Model-Based Hand Posture Estimation Using Monocular Camera

Liwen Zhang

A thesis
submitted in partial fulfillment of the
requirements for the degree of

Master of Science

University of Washington

2012

Committee:

J. Nathan Kutz, Chair

Ulrich Hetmaniuk

Program Authorized to Offer Degree:
Applied Mathematics

University of Washington

Abstract

Model-Based Hand Posture Estimation Using Monocular Camera

Liwen Zhang

Chair of the Supervisory Committee:
Professor J. Nathan Kutz
Department of Applied Mathematics

This work studies the problem of model-based hand posture estimation via monocular camera. Generally, a model-based posture estimation method manipulates a 3D hand model whose posture is determined by a set of parameters. It looks for parameters that align the model with hands in images best. In this thesis, the method for building hand model from truncated quadrics, algorithms for silhouette and edge extraction are studied and implemented into an application of hand posture detection, which is to recognize and locate given postures in real images. Moreover, as an attempt to recover hand articulation, a multidimensional scaling (MDS) based approach is tried out. By MDS, posture templates are lain out in an embedding space where the intrinsic dimensions of templates appear to be recovered. The accuracy of articulation estimation is shown through experiments. And the discussion of future work for improving the current approach follows.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGMENTS

I would like to express my deepest gratitude to the faculty, staff, and students of the Department of Applied Mathematics, University of Washington. I have got the chance to learn a variety of interesting academic problems in various fields from those excellent courses, miscellaneous talks and people around when studying in this program. Those problems have greatly broadened my mind and stimulated my curiosity, which added to my desire for further study. Especially, I would like to thank Professor Nathan Kutz for introducing me to the field of dimensionality reduction. It was during the discussion with Nathan and a few amazing students - Jacob Grosek, Andrew Scott Barr and Ian Talarico - that I began to learn the problem of hand gesture recognition which is highly relevant to this thesis.

I would also like to express my appreciation to the Graduate Program Coordinator, Professor Bernard Deconinck, and Graduate Program Assistant, Shari Jacobs, for offering me many precious advices and assistance during my time of study toward this degree.

To my parents, Yong and Lingdi, who have provided invaluable support and encouragement to my study, I am grateful forever.

DEDICATION

to my family

Chapter 1

INTRODUCTION

The problem of hand posture estimation has been studied over decades. A variety of approaches have been taken to solve this problem. Glove-based approaches, which require users to wear a glove or some markers on their hand, have demonstrated precise capture of hand posture in real-time application. But, gloves and markers can be cumbersome to use and might not be suitable for introducing posture recognition into daily usage. Instead, visual input could be a more natural interface. Ideally, if a computer is trained to recognize human's postures captured by a camera, it would be able to communicate with us more efficiently. Vision based approaches utilize one or a set of cameras as input devices. Images of hand is processed and analyzed for recognizing the hand guesture. This noncontact interface is undoubtly easier to use. But it raises lots of challenges for designing a satisfying algorithm of recognizing the gesture, as the inputs are merely 2D images of a 3D object. In some applications, the algorithms are designed for recognizing hand guesture from a set of candidate postures while in some others, people intend to reconstruct the full 3D articulation of hand postures. A common method used in the later type of application is to generate a sythesized 3D hand model and project it onto a 2D plane. Then the parameters of the 3D model is adjusted in order to align its 2D projection with the hand in real images. This approach is usually called the model based approach. In this work, model based approach, especially, model based pose recognition via monocular camera is investigated.

The estimation of hand posture from a single image may involve the following steps: 1) construct a 3D hand model. 2) extract features from hand images. 3) search for parameters that minimize the "difference" between the model and the image. Loosely speaking, the first two steps are to prepare a hand model and an objective function which depicts the

dissimilarity between the model and a real image based on features extracted. The last step tries to search for best configuration of the model that minimizes the objective function so that the 3D articulation of the hand can be approximated. This work studies a few methods that people have been using for the first and second steps. Combined with a relatively simple searching algorithm, these methods are later implemented in an application for detecting some particular postures in images. Besides, as an attempt to solve the problem of posture configuration estimation, an algorithm based on multidimensional scaling, is tried out. The feasibility of the algorithm is tested through experiments.

Chapter 2
HAND MODEL

By nature, human hands are very flexible and are able to perform lots of complicated gestures. To accurately articulate the posture of a hand, we need resort to hand model. The construction of a hand model could be divided into two portions: 1) determine the skeletal model, which is to decide the structure of a hand, the spatial relation among palm, fingers and their degrees of freedom. 2) determine the surface of a hand, which decide the appearance of the hand. In this chapter, the models that people have used will be discussed briefly first. Then, the method for constructing a hand model from truncated quadrics will be described in details.

2.1 Skeletal Model and Surface Model

2.1.1 Skeletal Model

The skeletal model is used for describing the spatial structure of a hand, especially how a hand could be deformed and what is the limitations of movement[16, 17, 36]. A common form of a hand skeleton is shown in Fig.2.1. The skeleton consists of bones and joints. In the rest of this thesis, fingers will be numbered from (I) to (V) with (I) for the thumb and (V) for the pinky. The joints on each finger will be numbered from 1 to 3 with 1 for the joint connecting a finger to the palm and 3 for the joint nearest to tips.

A hand has 16 joints, 3 for each finger and 1 for the wrist, each of which has different degrees of freedom in rotation. A hand posture could be determined by specifying values of rotation angles of these joints. According to the type of movement or possible rotation axes, joints are classified as flexion, twist, directive, or spherical. In [16], 27 degrees of freedom are applied totally, including six degrees of freedom for global possition and orientation.

Joints 2 and 3 on fingers (II) to (V) and joint 3 on finger (I) are classified as flexion which has a single degree of freedom. The rest joints on fingers are classified as directive which has 2 degrees of freedom as they permit flexion movement in two directions. The wrist joint is classified as a spherical joint which has 3 degrees of freedom and permits simultaneous directives and twist movements. The movement of a wrist is counted as a part of global rotation and position.

Although the hand has extensive flexibility, the movement of fingers still has certain constraints. As studied in [16, 17], the constraints on fingers can be categorized into different types: static constraints, dynamic constraints and natural motion constraints. Static constraints describes the limitation of the ranges of finger rotation angles without making assumption about pose and motion of a hand. It refers to the largest/least possible rotation angle that a joint is able to perform. For example, a finger could be hardly bended backwards and the angles of adduction and abduction performed by joint 1s(joints that connect a finger to the palm) have certain bounds. Dynamic constraints refer to intrafingers or interfingers constraints during finger motions. For example, one could barely flex the middle finger without flexing the ring finger at the same time. Also, the ranges of adduction and abduction of finger (II) to (V) decrease significantly when the angles of flexion increase. The third type of constraints, natural motion constraint, is based on the observation that, when people switch their hand posture from one to another, the movement of the hand follows some rules statistically. An example raised in [17] is that: the most natural way for a person to make a fist from an open hand would be curling all the fingers at the same time instead of curling one finger at a time. Not like previous types of constraints, this type of constraint is rather statistical. And it only applies to natural hand motions. Yet, in the situations where this type of constraints holds, the dimension of posture space could be reduced further by conducting principle component analysis (PCA), as proposed in [17].

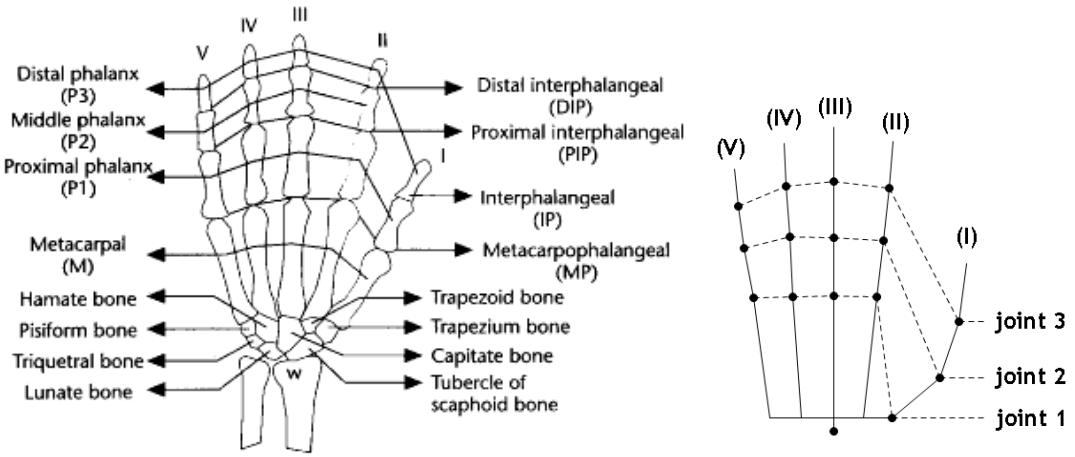

Figure 2.1: Examples of hand skeletal model. Left: the hand skeletal model appears in [16]. Right: a simplified skeletal model and labels of fingers and joints to be used in this thesis.

2.1.2 Surface Model

Different hand surface models have been proposed in the literature. The surface can be modeled with a set of quadrics (cones, cylinders and ellipsoids)[24, 13, 25], a simplex mesh[11], or a triangulated mesh[7]. The selection of surface model would decide the type of features available form the hand model. A model constructed with quadrics may generate features such as edge and silhouette. A triangluated surface can produce more features such as illumination and texture. In the next section, a hand model built from truncated quadrics, as proposed in [24] will be introduced.

2.2 A Hand Model Constructed with Truncated Quadrics

In this part, the method raised in [24] for constructing a hand model by assembling truncated quadrics is adapted and will be used in experiments later. To reduce the complexity, a few additional constraints and assumptions are made when generating the hand model. First, the camera is supposed to be set at the origin of a 3D coordinate and facing towards a fixed direction. Then, the location of hand wrist is supposed to be always fixed at a certain position. 3 degree of freedom are assigned for the rotation of the hand about its wrist. 4

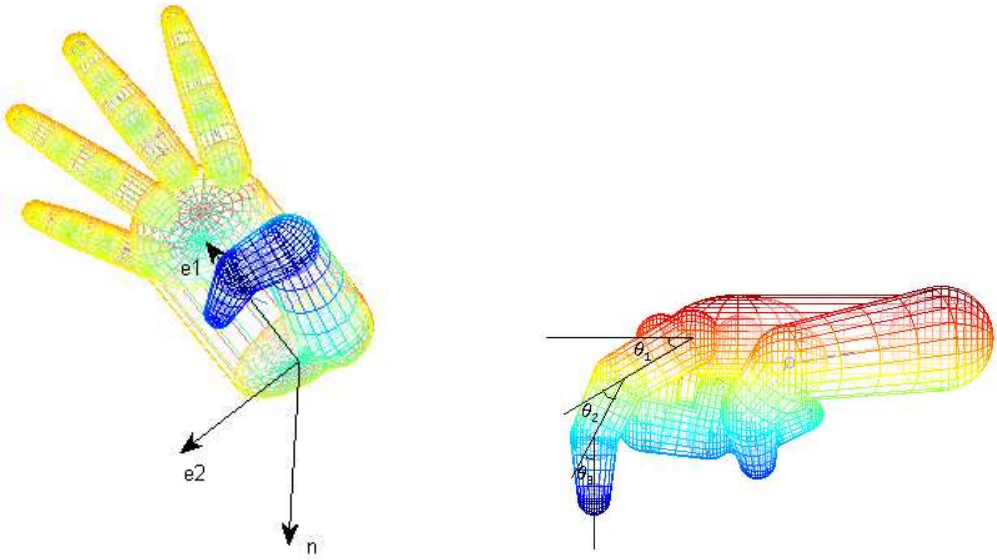

Figure 2.2: An illustration of hand model built up with quadrics.

degrees of freedom are assigned to each finger: the joints connecting fingers to the palm are assigned two degrees of freedom, one for flexion and another for abduction/adduction, while the rest joints are assigned one degree of freedom for flexion. In the rest of this thesis, a vector of variables representing rotation angles at joints will be denoted as $\boldsymbol{\theta}$. Three orthornormal vectors $\boldsymbol{e_1}$, $\boldsymbol{e_2}$ and $\boldsymbol{n}$ are used for representing the spherical rotation of the wrist. $\boldsymbol{e_1}$, $\boldsymbol{e_2}$ decide the plane of the palm. $\boldsymbol{e_1}$ indicates the upward direction of the hand, i.e. the direction that the middle finger points in when the hand is fully open and the middle finger performs no abduction, adduction and flexion. n is a unit vector orthogonal to the plane of the palm and pointing inward, as illustrated in Fig.2.2. Here, $\boldsymbol{\theta}$, e_1, e_2 and n are called configuration parameters. They are used to determine the locations of every joints on the hand.

2.2.1 Geometric Primitives

This subsection will explain how quadrics (ellipsoids, cones and cylinders) are built, represented and assembled into a hand model. The method for getting hand's contours by projecting the 3D model onto a 2D plane will be explained later. The work here follows [24] in building the hand model. [29] can be referred for relevant 3D geometry in computer graphics.

A point in three dimensions can be represented by inhomogeneous coordinates $\boldsymbol{x} = (x, y, z)^T \in R^3$. It can also be represented using homogeneous coordinates $\tilde{\boldsymbol{x}} = (\tilde{x}, \tilde{y}, \tilde{z}, \tilde{w}) \in P^3$, where vectors differ only by scale are considered to be equivalent. Besides, it is sometimes useful to denote a 3D point using the augmented vector $\bar{\boldsymbol{x}} = (x, y, z, 1) \in P^3$ with $\tilde{\boldsymbol{x}} = \tilde{w}\bar{\boldsymbol{x}}$.

Thus, 3D translations of a point $\boldsymbol{x}$ can be written as $\boldsymbol{x'} = \boldsymbol{x} + \boldsymbol{t} = \begin{bmatrix} I & t \end{bmatrix} \bar{\boldsymbol{x}}$, and the combination of 3D rotation and translation can be written as $\boldsymbol{x'} = R\boldsymbol{x} + \boldsymbol{t} = \begin{bmatrix} R & \boldsymbol{t} \end{bmatrix} \bar{\boldsymbol{x}}$, where R is a 3×3 orthonormal rotation matrix, $\boldsymbol{x'}$ and $\boldsymbol{x}$ are under inhomogeneous coordinates and $\bar{\boldsymbol{x}}$ is the augmented vector of $\boldsymbol{x}$.

To build a hand model, three types of quadrics are used: ellipsoids, cones and cylinders. These surface can be written in the form of $\bar{\boldsymbol{x}}^T Q \bar{\boldsymbol{x}} = 0$.

For ellipsoids centered at the origin, whose equation is $\frac{x^2}{w^2} + \frac{y^2}{h^2} + \frac{z^2}{d^2} = 1$,

$$Q = \begin{bmatrix} \frac{1}{w^2} & & & \\ & \frac{1}{h^2} & & \\ & & \frac{1}{d^2} & \\ & & & -1 \end{bmatrix};$$

For cones aligned with y-axis, whose equation is $\frac{x^2}{w^2} - \frac{y^2}{h^2} + \frac{z^2}{d^2} = 0$,

$$Q = \begin{bmatrix} \frac{1}{w^2} & & & \\ & -\frac{1}{h^2} & & \\ & & \frac{1}{d^2} & \\ & & & 0 \end{bmatrix};$$

For cylinders aligned with y-axis, whose equation is $\frac{x^2}{w^2} + \frac{z^2}{d^2} = 1$,

$$Q = \begin{bmatrix} \frac{1}{w^2} & & & \\ & 0 & & \\ & & \frac{1}{d^2} & \\ & & & -1 \end{bmatrix}.$$

Similarily, a pair of planes whose equation is $(y - y_0)(y - y_1) = 0$ can be written as $\bar{x}^T \Pi \bar{x} = 0$, where $\Pi = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & -\frac{y_0+y_1}{2} \\ 0 & 0 & 0 & 0 \\ 0 & -\frac{y_0+y_1}{2} & 0 & y_0 y_1 \end{bmatrix}$. Hence, the part of a quadric in between a pair of planes (a truncated quadric) can be written as: $\begin{cases} \bar{\boldsymbol{x}}^T Q \bar{\boldsymbol{x}} = 0 \\ \bar{\boldsymbol{x}}^T \Pi \bar{\boldsymbol{x}} \geq 0 \end{cases}$, where Q is the matrix representing the quadric and Π is a matrix representing a pair of clipping planes.

After the rotation angles $\boldsymbol{\theta}$ and vectors $\boldsymbol{e1}$, $\boldsymbol{e2}$ and $\boldsymbol{n}$, which determing the facing of palm, have been picked, the position of all joints and tips can be figured out. Then, different quadrics can be generated, rotated and translated onto designated positions to be assembled into a hand. In this work, the hand construction takes the idea from [24]. The palm is build with a cylinder and two ellipsoids at its two ends. Each joint on the fingers is built with an ellipsoid and trancated cones are used for building the phalanxes. Besides location, it is also necessary to know the size of every quadrics. We need to know values such as the length of each finger, the radius of joints and the size of the palm. These values will be called the calibration parameters. They are able to be measured by calibrating a real hand or, indirectly, by calibrate the image of an open hand.

2.2.2 Transformation of Quadrics

Given configuration and calibration parameters, quadrics can be made into a hand model by transforming and assembling to desired possitions. Fig.2.4 shows a hand model built from quadrics and its skeletal model. Suppose a transformation matrix T is applied on a quadric

whose equation is $\bar{\boldsymbol{x}}^T Q \bar{\boldsymbol{x}} = 0$, then, the transformed quadric satisfies this equation: $\bar{\boldsymbol{x}}^T \hat{Q} \bar{\boldsymbol{x}} = 0$, where $\hat{Q} = \left(T^{-1}\right)^T Q T^{-1}$ is also a symmetric matrix. Here, $\bar{\boldsymbol{x}}$ is under augmented coordinate, and $T = \begin{bmatrix} R & \boldsymbol{t} \\ 0 & 1 \end{bmatrix}$ where R and $\boldsymbol{t}$ are rotation and translation in inhomogenous coordinate.

Transformation of ellipsoids:

Ellipsoids are used at joints and the top and bottum ends of palm. If an ellipsoid, which is centered at the origin and has its axes aligned with the coordinates, needs to be centered at $\boldsymbol{p}$ and having its axes aligned in directions $\boldsymbol{e_w}$, $\boldsymbol{e_h}$and $\boldsymbol{e_d}$, then the transformation matrix should be $T = \begin{bmatrix} \boldsymbol{e_w} & \boldsymbol{e_h} & \boldsymbol{e_d} & \boldsymbol{p} \\ 0 & 0 & 0 & 1 \end{bmatrix}$.

Transformation of cones:

Truncated cones are used to represent phalanx. The centers and radius of two joints connected by a cone are needed to calculate the transformation matrix for a cone. The cone is truncated so that it is tangential to the ellispoids representing the joints. Suppose the centers of joints are $\hat{P}_1$ and $\hat{P}_2$, and their radius are R_1 and R_2. A cone shown in Fig.2.3 can be used as the building block. The cone should satisfies

$$\begin{cases} \bar{\boldsymbol{x}}^T Q \bar{\boldsymbol{x}} = 0 \\ \bar{\boldsymbol{x}}^T \Pi \bar{\boldsymbol{x}} \geq 0 \end{cases}$$

where

$$Q = \begin{bmatrix} \frac{1}{w^2} & & & \\ & -\frac{1}{h^2} & & \\ & & \frac{1}{d^2} & \\ & & & 0 \end{bmatrix}, \Pi = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & -\frac{h+g}{2} \\ 0 & 0 & 0 & 0 \\ 0 & -\frac{h+g}{2} & 0 & y_0 y_1 \end{bmatrix}, w = d = R_2 \sqrt{1 - \left(\frac{R_2 - R_1}{L}\right)^2},$$

$h = R_2 \left(\frac{L}{R_2 - R_1} - \frac{R_2 - R_1}{L}\right)$, $g = \frac{R_2 L}{R_2 - R_1} + \frac{R_1 (R_1 - R_2)}{L} - L$ and $L = \|\hat{P}_1 - \hat{P}_2\|$. Suppose $\phi = \frac{1}{2}\left(\frac{P_2 - P_1}{L} + \frac{\hat{P}_2 - \hat{P}_1}{L}\right)$, then $R = 2\phi\phi^T - I$ is a matrix that align $\overrightarrow{P_1 P_2}$ in the direction of $\overrightarrow{\hat{P}_1 \hat{P}_2}$. $\boldsymbol{t} = \hat{P}_1 - \left(2\phi\phi^T - I\right) P_1$ is the translation vector moving the cone into position. The

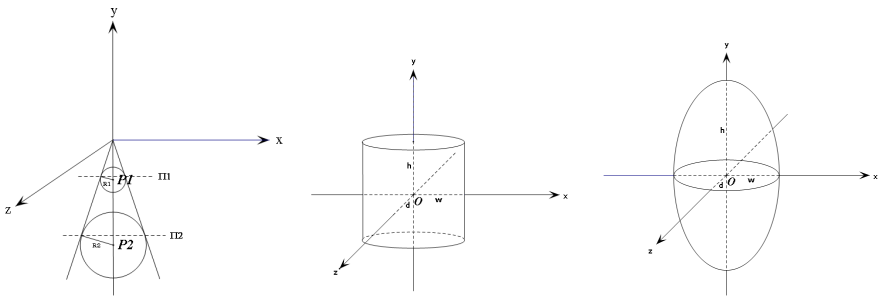

Figure 2.3: Samples of quadrics.

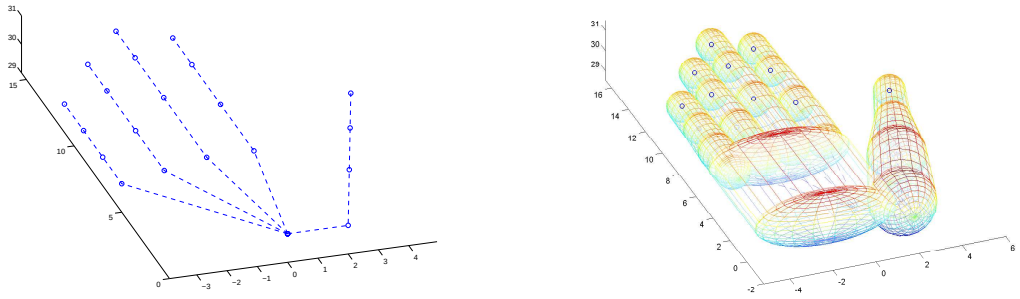

Figure 2.4: A sample skeletal model and a sample surface model. Left: A sample skeletal model. Right: A sample surface model built upon the skeletal model. The surface is constructed with truncated quadrics.

matrix fullfilling the complete transformation would be $T = \begin{bmatrix} R & \boldsymbol{t} \\ 0 & 1 \end{bmatrix}$.

Transformation of cylinder:

The main part of palm is constructed by a cylinder. The width w and height h of the cylinder is set according to calibration parameters. The depth d is set to the value that is just large enough to contain the MP joints of finger II to finger V in the cylinder. Given the center of palm $\hat{P}_0$ and configuration parameters $\boldsymbol{e_1}$, $\boldsymbol{e_2}$ and $\boldsymbol{n}$, the cylinder could be assembled onto the hand model by applying the transformation matrix $T = \begin{bmatrix} \boldsymbol{e_2} & \boldsymbol{e_1} & \boldsymbol{n} & \hat{P}_0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$.

2.2.3 Projection of Quadrics

Once the 3D hand model has been constructed, it can be projected onto a 2D plane to simulate the hand captured by a camera. As proposed in [24], suppose the camera is located at the origin $\boldsymbol{o}$, a ray starting from $\boldsymbol{o}$ and going through a point $\boldsymbol{x} \in R^3$ can be defined as $\tilde{\boldsymbol{x}} = \begin{bmatrix} \boldsymbol{x} \\ \xi \end{bmatrix}$, where $\xi \in R$ is the parameter of the ray and can be interpreted as inverse depth of a point. Each 3D point in the space defines such a corresponding ray. The projection of the 3D hand model is composed with interception of a plane and all such rays defined by points on the hand. As stated previously, after a quadric whose matrix representation is Q has been transformed by matrix T, it can be written as the equation $\bar{\boldsymbol{x}}^T \hat{Q} \bar{\boldsymbol{x}} = 0$, where $\hat{Q} = \left(T^{-1}\right)^T Q T^{-1}$. Then, the interception of the ray and the quadric can be obtained by solving the equation: $\tilde{\boldsymbol{x}}^T \hat{Q} \tilde{\boldsymbol{x}} = 0$. Partition $\hat{Q}$ into blocks as $\hat{Q} = \begin{bmatrix} A & \boldsymbol{b} \\ \boldsymbol{b}^T & c \end{bmatrix}$, the equation becomes $c\xi^2 + 2\boldsymbol{b}^T \boldsymbol{x} \xi + \boldsymbol{x}^T A \boldsymbol{x} = 0$. Hence, the outline of a quadric's projection consists of points that make the determinant of that equation zero, i.e. $\boldsymbol{x}^T \left(\boldsymbol{b}\boldsymbol{b}^T - cA\right) \boldsymbol{x} = 0$. Next, solve this equation for x.

Set $C = \boldsymbol{b}\boldsymbol{b}^T - cA$, then C is a real symmetric matrix and can be decomposed as $C = VDV^T$, where D is a real diagonal matrix whose diagonals are eigen values of C, V is an orthogonal matrix containing corresponding eigen vectors. Suppose the diagonals of D is sorted in a non-decreasing order. $\boldsymbol{x}$ is a vector of length 3. To get its projection onto 2D plane, we may consider $\boldsymbol{x}$ as a homogenious representation of a 2D point. Then, the set of points $\hat{\boldsymbol{x}}$ satisfying the equation $\hat{\boldsymbol{x}}^T D \hat{\boldsymbol{x}} = 0$ can be represented in parametric form. For exmaple, if the quadric is an ellipsoid, then the set of $\hat{\boldsymbol{x}}$ would form an ellipse which can be written as $\hat{\boldsymbol{x}}(t) = [a\cos(t), b\sin(t), 1]^T$, where $a = \sqrt{-\frac{d_{33}}{d_{11}}}$, $b = \sqrt{-\frac{d_{33}}{d22}}$, d_{ii} is the i-th component on D's diagonal. Then the solution of $\boldsymbol{x}^T \left(\boldsymbol{b}\boldsymbol{b}^T - cA\right) \boldsymbol{x} = 0$ would be $\boldsymbol{x} = V\hat{\boldsymbol{x}} = [x_1(t), x_2(t), \xi(t)]^T$, $\xi(t)$ represents the inverse depth from the camera in a 3D coordinate. Hence, its projection onto the plane $x_3 = 1$ would be $\left[\frac{x_1(t)}{\xi(t)}, \frac{x_2(t)}{\xi(t)}\right]$. To find out whether the projection of a point on the hand is visible, we may compare the depth

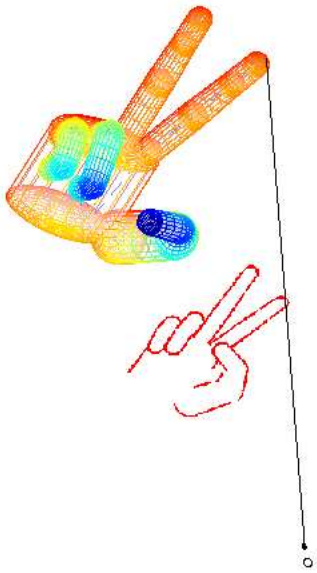

Figure 2.5: An illustration of projecting the outlines of a 3D hand model to a 2D plane.

of all interceptions of the hand quadrics and the ray defined by that point. After setting the visibility of points on the outline, the projection is accomplished. Fig.2.5 shows an illustration of the projection.

Given configuration parameters and calibration parameters, a hand model could be built and its 2D contour can be produced. By using the information of the contour, we may detect the hand with the same or similar posutre in an image. Fig.2.6 shows a few synthesized hand models and their projections onto a 2D plane. The outlines and edge map of these hands are produced with the method described above and the silhouettes of hands are generated by taking the area inside the outlines as foreground.

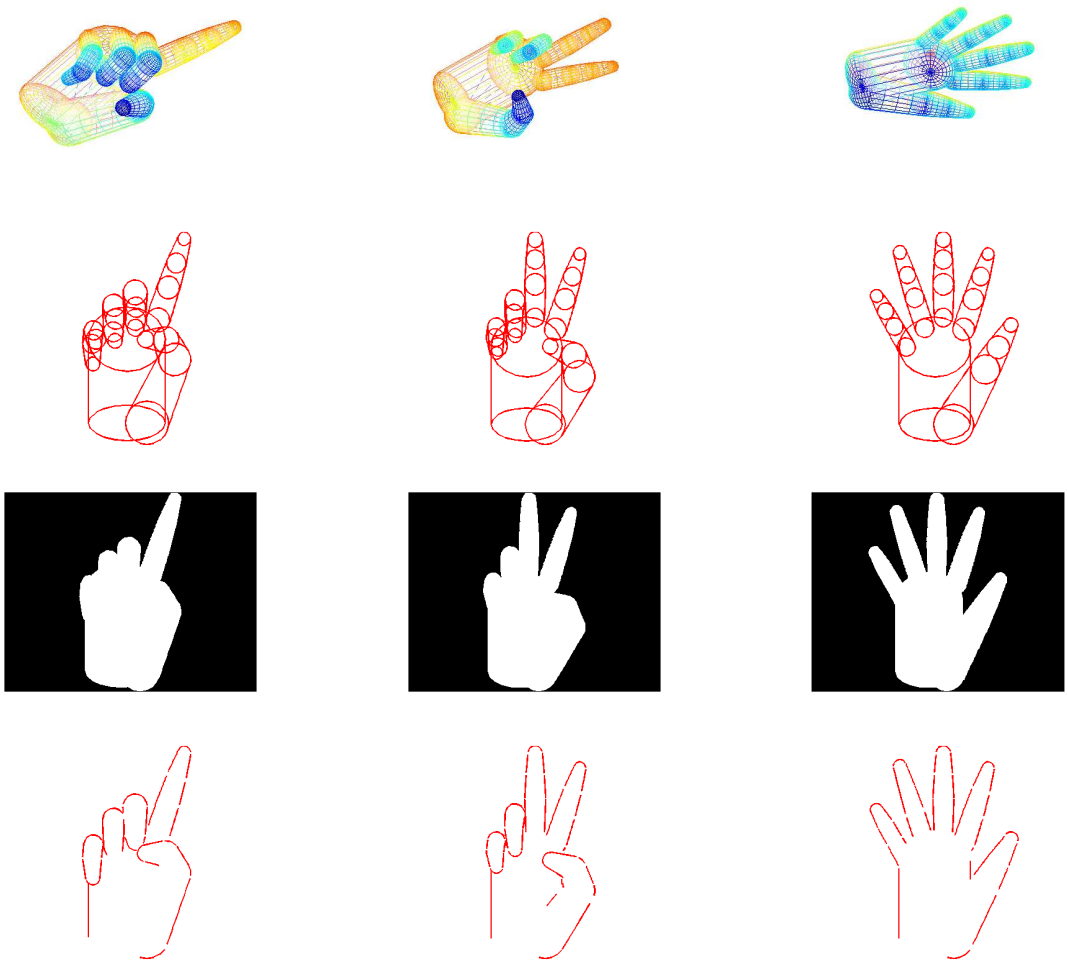

Figure 2.6: Some generated posture templates. The first row: 3D hand models. The second row: outlines of projections of all quadrics. The third row: Silhouette of hand's projection. The last row: edge map of the synthesized hand.

Chapter 3

FEATURE EXTRACTION AND MATCHING

3D hand models enable us to generate synthesized hand images. The synthesized hand will be matched with hands in images through features. Several different features have been proposed in literature, including edge, color, silhouette, shading, texture, etc. Unlike the problem of face recognition, hand posture recognition is much more difficult partly because hands have less landmarks for tracking. Hence, it is common that people may use more than one feature simultaneously in hand posture recognition. Two type of features are employed in this work: silhouette feature and edge feature.

3.1 Silhouette Extraction

A popular approach for extracting the hand silhouette and remove the background is to take advantage of the skin color [13, 24]. Usually, as a training step, hands in a picture is manually labeled, then the color of the hand is collected to form a distribution of skin color. During detection, the probability for a pixel to be a part of the hand is evaluated depending on its color and the learned skin color distribution. Cluttered background, fast motion of hand and the effect of different illumination can all add challenges and reduce accuracy of the detection.

In this work, a static background subtraction method will be employed. A static background subtraction is basically comparing the image containing the hand and the image without the hand. Here, the comparison is done in $L^*a^*b^*$ color space. In $L^*a^*b^*$ space, the L component reflects brightness while a and b reflects level of green/magenta and blue/yellow respectively. (Transformation between $L^*a^*b^*$and RGB color can be found in [29].) To conduct background subtraction, first, the image of background is taken and converted into

$L^*a^*b^*$ color space. Suppose the background image in $L^*a^*b^*$ space is P_0 and each pixel $P_0(i,j)$ have a vector value $(L_{i,j}, a_{i,j}, b_{i,j})$. Then, an image $P(i,j)$ containing hands are also converted into $L^*a^*b^*$ color space. Each pixel in the image is subtracted from the corresponding one in P_0 with L component neglected so that effect of brightness will be largely reduced. Pixels where color differ more than a preset threshold will be considered a pixel of the hand. Some examples are shown in Fig.3.1.

3.2 Edge Detection

Edge is a very important feature that has been widely used in hand pose estimation[27, 24, 13, 2, 31, 26, 28, 25]. Meanwhile, lots of researches have been conducted in edge detection. Methods such as Sobel filter, canny edge detector[4] and pb detector[1] are widely used in fields of computer vision and image processing. Here, canny edge detection is employed as it is fast in computation and possesses certain robustness. Canny edge detector was developed by John F.Canny in 1986 [4]. It aims to obtain good localization of the edge and increase detection rate of real edges while reducing responces to nonedge and noise. A brief introduction of Canny edge detector can be found at [34]. Canny edge detection contains four stages: noise reduction, gradient calculation, non-maximum suppression and hysteresis thresholding. During noise reduction, the image is convolved with a Gaussian filter to produce a slightly smoothed image. In the second stage, the magnitude and direction of gradient at every point is detected by applying operator such as Sobel and Prewitt on the image. The directions are discretize into four angles: 0°,45°,90° and 135°. During the next stage, every point at which gradient magnitude is greater than neighbours in the gradient direction is labeled as an edge point. After this stage, a set of edge points is obtained. During the last stage, hysteresis thresholding, two thresholds are required. An edge point whose magnitude of gradient is larger than the higher threshold will be marked as an edge pixel at once. An point whose magnitude of gradient is smaller than the lower threshold will be discarded immediately. For any of the rest points, it will be discarded unless it is connected to an edge pixel. Some results of edge detection are shown in Fig.3.2.

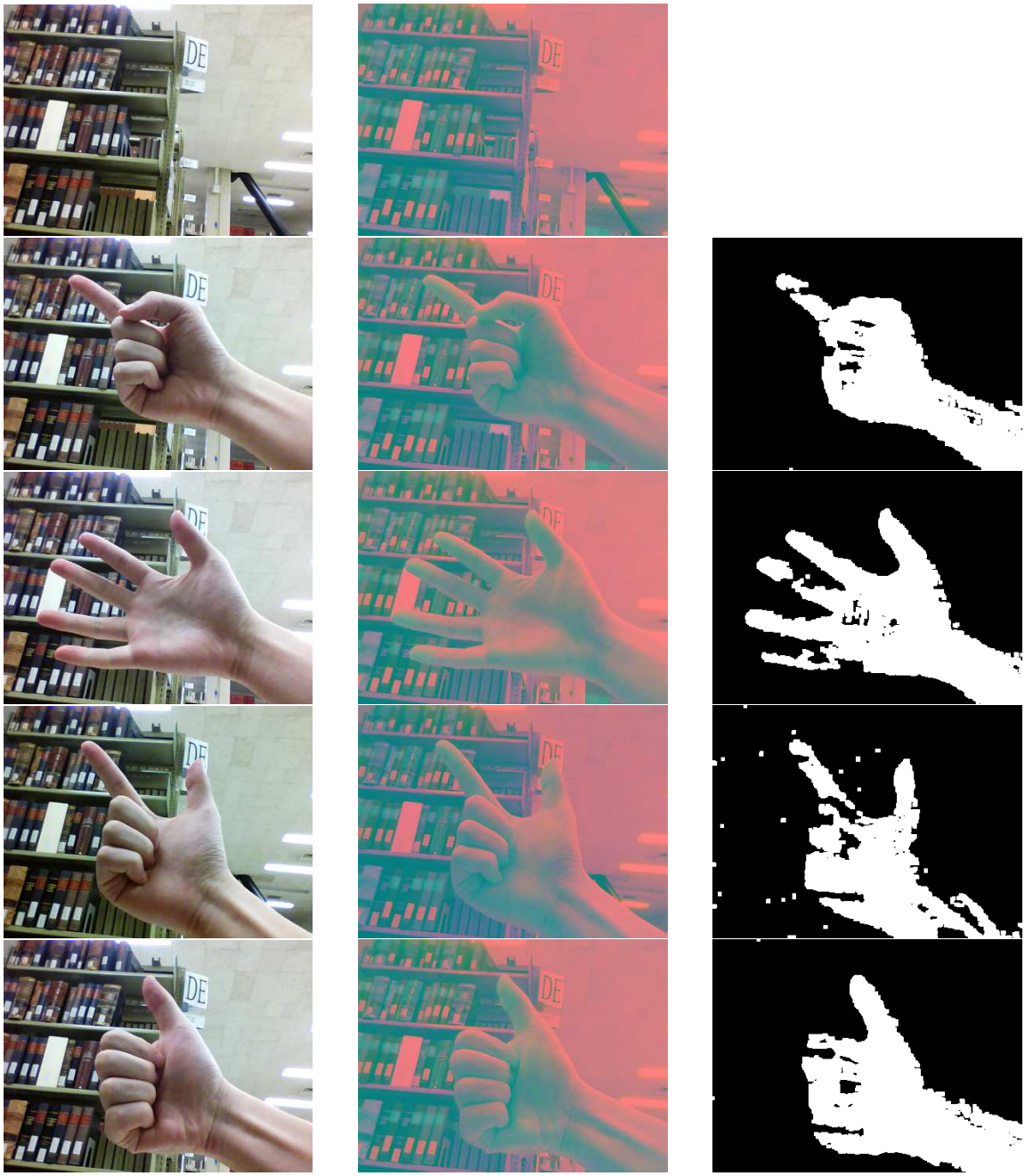

Figure 3.1: Extraction of hand silhouette. Left column: raw images. Middle column: images in $L^*a^*b^*$ color space (drawn as RGB images). Right column: binary images of extracted hand.

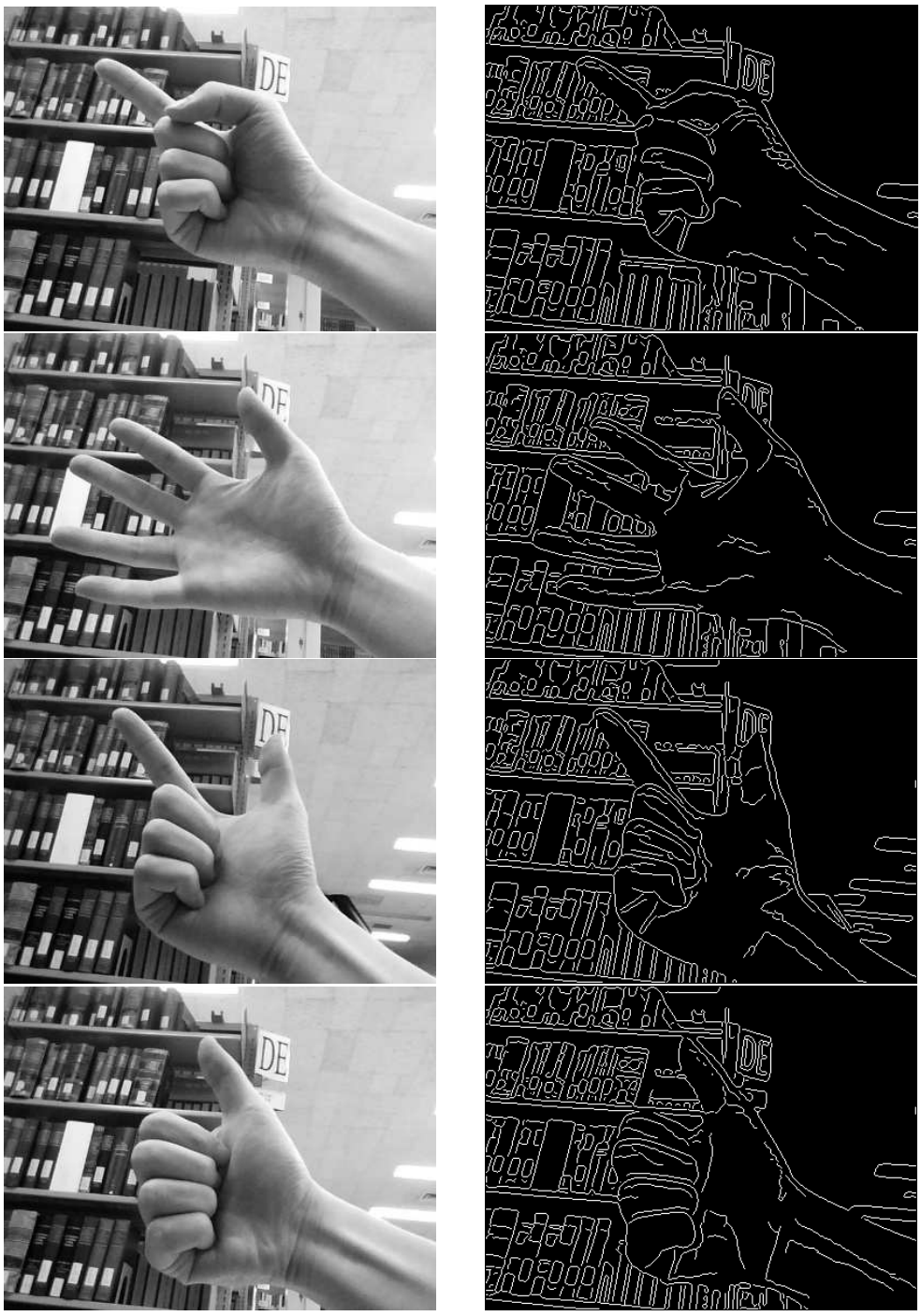

Figure 3.2: Examples of edge extraction. Left column: images converted into gray scale. Right column: edge map extracted from gray images on the left by applying Canny edge detector.

Chapter 4

POSTURE DETECTION

In this chapter, directional chamfer distance, which measures the degree of matching between two edge maps, is introduced first. Then, the hand model and feature extraction methods mentioned previously will be carried out for implementing posture detection in hand images.

4.1 Directional Chamfer Distance

A distance transformation is a basic morphological operation on binary images. It calculates the distance from every pixels on foreground to the closest boundary. Algorithms for computing distance transformation can be found in [29]. [5] introduces an algorithm for calculating chamfers distance transformation, which is an approximation to Euclidean distance transformation. People have been using chamfers distance as a matching metric for two curves. In model-based hand recognition, we may detect a hand in the image by comparing the contour of a hand model's projection, the template, with the edges detected in the image [13, 2].

Chamfer matching(CM) is a popular technique for finding the degree of alignment between two edge maps. As described in [20], suppose $U = \{u_i\}$ and $V = \{v_j\}$ are the sets of template and query edge maps, then the chamfer distance between U and V is given by:

$$d_{CM}(U, V) = \frac{1}{n} \sum_{u_i \in U} \min_{v_j \in V} |u_i - v_j|$$

To improve the performance of matching shapes in cluttered images, the technique of directional chamfer matching(DCM) [20] can be applied. Besides the distance in space, directional chamfer distance includes a term of edge's difference in orientation. For the same

sets of template and query edge maps, the directional chamfer matching is defined as:

$$d_{DCM}(U,V) = \frac{1}{n}\sum_{u_i \in U} \min_{v_j \in V} \left(|u_i - v_j| + \lambda|\phi(u_i) - \phi(v_j)|\right)$$

where λ is a weighting factor. A fast algorithm for computing directional chamfer matching is given in [20]. In the same article, an algorithm for converting an edge map to a line representation is provided. A line representation is a set of straight segments each of which is stored as a pair of end points. After converted to a line representation, the line direction of each pixel in the edge map is estimated. Precomputed line representation can speed up the calculation of DCM for two edge maps.

4.2 Experiments

This experiment aims to detect and locate two particular hand postures in real images. Silhouette and edge are used as features. A hand model is set to the objective posture and its projection to the 2D plane can be obtained with the method in Ch.2. From the projected hand, hand area and contour are stored as templates during the detection. Background subtraction mentioned in Ch.3 is applied to get silhouette information. And edge detection is done through Canny detector. In this work, the function "`edge`" available in Matlab's image processing toolbox is applied for implementing Canny detector. These two features are combined to give a measure of degree of matching between the template and the image. In this work, the detection process deals with 3 degrees of freedom, 2 for translation and 1 for in-plane rotation of the hand posture.

Suppose silhouette feature is stored as a binary image $S_{tpl} = (s_{ij})$ where 1s represent hand area and 0s represent the background. Suppose the edge map extracted from the template is represented in E_{tpl}. A sliding window in the query image will be used and image within the window will be compared with the template and a few rotated templates. Denote the silhouette and edge template rotated counter clockwise by an angle θ_{rot} as $S_{tpl}(\theta_{rot})$ and $E_{tpl}(\theta_{rot})$. Denote the image of background (image without the hand) as

$I = \left[I_{bg}^{(L*)}; I_{bg}^{(a*)}; I_{bg}^{(b*)}\right]$ and the hand image as $I_{hand} = \left[I_{hand}^{(L*)}; I_{hand}^{(a*)}; I_{hand}^{(b*)}\right]$. Then, by background subtraction, the binary picture representing hand area is calculated as:

$$I_{area} = \delta\left(\left(I_{bg}^{(a*)} - I_{hand}^{(a*)}\right).\hat{}2 + \left(I_{bg}^{(b*)} - I_{hand}^{(b*)}\right).\hat{}2 > T\right)$$

where $.\hat{}2$ is a component-wise square operation, $\delta(\cdot)$ is a component-wise characteristic function and T is a threshold whose value is set to 5 in this experiment. I_{area} will be a binary image with the same size as I_{bg} and I_{hand}. Denote the edge map extracted from the hand image as E_{query}. During the detection, a sliding window will be used for getting samples from the image. Suppose (x_s, y_s) , (x_e, y_e) are the positions of two diagonal corners of the window and the window has the same size as the template. Then the dissimilarity between image inside the window and a template rotated by θ_{rot} is defined as:

$$dis\,(x_s, y_s, x_e, y_e) = \\ -\,log\left(\frac{\|S_{tpl}(\theta_{rot}).*I_{area}(x_s:x_e, y_s:y_e))\|_F^2}{\|S_{tpl}(\theta_{rot})\|_F^2}\right) \cdot DCM\left(E_{tpl}(\theta_{rot}), E_{query}(x_s:x_e, y_s:y_e)\right)$$

where $.*$ is the component-wise multiplication, $DCM(\cdot,\cdot)$ is calculating the directional chamfer distance between two edge maps and $\|\cdot\|_F^2$ is the squared Frobenius norm which returns number of none zero entries when applied on a binary matrix.

The template, $\langle S_{tpl}, E_{tpl}\rangle$ is first rotated by evenly spacing angles to get a set of M templates: $\{\langle S_{tpl}(\theta_i), E_{tpl}(\theta_i)\rangle \,|\theta_i = \theta_{min} + (i-1)\Delta\theta, i = 1, 2, ..., M\}$. Then, each of these templates is searched in the image by sliding the window at a constant step size.

The searching can employ a coarse-to-fine approach. Initially, we may set $\Delta\theta = \frac{\pi}{4}$, $\theta_{min} = 0$ and $M = 8$. Later, we may reduce $\Delta\theta$ and select the set of θ as values around the angle which produces best matching window in the coarser searching. The calculation of directional Chamfer distance uses the code provided by the authors of [20]. The code was written in C++ and has been compiled into MEX files. It is available online [19]. The rest of the code is written in Matlab.

Fig.4.1,4.2,4.3,4.4 are 4 sets of images and the results of detection. The first and third sets contain images of an open hand. The second and fourth sets contain images of the "one" posture. In each figure, the left columns are the input images. The middle columns are edge maps detected via Canny edge detector. The right columns are silhouette extracted by using background subtraction. Besides, outline of estimated postures are superimposed on all images. Templates of these postures are generated from hand model under preset configuration parameters before taking the images. Images are taken by a regular webcam with a resolution of 400×300. Yet, during detection, images are first resized to have the height of 100 for speeding up the calculation. Each of the images was processed individually. The computation time for processing one image is about 30 seconds. From the results, we can see that although hand postures in images are not exactly the same as the template, the location and orientation of the postures were approximated with certain accuracy. It is worth mention that, during the experiment, the detection was relatively sensitive to scale selection. So, in the experiments, the scale of template was manually set approximately the same as hands in images. Otherwise, the accuracy of detection reduced obviously.

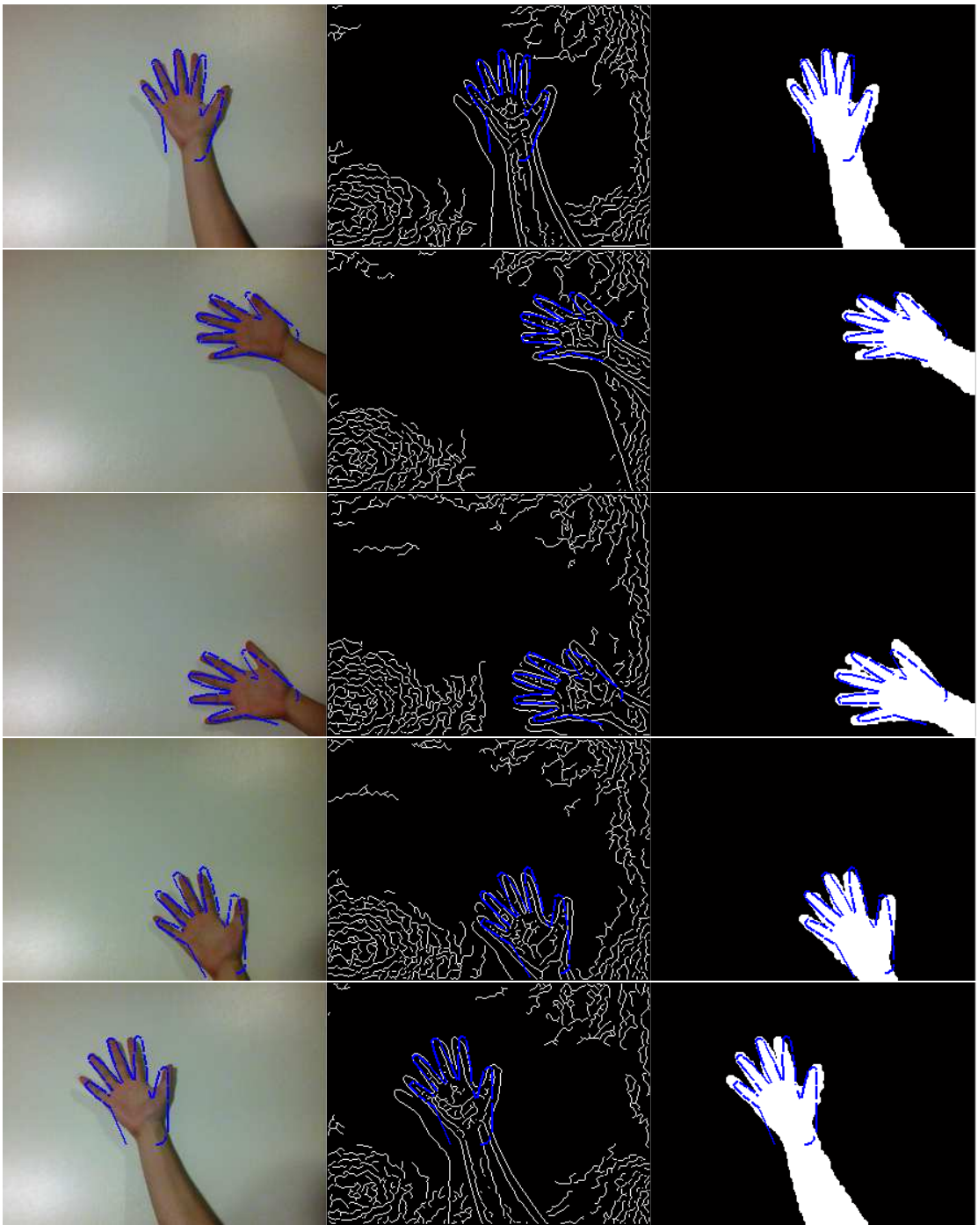

Figure 4.1: Detection of open hands in front of simple background.

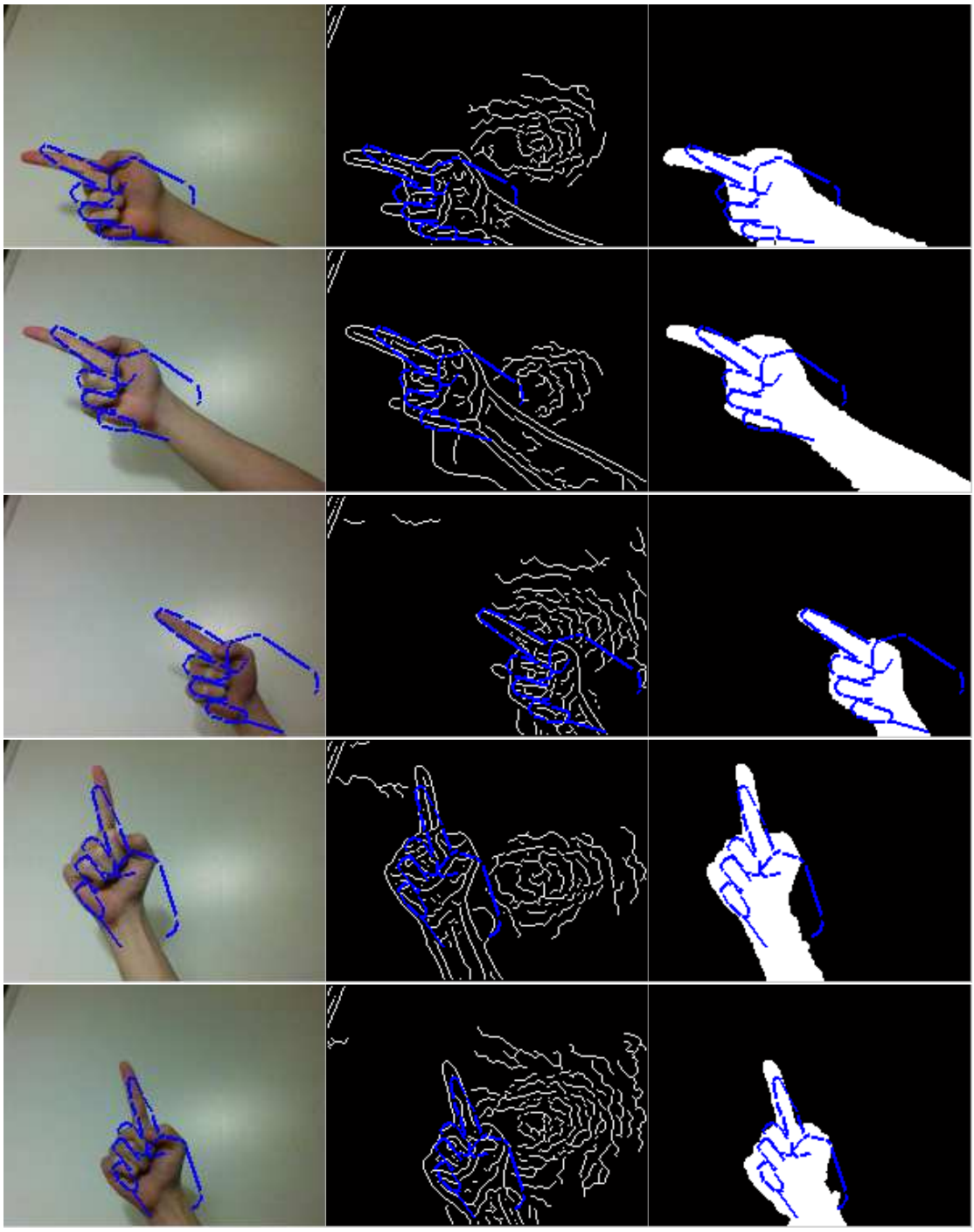

Figure 4.2: Detection of posture “one” in front of simple background.

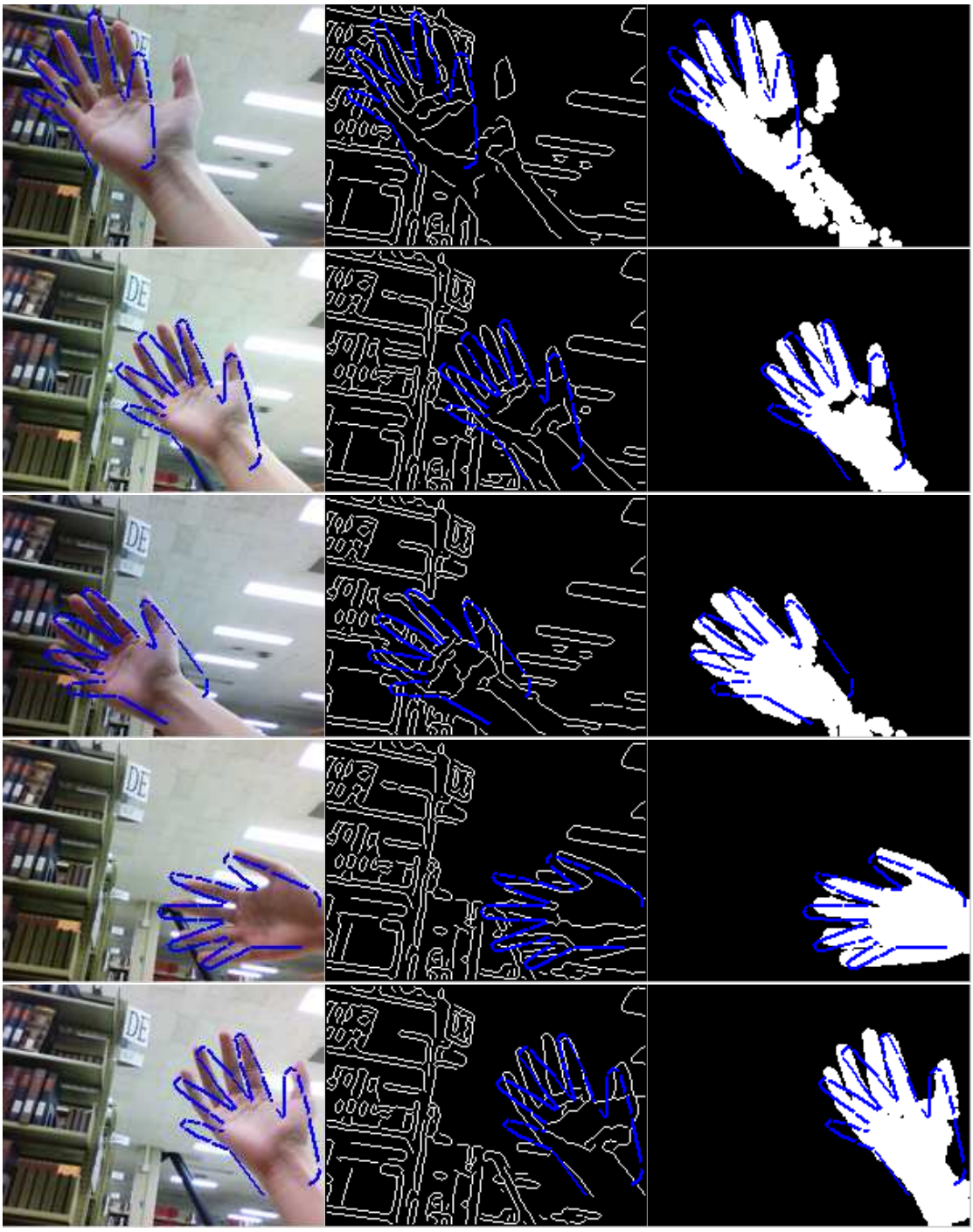

Figure 4.3: Detection of open hands in front of cluttered background.

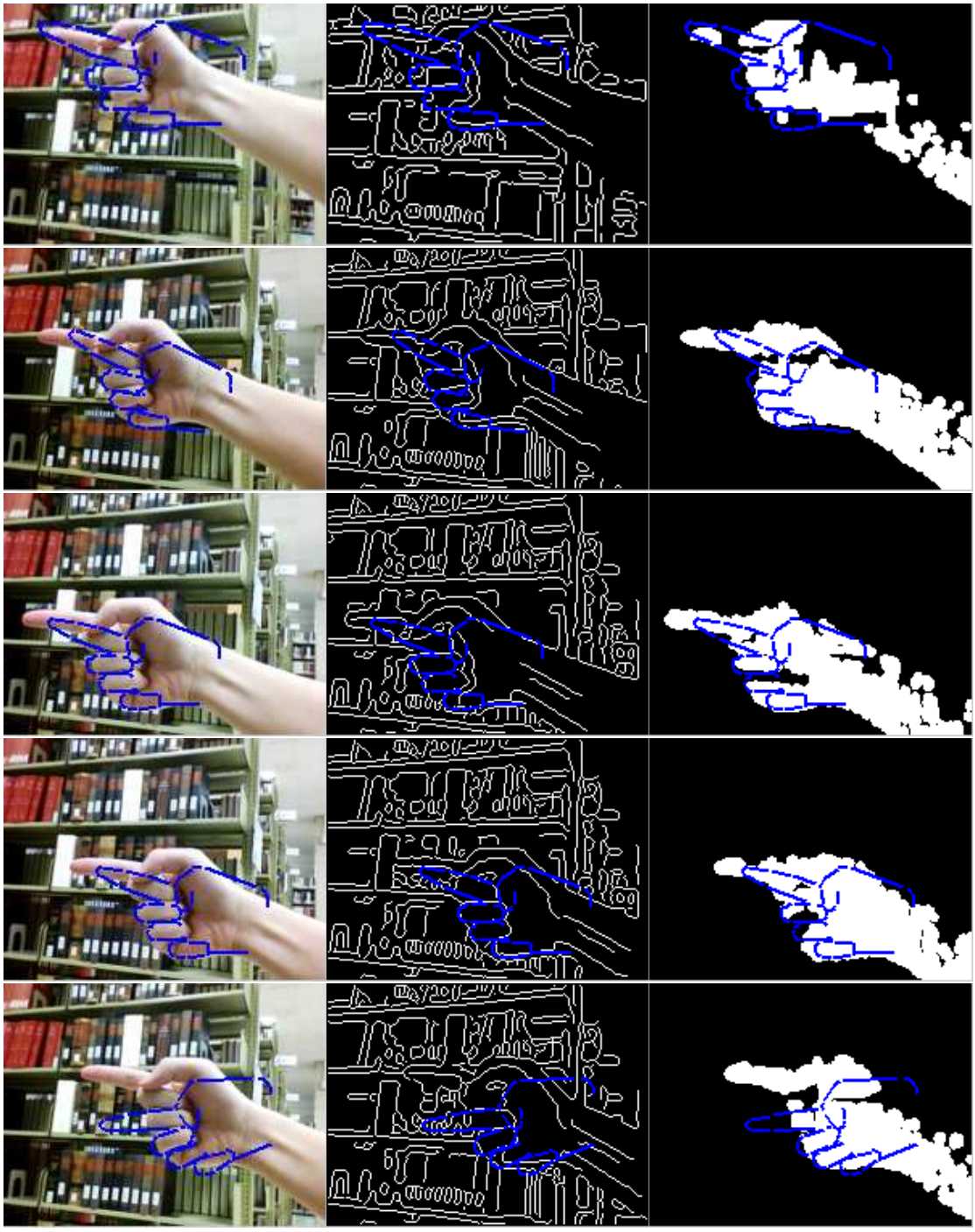

Figure 4.4: Detection of posture “one” in front of cluttered background.

Chapter 5

POSTURE ESTIMATION

The generation of a hand template from a 3D hand model can be considered as a mapping from the space of hand configuration to the space of features. The mapping could be highly nonlinear and suffers discontinuities[21, 24, 22, 10]. The process of hand pose estimation could be considered as searching in the configuration space for a point whose map in the feature space is close to the feature extracted from a hand image. To find the objective hand configuration, an optimization approach can be employed[7]. In [7], the authors use a generative model to generate a 3D sythesized hand. Its posture is decided by a vector θ comprised of 22 articulation parameters plus three translational parameters and a quaternion to define the global position and orientation of the wrist with respet to the camera. The appearance of the hand depends on an illuminant model, which is specified by a 4D vector denoted by L, and a texture model T. Then, they aim to reduce the difference between the observed image $I_{obs}(x)$ and the image of synthesized hand $I_{syn}(x; \theta, L, T)$.

Yet, optimization approaches may suffer from some drawbacks. The creation and rendering of a 3D hand model may take up certain amount of time, the high dimension of configuration space would slow down the optimization process and the nonlinearity of the space would lead to local minimals. An alternative method is to use a nonparametric approach[22], which is to create a database of sythesized hand model in advance. Meanwhile, from every sythesized model, features are extracted and registered in the feature space. During application, features will be extracted from a real hand image. Then, a search for the nearest neighbour of these features is conducted in the feature space. The corresponding hand model is considered an approximation of the hand in the image. In this chapter, the non-parametric approach for posture estimation is explored. A set of synthe-

sized data set will be generated first. Then, the technique of multidimensional scaling and its application in posture estimation will be introduced. An experiment follows to show the feasibility of the method.

5.1 Generation of A Synthesized Data Set

To try out the non-parametric approach in posture estimation, a set of synthesized images is generated first. These images are captured by projecting the outlines a hand model onto a plane when it is conducting a 2 dimensional motion: one dimension in twisting and another dimension in unfolding from an "one" posture to an open hand. Sample images are shown in Fig.5.1. The motion is parameterized with two variables, $\alpha \in \left[-\frac{\pi}{3}, \frac{\pi}{3}\right]$, representing degree of twisting, and $\beta \in [0, 1]$, representing degree of unfolding. For a more specific meaning of β, suppose α is fixed and the vectors of configuration parameters for "one" posture and open-hand posture are $\vec{\theta}_1$ and $\vec{\theta}_2$ respectively. Then β determines a posture $\vec{\theta}(\beta)$ in the form of linear combination: $\vec{\theta}(\beta) = (1-\beta)\,\vec{\theta}_1 + \beta\vec{\theta}_2$. After α and β are discretized into grid points uniformly distributed in their range of domain, each of the grid points determine a posture sample which is stored as an image in the data set. Here, domain of α is discretized into 9 values and β is discretized into 10 values. So there are 90 images in the data set.

5.2 Multidimensional Scaling and Posture Estimation

In the non-parametric approach, we look up in the database for the hand model (template) that is closest to the real hand image (query) in terms of the similarity between their features. Yet, a searching for the most similar posture among all candidates might be time consuming if query image is compared with every template. Therefore, to speed up the searching, we may resort to an organized structure for managing all templates.

For the set of images generated previously, its intrinsic dimension is 2, as the way it is produced. However, it would be useful and non-trival to recover its dimension purely from information available from the images without resorting the generation process. Meanwhile, laying the images in a metric space where their pairwise distances match dissimilarities might

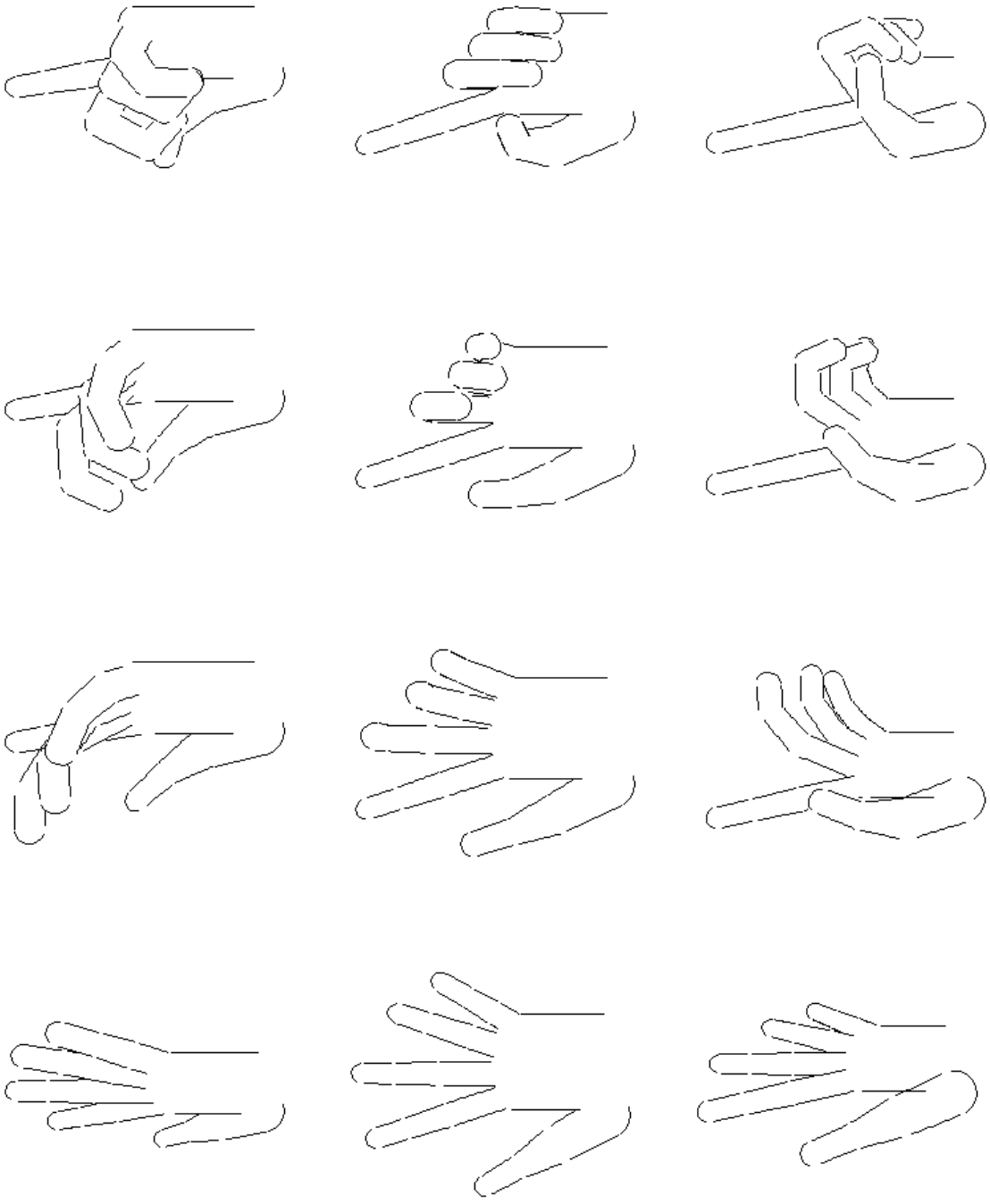

Figure 5.1: Sample images of sythesized hands.

be more meaningful than simply representing them in the α-β coordinate. Here, directional chamfer distance is used as the measure of dissimilarities among images in the dataset. The technique of non-metric multidimensional scaling[14] is adapted to perform the embedding.

5.2.1 Multidimensional Scaling

The objective of a multidimensional scaling is to present a set of n sample data as points in a t dimensional space, according to the matrix of data's pair-wise dissimilarities (δ_{ij}), in a way that their interpoint distances match dissimilarities as much as possible. In this thesis, (δ_{ij}) is assumed to be symmetric and $\delta_{ii} = 0$. Suppose, each of the embedded data point could be expressed in orthogonal coordinates by $\boldsymbol{x}_i = (x_{i1}, x_{i2}, \cdots, x_{it})^T$. Denote the distance between $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$ as $d_{ij} = \|\boldsymbol{x}_i - \boldsymbol{x}_j\|_2$. The objective of multidimensional scaling can be expressed as:

$$stress = \min_{(\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_n)} \sqrt{\frac{\sum_{i,j=1,\ldots,n, i<j} (\delta_{ij} - d_{ij})^2}{\sum_{i,j=1,\ldots,n, i<j} \delta_{ij}^2}} \tag{5.1}$$

This objective function is the same as the one in [14] except that monotony constraint is removed here since the preservation of distance is more important than the ranking of pair-wise dissimilarity at this step. Gradient descent methods could be employed to solve this problem numerically. In the experiment, the non-metric multidimensional scaling described above is applied on a set of synthesized hand postures possessing 2D motions described previously. Directional chamfer distance is used to measure the dissimilarities among postures.

5.2.2 Landmark Based Coordinate Estimation

For the problem of posture estimation, the embedding space help arrange posture templates in a coordinate system. Then, once a query posture comes in, we may estimate its coordinate in this embedding space and look up the closest point among templates. Now, the problem is

how to estimate the coordinate of the query posture. In [23], the authors raised a landmark-based classical multidimensional scaling method to provide efficient computation for the embedding of a very large set of data. A set of landmarks were first embedded into a space and then the embedding coordinates of rest points are calculated based on their distance to landmarks. Yet, their method is designed for classical multidimensional scaling, which is different from the non-metric situation of the posture estimation. But, we may apply a similar idea. Suppose $\{P_1, P_2, ..., P_K\}$ is a set of K landmark postures whose coordinates in the embedding space are $x_1, x_2, ..., x_K$ and similarity between query posture P^* and P_i is δ_i. Then the estimated coordinate of P^*, $\boldsymbol{x}^*$, could be calculated by solving this optimization problem:

$$\boldsymbol{x}^* = \arg\min_{\boldsymbol{x}} \sqrt{\frac{\sum_{i=1,2,...,K} \left(\delta_i - \|\boldsymbol{x}_i - \boldsymbol{x}\|_2\right)^2}{\sum_{i=1,2,...,K} \delta_i^2}}$$

After coordinate of P^* is obtained, the searching for closest template in the dataset only requires computing distance between two vectors, which is much faster than the computation of dissimilarity. Besides, after templates in dataset are represented in form of vectors, indexing techniques can be applied for fast looking up. The selection of landmarks may affect the accuracy of estimation as a set of landmarks which are relatively close to each other might not be able to provide a good estimation for a posture far from them. In the experiment, we may randomly generated a few sets of landmarks then choose the set of landmarkds whose sum of squared interpoint distance is the largest.

5.3 Experiments

By computing the directional chamfer distance for every pair of images, a matrix of dissimilarity, denoted as G here, is obtained. The diagonals of G should be zeros. Then, multidimensional scaling is applied on G. In this work, the computing is conducted by using the function "`mdscale`" in Matlab's statistics toolbox. When the dimensions of the embedding space increases, the amount of *stress* defined in Eq.5.1 reduces as shown in Fig.5.2. When dimension is set to 10, value of *stress* is around 0.056 and data's variance

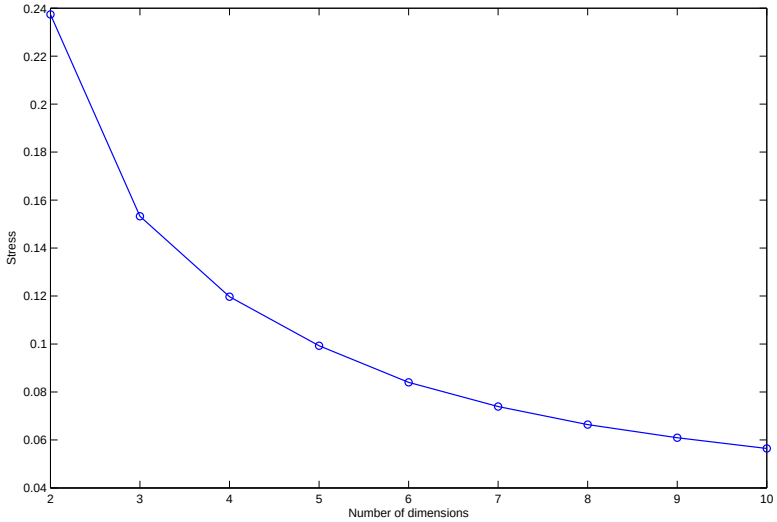

Figure 5.2: Stress of multidimensional scaling when using different number of dimensions.

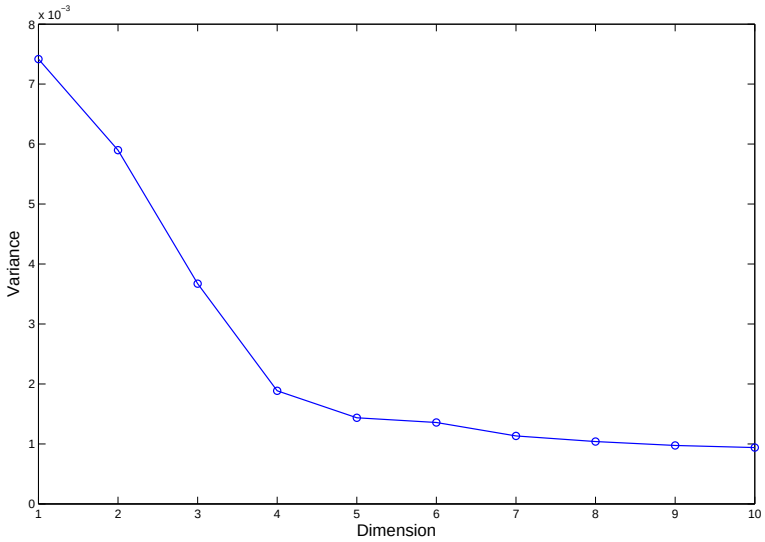

Figure 5.3: Data's variance along each dimension after they have been imbedded into a 10 dimensional Euclidean space.

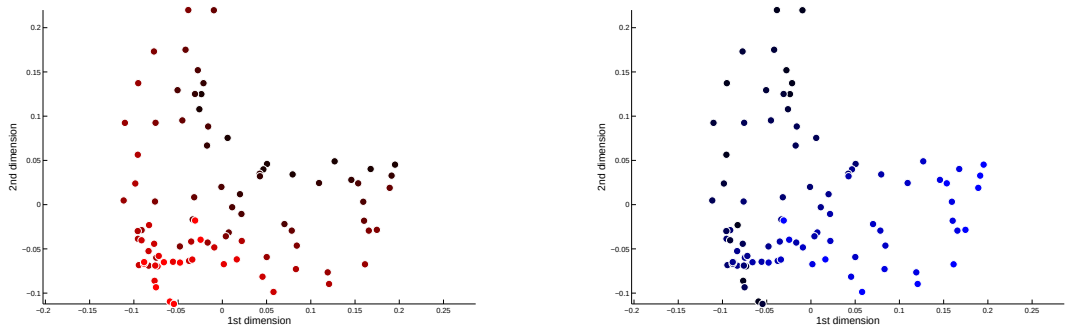

Figure 5.4: Data points' projection onto the first two dimensions. Left: brightness of a data point represents its degree of unfolding, i.e., value of β. The brighter a point is, the larger β is. Right: brightness of a data point represents its degree of twisting, i.e., value of α. The brighter a point is, the larger α is.

in each dimension is shown in Fig.5.3. We may see that the first four dimensions takes up most of the variance. Fig.5.4 shows data's projection onto their first two dimensions. Points are colored in two ways, by the degree of unfolding and by the degree of twisting respectively. From the figures, we can see that the original dimensions of α and β are reflected as the directions of down-right and down-left. This result is quite meaningful, as through multidimensional scaling we are able to recover the dimensions hidden in data. Next, we may estimate a query posture by using landmarks and their distance to the query. First, dissimilarities between each landmark and the query is calculated by computing their directional chamfer distance. Then, the coordinate of query is estimated by the method raised in Sec.5.2.2. The data point which is closest to query's estimated coordinate is considered the estimated posture. An example is shown in Fig.5.5. In the figure, data points with blue circles are the selected landmarks. The red circle shows the estimated coordinate of query posture and the solid red point connecting to it is its closest data point in 10 dimensional space. The magenta asterisk marker is the ideal estimation of query posture.

To test the accuracy of posture estimation via this method, leave-one-out cross validation is conducted. Every time, one posture from the dataset is left out as a query posture. The

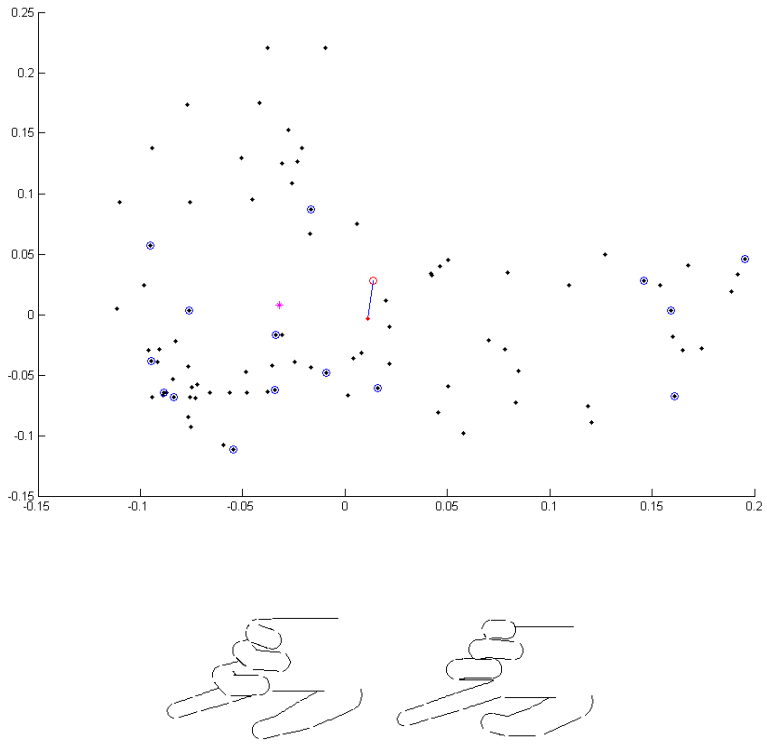

Figure 5.5: An example of posture estimation. Upper: Data points projected to the first two dimensions. 15 data points marked by blue circles are landmarks. The red circle is the estimated position of query postures. The solid red dot connected to the red circle is the data point that is closest to query posture's estimated position in 10 dimensional space, i.e. query posture's estimation. Point labeled with an asterisk marker is the posture that has least dissimilarity with query posture. Lower left: query posture. Lower right: estimated posture.

embedding space is built from the remaining 89 postures. Then, the query posture is estimated based on this space. Since estimation involves random selection of landmarks, the estimation for every query posture is repeated 10 times. Fig.5.6, 5.7, 5.8, 5.9 show the results of using 10D, 5D, 3D and 2D embedding space respectively. 15 landmarks are used during each estimation. In each figure, every blue point stands for the error of a single estimation. The X value is ID of query postures. Y value is the error defined as $(Diss_{Est} - Diss_{Opt})/Diss_{Opt} \times 100\%$, where $Diss_{Est}$ is the dissimilarity (directional chamfer distance) between query and estimated posture, $Diss_{Opt}$ is the dissimilarity between query and its most similar posture available in the dataset. Green, black and red curves connect min, average and max errors of tests for each posture. The average error of tests in 10D, 5D, 3D and 2D are 18.02%,21.63%,21.33% and 29.67% respectively. Fig.5.10 shows the situation for using 10D embedding space and using 30 landmarks in estimating query's coordinates. In this case, average error over all tests is reduced to 12.01%. Similarily, when 30 landmarks were used in cases of 5D, 3D and 2D embedding, the average errors were reduced to 16.04%, 14.71% and 21.92% respectively. It can be seen that increase in number of landmarks reduces errors in estimation.

To test whether a posture's location inside the space would affect the accuracy of estimation, in Fig.5.11, a posture's average dissimilarity to estimated postures is plotted against its radius, defined as the average distance to all postures in the dataset. It seems that these two quantities have some relation. The correlation coefficient between these two quantities is 0.6639. Yet, this correlation may partly result from the selection of data points. Alghouth data points are uniformly distributed in α-β space, they are not distributed so in the embedding space. In other words, some postures are more dissimilar to the rest postures by nature, as reflected by distances in embedding space, which causes the correlation between radius and dissimilarity. Hence, for a proper understanding towards the accuray of the estimation, the average dissimilarity needs to be converted to amount of error as shown in Fig.5.12. Now, the correlation becomes -0.389. These values were calculated in the case of

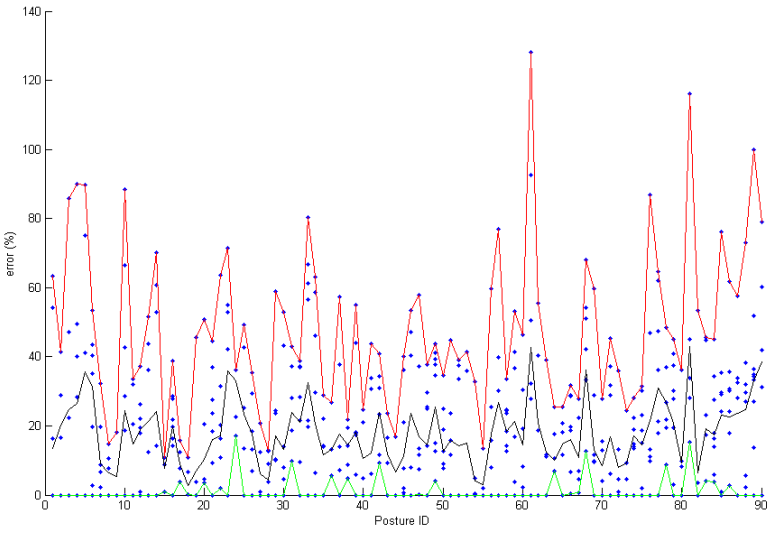

Figure 5.6: Error of posture estimation with 10 dimensional embedding and 15 landmarks. Green, black and red curves indicates the min, average and max errors of tests conducted for every single posture. Average error over all postures is 18.02%.

10D embedding and using 15 landmarks. When using 5, 3 and 2 dimensional spaces, this correlations between average errors and radius are -0.3175, -0.1657 and -0.0469 respectively, which are relatively weak.

5.4 Disscusion

The approach given above is supposed to provide a coarse estimation of hand posture. During the experiment, the accuray of estimation is found varying by query posture and landmark selection. Yet, the correlation between estimation error and the radius of a posture is not evident, especially in the cases of low dimensional embedding, as shown in the experiments. But, it may be worthwhile considering how to choose data points for forming the data set. A set of data point which uniformly distributed in the embedding space might be preferable as the error of estimation for a query posture could be managed in this case. Intuitively, embedding data to a higher dimensional space would preserve pair-

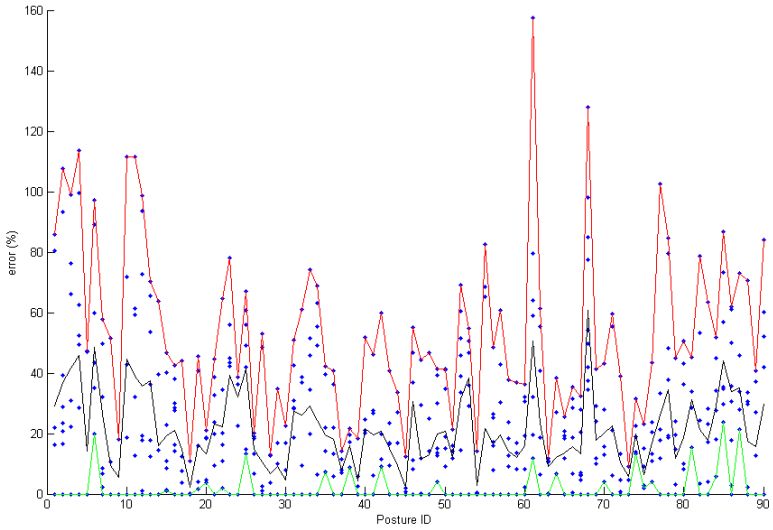

Figure 5.7: Error of posture estimation with 5 dimensional embedding and 15 landmarks. Green, black and red curves indicates the min, average and max errors of tests conducted for every single posture. Average error over all postures is 21.63%.

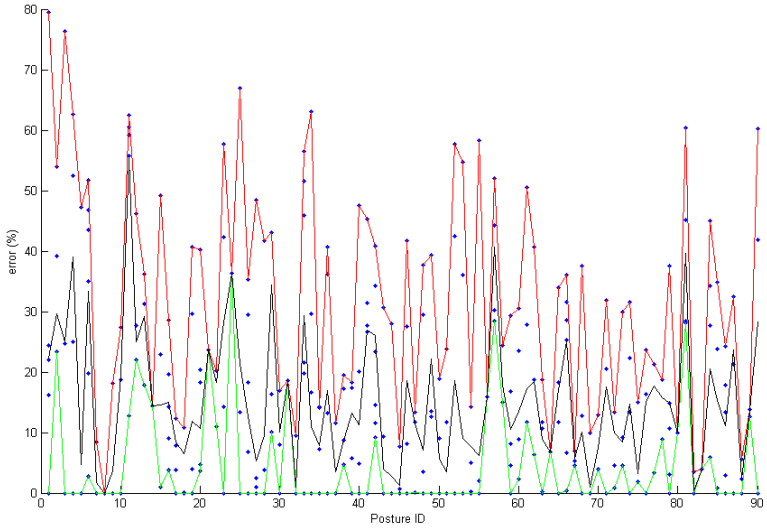

Figure 5.8: Error of posture estimation with 3 dimensional embedding and 15 landmarks. Green, black and red curves indicates the min, average and max errors of tests conducted for every single posture. Average error over all postures is 21.33%.

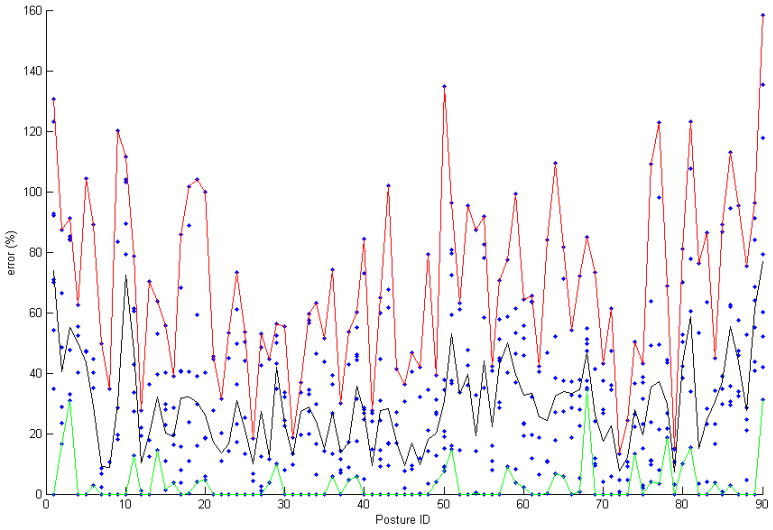

Figure 5.9: Error of posture estimation with 2 dimensional embedding and 15 landmarks. Green, black and red curves indicates the min, average and max errors of tests conducted for every single posture. Average error over all postures is 29.67%.

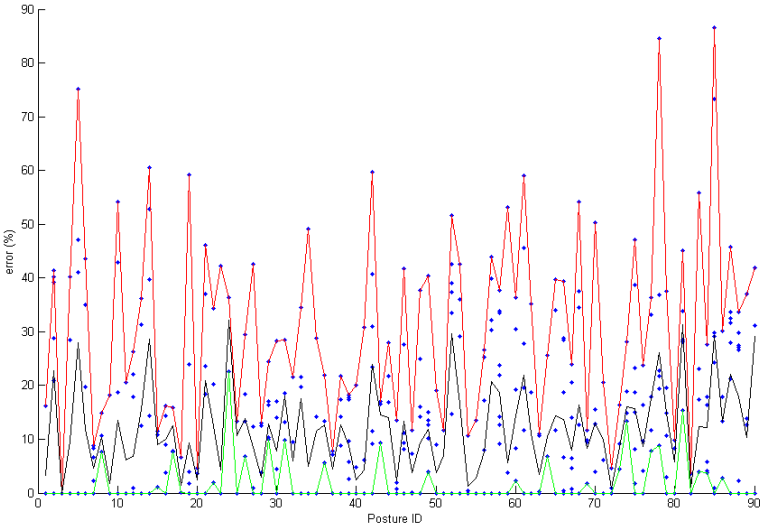

Figure 5.10: Error of posture estimation with 10 dimensional embedding and 30 landmarks. Green, black and red curves indicates the min, average and max errors of tests conducted for every single posture. Average error over all postures is 12.01%.

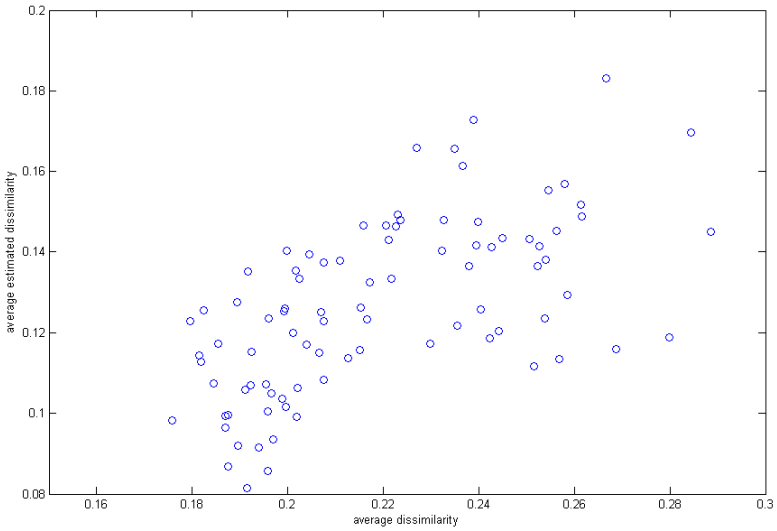

Figure 5.11: Plot of average dissimilarities against radius. X axis: the radius of a posture. Y axis: a posture's average dissimilarity to its estimations. Estimation is based on 10D embedding with 15 landmarks.

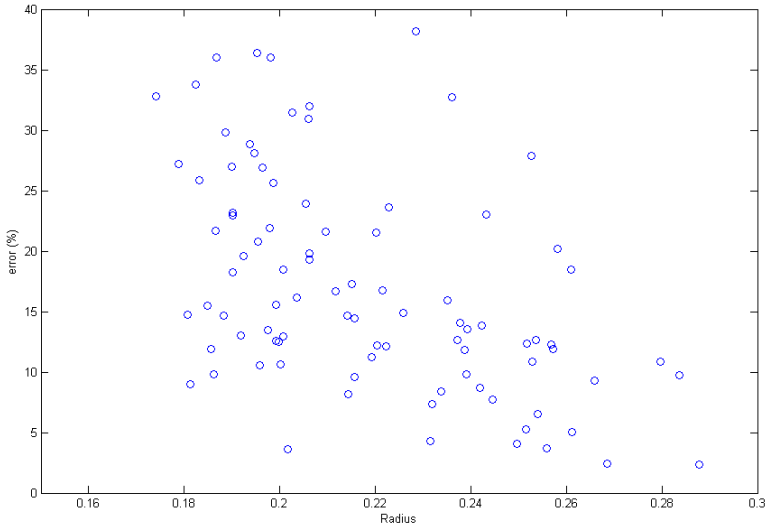

Figure 5.12: Plot of average estimation errors against radius. Estimation is based on 10D embedding with 15 landmarks.

wise dissimilarities better. But more landmarks need to be used to provide precise location estimation for the query image. Also, more templates are needed to reconstruct the space.

There are a few future work can be conducted for improving the current method. From one aspect, the highly nonlinear space of hand templates could be embedded more appropriately by making use of data's local relationship. For example, the direct application of multidimensional scaling might not be proper if the real imbedding space is a Swiss-roll-like manifold. Yet, the technique of isomap[30], which calculates data's geodesic distance based on nearest neighbours, may help recover the dimensions of embedding space more efficiently. But this results in a question about how to estimate the coordinates of a query posture since its geodesic distances to embedded data points are not easily available, which may lead to further studies. From another aspect, the measure of dissimilarity could be improved. A better hand model, such as a model using triangulated surface, could generate more realistic hand templates, which allows more features to be extracted and to be used for characterizing the dissimilarity between hand templates.

Chapter 6

CONCLUSION

This work explores the problem of hand posture estimation. Especially, people's work in model-based hand posture estimation via a single camera has been studied. The method for building a hand model from truncated quadrics and techniques for extracing edge and silhouette features are studied in details. Given the hand model, synthesized hand templates for any preset postures can be produced. Given the method for feature extraction, posture templates and images could be matched through features. Then, an application of hand posture detection is carried out,where directional chamfer distance is used as a measure for matching two edge maps. The experiments show that the location and orientation of postures could be detected in images fairly accurately. Yet, the algorithm is relatively sensitive to the selection of scale.

For the problem of posture estimation, which is to recover the articulation of a hand, a multidimensional scaling based approach is tried out. A set of hand posture templates that involve two degrees of freedom is generated first. Then, multidimensional scaling is applied for embedding them two a Euclidean space according to their pair-wise directional chamfer distances. The experiment shows that this method is able to recover the intrinsice dimensions in a sense that the layout of templates' projection onto the first two dimensions of embedding space forms a pattern that accords with the degrees of freedom contained in the data set. After the embedding space is set up, a landmark based algorithm is used for estimating the coordinate of a query posture and, further, finding the approximate posture of the query. Through experiments, it is shown that this approach leads to unneglectable errors in recoving posture articulation. The error can be caused by several factors. The high nonlinearity of the mapping from hand model to hand feature can be a significant one, which

is a well known difficulty in the problem of articulation estimation via monocular camera. The improvement of the current method may resort to a better hand model providing more features and an embedding technique that reconstructs the dimensions more efficiently, which can be studied in the future.

BIBLIOGRAPHY

[1] Pablo Arbelaez, Michael Maire, Charless Fowlkes, and Jitendra Malik. Contour detection and hierarchical image segmentation. *IEEE Trans. Pattern Anal. Mach. Intell*, 33(5):898–916, 2011.

[2] Vassilis Athitsos and Stan Sclaroff. Estimating 3D hand pose from a cluttered image. In *CVPR*, pages 432–442. IEEE Computer Society, 2003.

[3] M.K. Bhuyan, D.R. Neog, and M.K. Kar. Hand pose recognition using geometric features. In *Communications (NCC), 2011 National Conference on*, pages 1 –5, jan. 2011.

[4] J. Canny. A computational approach to edge detection. In *RCV87*, pages 184–203, 1987.

[5] Y. Chehadeh, D. Coquin, and P. Bolon. A skeletonization algorithm using chamfer distance transformation adapted to rectangular grids. In *ICPR*, pages II: 131–135, 1996.

[6] Qing Chen, Nicolas D. Georganas, and Emil M. Petriu. Hand gesture recognition using haar-like features and a stochastic context-free grammar. *IEEE T. Instrumentation and Measurement*, 57(8):1562–1571, 2008.

[7] Martin de La Gorce, David J. Fleet, and Nikos Paragios. Model-based 3D hand pose estimation from monocular video. *IEEE Trans. Pattern Anal. Mach. Intell*, 33(9):1793–1805, 2011.

[8] Martin de La Gorce and Nikos Paragios. A variational approach to monocular hand-pose estimation. *Computer Vision and Image Understanding*, 114(3):363–372, 2010.

[9] H. Freeman and L. S. Davis. A corner-finding algorithm for chain-coded curves. *IEEE Transactions on Computers*, C-26:297–303, 1977.

[10] S. S. Ge, Y. Yang, and T. H. Lee. Hand gesture recognition and tracking based on distributed locally linear embedding. *Image and Vision Computing*, 26(12):1607–1620, December 2008.

[11] T. Heap and D. Hogg. Towards 3D hand tracking using a deformable model. In *FG*, pages 140–145, 1996.

[12] B. Ionescu, D. Coquin, P. Lambert, and V. Buzuloiu. Dynamic hand gesture recognition using the skeleton of the hand. *EURASIP Journal on Applied Signal Processing*, 2005(13):2101–2109, 2005.

[13] Chutisant Kerdvibulvech and Hideo Saito. Model-based hand tracking by chamfer distance and adaptive color learning using particle filter. *EURASIP J. Image and Video Processing*, 2009, 2009.

[14] J. B. Kruskal. Multidimensional scaling by optimizing goodness-of-fit to a nonmetric hypothesis. *PSym*, 29:1–29, 1964.

[15] J. B. Kruskal. Nonmetric multidimensional scaling: A numerical method. *Psychometrika*, 29:115–129, 1964.

[16] Jintae Lee and Tosiyasu L. Kunii. Model-Based analysis of hand posture. *IEEE Computer Graphics and Applications*, 15(5):77–86, September 1995.

[17] J. Lin, Y. Wu, and T. S. Huang. Modeling the constraints of human hand motion. In *Workshop on Human Motion*, pages 121–126, 2000.

[18] Wei-Yang Lin, Yen-Lin Chiu, Kerry R. Widder, Yu Hen Hu, and Nigel Boston. Robust and accurate curvature estimation using adaptive line integrals. *EURASIP J. Adv. Sig. Proc*, 2010, 2010.

[19] Ming-Yu Liu. code for fast directional chamfer matching.

[20] Ming-Yu Liu, Oncel Tuzel, Ashok Veeraraghavan, and Rama Chellappa. Fast directional chamfer matching. In *CVPR*, pages 1696–1703. IEEE, 2010.

[21] Vladimir Pavlovic, Rajeev Sharma, and Thomas S. Huang. Visual interpretation of hand gestures for human-computer interaction: A review. *IEEE Trans. Pattern Anal. Mach. Intell*, 19(7):677–695, 1997.

[22] Javier Romero, Hedvig Kjellström, and Danica Kragic. Monocular real-time 3D articulated hand pose estimation. In *Humanoids*, pages 87–92. IEEE, 2009.

[23] Vin De Silva and Joshua B. Tenenbaum. Sparse multidimensional scaling using landmark points. 2004.

[24] B. Stenger. *Model-based hand tracking using a hierarchical bayesian filter*. PhD thesis, University of Cambridge, 2004.

[25] B. D. R. Stenger, P. R. S. Mendonca, and R. Cipolla. Model-based 3D tracking of an articulated hand. In *CVPR*, pages II:310–315, 2001.

[26] B. D. R. Stenger, A. Thayananthan, P. H. S. Torr, and R. Cipolla. Model-based hand tracking using a hierarchical bayesian filter. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 28(9):1372–1384, September 2006.

[27] B. D. R. Stenger, A. Thayananthan, P. H. S. Torr, and R. Cipolla. Estimating 3D hand pose using hierarchical multi-label classification. *Image and Vision Computing*, 25(12):1885–1894, December 2007.

[28] E. B. Sudderth, M. I. Mandel, W. T. Freeman, and A. S. Willsky. Visual hand tracking using nonparametric belief propagation. In *Workshop on Generative Model Based Vision*, page 189, 2004.

[29] Richard Szeliski. *Computer Vision: Algorithms and Applications*. Springer-Verlag New York, Inc., New York, NY, USA, 1st edition, 2010.

[30] J. B. Tenenbaum, V. de Silva, and J. C. Langford. A global geometric framework for nonlinear dimensionality reduction. *Science*, 290(5500):2319–2323, December 2000.

[31] A. Thayananthan, B. Stenger, P. H. S. Torr, and R. Cipolla. Learning a Kinematic Prior for Tree-Based Filtering. In *British Machine Vision Conference*, 2003.

[32] Paul Viola and Michael Jones. Robust Real-time Object Detection. *International Journal of Computer Vision*, 2001.

[33] Chieh-Chih Wang and Ko-Chih Wang. Hand posture recognition using adaboost with sift for human robot interaction. In *Recent Progress in Robotics: Viable Robotic Service to Human*, volume 370 of *Lecture Notes in Control and Information Sciences*, pages 317–329. Springer Berlin / Heidelberg, 2008.

[34] Wikipedia. Canny edge detector — wikipedia, the free encyclopedia, 2012. [Online; accessed 15-July-2012].

[35] Y. Wu and T. S. Huang. Capturing articulated human hand motion: A divide-and-conquer approach. In *ICCV*, pages 606–611, 1999.

[36] Ying Wu and T. S. Huang. Hand modeling, analysis and recognition. *IEEE Signal Processing Magazine*, 18(3):51–60, May 2001.

[37] E. Yoruk, E. Konukoglu, B. Sankur, and J. Darbon. Shape-based hand recognition. *IEEE Trans. Image Processing*, 15(7):1803–1815, July 2006.

[38] H. N. Zhou and T. S. Huang. Tracking articulated hand motion with eigen dynamics analysis. In *ICCV*, pages 1102–1109, 2003.

Appendix A

COMPUTER CODE

A.1 Matlab Code for Generating A Single Posture Template

Main script for generating a posutre template:

```
clear all; close all; clc;

% load preset configuration and calibration parameters
load .\config\config_openhand;
load .\config\HandParameters_1.mat;

% pass arguments via files
save('HandConfig','Theta1','Theta2');
save('HandParameters','Palm','R','L');

% generate skeleton and hand model
display('Started to generate hand...');
[HandSkeleton,HandPieces]=GenerateHand();
display('Started Plot Skeleton...');
PlotSkeleton(HandSkeleton,1);
axis auto
grid on
display('Started to plot 3d hand...');
PlotHand(HandPieces,1);
display('Started to plot contour...');
contourLineRep = PlotContour(HandPieces,2);
```

Code of the function ‘‘`GenerateHand`’’:

```
function [HandSkeleton,HandPieces]=GenerateHand()

load HandParameters;
load HandConfig;

HandSkeleton.Fingers = zeros(3,4,5);
HandSkeleton.Wrist = Palm.wrist;

HandPieces.Ellipsoids = cell(0);
```

```
HandPieces.Cones = cell(0);
HandPieces.Cylinders = cell(0);

FingerRoot=zeros(3,5);
FingerRoot(:,2) =Palm.wrist + L(1,2)*Palm.e1 + R(1,2)*Palm.e2;
for FingerIndex=3:5
    FingerRoot(:,FingerIndex) = FingerRoot(:,FingerIndex-1) ...
        + (L(1,FingerIndex)-L(1,FingerIndex-1))*Palm.e1 ...
        - (R(1,FingerIndex-1)+R(1,FingerIndex))*1.05*Palm.e2;
end
FingerRoot(:,1) = Palm.wrist + Palm.e2*Palm.w;

tmp = (FingerRoot(:,2) + FingerRoot(:,5)) / 2 - Palm.wrist;
Palm.center = Palm.wrist + Palm.e1*min(L(1,2:5))/2 ...
+ (tmp'*Palm.e2)*Palm.e2;

% generate fingers
for FingerIndex=1:5
    Finger = GenerateFinger( FingerRoot(:,FingerIndex), ...
    Theta1, Theta2, L, Palm, FingerIndex );
    HandSkeleton.Fingers(:,:,FingerIndex) = Finger;
    for j=2:4
        cone = GenerateCone(R(j,FingerIndex), ...
        R(j-1,FingerIndex),L(j,FingerIndex));
        cone = MakeConeTransform(cone, ...
        Finger(:,j),Finger(:,j-1));
        HandPieces.Cones{length(HandPieces.Cones) + 1, 1} ...
        = cone;
    end
    for j=1:4
        Ellipsoid = GenerateEllipsoid(R(j,FingerIndex), ...
        R(j,FingerIndex),R(j,FingerIndex));
        Ellipsoid = MakeEllipsoidTransform(Ellipsoid, ...
        Finger(:,j),[1;0;0],[0;1;0]);
        HandPieces.Ellipsoids{ length(HandPieces.Ellipsoids)+1, ...
         1 } = Ellipsoid;
    end
end

% generate palm
palmc = max(L(1,2:5)) - min(L(1,2:5));
palmb = max(R(1,2:5))*1.5;
x1 = abs((FingerRoot(:,5)-Palm.center)'*Palm.e2 );
x1_2 = abs((FingerRoot(:,2)-Palm.center)'*Palm.e2 );
```

```
r1 = R(1,5);
r1_2 = R(1,2);
palma = sqrt( (r1^2*palmb^2-palmb^2*x1^2-palmb^2)/(r1^2 - palmb^2)) ;
palma_2 = sqrt( (r1_2^2*palmb^2-palmb^2*x1_2^2-palmb^2) ...
/(r1_2^2 - palmb^2)) ;
palma = max([palma, palma_2]);
L = abs( (Palm.center - Palm.wrist)'*Palm.e1 );
Cylinder = GenerateCylinder(palmb,palma, L, -L );
Cylinder = MakeCylinderTransform(Cylinder, ...
Palm.center, Palm.n,Palm.e2);
HandPieces.Cylinders{ length(HandPieces.Cylinders)+1, 1} = Cylinder;

Ellipsoid = GenerateEllipsoid(palmc,palma,palmb);
Ellipsoid = MakeEllipsoidTransform(Ellipsoid, ...
Palm.center+Palm.e1*L,-Palm.e2,-Palm.n);
HandPieces.Ellipsoids{ length(HandPieces.Ellipsoids)+1, ...
 1 } = Ellipsoid;

Ellipsoid = GenerateEllipsoid(palmb,palma,palmb);
Ellipsoid = MakeEllipsoidTransform(Ellipsoid, ...
Palm.center-Palm.e1*L,-Palm.e2,-Palm.n);
HandPieces.Ellipsoids{ length(HandPieces.Ellipsoids)+1, ...
1 } = Ellipsoid;
```

Code of the function ‘‘`GenerateFinger`’’:

```
function Finger = GenerateFinger( FingerRoot, ...
Theta1, Theta2, L, Palm, FingerIndex )

thetahat = 20/180*pi;

Finger = zeros(3,4);
Finger(:,1) = FingerRoot;
Lloc = L(:,FingerIndex);
Theta1loc = Theta1(:,FingerIndex);
Theta2loc = Theta2(FingerIndex);

if( FingerIndex > 1 )
    Thetatmp = Theta1loc(1);
    Joint2 = Lloc(2)*[cos(Thetatmp)*cos(Theta2loc) ;
        cos(Thetatmp)*sin(Theta2loc) ;
        sin(Thetatmp) ];

    Thetatmp = Theta1loc(1)+Theta1loc(2);
    Joint3 = Joint2 + Lloc(3)*[cos(Thetatmp)*cos(Theta2loc) ;
```

```
        cos(Thetatmp)*sin(Theta2loc) ;
        sin(Thetatmp) ];

    Thetatmp = Theta1loc(1)+Theta1loc(2)+Theta1loc(3);
    Joint4 = Joint3 + Lloc(4)*[cos(Thetatmp)*cos(Theta2loc) ;
        cos(Thetatmp)*sin(Theta2loc) ;
        sin(Thetatmp) ];

    Finger(:,2:4) = [Joint2 Joint3 Joint4];

    Finger(:,2:4) = repmat(FingerRoot,[1,3]) ...
    + [Palm.e1 Palm.e2 Palm.n]*Finger(:,2:4);

else % for thumb
    theta2_loc = Theta2loc(1);
    theta1_loc = Theta1loc(1);
    PI2 = Lloc(2)*[sin(theta2_loc)*cos(theta1_loc); ...
    sin(theta2_loc)*sin(theta1_loc);cos(theta2_loc)];
    Xt = [cos(theta1_loc);sin(theta1_loc);0];
    Zt = PI2 / norm(PI2);
    Yt = cross(Zt,Xt);
    Yt = Yt/norm(Yt);
    Xt = cross(Yt,Zt);
    Xt = cos(thetahat)*Xt + sin(thetahat)*Yt;
    Yt = cross(Zt,Xt);
    theta1_loc = Theta1loc(2);
    PI3 = Lloc(3)*[Xt,Yt,Zt]*[0;sin(theta1_loc);cos(theta1_loc)]+PI2;
    theta1_loc2 = Theta1loc(3);
    PI4 = PI3 + Lloc(4)*[Xt,Yt,Zt] ...
    *[0;sin(theta1_loc+theta1_loc2);cos(theta1_loc+theta1_loc2)];
    Finger = [Palm.e2,Palm.n,Palm.e1]*[zeros(3,1), PI2, PI3, PI4] ...
     + repmat(FingerRoot,[1,4]);
end
```

Code of the function "GenerateCone":

```
function Cone = GenerateCone(r1, r2, h)
w = r2*sqrt(1-((r2-r1)/h)^2);
alpha = (r2-r1)/h*r2;
H = w^2/alpha;
beta = r1/r2*alpha;
d = H-(beta+h-alpha);
Cone.Q.w=w;
Cone.Q.d=w;
Cone.Q.h=H;
```

```
Cone.pi0 = [0;1;0;d];
Cone.pi1 = [0;1;0;H];
Cone.r1 = r1;
Cone.r2 = r2;
Cone.w = w;
Cone.alpha = alpha;
Cone.H = H;
Cone.beta = beta;
Cone.d = d;
```

Code of the function "`MakeConeTransform`":

```
function Cone=MakeConeTransform(Cone,P1,P2)

s1=-[0;Cone.d+Cone.beta;0];
s2=-[0;Cone.H+Cone.alpha;0];

n1=(s2-s1)/norm(s1-s2);
n2=(P2-P1)/norm(P1-P2);

phi = (n1+n2)/2;
phi = phi/norm(phi);

R=2*phi*phi'-eye(length(phi));
t=P1-R*s1;

T=[R t;0 0 0 1];
Cone.T = T;
Cone.P1 = P1;
Cone.P2 = P2;
vn = Cone.pi0(1:3,1);
vn2 = R*vn;
d = -Cone.pi0(4,1);
d2 = d+ t'*vn2/norm(vn2);
Cone.pi0 = [vn2 ; -d2];

vn = Cone.pi1(1:3,1);
vn2 = R*vn;
d = -Cone.pi1(4,1);
d2 = d+ t'*vn2/norm(vn2);
Cone.pi1 = [vn2 ; -d2];

Tinv = inv(T);
Q = diag([1/Cone.Q.w^2 -1/Cone.Q.h^2 1/Cone.Q.d^2 0]);
Q2 = Tinv'*Q*Tinv;
```

```
A = Q2(1:3,1:3);
b = Q2(1:3,4);
c = Q2(4,4);
C = c*A-b*b';
Cone.C = C;
Cone.A = A;
Cone.b = b;
Cone.c = c;
```

Code of the function "GenerateEllipsoid":

```
function Ellipsoid = GenerateEllipsoid( w,h,d, ...
clippingn,clippingd1,clippingd2)
Ellipsoid.Q.w = w;
Ellipsoid.Q.h = h;
Ellipsoid.Q.d = d;
Elliposid.clippingn = [0;1;0];

if nargin>3
    Ellipsoid.clippingn = clippingn;
    Ellipsoid.clippingd1 = clippingd1;
    Ellipsoid.clippingd2 = clippingd2;
end
```

Code of the function "MakeEllipsoidTransform":

```
function Ellipsoid = MakeEllipsoidTransform(Ellipsoid,center,eh,ed)
ew = cross(eh,ed);
R = [ew eh ed];
t=center;
T = [R t;0 0 0 1];
Ellipsoid.T = T;
Q = diag([1/Ellipsoid.Q.w^2 1/Ellipsoid.Q.h^2 1/Ellipsoid.Q.d^2 -1]);
Tinv = inv(T);
Q2 = Tinv'*Q*Tinv;
A = Q2(1:3,1:3);
b = Q2(1:3,4);
c = Q2(4,4);
C = c*A - b*b';
Ellipsoid.C = C;
Ellipsoid.A = A;
Ellipsoid.b = b;
Ellipsoid.c = c;
```

Code of the function "GenerateCylinder":

```
function Cylinder = GenerateCylinder(d,w,clippingd1,clippingd2)
Cylinder.Q.d = d;
Cylinder.Q.w = w;
Cylinder.clippingd1 = clippingd1;
Cylinder.clippingd2 = clippingd2;

Cylinder.pi0 = [0;1;0;-clippingd1];
Cylinder.pi1 = [0;1;0;-clippingd2];
```

Code of the function "`MakeCylinderTransform`":

```
function Cylinder = MakeCylinderTransform(Cylinder,center,ed,ew)
eh =cross(ed,ew);
R = [ew eh ed];
t = center;
T = [R t;zeros(1,length(t)) 1];
Cylinder.T = T;
Cylinder.eh = eh;
Cylinder.ew = ew;
Cylinder.ed = ed;
Cylinder.center = center;

vn = Cylinder.pi0(1:3,1);
vn2 = R*vn;
d = -Cylinder.pi0(4,1);
d2 = d+ t'*vn2/norm(vn2);
Cylinder.pi0 = [vn2 ; -d2];

vn = Cylinder.pi1(1:3,1);
vn2 = R*vn;
d = -Cylinder.pi1(4,1);
d2 = d+ t'*vn2/norm(vn2);
Cylinder.pi1 = [vn2 ; -d2];

Tinv = inv(T);
Q = diag([1/Cylinder.Q.w^2 0 1/Cylinder.Q.d^2 -1]);
Q2 = Tinv'*Q*Tinv;
A = Q2(1:3,1:3);
b = Q2(1:3,4);
c = Q2(4,4);
C = c*A-b*b';
Cylinder.C = C;
Cylinder.A = A;
Cylinder.b = b;
Cylinder.c = c;
```

Code of the function "PlotHand":

```
function PlotHand(HandPieces,figInd)
N = length(HandPieces.Cones);
for i=1:N
    [X,Y,Z]=PlotCone(HandPieces.Cones{i,1});
    figure(figInd);
    mesh(X,Y,Z);
    hold on;
    alpha(.5);
end

N = length(HandPieces.Ellipsoids);
for i=1:N
    [X,Y,Z]=PlotEllipsoid(HandPieces.Ellipsoids{i,1});
    figure(figInd);
    mesh(X,Y,Z);
    hold on;
    alpha(.5);
end

N = length(HandPieces.Cylinders);
for i=1:N
    [X,Y,Z]=PlotCylinder(HandPieces.Cylinders{i,1});
    figure(figInd);
    mesh(X,Y,Z);
    hold on;
    alpha(.5);
end
```

Code of the function "PlotCone":

```
function [x,y,z]=PlotCone(Cone)
t2 = linspace(-Cone.H,-Cone.d,5);
Rt = sqrt(Cone.r1^2-Cone.beta^2)*(t2+Cone.H)/(-Cone.d+Cone.H) + ...
Cone.w*(t2+Cone.d)/(-Cone.H+Cone.d);
[x,y,z]=cylinder(Rt);
z = z*(Cone.H-Cone.d);
tmp = z;
z = x;
x = y;
y = tmp;
y = y-Cone.H;
T = Cone.T;
R = T(1:3,1:3);
```

```
t = T(1:3,4);

[m,n] = size(x);
for i=1:m
    for j=1:n
        v = R*[x(i,j);y(i,j);z(i,j)]+t;
        x(i,j) = v(1);
        y(i,j) = v(2);
        z(i,j) = v(3);
    end
end
```

Code of the function "PlotEllipsoid":

```
function [x,y,z]=PlotEllipsoid(Ellipsoid)

T = Ellipsoid.T;
R = T(1:3,1:3);
t = T(1:3,4);

[x,y,z] = ellipsoid(0,0,0,Ellipsoid.Q.w, Ellipsoid.Q.h, Ellipsoid.Q.d);
[m,n]=size(x);
for i=1:m
    for j=1:n
        v = R*[x(i,j);y(i,j);z(i,j)]+t;
        x(i,j) = v(1);
        y(i,j) = v(2);
        z(i,j) = v(3);
    end
end
```

Code of the function "PlotCylinder":

```
function [x,y,z]=PlotCylinder(Cylinder)
[x,y,z] = cylinder();
z = z*abs(Cylinder.clippingd1-Cylinder.clippingd2);
z = z+Cylinder.clippingd2;
tmp = z;
z = x;
x = y;
y = tmp;
x = x*Cylinder.Q.w;
z = z*Cylinder.Q.d;
T = Cylinder.T;
R = T(1:3,1:3);
```

```
t = T(1:3,4);

[m,n] = size(x);
for i=1:m
    for j=1:n
        v = R*[x(i,j);y(i,j);z(i,j)]+t;
        x(i,j) = v(1);
        y(i,j) = v(2);
        z(i,j) = v(3);
    end
end
```

Code of the function "PlotContour":

```
function lineRep = PlotContour( HandPieces,figInd )
if nargin<2
    figInd = 3;
end
figure(figInd);
hold on;
axis equal;
lineRep = [];
N = length(HandPieces.Ellipsoids);
for i=1:N-1 % do not plot the ellipsoid at wrist
    [X,Y]=DrawEllipsoidContour(HandPieces.Ellipsoids{i,1},HandPieces);
    for j=1:length(X)
        if length(X{j})>1
            plot(X{j},Y{j},'r','LineWidth',2);
            lineRep = [lineRep ; ...
            X{j}(1:(end-1))',Y{j}(1:(end-1))',X{j}(2:end)',Y{j}(2:end)'];
        end
    end
end

N = length(HandPieces.Cones);
for i=1:N
    [X,Y]=DrawConeContour(HandPieces.Cones{i,1},HandPieces);
    for j=1:length(X)
        if length(X{j})>1
            plot(X{j},Y{j},'r','LineWidth',2);
            lineRep = [lineRep ; ...
            X{j}(1),Y{j}(1),X{j}(end),Y{j}(end)];
        end
    end
end
```

```
N = length(HandPieces.Cylinders);
for i=1:N
    [X,Y]=DrawCylinderContour(HandPieces.Cylinders{i,1},HandPieces);
    for j=1:length(X)
        if length(X{j})>1
            plot(X{j},Y{j},'r','LineWidth',2);
            lineRep = [lineRep ; X{j}(1),Y{j}(1),X{j}(end),Y{j}(end)];
        end
    end
end
```

Code of the function "DrawEllipsoidContour":

```
function [x,y]=DrawEllipsoidContour( Ellipsoid,HandPieces )
numpoints = 21;
C = Ellipsoid.C;
[V,D]=eigs(C);
a = sqrt(-D(3,3)/D(1,1));
b = sqrt(-D(3,3)/D(2,2));
t2 = linspace(0,2*pi,numpoints);
x1 = a*cos(t2);
x2 = b*sin(t2);
tmp = V*[x1;x2;ones(1,length(x1))];
flag = ones(size(x1));

for i=1:length(flag)
    xc = tmp(:,i);
    depthinv = -Ellipsoid.b'*xc/Ellipsoid.c;
    if depthinv < 0
        depthinv = depthinv*-1;
        xc = xc*-1;
    end
    for j=1:length(HandPieces.Cones)
        C2 = HandPieces.Cones{j};
        if IsBehindCone( xc, depthinv,C2 )
            flag(i) = 0;
            break;
        end
    end
    if flag(i) == 0
        continue;
    end
    for j=1:length(HandPieces.Cylinders)
        C2 = HandPieces.Cylinders{j};
```

```
        if IsBehindCylinder(xc, depthinv, C2)
            flag(i) = 0;
            break;
        end
    end
    if flag(i) == 0
        continue;
    end
    for j=1:length(HandPieces.Ellipsoids)
        E2 = HandPieces.Ellipsoids{j};
        if IsEqualEllipsoid(Ellipsoid,E2) == 1
            continue;
        end
        if IsBehindEllipsoid(xc,depthinv,E2)
            flag(i) = 0;
            break;
        end
    end
end
[x,y]=PlotSegments([tmp(1,:)./tmp(3,:);tmp(2,:)./tmp(3,:)],flag);
```

Code of the function "DrawConeContour":

```
function [x,y]=DrawConeContour(Cone,HandPieces)
numpoints=21;
C = Cone.C;
[V,D]=eigs(C);

Pii = cell(1,2);

Pii{1} = Cone.pi0;
Pii{2} = Cone.pi1;
k = cell(1,2);
for i=1:2
   k{i} = Pii{i}(1:3,1) + Cone.b*-Pii{i}(4,1)/Cone.c;
end
lhat = cell(1,2);
lhat{1} = [sqrt(abs(D(1,1)));sqrt(abs(D(2,2)));0];
lhat{2} = [sqrt(abs(D(1,1)));-sqrt(abs(D(2,2)));0];

l = cell(1,2);
l{1} = V*lhat{1};
l{2} = V*lhat{2};

n1 = cell(1,2);
```

```
n2 = cell(1,2);
d = cell(1,2);
xx = cell(0);
yy = cell(0);
for i=1:2
    n1{i} = l{i}(1);
    n2{i} = l{i}(2);
    d{i} = -l{i}(3);
    tmp = sqrt(n1{i}^2+n2{i}^2);
    n1{i} = n1{i}/tmp;
    n2{i} = n2{i}/tmp;
    d{i} = d{i}/tmp;
    l{i} = l{i}/tmp;
end

t = cell(2,2);
for i=1:2
    for j=1:2
        t{i,j} = ( d{i}*( k{j}(1)*n1{i}+k{j}(2)*n2{i} ) ...
        +k{j}(3) )/( k{j}(1)*n2{i} - k{j}(2)*n1{i} );
    end
end

x = cell(1,2);
y = cell(1,2);

for i=1:2
    s = linspace( t{i,1},t{i,2},numpoints );
    x{i} = -n2{i}*s + n1{i}*d{i};
    y{i} = n1{i}*s + n2{i}*d{i};
end

for outeri=1:2
    flag = ones(size(x{outeri}));
    tmp = [x{outeri};y{outeri};ones(1,length(x{outeri}))];
    for i=1:length(flag)
        xc = tmp(:,i);
        depthinv = -Cone.b'*xc/Cone.c;

        for j=1:length(HandPieces.Cones)
            C2 = HandPieces.Cones{j};
            if IsEqualCone(Cone,C2) == 1
                continue;
            end
```

```
            if IsBehindCone( xc,depthinv,C2 )
                flag(i) = 0;
                break;
            end
        end
        if flag(i) == 0
            continue;
        end
        for j=1:length(HandPieces.Cylinders)
            Cylinder = HandPieces.Cylinders{j};
            if IsBehindCylinder( xc, depthinv, Cylinder )
                flag(i) = 0;
                break;
            end
        end
        if flag(i) == 0
            continue;
        end
        for j=1:length(HandPieces.Ellipsoids)
            E2 = HandPieces.Ellipsoids{j};
            if IsBehindEllipsoid(xc, depthinv, E2)
                flag(i) = 0;
                break;
            end
        end
    end
    ind = flag>0;
    tmp = tmp(:,ind);
    [xtmp,ytmp] = PlotSegments([tmp(1,:);tmp(2,:)],flag);
    for k=1:length(xtmp)
        insertInd = length(xx) + 1;
        xx{ insertInd } = xtmp{k};
        yy{ insertInd } = ytmp{k};
    end
end
x = xx;
y = yy;
```

Code of the function "`DrawCylinderContour`":

```
function [x,y] = DrawCylinderContour(Cylinder,HandPieces)
numpoints = 31;
C = Cylinder.C;
[V,D]=eigs(C);
```

```
Pii = cell(1,2);
Pii{1} = Cylinder.pi0;
Pii{2} = Cylinder.pi1;
k = cell(1,2);
for i=1:2
   k{i} = Pii{i}(1:3,1) + Cylinder.b*-Pii{i}(4,1)/Cylinder.c;
end
lhat = cell(1,2);
lhat{1} = [sqrt(abs(D(1,1)));sqrt(abs(D(2,2)));0];
lhat{2} = [sqrt(abs(D(1,1)));-sqrt(abs(D(2,2)));0];

l = cell(1,2);
l{1} = V*lhat{1};
l{2} = V*lhat{2};

n1 = cell(1,2);
n2 = cell(1,2);
d = cell(1,2);
for i=1:2
    n1{i} = l{i}(1);
    n2{i} = l{i}(2);
    d{i} = -l{i}(3);
    tmp = sqrt(n1{i}^2+n2{i}^2);
    n1{i} = n1{i}/tmp;
    n2{i} = n2{i}/tmp;
    d{i} = d{i}/tmp;
    l{i} = l{i}/tmp;
end

t = cell(2,2);
for i=1:2
    for j=1:2
        t{i,j} = ( d{i}*( k{j}(1)*n1{i}+k{j}(2)*n2{i} )+k{j}(3) ) ...
        /( k{j}(1)*n2{i} - k{j}(2)*n1{i} );
    end
end

x = cell(1,2);
y = cell(1,2);

for i=1:2
    s = linspace( t{i,1},t{i,2},numpoints );
    x{i} = -n2{i}*s + n1{i}*d{i};
    y{i} = n1{i}*s + n2{i}*d{i};
```

```
end
xx = cell(0);
yy = cell(0);
for outeri=1:2
    flag = ones(size(x{outeri}));
    tmp = [x{outeri};y{outeri};ones(1,length(x{outeri}))];
    for i=1:length(flag)
        xc = tmp(:,i);
        depthinv = -Cylinder.b'*xc/Cylinder.c;

        for j=1:length(HandPieces.Cones)
            C2 = HandPieces.Cones{j};
            if IsBehindCone( xc,depthinv,C2 )
                flag(i) = 0;
                break;
            end
        end
        if flag(i) == 0
            continue;
        end
        for j=1:length(HandPieces.Cylinders)
            C2 = HandPieces.Cylinders{j};
            if IsEqualCylinder(Cylinder,C2)
                continue;
            end
            if IsBehindCylinder( xc, depthinv, Cylinder )
                flag(i) = 0;
                break;
            end
        end
        if flag(i) == 0
            continue;
        end
        for j=1:length(HandPieces.Ellipsoids)
            E2 = HandPieces.Ellipsoids{j};
            if IsBehindEllipsoid(xc, depthinv, E2)
                flag(i) = 0;
                break;
            end
        end
    end
    ind = flag>0;
    [xtmp,ytmp] = PlotSegments([tmp(1,:);tmp(2,:)],flag);
    for k=1:length(xtmp)
```

```
        insertInd = length(xx) + 1;
        xx{ insertInd } = xtmp{k};
        yy{ insertInd } = ytmp{k};
    end
end
x=xx;
y=yy;
```

Code of function "IsBehindCone":

```
function flag = IsBehindCone( xc, depthinv, Cone )
flag = 0;
delta = (2*Cone.b'*xc)^2 - 4*Cone.c*xc'*Cone.A*xc;
if( delta <=0 )
    return;
end
depth1 = (-2*Cone.b'*xc+sqrt(delta))/(2*Cone.c);
depth2 = (-2*Cone.b'*xc-sqrt(delta))/(2*Cone.c);
if ((depth1>depthinv) ...
&& ([xc/depth1;1]'*Cone.pi0)*([xc/depth1;1]'*Cone.pi1)<0 ) ...
||( (depth2 > depthinv) ...
&&(([xc/depth2;1]'*Cone.pi0)*([xc/depth2;1]'*Cone.pi1)<0  ))
    flag = 1;
end
```

Code of function "IsBehindCylinder":

```
function flag = IsBehindCylinder( xc, depthinv, Cylinder )
flag = 0;
delta = (2*Cylinder.b'*xc)^2 - 4*Cylinder.c*xc'*Cylinder.A*xc;
if( delta <=0 )
    return;
end
depth1 = (-2*Cylinder.b'*xc+sqrt(delta))/(2*Cylinder.c);
depth2 = (-2*Cylinder.b'*xc-sqrt(delta))/(2*Cylinder.c);
if ((depth1>depthinv) ...
&&( ([xc/depth1;1]'*Cylinder.pi0)*([xc/depth1;1]'*Cylinder.pi1)<0   ) )...
||( (depth2 > depthinv) ...
&&(([xc/depth2;1]'*Cylinder.pi0)*([xc/depth2;1]'*Cylinder.pi1)<0  ))
    flag = 1;
end
```

Code of the function "IsBehindEllipsoid":

```
function flag = IsBehindEllipsoid( xc, depthinv, Ellipsoid )
```

```
flag = 0;
delta = (2*Ellipsoid.b'*xc)^2 - 4*Ellipsoid.c*xc'*Ellipsoid.A*xc;
if( delta <=0 )
    return;
end
depth1 = (-2*Ellipsoid.b'*xc+sqrt(delta))/(2*Ellipsoid.c);
depth2 = (-2*Ellipsoid.b'*xc-sqrt(delta))/(2*Ellipsoid.c);
if (depth1>depthinv) || (depth2 > depthinv)
    flag = 1;
end
```

A.2 Matlab Code for Detecting A Given Posture in Images

The main script for detecting a given posture in images:

```
tic
clear al; close all; clc;

% load configuration and calibration parameters of a given posture
load .\config\config_openhand;
load .\config\HandParameters_1;

% load background image and query images
BackgroundImage = imread('.\images\series10\img_series10_0.jpg');
QueryImage0 = imread('.\images\series10\img_series10_2.jpg');
wfilename = 'img_series10_2';
QueryImage = QueryImage0;

% background subtraction
C = makecform('srgb2lab');
I_lab = applycform(BackgroundImage,C);

P_lab = applycform(QueryImage0,C);
P_diff = abs(P_lab - I_lab);
P_diff = sqrt(double( P_diff(:,:,2).^2 + P_diff(:,:,3).^2 ));
P_binary = 255*uint8(P_diff>5);

se = strel('disk',5);
P_binary = imerode(P_binary,se);
P_binary = imdilate(P_binary,se);
% end of background subtraction

[mm,nn,dd]=size(QueryImage);
if mm>130
```

```
    QueryImage = imresize(QueryImage,[130,NaN]);
    P_binary = imresize(P_binary,[130,NaN]);
    P_binary = P_binary>10;
end

figure(1);
imshow(QueryImage);
saveas(gcf,[wfilename,'_query.fig']);
imwrite(QueryImage,[wfilename,'_query.png']);

HandRegionSize = sum(sum(double(P_binary>0)));
QueryImage_color = QueryImage;
if dd>1
    QueryImage = rgb2gray( QueryImage );
end

% edge detection
query = edge(QueryImage,'canny');


threshold = 0.3;
lineMatchingPara = struct(...
    'NUMBER_DIRECTION',60,...
    'DIRECTIONAL_COST',0.5,...
    'MAXIMUM_EDGE_COST',30,...
    'MATCHING_SCALE',1.0,...
    'TEMPLATE_SCALE',1.0,...
    'BASE_SEARCH_SCALE',1.05,...
    'MIN_SEARCH_SCALE',-1,...
    'MAX_SEARCH_SCALE',5,...
    'BASE_SEARCH_ASPECT',1.0,...
    'MIN_SEARCH_ASPECT',-1,...
    'MAX_SEARCH_ASPECT',0,...
    'SEARCH_STEP_SIZE',2,...
    'SEARCH_BOUNDARY_SIZE',2,...
    'MIN_COST_RATIO',1.0...
    );

% Set the parameter for line fitting function
lineFittingPara = struct(...
    'SIGMA_FIT_A_LINE',0.5,...
    'SIGMA_FIND_SUPPORT',0.5,...
    'MAX_GAP',2.0,...
    'N_LINES_TO_FIT_IN_STAGE_1',300,...
```

```
    'N_TRIALS_PER_LINE_IN_STAGE_1',100,...
    'N_LINES_TO_FIT_IN_STAGE_2',100000,...
    'N_TRIALS_PER_LINE_IN_STAGE_2',1);

lineFittingPara2 = struct(...
    'SIGMA_FIT_A_LINE',0.5,...
    'SIGMA_FIND_SUPPORT',0.5,...
    'MAX_GAP',2.0,...
    'N_LINES_TO_FIT_IN_STAGE_1',0,...
    'N_TRIALS_PER_LINE_IN_STAGE_1',0,...
    'N_LINES_TO_FIT_IN_STAGE_2',100000,...
    'N_TRIALS_PER_LINE_IN_STAGE_2',1);

ModelGenerator;

templateLineRep0 = contourLineRep;
templateCenter = [0.5*(max(max( templateLineRep0(:,[1,3]) )) ...
    + min(min( templateLineRep0(:,[1,3]) ))); ...
    0.5*(max(max( templateLineRep0(:,[2,4]) )) ...
    + min(min( templateLineRep0(:,[2,4]) )))];

templateRegion0 = region;
[regionRowIdx,regionColIdx] = find( templateRegion0 );

angleLow = 0;
angleHigh = 360;
angleStep = 30;

scaleBase = 1.1;
scaleMin = -2;
scaleMax = 2;
while angleStep>5

    collection_counter = 0;
    collection_detWinds = cell(0);
    collection_lineRep = cell(0);
    collection_edgeCost = [];
    collection_regionCoverageRatio = [];
    collection_region = cell(0);
    collection_angle = cell(0);

    for i=angleLow:angleStep:angleHigh

        theta = i/180*pi;
```

```
rotationMatrix = [ cos( theta ), -sin( theta );
    sin( theta ), cos( theta )];
templateLineRep = 0*templateLineRep0;

for j=1:size(templateLineRep0,1)
    startpt = templateLineRep0(j,[1,2])';
    endpt = templateLineRep0(j,[3,4])';
    startpt = rotationMatrix*(startpt-templateCenter) ...
        + templateCenter;
    endpt = rotationMatrix*(endpt-templateCenter) ...
        + templateCenter;
    templateLineRep(j,:) = [startpt(1) startpt(2) ...
        endpt(1) endpt(2)];
end
minX = min(min( templateLineRep(:,[1,3]) ));
minY = min(min( templateLineRep(:,[2,4]) ));
maxX = max(max( templateLineRep(:,[1,3]) ));
maxY = max(max( templateLineRep(:,[2,4]) ));
ratio = min([100/(maxX-minX),100/(maxY-minY)]);
templateLineRep(:,[1,3]) ...
    = round((templateLineRep(:,[1,3]) - minX)*ratio + 2);
templateLineRep(:,[2,4]) ...
    = round((templateLineRep(:,[2,4]) - minY)*ratio + 2);

maxY2 = max(max( templateLineRep(:,[2,4]) )); % flip along y axis
templateLineRep(:,[2,4]) = maxY2 - templateLineRep(:,[2,4]) + 2;
marginX = min(min( templateLineRep(:,[1,3]) ));
marginY = min(min( templateLineRep(:,[2,4]) ));

canvasWidth = max(max(  templateLineRep(:,[1,3])  ));
canvasHeight = max(max(  templateLineRep(:,[2,4])  ));
canvas = zeros( canvasHeight+5 , canvasWidth+5  );

pts = [regionRowIdx';regionColIdx'];
minx2 = min( pts(1,:) );
maxx2 = max( pts(1,:) );
miny2 = min( pts(2,:) );
maxy2 = max( pts(2,:) );
ratio2 = min([100/(maxx2-minx2),100/(maxy2-miny2)]);

pts = rotationMatrix*pts;
minx22 = min( pts(1,:) );
miny22 = min( pts(2,:) );
% marginX and marginY need to be in such order:
```

```
pts(1,:) = round( (pts(1,:) - minx22)*ratio2+ marginY );
pts(2,:) = round( (pts(2,:) - miny22)*ratio2+ marginX );

for idx=1:length(pts)
    canvas( pts(1,idx) , pts(2,idx) ) = 255;
end

figure(50);
clf;
subplot(1,2,1);
PlotLineRep(templateLineRep);
subplot(1,2,2);
imshow(canvas);

figure(1);
clf;
subplot(1,2,1);
imshow(query);
subplot(1,2,2);
PlotLineRep(templateLineRep);
axis equal;
hold off;

template = cell(1);
template{1} = templateLineRep;

lineMatchingPara1 = struct(...
    'NUMBER_DIRECTION',60,...
    'DIRECTIONAL_COST',0.5,...
    'MAXIMUM_EDGE_COST',30,...
    'MATCHING_SCALE',1.0,...
    'TEMPLATE_SCALE',1.0,...
    'BASE_SEARCH_SCALE',scaleBase,...
    'MIN_SEARCH_SCALE',-1,...
    'MAX_SEARCH_SCALE',1,...
    'BASE_SEARCH_ASPECT',1.0,...
    'MIN_SEARCH_ASPECT',0,...
    'MAX_SEARCH_ASPECT',0,...
    'SEARCH_STEP_SIZE',2,...
    'SEARCH_BOUNDARY_SIZE',2,...
    'MIN_COST_RATIO',1.0...
    );

[detWinds] = mex_fdcm_detect(double(query),template,threshold,...
```

```
        lineFittingPara2,lineMatchingPara1);
    if size(detWinds,1) == 0
        continue;
    end

    for windInd = 1:size(detWinds,1)
        if detWinds(windInd,6)<10
            continue;
        end
        collection_counter = collection_counter + 1;
        collection_detWinds{ collection_counter } ...
            = detWinds(windInd,:);
        collection_edgeCost( collection_counter,1 )...
            = detWinds(windInd,5);
        collection_lineRep{ collection_counter } = templateLineRep;
        collection_region{ collection_counter } = canvas;
        collection_angle{ collection_counter } = i;
        sx = detWinds(windInd,1);
        ex = sx + detWinds(windInd,3);
        sy = detWinds(windInd,2);
        ey = sy + detWinds(windInd,4);
        a8251_patch = P_binary( sy:ey,sx:ex );
        colrange = min( [ex-sx+1,size(canvas,2)] );
        rowrange = min( [ey-sy+1,size(canvas,1)] );
        collection_regionCoverageRatio( collection_counter,1 ) ...
            = sum( sum( (a8251_patch(1:rowrange,1:colrange)>0) ...
            .*( canvas(1:rowrange,1:colrange) >0)  ) )/HandRegionSize;
    end
end

[minVal, minCollectionIdx] ...
    = min(collection_edgeCost.*(-log(collection_regionCoverageRatio)));
detWinds = collection_detWinds{ minCollectionIdx };
best_detWinds = collection_detWinds{ minCollectionIdx };
best_template.lineRep = collection_lineRep{ minCollectionIdx };
best_template.region = collection_region{ minCollectionIdx };
best_angle = collection_angle{ minCollectionIdx };

best_scale ...
    =  best_detWinds(1,3)/max(max(best_template.lineRep(:,[1,3])));
scaleBase = best_scale;
scaleMin = -2;
scaleMax = 2;
```

```
    angleStep = angleStep/2;
    angleLow = best_angle - 3*angleStep;
    angleHigh = best_angle + 3*angleStep;

end

nDetection = size(detWinds,1);
fprintf('\n/*Illustration of the detection result*/\n');
fprintf('Wind ID (  x0 ,  y0 , width , height, cost, count )\n');
for i=1:min(nDetection,2)
    fprintf('Wind %02d ( %d ,  %d ,    %d ,   %d , %2.2f ,  %d  )\n',...
        i,detWinds(i,1),detWinds(i,2),detWinds(i,3),detWinds(i,4), ...
        detWinds(i,5),detWinds(i,6));
end
fprintf('...\n');

figure(1);
subplot(1,2,1);
color = [0 1 0];
lineWidth = 3;
for i=1:size(detWinds)
    sx = detWinds(i,1);
    ex = sx + detWinds(i,3);
    sy = detWinds(i,2);
    ey = sy + detWinds(i,4);
    line([sx ex],[sy sy],'Color',color,'LineWidth',lineWidth);
    line([sx ex],[ey ey],'Color',color,'LineWidth',lineWidth);
    line([sx sx],[sy ey],'Color',color,'LineWidth',lineWidth);
    line([ex ex],[sy ey],'Color',color,'LineWidth',lineWidth);
end
title('Detection result');

figure(5);
imshow(query);
hold on;
trans = [best_detWinds(1,1), best_detWinds(1,2)];

xMin = min(min( best_template.lineRep(:,[1,3]) ));
xMax = max(max( best_template.lineRep(:,[1,3]) ));
yMin = min(min( best_template.lineRep(:,[2,4]) ));
yMax = max(max( best_template.lineRep(:,[2,4]) ));
ratioX = abs( detWinds(i,3) ) / abs(xMax - xMin);
ratioY = abs( detWinds(i,4) ) / abs(yMax - yMin);
```

```
matchedTemplate = 0*best_template.lineRep;
matchedTemplate(:,[1,3]) ...
    = (best_template.lineRep(:,[1,3]) - xMin) * ratioX + xMin;
matchedTemplate(:,[2,4]) ...
    = (best_template.lineRep(:,[2,4]) - yMin) * ratioY + yMin;

for i=1:size(best_template.lineRep,1)
    linestart = [matchedTemplate(i,1),matchedTemplate(i,2)] + trans;
    lineend = [matchedTemplate(i,3),matchedTemplate(i,4)] + trans;
    line([linestart(1),lineend(1)], ...
        [linestart(2),lineend(2)],'LineWidth',2);
end

saveas(gcf,[wfilename,'_5.fig']);
frame = getframe;
imwrite(frame.cdata,[wfilename,'_5.png']);

figure(7);
imshow(QueryImage_color);
for i=1:size(best_template.lineRep,1)
    linestart = [matchedTemplate(i,1),matchedTemplate(i,2)] + trans;
    lineend = [matchedTemplate(i,3),matchedTemplate(i,4)] + trans;
    line([linestart(1),lineend(1)], ...
        [linestart(2),lineend(2)],'LineWidth',2);
end

frame = getframe;
imwrite(frame.cdata,[wfilename,'_7.png']);
saveas(gcf,[wfilename,'_7.fig']);

figure(8);
imshow(P_binary);
for i=1:size(best_template.lineRep,1)
    linestart = [matchedTemplate(i,1),matchedTemplate(i,2)] + trans;
    lineend = [matchedTemplate(i,3),matchedTemplate(i,4)] + trans;
    line([linestart(1),lineend(1)], ...
        [linestart(2),lineend(2)],'LineWidth',2);
end
frame = getframe;
imwrite(frame.cdata,[wfilename,'_8.png']);
saveas(gcf,[wfilename,'_8.fig']);

locPatch = query( max([1,sy-10]):min([ey+10,size(query,1)]), ...
    max([1,sx-10]):min([ex+10,size(query,2)]) );
```

```
scale = 1;
templateLineRep0 = best_template.lineRep;
templateRegion0 = best_template.region;
[detWinds] = mex_fdcm_detect(double(locPatch),{templateLineRep0}, ...
    threshold,lineFittingPara2,lineMatchingPara);

toc
```

The function "`mex_fdcm_detect`" implements fast directional chamfer matching [20]. Function "`mex_fitline`" which will be used in the following codes converts an edge map into its line representation. Both of these two functions are provided on this webpage:

`http://www.umiacs.umd.edu/~mingyliu/research.html`.

in the section of "Fast Object Localization and Pose Estimation in Heavy Clutter for Robotic Grasping".

A.3 Matlab Code for Posture Estimation

Main script for posture estimation:

The following code implement the posture estimation mentioned in Ch.5.

```
K = 15; % number of landmarks
dim = 2; % dimension of embedding space
QueryIdx = 20; % select query image from the dataset
startInd = [1 1];
endInd = [10 9];
fileDir ='./05/';
load './05/DistMatrix90_fixed'
G = GraphDist;
for i=1:length(G)
    G(i,i) = 0;
end
[M,N] = size(G);
numNN = K;
imgN = M;
imgTemplates = cell(1,imgN);
for i=1:imgN
    if mod(i,100) == 0
        display(i);
    end
    c2 = mod(i-1,endInd(2)-startInd(2)+1)+1;
    c1 = floor((i-c2)/(endInd(2)-startInd(2)+1))+1;
    file = [fileDir,'sample_',num2str(c1),'_',num2str(c2),'.bmp'];
    tmp = imread(file);
    imgTemplates{i} = tmp;
```

```
    if i==QueryIdx
        queryFile = file;
    end
end

G_leftoneout = G([1:(QueryIdx-1),(QueryIdx+1):M], ...
    [1:(QueryIdx-1),(QueryIdx+1):M]);
opts = statset('MaxIter',10000000,'TolFun',1e-8);
[Y,stress] = mdscale( G_leftoneout,dim,'Options',opts, ...
    'criterion','metricstress' );
Y2 = mdscale( G, dim,'Options',opts,'criterion','metricstress' );
landmarkInd = [];
datapointInd = [];

maxInterDist = 0;
for iter=1:10
    landmarkInd0 = ceil(rand*(M-1));
    datapointInd0 = [1:(landmarkInd0-1),(landmarkInd0+1):(M-1)];
    for i=1:K-1
        ind = ceil(rand*length(datapointInd0));
        landmarkInd0 = [landmarkInd0, datapointInd0(ind)];
        datapointInd0 = datapointInd0([1:(ind-1), ...
            (ind+1):length(datapointInd0)]);
    end

    interDist = 0;
    for i=1:length(landmarkInd0)
        for j=(i+1):length(landmarkInd0)
            interDist = interDist ...
                + norm( Y(landmarkInd0(i),:) - Y(landmarkInd0(j),:) , 2);
        end
    end

    if interDist > maxInterDist
        maxInterDist = interDist;
        landmarkInd = landmarkInd0;
        datapointInd = datapointInd0;
    end
end
landmarkCoord = Y(landmarkInd,:);
GetDistToLandMark;
deltaA = G(QueryIdx,landmarkInd+(landmarkInd>=QueryIdx));
dist = deltaA;
[minVal,minInd] = sort(dist,'ascend');
```

```
nearestLandmarkDist = dist(minInd(1:numNN));
nearestLandmarkCoord = Y(landmarkInd(minInd(1:numNN)),:);
[coord,error] = GetOptimalCoord4NewPoint(nearestLandmarkCoord, ...
    nearestLandmarkDist, Y2(QueryIdx,:)');
tmp = Y - repmat(coord',[imgN-1,1]);
tmp = diag(tmp*tmp');
[val,ind] = min(tmp);
if ind >=QueryIdx
    ind = ind+1;
end
figure(1);
subplot(1,2,1);
img_query = imread(queryFile);
imshow(255-img_query);
title('query image');

res = GraphDist(QueryIdx,ind);

subplot(1,2,2);
imshow(255-imgTemplates{ind});
title(['estimated posture, res:',num2str(res)]);

figure(5111);
hold on;

for i=1:length(Y)
    plot(Y(i,1),Y(i,2),'k.');
end
for i=1:length(landmarkInd)
    plot( Y(landmarkInd(i),1),Y(landmarkInd(i),2),'bo');
end

bestmatch = [Y((ind>QueryIdx)*(ind-1) ...
    + (ind<=QueryIdx)*ind,1),Y((ind>QueryIdx)*(ind-1) ...
    + (ind<=QueryIdx)*ind,2)];
plot(coord(1),coord(2),'ro');
[a,b]=sort(G(:,QueryIdx));
if b(2)>QueryIdx
    optInd = b(2)-1;
else
    optInd = b(2);
end
plot(Y(optInd,1),Y(optInd,2),'m*');
plot([coord(1),bestmatch(1)], ...
```

```
    [coord(2),bestmatch(2)],'b');
plot(bestmatch(1),bestmatch(2),'r.');
```

Code of script "GetDistToLandMark":

```
tmp = (endInd - startInd + 1);
imgN = 1;
for i=1:length(tmp)
    imgN = imgN*tmp(i);
end
threshold = 1;
lineMatchingPara = struct(...
    'NUMBER_DIRECTION',60,...
    'DIRECTIONAL_COST',0.5,...
    'MAXIMUM_EDGE_COST',30,...
    'MATCHING_SCALE',1.0,...
    'TEMPLATE_SCALE',1.0,...
    'BASE_SEARCH_SCALE',1.0,...
    'MIN_SEARCH_SCALE',0,...
    'MAX_SEARCH_SCALE',0,...
    'BASE_SEARCH_ASPECT',1.0,...
    'MIN_SEARCH_ASPECT',0,...
    'MAX_SEARCH_ASPECT',0,...
    'SEARCH_STEP_SIZE',2,...
    'SEARCH_BOUNDARY_SIZE',2,...
    'MIN_COST_RATIO',1.0...
    );

lineFittingPara = struct(...
    'SIGMA_FIT_A_LINE',0.5,...
    'SIGMA_FIND_SUPPORT',0.5,...
    'MAX_GAP',2.0,...
    'N_LINES_TO_FIT_IN_STAGE_1',300,...
    'N_TRIALS_PER_LINE_IN_STAGE_1',100,...
    'N_LINES_TO_FIT_IN_STAGE_2',100000,...
    'N_TRIALS_PER_LINE_IN_STAGE_2',1);

lineFittingPara2 = struct(...
    'SIGMA_FIT_A_LINE',0.5,...
    'SIGMA_FIND_SUPPORT',0.5,...
    'MAX_GAP',2.0,...
    'N_LINES_TO_FIT_IN_STAGE_1',0,...
    'N_TRIALS_PER_LINE_IN_STAGE_1',0,...
    'N_LINES_TO_FIT_IN_STAGE_2',100000,...
    'N_TRIALS_PER_LINE_IN_STAGE_2',1);
```

```
LineRepSet = cell(1,imgN);
LineMapSet = cell(1,imgN);
for iter=1:length(landmarkInd)
    i = landmarkInd(iter);
    if mod(i,100) == 0
        display(i);
    end
    c2 = mod(i-1,endInd(2)-startInd(2)+1)+1;
    c1 = floor((i-c2)/(endInd(2)-startInd(2)+1))+1;
    file = [fileDir,'sample_',num2str(c1),'_',num2str(c2),'.bmp'];
    img = imread(file);
    img = imresize(img,[100,NaN]);
    img = double(img>10)*255;
    % convert the template edge map into a line representation
    [lineRep lineMap] = mex_fitline(double(img),lineFittingPara);
    LineRepSet{i} = lineRep;
    LineMapSet{i} = lineMap;
end

img = imread(queryFile);
img = imresize(img,[100,NaN]);

[mm,nn,dd] = size(img);
if dd>1
    img = rgb2gray(img);
    img = 255*double(edge(img,'canny'));
end

img = double(img>10)*255;
[lineRepQuery lineMapQuery] = mex_fitline(double(img),lineFittingPara);
query = lineMapQuery;

deltaA = zeros(length(landmarkInd),1);
template = cell(1);
for iter=1:length(landmarkInd)
    i = landmarkInd(iter);
    display(i);
    template{1} = LineRepSet{i};
    [detWinds] = mex_fdcm_detect(double(query),template,threshold,...
        lineFittingPara2,lineMatchingPara);
    [detWinds2] = mex_fdcm_detect(double(LineMapSet{i}),{lineRepQuery}, ...
    threshold,lineFittingPara2,lineMatchingPara);
    if isempty(detWinds) || isempty(detWinds2)
```

```
        deltaA(i) = 9999;
    else
        deltaA(i) = 0.5*(min(detWinds(:,5)) + min(detWinds2(:,5)));
    end
end

deltaA = deltaA(landmarkInd);
```

Code of the function "GetOptimalCoord4NewPoint":

```
function [Coord,res] = GetOptimalCoord4NewPoint( Y, dist, y0 )
Yt = Y';
[k,n] = size(Yt); % k is the dimension, n is number of landmarks
if nargin > 2
    Coord = y0;
else
    Coord = sum(Yt,2)/n;
end
maxiter = 1000;

for i=1:maxiter
    Coordold = Coord;
    grad = GetGradient(Yt,Coord,dist);
    fun = @(s)( GetCost(Yt,Coord - s*grad,dist) );
    [opts,val] = fminsearch(fun,0.1);
    Coord = Coord - opts*grad;
    res = val;
    if norm(Coordold - Coord) < 1e-10
        break;
    end
end

display(['num of iteration: ',num2str(i)]);

function grad = GetGradient(Yt,coord,dist)
[k,n] = size(Yt);
dcoord = 0.00001;
grad = zeros(k,1);
val0 = GetCost(Yt,coord,dist);
for i=1:k
    coord2 = coord;
    coord2(i) = coord(i) + dcoord;
    grad(i) = (GetCost(Yt,coord2,dist) - val0)/dcoord;
end
grad = grad / norm(grad);
```

```
function cost = GetCost(Yt,coord,dist)
cost = 0;
n = size(Yt,2);
for i=1:n
    cost = cost + (norm(coord-Yt(:,i))-dist(i))^2;
end
z = sum(dist.^2);
cost = cost/z;
```