Articulated Manipulator for Visually Impaired People (AMVIP)

Viet Trinh
Jack Baskin School of Engineering
University of California, Santa Cruz
California, USA, 95060
vqtrinh@ucsc.edu

Abstract—Assistive technology for visually impaired (VI) people has recently received tremendous attention in the industry: from a start-up to a multinational corporation. However, these efforts mostly concentrate in self-localization and guidance to specific destinations in a verbal format, but are unaware of other communication channels (i.e.: languages). A desire to help a VI individual communicating confidently with people having other disabilities is the main purpose of this project. The project aims to introduce an articulate manipulator for visually impaired people (AMVIP) that takes input as a sign language and outputs Braille characters for VI people. In the scope of kinematics in robotics and gesture recognition in computer vision, this report mainly focuses on spatial transformations, manipulator forward kinematics, manipulator inverse kinematics, and the control algorithm; including a 3D pose localization and gesture recognition.

Index Terms—robotics, mechanics, forward kinematics, inverse kinematics, computer vision, gesture recognition

1 Introduction

The project proposes a simple artificial eye-hand coordination system that processes a visual input, through a video stream captured from camera, to guide a rigid body's movement. The system contains a physical 3 degree-of-freedom robotic arm (Fig. 1), acting as a hand, and a mobile's camera, playing a role of an eye. More specifically, a user performs an elementary set of hand gestures in sign language (ie: moving fingers to indicate "CMPE 215"), the system is able to recognize and translate those into Braille by dotting its end-effector on an arbitrary surface (eg: writing as Fig 2 on a piece of paper).

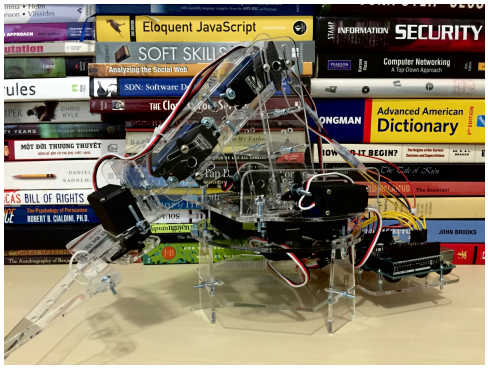

Fig. 1: The artificial robotic arm

The robotic arm's location in space is calculated from inverse kinematics and spatial frames transformation [6] [2]. In the scope of robotics, this report will focus on manipulator forward and inverse kinematics. More specifically, an iPhone is placed on a tabletop with its camera facing up, as an user moving his or her hand over a device in a real time, the control algorithm subtracts the background and detects shown sign-language characters. Once a sign-language character is determined, its corresponding Braille character is looked up from a pre-define database, and the next 3D pose location of the arm's end-effector is calculated. This information, is then fed into a set of inverse kinematics equations for solving joint angles. Last, these rotation angles are communicated to the arm via bluetooth BLE 4.0 channel for navigating the end-effector to a specific location.

Fig. 2: "CMPE 215" in Braille

This report is structured as follows: the next section studies the state-of-the-art related works and their limitation in solving the proposed problem. Section 3 introduces the system architecture in general, both hardware configuration and software development. Section 4 discusses the proposed methodology and main components of AMVIP, including localizing coordination in space, data communication between an iPhone and the arm's microprocessor, and gesture recognition technique. Section 5 briefly mentions implementation (i.e.: libraries, platform, and programming languages) containing a walk-through the source code solution. Section 6 analyzes obtained results and discusses possible

errors. Last, section 7 concludes the author's work with current limitation in the proposed approach and future improvement.

2 Literature Review

Attempts to applying gesture recognition into robotics are not new in computer vision. Linglong Lin and his colleagues developed a system to identify and locate the fragments semi-automatically [2]. Their manipulator has two-cameras mounted on two sides of its clamp, extracts region of interests (ROI) from double calibrated image pair, and applies edge filter to select desirable contours. However, this does not work in the proposed application as built-in cameras have low resolutions and lose a general scene description. Another approach for controlling a robotic arm with stereo cameras in introduced by Roland Szabo and his group in [1]. Their algorithm detects certain key parts on an arm, creates a robotic skeleton, computes angles of joints for arm's movement. Their approach has a certain limitation: each key parts must be color-coded, a detect algorithm is based on color recognition, the system requires at least two cameras. Also, it will not work in the author's proposed application, as a color of human skin is uniform; color-recognition algorithm will fail miserably. On another attempt, Chaudhury tracks hands through the video sequence by incorporating both color and depth information from a range camera (Kinect) [5]. The background subtraction models come from OpenCV library, namely BackgroundSubtractorMOG and BackgroundSubtractorMOG2, calculate the foreground mask performing a subtraction between the current frame and a background model. Although this technique yields a great result in background subtraction, it performs poorly in this case. As the background model must be a static scene, while gesture detection involves a video stream or a set of motion frame.

3 System Architecture

AMVIP contains two separate and independent elements: a physical manipulator and a control algorithm. The manipulator hardware (base and arms) is laser-cutted from 3-8mm acrylic sheets, following a provided CAD schema. Its assemble materials include an Arduino Uno board as a microprocessor, BLE 4.0 shield for wirelessly communication with a smartphone, 6 standard Hitec servos actings as revolute joints, and 12V power source. Fig 1, 3, 4 show a front view, a side view, and a top view of AMVIP's manipulator. More specifically, the base of the arm is made from 8mm sheets for a better balance, all other parts (eg: arm, wrist, gripper) are made from 3mm sheets. For dotting Brailled characters on a piece of paper, a marker is attached to the gripper so that the marker's tip directly aligns with the end-effector's. The Arduino BLE 4.0 shield is connected on top of Arduino Uno board for transmitting and receiving electronic signals. The standard Hitec servo has a rotation speed of 0.15/60 second/degrees and a torque of 3.7 kg/cm at 6.0V, a size of 39.88 x 19.81 x 36.32 millimeters, and a weight of 42.81 grams. The range of its rotation angle is from 0 to 180 degrees. To secure all parts, the hardware requires pieces of 3mm x 10mm screw, 3mm x 15mm screw, 3mm x 30mm screw, 3mm washers, 3mm nuts, and 3mm lock nuts. Overall, the physical manipulator has a size of 420 x 216 x 162 millimeters and an approximate weight of 300 grams.

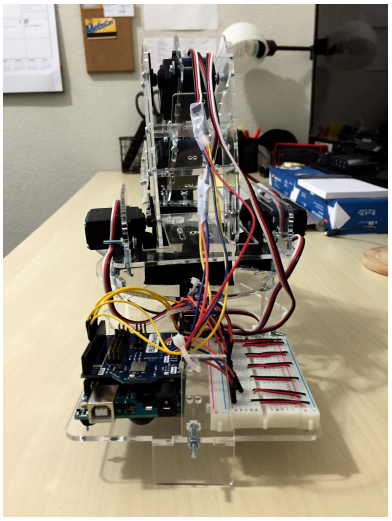

Fig. 3: AMVIP: Side View

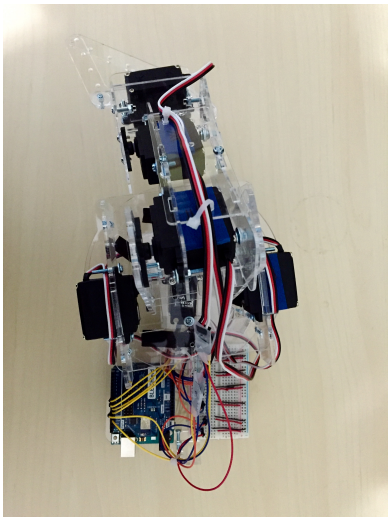

Fig. 4: AMVIP: Top View

The second element, a control algorithm, is implemented in a combination of different programming languages with supports from external libraries. To recognize finger's movement on an iPhone camera, a computer vision libraries, named OpenCV, and iOS SDK for mobile application development are employed. To locate the end-effector in place, inverse kinematics solving algorithm and coordinate frame transformation are implemented in C++. Sending data back and forth between

those two elements, as well as controlling Arduino Uno board's activities, C is the solely programming language in charge. Fig 5 illustrates the entire AMVIP systems.

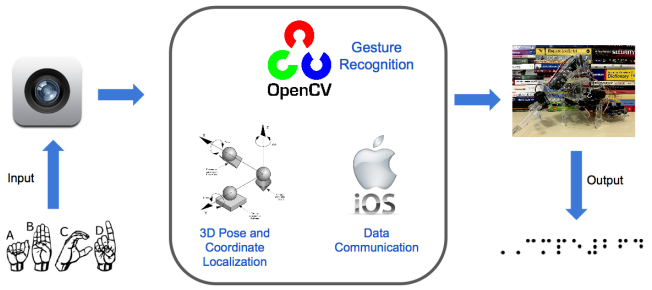

Fig. 5: The diagram illustrates AMVIP behavior

4 METHODOLOGY

The system pipeline (Fig 6) of AMVIP includes 3 components: Gesture Recognition, Coordinate Localization, and Data Communication. The first component is responsible for background subtraction and gesture detection, whenever it receives an incoming image stream from camera. The other two components are in charge to calculate and locate the arm's gripper to a spatial 3D pose for dotting Braille characters on a planar surface. In fact, these are the main discussion in this paper.

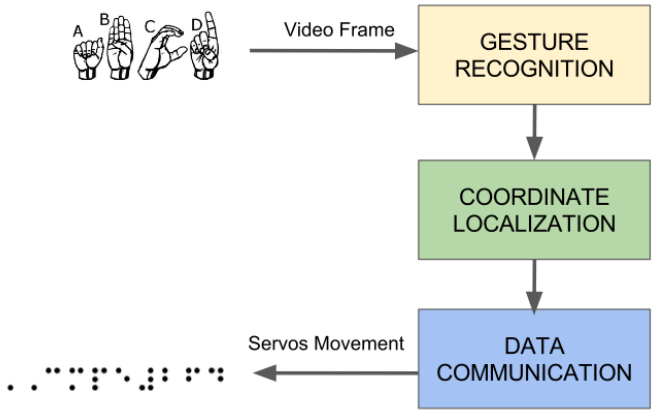

Fig. 6: System pipeline of AMVIP

4.1 Coordinate Localization & Data Communication

The kinematic parameters and frame assignments[1] of the AMVIP arm are shown in Fig 7. Let ${}^{i-1}_{i}T$ denotes a relatively transformation from frame $\{i\}$ to frame $\{i-1\}$. The Denavit-Hartenberg (DH) link parameters are defined as follows:

- a_{i-1}: distance from Z_{i-1} to Z_i measured along X_{i-1}
- α_{i-1}: angle from Z_{i-1} to Z_i measured about X_{i-1}
- d_i: distance from X_{i-1} to X_i measured along Z_i
- θ_i: angle from X_{i-1} to X_i measured about Z_i

1. Mathematical notation used in this paper is same as written in the text book Introduction to Robotics: Mechanics and Control [6]

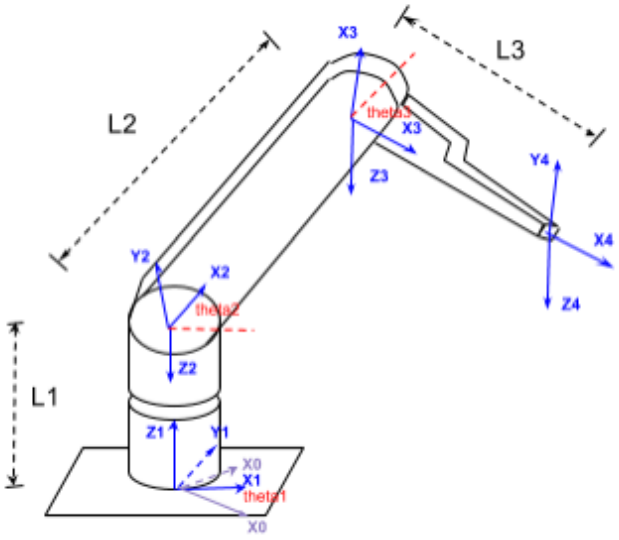

Fig. 7: The kinematic parameters and frame assignments of AMVIP

$i-1$	i	$i+1$	α_{i-1}	a_{i-1}	d_i	θ_i
0	1	2	0	0	0	θ_1
1	2	3	90	0	L_1	θ_2
2	3	4	0	L_2	0	θ_3
3	4	*	0	L_3	0	0

TABLE 1: Link parameters of AMVIP

The link parameters corresponding to this placement of link frames are shown in table 1. From the derivation of link transformations in [6], the general form of ${}^{i-1}_{i}T$ as:

$$ {}^{i-1}_{i}T = \begin{bmatrix} \mathrm{c}\theta_i & -\mathrm{s}\theta_i & 0 & a_{i-1} \\ \mathrm{s}\theta_i \mathrm{c}\alpha_{i-1} & \mathrm{c}\theta_i \mathrm{c}\alpha_{i-1} & -\mathrm{s}\alpha_{i-1} & -\mathrm{s}\alpha_{i-1} d_i \\ \mathrm{s}\theta_i \mathrm{c}\alpha_{i-1} & \mathrm{c}\theta_i \mathrm{s}\alpha_{i-1} & \mathrm{c}\alpha_{i-1} & \mathrm{c}\alpha_{i-1} d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (1) $$

In this report, for spacing purpose, $\mathrm{c}\theta_i$ and $\cos\theta_i$ are used interchangeably; and c_{12} has exactly the same meaning as $\cos(\theta_1 + \theta_2)$. Using (1), the author computes each of the link transformations as follow:

$$ {}^{0}_{1}T = \begin{bmatrix} \mathrm{c}\theta_1 & \text{-}\mathrm{s}\theta_1 & 0 & 0 \\ \mathrm{s}\theta_1 & \mathrm{c}\theta_1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (2) $$

$$ {}^{1}_{2}T = \begin{bmatrix} \mathrm{c}\theta_2 & \text{-}\mathrm{s}\theta_2 & 0 & 0 \\ 0 & 0 & \text{-}1 & \text{-}L_1 \\ \mathrm{s}\theta_2 & \mathrm{c}\theta_2 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (3) $$

$$ {}^{2}_{3}T = \begin{bmatrix} \mathrm{c}\theta_3 & \text{-}\mathrm{s}\theta_3 & 0 & L_2 \\ \mathrm{s}\theta_3 & \mathrm{c}\theta_3 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (4) $$

$$ {}^{3}_{4}T = \begin{bmatrix} 1 & 0 & 0 & L_3 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (5) $$

The transformation from frame {4} to frame {0} is obtained by multiplying all link transformations:

$$ {}^0_4T = {}^0_1T\,{}^1_2T\,{}^2_3T\,{}^3_4T \tag{6} $$

Thus the product of all 3 link transforms:

$$ {}^0_4T = \begin{bmatrix} c_1c_{23} & \text{-}c_1s23 & s_1 & c_1(L_3c_{23}+L_2c_2)+L_1s_1 \\ s_1c_{23} & \text{-}s_1s23 & \text{-}c_1 & s_1(L_3c_{23}+L_2c_2)-L_1c_1 \\ s_{23} & c23 & 0 & L_3s_{23}+L_2s_2 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{7} $$

in which:

$$ P_x = c_1(L_3c_{23}+L_2c_2)+L_1s_1 \tag{8} $$

$$ P_y = s_1(L_3c_{23}+L_2c_2)-L_1c_1 \tag{9} $$

$$ P_z = L_3s_{23}+L_2s_2 \tag{10} $$

Given a particular 3D location $P(P_x, P_y, P_z)$, which the end-effector needs to reach for dotting Braille characters, the AMVIP control algorithm calculates joint angles θ_1, θ_2, and θ_3, and feeds these information to the Arduino board for rotating servos. In other words, given P_x, P_y, P_z, AMVIP needs to solve the set of equations (8), (9), (10) for angle θ_i. In theory, these equations can be solved algebraically. Fortunately, the author finds that it is more interesting and easier to obtain a solution by geometry. Fig 8 shows the schematic of AMVIP, both in top view and front view.

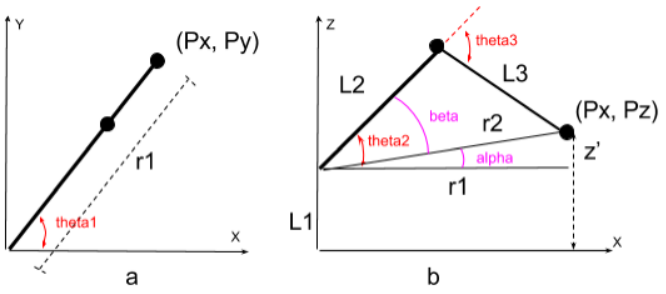

Fig. 8: Plane geometry associated with a three link arm. (a): top view; (b) front view

From Fig 8a:

$$ \theta_1 = Atan2(\frac{P_y}{P_z}) \tag{11} $$

$$ r_1 = \sqrt{P_x^2+P_y^2} \tag{12} $$

From Fig 8b:

$$ z' = P_z - L_1 \tag{13} $$

$$ r_2 = \sqrt{r_1^2+z'^2} = \sqrt{P_x^2+P_y^2+(P_z-L_1)^2} \tag{14} $$

From the law of cosines:

$$ \begin{aligned} r_2^2 &= L_2^2+L_3^2-2L_2L_3cos(180-\theta_3) \\ &= L_2^2+L_3^2-2L_2L_3(cos180cos\theta_3+sin180sin\theta_3) \\ &= L_2^2+L_3^2+2L_2L_3cos\theta_3 \end{aligned} \tag{15} $$

Thus:

$$ \begin{aligned} cos\theta_3 &= \frac{r_2^2-L_2^2-L_3^2}{2L_2L_3} \\ sin\theta_3 &= \sqrt{1-(\frac{r_2^2-L_2^2-L_3^2}{2L_2L_3})^2} \end{aligned} \tag{16} $$

$$ \Rightarrow \theta_3 = Atan2(\frac{\sqrt{1-(\frac{r_2^2-L_2^2-L_3^2}{2L_2L_3})^2}}{\frac{r_2^2-L_2^2-L_3^2}{2L_2L_3}}) \tag{17} $$

Also, from Fig 8b:

$$ \begin{aligned} \alpha &= Atan2(\frac{z'}{r_1}) \\ &= Atan2(\frac{P_z-L_1}{\sqrt{P_x^2+P_y^2}}) \end{aligned} \tag{18} $$

From the law of cosines:

$$ L_3^2 = r_2^2+L_2^2-2r_2L_2cos\beta \tag{19} $$

Thus:

$$ \begin{aligned} cos\beta &= \frac{r_2^2+L_2^2-L_3^2}{2L_2r_2} \\ sin\beta &= \sqrt{1-(\frac{r_2^2+L_2^2-L_3^2}{2L_2r_2})^2} \end{aligned} \tag{20} $$

$$ \Rightarrow \beta = Atan2(\frac{\sqrt{1-(\frac{r_2^2+L_2^2-L_3^2}{2L_2r_2})^2}}{\frac{r_2^2+L_2^2-L_3^2}{2L_2r_2}}) \tag{21} $$

Hence:

$$ \begin{aligned} \theta_2 &= \alpha+\beta \\ &= Atan2(\frac{P_z-L_1}{\sqrt{P_x^2+P_y^2}}) \\ &= +Atan2(\frac{\sqrt{1-(\frac{r_2^2+L_2^2-L_3^2}{2L_2r_2})^2}}{\frac{r_2^2+L_2^2-L_3^2}{2L_2r_2}}) \end{aligned} \tag{22} $$

After mathematical simplification, given a point $P(P_x, P_y, P_z)$ in space, the rotation angles of revolute

joints are calculated as follows:

$$r_1 = \sqrt{P_x^2 + P_y^2} \quad (23)$$

$$r_2 = \sqrt{P_x^2 + P_y^2 + (P_z - L_1)^2} \quad (24)$$

$$\theta_1 = Atan2(\frac{P_y}{P_x}) \quad (25)$$

$$\theta_2 = Atan2(\frac{P_z - L_1}{r_1}) + Atan2(\frac{\sqrt{4r_2^2L_2^2 - (r_2^2 + L_2^2 - L_3^2)^2}}{r_2^2 + L_2^2 - L_3^2}) \quad (26)$$

$$\theta_3 = Atan2(\frac{\sqrt{4L_3^2L_2^2 - (r_2^2 - L_2^2 - L_3^2)^2}}{r_2^2 - L_2^2 - L_3^2}) \quad (27)$$

$$(28)$$

To communicate joint angles with the AMVIP arm, a data stream is prepared in form of **B**θ_1-**S**θ_2-**W**θ_3, in which B is the servo at base, S is the servo at elbow, and W is the servo at wrist. Also, to avoid overloading Arduino UNO board, this data stream is sent every 0.25 second since it takes 1 second of delay for servo's movement.

4.2 Gesture Recognition

Gesture recognition mechanism is a two-step process, including Background Subtraction and Hand Detection, as listed in Fig 9.

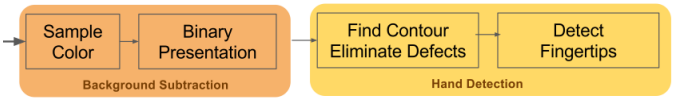

Fig. 9: Gesture recognition pipeline

4.2.1 Background Subtraction

Separating a hand from its background is based on color recognition. User is required to hold his or her hand over a scanned region so that the system can get a set of sample colors. This scan region is made up from 10 rectangular ROIs, size of 20x20 pixels, placed on the middle of a screen. For each ROI, the median value, m_k, of each color channel of RGB is calculated to get an approximated color value for a human skin. With a predefined constant, this median value is used as threshold to construct a RGB color range, in which, a pixel on an original image is on if in range, and is off if otherwise. In other words, given 10 median values from ROIs, a set of 10 binary images of an original image is generated, such that every pixels in each binary images fall into a certain color range. The color threshold, or a color range, for a binary image is listed in Table 2. In fact, in each binary image, the background color is off or black, and a pixel color of a hand region is on or white as it is similar to a median value.

Threshold	R	G	B
Min	m_R - 12	m_G - 30	m_B - 80
Max	m_R + 7	m_G + 40	m_B + 80

TABLE 2: Threshold values for producing binary images from an original image

Since binary images are generated from sample colors of ROIs, noise is inevitable. For each generated binary image, a median filter with kernel size of 9 is applied to reduce noise. After several experiments, it is concluded that the kernel size of 9, which is a 9x9, yields the best result in an optimal time. Lower kernel sizes do not eliminate noise on edges of detected contours, and the higher ones increase a smoothen executing time significantly. Last, these binary presentations are integrated all together; and a bilateral filter, a non-linear and edge-preserving image smoothing technique, is applied onto the resulting image to produce a final noise-free binary presentation of the scene. The entire process is shown in Fig 10.

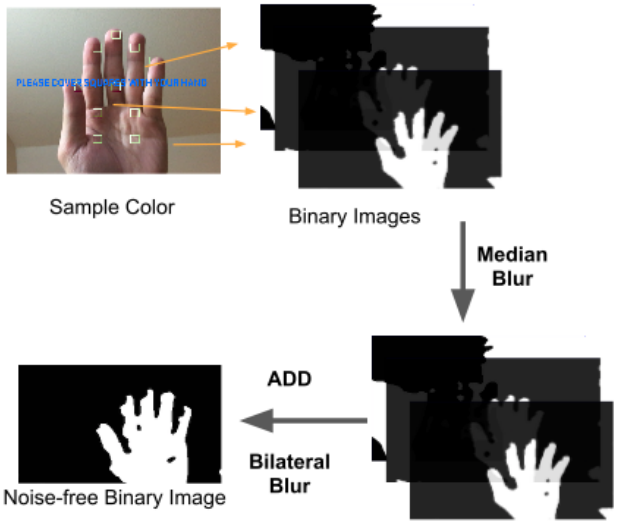

Fig. 10: The background subtraction process

4.2.2 Hand Detection

The proposed hand detection technique includes eliminating convexity defects on detected contours, and indexing fingertips. After background subtraction, the binary image now consists of a large white hand region on a black background; thus, a hand shape contour is guaranteed to be available. First, the OpenCV method `findContours` is invoked to retrieves available contours on scene. It is possible that other objects, having a similar color as human skin, is also visible on scene (eg: forehead, ear, cheeks, etc); so, this method will return a set of contours, instead of just a singular. Hence, the largest contour among all is the hand contour, assuming a hand covering a large area on screen.

Gesture detection requires positions and a number of shown fingertips. To obtain this information, the proposed approach employs a concept of **convex hull** and **convexity defects**. Convex hull is a polygon surrounding a particular contour; in this case, it will be an outline of a detected hand contour. Convexity defects is a data structure capturing differences between a contour and its convex hull, where it is defined as:

```
struct CvConvexityDefect{
    //  the point where the defect begins
        CvPoint* start;
    // the point where the defect ends
        CvPoint* end;
    // the farthest point from the convex hull
        CvPoint* defect;
    // distance between the farthest point
    // and the convex hull
        float depth;
};
```

As shown in Fig 11, the blue polygon is the convex hull of detected hand contour, red circles are start points and end points of convexity defects, and green circles are defect points. Logically, a hand with 5 fingers should only have 4 defects points. However, the openCV method will return more structures than that because of differences between convex hull and hand contour at a junction between hand and wrist (the most bottom right green point on Fig 11). Such "false" defect points should be eliminated. In this approach, the threshold for such elimination is based on a distance D between a defect point and convex hull, or a `depth` value in a convexity structure, and an angle α between lines connecting start point, defect point, and end point. Through experimentation, it is found that D less than $\frac{1}{5}$ of a screen height and α less than 90 degrees will make a reasonable combination for such threshold.

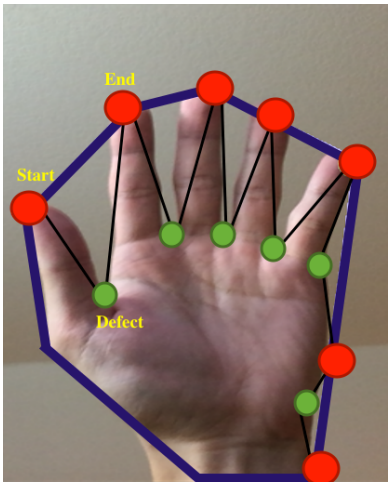

Fig. 11: Convex hull and detected convexity defects

Indexing fingertips, is easier once all "false" convexity defects are eliminated. Given a hand of five fingers, thus, the system is able to reserve 4 convexity structures, in which each start point locates at a fingertip. To indexing such finger, relatively position of those start points with a screen height is calculated. If there is only one finger shown, only one convexity defect is reserved; thus, the system will need to analyze a relationship among this defect's location and the maximum height of the contour to decide whether to discard or reserve this fingertip. The result for gesture recognition in the AMVIP control algorithm is shown in Fig 12.

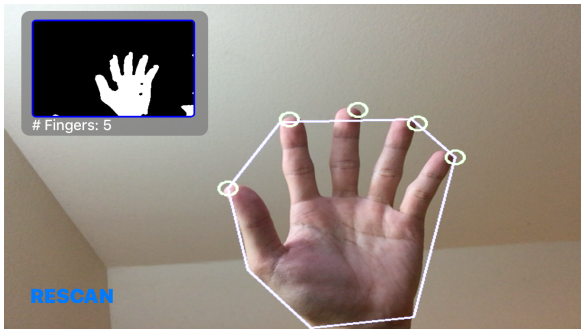

Fig. 12: The result of gesture recognition mechanism

5 Implementation

AMVIP control algorithm contains independent implementation on iPhone and Arduino Uno board. The components of Gesture Recognition and Coordinate Localization are written in Objective-C, Objective-C++ and C++, with supports from external libraries including OpenCV and iOS SDK. For data communication, iOS bluetooth framework is employed to initiate BLE connection and to transmit joint angles from an iPhone. From the side of AMVIP arm, C is the solely programming language in usage to control Arduino board for receiving information and directing servos' rotation. The integrated development environment is Xcode. The implement source code is categorized into three modules, namely `Manipulator` for Coordinate Localization, `HandDetection` for Gesture Recognition, and `BTCommunication` for Data Communication. As in the very first stage, the AMVIP system, so far, is able to recognize numerical values in sign-language.

In `Manipulator`, the class `VPoint` is responsible for 3D coordinate, and the class `Manipulator` serves the purpose of solving inverse kinematics equations and calculating new tip's location. In `Manipulator`, the method `setTipLocation` takes a detected number as an input parameter, looks up a predefine database for corresponding Braille characters, and calculates the next tip's location based on the current one. Next, the new calculated 3D pose orientation is fed into `IKJointAngles` to solve for 3 revolute joint angles as specified in the previous section. Its implementation is as follow:

```
float r1 = sqrtf(pow(self.tip.x, 2.0)
                 + pow(self.tip.y, 2.0));
float r2 = sqrtf(pow(r1, 2.0)
                 + pow(self.tip.z - self.L1, 2.0));
self.theta1 = atan2f(self.tip.y, self.tip.x);
self.theta2 = atan2f(self.tip.z-self.L1, r1)
                  + atan2f(sqrtf(4*pow(self.L2, 2.0)
                                 *pow(r2, 2.0)
                                 - pow(pow(self.L2, 2.0)
                                       +pow(r2, 2.0)
                                 - pow(self.L3, 2.0), 2.0)),
                                 (pow(self.L2, 2.0)
                                 +pow(r2, 2.0)
                                 -pow(self.L3, 2.0)));
self.theta3 = atan2f(sqrtf(4*pow(self.L2, 2.0)
                                 *pow(self.L3, 2.0)
                                 - pow(pow(r2, 2.0)
                                 -pow(self.L2, 2.0)
```

```
                        -pow(self.L3, 2.0), 2.0)),
                        (pow(r2, 2.0)
                        -pow(self.L2, 2.0)
                        -pow(self.L3, 2.0)));
return @[self.theta1,self.theta2,self.theta3];
```

For Gesture Recognition, there are four main classes: `ViewController` acts as a main thread of the application which is in charge of loading camera, feeding an image stream, and displaying recognition results; `ROI` simply displays and collects sample color values of ROIs; `HandDetector` is the implementation of Background Subtraction; and `Hand` is responsible for Hand Detection process. Once the application is launched, `ViewController` checks the device camera's availability via a method `(BOOL)deviceHasCamera`, sets up camera delegate with `CvVideoCamera` - a developed framework from OpenCV - in `(void)setupCamera`, creates `HandDetector` objects, starts feeding image streams to `(void)processImage:(cv::Mat &)image`, and sets timers, once every a quarter of a second, for tracking and updating detected gestures with `(void)tracking:(NSTimer*)timer`.

Below is a pseudocode of a background subtraction process:

```
1. If still looking at a hand placed on camera
  1.1. calling showROIsOn() to display ROIs
    live on a camera frame
  1.2. Stop all incoming image frames
  1.3. Calculate median value of sample
    colors
2. else
  2.1. calling OpenCV method pyrDown()
    to reduce a process image size
  2.2. produceBinaries
  2.3. detectContour
```

The `produceBinaries()` method first converts color channel from RGB to HLS for color averaging, determines color threshold from `normalizeColor()`, generates a set of binary images from ROIs by invoking OpenCV method `inRange()`, applies image smoothing technique with OpenCV function `medianBlur()`, integrates all binary images, and applies the bilateral filter with an OpenCV implementation `bilateralFilter()` for a final noise-free binary presentation. The `detectContour()` method is responsible for finding the hand contour and eliminating convexity defects. Its implementation invokes OpenCV methods `pyrUp()`, `findContours()`, `convexHull()`, `convexityDefects()`. In the class `Hand` that is responsible for a Hand Detection process, fingers indexing is based on methods `distanceBetweenTwoPoints (start, defect, end)`, and `angleBetweenLinesConnecting (start, defect, end)` to determine defect elimination threshold. Last, `eliminateDefects()` is invoked to get rid off "false" convexity defects.

To communicate joint angles with Arduino, AMVIP control algorithm initiates a bluetooth connection via external peripheral Bluetooth framework, provided by iOS SDK. The method `initWithPeripheral` in the module `BTCommunication` searches for available bluetooth-enable devices and sets up two-way connection. To distinguish AMVIP with other available peripherals, this method looks for a device with `SERVICE_UUID` and `TX_CHARACTERISTICS_UUID` having value of `B4BDB998-8F4A-45F6-A407-6B48D79CFC2F`. These are the unique UUID of Arduino BLE Shield on AMVIP. Fig 13 shows a hardware setup of Arduino Uno board and Arduino BLE Shield.

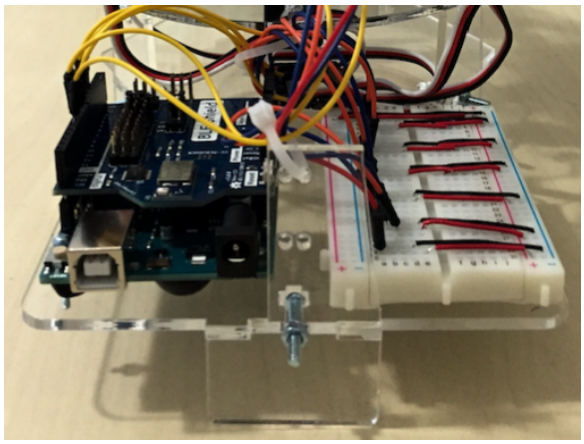

Fig. 13: Arduino BLE Shield sits on top of the Arduino Uno board for bluetooth communication

Once the connection is successfully setup, data is sent via BLE 4.0 channel between iPhone and Arduino BLE Shield. On the physical arm, data is received on `RX_PIN_4` of the Arduino BLE Shield, and transmitted to the Arduino Uno board on `TX_PIN_5`. Servos are wired into `OUTPUT_PIN_{7-12}`. The transmission baud rate is 9600 bits per second. To dot on paper, the control algorithm invokes `servo.write(alpha)`, a method provided by the Servo library of Arduino SDK, to rotate corresponding servos by `alpha` degrees. Fig 14 shows the wiring of servos, and Fig 15 shows the end result of the entire AVMIP: Braille numbers are dotted on a planar surface.

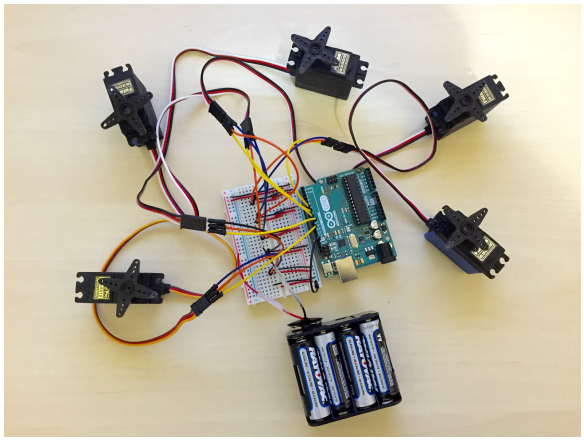

Fig. 14: Servos wiring schema on Arduino Uno board

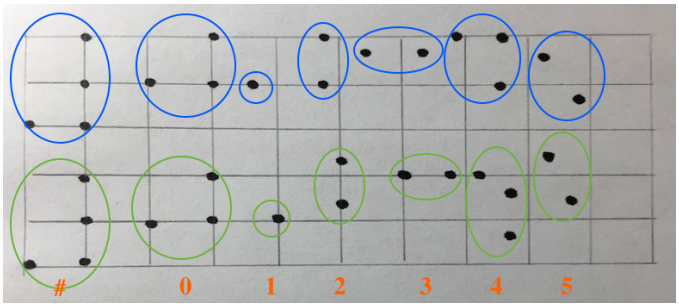

Fig. 15: Braille numbers are dotted on a paper. The first and second runs are circled in blue and green, respectively. The orange character explains its corresponding Braille.

6 Analysis and Discussion

To measure the system's reliability and correctness, the author performs a set of gestures, showing a sequence of numbers from 0 to 5, for AMVIP to translate into Braille. The result is shown in Fig 16. Table 3 and Fig 17 shows a distance error between the dotted locations and the correct ones. The error is calculated as $e = \sum_{i=1}^{N}(\sqrt{(x_i - x_i')^2 + (y_i - y_i')^2})$ where i is the number of dots for each Braille characters, (x, y) is the correct location, and (x', y') is the dotted position.

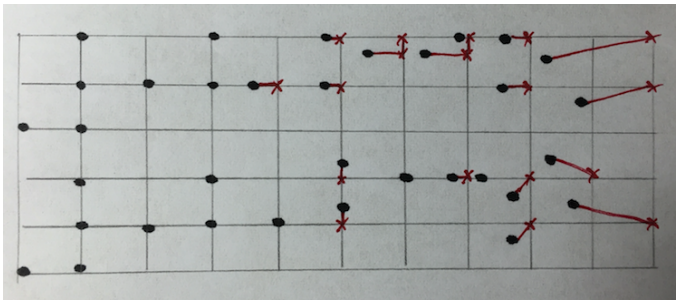

Fig. 16: The differences between the correct locations (red crosses) and the AMVIP's dotted positions (black points).

The data suggests that the more operations AMVIP performs, the more inaccurate it is. At first, the calculated positions are very precisely. After several iterations, the end-effector starts off-track, and the distance error keeps increasing. The author argues that these errors do not come from inverse kinematics calculation or data lost during transmission; instead, it comes from hardware limitation. Since the base of AMVIP sits on top of a table freely, it slides off the original location after several servo's movements due to hardware vibration. Also, since the built-in servos are standard and non-continuos, its maximum rotation angle is 180-degree. The Arduino `servo.write(alpha)` method will round up the calculated angles before actual movement. In other words, if the inverse kinematics solution is $(\theta_1, \theta_2, \theta_3)$ = (40.52, 34.53, 67.89), then physical servos will rotate angles of 41, 35, and 68 degrees, respectively. Another room for errors is the delay between transmission time, servo's rotation time, and control algorithm processing time. AMVIP transmits data every 0.25 seconds, and it takes up to 1 second for servo's movement. However, gesture recognition and joint angle calculation, took place in an iPhone processor, executes in milliseconds. This means that while a servo is rotating to a specific location, it is instructed to rotate to different angles immediately. This results a signal interruption and causes an incomplete rotation.

gesture	0	1	2	3	4	5
Trial 1 error (mm)	0	5	6	7	10	35
Trial 2 error (mm)	0	0	4	3	10	30

TABLE 3: Distance error between the dotted and the correct locations

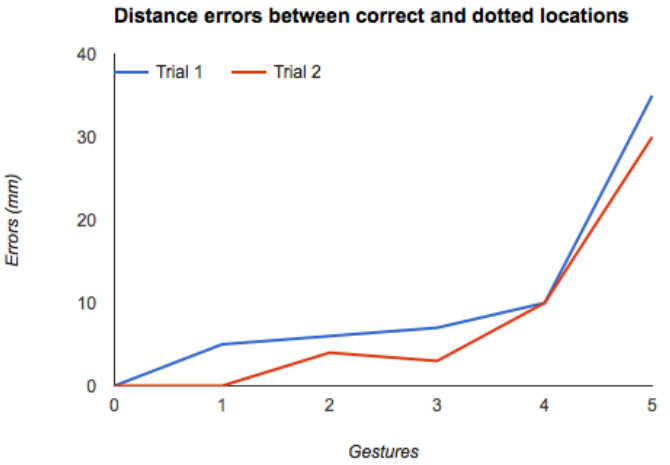

Fig. 17: The plot of errors between the dotted and the correct locations

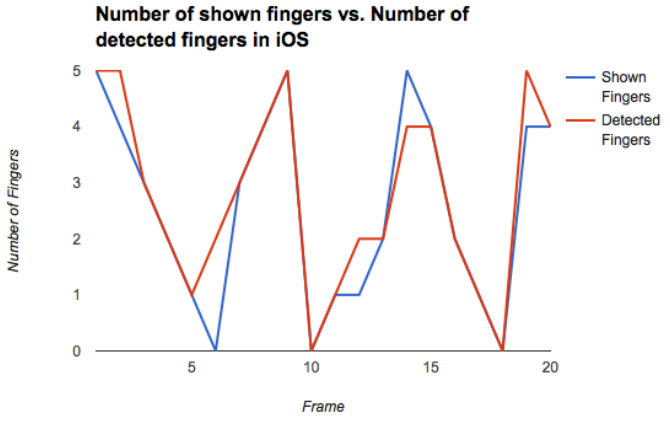

Fig. 18: Number of shown fingers vs Number of detected fingers

Similarly, the author also performs a set of pre-defined number of fingers to measure correctness in gesture recognition. This set contains 20 gestures, and each one is the number of shown fingers on a single image frame. The experiment is performed in 20 seconds, in which each frame is captured every second. For example, at the k^{th} second, the author shows x fingers, and the captured frame k^{th} detects either x or x' fingers. Fig 18 shows a graph of number of shown fingers versus number of detected fingers. As a result, it is reasonable to conclude

that although there is discrepancy between number of shown fingers and number of detected fingers, this difference is acceptable, and the output is reliable. Errors in gesture recognition can possibly depend on deployed device and camera. One possible explanation is that the image stream in iOS comes from CvVideoCamera, a third-party implementation on OpenCV, which stretches input images. The author believes that if AMVIP employs original iOS camera SDK, named UIImagePickerController, instead of CvVideoCamera, the system is able to perform better.

7 Conclusion

The proposed system AMVIP introduces a simple artificial eye-hand coordination system that translates sign-languages into Braille characters to help people with visual impairments collecting and retaining information. This report mainly focuses on spatial transformations, manipulator forward and inverse kinematics, 3D pose localization, and several aspects of computer vision, including background subtraction and gesture recognition. Throughout implementation and experiment, it is reasonable to conclude that the system functions properly as expected, and the outcome is reliable up to a certain degree. Although there are still errors in both coordinate localization and gesture detection, these are tolerable due to hardware limitation, differences in processor's speed, background complexity, and sampling position. Of course, this is the very first stage, and there are plenty of rooms for future improvement. Given the authors' novice knowledge in both image analysis and robotics, challenges arising could be coordinate-offset due to camera calibration, complexity in inverse kinematics equations involving more joints, and distinguishability physical locations on a pair of sign-language characters having a similar representation.

References

[1] Szabo, R., & Gontean, A. S. (2013). "Full 3D Robotic Arm Control with Stereo Cameras Made in LabVIEW". *FedCSIS Position Papers, 37-42*.

[2] Lin, L., Song, Y., Yang, Y., Feng, H., Cheng, Y., & Pan, H. (2015). "Computer vision system R&D for EAST Articulated Maintenance Arm robot". *Fusion Engineering and Design*, 100, 254-259.

[3] Jiang, H., Wachs, J. P., & Duerstock, B. S. (2013, June). "Integrated vision-based robotic arm interface for operators with upper limb mobility impairments". In *Rehabilitation Robotics (ICORR), 2013 IEEE International Conference* on (pp. 1-6). IEEE.

[4] Cabre, T. P., Cairol, M. T., Calafell, D. F., Ribes, M. T., & Roca, J. P. (2013). "Project-based learning example: controlling an educational robotic arm with computer vision". *Tecnologias del Aprendizaje, IEEE Revista Iberoamericana de*, 8(3), 135-142.

[5] Chaudhury, A., Ward, C., Talasaz, A., Ivanov, A. G., Huner, N., Grodzinski, B., ... & Barron, J. L. (2015, June). "Computer Vision Based Autonomous Robotic System for 3D Plant Growth Measurement". In *Computer and Robot Vision (CRV), 2015 12th Conference* on (pp. 290-296). IEEE.

[6] Craig, J. J. (1989). Chapter 3: Inverse manipulator kinematics.*Introduction to robotics: Mechanics and control* (3rd ed.). Reading, MA: Addison-Wesley

[7] Andresen, S. (Aug 2013). "Hand Tracking and Recognition With OpenCV". Retrieved on Feb 8, 2016 from *http://simena86.github.io/blog/2013/08/12/hand-tracking-and-recognition-with-opencv/*