

Segmentation Using Clustering:roups pixels into clusters based on feature similarity, such as color or texture. Common algorithms include K-Means and Mean Shift.

Variational Methods:Formulates segmentation as an optimization problem, minimizing an objective function to find the best region boundaries. Techniques like the Chan-Vese model use level sets for evolving contours.

OpenCV (Open Source Computer Vision Library) provides tools for image and video processing, with a focus on practical applications and ease of use.

Python Interface:Allows for integrating OpenCV functions into Python programs, facilitating rapid development and prototyping in computer vision tasks.

OpenCV Basics:Includes fundamental operations such as image reading, writing, resizing, cropping, and applying basic filters and transformations.

Processing Video:Involves capturing and processing video streams, including frame extraction and applying filters to video frames.

Tracking: Techniques for tracking objects or features across frames in a video. OpenCV offers various tracking algorithms, including Mean Shift and Kalman Filters.

Check your progress:

1. Which of the following methods is used to segment images by minimizing a cost function that balances between smoothness and fidelity?

A) K-Means Clustering

B) Mean Shift Clustering

C) Graph Cuts

D) Chan-Vese Model

Answer: C

Explanation: Graph Cuts involves modeling the segmentation problem as a graph where nodes represent pixels and edges represent the similarity between pixels. The method minimizes a cost function to find an optimal partition, balancing the smoothness of the segmentation with its fidelity to the actual image content.

2. Which image segmentation technique uses clustering algorithms to group pixels into segments based on their feature similarity?

A) Variational Methods

B) Graph Cuts

C) Segmentation Using Clustering

D) Epipolar Geometry

Answer: C

Explanation: Segmentation Using Clustering involves grouping pixels into clusters based on feature similarity (like color or texture) using clustering algorithms such as K-Means. This method is useful for segmenting images where regions with similar attributes need to be identified.

3. What is the primary approach of Variational Methods in image segmentation?

A) Using pre-defined labels to segment images

B) Applying geometric transformations to images

C) Formulating segmentation as an optimization problem to minimize an objective function

D) Using histograms of pixel intensities for segmentation

Answer: C

Explanation: Variational Methods approach image segmentation by formulating it as an optimization problem. The goal is to minimize an objective function that defines the best segmentation boundaries based on various criteria.

4. What advantage does the Python interface of OpenCV provide?

A) It supports direct hardware interfacing for real-time image processing.

B) It simplifies the integration of computer vision algorithms into Python applications for rapid development.

C) It offers advanced 3D rendering capabilities.

D) It automates image segmentation without manual input.

Answer: B

Explanation: The Python interface of OpenCV allows developers to rapidly develop and prototype computer vision algorithms in Python, providing a user-friendly way to implement and test vision-based applications.

5. Which function in OpenCV is used for basic image manipulation tasks like resizing or cropping?

A) cv2.VideoCapture

B) cv2.resize

C) cv2.imshow

D) cv2.dilate

Answer: B

Explanation: The cv2.resize function is used in OpenCV for resizing images. Basic image manipulation tasks such as cropping are performed using other related functions provided by the library.

5. In OpenCV, which function is used to track objects or features across video frames?

A) cv2.findContours

B) cv2.TrackObject

C) cv2.VideoCapture

D) cv2.Tracker

Answer: D

Explanation: The cv2.Tracker class in OpenCV provides various tracking algorithms to follow objects or features across video frames. This functionality is essential for applications involving object tracking and motion analysis.

6. To capture video from a camera and process it frame by frame in OpenCV, which function is typically used?

A) cv2.imshow

B) cv2.VideoCapture

C) cv2.drawMatches

D) cv2.findContours

Answer: B

Explanation: cv2.VideoCapture is used to capture video from a camera or read video files. It allows for processing each frame individually, which is crucial for tasks such as real-time video analysis and frame extraction.

Glossary

Graph Cuts:A method for image segmentation that models the problem as a graph. Pixels or superpixels are represented as nodes, and the edges between them represent similarity. The goal is to find a partition (or "cut") that minimizes a cost function balancing smoothness and fidelity.

Segmentation Using Clustering: A technique that groups pixels into clusters based on feature similarity, such as color or texture. Common algorithms include K-Means and Mean Shift.

Variational Methods:Techniques that formulate image segmentation as an optimization problem. An objective function is minimized to find the best boundaries between segments. Methods like the Chan-Vese model use level sets for evolving contours.

Python Interface: The interface provided by OpenCV for integrating computer vision functions into Python programs. It simplifies the development and testing of computer vision algorithms in Python.

OpenCV Basics: Fundamental operations provided by OpenCV for image processing, including reading, writing, resizing, cropping, and applying basic filters and transformations.
Processing Video: Involves capturing and manipulating video streams using OpenCV. This includes tasks like extracting frames, applying filters, and analyzing video content.

Tracking: Techniques for following objects or features across frames in a video. OpenCV provides algorithms such as Mean Shift and Kalman Filters for object tracking.

Books

1. "Computer Vision: Algorithms and Applications" by Richard Szeliski
2. "Digital Image Processing" by Rafael C. Gonzalez and Richard E. Woods
3. Learning OpenCV 4: Computer Vision with Python" by Adrian Kaehler and Gary Bradski

Web Resources

1. Graph Cuts and Image Segmentation (PDF from Carnegie Mellon University)
2. Image Segmentation Using K-Means Clustering (Towards Data Science)
3. OpenCV Python Tutorials (OpenCV Documentation)
4. Object Tracking with OpenCV (OpenCV Documentation)