Multi-process, Multi-threaded System for 6 DOF Robotics RGBD Vision Arm

Cong Fu
Mechanical Engineering
University of Michigan
Ann Arbor, Michigan 48105
Email: congfu@umich.edu

Zihang Wei
Electrical Computer Engineering
University of Michigan
Ann Arbor, Michigan 48109
Email: wzih@umich.edu

Pallav Kothari
Robotics
University of Michigan
Ann Arbor, Michigan 48105
Email: pallavk@umich.edu

***Abstract*—An Autonomous robot arm is synonymous with incredible precision, productivity and flexibility and therefore the growth of robotic arm has exponentially increased in industrial culture. The Goal of this project is to design a 6 DOF arm and to develop algorithms for effective task completion under time constraints as well as navigation requirement to determine best possible path. This is achieved by initially constructing a 3D location via Kinect RGBD camera that records synchronized and aligned RGB, depth images. Blob detection computer vision technique is used to locate different blocks. We design three different inverse kinematics modes to make the most of the gripper in order to augment the robot's workspace to meet different task requirement. Overlooking all the tasks is an effective multi thread and multi process system that handles failure and provide feedback for every step. In this project we use reliable TCP for all the communication needs and faster UDP for only tracking heartbeat messages. The experimental results of this project strongly encourages the implementation of Mutli-process, Multi threaded system for Automation of robotic arm.**

Keywords - 6 DOF arm; Inverse Kinematics; Blob detection; Cubic spline; Path planning; State machine; Multi thread and Multi process system; TCP/UDP communication

I. Introduction

In this report, we have discussed the effectiveness of using a Multi-process, Multi-threaded system for a 6 DOF arm which is utilizing RGBD kinect vision system. The whole project report is divided into different sections which discusses independent objectives and experimental results.

Section II: We discuss the system design for using multi-process and multi-thread mechanism. This section also deals with advantages of using potent TCP/UDP communication protocol.

Section III: We discuss the gripper design, and explain why our final design fit the tasks here.

Section IV: We talk about the forward kinematics algorithm to compute the position of the end-effector from specified values for the joint angle parameters and inverse kinematics algorithms which determine the values of the joint angle variables given the end effector's position and orientation. Due to our gripper's flexibility design with 2 additional DOF, we are able to implement 3 different inverse kinematics modes for greater efficiency.

Section V: This part focuses on computer vision technique for block recognition using blob detection technique and 3D coordinate construction. This section also describes our intelligent control and autonomous execution. At the end of this section, competition task results are published. Appendix A contains our figures, and appendix B contains our tables.

Section VI: We give an introduction about our intelligent control and autonomous execution. This includes our cubic spline method to make teach and repeat more smoothly, and different strategies to fulfill the given tasks.

II. Software design

In this project, we use multi-process, multi-threaded system to handle different tasks simultaneously with much less time than normal single thread in main function. We create the software architecture (Fig. 1) to run and communicate the command more efficiently, and build the control decision making architecture (Fig. 3) to make the task running in a more logical and systematical way.

A. Software Architecture

The software system could manipulate several robotics corporately in one or several remote computers during the same time and have the safe mechanism, both in communication channel check and subsystem condition check, to guarantee every task is executed successfully. Unless all the processes (on all the computers) or all the actuator (in this situation, the robotic arm) die, the task will be finished. This parallel computation framework is designed to handle much more heavy tasks in the same time, even we only use a small part of its potential in this project.

`The Master process` manages the jobs, distributes work and assigns them to worker process, and handles heart-beat check.

`The Worker process` wait for commands from the master process, and then perform tasks based on given input arguments.

In this project, we create the heartbeat thread in worker process, which will give heartbeat to master process every 2 seconds. If one worker process is killed by some reason, and the master does not detect heartbeat from that worker process for more than 10s, then it will initialize a new worker process and restart the unfinished job the dead process was doing.

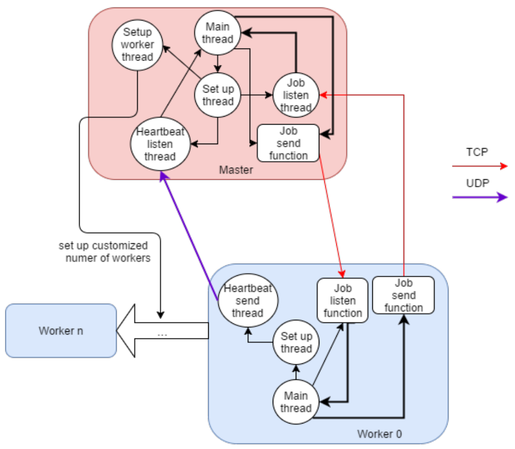


Fig. 1. Software Architecture.

The master process first creates a thread to initialize worker communication channel parameters and start a thread to build all the worker processes. Then the master will start job job listen thread to listen signals from worker process, and heartbeat listen thread to listen heartbeat from worker process. When master process creates new worker processes, these new processes start their own heartbeat thread, and keep listening on master TCP channel to receive command. Once worker process receives task command, they will execute these tasks and send task status feedback signal to master TCP listener. All the functions and process, thread correlations can be seen in Fig. 2 , and the key skeleton of the architecture code can be found in Appendix A or submission code.

B. Multi Threads and Multi Process

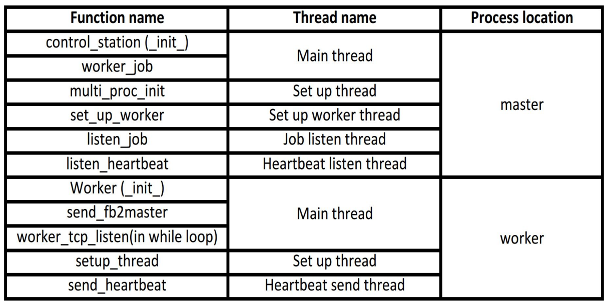
This system is able to run any job on our machine, using a actuator function (in this project, it is just motor control command) implementation we wrote.

A process is an executing program with a dedicated memory space. Many processes run at the same time on a computer. Processes have isolated memory spaces so one process cannot access the data of another process. Threads are similar to processes in that they allow for parallelization of work, but unlike processes, threads share the same memory space and can access each other's data. Threads are owned and created by a single process, and are only alive as long as the parent process is alive.

In this project, we create master process in `control_station.py` and start new processes in it. And we use TCP and UDP to communicate in different processes.

The system is designed to setup several processes in the same lab computer to do inverse kinematics calculation, and 7 motors (in arm) motion control separately, and continuously communicate with master server. The worker processes get the task, do the task, and give feedback to the master process, and the master process will just do the information arrangement and assign every work to the worker process.

We were also considering all the image processing job in the worker process to reduce the workload of master process, but the GUI system in main process still need to handle the kinect image, so there would be a conflict if we try to call the kinect image obtaining function in different process at the same time. Since the image processing takes very short time (in our algorithm, less than 0.1s), we decided to keep the image processing part in main process. Since the speed bottleneck is the arm motion part, we put more energy in the dynamic motion planning.



Function name	Thread name	Process location
control_station (_init_)	Main thread	master
worker_job		
multi_proc_init	Set up thread	
set_up_worker	Set up worker thread	
listen_job	Job listen thread	
listen_heartbeat	Heartbeat listen thread	
Worker (_init_)	Main thread	worker
send_fb2master		
worker_tcp_listen(in while loop)		
setup_thread	Set up thread	
send_heartbeat	Heartbeat send thread	

Fig. 2. Multi-thread and Function Correlation Chart.

C. communication protocol

The communication in this system uses TCP (Transmission Control Protocol) and UDP (User Datagram Protocol). A socket creates and manages a TCP/UDP connection, and all sockets use a specific port. Sockets can be used to send data to a specific port, and to listen for data on a specific port. In this project, we use TCP for all communication on the main thread (like processes status listening, job assigning, and feedback sending), and UDP for all other communication (here, specifically heartbeat messages).

`TCP` guarantees the recipient will receive the packets in order by numbering them. TCP is reliable. All packets sent with TCP are tracked so no data is lost or corrupted in transit. Packets using `UDP` are just sent to the recipient without error checking. The sender does not wait to make sure the recipient receive the packet. UDP is not very reliable when the communication noise is obvious, but it is faster than TCP. So in our project, we use TCP to mark the status of each process and do the job assignment and feedback return, and only use UDP to track heartbeat of each process.

For the communication data type, We use JSON as information package language for TCP and UDP. JSON (JavaScript Object Notation) is a lightweight data-interchange format. Since the JSON format is text only, it can easily be sent to and from a server, and used as a data format by any programming language.

All the massages in master and worker processes are packaged as a dictionary data structure we defined, and after jsonized, they will be sent and received in the TCP/UDP channel port.

D. State Machine and Decision Making Architecture

State machine is implemented in master process, and it is for block picking and arranging. But the real execution step is in worker process level, with angle check and speed control. As seen in Fig. 3, the master process level (in red box) decide the structure of one complete step move:

1) From initial location (zero point) go to the block location
2) Grab target block
3) Get to waypoint
4) Get to transition waypoint
5) Drop target block
6) Modify gesture of gripper and orientation of block (this includes series actions)
7) Grab block again
8) Get to another waypoint
9) Arrive final location (this includes series actions)
10) Drop block
11) Get to initial location

In step 6) and 9), we define several sub-steps to make sure the motion modification and block arrangement is smooth and accurate. For example, in the pyramid stacking task, when gripper puts the block in the final location, it first arrives above of the location, modifies its orientation, arrives the final location, drops block, moves above certain distance of the location again, and then goes to the next waypoint.

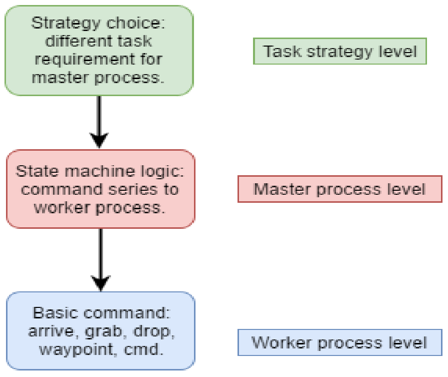

Strategy choice: different task requirement for master process.
Task strategy level
State machine logic: command series to worker process.
Master process level
Basic command: arrive, grab, drop, waypoint, cmd.
Worker process level

Fig. 3. Control Decision Making Architecture.

Each step above is defined in master process (in `control_station.py`), and the command message is sent as a package through TCP to and executed by worker process. Once the action is finished, the worker process will return a certain message to notify the master process this step has been achieved and the next command can be send and executed.

As seen in Fig. 3, for every worker process (in blue box), we define several basic commands:
`arrive` : get to certain target point (x,y,z) by certain inverse kinematics method. Check if the action is achieved, and return a message to master.
`grab` : grab target block. Check if the action is achieved by gripper motor servo torque and angle, and return a message to master.
`drop` : drop target block. Check if the action is achieved by gripper motor servo angle, and return a message to master.
`waypoint` : will get to waypoint. Similar to 'arrive', but we don't need target point coordinate. The waypoint is automatically calculated by current end effector location and angle. Check if the action is achieved, and return a message to master.
`cmd` : will give direct commend to motor. Check if the action is achieved, and return a message to master.

In this three level decision making architecture (as shown in Fig. 3), we package basic command and state machine combo in different level, and for the final task running level, we can easily just call the action in package to execute the strategy level move.

III. Gripper design

Gripper is a very important part of design to do tasks efficiently. We designed two kinds of grippers, a four bar gripper and a direct motor drive gripper. The four bar one (Fig 4) can always be parallel to the blocks easily for us to calibrate the grasp and drop angle. And the claw can move forward when gripping, which can compensate the tolerance of localization. But during the experimentation, we find out that XL-320 motor has very limited torque and OLLO rivet and hole mate are generating relatively large friction force. So when trying to grasp the block, it may overload sometime. Also when using the swap mode to grasp the block, the four bar gripper will take up extra space, which may collide with other blocks. So we designed a second gripper.

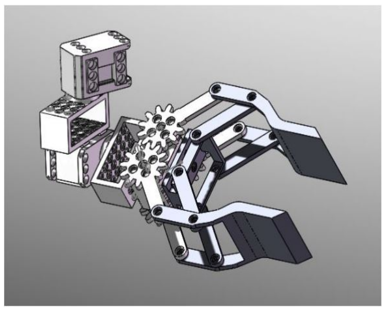


Fig. 4. Gripper Design 1.

The second gripper (Fig 5) is driven directly by motor without any middle mechanism part except for gear mate. In the test, this new design fits our grasp mode very well. It can save space up to 40mm along the z axis of the frame fixed on the gripper. And this compact gripper can make the most of the XL-320 torque to grasp more tightly than its predecessor. Additionally, having 2 extra DOF provides more freedom and is perfectly suitable with three different gripping mode (these will be introduced later in inverse kinematic section) which we implement in our target acquisition strategy.

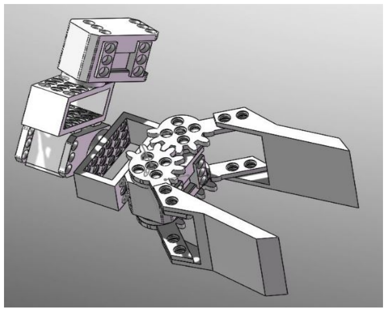


Fig. 5. Gripper Design 2.

IV. Kinematics Design

A. *Forward Kinematics*

Forward kinematics refers to the use of the kinematic equations of a robot to determine the position and orientation of the end effector given the values for the joint angle variables of the arm robot. We establish the relationship between the base fixed frame and the end effector fixed frame. A commonly used convention for selecting frames of reference for robots is the Denavit-Hartenberg (DH) convention, a convention for the definition of the joint matrices and link matrices to standardize the coordinate frame for spatial linkages[1]. When applying this convention to the arm robot, we can easily get the homogeneous transformation matrix connecting two adjacent links, which consist of four basic transformation matrices.

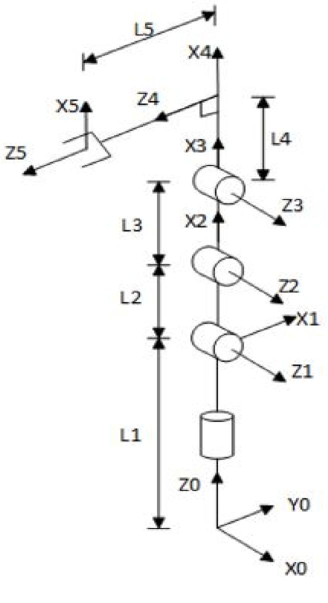


Fig. 6. Forward kinematics schematic.

DH convention defines four parameters associated with link i and joint i, which are θ_i(joint angle), d_i(link offset), a_i(link length), and α_i(link twist). The schematic of armbot(initial configuration) and DH table is showed in Fig. 6 and Table I.

Then we can express the origin of the end effector fixed frame in the base frame by multiplying $A1A2A3A4A5$ and

TABLE I
DH Table

Link	θ_i	d_i	a_i	α_i
1	$\theta_1 + \pi/2$	117.7	0	$\pi/2$
2	$\theta_2 + \pi/2$	0	101	0
3	θ_3	0	101	0
4	θ_4	0	72.5	$-\pi/2$
5	0	86	0	0

homogeneous coordinate of the origin of end effector fixed frame. In addition, the arm robot we use is a planar robot, so the end effector position can be defined by x, y, z and ϕ, where ϕ is the angle between end effector and horizontal plane and defined as $(\theta_2 + \theta_3 + \theta_4 - 90°)$.

B. *Mathematical Model for Inverse Kinematics*

In most cases, we want the arm robot to move to the given position to pick up blocks. Then we need inverse kinematics to determine the values of the joint angle variables given the end effector's position and orientation. In this case, we can use geometric decoupling to consider the position and orientation problems independently, and we can get closed form solution. Due to our gripper's flexibility with two additional wrist DOF, we design three inverse kinematics modes (Fig 7) to make the most of the gripper in order to augment the robot's workspace to meet different task requirements.

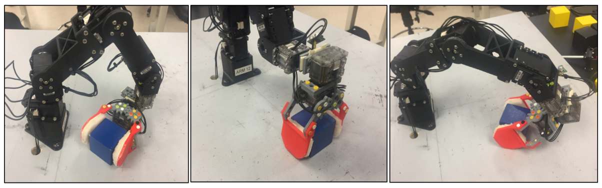


Fig. 7. Grab modes (Left:sweep;Middle:perpendicular;Right:free pick mode)

1) Perpendicular Pick Mode
This is a basic mode. In this mode, the front arm is perpendicular to the ground (as seen in the middle figure in Fig 7). Due to the arm robot is a planar robot, the base angle can be defined once given the coordinates x and y using $\theta_1 = atan2(y, x)$, as shown in Fig. 8. Then we only need to consider the planar configuration of the robot,as shown in Fig. 9. The new coordinate of the point fixed on last motor in new 2D plane can be derived from given position of blocks when we fix ψ equals to $0°$ and lock the swap DOF. The new coordinate is called (n, m), and then θ_4 can be derived as$(90° - \theta_2 - \theta_3)$. And we let the arm always be elbow up position, then $\theta_3 = arccos(\frac{n^2+m^2-l_2^2-l_3^2}{2l_2l_3})$. Then $\beta = atan2(m, n)$. And $\psi = arccos(\frac{n^2+m^2+l_2^2-l_3^2}{2l_2\sqrt{n^2+m^2}})$, then $\theta_2 = \beta - \psi$.

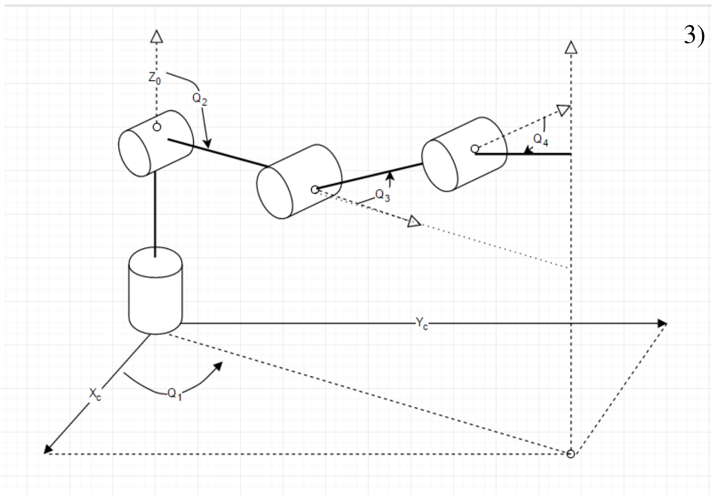


Fig. 8. 3D Schematic for perpendicular mode

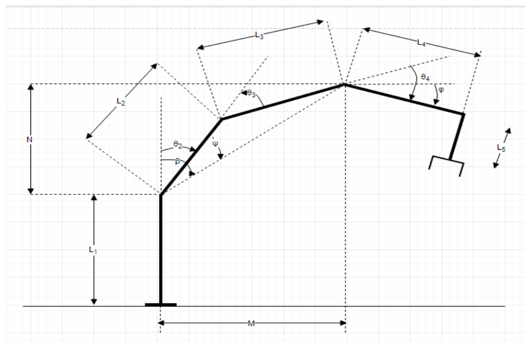


Fig. 9. Planar Schematic for perpendicular mode

2) Swap Mode
In this mode, the gripper is parallel to the ground (as seen in the left figure in Fig 7). When we need to pick up blocks near the base of the robot (or in the board edges), and also want to make the edge of the block align with the edge of the gripper, we could use this swap mode. Also when stacking the last block, we need gripper and block parallel to the penultimate block. So in this case, we also need to lock the rotation DOF and use the swap DOF and fix ϕ equals to $90°$. In this mode, program will return the position of the blocks four different orientations corresponding to four faces of the block. And through that we can we get the coordinate of the point fixed on last motor. Then the inverse kinematics can be solved similar to the perpendicular mode .

3) Free Pick Mode/ Iterative Mode
Sometimes we cannot reach to the block when fixing ϕ equals to $0°$ in swap mode and perpendicular mode. Then we need to loop ϕ iteratively in the program until get a solution satisfies the condition. Given the block position and different ϕ values, we can get different coordinate of the point fixed on last motor in defined 2D plane, as shown in Fig. 10. First we can get point 5 coordinate $(x + l_5 \sin\theta \cos\theta_1, y + l_5 \sin\theta \sin\theta_1, z + l_5 \cos\theta)$ and point 4 coordinate $(x + l_5 \sin\theta \cos\theta_1 - l_4 \cos\theta \cos\theta_1, y + l_5 \sin\theta \sin\theta_1 - l_4 \cos\theta \sin\theta 1, z + l_5 \cos\theta + l_4 \sin\theta)$. Then the joint angles can be decoupled and solved. $\theta_1 = atan2(y,x). l'^2 = x_4^2 + y_4^2 + (z_4 - l_1)^2$. Then θ_3 can be derived through the Law of Cosines, $\theta_3 = \alpha_2 + \alpha_3 = \arccos(\frac{l_2^2 + l'^2 - l_3^2}{2 l_2 l'})$. Similarly, $\theta_2 = \frac{\pi}{2} - \alpha_2 - \arctan(\frac{z_4 - l_1}{\sqrt{x_4^2 + y_4^2}})$. We fix θ_5 and θ_6 equal to $0°$. And through solve pentagon we can get relationship between α_4, α_5 and ϕ, then we can get θ_4 through $\theta_4 = \theta + \alpha_4 - \alpha_3$.

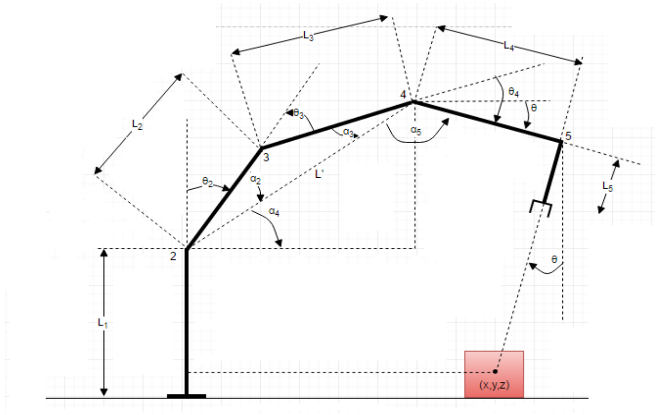


Fig. 10. Planar Schematic for iterative mode

V. Computer Vision

The overhead Kinect RGBD camera is used to determine target locations in the arm workspace.

A. Calibration

We can approximate the mapping transformation matrix from pixel coordinates in the camera image to world coordinates on the board with an affine transformation matrix because the overhead camera is relatively far away from the entire board, and it is roughly the same distance from the camera to the board. In depth camera, we also need intrinsic matrix to calibrate the camera coordinates to world coordinates.

When we start calibration, first click "Perform Affine Calibration" at GUI, and start to choose 4 corners of the board (in here, we use four point pairs for calibration, and the accuracy is good enough in the experiment). Then we estimate an affine mapping from each pixel (p_x, p_y) to its corresponding world coordinate (w_x, w_y) as in Equation below.

$$\begin{bmatrix} a & b & c \\ d & e & f \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ 1 \end{bmatrix} = \begin{bmatrix} w_x \\ w_y \\ 1 \end{bmatrix}$$

We change the form of the equations above by rearranging and stacking corresponding pixels and world coordinates into the form Ax = b as in equation below.

$$\begin{bmatrix} p_x & p_y & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & p_x & p_y & 1 \\ \dots & & & & & \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ d \\ e \\ f \end{bmatrix} = \begin{bmatrix} w_x \\ w_y \\ \dots \end{bmatrix}$$

We can obtain the least squares solution of x=$[a, b, c, d, e, f]^T$ by $x = (A^T A)^{-1} A^T b$. We can obtain camera projection matrix P from intrinsic matrix K and extrinsic matrix $[R|t]$ by $P = K * [R|t]$. After affine transform to align depth and RGB image, and transform pixels to 3D camera frame, we construct the 3D coordinates of the object for RGB and depth. All the RGB and depth calibration transformation matrix will be saved in a file and also in the global variables, so we only need to calibrate once as long as the board and kinect doesn't relocate.

B. HSV

HSV is a common cylindrical-coordinates representation of points in an RGB model. HSV stands for hue, saturation and value. HSV representation is more intuitive and perceptually for computer to distinguish. We convert RGB to HSV image when doing image processing, and detect HSV values for different color blocks for several times when putting the blocks in different position on the grey board. In this way, we can find more accurate HSV value range for different color. And in this way, false detections can be reduced by choose a reasonable range. The HSV range and its recognition accuracy approximation is shown in Fig. 12.

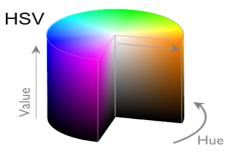
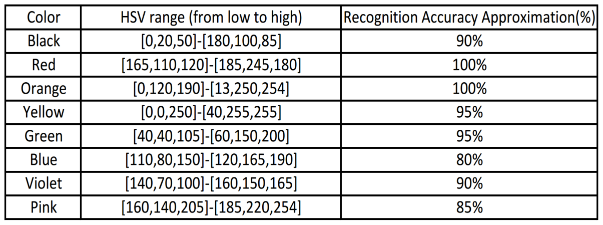


Fig. 11. HSV



Color	HSV range (from low to high)	Recognition Accuracy Approximation(%)
Black	[0,20,50]-[180,100,85]	90%
Red	[165,110,120]-[185,245,180]	100%
Orange	[0,120,190]-[13,250,254]	100%
Yellow	[0,0,250]-[40,255,255]	95%
Green	[40,40,105]-[60,150,200]	95%
Blue	[110,80,150]-[120,165,190]	80%
Violet	[140,70,100]-[160,150,165]	90%
Pink	[160,140,205]-[185,220,254]	85%

Fig. 12. HSV range and accuracy chart

C. Detect Method

We use opencv to detect the blocks and get corresponding information of a specific block. First, image captured from video is converted from RGB to HSV. HSV (Hue, Saturation, Value) is a way of encoding color images. It is sensitive for computer to segment image based on the color of the objects. Second, we use cv2.inRange() function in opencv to produces a binary image setting pixels in the range specified to white, and pixels outside to black. Next, morphological operations are performed on binary images to efficiently remove noise in binary images. We use cv2.morphologyEx() to remove noise and join broken parts of an object, which is same as erosion followed by dilation. In this way, we are able to detect different colors in high accuracy. Then we use cv2.findcontours() to get all the contour information of the specific block as well as the pixel coordinate of the centroid of the block. To increase the accuracy of detection and enhance the robustness of the detector, we use area, perimeter and corner number to check whether it's the block we want to detect. What's more, we use cv2.bitwise_and() to filter the image information outside the greyboard by setting pixels outside to black.

D. Analysis

we have robust computer vision algorithm. As can be seen from the video, we don't need cover board to shadow the block. There is a chance the recognition not working at specific location for specific block under specific light angle, but we can wave hand or use phone light to make it work.

VI. Intelligent Control and Autonomous Execution

A. Teach and Repeat

Teach and repeat is a very common task for robotic arms to do repetitive tasks. We teach the rexarm to play the board game operation. First we set the torque of the motor off while physically moving the arm around and recording a series of waypoint joint angles, then we implement cubic spline algorithm to make the movement smoothly. When we replay the waypoints at both low and high speed and save all the data for further analysis. We find that in lower speed, the repeat process is usually more stable.

B. Cubic Spline

If a command is given to make the arm move to a specific point, every joint will move directly to the desired angle. But sometimes we need to find a trajectory connect initial and final configuration to satisfy the velocity and acceleration constraints, and make the move smooth. In the teach and repeat task we use cubic spline method. There are basically two method to generate cubic spline: joint angle and end point of arm. We adopt method of generating cubic spline of joint angles, and use forward kinematics to get the end point coordinates. This could sample enough waypoints to make the motor move in a smooth trajectory and avoid angle go-stop spike.

If we want to generate a polynomial joint trajectory between two configuration and specify the start and end velocity, we require a polynomial with four independent coefficients. Plus, we also need to define the initial and final time t_0 and t_f, and the time step t_{delta}.

The cubic spline inputs are the initial time t_0, joint angle q_0, joint angle velocity v_0, and the final timet_f, joint angle q_f, joint angle velocity v_f. Then the cubic spline function will compute the cubic spline coefficients for each joint. For the cubic spline coefficients computation, we get four equations for each motor (the i-the step matrix equation is shown as below).

$$\begin{bmatrix} 1 & t_{i0} & t_{i0}^2 & t_{i0}^3 \\ 0 & 1 & 2t_{i0} & 3t_{i0}^2 \\ 1 & t_{if} & t_{if}^2 & t_{if}^3 \\ 0 & 1 & 2t_{if} & 3t_{if}^2 \end{bmatrix} \begin{bmatrix} a_{i0} \\ a_{i1} \\ a_{i2} \\ a_{i3} \end{bmatrix} = \begin{bmatrix} q_{i0} \\ v_{i0} \\ q_{if} \\ v_{if} \end{bmatrix}$$

And then we generate a series of waypoints between q_0 and q_f at a defined time interval t_{delta} (functions are shown as below).

$$q(t) = a_0 + a_1 t + a_2 t^2 + a_3 t^3$$
$$v(t) = \dot{q}(t) = a_1 + 2a_2 t + 3a_3 t^2$$
$$t = t_0, t_0 + t_{delta}, t_0 + 2t_{delta}, ..., t_f$$

As can be seen in Fig. 13, cubic spline can make the angle move continuously without so much go-stop spike, and also for the coordinates trajectory, as Fig. 14 shows, the process trajectory is more smooth than normal.

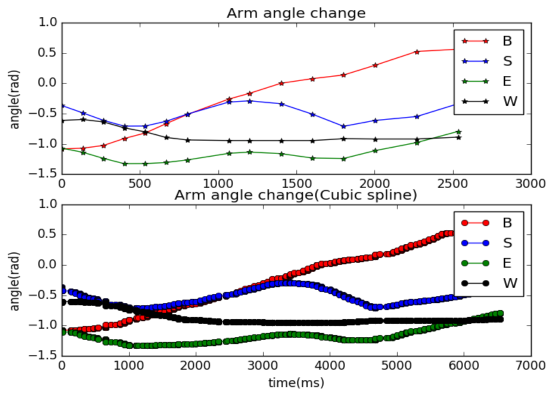


Fig. 13. Angle movement with cubic spline(a part of trajectory).

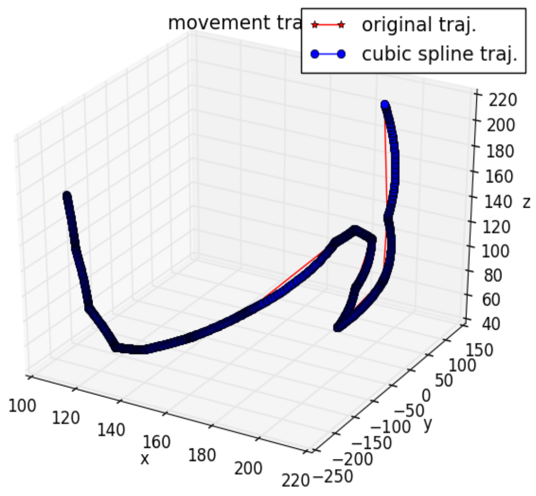


Fig. 14. End effector movement with cubic spline(a part of trajectory).

C. Planning and Decision Making

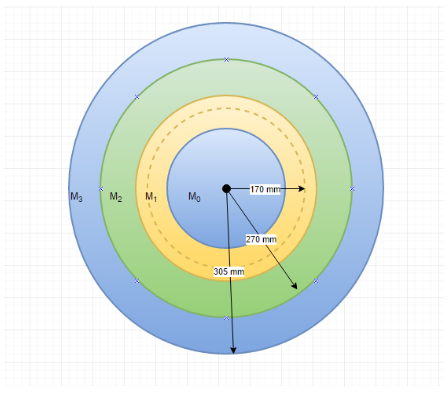


Fig. 15. Planning zone

Due to our Gripper design with 2 more DOF, we were able to implement 3 different modes of inverse kinematics. We divide the board in 3 different zones (as show in Fig. 15) and check the different modes performance over these areas, and this significantly increased our efficiency. We first use perpendicular mode for our safe zone (M2) where (L4, the front arm) was always parallel to the horizontal board surface. Zones (M0) and (M3) are too far for perpendicular mode to work efficiently hence we use swap mode. There is an overlapping area (M1) between zone (M0) and (M2) which serves as an advantage for being able to calibrate the arm in both modes simultaneously. However, inspired by the experimental and theoretical results we implement free pick / iterative mode for all the zones. It is an improved version of perpendicular mode and gives us extra degree of freedom, resulting of which it performed very well in experimental setup. The disadvantage is that the gripper edge may not align with the block edge. But we add transition point in safe zone to re-orientate the block and gripper direction. For tasks 3, 4 and 5 we use a specific location in safe zone for changing gripper orientation by 90,in counter clockwise direction along rotation axis (Z - Axis). Being able to use swap for far points in zone (M3) and free pick / iterative mode in all other points on board, significantly increases our effective grasping area for picking up blocks. The distance measurement is depicted in Fig. 15.

D. Task Completion Strategy

To employ fast but effective way to complete transitions from picking and placing any block, about 20 steps are recognized and manually assigned for tuned torque and speed. However, strategy used is specific to the tasks at hand. A total of five different tasks are performed:
Task 1 (Pick and Place): This task requires us to move 3 different blocks from their original positions with some (x,y) coordinates, which are generated through blob detection to the target position having (x,-y) coordinates with X-axis acting as the mirror. Since the generalized block dimensions are known and the blocks are oriented towards center (center of the bottom of arm), we aligned the arm in direction of block orientation with rotating base motor while maintaining initial position for remaining joints. Using inverse kinematics algorithm, gripper picks up the block and returns to initial position where it aligns itself to the target position using base motor. Basic maths operation flips the Y - Axis and save that as target position. Using Inverse kinematics, gripper drops the block at target position.

Task 2 (Pick and Stack): This task requires 3 blocks to be stacked in a particular order. Same strategy is employed for picking up the blocks as task 1. After picking up the blocks, they are placed at a specific location with same (x,y) coordinates but with a different Z - coordinate (also taking into account the offset generated because of the mechanical parts via observations), which is 38.5 mm (distance between 2 block centers) greater than previous one.

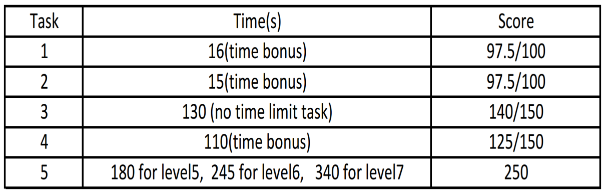
Task 3 (Line them up): This task requires blocks to be placed in a line with order according to required color code: Black, Red, Orange, Yellow, Green, Blue, Violet, White. They are initially placed in a specific configuration, some stacked (no more than 3 high). Using the RGBD output from Kinect camera, we calculate the coordinates and depth of each block. If the blocks appear to be stacked, we unstack each block and place them at specific position in safe zone (knowing that there won't be any block in safe zone). Then using computer vision blob detection we identify the color as well as coordinates of each block. Using Inverse Kinematics we pick up first block (Black) and placed it at a specific predefined (x,y,z) position. For next block placing, the Y and Z Coordinates remain constant while X coordinate is increased with a factor (Delta + width of the cubic block), where delta is the distance we assign between 2 blocks placed in line. However, there is an intermediate transition stage just after picking up block, where the block is placed (at predefined position in safe zone) for re-orientation by 90(counter clockwise along Z axis), to minimize the error in block orientation generated while picking up.

Task 4 (Stack them High): Same initial configuration as task 3 is given with some blocks being stacked (no more than 3 high). Blocks need to be stacked on top of each other in a certain order, as the required color code: Black, Red, Orange, Yellow, Green, Blue, Violet, White. Same blob detection and depth calculating strategy is used to unstack and identify the color of the blocks. After all the blocks are finally placed in fixed predefined (X,Y) coordinates, we start to stack. The Z-coordinates for every next block is increased by a factor equal to the width of the cubic block (38mm) with certain compensation error measured in experiment. Intermediate stage (as explained in task 3) is used for correcting orientation of gripper and block and minimizing errors from initial block orientation while picking.

Task 5 (Pyramid Builder): In this task, blocks are placed randomly on the board. More blocks are added at any time if needed. Final position should resemble a 2D pyramid as wide and tall as it can, regardless to color specification. To make this task fast, we eliminate the color recognition task and only utilize blob detection to calculate the position of block. During training, the coordinates for every consecutive block placing is recorded with respect to the center of the arm. This task also use the intermediate transition stage (mentioned in task 3 and 4) for greater accuracy.

E. Overall Performance

The performance matrix (Fig 13) indicates the time taken (in seconds) for individual task as well as task scores. Every time constraint task was completed well before the time limit indicating the effectiveness of our strategy and the algorithm implementation. The performance video for referencing can be found on google drive file attached.



Task	Time(s)	Score
1	16(time bonus)	97.5/100
2	15(time bonus)	97.5/100
3	130 (no time limit task)	140/150
4	110(time bonus)	125/150
5	180 for level5, 245 for level6, 340 for level7	250

Fig. 16. Task performance chart.

VII. Discussion

The overall performance of the whole system is precise, smooth and fast. In this project, we developed 1) a multi process multi thread system and TCP/UDP communication system to handle different tasks simultaneously with less time than normal single thread in main function; 2) software architecture to run and communicate the command more efficiently, and build the control decision making architecture (including state machine) to make the task running in a more logical and systematical way; 3) appropriate gripper design that makes the gripping job faster and tighter 4) forward kinematic to get the end effector location, and inverse kinematic to use 3 mode 6 DOF manipulation strategy; 5) a computer vision system based on OpenCV that detects objects and extract the coordinates information of them; 6) we optimized our strategy and task orientated parameters to make them fit in experiment environment.

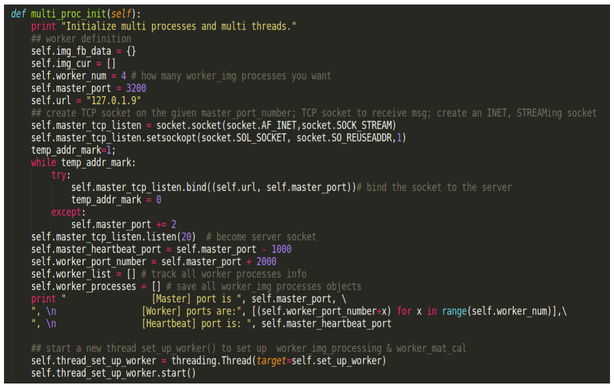
Acknowledgment

Authors would like to thank Prof. Ella Atkins, course instructor, for scientific guidance, Dr. Peter gaskell, lab instructor, for providing us with the proper equipment set and guidance throughout the experimentation process as well as lab sessions, Mr. Theodore Nowak, graduate student instructor (GSI) and Mr. Abhiram Krishnan, instructional aide (IA) for comments, insight and expertise that greatly improved our understanding of all hardware and software systems.

References

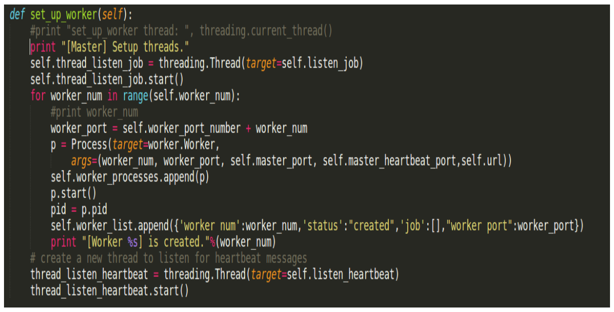
[1] Spong, Mark W., Seth Hutchinson, and Mathukumalli Vidyasagar. Robot modeling and control. Vol. 3. New York: wiley, 2006.

Appendix A



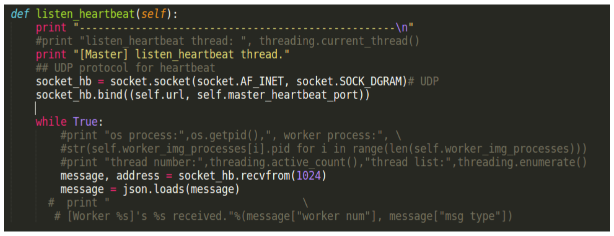
```python
def multi_proc_init(self):
    print "Initialize multi processes and multi threads."
    ## worker definition
    self.img_fb_data = {}
    self.img_cur = []
    self.worker_num = 4 # how many worker_img processes you want
    self.master_port = 3200
    self.url = "127.0.1.9"
    ## create TCP socket on the given master_port_number; TCP socket to receive msg; create an INET, STREAMing socket
    self.master_tcp_listen = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
    self.master_tcp_listen.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR,1)
    temp_addr_mark=1;
    while temp_addr_mark:
        try:
            self.master_tcp_listen.bind((self.url, self.master_port))# bind the socket to the server
            temp_addr_mark = 0
        except:
            self.master_port += 2
    self.master_tcp_listen.listen(20)  # become server socket
    self.master_heartbeat_port = self.master_port - 1000
    self.worker_port_number = self.master_port + 2000
    self.worker_list = [] # track all worker processes info
    self.worker_processes = [] # save all worker_img processes objects
    print "                  [Master] port is ", self.master_port, \
    ", \n                 [Worker] ports are:", [(self.worker_port_number+x) for x in range(self.worker_num)],\
    ", \n                 [Heartbeat] port is: ", self.master_heartbeat_port

    ## start a new thread set_up_worker() to set up  worker_img_processing & worker_mat_cal
    self.thread_set_up_worker = threading.Thread(target=self.set_up_worker)
    self.thread_set_up_worker.start()
```

Fig. 17. multi_proc_init (master process) Function.



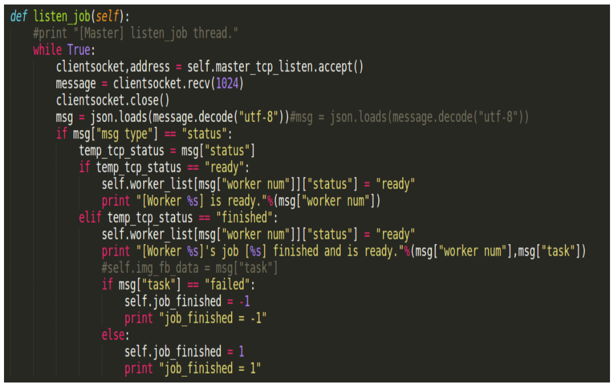
```python
def set_up_worker(self):
    #print "set_up_worker thread: ", threading.current_thread()
    print "[Master] Setup threads."
    self.thread_listen_job = threading.Thread(target=self.listen_job)
    self.thread_listen_job.start()
    for worker_num in range(self.worker_num):
        #print worker_num
        worker_port = self.worker_port_number + worker_num
        p = Process(target=worker.Worker,
            args=(worker_num, worker_port, self.master_port, self.master_heartbeat_port,self.url))
        self.worker_processes.append(p)
        p.start()
        pid = p.pid
        self.worker_list.append({'worker num':worker_num,'status':"created",'job':[],"worker port":worker_port})
        print "[Worker %s] is created."%(worker_num)
    # create a new thread to listen for heartbeat messages
    thread_listen_heartbeat = threading.Thread(target=self.listen_heartbeat)
    thread_listen_heartbeat.start()
```

Fig. 18. set_up_worker (master process) Function.



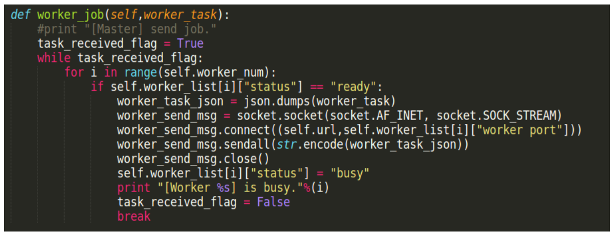
```python
def listen_heartbeat(self):
    print "--------------------------------------------------\n"
    #print "listen_heartbeat thread: ", threading.current_thread()
    print "[Master] listen_heartbeat thread."
    ## UDP protocol for heartbeat
    socket_hb = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)# UDP
    socket_hb.bind((self.url, self.master_heartbeat_port))

    while True:
        #print "os process:",os.getpid(),", worker process:", \
        #str(self.worker_img_processes[i].pid for i in range(len(self.worker_img_processes)))
        #print "thread number:",threading.active_count(),"thread list:",threading.enumerate()
        message, address = socket_hb.recvfrom(1024)
        message = json.loads(message)
      #  print "                                  \
       # [Worker %s]'s %s received."%(message["worker num"], message["msg type"])
```

Fig. 19. listen_heartbeat (master process) Function.



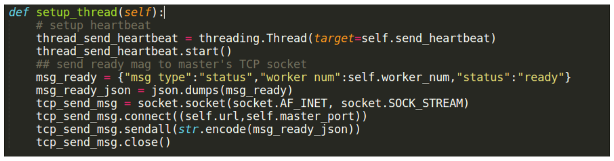
```python
def listen_job(self):
    #print "[Master] listen_job thread."
    while True:
        clientsocket,address = self.master_tcp_listen.accept()
        message = clientsocket.recv(1024)
        clientsocket.close()
        msg = json.loads(message.decode("utf-8"))#msg = json.loads(message.decode("utf-8"))
        if msg["msg type"] == "status":
            temp_tcp_status = msg["status"]
            if temp_tcp_status == "ready":
                self.worker_list[msg["worker num"]]["status"] = "ready"
                print "[Worker %s] is ready."%(msg["worker num"])
            elif temp_tcp_status == "finished":
                self.worker_list[msg["worker num"]]["status"] = "ready"
                print "[Worker %s]'s job [%s] finished and is ready."%(msg["worker num"],msg["task"])
                #self.img_fb_data = msg["task"]
                if msg["task"] == "failed":
                    self.job_finished = -1
                    print "job_finished = -1"
                else:
                    self.job_finished = 1
                    print "job_finished = 1"
```

Fig. 20. listen_job (master process) Function.



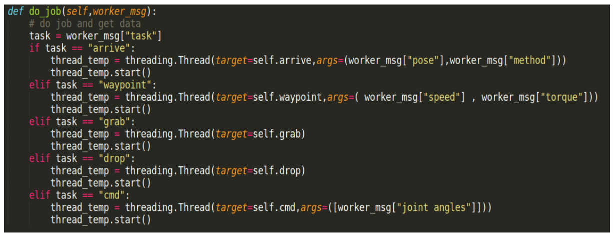
```python
def worker_job(self,worker_task):
    #print "[Master] send job."
    task_received_flag = True
    while task_received_flag:
        for i in range(self.worker_num):
            if self.worker_list[i]["status"] == "ready":
                worker_task_json = json.dumps(worker_task)
                worker_send_msg = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
                worker_send_msg.connect((self.url,self.worker_list[i]["worker port"]))
                worker_send_msg.sendall(str.encode(worker_task_json))
                worker_send_msg.close()
                self.worker_list[i]["status"] = "busy"
                print "[Worker %s] is busy."%(i)
                task_received_flag = False
                break
```

Fig. 21. worker_job (master process) Function.



```python
def setup_thread(self):
    # setup heartbeat
    thread_send_heartbeat = threading.Thread(target=self.send_heartbeat)
    thread_send_heartbeat.start()
    ## send ready mag to master's TCP socket
    msg_ready = {"msg type":"status","worker num":self.worker_num,"status":"ready"}
    msg_ready_json = json.dumps(msg_ready)
    tcp_send_msg = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    tcp_send_msg.connect((self.url,self.master_port))
    tcp_send_msg.sendall(str.encode(msg_ready_json))
    tcp_send_msg.close()
```

Fig. 22. worker_setup_thread (worker process) Function.



```python
def do_job(self,worker_msg):
    # do job and get data
    task = worker_msg["task"]
    if task == "arrive":
        thread_temp = threading.Thread(target=self.arrive,args=(worker_msg["pose"],worker_msg["method"]))
        thread_temp.start()
    elif task == "waypoint":
        thread_temp = threading.Thread(target=self.waypoint,args=( worker_msg["speed"] , worker_msg["torque"]))
        thread_temp.start()
    elif task == "grab":
        thread_temp = threading.Thread(target=self.grab)
        thread_temp.start()
    elif task == "drop":
        thread_temp = threading.Thread(target=self.drop)
        thread_temp.start()
    elif task == "cmd":
        thread_temp = threading.Thread(target=self.cmd,args=([worker_msg["joint angles"]]))
        thread_temp.start()
```

Fig. 23. do_job (worker process) Function.

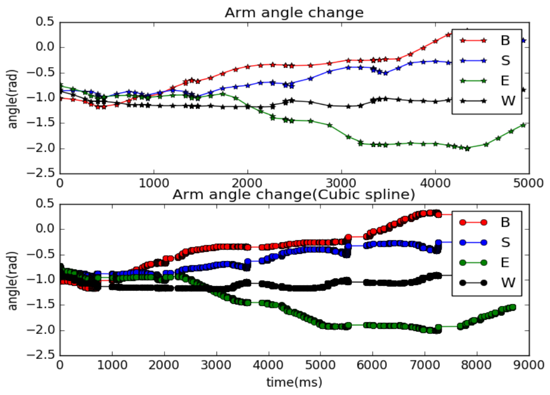


Fig. 24. Angle movement with cubic spline(teach and repeat trajectory).

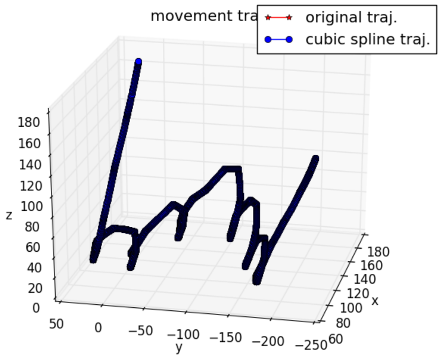


Fig. 25. End effector movement with cubic spline (teach and repeat trajectory).