UGV PROJECT REPORT

MUNYAKABERA JEAN CLAUDE 2019040146006

DECEMBER 17, 2021

UESTC

INTRODUCTION

THE UGV

Figure 1 dji robomaster ep ugv

The unmanned ground vehicle used was the DJI ROBOMASTER EP, it has multiple sensors like an infrared depth sensor, various hit sensors on the side and a 5 megapixels camera sensor that outputs at a maximum of 2560 by 1440 pixels.

The arm and claw can stretch to a length of 10 cm it is composed of 3 servo motors and a number linkages that enables the claw to stay straight while the arm is moving.

THE TASK

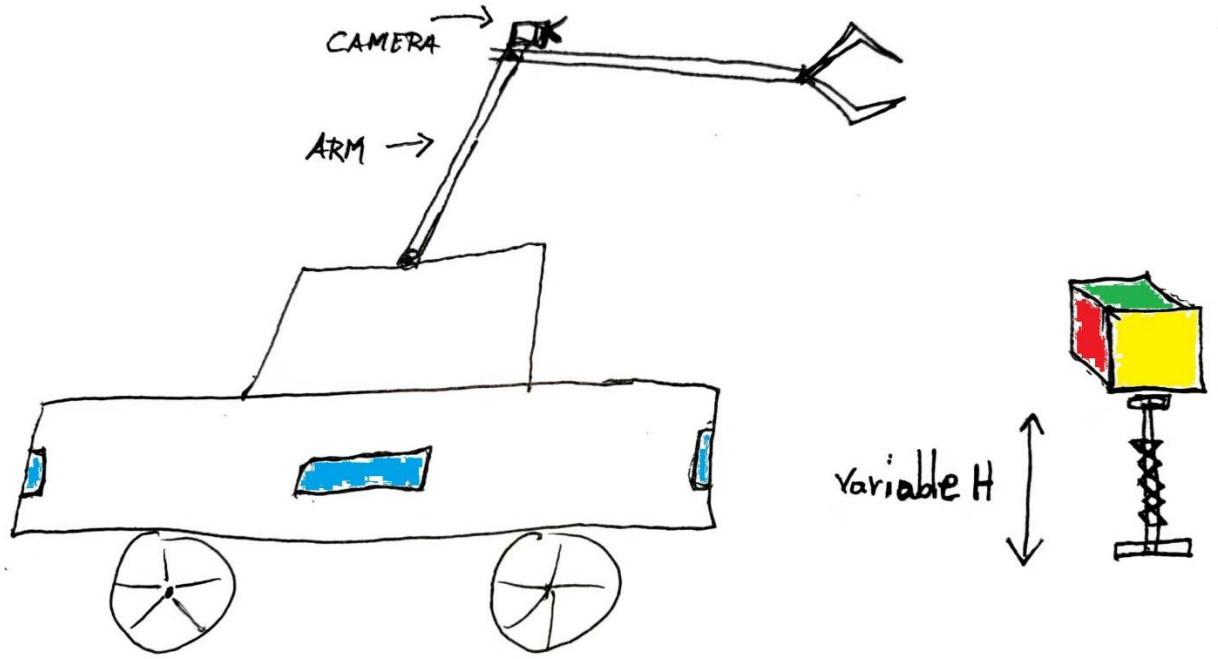

Figure 2 UGV colored object picking task

The goal is for the UGV to be able to pick up a colored object (a Rubik's cube's red side) at any height and length from the UGV.

This poses different problems since we only have a single camera and the output is a 2D matrix with a maximum of 2560 by 1440 but in practice it is reduced to 640 by 480, and so we don't have any 3D data that we can use to infer the length and depth of various object, in this report we will go through how this was solved, another issue pertaining mainly to robotics is how to move the arm to a particular point, fortunately the robomaster SDK and the design of the arm and claw

make it easy to work out without using the usual inverse and forward kinematics math, this report will provide a derivation of both inverse and forward kinematics.

IMPLEMENTATION

COMPUTER VISION

The computer vision task uses the HSV color space parameters and canny edge detection parameters in openCV to be able to emphasize a certain color, following is the process by which this is done.

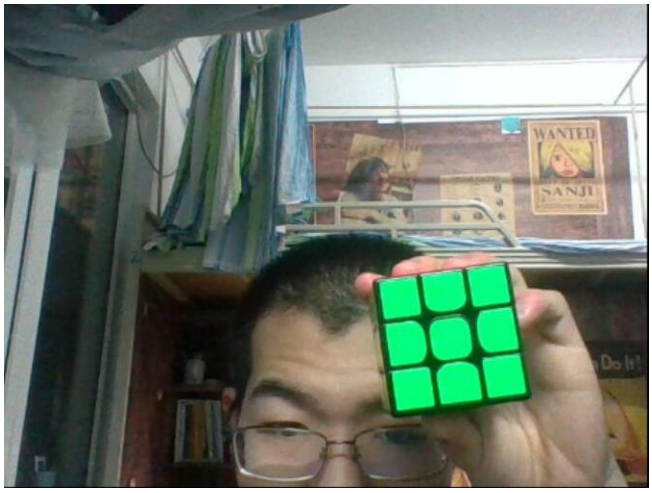

Figure 3 a classmate holding a rubik's cube

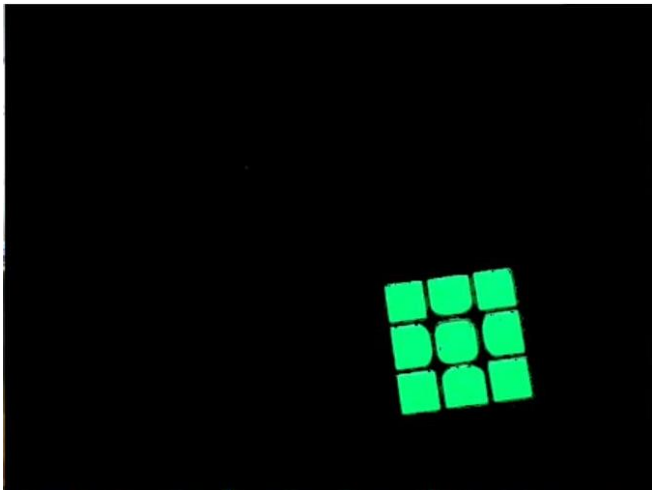

Figure 4 when HSV parameters have been applied to extract the green color

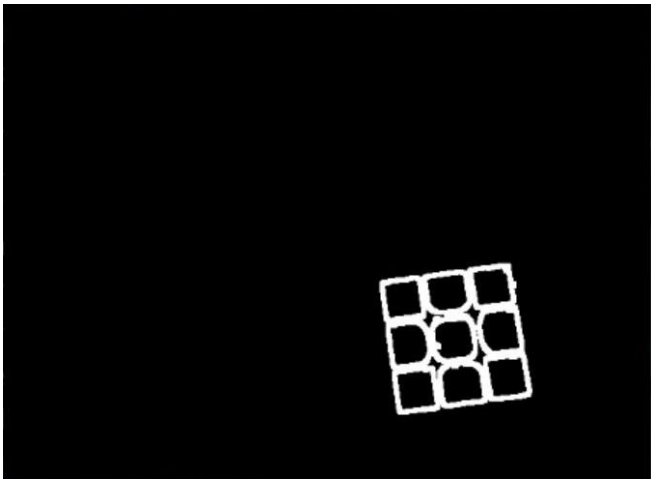

Figure 5 after applying canny edge detection

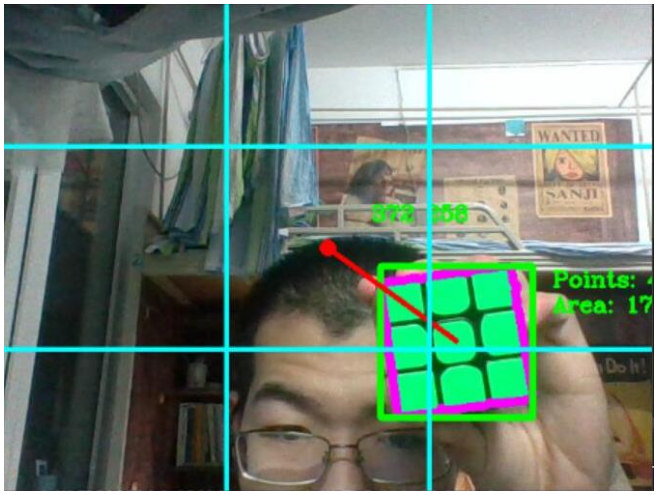

Figure 6 here we see the contours and area of the detected object

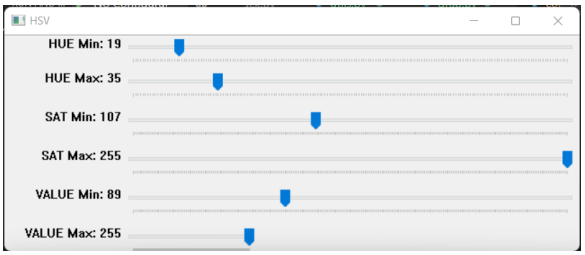

Figure 7 HSV parameters

Figure 8 canny edge detection parameters

And so we use this parameters to extract objects of a defined color and use their position relative to the picture's center and the area they occupy in the picture to infer if the robot is left or right to them and if it is far or near them.

PID AND CONTROLLING MOTION

```python
def trackObj(me, info, w,h, pidSpeed, pErrorSpeed,pidUd, pErrorUd):
    global change,timeWaited,waitTime
    area = info[1]
    x, y,radius = info[0]
    fb = 0

    errorSpeed = x - w // 2
    errorUd = y - h // 2
    speed = pidSpeed[0] * errorSpeed + pidSpeed[1] * (errorSpeed - pErrorSpeed)
    speed = int(np.clip(speed, -100, 100))
    ud = pidUd[0] * errorUd + pidUd[1] * (errorUd - pErrorUd)
    ud = int(np.clip(ud, -20, 20))

    if area > fbRange[0] and area < fbRange[1]:
        fb = 0
    if area > fbRange[1]:
        fb = -0.1
    elif area < fbRange[0] and area > 0:
        fb = 0.1

    if x == 0:
        speed = 0
        fb=0
        ud=0
        errorUd = 0
        errorSpeed = 0
    ep_chassis.drive_speed(x=fb, y=0, z=speed, timeout=1)
    return errorSpeed,errorUd,fb
```

Figure 9 tracking object with areas and PID

This function controls the robot motion to follow an object, it uses the area of a detected object from the earlier section and using 2 constraints (in fbrange) where you constraint the areas where the robot should stop moving once it is in that range, but if it is large that fbrange[0] it moves forward and if it is larger than fbrange[1] it moves backwards.

For the robot to move left and right we use yaw not just changing the Y vector, the reason is that if PID was used on the Y vector it would oscillate before it reaches an optimum range, but using yaw means the oscillation won't add more errors and hence leads to reaching a faster optimum range, the PID parameters are P=0.4, I=0.4, D=0, from this even though one might argue that the differential argument D is needed to negate some of the error incurred, from this project we saw that if a D value is introduced it oscillates more and takes more time to reach the optimum range and since the target is not very far reaching faster optimum range, made us choose these parameters.

After the PID and forward and backward ranges are calculated they are sent to the UGV to change its x and z (yaw) vectors, this is done every time we process an image feed and recognize objects, when the robot has reached the target and it hasn't moved for a certain number of frames (currently 100) it starts to pick the object

FORWARD & INVERSE KINEMATICS

The robot arm must be told where to go, but even though in the robomaster SDK it is as simple as giving it the X and Y coordinates, behind the scenes it does more than that, so below is my attempt at explain how it achieves this, the reader must understand that this is not how exactly the robomaster EP achieves this since it is engineered in a way to make giving the X and Y much simpler that what we discuss here due, to its system of linkages but nonetheless here we provide an idealized version of a 2 DOF robot arm.

FORWARD FINEMATICS

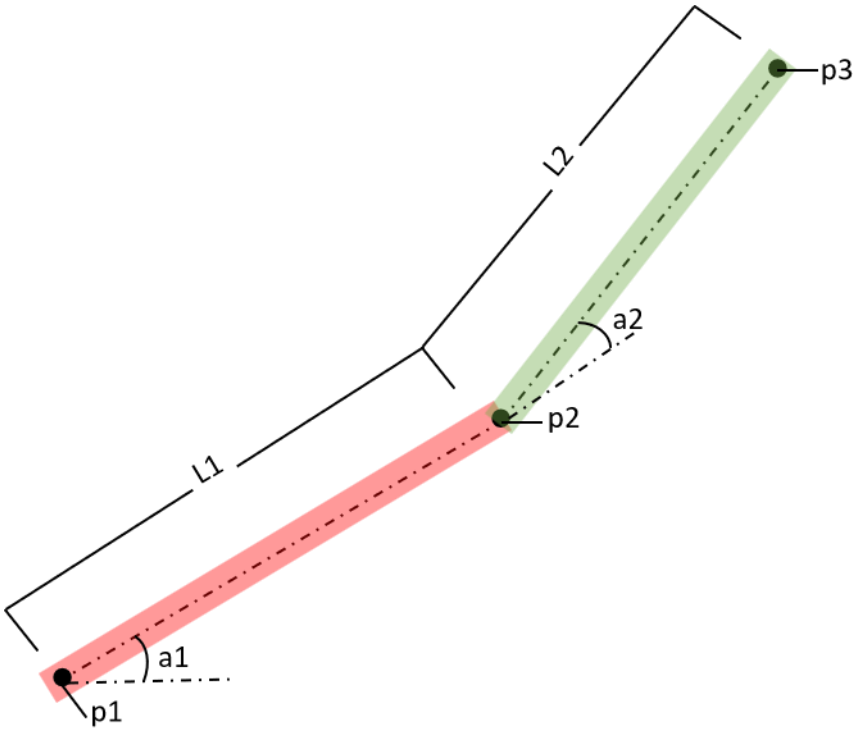

Figure 10 robot arm diagram

In this section we are going to ask a question in the form of "**given all the angles (a1, a2) where would all the joints be**" then we derive the equations to calculate the x and y coordinates of each joint (p1(x1, y1), p2(x2, y2), p3(x3, y3))

Since p1 will always be at the origin it is equal to:

$x1 = 0$

$y1 = 0$

At joint p2 we will need to apply rotation on a1 R (a1) and translation T_x (L1) from p1 to p2 on the x axis which gives us:

$$p2 = \begin{bmatrix} \cos a1 & -\sin a1 & L1\cos a1 \\ \sin a1 & \cos a1 & L1\sin a1 \\ 0 & 0 & 1 \end{bmatrix}$$

Therefore:

$x2 = L1\cos a1$

$y2 = L1\sin a1$

Since p3 depends on the rotation and translation of p2, it needs to be taken into account also it will rotate on a2 R (a2) and translate T_x (L2) from p2 to p3 on the x axis which gives us:

$$p2 = \begin{bmatrix} \cos a1 + a2 & -\sin a1 + a2 & (L2\cos a1 + a2) + L1\cos a1 \\ \sin a1 + a2 & \cos a1 + a2 & (L2\sin a1 + a2) + L1\sin a1 \\ 0 & 0 & 1 \end{bmatrix}$$

Which gives us

$x3 = (L2\cos a1 + a2) + L1\cos a1$

$y3 = (L2\sin a1 + a2) + L1\sin a1$

INVERSE KINEMATICS

In this section we are asking a question of this form **"given p3 (x3, y3) and L2, L1 what is the angles a1 and a2"** and so let's start for the equation of p3.

$x3 = (L2\cos a1 + a2) + L1\cos a1$

$y3 = (L2\sin a1 + a2) + L1\sin a1$

Let's square and add x3 and y3 together

$x_3^2 + y_3^2 = L_1^2 + L_2^2 + 2L_1L_2\cos a_2$ (1)

So $a_2 = \cos^{-1}\frac{x_3^2+y_3^2-L_1^2-L_2^2}{2L_1L_2}$

Now if we apply the sum of angles identities on p3 (x3, y3) we get

$x_3 = (L_1 + L_2\cos a_2)\cos a_1 - L_2\sin a_2\sin a_1$

$y_3 = (L_1 + L_2\cos a_2)\sin a_1 - L_2\sin a_2\cos a_1$

It is known that

$a\cos\theta + b\sin\theta = c$

$$\theta = \tan^{-1}\frac{c}{\pm\sqrt{a^2+b^2-c^2}} - \tan^{-1}\frac{a}{b}$$

So from the above if we consider the equation of y3

$a = L_2\sin a_2\,,\ \ b = L_2 + L_1\cos a_2\ , c = y_3$

So we get

$$a_1 = \tan^{-1}\frac{y_3}{\sqrt{L_1^2 + L_2^2 + 2L_1L_2\cos a_2 - y_3^2}} - \tan^{-1}\frac{L_2 \sin a_2}{L_2 + L_1 \cos a_2}$$

From equation (1) a1 becomes

$$a_1 = \tan^{-1}\frac{y_3}{x_3} - \tan^{-1}\frac{L_2 \sin a_2}{L_2 + L_1 \cos a_2}$$

If we only use $\tan^{-1}$ we can only find solution in the positive x axis, so we can replace $\tan^{-1}$ with atan2 to be able to find solutions in all quadrants

So $a_1 = atan2(y_3, x_3) - atan2(L_2 \sin a_2, L_2 + L_1 \cos a_2)$

To summarize

$$a_2 = \pm\cos^{-1}\frac{x_3^2 + y_3^2 - L_1^2 - L_2^2}{2L_1L_2}$$

$$a_1 = atan2(y_3, x_3) - atan2(L_2 \sin a_2, L_2 + L_1 \cos a_2)$$

PUTTING IT ALL TOGETHER

Following is how inverse and forward kinematics work together, briefly the robot will need to move to a certain point (x3, y3) then we pass that to the inverse kinematics equations then get the angles after which we pass it to the forward kinematics equations, where we get the values of each joint p1 (always at origin (0, 0), p2 and p3).

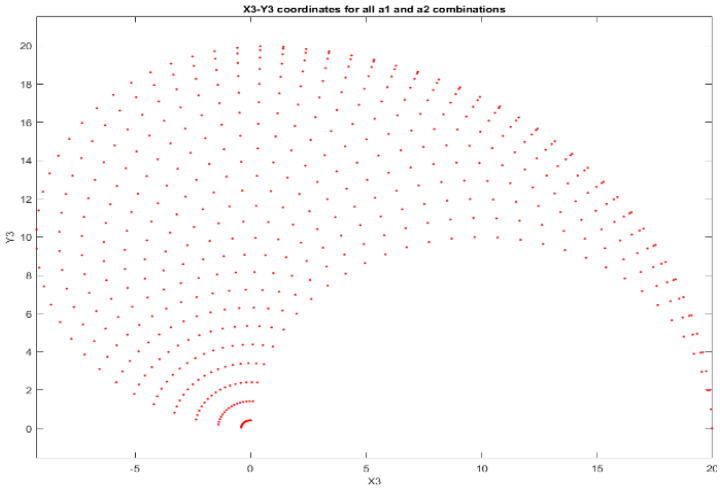

Figure 11 all possible positions the robot arm can reach

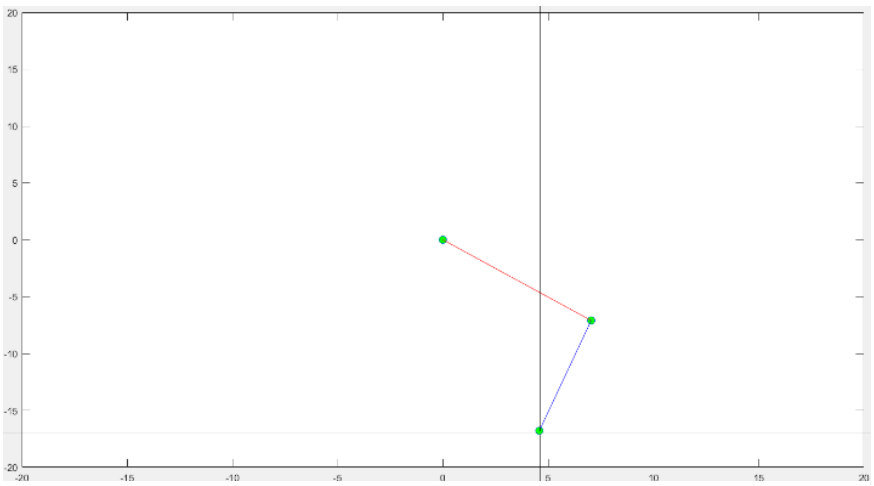

Figure 12 the robot arm simulation in a graph

The figure above shows all the possible positions the robot arm can reach, following is a figure of the simulation done in Matlab, it was made to be simple and so just use a graph for both input and output.

INFERING DEPTH & HEIGHT OF THE OBJECTS TO PICK

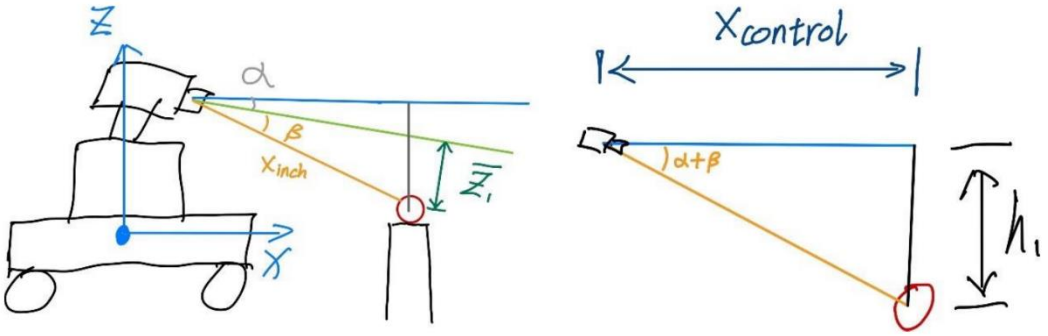

Figure 13 a diagram of the robot camera perspective to the object

Following will be a derivation of how we get the values we use in the following code for the robot to be able to pick up objects.

```python
def calculateHeightAndBeta(radius, center, inches):
    KNOWN_WIDTH = 6.5
    KNOWN_DIS = 32
    KNOWN_HEIGHT = 21.5
    KNOWN_RADIUS = 35
    KNOWN_INCHES = 40.83

    Z_1_BA = KNOWN_WIDTH / (KNOWN_RADIUS * 2) * 44
    BETA = np.arcsin(Z_1_BA / KNOWN_INCHES)
    ALPHA = np.arctan(KNOWN_HEIGHT / KNOWN_DIS) - BETA

    beta = np.arcsin((KNOWN_WIDTH * (center[1] - 240)) / (2 * radius * inches))
    height = inches * np.sin(ALPHA + beta)
    print("height:", height)

    height_real = (65 - 0) / (20.2258 - 27.9855) * (height - 27.9855)

    return height_real

def grab(ep_robot, radius, center, inches):
    ep_chassis = ep_robot.chassis
    ep_arm = ep_robot.robotic_arm
    ep_gripper = ep_robot.gripper

    real_height = calculateHeightAndBeta(radius, center, inches)-3
    print("real_height:", real_height)

    ep_gripper.open()
    time.sleep(3)
    ep_arm.moveto(x=180, y=real_height).wait_for_completed()
    ep_chassis.drive_speed(x=0.1, y=0, z=0, timeout=1)
    time.sleep(0.2)
    ep_gripper.close()
    time.sleep(3)

def distance_to_camera(knownWidth, focalLength, perWidth):
    return (knownWidth * focalLength) / perWidth
```

Figure 14 the 3 functions that are used to coordinate the arm

Function *calculateHeightAndBeta calculates the real height and angle at which the object is relative to the arm, the* distance_to_camera gives the distance of the object relative to the robot and once we have that we pass it to *grab* use both to grab the object, following is the derivation,

$$K = \frac{W}{2r}$$

$$\overline{Z_1} = \frac{W}{2r} Z_1$$

And $\beta = \sin^{-1}\left(\frac{\overline{Z_1}}{X_{inch}}\right) = \sin^{-1}\left(\frac{W \cdot Z_1}{2r \cdot X_{inch}}\right)$

$$\tan(\alpha + \beta) = \frac{H_1}{X_{control}}$$

$$(\alpha + \beta) = \tan^{-1}\left(\frac{H_1}{X_{control}}\right)$$

$$\alpha = \tan^{-1}\left(\frac{H_1}{X_{control}}\right) - \sin^{-1}\left(\frac{W \cdot Z_1}{2r \cdot X_{inch}}\right)$$

And then we can obtain the value *height* and *beta* that we used in the function *calculateHeightAndBeta* with

$$\beta' = \sin^{-1}\left(\frac{W \cdot Z_1}{2r \cdot X_{inch}}\right)$$

$$H_1{}' = X_{inch} \cdot \sin(\alpha + \beta')$$

In the *calculateHeightAndBeta* function these calculations are passed to the *grab* function, which then instructs the arm to go to the desired height and moves the arm and picks the object (gripper open then close) after picking the object we return to a predefined place (not included in the above code snippet).

CONCLUSIONS: results, limitations & future work

The UGV was successfully able to pick an object from different heights (depending on the capability of the UGV arm to reach there) and it could drive to the target autonomously and come back with the object.

The UGV is limited in that when it has reached the target it does the calculations on one frame and can't keep adjusting as it goes to pick the object, we were provided with equations that can achieve that but with limited time it was not implement in the code, perhaps it will be done in the future.

Since it relies only on color it sometimes has issues seeing the right color when the lighting is dramatically changed or it might see clearly in a time we want to make a single color homogenous (since we are using a Rubik's cube that has smaller squares of the same color and we want to see the whole face not just one small square) the image had to be dilated so we don't consider small corners, and being that it just uses color other objects that have the same color might confused it, we corrected this by just considering the biggest area of that color which would usually be the target but a machine learning / deep learning technic like YOLO that we can train to know the shape and color would be much better.

APPENDIX

Forward kinematics implementation

```
function [x1,x2,x3,y1,y2,y3] = forwardkinematics (a1,a2,a3)
   l1=10;
   l2=10;
   x1=0;
   y1=0;
   x2=l1*cos(a1);
   y2=l1*sin(a1);
   x3=x2+l2*cos(a1+a2);
   y3=y2+l2*sin(a1+a2);
end
```

Inverse kinematics implementation

```
function [a1,a2,a3] = inversekinematics (x,y)
l1=10;
l2=10;
a2=acos((x^2+y^2-l1^2-l2^2)/(2*l1*l2));
a1=atan2(y,x)-atan2((l2*sin(a2)),(l1+l2*cos(a2)));
a3=0;
end
```

Robot animation implementation

```
a1=deg2rad(-120);
a2=deg2rad(45);
a3=deg2rad(90);

figure;

while(1)

[x1,x2,x3,y1,y2,y3]=forwardkinematics(a1,a2,a3);
disp([x3,y3,rad2deg(a2)]);
plot([x1,x2,x3],[y1,y2,y3],'o','MarkerFaceColor','g','Marke
rSize',8);
line([x1,x2],[y1,y2],'color','r');
line([x2,x3],[y2,y3],'color','b');
axis([-20 20 -20 20]);

[x,y]=ginput(1);
[a1,a2,a3]=inversekinematics(x,y);
```

```
disp('      x           y       deg');
disp([x,y,rad2deg(atan2(y,x))]);

end
```

Robot workspace graph implementation

```
l1 = 10;
l2 = 10;

a1 = 0:0.1:pi/2;
a2 = 0:0.1:pi;
[a1,a2] = meshgrid(a1,a2);

X = l1 * cos(a1) + l2 * cos(a1 + a2);
Y = l1 * sin(a1) + l2 * sin(a1 + a2);

plot(X(:),Y(:),'r.');
  axis equal;
  xlabel('X3','fontsize',10)
  ylabel('Y3','fontsize',10)
  title('X3-Y3 coordinates for all a1 and a2
combinations','fontsize',10)
```

UGV control implementation

```
import cv2
import numpy as np
from robomaster import robot
from robomaster import camera
import time

frameWidth = 640
w=frameWidth
frameHeight = 480
h=frameHeight


deadZone=100
global imgContour

debug=False
testTime=0
waitTime=0.5
change=0
```

```python
timeWaited=0
if not debug:
    ep_robot = robot.Robot()
    ep_robot.initialize(conn_type="ap")
    ep_camera = ep_robot.camera
    ep_camera.start_video_stream(display=False, resolution='480p')
    ep_chassis = ep_robot.chassis
else:
    me=""
multiplier=1
w, h = frameWidth*multiplier, frameHeight*multiplier
frameWidth,frameHeight,deadZone=w,h,50
fbRange = [4900*(multiplier*multiplier),6000*(multiplier*multiplier)]#[6200,
6800]
pidSpeed = [0.4, 0.4, 0]
pErrorSpeed = 0
pidUd = [0.4, 0.4, 0]
pErrorUd = 0

def calculateHeightAndBeta(radius, center, inches):
    KNOWN_WIDTH = 6.5
    KNOWN_DIS = 32
    KNOWN_HEIGHT = 21.5
    KNOWN_RADIUS = 35
    KNOWN_INCHES = 40.83

    Z_1_BA = KNOWN_WIDTH / (KNOWN_RADIUS * 2) * 44
    BETA = np.arcsin(Z_1_BA / KNOWN_INCHES)
    ALPHA = np.arctan(KNOWN_HEIGHT / KNOWN_DIS) - BETA

    beta = np.arcsin((KNOWN_WIDTH * (center[1] - 240)) / (2 * radius * inches))
    height = inches * np.sin(ALPHA + beta)
    print("height:", height)

    height_real = (65 - 0) / (20.2258 - 27.9855) * (height - 27.9855)

    return height_real

def grab(ep_robot, radius, center, inches):
    ep_chassis = ep_robot.chassis
    ep_arm = ep_robot.robotic_arm
    ep_gripper = ep_robot.gripper

    real_height = calculateHeightAndBeta(radius, center, inches)-3
    print("real_height:", real_height)
```

```
    ep_gripper.open()
    time.sleep(3)
    ep_arm.moveto(x=180, y=real_height).wait_for_completed()
    ep_chassis.drive_speed(x=0.1, y=0, z=0, timeout=1)
    time.sleep(0.2)
    ep_gripper.close()
    time.sleep(3)

def distance_to_camera(knownWidth, focalLength, perWidth):
    return (knownWidth * focalLength) / perWidth

def getContours(img,imgContour):
    myObjectListData = []
    myObjectListC = []
    myObjectListArea = []
    contours, hierarchy = cv2.findContours(img, cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_NONE)
    for cnt in contours:
        area = cv2.contourArea(cnt)
        areaMin =200# cv2.getTrackbarPos("Area", "Parameters")
        if area > areaMin:
            cv2.drawContours(imgContour, cnt, -1, (255, 0, 255), 7)
            peri = cv2.arcLength(cnt, True)
            approx = cv2.approxPolyDP(cnt, 0.02 * peri, True)
            #print(len(approx))
            x , y , w, h = cv2.boundingRect(approx)
            cx = x + w // 2
            cy = y + h // 2
            area = w * h
            ((x_, y_), radius) = cv2.minEnclosingCircle(cnt)
            myObjectListC.append((cx,cy,radius))
            myObjectListArea.append((area))
            myObjectListData.append([x,y,w,h,approx])
    if len(myObjectListArea) > 0:
        i = myObjectListArea.index(max(myObjectListArea))

        cv2.circle(imgContour, (myObjectListC[i][0],myObjectListC[i][1]), 5, (0,
255, 0), cv2.FILLED)
        x,y,w,h,approx=myObjectListData[i]
        cv2.rectangle(imgContour, (x , y ), (x + w , y + h ), (0, 255, 0), 5)

        cv2.putText(imgContour, "Points: " + str(len(approx)), (x + w + 20, y +
20), cv2.FONT_HERSHEY_COMPLEX, .7,
                    (0, 255, 0), 2)
```

```
        cv2.putText(imgContour, "Area: " + str(int(area)), (x + w + 20, y + 45),
cv2.FONT_HERSHEY_COMPLEX, 0.7,
                    (0, 255, 0), 2)
        cv2.putText(imgContour, " " + str(int(x))+ " "+str(int(y)), (x - 20, y-
45), cv2.FONT_HERSHEY_COMPLEX, 0.7,
                    (0, 255, 0), 2)

        cx = int(x + (w / 2))
        cy = int(y + (h / 2))

        if (cx <int(frameWidth/2)-deadZone):
            cv2.putText(imgContour, " GO LEFT " , (20, 50),
cv2.FONT_HERSHEY_COMPLEX,1,(0, 0, 255), 1)

        elif (cx > int(frameWidth / 2) + deadZone):
            cv2.putText(imgContour, " GO RIGHT ", (20, 50),
cv2.FONT_HERSHEY_COMPLEX,1,(0, 0, 255), 1)

        elif (cy < int(frameHeight / 2) - deadZone):
            cv2.putText(imgContour, " GO UP ", (20, 50),
cv2.FONT_HERSHEY_COMPLEX,1,(0, 0, 255), 1)

        elif (cy > int(frameHeight / 2) + deadZone):
            cv2.putText(imgContour, " GO DOWN ", (20, 50),
cv2.FONT_HERSHEY_COMPLEX, 1,(0, 0, 255), 1)

        cv2.line(imgContour, (int(frameWidth/2),int(frameHeight/2)), (cx,cy),
                 (0, 0, 255), 3)

        return imgContour, [myObjectListC[i], myObjectListArea[i]]
    else:
        return imgContour,[[0,0,0],0]



def trackObj(me, info, w,h, pidSpeed, pErrorSpeed,pidUd, pErrorUd):
    global change,timeWaited,waitTime
    area = info[1]
    x, y,radius = info[0]
    fb = 0

    errorSpeed = x - w // 2
    errorUd = y - h // 2
    speed = pidSpeed[0] * errorSpeed + pidSpeed[1] * (errorSpeed - pErrorSpeed)
    speed = int(np.clip(speed, -100, 100))
```

```
    ud = pidUd[0] * errorUd + pidUd[1] * (errorUd - pErrorUd)
    ud = int(np.clip(ud, -20, 20))

    if area > fbRange[0] and area < fbRange[1]:
        fb = 0
    if area > fbRange[1]:
        fb = -0.1
    elif area < fbRange[0] and area > 0:
        fb = 0.1

    if x == 0:
        speed = 0
        fb=0
        ud=0
        errorUd = 0
        errorSpeed = 0

    cv2.putText(imgContour, "LR: "+str(speed)+" FB: "+str(fb)+" UD: "+str(-
ud),( 5, 200), cv2.FONT_HERSHEY_COMPLEX_SMALL, 1.2,(0, 0, 255), 2)
    cv2.putText(imgContour, "err LR: "+str(errorSpeed)+" err UD: "+str(x)+" tm
wtd: "+str(timeWaited),( 5, 220), cv2.FONT_HERSHEY_COMPLEX_SMALL, 1.2,(0, 0,
255), 2)
    #time.sleep(0.5)
    if not debug:
        ep_chassis.drive_speed(x=fb, y=0, z=speed, timeout=1)
        pass
    return errorSpeed,errorUd,fb

if debug:
    cap = cv2.VideoCapture(0)

now=time.time()
fbcount=0
fbsum=0
ffb=1
while True:
    if testTime !=0 and (time.time()-now >=testTime):
        if not debug:
            pass
        break
    if debug:
        _, img = cap.read()
        img = cv2.resize(img, (w, h))
    else:
        img = ep_camera.read_cv2_image(timeout=3, strategy='newest')
```

```
        img = cv2.resize(img, (w, h))

    imgContour = img.copy()
    imgHsv = cv2.cvtColor(img, cv2.COLOR_BGR2HSV)
    lower = np.array([0,100,100])#red
    upper = np.array([10,255,255])#red
    mask = cv2.inRange(imgHsv,lower,upper)
    result = cv2.bitwise_and(img,img, mask = mask)
    mask = cv2.cvtColor(mask, cv2.COLOR_GRAY2BGR)
    imgBlur = cv2.GaussianBlur(result, (7, 7), 1)
    imgGray = cv2.cvtColor(imgBlur, cv2.COLOR_BGR2GRAY)
    threshold1 = 0
    threshold2 = 27
    imgCanny = cv2.Canny(imgGray, threshold1, threshold2)
    kernel = np.ones((5, 5))
    imgDil = cv2.dilate(imgCanny, kernel, iterations=2)
    img, info = getContours(imgDil , imgContour)

    if ffb==0:
        fbcount+=1
    if fbcount<100:
        pErrorSpeed,pErrorUd,ffb = trackObj(ep_chassis, info, w,h, pidSpeed,
pErrorSpeed,pidUd,pErrorUd)
    else:
        radius=info[1]
        centerx,centery, radius =info[0]
        center=(centerx, centery)
        KNOWN_DISTANCE = 100
        KNOWN_WIDTH = 6.5
        focalLength = (2 * 14.3 * KNOWN_DISTANCE) / KNOWN_WIDTH
        dist = distance_to_camera(KNOWN_WIDTH, focalLength, 2 * radius)
        grab(ep_robot, radius, center, dist)
        ep_chassis.drive_speed(x=-0.5, y=0, z=0, timeout=1)
        time.sleep(3)
        ep_robot.robotic_arm.moveto(x=70, y=40).wait_for_completed()
        ep_robot.gripper.open()
        time.sleep(2)
        ep_chassis.drive_speed(x=-0.1, y=0, z=0, timeout=1)
        time.sleep(3)
        ep_robot.gripper.close()
        time.sleep(1)
        break

    cv2.imshow("output", img)
    if cv2.waitKey(1) & 0xFF == ord('q'):
```

```
if not debug:
    pass
break
```