Real-Time Hand Gesture Detection and Recognition Using Simple Heuristic Rules

Submitted by : **MARIA ABASTILLAS**

Supervised by : **DR. ANDRE L. C. BARCZAK**
DR. NAPOLEON H. REYES

Paper : **159.333 PROJECT IMPLEMENTATION**

Acknowledgments

I would like to express my sincerest and heartfelt thanks to my supervisors, Dr. Andre L. C. Barczak and Dr. Napoleon H. Reyes, for their unwavering teaching, support and encouragement in the making of this Project Report for 159.333. Without their patience and dedication, I would just have thrown in the towel and called it a day, perhaps never to know the wonders of computer vision and robotics.

Maria Abastillas, 17 June 2011.

Contents

1. Introduction

Robots are used successfully in many areas today, particularly in industrial production, military operations, deep sea drilling, and space exploration. This success drives the interest in the feasibility of using robots in human social environments, particulary in the care of the aged and the handicapped. In social environments, humans communicate easily and naturally by both speech (audio) and gesture (vision) without the use of any external devices (like keyboards) requiring special training. Robots have to adapt to human modes of communication to promote a more natural interaction with humans. Given a choice between speech and gesture, some researchers have opined that gesture recognition would be more reliable than speech recognition because the latter would need a greater number of training datasets to deal with the greater variability in human voice and speech [1].

This project is about implementing the control of a robot through simple hand gestures. The main motivation is the desirability of developing robots that can interact smoothly with humans without the need of any special devices.

The objectives of the project are:

1) Study and apply the needed tools, namely:
    a) An omnidirectional robot from the Massey University Robot Laboratory
    b) The OpenCV Computer Vision Library (version 2.0)
    c) Algorithms for computer vision and artificial intelligence
2) Develop a computer vision application for simple gesture recognition
3) Test the computer application
4) Document the results of the project

During the project, four gestures were chosen to represent four navigational commands for the robot, namely Move Forward, Move Left, Move Right, and Stop. A simple computer vision application was written for the detection and recognition of the four gestures and their translation into the corresponding commands for the robot. The appropriate OpenCV functions and image processing algorithms for the detection and interpretation of the gestures were used. Thereafter, the program was tested on a webcam with actual hand gestures in real-time and the results were observed.

The results of the project demonstrated that a simple computer vision application can be designed to detect and recognize simple hand gestures for robot navigational control based on simple heuristic rules. The program was able to correctly interpret the gestures and translate it into the corresponding commands most of the time.

This report will present a short review of computer vision and digital image processing, introduce OpenCv as a popular tool for the development of computer vision applications, discuss pattern recognition and classifiers, Hu's Moment Invariants, and the AdaBoost algorithm, discuss the project tools and methodology, outline the steps undertaken to complete the project, and discuss the results and conclusion.

2. Literature Review

2.1 Computer vision and Digital Image Processing

The sense of sight is arguably the most important of man's five senses. It provides a huge amount of information about the world that is rich in detail and delivered at the speed of light. However, human vision is not without its limitations, both physical and psychological. Through digital imaging technology and computers, man has transcending many visual limitations. He can see into far galaxies, the microscopic world, the sub-atomic world, and even "observe" infra-red, x-ray, ultraviolet and other spectra for medical diagnosis, meteorology, surveillance, and military uses, all with great success.

While computers have been central to this success, for the most part man is the sole interpreter of all the digital data. For a long time, the central question has been whether computers can be designed to analyze and acquire information from images autonomously in the same natural way humans can. According to Gonzales and Woods [2], this is the province of computer vision, which is that branch of artificial intelligence that ultimately aims to "use computers to emulate human vision, including learning and being able to make inferences and tak[ing] actions based on visual inputs."

The main difficulty for computer vision as a relatively young discipline is the current lack of a final scientific paradigm or model for human intelligence and human vision itself on which to build a infrastructure for computer or machine learning [3]. The use of images has an obvious drawback. Humans perceive the world in 3D, but current visual sensors like cameras capture the world in 2D images. The result is the natural loss of a good deal of information in the captured images. Without a proper paradigm to explain the mystery of human vision and perception, the recovery of lost information (reconstruction of the world) from 2D images represents a difficult hurdle for machine vision [4]. However, despite this limitation, computer vision has progressed, riding mainly on the remarkable advancement of decades-old digital image processing techniques, using the science and methods contributed by other disciplines such as optics, neurobiology, psychology, physics, mathematics, electronics, computer science, artificial intelligence and others.

Computer vision techniques and digital image processing methods both draw the proverbial water

from the same pool, which is the digital image, and therefore necessarily overlap. Image processing takes a digital image and subjects it to processes, such as noise reduction, detail enhancement, or filtering, for the purpose of producing another desired image as the end result. For example, the blurred image of a car registration plate might be enhanced by imaging techniques to produce a clear photo of the same so the police might identify the owner of the car. On the other hand, computer vision takes a digital image and subjects it to the same digital imaging techniques but for the purpose of analyzing and understanding what the image depicts. For example, the image of a building can be fed to a computer and thereafter be identified by the computer as a residential house, a stadium, high-rise office tower, shopping mall, or a farm barn. [5]

Russell and Norvig [6] identified three broad approaches used in computer vision to distill useful information from the raw data provided by images. The first is the feature extraction approach, which focuses on simple computations applied directly to digital images to measure some useable characteristic, such as size. This relies on generally known image processing algorithms for noise reduction, filtering, object detection, edge detection, texture analysis, computation of optical flow, and segmentation, which techniques are commonly used to pre-process images for subsequent image analysis. This is also considered an "uninformed" approach.

The second is the recognition approach, where the focus is on distinguishing and labelling objects based on knowledge of characteristics that sets of similar objects have in common, such as shape or appearance or patterns of elements, sufficient to form classes. Here computer vision uses the techniques of artificial intelligence in knowledge representation to enable a "classifier" to match classes to objects based on the pattern of their features or structural descriptions. A classifier has to "learn" the patterns by being fed a training set of objects and their classes and achieving the goal of minimizing mistakes and maximizing successes through a step-by-step process of improvement. There are many techniques in artificial intelligence that can be used for object or pattern recognition, including statistical pattern recognition, neural nets, genetic algorithms and fuzzy systems.

The third is the reconstruction approach, where the focus is on building a geometric model of the world suggested by the image or images and which is used as a basis for action. This corresponds to the stage of image understanding, which represents the highest and most complex level of computer vision processing. Here the emphasis is on enabling the computer vision system to construct internal models based on the data supplied by the images and to discard or update these internal

models as they are verified against the real world or some other criteria. If the internal model is consistent with the real world, then image understanding takes place. Thus, image understanding requires the construction, manipulation and control of models and at the moment relies heavily upon the science and technology of artificial intelligence.

2.2 OpenCV

OpenCv is a widely used tool in computer vision. It is a computer vision library for real-time applications, written in C and C++, which works with the Windows, Linux and Mac platforms. It is freely available as open source software from http://sourceforge.net/projects/opencvlibrary/.

OpenCv was started by Gary Bradsky at Intel in 1999 to encourage computer vision research and commercial applications and, side-by-side with these, promote the use of ever faster processors from Intel [7]. OpenCV contains optimised code for a basic computer vision infrastructure so developers do not have to re-invent the proverbial wheel. The reference documentation for OpenCV is at http://opencv.willowgarage.com/documentation/index.html. The basic tutorial documentation is provided by Bradsky and Kaehler [6]. According to its website, OpenCV has been downloaded more than two million times and has a user group of more than 40,000 members. This attests to its popularity.

A digital image is generally understood as a discrete number of light intensities captured by a device such as a camera and organized into a two-dimentional matrix of picture elements or pixels, each of which may be represented by number and all of which may be stored in a particular file format (such as jpg or gif) [8]. OpenCV goes beyond representing an image as an array of pixels. It represents an image as a data structure called an `IplImage` that makes immediately accessible useful image data or fields, such as:

- `width` – an integer showing the width of the image in pixels
- `height` – an integer showing the height of the image in pixels
- `imageData` – a pointer to an array of pixel values
- `nChannels` – an integer showing the number of colors per pixel
- `depth` – an integer showing the number of bits per pixel
- `widthStep` – an integer showing the number of bytes per image row
- `imageSize` – an integer showing the size of in bytes

- `roi` – a pointer to a structure that defines a region of interest within the image [9].

OpenCV has a module containing basic image processing and computer vision algorithms. These include:

- smoothing (blurring) functions to reduce noise,
- dilation and erosion functions for isolation of individual elements,
- floodfill functions to isolate certain portions of the image for further processing,
- filter functions, including Sobel, Laplace and Canny for edge detection,
- Hough transform functions for finding lines and circles,
- Affine transform functions to stretch, shrink, warp and rotate images,
- Integral image function for summing subregions (computing Haar wavelets),
- Histogram equalization function for uniform distribution of intensity values,
- Contour functions to connect edges into curves,
- Bounding boxes, circles and ellipses,
- Moments functions to compute Hu's moment invariants,
- Optical flow functions (Lucas-Kanade method),
- Motion tracking functions (Kalman filters), and
- Face detection/ Haar classifier.

OpenCV also has an ML (machine learning) module containing well known statistical classifiers and clustering tools. These include:

- Normal/ naïve Bayes classifier,
- Decision trees classifier,
- Boosting group of classifiers,
- Neural networks algorithm, and
- Support vector machine classifier.

2.3 Pattern Recognition and Classifiers

In computer vision a physical object maps to a particular segmented region in the image from which object descriptors or features may be derived. A *feature* is any characteristic of an image, or any region within it, that can be measured. Objects with common features may be grouped into classes,

where the combination of features may be considered a *pattern*. Object recognition may be understood to be the assignment of classes to objects based on their respective patterns. The program that does this assignment is called a *classifier*. [10]

The general steps in pattern recognition may be summarized in Figure 1 below:

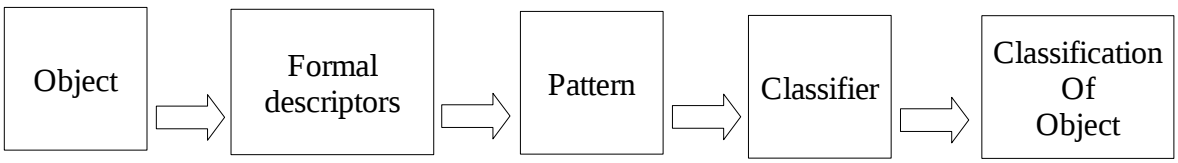


Figure 1. General pattern recognition steps. [3]

The most important step is the design of the formal descriptors because choices have to be made on which characteristics, quantitative or qualitative, would best suit the target object and in turn determines the success of the classifier.

In statistical pattern recognition, quantitative descriptions called features are used. The set of features constitutes the pattern vector or feature vector, and the set of all possible patterns for the object form the *pattern space X* (also known as *feature space*). Quantitatively, similar objects in each class will be located near each other in the feature space forming clusters, which may ideally be separated from dissimilar objects by lines or curves called *discrimination functions*. Determining the most suitable discrimination function or *discriminant* to use is part of classifier design.

A statistical classifier accepts *n* features as inputs and gives 1 output, which is the classification or decision about the class of the object. The relationship between the inputs and the output is a *decision rule*, which is a function that puts in one space or subset those feature vectors that are associated with a particular output. The decision rule is based on the particular discrimination function used for separating the subsets from each other.

The ability of a classifier to classify objects based on its decision rule may be understood as *classifier learning*, and the set of the feature vectors (objects) inputs and corresponding outputs of classifications (both positive and negative results) is called the *training set.* It is expected that a

well-designed classifier should get 100% correct answers on its training set. A large training set is generally desirable to optimize the training of the classifier, so that it may be tested on objects it has not encountered before, which constitutes its *test set*. If the classifier does not perform well on the test set, modifications to the design of the recognition system may be needed.

2.4 Moment Invariants

As mentioned previously, feature extraction is one approach used in computer vision. According to A.L.C. Barczak, feature extraction refers to the process of distilling a limited number of features that would be sufficient to describe a large set of data, such as the pixels in a digital image [8]. The idea is to use the features as a unique representation of the image.

Since a digital image is a two-dimentional matrix of pixels values, region-based object descriptions are affected by geometric transformations, such as scaling, translation, and rotation. For example, the numerical features describing the shape of a 2D object would change if the shape of the same object changes as seen from a different angle or perspective. However, to be useful in computer vision applications, object descriptions must be able to identify the same object irrespective of its position, orientation, or distortion.

One of the most popular quantitative object descriptors are moments. The concept of statistical characteristics or moments that would be indifferent to geometric transformations was first formulated by Hu in 1962. Moments are polynomials of increasing order that describe the shape of a statistical distribution [10]. The order of a moment is indicated by its exponent. The geometric moments of different orders represent different spatial characteristics of the image intensity distribution. A set of moments can thus form a global shape descriptor of an image [11].

Hu proposed that the following seven functions (called 2D *moment invariants*) were invariant to translation, scale variation, and rotation of an image [12]:

$$\phi_1 = \eta_{20} + \eta_{02}$$

$$\phi_2 = (\eta_{20} - \eta_{02})^2 + 4\eta_{11}^2$$

$$\phi_3 = (\eta_{30} - 3\eta_{12})^2 + (\eta_{03} - 3\eta_{21})^2$$

$$\phi_4 = (\eta_{30} + \eta_{12})^2 + (\eta_{03} + \eta_{21})^2$$

$$\phi_5 = (\eta_{30} - 3\eta_{12})(\eta_{30} + \eta_{21})[(\eta_{30} + \eta_{12})^2 - 3(\eta_{21} + \eta_{03})^2]$$
$$+ \ (3\eta_{21} - \eta_{03})(\eta_{21} + \eta_{03})[3(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2]$$

$$\phi_6 = (\eta_{20} - \eta_{02})[(\eta_{30} + \eta_{21})^2 - (\eta_{21} + \eta_{03})^2] + 4\eta_{11}(\eta_{30} + \eta_{12})(\eta_{21} + \eta_{03})$$

$$\phi_7 = (3\eta_{21} - \eta_{03})(\eta_{30} + \eta_{12})[(\eta_{30} + \eta_{12})^2 - 3(\eta_{21} + \eta_{03})^2]$$
$$+ \ (3\eta_{12} - \eta_{30})(\eta_{21} + \eta_{03})[3(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2]$$

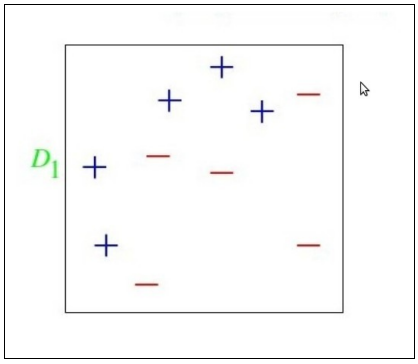
Since they are invarient to geometric transformations, a set of moment invariants computed for an image may be considered as a feature vector. A set of feature vectors might constitute a class for object detection and recognition. The feature vectors of a class of reference images can be compared with the feature vectors of the image of an unknown object, and if their feature vectors do not match, then they may be considered as different objects. The usefulness of moment invariants as image shape descriptors in pattern recognition and object identification is well established [11]. A code fragment implementing an approximation of the first of Hu's moment invariants is presented in the next section. OpenCV has built-in functions for the calculation of moments: cvMoments(), cvGetCentralMoment(), cvGetNormalizedCentralMoment() and cvGetHuMoments() [9].

2.5 AdaBoost Algorithm

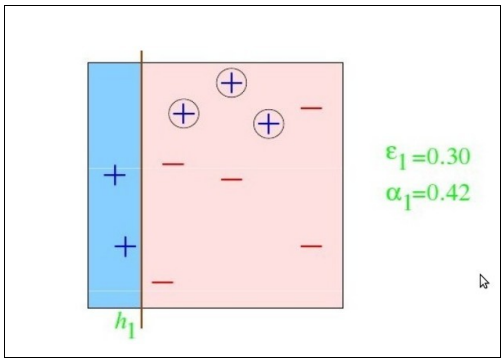
AdaBoost (*Ada*ptive *Boost*ing) is a machine learning algorithm to build a classifier based on *boosting*, a technique which was introduced by Freund and Schapire in 1999 [13]. The idea of boosting is to construct a single strong classifier from a linear combination of weak classifiers. A weak classifier uses a decision rule that has a performance value of only slightly above 50%, just over chance. A weak classifier might use a simple threshold function. In the method of boosting, at each round of training, the training features are given new weights based on whether they were correctly classified or not in the previous round by the weak classifiers. Mis-classified features are given bigger weights and correctly classified features are given lower weights for the next training

round. The weak classifiers themselves are also given new weights based on their average error in the previous round. The method results in a final classifier that combines in an optimal way the strengths of the individual classifiers.

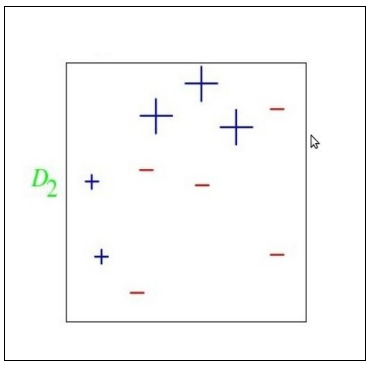
Figure 2 shows an illustration of boosting.



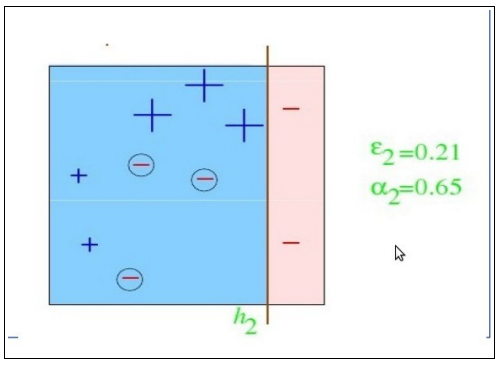
(a)



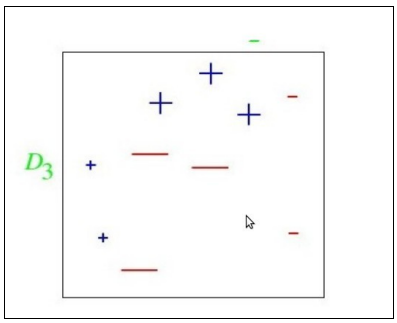
(b)



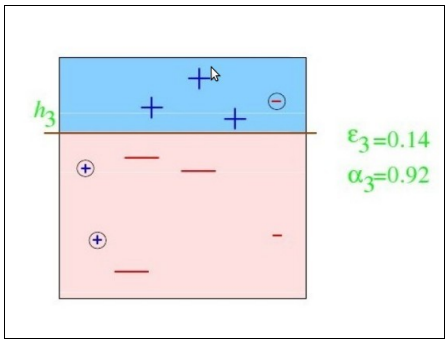
(c)



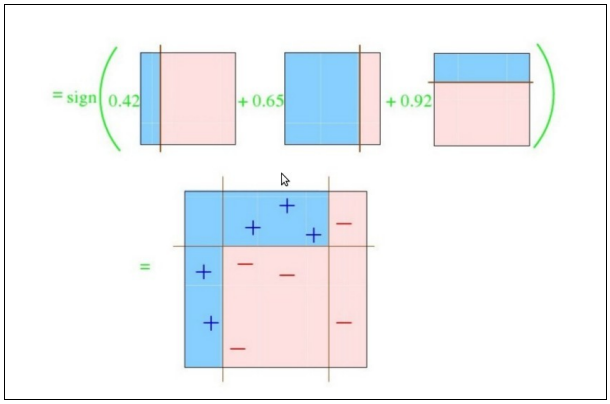
(d)



(e)



(f)



(g)

Figure 2. Illustration of stages in boosting. (a) Elements are given equal weights in D_1. *(b)* The classifier h_1 incorrectly identifies 3 elements. (c) In D_2 the 3 mis-identified elements are given greater weight, the correct ones are minimized. (d) The classifier h_2 incorrectly identifies 3 elements. (e) In D_3 the 3 misidentified elements are given greater weight, the correct ones are minimized. (f) The classifier h_3 incorrectly identifies 1 elements. (g) The final classifier combines the acceptable performances of classifiers h_1, h_2, and h_3. [8]

The algorithm for a discrete version of AdaBoost is shown below [4, 8]:

1 Input the training examples with labels $\{(x_1, y_1), (x_2,y_2),\ldots (x_i, y_i)$, where x_i is a feature vector in the feature space X and y_i is a label from $Y = \{-1 \text{ or } +1\}$ for negative or positive results.

2 Initialize all the weights $D_1(i) = 1/(n)$, where n is the number of elements in the feature vector.

3 For $t = 1,2, \ldots T$ (T is the number of rounds):

3.1 Train a weak classifier h_t, using the weights in D_1 so that $h_t \in \{-1,+1\}$.

3.2 Compute the error associated with the weak classifier (the sum of the elements' weights that are incorrectly classified):

$$error_t = \sum_{i=0}^{n} D_t(i) h_t(x_i) y_i$$

3.3 Compute the weight factor α_t for the classifier:

$$\alpha_t = 0.5 \ln\left(\frac{(1-error_t)}{error_t}\right)$$

3.4 Update the weights $D_{t+1}(i)$ such that D_{t+1} is a new distribution, and the weights decrease if the element is correctly classified and increase if incorrectly classified:

$$D_{t+1}(i)=D_t(i)^{-\alpha_t y_t h_t(x_i)}$$

3.5 Re-normalize the weights:

$$D_{t+1}(i)=\frac{D_{t+1}(i)}{\sum_{i=0}^{n} D_{t+1}(i)}$$

4 After *T* rounds, output the final classifier as a linear combination of selected classifiers:

$$H(x)=sign[\sum_{i=0}^{n} \alpha_t h_t(x)]$$

In a recent study to improve pecan defect classification using machine learning classifiers, it was found that AdaBoost algorithms performed better compared to support vector machines and Bayesian classifiers, taking less time with more accuracy [15]. It was also recently combined with scale invariant feature transform (SIFT) features to improve multi-class hand posture recognition for human-robot interaction [16].

2.6 The Special Role of Heuristic Rules

Even if computer vision achieves pattern recognition using complex and sophisticated techniques in artificial intelligence, there is a place in it for the application of heuristic rules. A *heuristic* is knowledge that is empirical but vague. Sometimes in problems of high complexity looking for an exact and exhaustive solution is impracticable or overly expensive, if not impossible. Sometimes no algorithm is known or can be found to provide the solution. Often the intention is only one of exploration where an acceptable estimate or approximation of a possible solution is all that is required for the present. In these cases, a good heuristic or rule of thumb is all that is required to have a good working solution [3].

2.7 Description of the Robot

The Massey robot (known as Omni-Robot) has an omni-directional camera that gives it a 360 degree field of vision so that it can “see” in all directions. It is also capable of omni-directional movement (at any angle) with the use of its 3 Omni-Wheel sets. It can run at a maximum

acceleration of 1.8 m/sec^2. The robot has an on-board PC that runs on Windows and can communicate with it directly by serial connection. A remote network connection is also possible with the server program running on the robot PC. It has direct power connection as well as a battery pack. It has a native manual dialog control program for robot movement in serial mode with buttons for the following movements: Forward, Left, Halt, Right, Backward, Rotate Clock-wise, and Rotate Counter Clock-wise. [14]

A computer vision application can be made to capture images containing hand gestures from the robot's omni-directional camera, detect and recognize the hand gestures, and output the applicable commands to navigate the robot.

3. Methodology

This project availed of several algorithms commonly used in computer vision. These include those used in colour segmentation, labelling, blob detection, feature extraction, and gesture recognition.

3.1 Algorithm for Colour Segmentation Using Thresholding

Segmentation is the process of identifying regions within an image [10]. Colour can be used to help in segmentation. In this project, the hand on the image was the region of interest. To isolate the image pixels of the hand from the background, the range of the HSV values for skin colour was determined for use as the threshold values. Segmentation could then proceed after the conversion of all pixels falling within those threshold values to white and those without to black.

The algorithm used for the colour segmentation using thresholding is shown below:

1. Capture an image of the gesture from the camera.
2. Determine the range of HSV values for skin colour for use as threshold values.
3. Convert the image from RGB colour space to HSV colour space.
4. Convert all the pixels falling within the threshold values to white.
5. Convert all other pixels to black.
6. Save the segmented image in an image file.

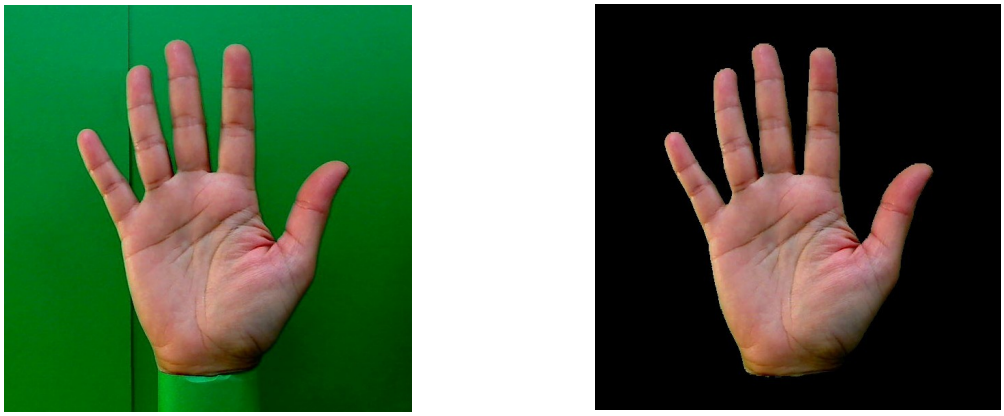
The pseudocode for the segmentation function is as follows:

```
//initialize threshold values for skin colour
maxH = maximum value for Hue
minH = minimum value for Hue
maxS = maximum value for Saturation
minS = minimum value for Saturation
maxV = maximum value for Value
minV = minimum value for Value

//initialize values for RGB components
```

```
pixelR = red component
pixelG = green component
pixelB = blue component

//the function uses two identical 3-channel colour images and
//saves the segmented pixels in imageA
segmentationfunction (imageA, imageB)
{
        //convert imageB to HSV, using cvCvtColor function of OpenCV
        cvCvtColor(imageA, imageB, CV_RGB2HSV);

        //access all the image pixels
        for ( x=0; x < width of imageB; x++)
        {
                for ( y=0; y < height of imageB; y++)
                {
                        if(     //imageB pixels are pixels outside the threshold range
                                pixelR of imageB < minV ||
                                pixelR of imageB > maxV ||
                                pixelG of imageB < minS ||
                                pixelG of imageB > maxS ||
                                pixelB of imageB < minH ||
                                pixelB of imageB > maxH   )
                        {
                                //convert pixels to colour black
                                pixelR of imageA =0;
                                pixelG of imageA =0;
                                pixelB of imageA =0;
                        }
                        else {
                                //convert pixels to colour white
                                pixelR of imageA =255;
                                pixelG of imageA =255;
                                pixelB of imageA =255;
                        }
                }
        }
```

}

Figure 3 shows a sample raw image and the resulting image after colour segmentation by thresholding.



(a) (b)

Figure 3: Sample images. (a) Original image. (b) Image after colour segmentation.

3.2 Algorithm for Labeling and Blob Detection

The gesture in the colour segmented image should be recognized as one object before it can be interpreted. This can be done through the process of labeling and blob detection.

Labeling is the process of giving each region a unique integer number or label for the purpose of regional identification [3]. In effect, while no two neighboring regions should have the same label, the pixels within one region should have the same label or description so that the region could be interpreted as one object or blob.

For the purpose of determining whether pixels might belong to the same region, their adjacency relationships can be examined. The two most common adjacency relationships are:

- 4-adjacency and
- 8-adjacency [8]

In 4-adjacency, a pixel is considered connected to its neighboring pixels if they occupy the left-most, top-most, right-most, and bottom positions with respect to the pixel. Using the (x,y) coordinate descriptions, a 4-adjacency relationship for pixel (x, y) is shown in Figure 4.

	(x, y-1)	
(x-1, y)	(x, y)	(x+1, y)
	(x, y+1)	

Figure 4. 4-Adjacency connectivity model.

In 8-adjacency, the neighboring pixels also include the top-left-most, top-right-most, bottom-left-most, and the bottom-right-most positions. Using the (x,y) coordinate descriptions, a 8-adjacency relationship for pixel (x, y) is shown in Figure 5.

(x-1, y-1)	(x, y-1)	(x+1, y-1)
(x-1, y)	(x, y)	(x+1, y)
(x-1, y+1)	(x, y+1)	(x+1, y+1)

Figure 5. 8-Adjacency connectivity model.

In labelling algorithms, pixels are examined one by one, row by row, and moving from left to right. Therefore, for the practical implementation of the algorithm, only the pixels that may be considered as existing at each point in time with respect to the pixel under scrutiny are considered. For the 4-adjacency model, these pixels would be the top-most and the left-most, as shown in Figure 6.

	(x, y-1)
(x-1, y)	(x, y)

Figure 6. Significant pixels in 4-adjacency model.

In the 8-adjacency model, these pixels would be the top-left-most, the top-most, the top-right-most, and the left-most, as shown in Figure 7.

(x-1, y-1)	(x, y-1)	(x+1, y-1)
(x-1, y)	(x,y)	

Figure 7. Significant pixel in 8-adjacency model.

Usually, images that undergo labelling are given preliminary processing to become binary images or grayscale images. This will make it easier to identify the pixels of interest because the background pixels would either be zero-valued (coloured black) or measured against a threshold value in grayscale.

The labelling procedure looks at each pixel in sequence, checks it against the threshold value, and identifies its neighboring pixels based on the adjacency relationship model being used. If the neighboring pixels belong to a set, the pixel under consideration will be placed in that set. If the pixel has no neighbors, then it will be placed in a new set. Thereafter, all sets with connecting pixels will be merged together into one set or blob and be considered as one object.

The algorithm for labelling and blob detection using an 8-adjacency relationship and a threshold value for a grayscale image is as follows:

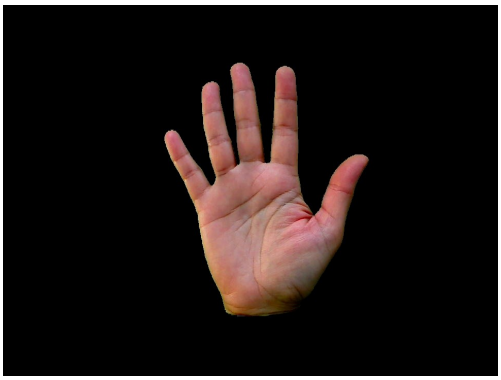
1. Select the threshold value for the grayscale image.
2. For each non-zero pixel in the image that is above the threshold value:
   (a) If the pixel has non-zero labeled pixels in its immediate 8-adjacency neighborhood (namely, the top-right, top, top-left and left pixels), then give it a non-zero label (for example, assign the minimum value of the pixels in this neighborhood to the pixel as its label).
   (b) If the pixel has no neighboring pixels, then give it a new unused label and include it as part of a new set.
3. After all the pixels have been labeled and placed in sets, merge together the sets with connected pixels into one blob or object. Objects may be given different colours to distinguish them visually.

In pseudocode, the algorithm may be shown as follows:

```
Require: Set of sets, counters, threshold, input image I = i(x,y)

for (n=0; n < width of image; n++)
{
    for (m=0; m < height of image; m++)
    {
        if ( i(x,y) != 0 && i(x,y) > threshold)
        // not a background pixel and above the threshold
        {
            if ( i(x-1, y-1) != 0 || //it has at least one labeled neighbor
                i(x-1, y) != 0 ||
                i(x, y-1) != 0 ||
                i(x+1, y-1) != 0 )
            {//give it a label
                i(x,y) → set(i(a,b)) {where (a,b) ∈
                    {(x-1, y-1), (x-1, y), (x, y-1), (x+1, y-1)}}
                //merge sets, if possible
                if ( set(i(x-1, y-1)) != set(i( x-1, y)) )
                {
                    set(i(x-1, y-1))  ∪  set(i( x-1, y))
                }

                if ( set(i(x-1, y)) != set(i( x-1, y)) )
                {
                    set(i(x-1, y))  ∪  set(i(x-1, y))
                }

                if ( set(i(x, y-1)) != set(i( x, y-1)) )
                {
                    set(i(x, y-1))  ∪  set(i(x, y-1))
                }

                if ( set(i(x+1, y-1)) != set(i( x+1, y-1)) )
```

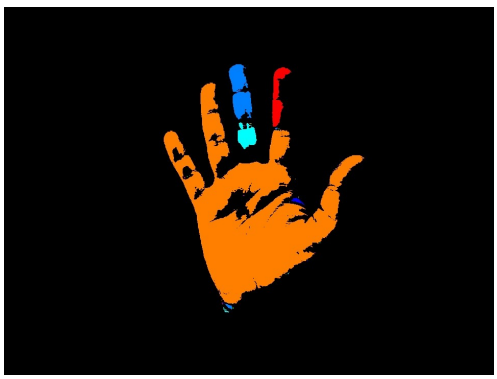
```
                    {
                        set(i(x+1, y-1))  ∪  set(i(x+1, y-1))
                    }
                }
                else  //make new set for pixels without neighbors
                {
                    makeNewSet( set(i(x, y)))
                    i(x,y) → set(i(x, y))
                }
            }
        }
    }
```

Figure 8 shows a sample original segmented image of hand with skin colour that is converted to a grayscale image, and the finally into two separate images, one showing a single blob using a lower lower threshold and another showing multiple blob using a higher threshold value.



(a)

(b)



(c) (d)

Figure 8: Sample images showing the results of labelling and blob detection. (a) Original segmented image. (b) Image in grayscale. (c) Image with a single blue-coloured blob, using of a threshold pixel value = 10. (d) Image showing multiple blobs with individual colours, using a higher threshold pixel value = 120.

3.3 Algorithm for Feature Extraction (Moment Invariants)

A digital approximation of finding the geometric moments of an image involves the product of pixel values and pixel positions, as shown by the formulas below with respect to an M x N image *i(x, y)* for the moment of order (*p* + *q*):

$$m_{pq} = \sum_{x=0}^{M} \sum_{y=0}^{N} x^p y^q i(x, y)$$

The moment of order zero (m_{00}) is equivalent to the total intensity of the image and for a blob the total geometrical area. The first-order functions m_{10} *and* m_{01} give the intensity moments about the y-axis and x-axis respectively.

The intensity centroid ($\bar{x}$, $\bar{y}$) gives the geometric center of the image and the formula to obtain it is:

$$\bar{x} = \frac{m_{10}}{m_{00}} \qquad \bar{y} = \frac{m_{01}}{m_{00}}$$

The central moments are moments computed with respect to the centroid. The central moment μ_{pq} is calculated as follows:

$$\mu_{pq} = \sum_{x=0}^{M} \sum_{y=0}^{N} (\bar{x} - x)^p (\bar{y} - y)^q i(x, y)$$

The normalized central moment η_{pq} is computed as:

$$\eta_{pq} = \frac{\mu_{pq}}{\mu_{00}^{\gamma}} \quad \text{where} \quad \gamma = \frac{p+q+2}{2}$$

The first of Hu's moment invariants is given by:

$$\phi_1 = \eta_{20} + \eta_{02}$$

As an illustrative example, a C++ code fragment implementing the computation of the first Hu's moment invariant using an arbitrary 3x3 image is shown below.

```
int i, j; double sum = 0.0;
double image[3][3] = {{4,5,6},{7,8,9},{10,11,12}};
double m00 = 0.0; //representing m00
double m10 = 0.0; //representing m10
double m01 = 0.0; //representing m01

//computation of   m00 = sum_{x=0}^{3} sum_{y=0}^{3} x^0 y^0 i(x,y)

      for(i=0; i<3; i++)
      {
         for(j=0; j<3; j++)
         {
            cout<< setw(5)<< pow(i,0) * pow(j,0) * image[i][j] << " ";
            m00 = m00 + pow(i,0) * pow(j,0) * image[i][j];
         }
            cout<<endl;
      }

// computation of   m10 = sum_{x=0}^{3} sum_{y=0}^{3} x^1 y^0 i(x,y)
```

```
      for(i=0; i<3; i++)
      {
         for(j=0; j<3; j++)
         {
            cout<< setw(5)<< pow(i,1) * pow(j,0) * image[i][j] << " ";
            m10 = m10 + pow(i,1) * pow(j,0) * image[i][j];
         }
            cout<<endl;
      }
```

//computation of $m_{01}=\sum_{x=0}^{3}\sum_{y=0}^{3} x^0 y^1 i(x,y)$

```
      for(i=0; i<3; i++)
      {
         for(j=0; j<3; j++)
         {
            cout<< setw(5)<< pow(i,0) * pow(j,1) * image[i][j] << " ";
            m01 = m01 + pow(i,0) * pow(j,1) * image[i][j];
         }
            cout<<endl;
      }

  //computation of average x and average y;

  double aveX=0.0, aveY=0.0;
  aveX = m10/m00;     //for m10/m00;
  cout<<endl<<"aveX= "<<aveX<<endl;

  aveY = m01/m00;       //for m01/m00;
   cout<<endl<<"aveY= "<<aveY<<endl<<endl;
```

// computation of the central moment : upq00 for

// $\mu_{00}=\sum_{x=0}^{M}\sum_{y=0}^{N}(\overline{x}-x)^0(\overline{y}-y)^0 i(x,y)$

```
  double upq00 = 0.0, difx=0.0, dify=0.0;
      cout<<endl<<"upq00"<<endl;
```

```
  for(i=0; i<3; i++)
      {
            for(j=0; j<3; j++)
            {
                  difx = aveX - i;
      dify = aveY - j;
      cout<< setw(10)<< pow(difx,0) * pow(dify,0) * image[i][j] << " ";
      upq00 = upq00 + pow(difx,0) * pow(dify,0) * image[i][j];
            }
            cout<<endl;
      }
      cout<<endl<<"upq00= "<< upq00 <<endl;


//normalized central moment : npq00 for    η00 = μ00 / μ00^γ

  double uxpo=0.0; int p=0, q=0;
  uxpo = (p + q + 2)/2;
  cout<<endl<<"uxpo= " << uxpo <<endl;

  double npq00 =0.0;
  npq00 = upq00/pow(upq00,uxpo);
  cout<<endl<<"npq00= "<< npq00 << endl;

 //computation of Hu's first invariant    φ1 = η20 + η02
  double invar1=0.0;
  double upq20 = 0.0;
            //upq20 for     μ20 = Σ_{x=0}^{M} Σ_{y=0}^{N} (x̄ − x)^2 (ȳ − y)^0 i(x,y)
  difx=0.0; dify=0.0;


// npq20 for    η20 = μ20 / μ00^γ
  cout<<endl<<"upq20"<<endl;
  for(i=0; i<3; i++)
      {
            for(j=0; j<3; j++)
            {
                  difx = aveX - i;
      dify = aveY - j;
```

```
    cout<< setw(10)<< pow(difx,2) * pow(dify,0) * image[i][j] << " ";
    upq20 = upq20 + pow(difx,2) * pow(dify,0) * image[i][j];
          }
          cout<<endl;
    }
    cout<<endl<<"upq20= "<< upq20 <<endl;

uxpo=0.0;  p=2, q=0;
uxpo = (p + q + 2)/2;
cout<<endl<<"uxpo= " << uxpo <<endl;

double npq20 =0.0;
npq20 = upq20/pow(upq00,uxpo);
cout<<endl<<"npq20= "<< npq20 << endl;


// npq02 for    \eta_{02}=\frac{\mu_{02}}{\mu_{00}^{\gamma}}
double upq02 = 0.0;
difx=0.0; dify=0.0;
cout<<endl<<"upq02"<<endl;
    for(i=0; i<3; i++)
    {
          for(j=0; j<3; j++)
          {
                difx = aveX - i;
    dify = aveY - j;
    cout<< setw(10)<< pow(difx,0) * pow(dify,2) * image[i][j] << " ";
    upq02 = upq02 + pow(difx,0) * pow(dify,2) * image[i][j];
          }
          cout<<endl;
    }
    cout<<endl<<"upq02= "<< upq02 <<endl;

uxpo=0.0; p=0, q=2;
uxpo = (p + q + 2)/2;
cout<<endl<<"uxpo= " << uxpo <<endl;


double npq02 =0.0;          //    \eta_{02}=\frac{\mu_{02}}{\mu_{00}^{\gamma}}
npq02 = upq02/pow(upq00,uxpo);
```

```
cout<<endl<<"npq02= "<< npq02 << endl;

invar1 = npq20 + npq02;  //   φ1=η20+η02
cout<<endl<<"invar1 = npq20 + npq02 "<<endl;
cout<<endl<<"invar1= "<< invar1<<endl;
```

3.4 Algorithm for Feature Extraction (Center of Mass)

The center of mass is a simple feature of the image or object that is easy to compute. It is simple average or mean of the x and y coordinates, computed individually, to determine the *x* and *y* coordinates of the center of the image or object.

$$center_x = \frac{\sum_{i=0}^{n} x_i}{n} \qquad center_y = \frac{\sum_{i=0}^{n} y_i}{n}$$

where *n* is the total number of points of the image or object.

The pseudocode for computing the center of mass appears below:

```
//getting the center of mass of the blob
int centerx=0, centery=0; int totalpts = total points in blob;
for (int w=0; w < totalpts; w++)
{
        //access each point in the blob
        eachpt = getNextPoint(blob, w);

         //add total x's and total y's
                centerx=centerx+eachpt->x;
                centery=centery+eachpt->y;
}
//get the center of mass
centerx=centerx/totalpts;
centery=centery/totalpts;
```

Figure 9 shows the results of the computation of the center of mass of a blob object. The center of mass is represented by a red cross.

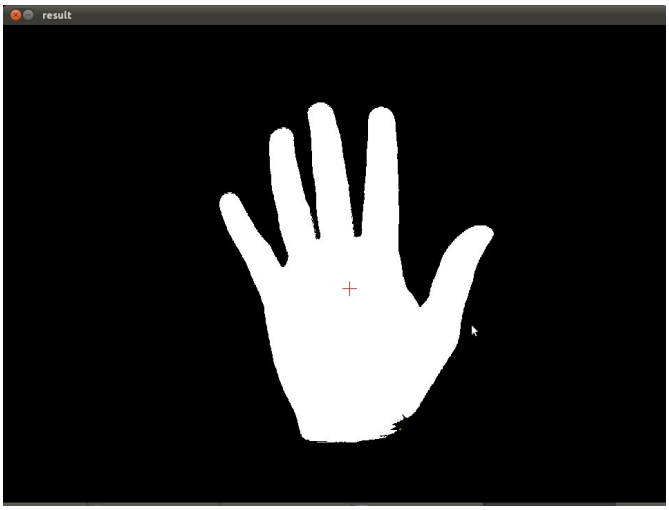


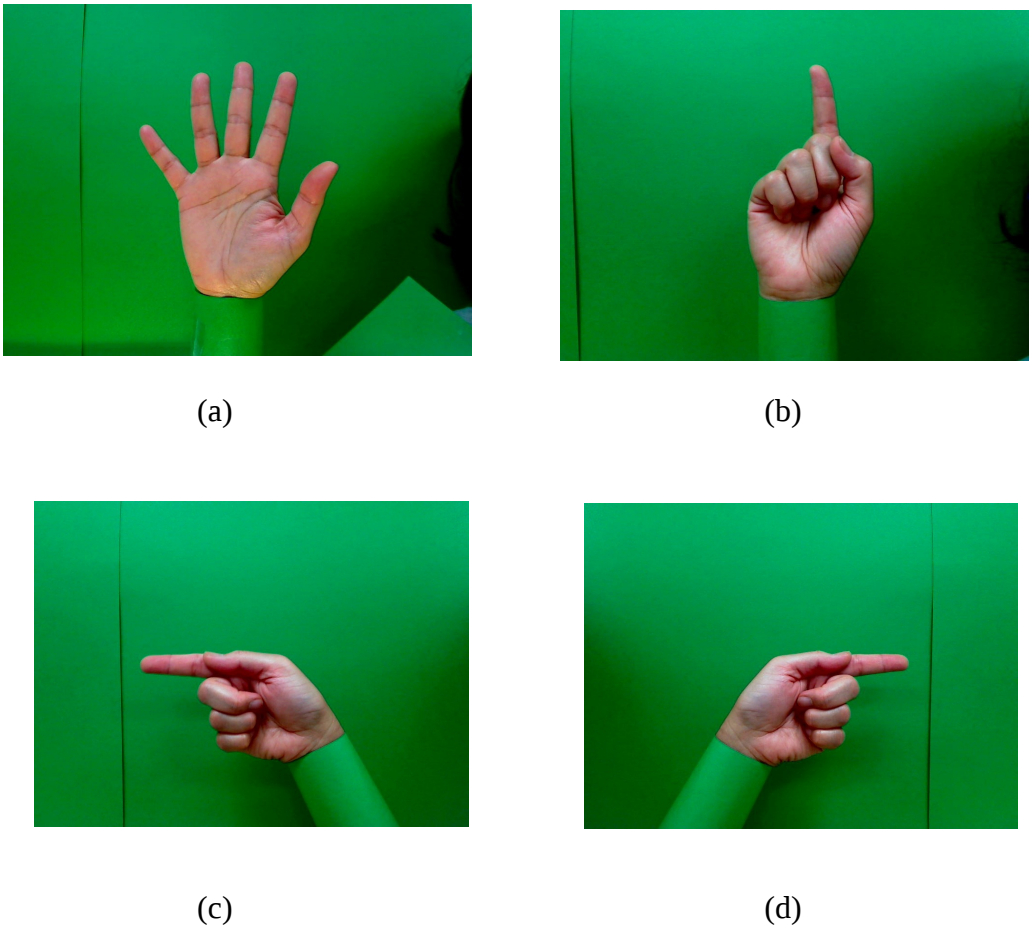
Figure 9. Screenshot of blob with the center of mass marked with a red cross.

4. Project Steps and Results

4.1 Step 1: Choose Four Gestures for Simple Robot Control

The first step in the project was to choose four gestures that would be used for simple robot control. The gestures should be simple, easy to understand, and easy for ordinary people to use. Also, the gestures should be different enough from each other so it would be easy to formulate some unique features for each of them for purposes of object recognition. Finally, the most common navigation commands were chosen, namely Stop, Move Forward, Move Left, and Move Right.

Figure 10 below shows some still images of the chosen gestures.



(a) (b)

(c) (d)

Figure 10 Sample images of chosen gestures. (a) Gesture for stop, (b) for move forward, (c) move left, and (d) move right.

4.2 Step 2: Implement the Algorithms Using C/C++ and OpenCV

The next step was to implement the needed algorithms to process the images and to recognize the gestures, using C/C++ and OpenCV. The resulting program should do the following procedure:

- capture the gestures through a web camera
- set-up an interface to convert from RGB to HSV colour space and determine the HSV values for skin colour
- use the HSV values to reduce the image to black and white
- produce a grayscale image in preparation for labelling and blob detection
- produce a blob of the hand in preparation for feature extraction
- extract features from the blob for gesture recognition
- output the recognized gesture/ command for robot navigation

4.2.1 Capture of Images Through a Webcam

If the webcam is properly installed, OpenCV would easily detect it and OpenCV capture functions can be used to get the images from the camera. The cvCaptureFromCAM() function initializes capturing video from the camera and stores it in the CvCapture structure, which is the video capturing structure. The cvCaptureFromCAM() function may be passed an index number to identify which camera is to be the source of images. In case there is only 1 camera, the parameter (0) is used. Thereafter, the CvGrabFrame() function may grab the frame (image) from the CvCapture structure, and from there the frame may be retrieved through the CvRetrieveFrame function and stored in an IplImage structure [9]. From there, the image can be displayed on the screen by the cvShowImage() function in a window created through the cvNamedWindow() function. For video streaming the process of capture and display may be placed inside a loop.

Below is the code fragment for capturing and displaying images through the webcam:

```
IplImage *imagecaptured;
CvCapture* capture = 0;
capture = cvCaptureFromCAM(0);
for ( ; ; )
```

```
{
	if( cvGrabFrame( capture )) {
		imagecaptured = cvRetrieveFrame( capture );
	}
	cvNamedWindow( "captured image", 0 );
	cvShowImage( "captured image", imagecaptured );
}
```

4.2.2 Interface for Adjustment of HSV Values for Skin Colour

The range of HSV values for skin colour had to be determined in preparation for the reduction of the image into two colours for eventual segmentation. OpenCV trackbars were used for this purpose. The trackbars or sliders represented the minimum and maximum range of the Hue, Saturation and Value of the HSV colour space. Skin colour was successfully isolated using the following combination of slider values:

```
const int upH=155;
const int loH=109;
const int upS=112;
const int loS=35;
const int upV=255;
const int loV=208;
```

However, the combination did not work all of the time. The HSV values for skin colour had to be adjusted each time, depending on the type of camera used and the amount of illumination present. Fortunately, making the periodic adjustments was facilitated by the use of the trackbars.

The cvCreateTrackbar() function used to create the trackbars have the following parameters: the trackbar name, window name, pointer to integer value showing the trackbar position, maximum position of the trackbar, and the pointer to the function to be called when the trackbar changes position. The cvSetTrackbarPos() function sets the new position of a trackbar [9].

The code fragment to set up the trackbars is shown below, using the combination of HSV values represented by the constant variables mentioned above:

```
int marker_upH = upH;
int marker_loH = loH;
int marker_upS = upS;
int marker_loS = loS;
int marker_upV = upV;
int marker_loV = loV;

cvCreateTrackbar( "upH", "result", &marker_upH, 255, NULL);
cvSetTrackbarPos("upH","result",upH);
cvCreateTrackbar( "loH", "result", &marker_loH, 255, NULL);
cvSetTrackbarPos("loH","result",loH);

cvCreateTrackbar( "upS", "result", &marker_upS, 255, NULL);
cvSetTrackbarPos("upS","result",upS);
cvCreateTrackbar( "loS", "result", &marker_loS, 255, NULL);
cvSetTrackbarPos("loS","result",loS);

cvCreateTrackbar( "upV", "result", &marker_upV, 255, NULL);
cvSetTrackbarPos("upV","result",upV);
cvCreateTrackbar( "loV", "result", &marker_loV, 255, NULL);
cvSetTrackbarPos("loV","result",loV);
```

Figure 11 shows a sample window with trackbars for colour segmentation.

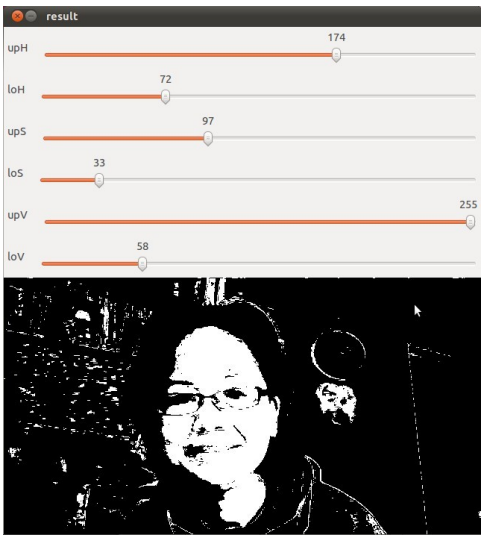


Figure 11. Sample window with trackbars for HSV values

4.2.3 Production of a Black and White Image

After the HSV skin colour range was determined, it was used as the threshold range to convert the image into two colours: black for the background and white for the hand. It was easier to implement the convertion through a separate function. Here the Find_markers() function first converts the source RGB image to an HSV image (using cvCvtColor()), and then loops through each pixel in the HSV image to pick out the skin-coloured pixels, turns the corresponding pixels in the original source image to white, and finally turns all other pixels in the original source image to black. This converts the original image into black and white. This step facilitates the isolation of the hand as the object of interest.

The code fragment to convert the image into black and white using the HSV values is shown below:

```
//skin colour setup
const int upH=155;
const int loH=109;
const int upS=112;
const int loS=35;
const int upV=255;
const int loV=208;

int marker_upH=upH;
int marker_loH=loH;
int marker_upS=upS;
int marker_loS=loS;
int marker_upV=upV;
int marker_loV=loV;

void Find_markers(IplImage* image, IplImage *imageHSV){
//convert to HSV, the order is the same as BGR
      cvCvtColor(image,imageHSV, CV_RGB2HSV);
//segment markers
      for (int x=0; x < imageHSV->width; x++){
            for (int y=0; y < imageHSV->height; y++){
                  if(   pixelR(imageHSV,x,y) < marker_loV ||
```

```
                        pixelR(imageHSV,x,y) > marker_upV ||
                        pixelG(imageHSV,x,y) < marker_loS ||
                        pixelG(imageHSV,x,y) > marker_upS ||
                        pixelB(imageHSV,x,y) < marker_loH ||
                        pixelB(imageHSV,x,y) > marker_upH   ){
                        pixelR(image,x,y)=0;
                        pixelG(image,x,y)=0;
                        pixelB(image,x,y)=0;
                  }
                  else {
                        pixelR(image,x,y)=255;
                        pixelG(image,x,y)=255;
                        pixelB(image,x,y)=255;
                  }
            }
      }
}
```

4.2.4 Production of a Grayscale Image

In OpenCV one of the methods to convert images to grayscale is through the use of the cvCvtColor() function. This function is actually a general function that converts an image from one colour space to another. Grayscale is created by changing the pixel intensities of each colour by specific amounts so that it results in a single gray intensity for output in a single-channel image, as follows:

Red * 0.299 + Green * 0.587 + Blue * 0.114 = Gray scale pixel value [9]

The cvCvtColor() function uses the following parameters: a source 3-bit, 16-bit or single-precision floating-point image, a destination image of the same data type, although channels are allowed to differ (in this case only 1-channel), and the colour conversion code, which follows the format CV_ “colour space” 2 “colour space”, namely: CV_BGR2GRAY.

The cvCreateImage() function creates an image and allocates the image data accordingly. Its parameters are: image size(width and height), image bit-depth, and number of channels per pixel. The valid image bit-depths values are:

- IPL_DEPTH_8U (unsigned 8-bit integers)
- IPL_DEPTH_8S (signed 8-bit integers)
- IPL_DEPTH_16U (unsigned 16-bit integers)
- IPL_DEPTH_16S (signed 16-bit integers)
- IPL_DEPTH_32S (signed 32-bit integers)
- IPL_DEPTH_32F (single precision floating-point numbers)
- IPL_DEPTH_64F (double precision floating-point numbers)

The code fragment below was used to convert a colour image (here already in black and white) to grayscale and display it. Imagemarkers is the name for the source image, imageblob for the destination image and “gray” for the window.

```
IplImage * imageblob;
imageblob = cvCreateImage( cvSize(imagemarkers->width,imagemarkers-
   >height), IPL_DEPTH_8U, 1 );
cvCvtColor(imagemarkers,imageblob, CV_BGR2GRAY);
cvShowImage( "gray", imageblob );
```

Conversion of the image to grayscale facilitates the next step of blob detection.

4.2.5 Production of a Blob or Object

The algorithm for labelling and blob or object detection using the 8-adjacency model together with a threshold value was discussed in the preceding section. The algorithm was implemented in the program using the following data structures:

- a array called blobs[][] to represent the names of the sets the points will belong to, and
- an array of CvSeq, each of which is an OpenCV sequence or a dynamic, growable data structure or set of points.

The implementation used three functions:

- a search function called minimumnonzero(),
- a labelling function called Union_relabel() and
- a blob-forming function called bar_blobfinder().

The minimumnonzero() function searches for the minimum value among neighboring pixels to use for labelling the pixel under consideration. The Union_relabel() function relabels the pixels and carries out unions between sets of pixels. The bar_blobfinder() begins with the black and white image as input, loops through each pixel in the image, uses a threshold value to determine which pixels should be part of a blob, groups and re-labels the pixels into sets based on their connectivity to other pixels, calls the Union_relabel() function to coalesce sets where possible, gives the resulting sets different colors, and counts the total number of blobs produced.

The implementation also uses the following OpenCV functions for sequences [9]:

- cvGetSeqElem() - to access the next element in the sequence
- cvSeqInsertSlice() - to insert an array of elements in the middle of a sequence
- ClearSeq() - remove all elements from a sequence
- cvSeqPush() - adds an element to the end of a sequence
- cvCreateSeq() - creates a sequence

The code implementing this algorithm is shown below.

```
//blobs is used to indicate which set each point belongs to
int blobs[WIDTH][HEIGHT];
int labelcount=0;
//a set of sequences (a set of sets)
CvSeq * setofseq[(WIDTH*HEIGHT)/2];
//maximum number of sets, NEVER use index=0 because of the blobs[x][y]
//numbers,(there is no set number 0)

void Union_relabel (int curr, int minA){

      if(   setofseq[curr]==NULL ||
            setofseq[minA]==NULL ||
            curr>(WIDTH*HEIGHT/2))  {
                  printf("curr or minA NULL or curr too big\n");
                  exit(0);}

      for (int t=0; t < (setofseq[curr] ? setofseq[curr]->total : 0); t++)
{
```

```
            CvPoint *pt = (CvPoint* )cvGetSeqElem(setofseq[curr], t);
            blobs[pt->x][pt->y] = minA;
            //change label on each point setofseq[curr]
      }

      cvSeqInsertSlice(setofseq[minA],0,setofseq[curr]); //union of sets
      cvClearSeq(setofseq[curr]);  //empty setofseq[curr]
} //end of Union_relabel function

void bar_blobfinder(IplImage * imagesource,IplImage * imageres, int
threshold){
      //setofseq[0]=NULL;
      labelcount=0;
      memset(&blobs[0],0,(sizeof(int)*WIDTH*HEIGHT));
      //re-initialize array
      for (int y=1; y < imagesource->height-1; y++) {
            for (int x=0; x<imagesource->width-1; x++) {
                  if (pixel(imagesource,x,y)>threshold) {
                        //coalesce sets if we can
                        if(x!=0) {
                              int minA = minimumnonzero( &(blobs[x][y-1]),
                                          &(blobs[x+1][y-1]) ,&(blobs[x-1]
                                          [y-1]) , &(blobs[x-1][y]) );
                              int curr=0;
                              curr=blobs[x][y-1];
                              if(curr>(WIDTH*HEIGHT/2)){exit(0);}
                              if (minA!=curr && curr!=0)
                                    Union_relabel(curr,minA);
                              curr=blobs[x-1][y-1];
                              if(curr>(WIDTH*HEIGHT/2)){exit(0);}
                              if (minA!=curr && curr!=0)
                                    Union_relabel(curr,minA);
                              curr=blobs[x+1][y-1];
                              if(curr>(WIDTH*HEIGHT/2)){exit(0);}
                              if (minA!=curr && curr!=0)
                                    Union_relabel(curr,minA);
                              curr=blobs[x-1][y];
                              if(curr>(WIDTH*HEIGHT/2)){exit(0);}
                              if (minA!=curr && curr!=0)
                                    Union_relabel(curr,minA);
```

```
}
//if the top pixel is labelled, include
//the scanned pixel
if( blobs[x][y-1]!=0 ) {
      blobs[x][y]=blobs[x][y-1];
      CvPoint pt={x,y};
      cvSeqPush(setofseq[blobs[x][y-1]],&pt);
      //include this point in the new set
      continue;
}
//if the top-right pixel is labelled, include
//the scanned pixel
if( blobs[x+1][y-1]!=0 ) {
      blobs[x][y]=blobs[x+1][y-1];
      CvPoint pt={x,y};
      cvSeqPush(setofseq[blobs[x+1][y-1]],&pt);
      //include this point in the new set
      continue;
}

if (x!=0){
//if the top-left pixel is labelled, include
// the scanned pixel
      if( blobs[x-1][y-1]!=0 ) {
            blobs[x][y]=blobs[x-1][y-1];
            CvPoint pt={x,y};
            cvSeqPush(setofseq[blobs[x-1]
                  [y-1]],&pt);
            //include this point in the new set
            continue;
      }
      //if the left pixel is labelled, include
      // the scanned pixel
      if( blobs[x-1][y]!=0 ) {
            blobs[x][y]=blobs[x-1][y];
            CvPoint pt={x,y};
            cvSeqPush(setofseq[blobs
                  [x-1][y] ],&pt);
            //include this point in the new set
            continue;
```

```
                        }
                    }
                    //if the scanned point is disconnected,
                    //start a new sequence
                    if (blobs[x][y]==0) {
                        blobs[x][y]=++labelcount;
                        setofseq[labelcount]    = cvCreateSeq
                            (CV_SEQ_ELTYPE_POINT, sizeof(CvSeq),
                            sizeof(CvPoint),storage);
                        //CREATE new set
                        CvPoint pt={x,y};
                        cvSeqPush(setofseq[labelcount],&pt);
                        //include this point in the new set
                    }
                }
            }
        }
        //this goes through all sequences, unmark the empty ones,
        //count the remaining number of sets put same colours for pixels
        //that belong to the same sequence, so we see as one object

        int quantos=0;
        for (int z=1; z < ((WIDTH*HEIGHT)/2); z++)
        {   //all sets from 1
            if (setofseq[z]==NULL)
                continue;//save time by not searching empty sets
            for (int t=0; t< (setofseq[z] ? setofseq[z]->total : 0);t++)
            {
                CvPoint *pt=(CvPoint*)cvGetSeqElem(setofseq[z],t);
                pixelR(imageres,(pt->x),(pt->y))=colors[(z)%8].val[0];
                pixelG(imageres,(pt->x),(pt->y))=colors[(z)%8].val[1];
                pixelB(imageres,(pt->x),(pt->y))=colors[(z)%8].val[2];
            }
            if (setofseq[z]->total!=0) {
                quantos++;
            }
        }
        if(quantos==0) return;
        printf("There are %d objects in this image \n",quantos);
        // . . rest of the code
```

```
}// end of bar_blobfinder()
```

4.2.6 Feature Extraction from the Blob for Gesture Recognition

The approach taken in the project was to extract simple features from the detected blobs and use them to come up with a simple set of heuristic rules for gesture recognition.

The following quantitative features of the blob were observed and used:

- Size of the blob – it was assumed that the hand should be the most prominent and biggest blob in the image and could therefore be identified.
- Center of mass – it was possible to compute and identify the location of the hand's center of mass and mark it.
- Distances of points from the center of mass – it was possible to compute the distances of some fingers of the hand from the hand's center of mass.
- Number of edges along a row of pixels – it was possible to do this by examining the change in colour (interpreted as an edge) along a row of pixels traversing the fingers of the hand.

The use of moment invariants was not considered suitable because the Move Left and Move Right gestures involved the same hand just changing pointing in opposite directions. If moments were used, the moments output would be the same for both gestures.

The code fragment implementing the extraction of the chosen features from the blob are shown below:

```
//find biggest blob
int blobindex=0; int biggest = 0;
for (int zz = 1; zz <((WIDTH*HEIGHT)/2); zz++)
{//all sets from 1 , no set 0
      if (setofseq[zz]==NULL)
            break;   //save time by not searching empty sets
      if ((setofseq[zz]->total) > biggest){
            biggest = setofseq[zz]->total;
            blobindex = zz;
      }
}
```

```
//find the center of mass of the blob
int centerx=0, centery=0;
int minx=99999, miny=99999, maxx=0, maxy=0;

for (int w=0; w < setofseq[blobindex]->total; w++)
{
      CvPoint *eachpt=(CvPoint*)cvGetSeqElem(setofseq[blobindex],w);

      //get total x's and total y's
      centerx=centerx+eachpt->x;
      centery=centery+eachpt->y;

      //find maxx,maxy,minx,miny
      if ((eachpt->x) > maxx)
            maxx = eachpt->x;
      if ((eachpt->y) > maxy)
            maxy = eachpt->y;
      if ((eachpt->x) < minx)
            minx = eachpt->x;
      if ((eachpt->y) < miny)
            miny = eachpt->y;
}
//compute the center of mass
centerx=centerx/setofseq[blobindex]->total;
centery=centery/setofseq[blobindex]->total;

//draw cross lines on the center of mass
cvLine(imageres, cvPoint(centerx-10,centery),
      cvPoint(centerx+10,centery),CV_RGB(255,0,0),1,8);
cvLine(imageres, cvPoint(centerx,centery-10),
      cvPoint(centerx,centery+10),CV_RGB(255,0,0),1,8);

//draw red rectangle enclosing the blob
cvRectangle(imageres,cvPoint(minx,miny),
      cvPoint(maxx,maxy),CV_RGB(255,0,0),1);

//rest of the code ….
```

4.2.7 Implementation of Heuristic Rules and Output

Based on the features extracted as discussed above, the following heuristic rules for gesture recognition were easily set out:

- For the Move Left gesture, the longest distance from the center of mass is from the left-most pixel of the hand (the tip of the forefinger) to the hand's center of mass.
- For the Move Right gesture, the longest distance from the center of mass is from the right-most pixel of the hand (the tip of the forefinger) to the hand's center of mass.

However, preliminary testing with the Move Forward and Stop gestures revealed that just getting the longest distance from the center of mass to the top-most pixel (tip of the forefinger) alone was not enough to distinguish between the two gestures, since both gestures have the fingers pointing upwards or in the same direction. The following additional heuristic rules were constructed to remedy this problem:

- If the longest distance from the hand's center of mass points upward and only one finger is detected, then the gesture is interpreted as Move Forward.
- If the longest distance from the hand's center of mass points upward and more than one finger is detected, then the gesture is interpreted as Stop.

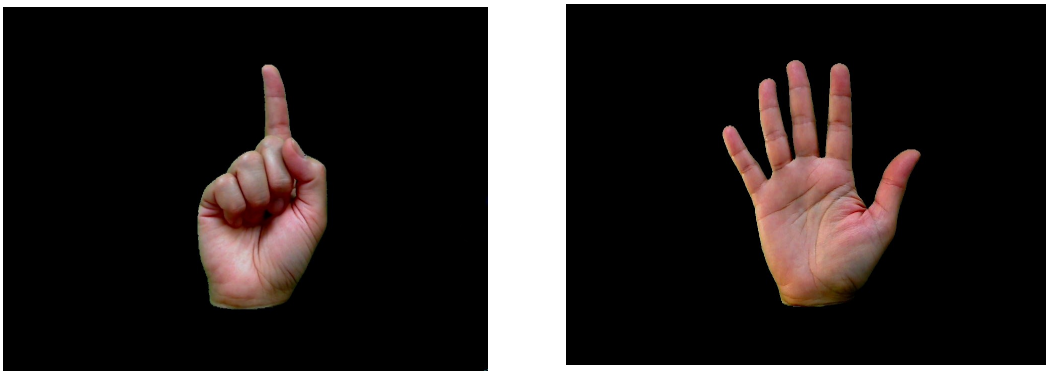
The code fragment implementing the heuristic rules and providing the recognized gestures as output are shown below:

```
//compute distances from the center of mass:
// d1(top), d2(left), d3(right), d4(bottom)
double d1=centery-miny;
double d2=centerx-minx;
double d3=maxx-centerx;
double d4=maxy-centery;

//normalize the distance values proportionally
d2=(d2/d1);
d3=(d3/d1);
d4=(d4/d1);
d1=(d1/d1);

//use an array to find the maximum distance
```

```
double distances[4] = {d1,d2,d3,d4};
double maxidist=0;
int maxiIndex;
for(int j=0; j<4; j++){
      if(distances[j] > maxidist){
            maxidist = distances[j];
            maxiIndex = j;
      }
}
//implement the heuristic rules
if (maxiIndex == 1)
      //d2 is maximum and left-most pixel is longest
      printf("Move left. \n");

else if (maxiIndex == 2)
      //d3 is maximum and the right-most pixel is longest
      printf("Move right. \n");

else
{     //d1 is maximum and the top-most pixel is the longest
      //select a line of pixels below the top-most pixel
      int testy = ((centery - miny)/3) + miny;

      //count the changes in pixel value along the line as edges
      int edge=0;
      for(int k=minx; k < maxx; k++){
            if(pixel(imageres, k, testy) != pixel
            (imageres, k+1,testy))
            {    edge++; }
      }
      if(edge == 2) //2 edges mean only one finger
            printf("Move Forward. \n");
      else
            printf("Stop. \n");

       //show position of the test line as a red line
      cvLine(imageres, cvPoint(minx,testy),
            cvPoint(maxx,testy),CV_RGB(255,0,0),1,8);
}
//rest of the code ...
```

4.3. Step 3: Preliminary Testing of Program Functions Using Static Images

Figure 12 shows all the chosen gestures with their corresponding control commands for the robot after colour segmentation into skin colour and black.



(a) Move Forward (b) Stop

(c) Move Left (d) Move Right

Figure 12. Images of hand gestures and their corresponding commands. (a) Move Forward. (b) Stop. (c) Move Left. (d) Move Right

Figure 13 shows the skin coloured portions converted to white.

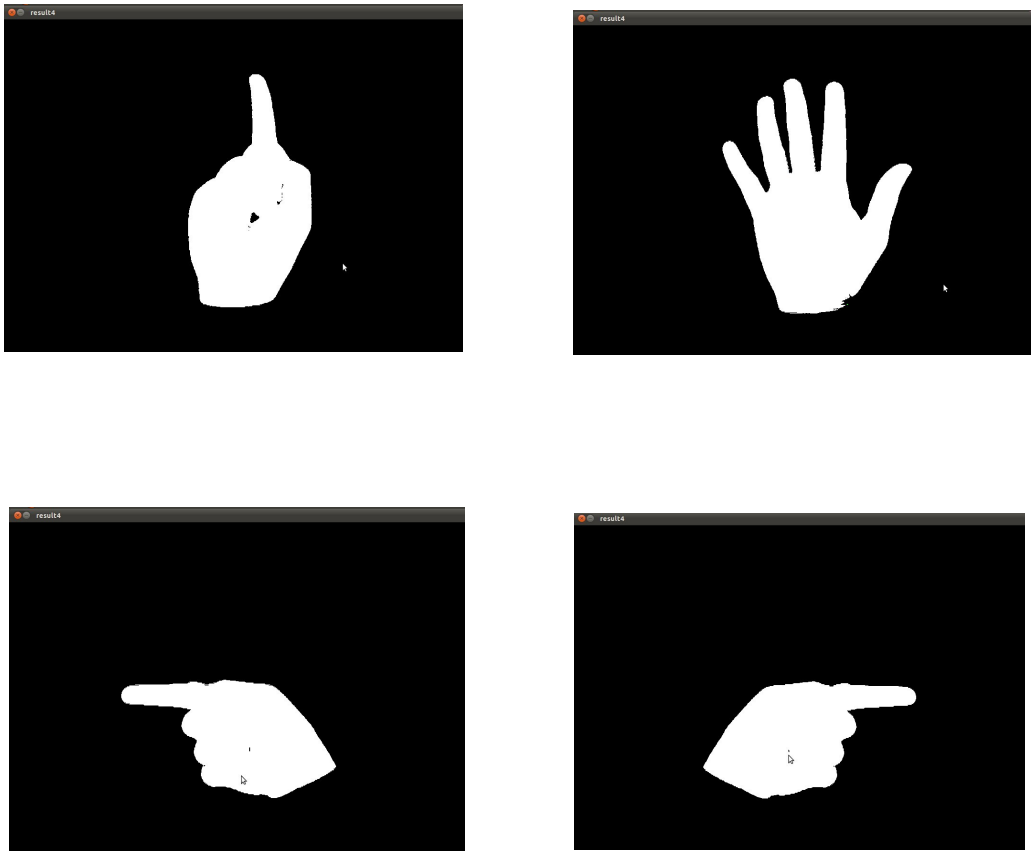
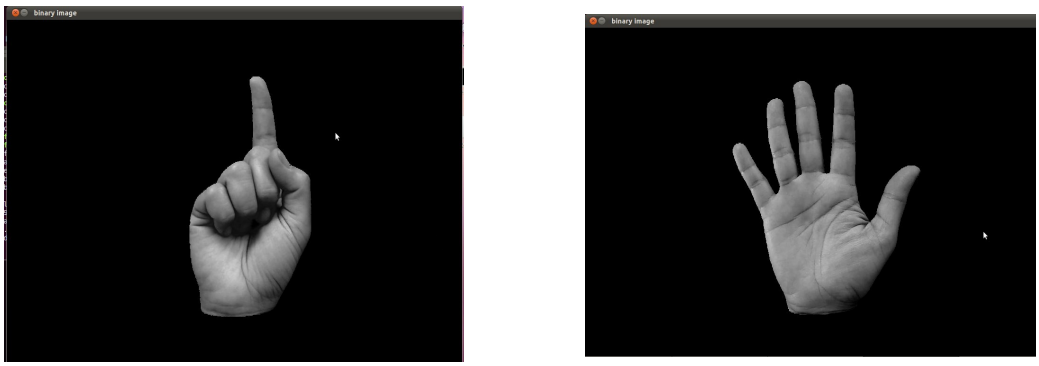


Figure 13. Images of gesture coloured white against black background.

Figure 14 shows the colour segmented images converted to grayscale.



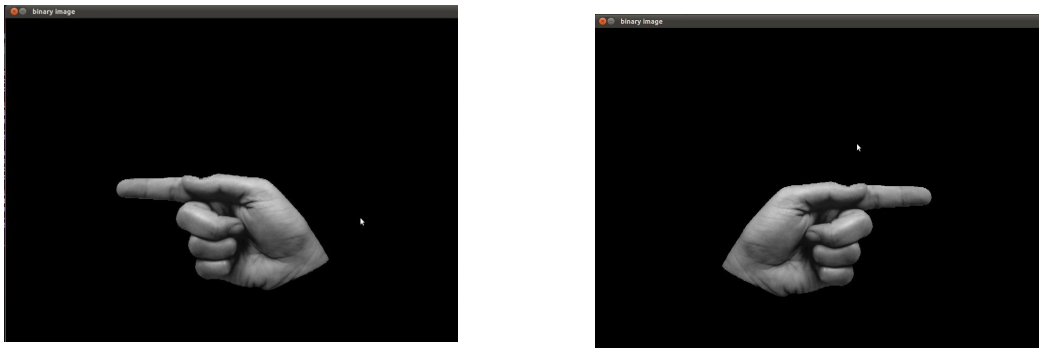


Figure 14. Grayscale images of the gestures.

Figure 15 shows the blobs detected from the grayscale images.

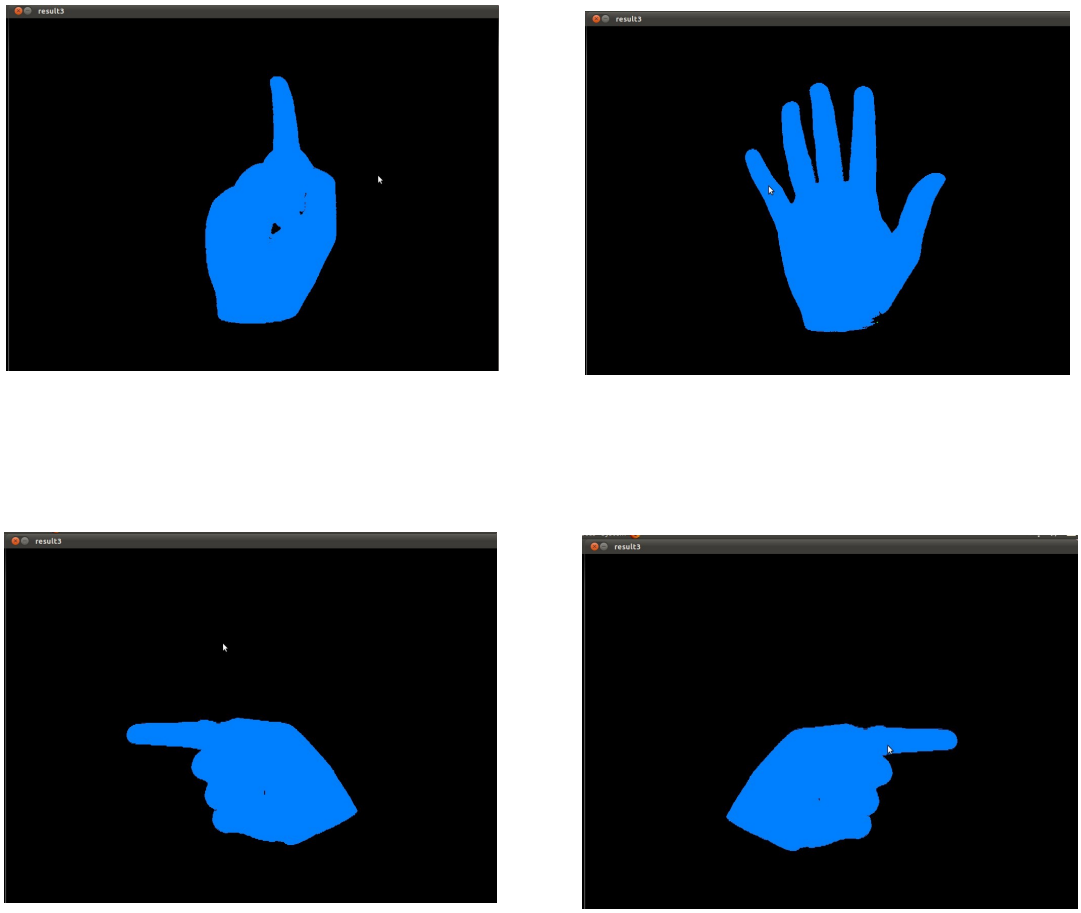


Figure 15. Images showing blue-coloured blobs of the gestures.

Figure 16 shows the images after the blobs are re-coloured as white and after feature extraction, identification of the center of mass, and enclosure of the blobs in red rectangles to illustrate the minimum and maximum distances of blob edges from the center of mass.

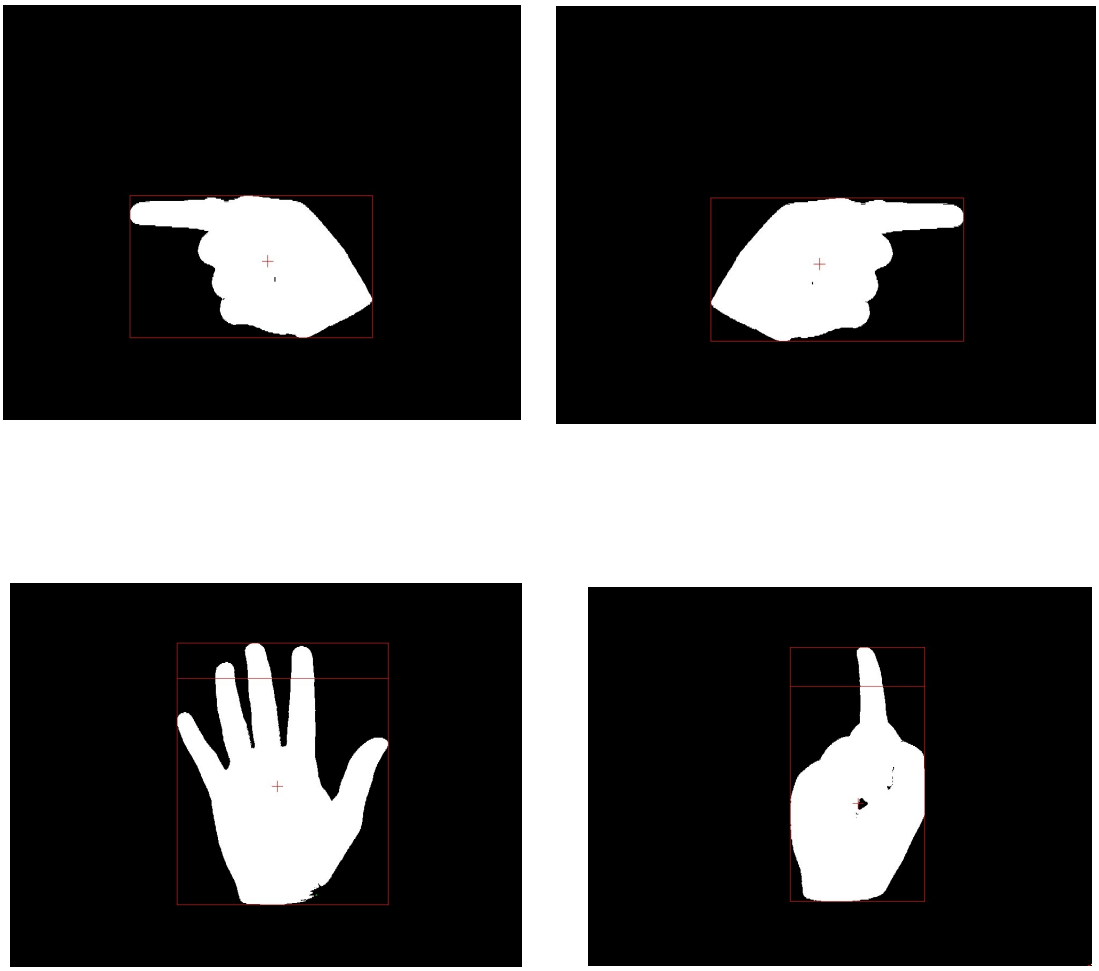
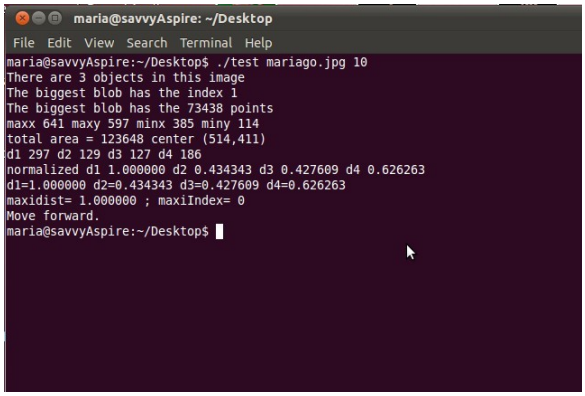
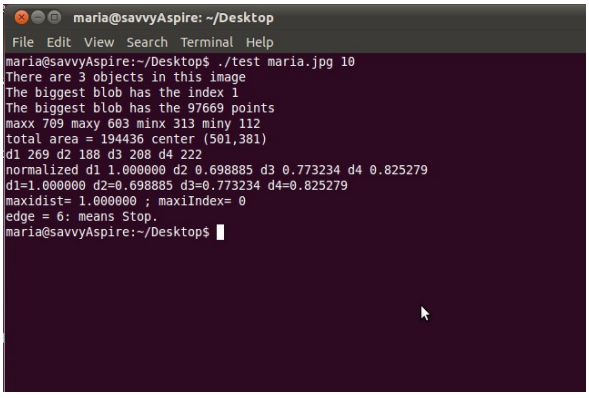


Figure 16. Images of blobs with the center of mass marked by red cross and enclosed in red rectangles.

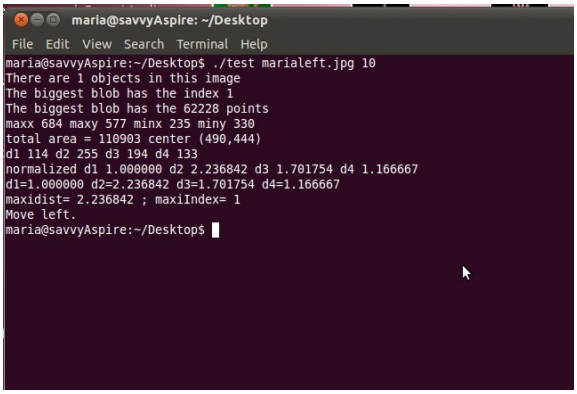
Figure 17 below shows the output of the program on the terminal window for each of the gestures.



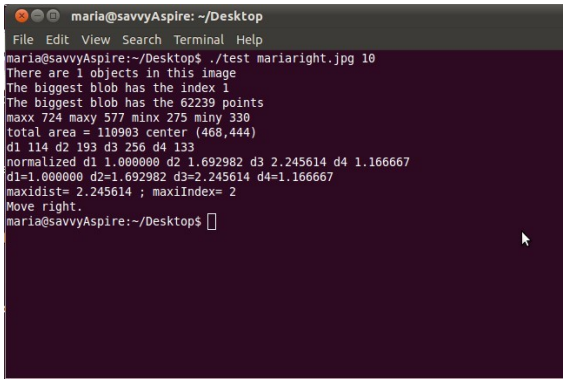
(a)



(b)



(c)



(d)

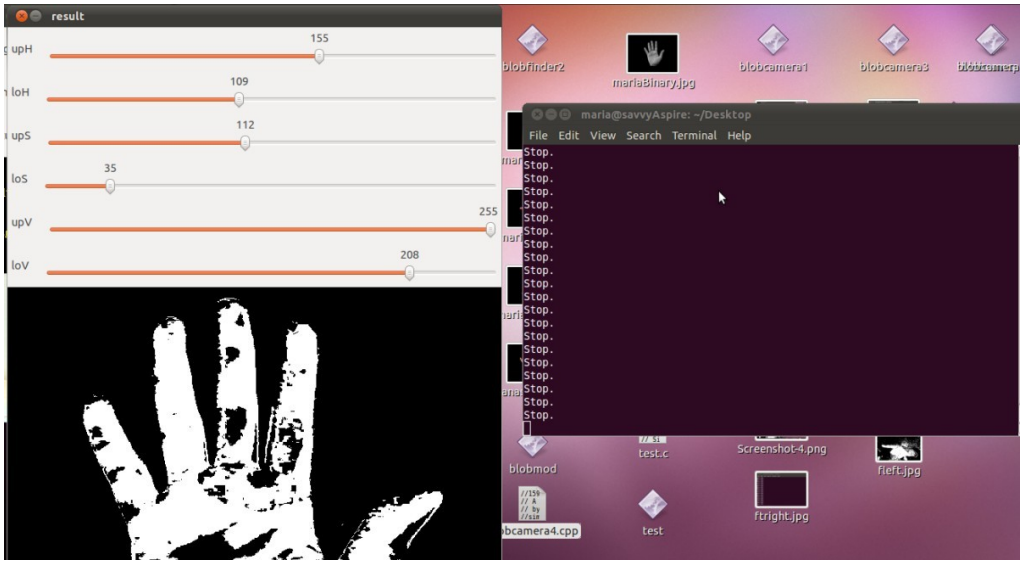
Figure 17 Images showing the program output. (a) Move Forward. (b) Stop (c) Move Left. (d) Move Right

4.4 Step 4. Testing the Gesture Recognition Program with Video Images

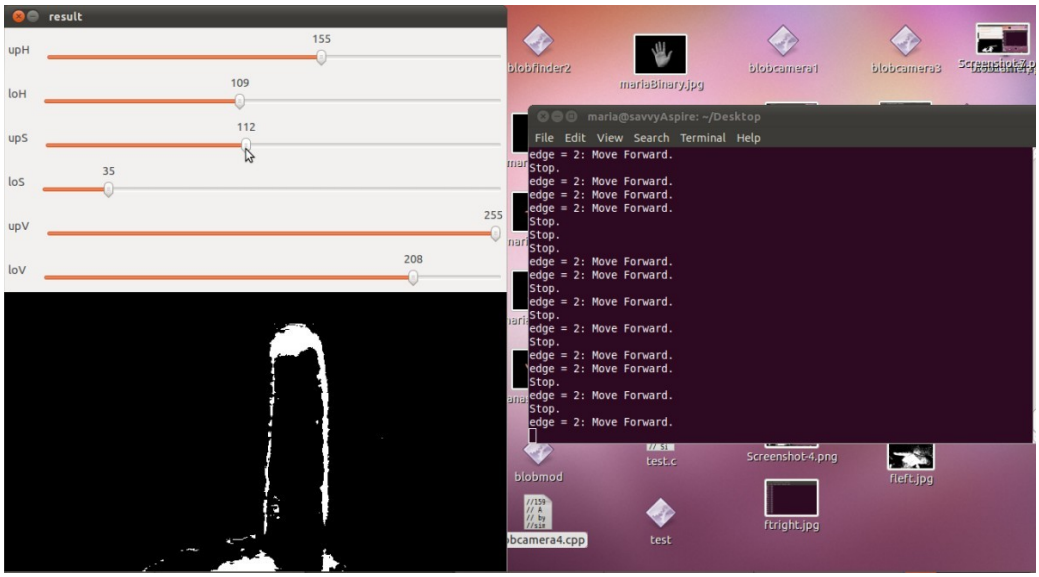
After the program showed good results with the static images, the program was modified to work with video capture of images, using the OpenCV functions as discussed previously.

When the hand gestures were made before the webcam, the program successfully implemented the OpenCV trackbars, the segmentation of images into black and white, the detection of blobs, the recognition of the images, and the output of the control commands.

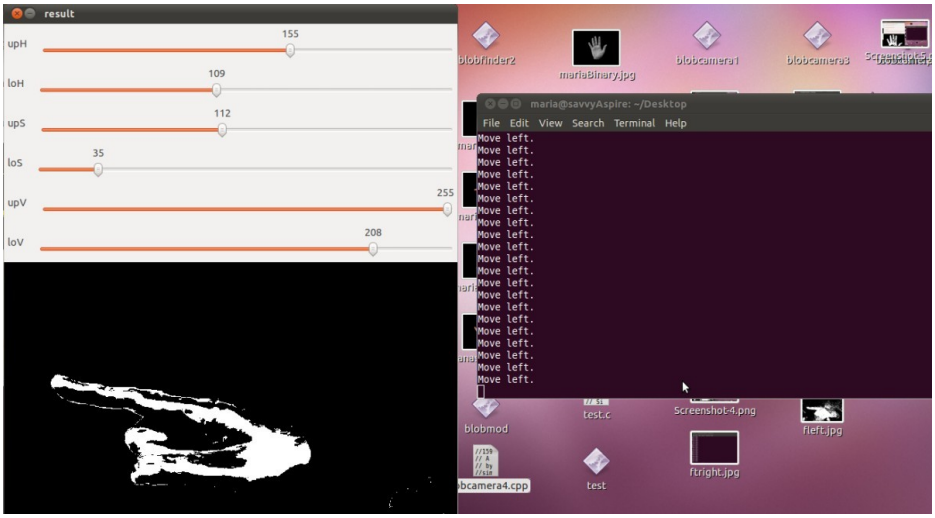
Figure 18 shows the resulting video images for all the gestures and the corresponding output for image recognition as shown by the images of the corresponding terminal windows.



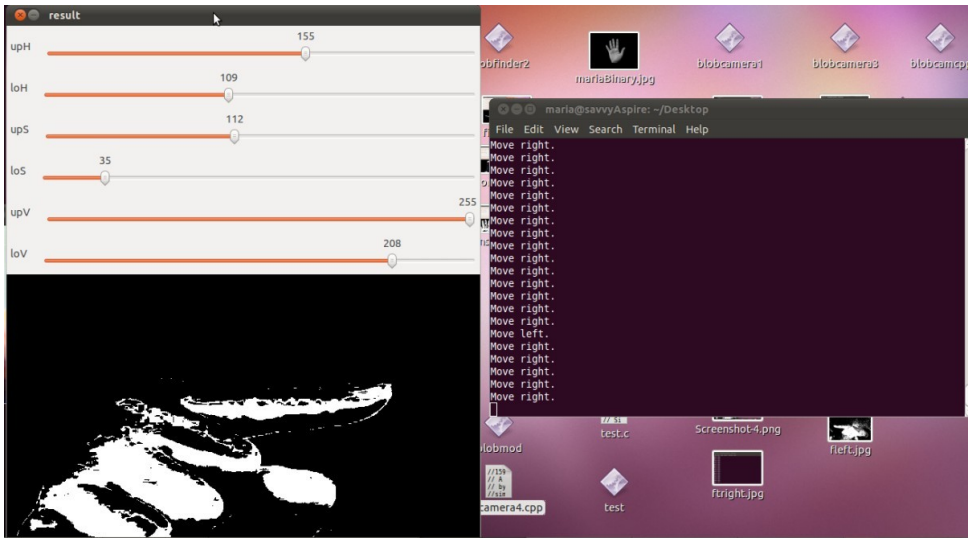
(a)



(b)



(c)



(d)

Figure 18. Webcam images with output on terminal. (a) Stop gesture and output command. (b) Move Forward gesture and output command. (c) Move Left gesture and output command. (d) Move Right gesture and output command.

5. Discussion and Conclusion

The results also showed that the gesture recognition application was quite robust for static images. However, the video version was enormously affected by the amount of illumination, such that is was necessary to check and adjust the HSV values for skin colour when starting the program to get the proper output. Sometimes the adjustment was difficult to do because of the lighting conditions and the amount of objects in the background.

The application was very susceptible to noise on the video stream. Slight hand movements could affect gesture recognition. Nevertheless, if the hand is steady enough for long enough, the program outputs the correct command.

It was also observed that while the program was executing there were memory leaks. Attempts to remedy the problem were made by using the OpenCV functions to release memory. Despite this, the leaks continued. Perhaps the leaks were due to the implementation of OpenCV functions for the sequences behind the scenes.

For integrating the program with the robot in the future, it would be necessarily to consider other output such as speed or velocity as part of the navigational control commands.

Based on the results, a computer vision application could detect and recognize simple hand gestures for robot navigation using simple heuristic rules. While the use of moment invariants was not considered suitable because the same gestures could be used pointing in opposite directions, other learning algorithms like AdaBoost could be explored to make the program more robust and less affected by extraneous objects and noise.

Bibliography

[1] Md. Hasanuzzaman, T. Zhang, V. Ampornaramveth, H. Gotoda, Y. Shirai, and H. Ueno, "Adaptive visual gesture recognition for human-robot interaction using a knowledge-based software platform", *Robotics and Autonomous Systems*, vol. 55, Issue 8, 31 August 2007, pp. 643-657. http://dx.doi.org.ezproxy.massey.ac.nz/10.1016/j.robot.2007.03.002

[2] R.C. Gonzales and R.E. Woods, *Digital Image Processing*, Prentice Hall, 2008.

[3] M. Sonka, V. Hlavac, and R. Boyle, *Image Processing, Analysis, and Machine Vision*, PWS Publishing, 1999.

[4] R. Szeliski, *Computer Vision: Algorithms and Applications*, Springer, 2011.

[5] S.E. Umbaugh, *Digital Image Processing and Analysis*, CRC Press, 2011.

[6] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, Prentice Hall, 2010.

[7] OpenCv, http://www.willowgarage.com/pages/software/opencv, accessed 11 May 2011.

[8] A.L.C. Barczak, *Computer Vision: Study Guide*, IIMS, Massey University, 2010.

[9] G. Bradsky and A. Kaehler, *Learning OpenCV: Computer Vision with the OpenCV Library*, O' Reilly, 2008.

[10] J.R. Parker, *Algorithms for Image Processing and Computer Vision*, Wiley Publishing, 2011

[11] R. Mukundan and K.R. Ramakrishnan, *Moment Functions in Image Analysis*, World Scientific, 1998.

[12] M.K. Hu, "Visual pattern recognition by moment invariants," *IRE Transactions on Information Theory*, vol. 8, pp. 179-187, 1962.

[13] Y. Freund and R.E. Schapire, "A short introduction to boosting," *Journal of Japanese Society*

for Artificial Intelligence, vol. 14, no. 5, pp. 771-780, 1999.

[14] "IIMS Robotics Lab: Omni-Robot Manual and References," IIMS, Massey University, 2010.

[15] S.K. Mathanker, P.R. Weckler, T.J. Bowser, N. Wang and N.O. Maness, "AdaBoost classifiers for pecan defect classification," *Computers and Electronics in Agriculture*, vol. 77, pp. 60-68, 2011.

[16] Chieh-Chih Wang and Ko-Chih Wang, "Hand posture recognition using Adaboost with SIFT for human robot interaction," *Recent Progress in Robotics: Viable Robotic Service to Human, Lecture Notes in Control and Information Sciences*, vol. 370, pp. 317-329, 2008.

List of Figures