

B.E. PROJECT ON

HAND GESTURE RECOGNITION

SUBMITTED BY

RITVIK SADANA	(134/EC/12)
SARTAJ SINGH BAVEJA	(145/EC/12)
SHASHANK JAIN	(151/EC/12)
UTKARSH GARG	(177/EC/12)

UNDER THE GUIDANCE OF
DR. JYOTSNA SINGH

A PROJECT IN PARTIAL FULFILLMENT OF REQUIREMENT FOR THE
AWARD OF
B.E. IN ELECTRONICS & COMMUNICATION ENGINEERING



DIVISION OF ELECTRONICS AND COMMUNICATIONS ENGINEERING
NETAJI SUBHAS INSTITUTE OF TECHNOLOGY
(UNIVERSITY OF DELHI)
NEW DELHI – 110078
JUNE 2016

ACKNOWLEDGEMENT

We are sincerely thankful to **Dr. Jyotsna Singh** for making this project possible and constantly motivating us. We would like to articulate our deep gratitude for her guidance, advice and constant support in the project work. We would like to thank her for being our guide here at Netaji Subhas Institute of Technology, Delhi. We would like to thank all faculty members and staff of the department of Electronics and Communication Engineering (ECE) for their generous help in various ways for this project.

Last but not the least, our sincere thanks to all of our friends who have patiently extended all sorts of help in this project.



स्थान

NETAJI SUBHAS INSTITUTE OF TECHNOLOGY

(Formerly Delhi Institute of Technology)

Azad Hind Fauz Marg, Sector 3, Dwarka, New Delhi, Delhi 110078

Telephone: 011 2509 9050

CERTIFICATE

This is to certify that the thesis titled “**Hand Gesture Recognition**” submitted by:

RITVIK SADANA	(134/EC/12)
SARTAJ SINGH BAVEJA	(145/EC/12)
SHASHANK JAIN	(151/EC/12)
UTKARSH GARG	(177/EC/12)

is a record of bonafide work carried out by them in the Division of Electronics & Communication Engineering, Netaji Subhas Institute Of Technology, New Delhi during session 2012-2016 under the supervision and guidance of the undersigned in partial fulfillment of the requirements for the award of Bachelor of Engineering degree in Electronics and Communication Engineering.

Dr. Jyotsna Singh
Division of Electronics & Communication
Netaji Subhas Institute of Technology
New Delhi - 110078

ABSTRACT

Hand gesture recognition system can be used for interfacing between computer and human using hand gesture. This work presents a technique for a human computer interface through hand gesture recognition that is able to recognize static gestures from the standard numerical gestures. The objective of this thesis is to develop an algorithm for efficient segmentation and recognition of hand gestures with reasonable accuracy.

The segmentation of hands is performed using the Codebook method for background subtraction. In the method, the hand is removed at first and the program is trained for the give background. After sufficient number of frames are recorded, a Codebook model is formed against which all the subsequent frames are compared. If any difference, the resultant is shown as a white hole on black background. This may contain a few holes which are removed using masking the segments.

For the feature extraction, Krawtchouk moments are used to create a feature vector. A matrix is formed containing the features of all the images from the training data set. The extracted features are applied to a knn classifier. The K-nearest neighbours are then found for the given input image and the hand gesture is predicted accordingly.

MOTIVATION

Despite a lot of previous researches and works, traditional vision-based hand gesture recognition methods are still far from satisfactory for real-life applications. Because of the limitations of the optical sensors, the quality of the captured images is sensitive to lighting conditions and cluttered backgrounds, thus it is usually not able to detect and track the hands robustly, which largely affects the performance of hand gesture recognition. Due to poor image quality as mentioned above, hand segmentation is a difficult task. Also, the hardware involved in this process is expensive & not commonly used, hence not available easily.

TABLE OF CONTENTS

Chapter 1: Introduction

- 1.1 Human Computer Interface
- 1.2 Literature Survey
- 1.3 Hand gesture recognition
- 1.4 Steps for hand gesture recognition

Chapter 2: Preprocessing

- 2.1 Codebook method
 - 2.1.1 Background Subtraction
 - 2.1.2 Weaknesses of Background Subtraction
 - 2.1.3 Scene Modeling
 - 2.1.4 A Slice of Pixels

Chapter 3: Feature Extraction

- 3.1 Moments
- 3.2 Orthogonal Moments
 - 3.2.1 Types of Orthogonal Moments
- 3.3 Moment Invariants
- 3.4 Krawtchouk Moments
 - 3.4.1 Krawtchouk *Polynomials*
 - 3.4.2 Krawtchouk *Moments*
 - 3.4.3 Hypergeometric *Function*

Chapter 4: Classification

- 4.1 Knn classifier algorithm
- 4.2 Application of Knn on current data
 - 4.2.1 Population of the Database
 - 4.2.2 Testing Knn Scheme

Chapter 5: Experimental Setup and Results

Chapter 6: References

Chapter 7: Appendix

- A1: OpenCV
- A2: Code

INTRODUCTION

1.1 HUMAN COMPUTER INTERFACE SYSTEM

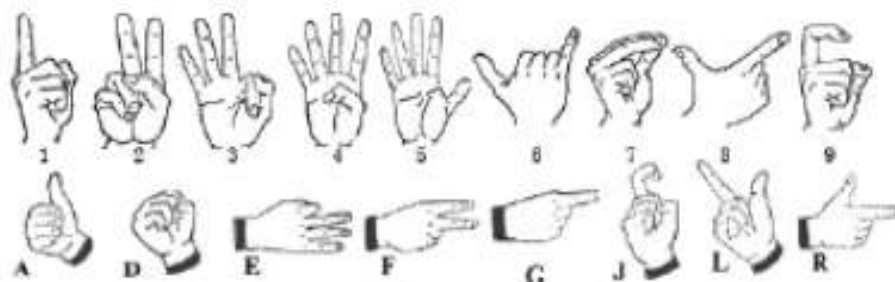
Gesture recognition is a method implemented with the goal of interpreting human gestures via mathematical algorithms. Gestures can originate from any bodily motion or state but commonly originate from the face or hand. Current focuses in the field include emotion recognition from face and hand gesture recognition. Many approaches have been made using cameras and computer vision algorithms to interpret sign language. However, the identification and recognition of posture, gait, proxemics, and human behaviours is also the subject of gesture recognition techniques.

Gesture recognition can be seen as a way for computers to begin to understand human body language, thus building a richer bridge between machines and humans than primitive text user interfaces or even GUIs (graphical user interfaces), which still limit the majority of input to keyboard and mouse.

Gesture recognition enables humans to communicate with the machine and interact naturally without any mechanical devices. Using the concept of gesture recognition, it is possible to point a finger at the computer screen so that the cursor will move accordingly. This could potentially make conventional input devices such as mouse, keyboards and even touch-screens redundant.

Gesture recognition should be made easy, natural and convenient, without gloves. It may find its application in games control, robotics control, virtual reality, and smart homes systems.




Gesture recognition can be conducted with techniques from computer vision and image processing. The literature includes on-going work in the computer vision field on capturing gestures or more general human pose and movements by cameras connected to a computer.



Input devices

The ability to track a person's movements and determine what gestures they may be performing can be achieved through various tools. Although there is a large amount of research done in image/video based gesture recognition, there is some variation within the tools and environments used between implementations.

1. **Wired gloves.** These can provide input to the computer about the position of the hands using magnetic or inertial tracking devices. Furthermore, some gloves can detect finger bending with a high degree of accuracy (5-10 degrees), or even provide haptic feedback to the user, which is a simulation of the sense of touch.
2. **Depth-aware cameras.** Using specialized cameras such as structured light or time-of-flight cameras, one can generate a depth map of what is being seen through the camera at a short range, and use this data to approximate a 3d representation of what is being seen. These can be effective for detection of hand gestures due to their short range capabilities.
3. **Stereo cameras.** Using two cameras whose relations to one another are known, a 3d representation can be approximated by the output of the cameras.
4. **Controller-based gestures.** These controllers act as an extension of the body so that when gestures are performed, some of their motion can be conveniently captured by software. Mouse gestures are one such example, where the motion of the mouse is correlated to a symbol being drawn by a person's hand, as is the Wii Remote
5. **Single camera.** A standard 2D camera can be used for gesture recognition where the resources/environment would not be convenient for other forms of image-based recognition. Software-based gesture recognition technology using a standard 2D camera that can detect robust hand gestures.

The hand gesture definition	5	2	0
Image			

1.2 LITERATURE SURVEY

Human-computer interaction (HCI) is an interesting area of research factoring development in the field of automaton. The recent advancement has led to the emergence of HCI systems that embody the ‘natural’ way of communication between humans. Therefore, attempts in integrating communication modalities like speech, handwriting and hand gestures into HCI have gained importance. Researchers have focused in developing advanced hand gesture interfaces resulting in successful applications like robotics, assistive systems, sign language communication and virtual reality.

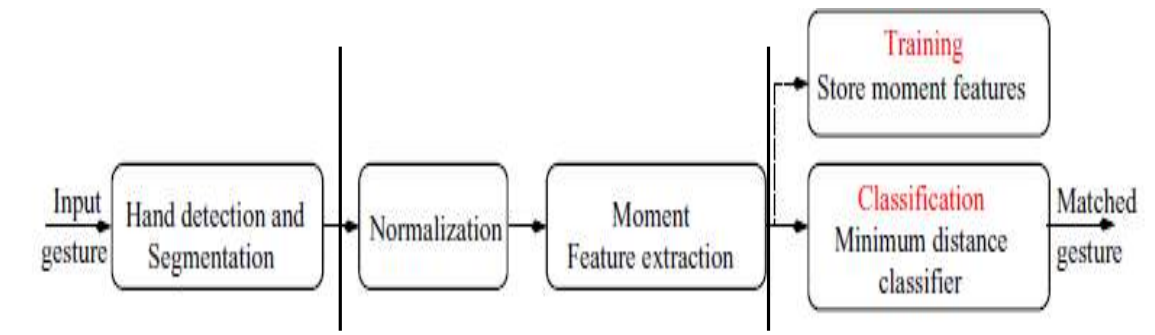
Researchers have successfully developed some scale and rotation invariant features, while very few works concentrate on view and user independence. These problems are addressed to some extent using the *3D* hand modelling techniques [1]. The research on developing view and user independent methods based on *2D* hand models is yet to mature.

Application of continuous orthogonal moments for hand recognition has been actively researched either on the improvement of existing techniques, hybrid techniques or fusion techniques. The geometric moment invariants derived from the binary hand silhouettes form the feature set in [2] and are not robust to view point variations. The Zernike and pseudo-Zernike moment features for rotation invariant gesture recognition is introduced in [3]. Gu and Su [4] investigated the Zernike moment features for view and user independent representations. But these are continuous moments.

KMs and TMs are discrete orthogonal moments and not commonly used in image application especially in face recognition. KMs are normally used to localize image and able to segment the image according to Region of Interest (ROI) while TMs have been reported to be able to reconstruct image with lower reconstruction error compared to using continuous orthogonal moments [5][6][7].

A recent study by S. Padam Priyal and Prabin. K. Bora [8] on hand gesture recognition using various orthogonal moments has shown Krawtchouk moments and Tchebichef to be better forming than other moments in image reconstruction and detection.

1.3 HAND GESTURE RECOGNITION



The HGR system can be divided into three parts according to its processing steps: hand detection and segmentation, moment feature extraction and gesture recognition. The system has two major advantages. First, it is highly modularized, and each of the three steps is capsuled from others. Second, the edge/contour detection of hand as well as gesture recognition is an add-on layer, which can be easily transplanted to other applications. This system has several key features:

- Capable of capturing images via normal web camera of a laptop
- Identifying and Segmenting Hands using Codebook Method for Background Subtraction
- Extracting Higher Order Moments from images
- Classifying Images using K-Nearest Neighbour Algorithm

1.4 STEPS OF HAND GESTURE RECOGNITION

1.4.1. Image Capture

We use the web camera of a laptop to capture images and create a dataset. The image is captured when a user input via key is detected. At that moment, the image is saved onto the current directory.

1.4.2. Hand Detection and Segmentation

The original images used for hand gesture recognition in the work are demonstrated in the above figure. These images are captured with a normal camera. These hand images are taken under the same condition. The background of these images is identical. So, it is easy and effective to detect the hand region from the original image using the background subtraction method.

1.4.3. Normalization

The binary hand images are normalized for orientation changes and scale variations. The image is aligned such that the major axis lying along the forearm region is at 90 degrees with respect to the horizontal axis of the image. The resolution of the resultant image is fixed at 350×350 pixels with the hand object normalized to 200×200 pixels

1.4.4. Moment Feature Extraction

The moments computed from the normalized hand gesture image form the feature vectors. The orders of the orthogonal moments are selected experimentally based on its accuracy in reconstruction. The order of geometric moments is chosen based on the recognition performance.

1.4.5. Training and Classification using Minimum Distance Classifier

Classification mainly consists of finding the best matching reference features for the features extracted in the previous phase. Classification is done using the minimum distance classifier.



PREPROCESSING

2.1 BACKGROUND SUBTRACTION

Here we talk about how to isolate objects or parts of objects from the rest of the image. The reasons for doing this should be obvious. In video security, for example, the camera mostly looks out on the same boring background, which really isn't of interest.

What is of interest is when people or vehicles enter the scene, or when something is left in the scene that wasn't there before. We want to isolate those events and to be able to ignore the endless hours when nothing is changing.

Beyond separating foreground objects from the rest of the image, there are many situations where we want to separate out parts of objects, such as isolating just the face or the hands of a person. We might also want to preprocess an image into meaningful *super pixels*, which are segments of an image that contain things like limbs, hair, face, torso, tree leaves, lake, path, lawn and so on. Using super pixels saves on computation; for example, when running an object classifier over the image, we only need search a box around each super pixel. We might only track the motion of these larger patches and not every point inside.

2.1.1 Codebook Method [9]

Because of its simplicity and because camera locations are fixed in many contexts, *background subtraction* (aka *background differencing*) is probably the most fundamental image processing operation for video security applications.

In order to perform background subtraction, we first must “learn” a model of the background. Once learned, this *background model* is compared against the current image and then the known background parts are subtracted away. The objects left after subtraction are presumably new foreground objects.

“Background” is an ill-defined concept that varies by application. For example, if you are watching a highway, perhaps average traffic flow should be considered background. Normally, background is considered to be any static or periodically moving parts of a scene that remain static or periodic over the period of interest. The whole ensemble may have time-varying components, such as trees waving in morning and evening wind but standing still at noon.

Two common but substantially distinct environment categories that are likely to be encountered are indoor and outdoor scenes.

First we will discuss the weaknesses of typical background models and then will move on to discuss higher-level scene models. Next we present a quick method that is mostly good for indoor static background scenes whose lighting doesn't change much.

We will follow this by a “**codebook**” method that is slightly slower but can work in both outdoor and indoor scenes; it allows for periodic movements (such as trees waving in the wind) and for lighting to change slowly or periodically. This method is also tolerant to learning the background even when there are occasional foreground objects moving by.

We'll top this off by another discussion of connected components in the context of cleaning up foreground object detection. Finally, we'll compare the quick background method against the codebook background method.

2.1.2 Weaknesses of Codebook method

Although the background modelling methods mentioned here work fairly well for simple scenes, they suffer from an assumption that is often violated: that all the pixels are independent.

The methods we describe learn a model for the variations a pixel experiences without considering neighbouring pixels. In order to take surrounding pixels into account, we could learn a multipart model, a simple example of which would be an extension of our basic independent pixel model to include a rudimentary sense of the brightness of neighbouring pixels.

In this case, we use the brightness of neighbouring pixels to distinguish when neighbouring pixel values are relatively bright or dim. We then learn effectively two models for the individual pixel: one for when the surrounding pixels are bright and one for when the surrounding pixels are dim. In this way, we have a model that takes into account the surrounding *context*.

But this comes at the cost of twice as much memory use and more computation, since we now need different values for when the surrounding pixels are bright or dim. We also need twice as much data to fill out this two-state model.

We can generalize the idea of “high” and “low” contexts to a multidimensional histogram of single and surrounding pixel intensities as well as make it even more complex by doing all this over a few time steps.

Of course, this richer model over space and time would require still more memory, more collected data samples and more computational resources.

Because of these extra costs, the more complex models are usually avoided. We can often more efficiently invest our resources in cleaning up the *false positive* pixels that result when the independent pixel assumption is violated.

2.1.3 Scene Modelling

How do we define background and foreground? If we're watching a parking lot and a car comes in to park, then this car is a new foreground object. But should it stay foreground forever? How about a trash can that was moved? It will show up as foreground in two places: the place it was moved to and the “hole” it was moved from. How do we tell the difference? And again, how long should the trash can (and its hole) remain foreground?

If we are modelling a dark room and suddenly someone turns on a light, should the whole room become foreground? To answer these questions, we need a higher-level “scene” model, in which we define multiple levels between foreground and background states, and a timing-based method of slowly relegating unmoving foreground patches to background patches. We will also have to detect and create a new model when there is a global change in a scene.

In general, a scene model might contain multiple layers, from “new foreground” to older foreground on down to background. There might also be some motion detection so that, when an object is moved, we can identify both its “positive” aspect (its new location) and its “negative” aspect (its old location, the “hole”).

In this way, a new foreground object would be put in the “new foreground” object level and marked as a positive object or a hole. In areas where there was no foreground object, we could continue updating our background model. If a foreground object does not move for a given time, it is demoted to “older foreground,” where its pixel statistics are provisionally learned until its learned model joins the learned background model.

For global change detection such as turning on a light in a room, we might use global frame differencing. For example, if many pixels change at once then we could classify it as a global rather than local change and then switch to using a model for the new situation.

2.1.4 A Slice of Pixels

Before we go on to modelling pixel changes, let’s get an idea of what pixels in an image can look like over time. Consider a camera looking out a window to a scene of a tree blowing in the wind.

Figure shows what the pixels in a given line segment of the image look like over 60 frames. We wish to model these kinds of fluctuations. Before doing so, however, we make a small digression to discuss how we sampled this line because it’s a generally useful trick for creating features and for debugging.

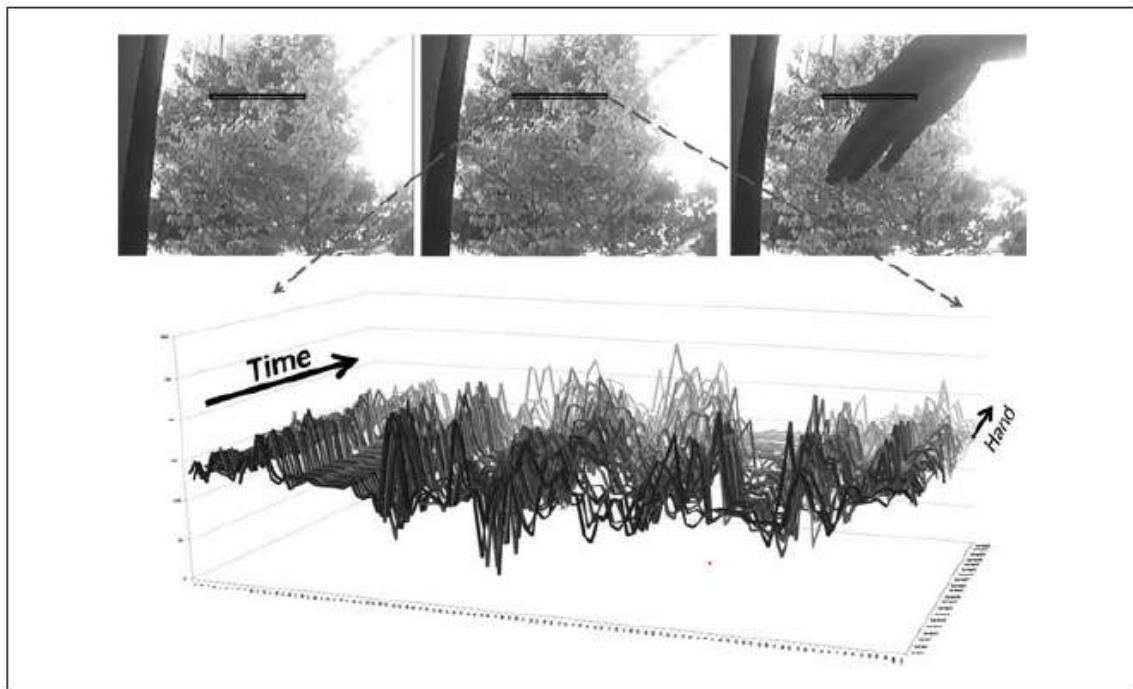


Figure 9-1. Fluctuations of a line of pixels in a scene of a tree moving in the wind over 60 frames: some dark areas (upper left) are quite stable, whereas moving branches (upper center) can vary widely

FEATURE EXTRACTION

3.1 MOMENTS

Recognition of objects and patterns that are deformed in various ways has been a goal of much recent research. There are basically three major approaches to this problem – **brute force, image normalization and invariant features**. In the brute-force approach we search the parametric space of all possible image degradations. That means the training set of each class should contain not only all class representatives but also all their rotated, scaled, blurred and deformed versions. Clearly, this approach would lead to extreme time complexity and is practically inapplicable. In the normalization approach, the objects are transformed into a certain standard position before they enter the classifier. This is very efficient in the classification stage but the object normalization itself usually requires the solving of difficult inverse problems that are often ill-conditioned or ill-posed. For instance, in the case of image blurring, “normalization” means in fact blind de-convolution and in the case of spatial image deformation, “normalization” requires registration of the image to be performed to some reference frame.

The approach using invariant features appears to be the most promising and has been used extensively. Its basic idea is to describe the objects by a set of measurable quantities called *invariants* that are insensitive to particular deformations and that provide enough discrimination power to distinguish objects belonging to different classes. From a mathematical point of view, invariant I is a functional defined on the space of all admissible image functions that does not change its value under degradation operator D , i.e. that satisfies the condition $I(f) = I(D(f))$ for any image function f . This property is called *invariance*.

The existing invariant features used for describing 2D objects can be categorized from various points of view. Most straightforward is the categorization according to the type of invariance. We recognize translation, rotation, scaling, affine, projective, and elastic geometric invariants.

Radiometric invariants exist with respect to linear contrast stretching, nonlinear intensity transforms, and to convolution.

Categorization according to the mathematical tools used may be as follows:

1. simple shape descriptors – compactness, convexity, elongation, etc.
2. transform coefficient features are calculated from a certain transform of the image
3. point set invariants use positions of dominant points
4. differential invariants employ derivatives of the object boundary
5. moment invariants are special functions of image moments.

Another viewpoint reflects what part of the object is needed to calculate the invariant.

1. *Global* invariants are calculated from the whole image (including background if no segmentation was performed).
2. *Local* invariants are, in contrast, calculated from a certain neighbourhood of dominant points only.
3. *Semilocal* invariants attempt to retain the positive properties of the two groups above and to avoid the negative ones. They divide the object into stable parts (most often this division is based on inflection points or vertices of the boundary) and describe each part by some kind of global invariant.

3.2 ORTHOGONAL MOMENTS

They are moments to an orthogonal or weighted-orthogonal polynomial basis [10]. The main reasons for introducing them are: stable and fast numerical implementation (OG polynomials can be evaluated by recurrent relations which can be efficiently implemented by means of multiplication with special matrices); avoidance of high dynamic range of moment values that may lead to the loss of precision due to overflow or underflow (unlike standard powers, the values of OG polynomials lie inside a narrow interval such as $(-1, 1)$ and a higher robustness to random noise reached.

3.2.1 Types of Orthogonal Moments

- **Legendre moments:**

The Legendre moments are defined as

$$\lambda_{mn} = \frac{(2m+1)(2n+1)}{4} \int_{-1}^1 \int_{-1}^1 P_m(x) P_n(y) f(x, y) dx dy \quad m, n = 0, 1, 2, \dots,$$

where $P_n(x)$ is the n th degree Legendre polynomial.

Calculation of higher order moments takes less computation cost.

- **Zernike Moments and Pseudo-Zernike moments:**

These moments give the feature vector in a polar form[4]. Higher order gives the local info and lower order gives global info.

- **Tchebichef moments:**

Since they are discrete, therefore there is no loss of information in the moment set.

- **Krawtchouk**

moments:

These invariants are constructed using a linear combination of geometric moment invariants[5]. A new set of orthogonal moments based on the discrete classical Krawtchouk polynomials is introduced. The Krawtchouk polynomials are scaled to ensure numerical stability, thus creating a set of weighted Krawtchouk polynomials. The set of proposed Krawtchouk moments is then derived from the weighted Krawtchouk polynomials. The orthogonality of the proposed moments ensures minimal information redundancy. No numerical approximation is involved in deriving the moments, since the weighted Krawtchouk polynomials are discrete. These properties make the Krawtchouk moments well suited as pattern features in the analysis of two-dimensional images. It is shown that the Krawtchouk moments can be employed to extract local features of an image, unlike other orthogonal moments, which generally capture the global features. The computational aspects of the moments using the recursive and symmetry properties are discussed. Krawtchouk moment invariants are constructed using a linear combination of geometric moment invariants; an object recognition experiment

shows Krawtchouk moment invariants perform significantly better than Hu's moment invariants in both noise-free and noisy conditions.

3.3 MOMENT INVARIANTS

1. Invariants to translation:

Invariance to translation can be achieved simply by seemingly shifting the object such that its centroid coincides with the origin of the coordinate system or, vice versa, by shifting the polynomial basis into the object centroid.

2. Invariants to uniform scaling:

Scaling invariance is obtained by proper normalization of each moment. In principle, any moment can be used as a normalizing factor provided that it is nonzero for all images in the experiment.

3. Invariants to rotation (Hu):

The low-order moments describe the overall characteristics of the image, the zero-order moment reflects the target area, the first order moment reflects the target center of mass, second moment reflects the length of principal, auxiliary axis and the orientation angle of principal axis, higher moments describe the details of the image.

Although the Hu invariants suffer from limited recognition power, mutual dependence and restriction to second- and third-order moments only, they have become classics and, inspite of their drawbacks, they have found numerous successful applications in various areas. The major weakness of Hu's theory is that it does not provide for the possibility of any generalization. Using it, we could not derive invariants from higher-order moments and invariants to more general transformations.

3.4 KRAWTCHOUK MOMENTS

Moments due to its ability to represent global features have found extensive applications in the field of image processing. As mentioned before, the continuous orthogonal polynomials face the problem of discretization error. To overcome that problem, a new set of moments called Tchebichef. Study showed that the implementation of **Tchebichef moments** does not involve any numerical approximation since the basis set is orthogonal in the discrete domain of the image coordinate space. This property makes Tchebichef moments superior to the conventional continuous orthogonal moments in terms of preserving the analytical property needed to ensure information redundancy in a moment set.

On the similar lines, Krawtchouk Moments [5], based on Krawtchouk polynomials are found to have discrete orthogonal property. Similar to Tchebichef moments, there is no need for spatial normalization; hence, the error in the computed Krawtchouk moments due to discretization is nonexistent. Unlike the above-mentioned moments, Krawtchouk moments have the ability of being able to extract local features from any region-of-interest in an image. This can be accomplished by varying parameter of the binomial distribution associated with the Krawtchouk polynomials.

The Krawtchouk polynomials are scaled to ensure that all the computed moments have equal weights. A three-term recurrence relation is used to avoid the direct computation of the *hypergeometric* and gamma functions, which tends to be numerically unstable as the order of the moments increases. The symmetry property of the Krawtchouk polynomials is utilized to reduce the number of the computation points of the Krawtchouk polynomials by half and correspondingly reduce the computation time to determine the moments.

For the purpose of object recognition, it is vital that Krawtchouk moments be independent of rotation, scaling, and translation of the image. It is shown that Krawtchouk moment invariants can be formed by a linear combination of geometric moment invariants. The recognition accuracy of Krawtchouk moment invariants is tested with Hu moment invariants.

3.4.1 Krawtchouk *Polynomials*

The definition of n-th order classical Krawtchouk polynomial is defined as:

$$K_n(x; p, N) = \sum_{k=0}^N a_{k,n,p} x^k = {}_2F_1 \left(-n, -x; -N; \frac{1}{p} \right) \quad \dots(1)$$

where $x, n = 0, 1, 2, \dots, N, N > 0, p \in (0,1)$. ${}_2F_1$ is the *hypergeometric function* defined as:

$${}_2F_1(a, b, c; z) = \sum_{k=0}^{\infty} \frac{(a)_k (b)_k}{(c)_k} \frac{z^k}{k!} \quad \dots(2)$$

and $(a)_k$ is the *Pochhammer symbol* given by:

$$(a)_k = a(a+1) \dots (a+k-1) = \frac{\Gamma(a+k)}{\Gamma(a)} \quad \dots(3)$$

The set of $(N+1)$ Krawtchouk polynomials $\{K_n(x; p, N)\}$ forms a complete set of discrete basis functions with weight function:

$$w(x; p, N) = \binom{N}{x} p^x (1-p)^{N-x} \quad \dots(4)$$

and satisfies the orthogonality condition

$$\sum_{x=0}^N w(x; p, N) K_n(x; p, N) K_m(x; p, N) = \rho(n; p, N) \delta_{nm} \quad \dots(5)$$

Where $n, m = 1, 2, \dots, N$ and

$$\rho(n; p, N) = (-1)^n \left(\frac{1-p}{p} \right)^n \frac{n!}{(-N)_n}$$

Examples of Krawtchouk polynomials up to the second order are:

$$\begin{aligned} K_0(x; p, N) &= 1 \\ K_1(x; p, N) &= 1 - \left[\frac{1}{Np} \right] x \\ K_2(x; p, N) &= 1 - \left[\frac{2}{Np} + \frac{1}{N(N-1)p^2} \right] x + \left[\frac{1}{N(N-1)p^2} \right] x^2 \end{aligned}$$

3.4.2 Krawtchouk Moments

Krawtchouk moments have the interesting property of being able to extract local features of an image. The Krawtchouk moments of order $(n+m)$ in terms of weighted Krawtchouk polynomials, for an image with intensity function, $f(x, y)$, is defined as

$$Q_{nm} = \sum_{x=0}^{N-1} \sum_{y=0}^{M-1} K_n(x; p_1, N-1) K_m(y; p_2, M-1) f(x, y) \quad \dots(6)$$

The parameters N and M are substituted with $N-1$ and $M-1$ respectively to match the $N \times M$ pixel points of an image. The Krawtchouk moment corresponding to $n = m = 0$ is the weighted mass of the image, i.e.,

$$Q_{00} = \sum_{x=0}^{N-1} \sum_{y=0}^{M-1} \sqrt{w(x; p_1, N-1) w(y; p_2, M-1)} f(x, y) \quad \dots(7)$$

Hence the image intensity function can be written completely in terms of the Krawtchouk moments, i.e.,

$$f(x, y) = \sum_{n=0}^{N-1} \sum_{m=0}^{M-1} Q_{nm} K_n(x; p_1, N-1) \times K_m(y; p_2, M-1) \quad \dots(8)$$

One way of interpreting the above equation is that the image intensity function can be represented as a series of Krawtchouk polynomials weighted by the Krawtchouk moments. Hence the sum total of moments obtained is used to define on feature of the feature vector.

3.4.3 Hypergeometric Function

The hypergeometric function is defined for $|z| < 1$ by the power series

$${}_2F_1(a, b, c; z) = \sum_{k=0}^{\infty} \frac{(a)_k (b)_k}{(c)_k} \frac{z^k}{k!} \quad \dots(2)$$

The series terminates if either a or b is a non-positive integer, in which case the function reduces to a polynomial:

$${}_2F_1(-m, b, c; z) = \sum_{n=0}^m (-1)^n \binom{m}{n} \frac{(b)_n}{(c)_n} z^n$$

For complex arguments z with $|z| \geq 1$ it can be analytically continued along any path in the complex plane that avoids the branch points 0 and 1.

As $c \rightarrow -m$, where m is a positive integer, ${}_2F_1(z) \rightarrow \infty$, but if we divide by $\Gamma(c)$, we have a limit:

$$\lim_{c \rightarrow -m} \frac{{}_2F_1(-m, b, c; z)}{\Gamma(c)} = \frac{(a)_{m+1} (b)_{m+1}}{(m+1)!} z^{m+1} {}_2F_1(-m, b, c; z) \quad \dots(9)$$

CLASSIFICATION

Classification mainly consists of finding the best matching reference features for the features extracted in the previous phase. Classification is done using the minimum distance classifier.

4.1 Knn classifier algorithm

In pattern recognition, the k-Nearest Neighbours algorithm (or k-NN for short) is a non-parametric method used for classification and regression. In both cases, the input consists of the k closest training examples in the feature space. The output depends on whether k-NN is used for classification or regression:

In k-NN classification, the output is a class membership. An object is classified by a majority vote of its neighbours, with the object being assigned to the class most common among its k nearest neighbours (k is a positive integer, typically small). If $k = 1$, then the object is simply assigned to the class of that single nearest neighbour.

In k-NN regression, the output is the property value for the object. This value is the average of the values of its k nearest neighbours.

k-NN is a type of instance-based learning, or lazy learning, where the function is only approximated locally and all computation is deferred until classification. The k-NN algorithm is among the simplest of all machine learning algorithms.

Both for classification and regression, it can be useful to assign weight to the contributions of the neighbours, so that the nearer neighbours contribute more to the average than the more distant ones. For example, a common weighting scheme consists in giving each neighbour a weight of $1/d$, where d is the distance to the neighbour.

The neighbours are taken from a set of objects for which the class (for k-NN classification) or the object property value (for k-NN regression) is known. This can be thought of as the training set for the algorithm, though no explicit training step is required.

A shortcoming of the k-NN algorithm is that it is sensitive to the local structure of the data.

4.2 Application of knn on current data

4.2.1 Population of the Database

The following steps are taken to populate the database:

1. Input image
2. Find the Krawtchouk moments of the order 14, 16...,20
3. Create a vector based on the above moments.
4. Repeat step 1 - 3 for each of the image in the directory. A $n \times m$ matrix hence obtained contains n set of images and $m = 4$ feature vector.
5. A label vector of length n is also populated to classify each of the image into its numerical form.

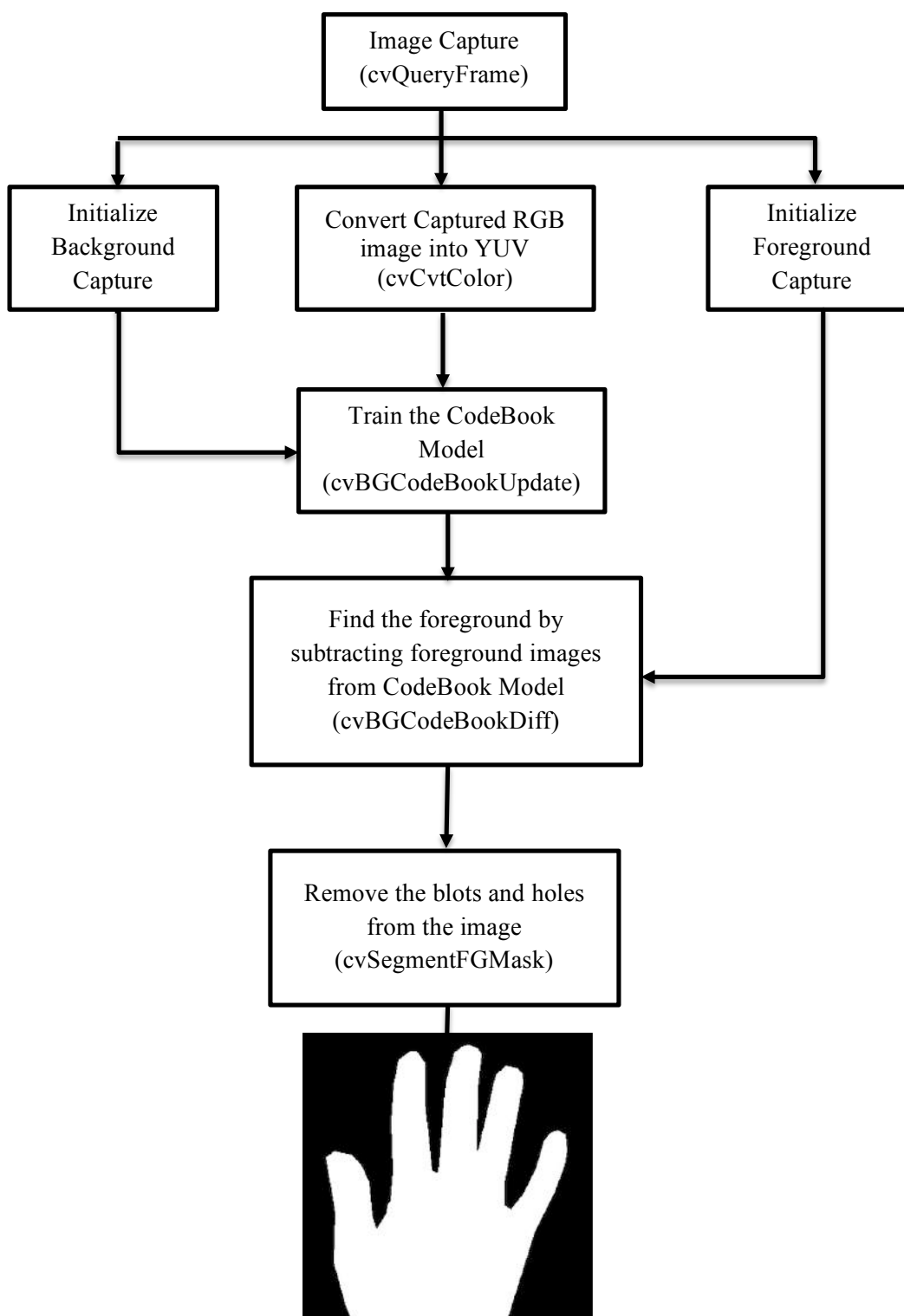
4.2.2 Testing Knn Scheme

To test the knn scheme:

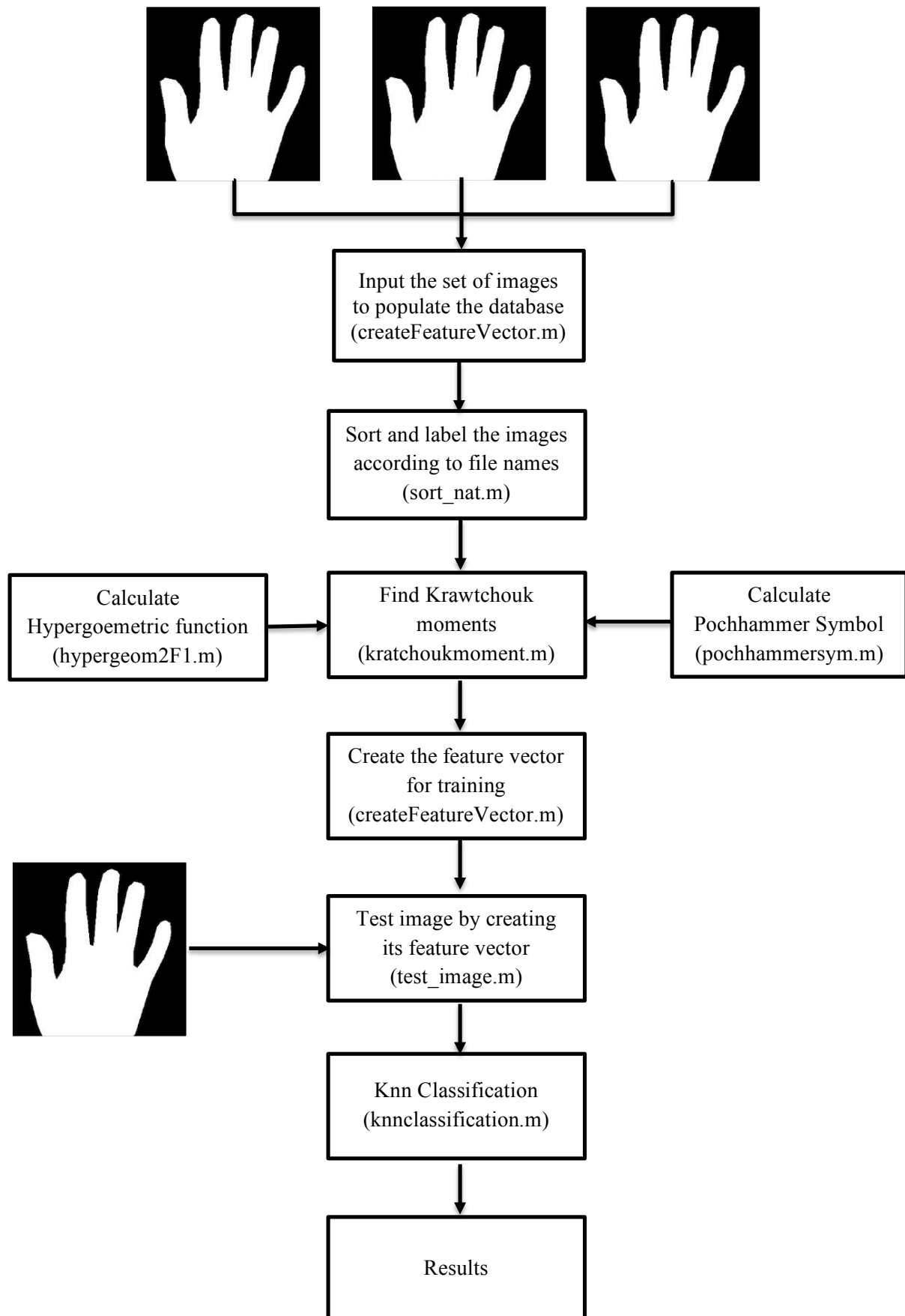
1. Input the image
2. Find the moments and create a vector based on steps 2 and 3 above.
3. Now the *feature vector* and *label vector* obtained above it test against the vector in step 3, giving the value of the number of nearest neighbours, k . In this case $k = 1$.

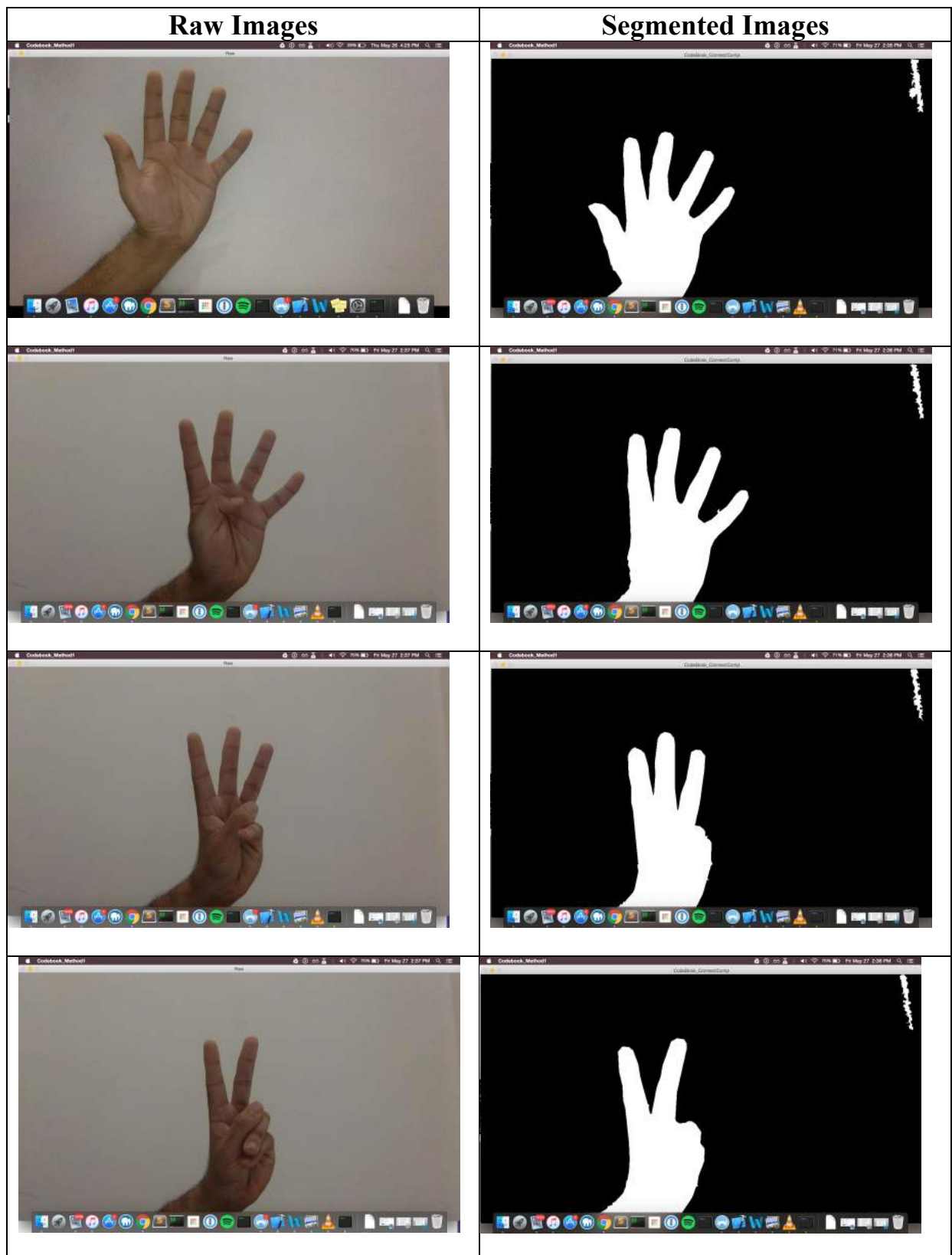
EXPERIMENTAL SETUP

(Hand Detection and Segmentation)



(Training and Classification)







RESULTS

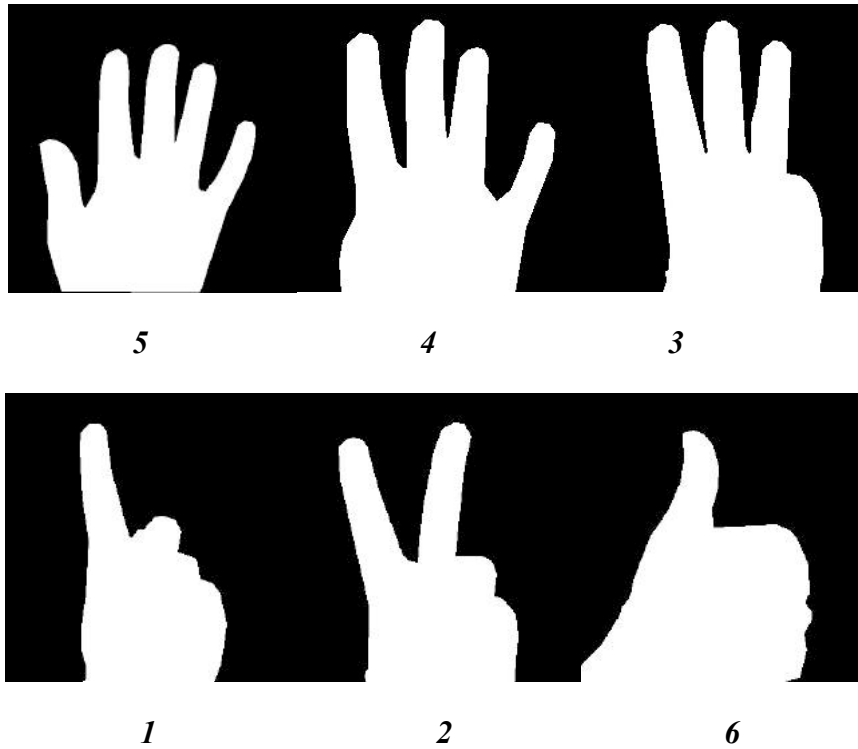


Table 1: Krawtchouk moments results without transformation (Number of training samples in Dataset 300)

No of testing samples	Correct classification (CC)	Percentage CC
100	92	92%

Table 2: Krawtchouk moments results with transformation (Number of testing samples in Dataset 400)

No of testing samples	Correct classification (CC)	Percentage CC
70	62	88.57%

Table 3: Comparison of time using different forms of hypergeometric functions

Hypergeometric function	No of testing samples	Runtime(seconds)
Recursive method	36	62.98
Iterative method	2	890.87

REFERENCES

- [1] Ong and Ranganath, "Automatic Sign Language Analysis: A Survey and the Future Beyond Lexical Meaning," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 27, no. 6, pp. 873–891, June 2005.
- [2] A. Chalechale, F.Safaei, G.Naghdy, and P.Premaratne, "Hand Posture Analysis for Visual based Human Machine Interface," in *Proceedings of the Workshop on Digital image Computing*, 2005, pp. 91–96.
- [3] C.-C. Chang, J. J. Chen, W.-K. Tai, and C.-C. Han, "New Approach for Static Gesture Recognition," *Journal of Information Science and Engineering*, vol. 22, pp. 1047–1057, January 2006.
- [4] L. Gu and J. Su, "Natural Hand Posture Classification based on Zernike Moments and Hierarchial Classifier," in *Proceedings of the Internatinal Conference on Robotics and Automaton. IEEE*, May 2008, pp. 3088–3093.
- [5] P.T.Yap, P.Raveendran and S.H.Ong, "Image Analysis by Krawtchouk Moments", *IEEE Transaction on Image Processing*, Vol. 12, No II, pp1367-1377, November 2003.
- [6] P.T.Yap, P.Raveendran and S.H.Ong, "Krawtchouk Moments as a New Set of Discrete Orthogonal Moments for Image Reconstruction", *Proceeding of the 2002 International Joint Conference on Neural Networks*, VoU, pp 908-912, 12-17 th May 2002.
- [7] P.T.Yap, P. Ravendran, S.H.Ong, "Chebychev Moments as a New Set of Moments for Image Reconstruction", *Proceeding of IEEE*, pp2856-2860, 200 I.
- [8] S. Padam Priyal and Prabin. K. Bora, "A study on static hand gesture recognition using moments", *Conference on Signal Processing*, 18-21 July 2010
- [9] Gary Bradski, Adrian Kaehler, (September 2008) "Learning OpenCV", OReilly Media, pp 281-285.
- [10] A. J. Nor'aini, "A comparative study of face recognition using discrete orthogonal moments" *10th International Conference on Information Science, ISSPA*, 10-13 May 2010

APPENDIX

1. OpenCV

Open Source Computer Vision Library (OpenCV) is an open source computer vision library. The library is written in C and C++ and runs under Linux, Windows and Mac OS X. There is active development on interfaces for Python, Ruby, Matlab, and other languages.

OpenCV was designed for computational efficiency and with a strong focus on realtime applications. OpenCV is written in optimized C and can take advantage of multicore processors. If you desire further automatic optimization on Intel architectures [Intel], you can buy Intel's Integrated Performance Primitives (IPP) libraries [IPP], which consist of low-level optimized routines in many different algorithmic areas. OpenCV automatically uses the appropriate IPP library at runtime if that library is installed.

One of OpenCV's goals is to provide a simple-to-use computer vision infrastructure that helps people build fairly sophisticated vision applications quickly. The OpenCV library contains over 500 functions that span many areas in vision, including factory product inspection, medical imaging, security, user interface, camera calibration, stereo vision, and robotics. Because computer vision and machine learning often go hand-in-hand, OpenCV also contains a full, general-purpose Machine Learning Library (MLL).

This sublibrary is focused on statistical pattern recognition and clustering. The MLL is highly useful for the vision tasks that are at the core of OpenCV's mission, but it is general enough to be used for any machine learning problem.

Applications

- 2D and 3D feature toolkits
- Egomotion estimation
- Facial recognition system
- Gesture recognition
- Human–computer interaction (HCI)
- Mobile robotics
- Motion understanding
- Object identification
- Segmentation and recognition
- Stereopsis stereo vision: depth perception from 2 cameras
- Structure from motion (SFM)
- Motion tracking
- Augmented reality

2. Computer Vision

Computer vision is the transformation of data from a still or video camera into either a decision or a new representation. All such transformations are done for achieving some particular goal. The input data may include some contextual information such as “the camera is mounted in a car” or “laser range fi nder indicates an object is 1 meter away”. The decision might be “there is a person in this scene” or “there are 14 tumor cells on this slide”.

A new representation might mean turning a color image into a grayscale image or removing camera motion from an image sequence.

The next problem facing computer vision is noise. We typically deal with noise by using statistical methods.

For example, it may be impossible to detect an edge in an image merely by comparing a point to its immediate neighbours. But if we look at the statistics over a local region, edge detection becomes much easier. A real edge should appear as a string of such immediate neighbor responses over a local region, each of whose orientation is consistent with its neighbors.

It is also possible to compensate for noise by taking statistics over time. Still other techniques account for noise or distortions by building explicit models learned directly from the available data.

The actions or decisions that computer vision attempts to make based on camera data are performed in the context of a specific purpose or task. We may want to remove noise or damage from an image so that our security system will issue an alert if someone tries to climb a fence or because we need a monitoring system that counts how many people cross through an area in an amusement park. Vision software for robots that wander through office buildings will employ different strategies than vision software for stationary security cameras because the two systems have significantly different contexts and objectives.

As a general rule: the more constrained a computer vision context is, the more we can rely on those constraints to simplify the problem and the more reliable our final solution will be.

OpenCV is aimed at providing the basic tools needed to solve computer vision problems. In some cases, high-level functionalities in the library will be sufficient to solve the more complex problems in computer vision. Even when this is not the case, the basic components in the library are complete enough to enable creation of a complete solution of your own to almost any computer vision problem.

In the latter case, there are several tried-and-true methods of using the library; all of them start with solving the problem using as many available library components as possible. Typically, after you’ve developed this first-draft solution, you can see where the solution has weaknesses and then fix those weaknesses using your own code and cleverness. You can then use your draft solution as a benchmark to assess the improvements you have made. From that point, whatever weaknesses remain can be tackled by exploiting the context of the larger system in which our problem solution is embedded.

CODE

1. Codebook Method using Background Subtraction

```
#include <iostream>
#include <opencv/cv.h>
#include "opencv/cv_aux.h"
#include "opencv/cxmisc.h"
#include "opencv2/highgui/highgui.hpp"
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <opencv/highgui.h>

using namespace std;
using namespace cv;
//VARIABLES for CODEBOOK METHOD:
CvBGCodeBookModel* model = 0;
const int NCHANNELS = 3;
bool ch[NCHANNELS]={true,true,true}; // This sets what channels should be adjusted for
background bounds

void detect(IplImage* img_8uc1,IplImage* img_8uc3);

void help(void)
{
    printf("\nLearn background and find foreground using simple average and average
difference learning method:\n"
        "\nUSAGE:\nbgfg_codebook [--nframes=300] [movie filename, else from camera]\n"
        "***Keep the focus on the video windows, NOT the consol***\n\n"
        "INTERACTIVE PARAMETERS:\n"
        "\tESC,q,Q - quit the program\n"
        "\th      - print this help\n"
        "\tp      - pause toggle\n"
        "\ts      - single step\n"
        "\tr      - run mode (single step off)\n"
        "=== AVG PARAMS ===\n"
        "\t-      - bump high threshold UP by 0.25\n"
        "\t=      - bump high threshold DOWN by 0.25\n"
        "\t[      - bump low threshold UP by 0.25\n"
        "\t]      - bump low threshold DOWN by 0.25\n"
        "=== CODEBOOK PARAMS ===\n"
        "\ty,u,v- only adjust channel 0(y) or 1(u) or 2(v) respectively\n"
        "\ta      - adjust all 3 channels at once\n"
        "\tb      - adjust both 2 and 3 at once\n"
        "\ti,o    - bump upper threshold up,down by 1\n"
        "\tk,l    - bump lower threshold up,down by 1\n"
        "\tSPACE - reset the model\n"
    );
}
```

```

//
//USAGE: ch9_background startFrameCollection# endFrameCollection# [movie filename,
else from camera]
//If from AVI, then optionally add HighAvg, LowAvg, HighCB_Y LowCB_Y HighCB_U
LowCB_U HighCB_V LowCB_V
//

String inttostr(int input)
{
    stringstream ss;
    ss << input;
    return ss.str();
}

int main(int argc, char** argv)
{
    int photocount = 0; //initialize image counter
    int p[3];
    //specify the compression technique
    p[0] = CV_IMWRITE_JPEG_QUALITY;
    //specify the compression quality
    p[1] = 100;
    p[2] = 0;

    const char* filename = 0;
    IplImage* rawImage = 0, *yuvImage = 0; //yuvImage is for codebook method
    IplImage *ImaskCodeBook = 0, *ImaskCodeBookCC = 0, *showImageCB = 0;
    CvCapture* capture = 0;

    int c, n, nframes = 0;
    int nframesToLearnBG = 300;

    model = cvCreateBGCodeBookModel();

    //Set color thresholds to default values
    model->modMin[0] = 3;
    model->modMin[1] = model->modMin[2] = 3;
    model->modMax[0] = 10;
    model->modMax[1] = model->modMax[2] = 10;
    model->cbBounds[0] = model->cbBounds[1] = model->cbBounds[2] = 10;

    bool pause = false;
    bool singlestep = false;

    for( n = 1; n < argc; n++ )
    {
        static const char* nframesOpt = "--nframes=";
        if( strcmp(argv[n], nframesOpt, strlen(nframesOpt))==0 )
        {

```

```

        if( sscanf(argv[n] + strlen(nframesOpt), "%d", &nframesToLearnBG) == 0 )
        {
            help();
            return -1;
        }
    }
    else
        filename = argv[n];
}

if( !filename )
{
    printf("Capture from camera\n");
    capture = cvCaptureFromCAM( 0 );
}
else
{
    printf("Capture from file %s\n",filename);
    capture = cvCreateFileCapture( filename );
}

if( !capture )
{
    printf( "Can not initialize video capturing\n\n" );
    help();
    return -1;
}

//MAIN PROCESSING LOOP:
for(;;)
{
    if( !pause )
    {
        rawImage = cvQueryFrame( capture );
        ++nframes;
        if(!rawImage)
            break;
    }
    if( singlestep )
        pause = true;

    //First time:
    if( nframes == 1 && rawImage )
    {
        // CODEBOOK METHOD ALLOCATION
        yuvImage = cvCloneImage(rawImage);
        ImaskCodeBook = cvCreateImage( cvGetSize(rawImage), IPL_DEPTH_8U, 1 );
        ImaskCodeBookCC = cvCreateImage( cvGetSize(rawImage), IPL_DEPTH_8U, 1 );
        showImageCB = cvCreateImage(cvGetSize(rawImage), IPL_DEPTH_8U, 1);
    }
}

```

```

cvSet(ImaskCodeBook,cvScalar(255));

cvNamedWindow( "Raw", 1 );
cvNamedWindow( "ForegroundCodeBook",1);
cvNamedWindow( "CodeBook_ConnectComp",1);
}

// If we've got an rawImage and are good to go:
if( rawImage )
{
    cvCvtColor( rawImage, yuvImage, CV_BGR2YCrCb );//YUV For codebook method
    //This is where we build our background model
    if( !pause && nframes-1 < nframesToLearnBG )
        cvBGCodeBookUpdate( model, yuvImage );

    if( nframes-1 == nframesToLearnBG )
        cvBGCodeBookClearStale( model, model->t/2 );

    //Find the foreground if any
    if( nframes-1 >= nframesToLearnBG )
    {
        // Find foreground by codebook method
        cvBGCodeBookDiff( model, yuvImage, ImaskCodeBook );
        // This part just to visualize bounding boxes and centers if desired
        cvCopy(ImaskCodeBook,ImaskCodeBookCC);
        cvSegmentFGMask( ImaskCodeBookCC );
        //bwareaopen_(ImaskCodeBookCC,100);
        cvCopy(ImaskCodeBookCC, showImageCB);
        cvShowImage( "CodeBook_ConnectComp",ImaskCodeBookCC);
        detect(ImaskCodeBookCC,rawImage);

        photocount++;// increment image number for each capture
        //cvShowImage("Captured", showImageCB);

        String imagename = "/Users/sartaj10/Desktop/Images/" + inttostr(photocount) +
".jpg";
        const char * c = imagename.c_str();
        cvSaveImage(c, showImageCB, p);

    }
    //Display
    cvShowImage( "Raw", rawImage );
    cvShowImage( "ForegroundCodeBook",ImaskCodeBook);
}

// User input:
c = cvWaitKey(10)&0xFF;
c = tolower(c);
// End processing on ESC, q or Q

```



```

if(c == 27 || c == 'q')
    break;

switch( c )
{
    case 'h':
        help();
        break;
    case 'p':
        pause = !pause;
        break;
    case 's':
        singlestep = !singlestep;
        pause = false;
        break;
    case 'r':
        pause = false;
        singlestep = false;
        break;
    case ' ':
        cvBGCodeBookClearStale( model, 0 );
        nframes = 0;
        break;
        //CODEBOOK PARAMS
    case 'y': case '0':
    case 'u': case '1':
    case 'v': case '2':
    case 'a': case '3':
    case 'b':
        ch[0] = c == 'y' || c == '0' || c == 'a' || c == '3';
        ch[1] = c == 'u' || c == '1' || c == 'a' || c == '3' || c == 'b';
        ch[2] = c == 'v' || c == '2' || c == 'a' || c == '3' || c == 'b';
        printf("CodeBook YUV Channels active: %d, %d, %d\n", ch[0], ch[1], ch[2] );
        break;
    case 'i': //modify max classification bounds (max bound goes higher)
    case 'o': //modify max classification bounds (max bound goes lower)
    case 'k': //modify min classification bounds (min bound goes lower)
    case 'l': //modify min classification bounds (min bound goes higher)
    {
        uchar* ptr = c == 'i' || c == 'o' ? model->modMax : model->modMin;
        for(n=0; n<NCHANNELS; n++)
        {
            if( ch[n] )
            {
                int v = ptr[n] + (c == 'i' || c == 'l' ? 1 : -1);
                ptr[n] = CV_CAST_8U(v);
            }
            printf("%d,", ptr[n]);
        }
        printf(" CodeBook %s Side\n", c == 'i' || c == 'o' ? "High" : "Low" );
    }
}

```

```

        }
        break;
    }
}

cvReleaseCapture( &capture );
cvDestroyWindow( "Raw" );
cvDestroyWindow( "ForegroundCodeBook");
cvDestroyWindow( "CodeBook_ConnectComp");
return 0;
}

void detect(IplImage* img_8uc1,IplImage* img_8uc3) {

    //cvNamedWindow( "aug", 1 );

    //cvThreshold( img_8uc1, img_edge, 128, 255, CV_THRESH_BINARY );
    CvMemStorage* storage = cvCreateMemStorage();
    CvSeq* first_contour = NULL;
    CvSeq* maxitem=NULL;
    double area=0,areamax=0;
    int maxn=0;
    int Nc = cvFindContours(
        img_8uc1,
        storage,
        &first_contour,
        sizeof(CvContour),
        CV_RETR_LIST // Try all four values and see what happens
    );

    int n=0;
    //printf( "Total Contours Detected: %d\n", Nc );

    if(Nc>0)
    {
        for( CvSeq* c=first_contour; c!=NULL; c=c->h_next )
        {

            //cvCvtColor( img_8uc1, img_8uc3, CV_GRAY2BGR );

            area=cvContourArea(c,CV_WHOLE_SEQ );

            if(area>areamax)
            {
                areamax=area;
                maxitem=c;
                maxn=n;
            }
        }
    }
}

```

```

n++;

}
CvMemStorage* storage3 = cvCreateMemStorage(0);
//if (maxitem) maxitem = cvApproxPoly( maxitem, sizeof(maxitem), storage3,
CV_POLY_APPROX_DP, 3, 1 );

if(areamax>5000)
{
    maxitem = cvApproxPoly( maxitem, sizeof(CvContour), storage3,
CV_POLY_APPROX_DP, 10, 1 );

    CvPoint pt0;

    CvMemStorage* storage1 = cvCreateMemStorage(0);
    CvMemStorage* storage2 = cvCreateMemStorage(0);
    CvSeq* ptseq = cvCreateSeq( CV_SEQ_KIND_GENERIC|CV_32SC2,
sizeof(CvContour),
                                sizeof(CvPoint), storage1 );
    CvSeq* hull;
    CvSeq* defects;

    for(int i = 0; i < maxitem->total; i++ )
    {   CvPoint* p = CV_GET_SEQ_ELEM( CvPoint, maxitem, i );
        pt0.x = p->x;
        pt0.y = p->y;
        cvSeqPush( ptseq, &pt0 );
    }
    hull = cvConvexHull2( ptseq, 0, CV_CLOCKWISE, 0 );
    int hullcount = hull->total;

    defects= cvConvexityDefects(ptseq,hull,storage2 );

    //printf(" defect no %d \n",defects->total);

    CvConvexityDefect* defectArray;

    int j=0;
    //int m_nomdef=0;
    // This cycle marks all defects of convexity of current contours.
    for(;defects;defects = defects->h_next)
    {

```

```

int nomdef = defects->total; // defect amount
//outlet_float( m_nomdef, nomdef );

//printf(" defect no %d \n",nomdef);

if(nomdef == 0)
    continue;

// Alloc memory for defect set.
//fprintf(stderr,"malloc\n");
defectArray = (CvConvexityDefect*)malloc(sizeof(CvConvexityDefect)*nomdef);

// Get defect set.
//fprintf(stderr,"cvCvtSeqToArray\n");
cvCvtSeqToArray(defects,defectArray, CV_WHOLE_SEQ);

// Draw marks for all defects.
for(int i=0; i<nomdef; i++)
{ printf(" defect depth for defect %d %f \n",i,defectArray[i].depth);
  cvLine(img_8uc3, *(defectArray[i].start),
*(defectArray[i].depth_point),CV_RGB(255,255,0),1, CV_AA, 0 );
  cvCircle( img_8uc3, *(defectArray[i].depth_point), 5, CV_RGB(0,0,164), 2,
8,0);
  cvCircle( img_8uc3, *(defectArray[i].start), 5, CV_RGB(0,0,164), 2, 8,0);
  cvLine(img_8uc3, *(defectArray[i].depth_point),
*(defectArray[i].end),CV_RGB(255,255,0),1, CV_AA, 0 );

}
char txt[]="0";
txt[0]='0'+nomdef-1;
CvFont font;
cvInitFont(&font, CV_FONT_HERSHEY_SIMPLEX, 1.0, 1.0, 0, 5, CV_AA);
cvPutText(img_8uc3, txt, cvPoint(50, 50), &font, cvScalar(0, 0, 255, 0));

j++;

// Free memory.
free(defectArray);
}

cvReleaseMemStorage( &storage );
cvReleaseMemStorage( &storage1 );
cvReleaseMemStorage( &storage2 );
cvReleaseMemStorage( &storage3 );
//return 0;
}
}
}

```

2. Kratchouk Moments

```

function T = kratchoukmoment(g,h,p,img)
[nx,ny,~]=size(img);
img = rgb2gray(img);
img=im2double(img);

if nx ~= ny
    fprintf('img is not square');
end
for x = 0 : nx - 1
    for n = 0 : g-1
        % w = nchoosek(nx-1,x)*p^x*(1-p)^(nx-x-1);
        rho(n+1) = (-1)^n*(1/p-1)^n*(factorial(n)/pochhammersym(-nx+1,n));
        A(n+1,x+1) = sqrt(1/rho(n+1))*hypergeom2F1(-n,-x,-(nx-1),1/p,0.01);
    end
end
for y = 0 : ny - 1
    for n = 0 : h-1
        % w = nchoosek(ny-1,y)*p^y*(1-p)^(ny-y-1);
        rho(n+1) = (-1)^n*(1/p-1)^n*(factorial(n)/pochhammersym(-ny+1,n));
        B(n+1,y+1) = sqrt(1/rho(n+1))*hypergeom2F1(-n,-y,-(ny-1),1/p,0.01);
    end
end
for i = 0 : g-1
    for j = 0 : h-1
        px = A(i + 1,:);
        py = B(j + 1,:);
        Q = px' * py;
        temp = (img .* Q);
        T(i + 1,j + 1) = sum(temp(:));
    end
end
s=abs(sum(T(:)));

```

3. Pochhammersym Function

```

function o = pochhammersym(a,k)
if k<0
    print (' formula is different for -ve k value');
end
h=a;
for j = 0 : k-2
    a(j+2)=a(j+1)*(h+j+1);
end
o=a(end);

```

4. Hypergeometric Function

```

function h=hypergeom2F1(a,b,c,z,tol)
%HYPERGEOM2F1 Gaussian or ordinary hypergeometric function for ABS(Z) < 1

```

```

% H = HYPERGEOM2F1(A,B,C,Z) returns the hypergeometric function 2F1 for scalar
% parameters A, B, C, and complex inputs Z. A finite number of terms from the
% infinite summation representation of the function are used until the
% absolute tolerance EPS is met.
%
% H = HYPERGEOM2F1(...,TOL) specifies the absolute tolerance used to terminate
% the summation of terms.
%
% Note:
% Unless, C = A or C = B, if C is an integer <= 0, NaN is returned.
% Additionally, the simple method of computation used can be very
% inaccurate when ABS(Z) is close to 1 for some parameter combinations.
%
% See also HYPERGEOM.

% Check four required inputs
if isempty(a) || ~isscalar(a) || ~isfloat(a) || ~isreal(a) || ~isfinite(a)
    error('hypergeom2F1:AInvalid',...
        'A must be a non-empty finite real floating-point scalar.');
```

end

```

if isempty(b) || ~isscalar(b) || ~isfloat(b) || ~isreal(b) || ~isfinite(b)
    error('hypergeom2F1:BInvalid',...
        'B must be a non-empty finite real floating-point scalar.');
```

end

```

if isempty(c) || ~isscalar(c) || ~isfloat(c) || ~isreal(c) || ~isfinite(c)
    error('hypergeom2F1:CInvalid',...
        'C must be a non-empty finite real floating-point scalar.');
```

end

```

if ~isfloat(z) || ~all(isfinite(z))
    error('hypergeom2F1:ZInvalid','Z must be a finite floating-point array.');
```

end

```

if any(abs(z)>=1)
    error('hypergeom2F1:ZUndefined',...
        ['The standard Gaussian hypergeometric function is only defined ' ...
        'for ABS(Z) < 1']);
```

end

```

% Calculate Gaussian hypergeometric function via summation of terms
if isempty(z)
    h = z;
else
    % Set relative tolerance and maximum number of iterations
    dtype = superiorfloat(a,b,c,z);
    if nargin < 5
        tol = eps(dtype);
    end
    itermax = 2^15;

    h = zeros(size(z),dtype);
    for j = 1:numel(z)
```

```

Z = z(j);

if a == 1
    if b == c
        yi = Z;
        y = 1+yi;
        for i = 1:itermax
            yi = yi*Z;
            y = y+yi;
            if abs(yi) < tol
                break;
            end
        end
    else
        yi = b*Z/c;
        y = 1+yi;
        for i = 1:itermax
            yi = yi*(b+i)*Z/(c+i);
            y = y+yi;
            if abs(yi) < tol
                break;
            end
        end
    end
elseif b == 1
    if a == c
        yi = Z;
        y = 1+yi;
        for i = 1:itermax
            yi = yi*Z;
            y = y+yi;
            if abs(yi) < tol
                break;
            end
        end
    else
        yi = a*Z/c;
        y = 1+yi;
        for i = 1:itermax
            yi = yi*(a+i)*Z/(c+i);
            y = y+yi;
            if abs(yi) < tol
                break;
            end
        end
    end
elseif a == c
    yi = b*Z;
    y = 1+yi;
    for i = 1:itermax

```

```

        yi = yi*(b+i)*Z/(i+1);
        y = y+yi;
        if abs(yi) < tol
            break;
        end
    end
elseif b == c
    yi = a*Z;
    y = 1+yi;
    for i = 1:itermax
        yi = yi*(a+i)*Z/(i+1);
        y = y+yi;
        if abs(yi) < tol
            break;
        end
    end
else
    yi = a*b*Z/c;
    y = 1+yi;
    for i = 1:itermax
        yi = yi*(a+i)*(b+i)*Z/((i+1)*(c+i));
        y = y+yi;
        if abs(yi) < tol
            break;
        end
    end
end
h(j) = y;
end
end

```

5. Sorting Function

```

function [cs,index] = sort_nat(c,mode)
%sort_nat: Natural order sort of cell array of strings.
% usage: [S,INDEX] = sort_nat(C)
%
% where,
%   C is a cell array (vector) of strings to be sorted.
%   S is C, sorted in natural order.
%   INDEX is the sort order such that S = C(INDEX);
%
% Natural order sorting sorts strings containing digits in a way such that
% the numerical value of the digits is taken into account. It is
% especially useful for sorting file names containing index numbers with
% different numbers of digits. Often, people will use leading zeros to get

```



```

% the right sort order, but with this function you don't have to do that.
% For example, if C = {'file1.txt','file2.txt','file10.txt'}, a normal sort
% will give you
%
%      {'file1.txt' 'file10.txt' 'file2.txt'}
%
% whereas, sort_nat will give you
%
%      {'file1.txt' 'file2.txt' 'file10.txt'}
%
% See also: sort

% Set default value for mode if necessary.
if nargin < 2
    mode = 'ascend';
end

% Make sure mode is either 'ascend' or 'descend'.
modes = strcmpi(mode,{'ascend','descend'});
is_descend = modes(2);
if ~any(modes)
    error('sort_nat:sortDirection',...
        'sorting direction must be "ascend" or "descend".')
end

% Replace runs of digits with '0'.
c2 = regexp(c,'\d+','0');

% Compute char version of c2 and locations of zeros.
s1 = char(c2);
z = s1 == '0';

% Extract the runs of digits and their start and end indices.
[digruns,first,last] = regexp(c,'\d+','match','start','end');

% Create matrix of numerical values of runs of digits and a matrix of the
% number of digits in each run.
num_str = length(c);
max_len = size(s1,2);
num_val = NaN(num_str,max_len);
num_dig = NaN(num_str,max_len);
for i = 1:num_str
    num_val(i,z(i,:)) = sscanf(sprintf('%s ',digruns{i}{'{:}'}),'%f');
    num_dig(i,z(i,:)) = last{i} - first{i} + 1;
end

% Find columns that have at least one non-NaN. Make sure activecols is a
% 1-by-n vector even if n = 0.
activecols = reshape(find(~all(isnan(num_val))),1,[]);
n = length(activecols);

```

```

% Compute which columns in the composite matrix get the numbers.
numcols = activecols + (1:2:2*n);

% Compute which columns in the composite matrix get the number of digits.
ndigcols = numcols + 1;

% Compute which columns in the composite matrix get chars.
charcols = true(1,max_len + 2*n);
charcols(numcols) = false;
charcols(ndigcols) = false;

% Create and fill composite matrix, comp.
comp = zeros(num_str,max_len + 2*n);
comp(:,charcols) = double(s1);
comp(:,numcols) = num_val(:,activecols);
comp(:,ndigcols) = num_dig(:,activecols);

% Sort rows of composite matrix and use index to sort c in ascending or
% descending order, depending on mode.
[unused,index] = sortrows(comp);
if is_descend
    index = index(end:-1:1);
end
index = reshape(index,size(c));
cs = c(index);

```

6. Create Feature Vector

```

function [learnset] = createFeatureVector()
    learnset = []; c={};
    imagefiles = dir('Custom1\*.jpg');
    c={c,imagefiles.name};
    c=c(2:end);
    c=sort_nat(c);
    len = length(imagefiles);

    for i=1:len
        img = imread(c{i});
        a = [];
        for j=7:10
            T = kratchoukmoment(j, j, 2, img);
            val=abs(sum(T(:)))/10e+18;
            a = [a,val];
        end
        learnset = [learnset;a];
    end
end

```

7. KNN Classification

```
function result = knnclassification(testsamplesX,samplesX, samplesY, Knn,type)
```

```
% Classify using the Nearest neighbor algorithm
```

```
% Inputs:
```

```
% samplesX - Train samples
```

```
% samplesY - Train labels
```

```
% testsamplesX - Test samples
```

```
% Knn - Number of nearest neighbors
```

```
%
```

```
% Outputs
```

```
% result - Predicted targets
```

```
if nargin < 5
```

```
    type = '2norm';
```

```
end
```

```
L = length(samplesY);
```

```
Uc = unique(samplesY);
```

```
if (L < Knn),
```

```
    error('You specified more neighbors than there are points.')
```

```
end
```

```
N = size(testsamplesX, 1);
```

```
result = zeros(N,1);
```

```
switch type
```

```
case '2norm'
```

```
    for i = 1:N,
```

```
        dist = sum((samplesX - ones(L,1)*testsamplesX(i,:)).^2,2);
```

```
        [m, indices] = sort(dist);
```

```
        n = hist(samplesY(indices(1:Knn)), Uc);
```

```
        [m, best] = max(n);
```

```
        result(i) = Uc(best);
```

```
    end
```

```
case '1norm'
```

```
    for i = 1:N,
```

```
        dist = sum(abs(samplesX - ones(L,1)*testsamplesX(i,:)),2);
```

```
        [m, indices] = sort(dist);
```

```
        n = hist(samplesY(indices(1:Knn)), Uc);
```

```
        [m, best] = max(n);
```

```
        result(i) = Uc(best);
```

```
    end
```

```
case 'match'
```

```
    for i = 1:N,
```

```
        dist = sum(samplesX == ones(L,1)*testsamplesX(i,:),2);
```

```
        [m, indices] = sort(dist);
```

```
        n = hist(samplesY(indices(1:Knn)), Uc);
```

```
        [m, best] = max(n);
```

```
        result(i) = Uc(best);
```

```

    end
otherwise
    error('Unknown measure function');
end

```

8. Test Image

```

function [result] = test_image(a, learn_array, label_array, k)
    img = imread(a);
    sol = [];
    for j=7:10
        T = kratchoukmoment(j, j, 2, img);
        val=abs(sum(T(:)))/10e+18;
        sol = [sol,val];
    end
    result = knnclassification( sol,learn_array, label_array, k);
end

```