# Hand Gesture Recognition using Kinect

*Cliff Chan and Seyed Sepehr Mirfakhraei*

Boston University
Department of Electrical and Computer Engineering
8 Saint Mary's Street
Boston, MA 02215
www.bu.edu/ece

Dec 13, 2013

Technical Report No. ECE-2013-04

# Contents

# List of Figures

# List of Tables

# 1 Introduction

The Microsoft Kinect is a low cost device that combines an RGB camera with a depth sensor. Depth information is inferred by comparing a scene illuminated by a structured IR speckle dot pattern against a calibrated reference. Specifically, a depth image is calculated internally by comparing the spacing of the returned dots against known values at specific depths [1]. The end result is an inexpensive RGB-D (red, green, blue, and depth) sensor, a device that has proven useful in various Human Computer Interface problems from finger counting to gesture recognition.

## 1.1 Problem Statement

One particularly interesting application of such sensors is the classification of gestures for automatic sign language recognition. Implementations have been created for various sign language alphabets and words including Greek, English, and Japanese [2]. While such sign language dictionaries are well beyond the scope of this project, our goal is to implement a Kinect based gesture recognition system that is capable of resolving numerical digits from 0 through 5 as well as more general single hand gestures.

# 2 Literature Review

Numerous papers have been written on the subject of generalized hand gesture recognition using RGB-D sensors. One technique considered early on was proposed by Du [3]. The idea is to trace the contours of a hand silhouette and detect concavities and convexities. The convexity points correspond to the finger tips, while the concavity points define the region between fingers. A simple classifier is defined for the digits 0 to 5 by counting the number of these points. However, this descriptor and classification scheme are already applied to Kinect sensor information for gesture recognition. So, instead, the Malima [4] paper is considered. As with the Du method, the scheme proposed in Malima has simple, yet robust feature selection and classification. In addition, it is scale, translation, and rotation invariant. However, their method is applied to images captured with an RGB camera, where segmentation is performed by thresholding the color channels based on skin color values. In principle, though, the circle-based descriptor is applicable to depth images as well. This is one of the descriptors implemented as part of the project. It is described in greater detail in the following section.

Unfortunately, although simple and relatively robust, the Du and Malima systems are limited to the counting digits, 0 to 5. They are both analyzing local contour features of the hand outline, but do not recognize the shape of the hand as a whole. So, more generalized gesture methods were explored.

One such paper is the covariance matrix approach in Guo [7]. This approach performs human action recognition by looking at a sequence of whole body silhouettes over time, a silhouette tunnel, captured by the Kinect. A 13 dimensional feature vector is defined, where 3 values are row, column, and time, 8 values are based on the shape of the silhouette (described later), and the last 2 are a measure of the temporal similarity. However, this feature vector is general enough to work with hand silhouettes in addition to full bodies. So, it is possible to modify the shape feature vector to work with static hand gestures by reducing the dimensionality. This can be accomplished by removing the time dependence and temporal similarity terms. The result is a generalized 10 dimensional feature vector applicable to static shape recognition. The specific implementation and classification details are defined in the next section.

Fourier descriptors were also researched. A nice survey of Fourier based shape representations is provided in [6]. Out of all the methods surveyed, the centroid (central) distance Fourier descriptor provides the best results. This method is then used by Kulshreshth [5] to perform gesture recognition with the Kinect. Surprisingly, however, they limit themselves to just recognizing digits as in finger counting. It is implied that the comparatively low resolution of the depth images from the Kinect is the limiting factor. In addition, there is no testing done on the optimal number of contour points to sample. So, we chose to explore the central distance Fourier descriptor further.

## 3 Methods

The development environment consisted of the Microsoft Kinect for image capture, the OpenCV library for image processing, OpenNI for interfacing with the Kinect, and a third party NITE library for hand tracking.

## 3.1 Preprocessing

The preprocessing steps include: image capture, hand localization, silhouette generation, and finding the center of the hand. There are three available stream sources for image capture from the Kinect: RGB, the raw infrared (IR) dot pattern, and the processed depth image.

**IR Stream:**

Our initial method used the raw IR stream for image capture. Hand localization is performed by cropping to a rectangle around the hand coordinates provided by the NITE library. A 3x3 median filter is applied to the resulting image, followed by global thresholding with Otsu's Method [8] using the openCV function, cv2.threshold().

Otsu's Method is a binary thresholding scheme that separates the intensity values into two classes (foreground and background in this case). The problem is that the intensity ranges of the two classes may overlap in the histogram of the image. Intuitively, the basic idea is to find the value which best splits the histogram into two distinct, well-separated histogram regions. Practically, this is done by finding the threshold value on the histogram, $k \in \{0 \ldots 255\}$, which maximizes the between-class variance, $\sigma^2(k)$, [9]:

$$\underset{k}{arg\,max}\, \sigma^2(k)$$

$$where\ \sigma^2(k) = P_B(k)P_F(k)(m_B(k) - m_F(k))^2$$

$$P_B(k) = \sum_{i=0}^{k} p_i \text{ = prob pixel assigned to background}$$

$$P_F(k) = \sum_{i=k+1}^{255} p_i = 1 - P_B \text{ = prob pixel assigned to foreground}$$

$$m_B(k) = \frac{1}{P_B}\sum_{i=0}^{k} ip_i \text{ = mean of the background pixels}$$

$$m_F(k) = \frac{1}{P_F}\sum_{i=k+1}^{255} ip_i \text{ = mean of the foreground pixels}$$

However, although processing the IR stream produces useful hand silhouette images, there is no depth information included. Segmentation is based on the intensity values of the IR image itself, which is quite dim. So, this segmentation method only provides good results in the case where the user's hand is separated far enough from the body and is shown against a white background. Instead, we switched to using the resolved depth stream.

**Depth Stream:**

Image Capture

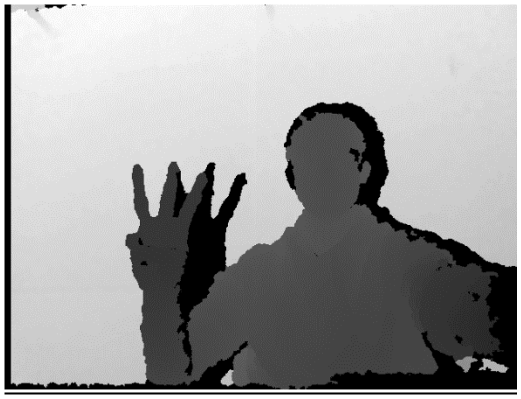

Fig 1 – Raw Depth Image

1. Capture a raw depth image from the Kinect and quantize to 0 through 255
2. Background Elimination [3]
    a. Invert the depth values
    b. Assign the background pixels to 0
    c. (0 = background, 255 = foreground)

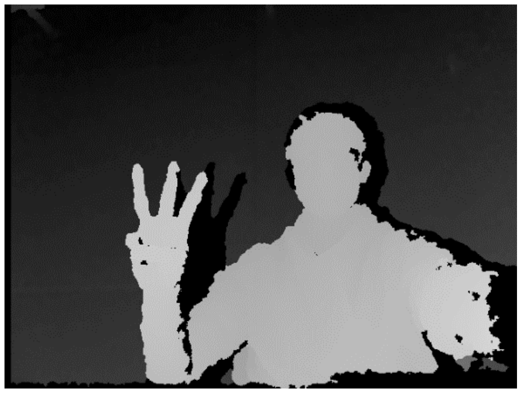

Fig 2 – Processed Depth Image

Hand Localization:

- Finer Thresholding
    a. Use hand location provided by NITE tracker **(Px, Py)**
    b. Find the average depth value in 3x3 window centered at (Px, Py)
    c. Threshold to a certain range around this average depth value (empirical)

d. Crop to box around hand using depth at (Px, Py) [5]

i. $Scale = \frac{13000}{Depth(Px,Py)}$

1. Constant 13000 found empirically
2. Proportional to the area of bounding box

ii. $Top\ Left\ Pixel = (Px - 10 * Scale * 0.5, Py - 10 * Scale * 0.6)$

1. This formula scales an initial 10x10 box centered at (Px, Py) to the appropriate dimensions based on the depth
2. Constants 0.5 and 0.6 take into account that most hand silhouettes are more tall than wide

iii. Width = Height = 10*Scale

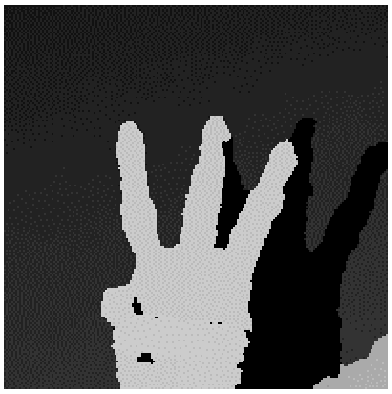

Fig 3 – Cropped Hand

Generate Silhouette:

1. Trace the contours using OpenCV's findContour function.
2. Only keep the contour with the largest area
3. Fill in the contour to generate a hand silhouette

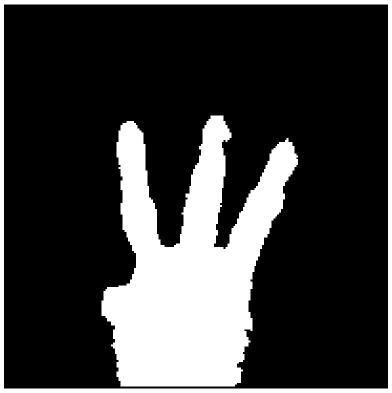

Fig 4 – Hand Silhouette

Locate Center of Hand Silhouette (COG):

$$x_{ctr} = \frac{\sum_{i=0}^{k} x_i}{k}, y_{ctr} = \frac{\sum_{i=0}^{k} y_i}{k}$$

where $(x_i,y_i)$ = silhouette coordinates, $(\mathbf{x_{ctr}}, \mathbf{y_{ctr}})$ = center of hand, and k = total number of pixels in the hand.

## 3.2 Circle Method

This detection method is based on Malima et al [4]. The idea is to draw a circle centered at $(x_{ctr}, y_{ctr})$, which cuts through all the fingers of the hand. The radius of this circle needs to be large enough that it encompasses more than just the palm, but small enough that no fingers are missed. The intersection of the circle with the silhouette forms the basis of the finger detection scheme. The circumference of the circle is traced for discrete values of theta from 0 to 360. For each theta, the algorithm assigns a value of 0 or 1 depending on whether or not the underlying pixel is in the silhouette of the hand. The result, shown in Fig 5, is a 1D vector (descriptor) for each gesture. (Note: The vectors on the right have been stretched out in two dimensions for visualization purposes only.)

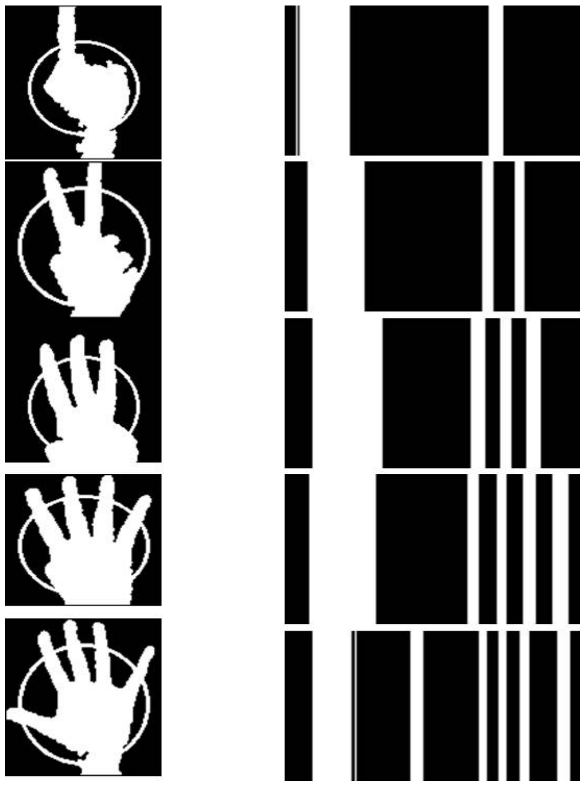
Fig 5 – Circle Method Descriptor

Classification Steps:

1. Find the distance between the center of the hand at ($x_{ctr}$, $y_{ctr}$) and the furthest point on the silhouette
2. Define five circles with radii scaled at 0.65, 0.67, 0.7, 0.73, 0.75 of this maximum distance
3. Generate feature vector defined previously for each circle
4. Count the number of discontinuities and map to the appropriate finger count for each circle
5. Apply majority rule to classify the gesture

Because the decisions are based on a scaled value of the distance computed in Step 2, this classification scheme is scale invariant. Rotation invariance is the result of using a circle for feature selection. This method is robust given a perfectly segmented hand. However, the classifier is limited to counting the numbers between 0 and 5 and not adaptable to more generalized gestures. Therefore, we tested other methods such as covariance matrix and Fourier descriptor.

## 3.3 Covariance Matrix Approach

As previously stated in the Literature Review, the Covariance Matrix approach proposed in Guo [7] can be easily adapted for use with static hand gestures.

**Create the Feature Vector**

1. Compute the 10-dimensional feature vector adapted from Guo [7]:
2. $\mathbf{f}(\mathrm{x,y}) = [\mathrm{x, y, de, dw, dn, ds, dne, dsw, dse, dnw}]$
   - x = col, y = row
   - d = Euclidean distance from (x,y) to the nearest boundary point in the specified direction
     i. east, west, north, south, northeast, southwest, southeast, northwest
3. Scale Invariance
   - Divide each spatial feature by the square root of the silhouette area
4. Compute the Covariance Matrix:
   - $cov(f(\boldsymbol{S})) = \frac{1}{|\boldsymbol{S}|}\sum_{(x,y)\in \boldsymbol{S}}(f(x,y) - \mu_F)(f(x,y) - \mu_F)^T$
   - $\boldsymbol{S} =$ area of silhouette
   - $\mu_F = \sum_{(x,y)\in \boldsymbol{S}} \frac{1}{|\boldsymbol{S}|}\mathrm{f(x,y)} =$ mean feature vector for silhouette

**Build a Dictionary**

For a set of images of the same gesture, compute the Covariance Matrix and add it to the dictionary with the same label. Repeat for all desired gestures.

**Classification**

As noted in Guo [6], the set of all covariance matrices lie on a Riemannian manifold. So, a Euclidean distance measure cannot be used. Instead, the distance between two covariance matrices on this manifold is defined as:

$$d(C, C') = \sqrt{\sum_{k=1}^{10} (\ln \lambda_k(C, C'))^2}$$

$$where\ \lambda_k(C,C')\ are\ the\ generalized\ eigenvalues\ of\ C\ and\ C'$$
$$C = Covariance\ Matrix\ of\ new\ sample, C' = Reference\ from\ Dictionary$$

The label of the minimum distance gesture is used for classification.

| **Pros:** | **Con:** |
|---|---|
| 1. Complex Gestures<br>2. High accuracy<br>3. Scale invariant | 1. Not rotation invariant |

## 3.4 Fourier Shape Descriptor

The Fourier Shape descriptor used in this paper is based on the work in Kulshreshth [5]. The basic idea is to first create a set of Euclidean distances between the center of the hand and equidistant points along the contour [see Fig 6]. The magnitude of the Fourier Transform of this set forms a unique shape signature, which can be used for generalized gesture classification.

In addition, this descriptor is rotationally invariant. Shifts in the silhouette contour points, which is the cause of rotation, will be appear as phase delays in frequency domain. However, since only the magnitude of the Fourier coefficients is considered, the phase (or equivalently, the rotation) is ignored. So, this method is rotationally invariant while remaining computationally fast.

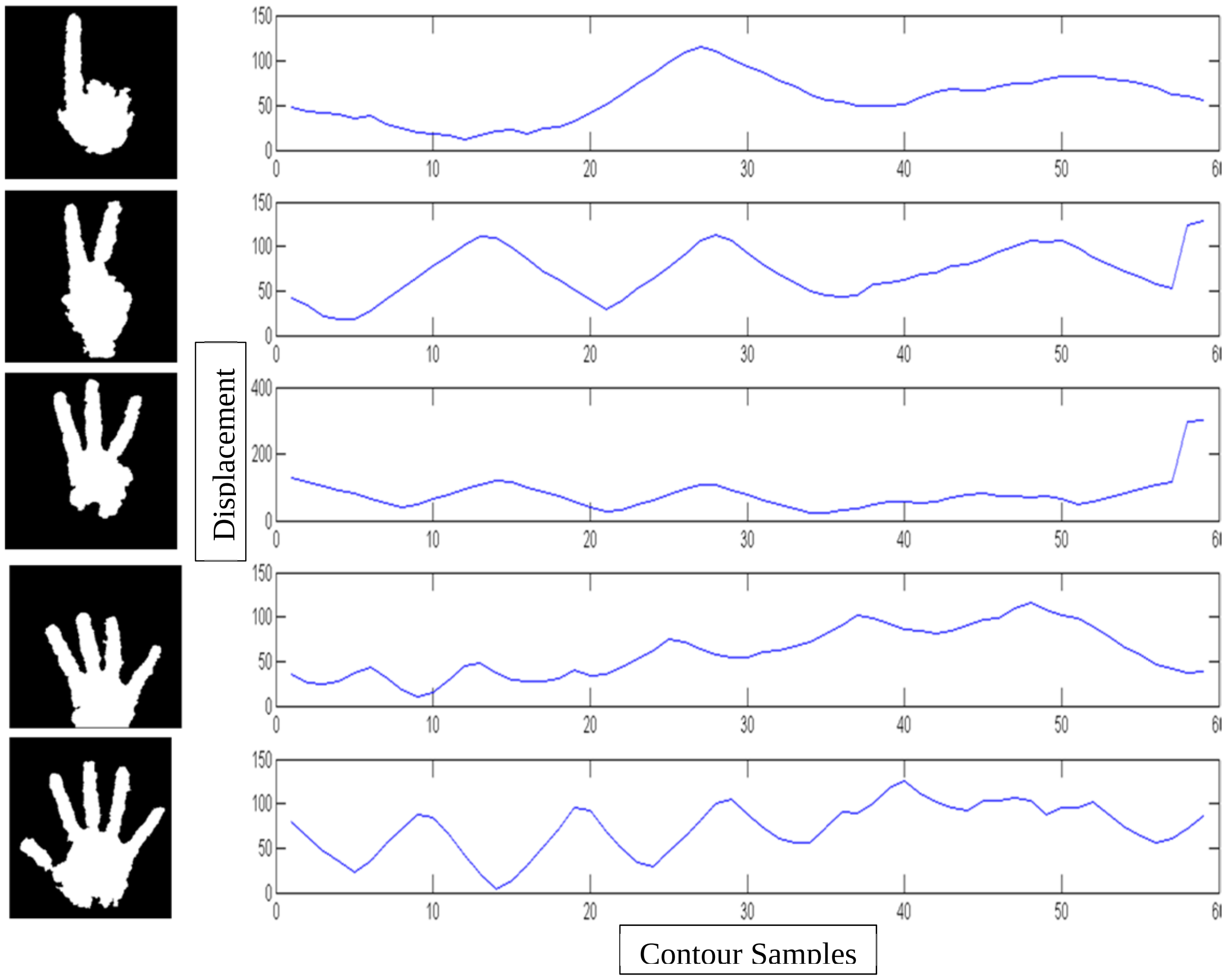

Fig 6 – Centroid Displacement Diagram (Full contour before sampling)

**Generating the Fourier Descriptor:**

1. Sample N = 16 equidistant points along the full contour of the hand (Fig 6)
2. Calculate Euclidean distance between each sample point and the center of the hand

$$r[n] \text{ for n = 1…N}$$

3. Take the magnitude of the N point DFT of these points

$$abs(FT\{r[n]\}) = a[m] \; for \; m = 1…N$$

4. Normalize the Fourier coefficients by the DC value (Scale Invariance)
5. Keep the first 7 normalized coefficients (skip DC, which is always 1)

This is the Fourier Descriptor for a single shape.

**Make a Dictionary**

For a set of images of the same gesture, compute the average Fourier shape descriptor and add it to the dictionary with the same label. Repeat for all desired gestures.

**Classification**

Compute the Fourier descriptor for each new sample. Compare it with each stored gesture in the dictionary using a Euclidean distance measure. The label of the minimum distance is the desired gesture.

## 4 Experimental results

**Comparison between Covariance Matrix and Fourier Descriptor**

Testing Setup:

- Covariance Matrix and Fourier Descriptor Methods implemented in MATLAB
- 600 images = 6 gestures, labeled 0 to 5, 100 images per gesture
- Vertical Orientation (No Rotation)

Confusion matrices for the Covariance Matrix and Fourier Descriptor Methods:

Predicted (CCR = 89.5%)

| Actual Data | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0.85 | 0 | 0.11 | 0 | 0.04 |
| 2 | 0 | 0 | 0.85 | 0.15 | 0 | 0 |
| 3 | 0 | 0 | 0.13 | 0.87 | 0 | 0 |
| 4 | 0 | 0.08 | 0 | 0 | 0.84 | 0.08 |
| 5 | 0 | 0 | 0 | 0.04 | 0 | 0.96 |

Table 1 - Confusion Matrix for Covariance Matrix Method

Predicted (CCR = 90. 6%)

| Actual Data | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 | 0 | 0 | 0 |
| 2 | 0 | 0.04 | 0.85 | 0.11 | 0 | 0 |
| 3 | 0 | 0 | 0 | 0.88 | 0.12 | 0 |
| 4 | 0 | 0 | 0 | 0.07 | 0.93 | 0 |
| 5 | 0 | 0 | 0 | 0 | 0.22 | 0.78 |

Table 2 - Confusion Matrix for Fourier Descriptor

Results:

As is evident in the tables above, both methods are quite effective. However, during additional testing, we found that rotation in the input images decreased the effectiveness of the covariance matrix method. This result is reasonable since the covariance matrix is not rotationally invariant. One simple solution is to first find the orientation of the hand, re-center the silhouette vertically, and then run the covariance matrix method. However, because the Fourier descriptor is already rotationally invariant and much faster computationally, it was chosen as the primary shape signature and subsequently ported to C++/OpenCV for real time classification with the Kinect.

**Determining the Optimal DFT Length**

Testing Setup (MATLAB):

- 600 images = 6 gestures, labeled 0 to 5 (Same as previous test)
- N = 16, 32, 64, and 128 point DFT
- N/2 – 1 = 7, 15, 31, 63 Coefficients kept

Gestures

N, Fourier Transform length

| | 0 | 1 | 2 | 3 | 4 | 5 | CCR |
|---|---|---|---|---|---|---|---|
| 16 | 1 | 1 | 0.85 | 0.88 | 0.93 | 0.78 | 0.906 |
| 32 | 1 | 1 | 0.88 | 0.80 | 0.74 | 0.66 | 0.85 |
| 64 | 1 | 1 | 0.88 | 0.73 | 0.29 | 0.25 | 0.69 |
| 128 | 0.73 | 0.87 | 0.46 | 0.46 | 0.03 | 0.14 | 0.45 |

Table 3 - Decisions made with N/2-1 of Coefficients

Results:

The 16 point DFT results in the highest correct classification rate (CCR), where CCR is calculated as the average of the values in each row (i.e. across all gestures). Interestingly, additional testing revealed that the global optimum for our testing data was actually an 18 point DFT. However, we kept N = 16, a power of 2, to maximize the efficiency of the Fourier transform computation.

Recall that N is also the number of points along the contour of the hand silhouette. Surprisingly, increasing the number of samples actually decreases the CCR. This is a bit counter intuitive. So, we decided to investigate methods for utilizing a higher order DFT while maintaining lower dimensionality.

## Investigating Fourier Coefficient Pruning

One suggested alteration is to only keep the first few coefficients of a 128 point DFT. The idea is that throwing out Fourier coefficients of such a high order DFT will make the overall contours smoother (remove noise), while retaining the overall shape.

| Feature vector length | Gesture 0 | 1 | 2 | 3 | 4 | 5 | CCR |
|---|---|---|---|---|---|---|---|
| 7 | 0.80 | 0.76 | 0.42 | 0.38 | 0.07 | 0.04 | 0.41 |
| 15 | 0.76 | 0.76 | 0.42 | 0.38 | 0.07 | 0.04 | 0.40 |
| 31 | 0.71 | 0.75 | 0.41 | 0.40 | 0.07 | 0.04 | 0.40 |
| 63 | 0.73 | 0.87 | 0.46 | 0.46 | 0.03 | 0.14 | 0.45 |

Table 4 - Results of 128 Point DFT Coefficient Pruning

Results:

From the table above, it appears that the suggested alteration is ineffective. One reason may be that there simply aren't enough contour points (i.e. N > # contour points). For example, when the hand is far from the camera, the resulting silhouette is small, resulting in fewer contour points.

**Additional Gestures (beyond counting fingers)**

Testing Setup (C++/OpenCV):

- Fourier Descriptor
- 4000 images, 10 gestures, 400 images per gesture Dictionary
- IR Depth Images (prior to using the Depth Stream)
- Real time testing with the Kinect using IR segmentation scheme

Results:

The Fourier shape description method was implemented using the IR depth data. The segmentation was poor when the user held the hand in front of the body. However, for the limited case where the hand was positioned away from the body against a remote background, we were able to create a fairly accurate gesture recognition system which included more generalized shapes. This was due in part to the comparatively higher resolution silhouettes captured by the processed IR images. The lack of depth integration in the segmentation ultimately forced us to abandon this approach. However, it provided compelling evidence that the Fourier descriptor has great potential when paired with a higher resolution depth sensor.

# 5 Conclusions

In conclusion, the centroid Fourier descriptor method was able to perform gesture recognition with a CCR of 90.6% in MATLAB. Furthermore, we were able to implement it in C++/OpenCV to run in real time. In addition, we successfully implemented temporal dampening to minimize fluctuations in the classification by buffering the results and showing the mode over the last 30 frames (one second).

Unfortunately, because of time constraints, we were unable to add more gestures beyond the usual counting numbers from 0 to 5 after switching from the IR channel to the Depth stream. The only additional gestures added were alternate versions of the same 0 to 5 digits to account for the different ways that people might perform those gestures. Even so, we believe that this codebase could form the basis of a generalized gesture recognition system. Unlike the Du and Malima methods, the Fourier descriptor is looking at the overall shape, not just local contour information. So, the fact that it can successfully resolve any group of 6 shapes suggests that, in principle, there should be no issues adding any new gestures to the dictionary (provided they have a different enough silhouette).

## 5.1 Future Plans

We foresee several avenues for improvement:

- Applying a threshold method for unassigned gestures
- Ability to add new gestures to the dictionary
- Trying a new distance metric (Mahalanobis)
- Trying a more sophisticated Classification scheme
- Using two hands simultaneously for combined gestures
- Adding more gestures from the single hand sign language alphabet

## References

[1] Jungong Han; Ling Shao; Dong Xu; Shotton, J., "Enhanced Computer Vision With Microsoft Kinect Sensor: A Review," *Cybernetics, IEEE Transactions on* , vol.43, no.5, pp.1318,1334, Oct. 2013

[2] Suarez, J.; Murphy, R.R., "Hand gesture recognition with depth images: A review," *RO-MAN, 2012 IEEE* , vol., no., pp.411,417, 9-13 Sept. 2012

[3] H. Du and T. To, "Hand Gesture Recognition Using Kinect," Boston University, 2011.

[4] Malima, A.; Ozgur, E.; Cetin, M., "A Fast Algorithm for Vision-Based Hand Gesture Recognition for Robot Control," *Signal Processing and Communications Applications, 2006 IEEE 14th* , vol., no., pp.1,4, 17-19 April 2006

[5] Kulshreshth, A.; Zorn, C.; LaViola, J.J., "Poster: Real-time markerless Kinect based finger tracking and hand gesture recognition for HCI," *3D User Interfaces (3DUI), 2013 IEEE Symposium on* , vol., no., pp.187,188, 16-17 March 2013

[6] D. Zhang and G. Lu, “A comparative study of Fourier descriptors for shape representation and retrieval,” in Proc. 5th Asian Conference on Computer Vision, 2002.

[7] Kai Guo; Ishwar, P.; Konrad, J., "Action Recognition in Video by Covariance Matching of Silhouette Tunnels," *Computer Graphics and Image Processing (SIBGRAPI), 2009 XXII Brazilian Symposium on* , vol., no., pp.299,306, 11-15 Oct. 2009

[8] A Threshold Selection Method from Gray-Level Histograms," *Systems, Man and Cybernetics, IEEE Transactions on* , vol.9, no.1, pp.62,66, Jan. 1979

[9] Rafael C. Gonzalez and Richard E. Woods. 2006. *Digital Image Processing (3rd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

## Appendix

```
#include <opencv\cv.h>
#include <opencv\highgui.h>
#include <iostream>
#include <OpenNI.h>
#include <opencv2\imgproc\imgproc.hpp>
#include <NiTE.h>
#include <NiteSampleUtilities.h>
#include <string>
#include <sstream>
#include <iomanip>
#include <cmath>

using namespace cv;
using namespace std;
using namespace openni;
using namespace nite;

Point findCenter(Mat);
Mat calcFourierDescriptor(Mat);
int classifier(vector<Mat>, Mat);

int main(){
     openni::Device device;
    openni::Status rc = openni::STATUS_OK;
    VideoStream ir;       // IR VideoStream Class Object
    VideoFrameRef irf;    //IR VideoFrame Class Object
    VideoMode vmode;      // VideoMode Object

     nite::HandTracker handTracker;
     nite::Status niteRc;
     niteRc = handTracker.create();
     handTracker.setSmoothingFactor(0.0f);
     handTracker.startGestureDetection(nite::GESTURE_WAVE);
//   handTracker.startGestureDetection(nite::GESTURE_CLICK);
     handTracker.startGestureDetection(nite::GESTURE_HAND_RAISE);
     nite::HandTrackerFrameRef handTrackerFrame;


rc = openni::OpenNI::initialize();    // Initialize OpenNI
rc = device.open(openni::ANY_DEVICE); // Open the Device
rc = ir.create(device, openni::SENSOR_DEPTH);    // Create the VideoStream for IR
    rc = ir.start();                      // Start the IR VideoStream

     Mat result(240, 320, CV_8UC1, cv::Scalar::all(0));              // Display
detected Gesture

     int h, w;                                                           // Height
and Width of the IR VideoFrame
     const uint16_t* imageBuffer;
     double maxVal=0,minVal=0;
     RNG rng(12345);

     bool keepTracking = true;
     int fileNum = 0;

///////////////////////////////////////
//   Load Library
///////////////////////////////////////

     FileStorage lib("lib.xml", FileStorage::READ);

     vector<Mat> dictionary;
     lib["trainingLibrary"] >> dictionary;         // load library into mat vector
     lib.release();


//////////////////////////////////////
// Continually Process images
///////////////////////////////////////

// Buffer size
int bufferSize = 30, nextIdx = 0, maxGestures = 20;
std::vector<int> buffer;
buffer.reserve(bufferSize);
std::vector<int> hist(maxGestures, 0);

while(1)
{
     niteRc = handTracker.readFrame(&handTrackerFrame);
     rc = ir.readFrame(&irf);   // Read one IR VideoFrame at a time

     vmode = ir.getVideoMode(); // Get the IR VideoMode Info for this video stream.
                             // This includes its resolution, fps and stream format.


          Mat depth(480,640,CV_16UC1,(void*)irf.getData(), 2*640);

          Mat frame;
          cv::normalize(depth, frame, 0, 255, CV_MINMAX, CV_8UC1);
          frame = 255 - frame;

          // shadow elimination
          for(int ii = 0; ii < frame.rows; ii++)
               for(int jj = 0; jj < frame.cols; jj++ )
                    if(frame.at<uchar>(ii,jj) == 255)
                         frame.at<uchar>(ii,jj) = 0;
          double minVal, maxVal;
          minMaxLoc(frame, &minVal, &maxVal, NULL, NULL);

          const nite::Array<nite::GestureData>& gestures =
handTrackerFrame.getGestures();
          for (int i = 0; i < gestures.getSize(); ++i)
          {
               if (gestures[i].isComplete())
               {
                    nite::HandId newId;
                    handTracker.startHandTracking(gestures[i].getCurrentPosition(),
&newId);
```

```
                }
            }

            const nite::Array<nite::HandData>& hands = handTrackerFrame.getHands();
            for (int i = 0; i < hands.getSize(); ++i)
            {
                const nite::HandData& hand = hands[i];
                if (hand.isTracking() && keepTracking == true)
                {

                float pX = 0, pY = 0;
                    handTracker.convertHandCoordinatesToDepth(hand.getPosition().x,
hand.getPosition().y, hand.getPosition().z, &pX, &pY);
                    float zz = hand.getPosition().z;

        Point ctrLoc = Point(pX, pY);              // hand location
        int ctrVal = frame.at<uchar>(ctrLoc);            // depth value at hand location
                    double meanVal = 0;

for(int ii = -1; ii <= 1; ii++)
       for(int jj = -1; jj <= 1; jj++)
            if(((pX + ii) < frame.cols) && ((pX + ii) >= 0) && ((pY + jj) < frame.cols)
&& ((pY + jj) >= 0))
            meanVal += frame.at<uchar>(Point(pX + ii,pY + jj));

            meanVal /= 9;

                    // Extract window around detected hand
        double winConst = 13000.0f; // defines the relative size of the bounding box
        double scale = winConst/zz;

                    // Define coords for bounding box
        double colStart = max(0, pX - 10*scale*0.5);
        double colEnd = min(frame.cols, colStart + 10*scale);
        double rowStart = max(0, pY - 10*scale*0.6);
        double rowEnd = min(frame.rows, rowStart + 10*scale);

                    // crop image to roi
                    Mat num = frame.rowRange((int)rowStart,
(int)rowEnd).colRange((int)colStart,(int)colEnd);

                    Mat wdw;
                    num.copyTo(wdw);        // make working copy
                    wdw.convertTo(wdw, CV_8U);

                    vector<vector<Point> > contours;
                    vector<Vec4i> hierarchy;
                    threshold( wdw, wdw, meanVal*0.98f, 255, CV_THRESH_BINARY );

/////////////////////////////////////
//   Compute Fourier Descriptor
/////////////////////////////////////

Mat ft;                     // Fourier descriptor vector
int dist;               // store the label of the minimum distance neighbor

ft = calcFourierDescriptor(wdw);// compute the Fourier Descriptor for this image
```

```
/////////////////////////////////////
//   Find Nearest Neighbor
/////////////////////////////////////

dist = classifier(dictionary, ft);     // returns detected gesture from dictionary

// Implement Buffer for "temporal dampening" (i.e. take mode of values in the buffer)
         if (nextIdx >= buffer.size())   {// initial fill of the first bufferSize
number of values
         buffer.push_back(dist);
         hist[dist]++;      // produce histogram, 0 = unknown gesture
                  }
         else {        // subsequent overwrites
     hist[buffer[nextIdx]]--;    // delete previously found index from histogram
     buffer[nextIdx] = dist;     // overwrite buffer with new value
     hist[dist]++;          // update the histogram
                  }

                  ++nextIdx;

         if (nextIdx >= bufferSize)      // reset the next index (circle back)
              nextIdx = 0;

// Find the mode of the histogram (should be fast, since histogram size is limited to
maxGestures, which is typically small
                  dist = std::max_element( hist.begin(), hist.end() ) - hist.begin();


////////////////////////////////////////
// Output results to the Screen
///////////////////////////////////////

                  result.setTo(cv::Scalar::all(0));        // clears the output
window

                  ostringstream displaystream;                      // output
stream
                  if (dist == 6)
                       displaystream << "1";
                  else if(dist >= 1 && dist <= 5)
                       displaystream << dist;
                  else if(dist == 7)
                       displaystream << "3";
                  else if(dist == 8)
                       displaystream << "3";
                  else if(dist == 9)
                       displaystream << "0";
                  else if(dist == 13)
                       displaystream << "2";
                  else if(dist == 12)
                       displaystream << "2";
                  else if(dist == 11)
                       displaystream << "1";

                  putText(result, displaystream.str(), Point(75,225),
FONT_HERSHEY_COMPLEX, 10.0, cvScalar(200,0,0), 4, CV_AA);
         namedWindow("Result", 2);      // Create a named window
         imshow("Result",result);       // Show the IR VideoFrame in this window
         namedWindow("drawing", 2);       // Create a named window
         imshow("drawing",wdw);       // Show the IR VideoFrame in this window

     char key = waitKey(10);
     if( key == 27 )        // escape key
     break;
     else if(key == 32){    // space bar
     ostringstream fileName;
     fileName << "../Data/" << setw(5) << setfill('0') << fileNum++ << ".tiff";
     imwrite(fileName.str(), wdw);
                   }
              }
              else
              {
     char key = waitKey(10);
     cvDestroyWindow("Result");
     cvDestroyWindow("drawing");}
          }

         // Convert to 8-bit data for display
         frame.convertTo(frame, CV_8U);

         char key = waitKey(10);

         if( key == 27 )        // escape key
              break;

        namedWindow("frame",1);      // Create a named window
        imshow("frame", frame);       // Show the IR VideoFrame in this window

    }

    ir.stop();                               // Stop the IR VideoStream
    ir.destroy();
    device.close();                           // Close the PrimeSense Device
     nite::NiTE::shutdown();

     return 0;
}         // end main()


//////////////////////////////////
// Find center of mass for hand silhouette
//////////////////////////////////

Point findCenter(Mat wdw)   {

     // find the center position of the hand
     double colSum = 0.f, rowSum = 0.f, totSum = 0.f;
     int ctrRow = 0, ctrCol = 0;
     Point ctrPt;

     for(int ii=0; ii < wdw.cols; ii++){
          for(int jj=0; jj < wdw.rows; jj++){
               if (wdw.at<uchar>(Point(ii,jj))!=0){
                    colSum = colSum + ii;
                    rowSum = rowSum + jj;
                    totSum++;
               }
          }
     }
     // center of the hand in cartesian coords
     ctrPt.x = static_cast<int>(colSum/totSum);
     ctrPt.y = static_cast<int>(rowSum/totSum);

     return ctrPt;
}         // end findCenter()

Mat calcFourierDescriptor(Mat wdw)   {

     // Find the center position of the hand
     Point ctrPt;
     ctrPt = findCenter(wdw);

     // Find the contours of the silhouette
     vector<vector<Point> > contours;
     vector<Vec4i> hierarchy;
     findContours( wdw, contours, hierarchy, CV_RETR_EXTERNAL, CV_CHAIN_APPROX_SIMPLE
);

     /// Approximate contours to polygons + get bounding rects and circles
     vector<vector<Point> > contours_poly( contours.size() );
     vector<Rect> boundRect( 1 );

     double maxArea = 0.0f;      // store the max area of all the convex hulls
     int maxInd = 0;                  // store contour id of max area

     for( int i = 0; i < contours.size(); i++ )
     {
          approxPolyDP( Mat(contours[i]), contours_poly[i], 3, true );
          double currArea = cv::contourArea(contours_poly[i]);
          if( currArea > maxArea) {
               maxInd = i;
               maxArea = currArea;
          }
     }

     int lineThickness = -1;

     // blank out the window
     wdw(Range::all(),Range::all()) = 0;

     // maxInd = largest contour
     drawContours( wdw, contours, maxInd, 255, lineThickness, 8, vector<Vec4i>(), 0,
Point() );

     Mat frame;
     wdw.copyTo(frame);          // Make working copy


////////////////////////
// Repeat previous steps
///////////////////////

     ctrPt = findCenter(frame);
     // Find the contours of the silhouette
     contours.clear();
     hierarchy.clear();

     findContours( frame, contours, hierarchy, CV_RETR_EXTERNAL,
CV_CHAIN_APPROX_SIMPLE );

     // Remove contours that are along the bottom edge of the window
     size_t dim1 = contours[0].size();                              // number of
contour points
     int r = wdw.rows, c = wdw.cols;

     for(int ii = 0; ii < dim1; ii++)
          if((contours[0][ii].x == 0) || (contours[0][ii].x == c-1) ||
(contours[0][ii].y == 0) || (contours[0][ii].y == r-1) )
               contours[0].erase (contours[0].begin()+ii);

     // Find equidistant points along the contour
     int N = 16;         // N point DFT
//   size_t dim1 = contours.size();
     size_t dim2 = contours[0].size();                              // number of
contour points
     int shift = static_cast<int>(floor((dim2/(N)) + 0.5));      // delta between
contour samples along the contour
     Vector<Point> K;                                            // vector to store
the sample contour points

     // Find and store equidistant points along the contour
     for(int ii = 0; ii < N; ii++)   {                              // Get N
samples along the contour
          if(shift*ii > dim2)        {
               break;
          }
          else {
               K.push_back(contours[0][shift*ii]);

          }
     }
     vector<double> rt;     // Euclidean distance vector
     for(int ii = 0; ii < K.size(); ii++) {
          // find distance from each point to the hand center
          K[ii] -= ctrPt;
          // compute Euclidean distance, r(t)
          rt.push_back(norm(K[ii]));
     }

     // take the magnitude of the N point FFT
     Mat img = Mat(rt);
     Mat planes[] = {Mat_<double>(img), Mat::zeros(img.size(), CV_64F)};
     Mat complexI;
     merge(planes, 2, complexI);
     dft(complexI, complexI, DFT_COMPLEX_OUTPUT);
     split(complexI, planes);    // planes[0] = Re(DFT(I), planes[1] = Im(DFT(I))
     magnitude(planes[0], planes[1], planes[0]);       // planes[0] = magnitude
     Mat f_all = planes[0].t();

     // Normalize by the f0 (DC value)
     f_all /= f_all.at<double>(0);    // Note:  This depends on the data type of
planes[] (BOTH parts of the constructor)

// Only store the first (N/2) unique coefficient values (excludes the DC coeff, which
is always 1)
     // Ex:  If N = 16, then keep 8 of the coeffs.
     int offset = static_cast<int>(floor((N/2) + 0.5)+1);        // offset =
round(N/2) + 1;
     Mat ft = f_all.colRange(Range(1,offset));                   // Note: Adding 1 to
get all unique coeffs


     return ft;
}         // end calcFourierDescriptor()

//////////////////////////////////////
//   Basic classifier using Euclidean distance
//        Inputs:
//   lib = Lib loaded from premade dictionary
//   src = Fourier descriptors for input image
//        Outputs:
//   Mat = Euclidean distance between the src Fourier descriptor and each label in the
dictionary
//////////////////////////////////////

int classifier(vector<Mat> dict, Mat src){

     int ctr = 1;
     int minIdx = -1;
     double minVal = 999999999;
     double currVal;

     // Iterate through Dictionary
     for (vector<Mat>::iterator it = dict.begin() ; it != dict.end(); ++it)    {
          currVal = norm(*it, src, NORM_L2);        // compute Euclidean distance

          if(currVal < minVal)   {
               minVal = currVal;
               minIdx = ctr;
          }
          ctr++;
     }
     return minIdx;
}         // end classifier()
```