Term Project Report **Siddharth Shankar Jha**
ME60352 14EE32002
Prof. Dilip Kumar Pratihar

Problem Statement

Design and development of a visual servoing methodology for a 4DOF robotic arm

Introduction

Visual servoing, also known as vision-based robot control and abbreviated VS, is a technique which uses feedback information extracted from a vision sensor to control the motion of a robot. There are multiple types of visual servoing, for example image-based (IBVS) and pose-based (PBVS). In IBVS, the control law is based on the error between current and desired features on the image plane. In PBVS, the pose of the object of interest is estimated with respect to the camera and then a command is issued to the robot controller, which in turn controls the robot.

The feedback for camera position and the path planning are heavily dependent on forward and inverse kinematics, respectively. In forward kinematics, link parameters (link lengths) and joint variables (typically angles) are given and one has to find out the position and orientation of the end-effector. On the other hand, in inverse kinematics, given link parameters and position and orientation of the end effector, one has to find joint variables.

Camera calibration is the process of estimating intrinsic and/or extrinsic parameters. Intrinsic parameters deal with the camera's internal characteristics, such as, its focal length, skew, distortion, and image center. Extrinsic parameters describe its position and orientation in the world.

ArUco is an OpenSource library for camera pose estimation using square markers. ArUco is written in C++ and is extremely fast. Given size of the square markers, an accurate estimate of the 6D pose of the marker can be generated using only a single calibrated monocular camera.

The Hardware

The work done in this report is performed using a single monocular camera, which is used to detect ArUco tags (which represent goals) and plan kinematically constrained paths to reach the goal. Thus it is a position-based visual servoing (PBVS) implementation.

The robotic arm used for the project has a reach of 32cm and has 4 total degrees of freedom. The arm is powered using 5 servo motors, controlled using a Raspberry Pi 2 using an Adafruit PCA9685 PWM shield (since a Raspberry Pi 2 only has 3 PWM supporting pins. The communication between the processor and the PWM driver is through Inter-integrated Circuit (I2C) Protocol. All the image processing and calculations are done on the Raspberry Pi 2.

The robotic arm consists of a rotating base having a revolute joint that rotates the entire top half of the arm. The arm itself consists of two links connected via revolute joints and an end effector, which also houses the camera. The camera used is a calibrated Pi Cam v1.3 with a horizontal FOV of 53 degrees. The parameters for the robotic arm in the DH (Denavit-Hartenberg) notations are as given in the table. Since only 3 servos out of the 4 are manipulated, the end of the 3rd link containing the final servo is considered as the end effector.

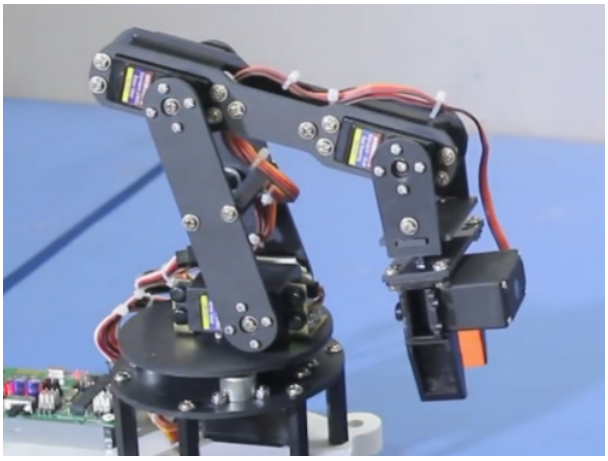

Figure 1: The Robotic arm used for experiments

Table 1: DH parameter table

$\alpha(^o)$	$\theta(^o)$	Joint Offset(cm)	Link Length(cm)
90	0 to 180	3.5	1.7
0	60 to 150	0	9.5
0	0 to 150	0	9.5

Implementation

Camera Calibration

The first step in the implementation is the calibration of the camera. A pinhole camera model with radial and tangential distortions was assumed. The distortion model is the one used by OpenCV. A fixed size checkerboard was clicked in many different orientations at many different positions in the image. The camera matrix and distortion matrix was determined using these input images. Once the matrices were determined, they were used to undistort the image.

ArUco pose estimation

The ArUco tag is detected in the image plane. The side length of the ArUco tag was fed as an input to the program (4cm in this case). 6X6 ArUco tags were used as they offer a good compromise between speed and accuracy. Perspective 4-point technique was used to identify the pose of the tag after the detection. It is used to get the T_{goal}^{camera}.

Forward Kinematics

To get the T_{goal}^{base}, a transformation chain is required. It is given as $T_{goal}^{base} = T_{link1}^{base} \times T_{link2}^{link1} \times T_{end}^{link2} \times T_{camera}^{end} \times T_{goal}^{camera}$. To get the rest of the configurations, forward kinematics need to be solved. This was done using a library called Maddux. Maddux is A Python Robot Arm Toolkit and Simulation Environment. A one-to-one mapping was estabilished between servo angles and the joint angles specified by Maddux and correspondingly T_{end}^{base} was calculated using forward kinematics functions of Maddux.

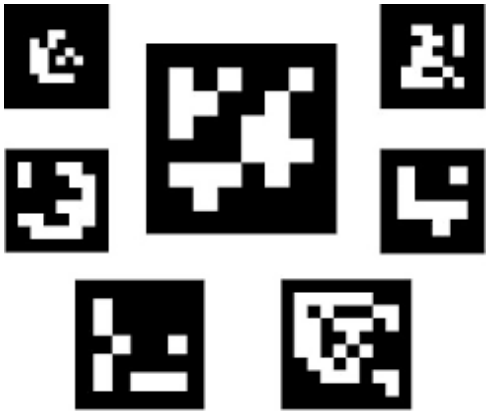

Figure 2: Example ArUco tags

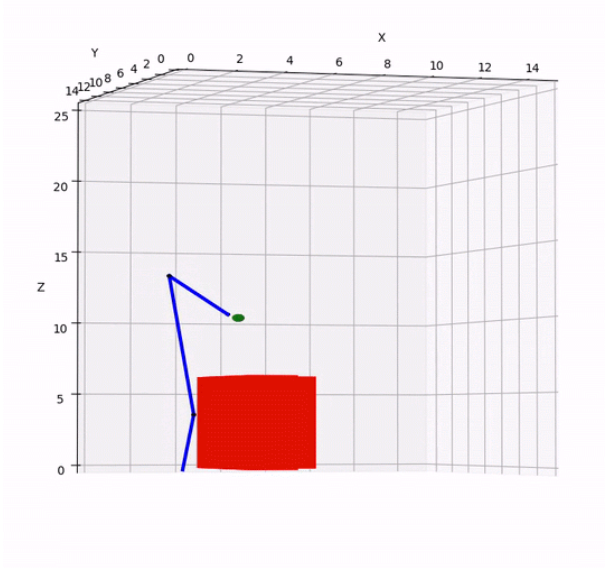

Figure 3: The Simulation Environment in Maddux

Path Planning and Inverse Kinematics

The ArUco tag position was specified as goal in the Maddux simulation environment. Only the translational components of the tag position with respect to the camera were considered, the rotational ones were ignored. Once the goal is specified, maddux calculates incrementally the path which is kinematically constrained and leads the arm to the goal position using the inverse kinematics functions. The final joint states are returned after each step. Again a one-to-one mapping is estabilished between Maddux angles and the servo angles and corresponding angle commands are sent to the PCA9685 board as servo motor commands. Since the final joint is held at a fixed orientation, the T_{camera}^{end} is another parameter that needs to be passed in to the file.

Code

Main Algorithm

```
from __future__ import division
# Import the PCA9685 module.
import Adafruit_PCA9685
from picamera.array import PiRGBArray
from picamera import PiCamera
import time
import numpy as np
import cv2
from maddux.robots.compute_kine import compute_kine

pwm = Adafruit_PCA9685.PCA9685()
camera = PiCamera()
camera.resolution = (640, 480)
camera.framerate = 32
rawCapture = PiRGBArray(camera, size=(640, 480))

#dictionary = cv2.aruco.getPredefinedDictionary(cv2.aruco.DICT_5X5_1000)
dictionary = cv2.aruco.getPredefinedDictionary(cv2.aruco.DICT_6X6_50)
#dictionary = cv2.aruco.getPredefinedDictionary(cv2.aruco.DICT_ARUCO_ORIGINAL)
servo_min = 200  # Min pulse length out of 4096
servo_max = 650  # Max pulse length out of 4096
pwm.set_pwm_freq(60)
time.sleep(0.1)

def set_servo_pulse(channel, pulse):
    pulse_length = 1000000    # 1,000,000 us per second
    pulse_length //= 60       # 60 Hz
    print('{0}us per period'.format(pulse_length))
    pulse_length //= 4096     # 12 bits of resolution
    print('{0}us per bit'.format(pulse_length))
    pulse *= 1000
    pulse //= pulse_length
    pwm.set_pwm(channel, 0, pulse)

def servo_move(channel, angle):
    servo_input = 2.5*angle + 200 # eq of line passing through 0,servo_min and 1
    pwm.set_pwm(channel, 0, int(servo_input))
    time.sleep(1)

def servo_to_maddux(theta, link_no):
    if link_no == 1:
        a = 10*(theta - np.pi/3)/9
        print "angle ", a
        return a
    elif link_no == 2:
```

```
        a = (5*np.pi - 16*theta)/15
        print "angle, ", a
        return a

def maddux_to_servo(theta, link_no):
    if link_no == 1:
        return 0.9 * theta + np.pi/3
    elif link_no == 2:
        return (-15/16)*(theta - np.pi/3)

servo_move(0, 90)
servo_move(1,150)
servo_move(2,150)
servo_move(3,75)

for raw_frame in camera.capture_continuous(rawCapture, format="bgr", use_video_p
    goal_in_camera = np.eye(4)
    frame = raw_frame.array
    # Our operations on the frame come here
    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

    res = cv2.aruco.detectMarkers(gray,dictionary)
    # print(res[0],res[1],len(res[2]))

    if len(res[0]) > 0:
        cv2.aruco.drawDetectedMarkers(gray,res[0],res[1])
        print "ID: ", res[1]
        # print(res[0],res[1],len(res[2]))
        # print(res[0])
        cam = np.matrix([[400, 0, 320], [0, 400, 240], [0, 0, 1]])
        rvec, tvec, _ = cv2.aruco.estimatePoseSingleMarkers(res[0], 1.4, cam, n
        rotationvec = cv2.Rodrigues(rvec)[0]
        print(rotationvec)
        print(tvec)
        goal_in_camera[0:3, 0:3] = rotationvec
        goal_in_camera[0:3, 3:4] = tvec.reshape(3,1)
        # theta_1 = theta_2 = theta_3 = theta4 = 0
        goal_config = compute_kine( np.pi/2, servo_to_maddux(2.618, 1), servo_to_
        print (180/np.pi)*goal_config.ravel().tolist()[0]
        print maddux_to_servo((180/np.pi)*goal_config.ravel().tolist()[1], 1)
        print maddux_to_servo((180/np.pi)*goal_config.ravel().tolist()[2], 2)
        servo_move(0, (180/np.pi)*goal_config.ravel().tolist()[0])
        servo_move(1, maddux_to_servo((180/np.pi)*goal_config.ravel().tolist()[1
        servo_move(2, maddux_to_servo((180/np.pi)*goal_config.ravel().tolist()[2
        print "\n\n Goal Config", goal_config
    # Display the resulting frame
    # cv2.imshow('frame',gray)
    if cv2.waitKey(1) & 0xFF == ord('q'):
        break
```

```
    rawCapture.truncate()
    rawCapture.seek(0)

# When everything done, release the capture
cv2.destroyAllWindows()
```

Maddux Computation

```
import numpy as np
from maddux.environment import Environment
from maddux.objects import Ball
from maddux.robots.arm import Arm
from maddux.robots.link import Link

def custom_arm(q=None):
    # theta, offset, length, twist
    # L1 = Link(0, 0, 0, 0, q_lim=np.array([0, 0]))
    L2 = Link(0, 3.5, 1.7, 1.571, q_lim = np.array([-4.36/2, 4.36/2]))
    L3 = Link(0, 0, 9.5, 0, q_lim = np.array([-2.44/2, 2.44/2]))
    L4 = Link(0, 0, 9.5, 0, q_lim = np.array([-4.36/2, 4.36/2]))
    L5 = Link(0, 0, 0, 0, q_lim = np.array([-4.36/2, 4.36/2]))
    links = np.array([L2, L3, L4])
    q0 = [0, 0, 0, 0, 0]
    q01 = [1.571,0,0,0,0]
    if q is None:
        q02 = [ 1.571,  1.833,  -1.833]
    else: q02 = q
    robot = Arm(links, np.array(q02), 'Custom', 3, None)


    return robot


def compute_kine(theta1, theta2, theta3, goalInCam):
    # goalInCam: 4x4
    si = np.sin(0.0872665)
    co = np.cos(0.0872665)

    camInEndEffector = np.matrix([[1, 0, 0, 0], [0, 0, 1, 0], [0, -1, 0, 0], [0,

    goalInEndEffector = camInEndEffector*goalInCam
    print "goal in end"
    print goalInEndEffector

    custom = custom_arm(q=[theta1, theta2, theta3])
    endEffectorInBaseLink = custom.fkine()
    print "end in base"
    print endEffectorInBaseLink
    goalInBaseLink = endEffectorInBaseLink*goalInEndEffector
```

```
X = goalInBaseLink.ravel().tolist()[0][3]
Y = goalInBaseLink.ravel().tolist()[0][7]
Z = goalInBaseLink.ravel().tolist()[0][11]
print("X = ", X)
print("Y = ", Y)
print("Z = ", Z)
# print X, Y, Z
# Create a ball as our target
ball = Ball(np.array([X, Y, Z]), 0.5, target=True)

# Create our environment
env = Environment([15.0, 15.0, 15.0], dynamic_objects=[ball],
                  robot=custom)

# env.animate(5.0)
# Run inverse kinematics to find a joint config that lets arm touch ball
return custom.ikine(ball.position)
```

Results and Discussions

A fast and simple way to follow a fiducial marker using only visual feedback was proposed. Tests were carried out with different orientations of end effector and successful path planning was achieved. Future work can be incorporation of obstacle avoidance in real-time and tracking real objects instead of fiducial markers, which is a research problem in itself.

Attachments

Demonstration Video

Declaration

I solemnly acknowledge that all the work in this project was done by me, i.e. Siddharth Shankar Jha sans any plagiarism.