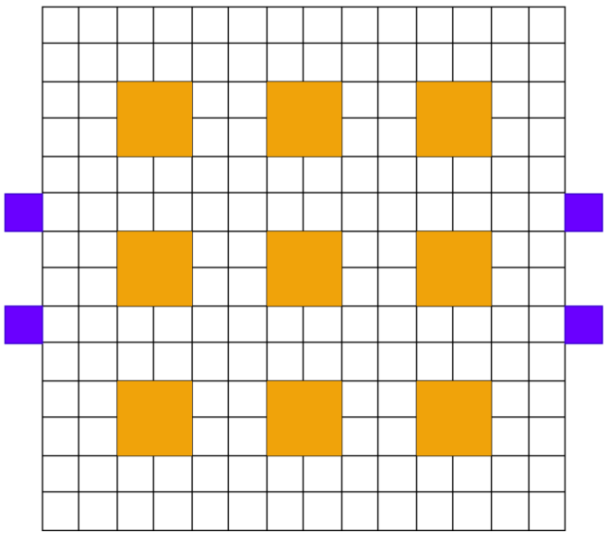
Operating bots via a central navigation system

The problem statement:

We were given a 7ft x 7ft grid as shown.



Each unit of this grid was 6inches. The purple squares are supply points, each with a different ID, i.e., s1, s2, s3, s4. All the yellow squares were drop points d1 - d9.

As input- we would be given an excel sheet of 3 columns and n-rows, the 3 columns being 'package ID', 'supply chute' and 'drop chute'.

We were also given an excel sheet with a list of package IDs, supply and drop chutes.

The task required us to iterate through the list, and transport cubes of measurement 20mmx20mmx20mm from supply to drop chutes.

We were allowed to use as many bots as we wanted.

Sample Sheet

Package ID	Induct Station	Destination
FKMP0001	1	Destination 1
FKMP0002	2	Destination 2
FKMP0003	3	Destination 3
FKMP0004	4	Destination 4

There would be a random distribution of

- Packages to destinations
- Packages to induct stations

Conditions:

One of the main differentiating factors in this task was that we were NOT allowed to put any sensors on our bot. The bot had to be blind and deaf, and only receive instructions via a Bluetooth or Wi-Fi module. This made the task challenging because obstacle detection and line following both heavily depend on the bot having a sensor mounted.

Also, sensors are much faster responders to stimuli than Wi-Fi/Bluetooth modules. Using a Wi-Fi module brought delays in response time and that reduced the maximum speed that the bots could travel at.

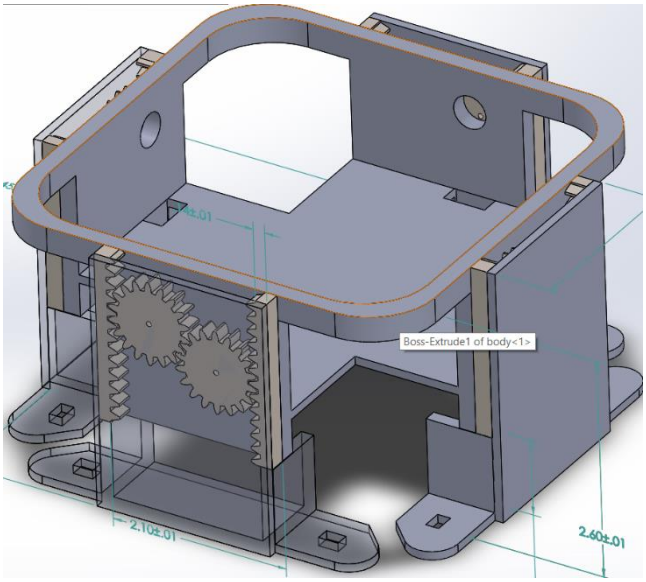
Preparing the mechanical design:

First it was necessary to conclude whether it was in my ability to even create this project. I had a roadmap in mind but I had never implemented it before and hence didn't know about what could go wrong.

I started with watching videos, reading papers and doing research on warehouse robots. I came across many designs which intrigued and settled on a (as I realized later) very inefficient and cumbersome design for the

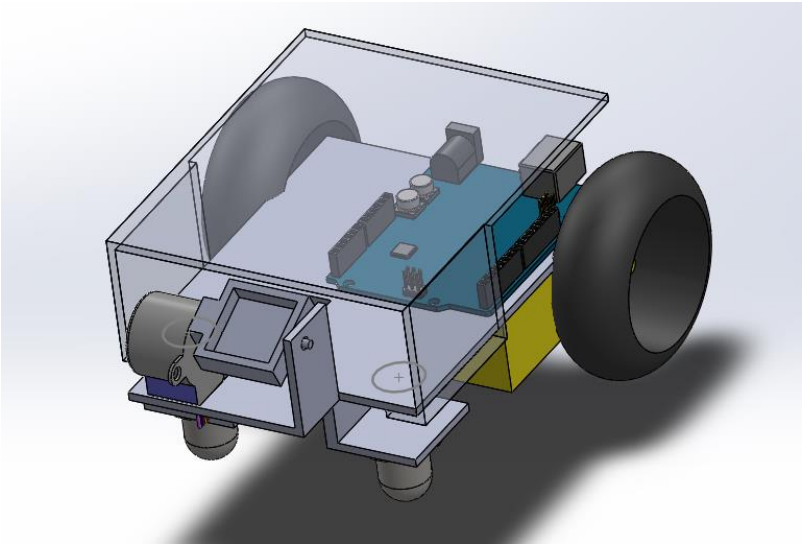
bot. As I created the SOLIDWORKS model of the same I realized how difficult this would be. My model had 8 gears, more than 16 3-D printed chassis parts, 12 motors, 8 wheels and this would take at least 2 arduinos and 6 motor drivers to run. Also, the model was slightly larger than the 6x6inch limit on each bot.

My first model was an imitation of the bot shown in this warehouse robotics video: https://www.youtube.com/watch?v=4DKrcpa8Z_E.



The plan was that the robot would have 8 wheels, 2 on each of the 4 sides of the bot. The walls would be movable up and down, controlled by stepper motors. Now if the bot wanted to move straight, the walls perpendicular to that direction would lift up and then the 2 remaining walls would run the bot. whenever the bot wanted to turn (90 degrees only), the walls would switch. Using this engage-disengage mechanism I planned to move the bot in 4 directions (using forward and backward motion of motors).

This mechanism, however would not work. I soon realized that I needed a much simpler bot, that would be easy to build, code, setup and run. This bot was none of those things.



I once again took to researching other bot designs. I was instantly attracted to the simplicity of Line Following Robots. They used only 2 DC motors, could turn in any direction, required no more than 2 3d printed parts to build. I also saw a connection between how Line Following Robots were coded and how I could make my bot follow a virtual path using python and OpenCV.

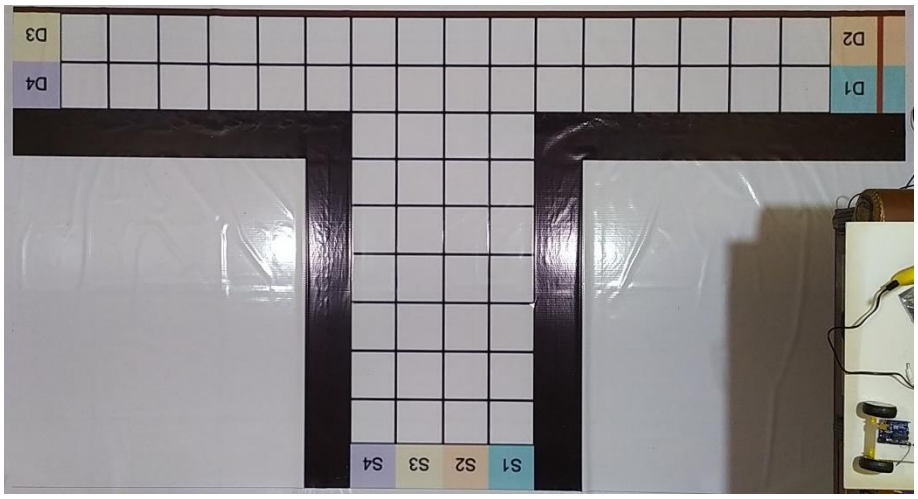
Inspired by such a design, I got to work. My design this time around was much better. It used only 3 motors, 2 wheels, 3 3d printed parts and I could turn/move in diagonal paths too.

Configuring OpenCV:

After I set up my camera at the top of the arena, I encountered quite some problems, the first of which was that I couldn't set up the map for my arena consistently. This was because I had to recharge my camera from time to time, and remounting the camera would introduce slight changes in the pixel values. Because of sunlight, the lighting in my room also was constantly changing.

I took a plain printout of the arena and placed it on the floor. (if you have noticed the arena is slightly changed. That's because this is the STAGE 1 arena tailor made so that there was no possibility of bots colliding. After clearing STAGE 1participants would qualify to the full-fledged stage 2 arena where bots would have to move together and not collide.)

With my camera mounted on top of the ceiling I took this snapshot of the arena.



From here (using python and OpenCV) I tried to make a map using only the black pixels in the image. To create contours, and then using the ids and centers of those contours as grids for my map. That didn't work for a multitude of reasons:

Firstly, my light reflects off the map resulting in awkward borders. Secondly the shadow cast by my table was messing the algorithm up. Also, if the camera angle/lighting changed even one bit the map structure could change and things could go wrong. And my arena was prone to tiny folds/crevices as seen in the picture, which already messed up the contour algorithm enough. The algorithm was noticing too many contours and it was very difficult to accurately point out the right ones.

As an alternative, I took the original track image, made contours on it and extracted coordinates of the centers in an array. After that all I had to do was mark those coordinates on my new map and I would be done. This would also help align my arena to the map I made and since my arena was entirely virtual it didn't depend on tiny external factors that couldn't be controlled.

This is the final code I used to scan the arena and get the map.

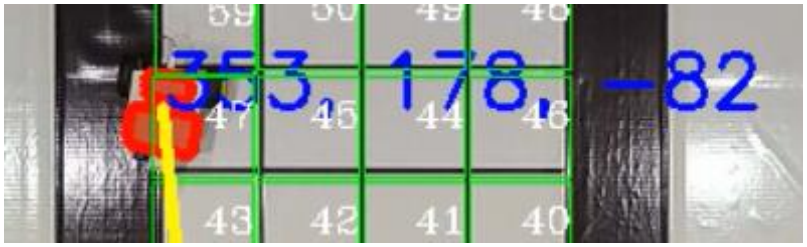
```
image2 = cv2.imread("map.png")
all_cells = [11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27,
28, 29, 30, 31, 32, 33, 34, 35, 36, 37,
             38, 39, 40, 41, 42, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63,
64, 65, 66, 67, 68, 69, 71, 72, 73, 74,
             75, 76, 77, 78, 79]
gray = cv2.cvtColor(image2, cv2.COLOR_BGR2GRAY)
thresh = cv2.adaptiveThreshold(gray, 255, 1, 1, 11, 2)
contours, _ = cv2.findContours(thresh, cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)
c = 0
viablecontours = []
for i in contours:
    area = cv2.contourArea(i)
    if area > 1000:
        viablecontours.append(c)
        #cv2.drawContours(image2, contours, c, (0, 255, 0), 3)
    c += 1
print(viablecontours)
for i in all_cells:
    cv2.drawContours(image2, contours, i, (0, 255, 0), 3)
cv2.imshow(str(22), image2)
cv2.waitKey(0)
```

After that I had to scan the position of the bot. I had decided to give all the bots a distinct color and then filter those colors out in my image. This is how the red was filtered out.

I had to set the hsv-ranges of 4 different colors by trial and error. Once I was done, I could consistently detect 4 colors on my arena, Red, Blue, Orange and Violet.

```
def update_color_range(self):
    if self.color == 'red':
        self.color_range = [[0, 150, 150], [6, 255, 230]]
    elif self.color == 'blue':
        self.color_range = [[50, 150, 100], [100, 250, 250]]
    elif self.color == 'orange':
        self.color_range = [[10, 150, 150], [20, 250, 250]]
    elif self.color == 'violet':
        self.color_range = [[140, 50, 100], [165, 190, 250]]
    else:
        print("WRONG COLOR")
```

And this is where new problems arose. The marker on my bot was symmetrical, so the camera never knew if the bot was facing in the direction it was supposed to, or in a direction that was the exact opposite of that. And I needed 3 more colors to run the algorithm on before I would be done.



So, I decided to place 2 different sized stickers on the bot, calculate each of their centers and then calculate the midpoint between both of those to get the position of the bot. The orientation of the bot would simply be the angle of the line that the two centers formed.

```
def find_bot(self, video_frame):
    hsv = cv2.cvtColor(video_frame, cv2.COLOR_BGR2HSV)

    lower_red = np.array([20, 220, 100])
    upper_red = np.array([30, 250, 250])

    mask = cv2.inRange(hsv, lower_red, upper_red)
    res = cv2.bitwise_and(video_frame, video_frame, mask=mask)
    # grayscale
    gray = cv2.cvtColor(res, cv2.COLOR_BGR2GRAY)
    # make thresh
    thresh = cv2.adaptiveThreshold(gray, 225, 1, 1, 19, 2)
    # find all contours
    contours, _ = cv2.findContours(thresh, cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)
    contours = sorted(contours, key=lambda i: cv2.contourArea(i), reverse=True)
    areas = []
    for i in contours:
        areas.append(cv2.contourArea(i))
    marker_locations = []
    for i in [contours[0], contours[1]]:
        (x, y), (_, _), angle = cv2.minAreaRect(i)
        marker_locations.append([x, y, angle])

    x = (marker_locations[0][0] + marker_locations[1][0]) / 2
    y = (marker_locations[0][1] + marker_locations[1][1]) / 2
    angle = find_line_angle(marker_locations[1], marker_locations[0])
```

```
    cv2.drawContours(video_frame, contours, 1, (0, 0, 255), 3)
    cv2.drawContours(video_frame, contours, 2, (0, 0, 255), 3)
    text_to_show = str(int(x)) + ', ' + str(int(y)) + ', ' + str(int(angle))
    cv2.putText(video_frame, text_to_show, (int(x), int(y)),
cv2.FONT_HERSHEY_SIMPLEX, 1, (255, 0, 0), 2)
    self.position = [int(x), int(y)]
    self.angle = int(angle)

    return self.position, self.angle
```

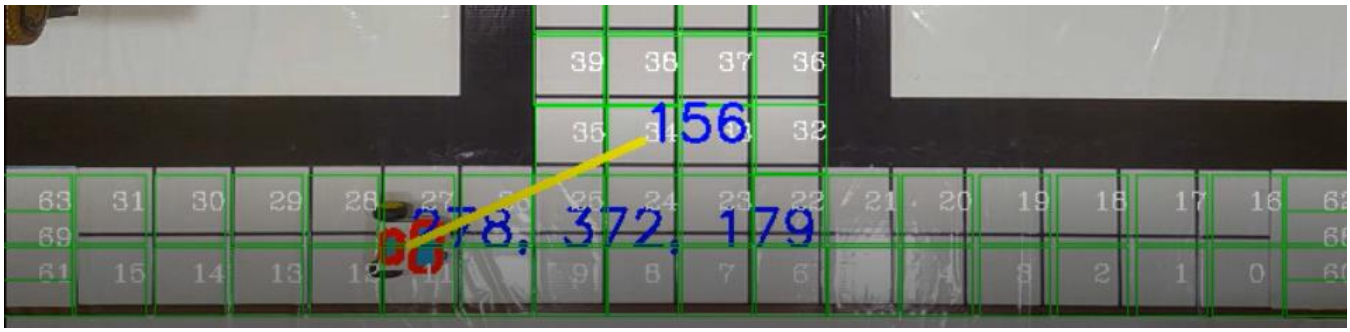
But simply calculating the slope of the line wouldn't solve the problem, since the slope is of the line itself, is the same for angles differing by 180 degrees. Had run into the same problem as I had before, but in a different way. Also, remembering the contours by their default order wouldn't work since OpenCV sorts contours by right-most and top-most first. So, when my bot was upside down a different contour would be first.

So, I took both of my bots contours (the two markers on top of my bot) sorted them according to size, then found out which one was higher (had a lower y-coordinate), and then calculated the if the bot was facing upwards or downwards. If my bot was facing upwards the angle wouldn't change, but if it was facing downwards then I would change it to an appropriate value.

I then took this code out and just made it a different function so that I could find the exact angle between any 2 points.

```
def find_line_angle(a, b):
    line_equation = find_line_equation(a, b)
    line_angle = int(np.degrees(np.arctan(line_equation[0])))
    if b[1] > a[1]:
        # line pointing below
        if line_angle > 0:
            line_angle = line_angle - 180
    else:
        # line pointing upwards or right/left
        if line_angle < 0:
            line_angle = 180 + line_angle
    return line_angle
```

So now I could find the position of all the grid-units of my map, the angle between any two points, the angle of my bot and the position of my bot, I was ready.



Communicating with the bot:

I had intended to use a NodeMCU to communicate with my processing unit to give instructions. And this worked well at first. Response time was around 1 second, which was very slow in itself. But things deteriorated quickly. On day-2 I saw responses as slow as 6 seconds and as day-2 ended not a single instruction would be heard in less than 5 seconds. This response time was incredibly slow and the project would not be possible under conditions, so I went to research other ways of communicating. I knew it was possible because we constantly play video games where we communicate with the server in milliseconds. Even with my laptop and a tiny NodeMCU I should have response times as low as 500 milliseconds at-least.

```
#include <ESP8266WiFi.h>
#include<SoftwareSerial.h>

//define the client and server
WiFiClient client;
WiFiServer server(80);
//define the communicator to arduino
SoftwareSerial ESP8266Comms(D2, D3);

void setup()
  {Serial.begin(9600); //begin serial monitor
  ESP8266Comms.begin(9600); //begin communicator
  pinMode(D1, INPUT);
  pinMode(D2, OUTPUT);
  WiFi.begin("7star", "00000000"); //connect to wifi
  while (WiFi.status() != WL_CONNECTED) //if connected to wifi move further
    {delay(500);
     Serial.print("*");}
  Serial.println();
  Serial.println("WiFi connection Successful");
  Serial.print("The IP Address of ESP8266 Module is: ");
  Serial.println(WiFi.localIP());
  server.begin();} //start server

void loop()
  {client = server.available(); //if a request is made, this turns true
  if (client == 1)
    Serial.println("server avail");
    {String request = client.readStringUntil('\n');
    request.trim();
    Serial.println(request);
    ESP8266Comms.println(request);}} //send data to arduino
```

I soon figured out that I was using the TCP-IP protocol which was just a bad way of communicating in this project. It was secure (which was not needed for me), checked if all the data was sent (again not necessary), waited for the bot to confirm if it had received the info (not necessary), and very slow (speed was crucial).

I quickly shifted to the Udp protocol and once I had configured everything the bot's response time dropped from 5 seconds on average, to a few milliseconds.

```
#include <ESP8266WiFi.h>
#include<SoftwareSerial.h>
#include<WiFiUDP.h>

//define the client and server
WiFiClient client;
WiFiServer server(80);
SoftwareSerial ESP8266Comms(D2, D3); //define the communicator to arduino

WiFiUDP Udp;
unsigned int localUdpPort = 4210;
char incomingPacket[256];
char replyPacket[256];

void setup()
  {Serial.begin(9600); //begin serial monitor
  ESP8266Comms.begin(9600); //begin communicator
  pinMode(D2, INPUT);
  pinMode(D3, OUTPUT);
  WiFi.begin("7star", "00000000"); //connect to wifi
  while (WiFi.status() != WL_CONNECTED) //if connected to wifi move further
    {delay(500);
     Serial.print("*");}
  Serial.println();
  Serial.println("WiFi connection Successful");
  Serial.print("The IP Address of ESP8266 Module is: ");
  Serial.println(WiFi.localIP());
  Udp.begin(localUdpPort);} //start server

void loop()
  {
  int packetSize = Udp.parsePacket();
  if (packetSize){
    Serial.printf("received %d bytes from %s port %d\n", packetSize,
Udp.remoteIP().toString().c_str(), Udp.remotePort());
    int len = Udp.read(incomingPacket, 255);
    if(len>0){
      incomingPacket[len] = '\0';
    }
```

```
    Serial.printf("UDP Packet contains %s\n", incomingPacket);
    ESP8266Comms.print(incomingPacket);
  }
  } //send data to arduino
```

Making the bot transport the package to the destination:

I had all the cells on the map as ID-coordinate pairs, the position and angle of my bot, and I could now communicate with my bot.

```
def send_command(self, arr):
    byte_message = bytes(str(arr), " utf-8")
    opened_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    opened_socket.sendto(byte_message, (str(self.ip), 4210))
```

Now all I had to do was write an algorithm that would run the bot to its required destination automatically, make the bot drop the package, and have it return to the supply cell for a refill.

I started with giving the bot an array of cells to follow. Something like [9, 25, 56, 25, 9]. This meant the bot would travel to cell number 9, then 25, then 56 and so on.

I coded a "bot" class to handle finding the bots position/coordinates, move the bot, stop the bot, etc.

```
class bot:
    position = [0, 0]
    angle: int = 0
    destination = []
    journey = [58, 25, 31, 25, 58]
    path_equations = [[]]
    next_target = 0
    next_target_angle = 0
    ip = "192.168.15."
    speed = 0
    delay_before = 0.15
    delay_after = 0.15
    turn_speed = 0
    turn_delay_before = 0.15
    turn_delay_after = 0.15
    color_range = []
    color = " "
    ejected = False

def __init__(self, ip, color, journey):
    self.ip = self.ip+str(ip)
    print(self.ip)
    self.color = color
    self.journey = journey
```

I followed a simple approach: I would calculate the angle of the path to follow from the bot's current position to the target position. I already have the bot's angle and using these two I can calculate whether to turn right

or left. Now when the bot comes within a 40-degree range of the target angle, it returns “True” and then I just tell it to move straight.

```
def align(self):
    b = self.angle
    p = self.next_target_angle
    if angle_equality(self.angle, self.next_target_angle):
        print(self.angle)
        print(self.next_target_angle)
        self.stop()
        return True
    else:
        if p > 0:
            if b > 0:
                if b > p:
                    self.turn(0)
                else:
                    self.turn(1)
            else:
                if b > -(180 - p):
                    self.turn(1)
                else:
                    self.turn(0)
        else:
            if b > 0:
                if b > 180 + p:
                    self.turn(1)
                else:
                    self.turn(0)
            else:
                if b > p:
                    self.turn(0)
                else:
                    self.turn(1)
    return False
```

I had a “stop” function for emergency stops and to stop the bot when it had finished its path.

```
def stop(self):
    arr = ['z', 0, 0]
    self.send_command(arr)
```

When the bot came in range of 40 pixels of the required target position then I would update the target position. If the target position was a drop point, I would drop the package. If the target was the last point in the path of the bot, I would start the next bot.

```
def find_next_target(self, video_frame):
    next_target_coords = all_centers[self.journey[self.next_target]]
    cv2.line(video_frame, tuple(self.position), tuple(next_target_coords), (0, 255,
255), 3)
    if position_equality(self.position, next_target_coords):
        self.next_target += 1

def find_next_angle(self, video_frame):
```

```
    next_target_coords = all_centers[self.journey[self.next_target]]
    self.next_target_angle = find_line_angle(self.position, next_target_coords)
    cv2.putText(video_frame, str(self.next_target_angle),
tuple(next_target_coords), cv2.FONT_HERSHEY_SIMPLEX, 1,
                (255, 0, 0), 2)

def find_path(self, video_frame):
    # find next target and angle
    self.find_next_target(video_frame)
    self.find_next_angle(video_frame)
```

It took a while to troubleshoot and synchronize the camera, the bot-speed and the response time to figure the optimum speed for the bot to travel at. A couple of trial-and-error runs solved that.

I didn't have the bot run continuously. Instead, I had it move in increments. Each time the bot moved straight, it would move for 1 second and then stop for 0.25 seconds to allow my algorithm to calculate the next move of the bot.

```
def go(self):
    self.p_control_speed()
    arr = ['z', self.speed, -self.speed]
    self.send_command(arr)
    time.sleep(self.delay_before)
    self.stop()
    time.sleep(self.delay_after)
    print("go")
```

In this time, I would calculate whether the bot should keep going straight (execute the "go" function), or turn to align itself better to the target (execute the "align" function which called the "turn" function after deciding to turn right or left). The turn function had a slightly larger pause and a slightly shorter run time since I needed turns to be more accurate.

```
def turn(self, direction):
    arr = ['z', 0, 0]
    self.p_control_angle()
    if direction:
        # turn right
        arr[2] = -self.turn_speed
        arr[1] = -self.turn_speed
        self.send_command(arr)
        time.sleep(self.turn_delay_before)
        self.stop()
        time.sleep(self.turn_delay_after)
        print("right")
    else:
        # turn left
        arr[2] = self.turn_speed
        arr[1] = self.turn_speed
        self.send_command(arr)
        time.sleep(self.turn_delay_before)
        self.stop()
        time.sleep(self.turn_delay_after)
        print("left")
```

I even coded some potential-control to my bot. This essentially meant that if my bot was really close to the target position or angle, it would go slower. It helped with the accuracy and my bot didn't overshoot or overturn half as much anymore.

```
def p_control_speed(self):
    next_target_coords = all_centers[self.journey[self.next_target]]
    if distance_between(self.position, next_target_coords) > 50:
        self.speed = 190
        self.delay_before = 0.2
        self.delay_after = 0.2
    elif distance_between(self.position, next_target_coords) <= 50:
        self.speed = 190
        self.delay_before = 0.15
        self.delay_after = 0.3

def p_control_angle(self):
    if angle_between(self.angle, self.next_target_angle) > 60:
        self.turn_speed = 180
        self.turn_delay_before = 0.1
        self.turn_delay_after = 0.2
    elif angle_between(self.angle, self.next_target_angle) <= 60:
        self.turn_speed = 180
        self.turn_delay_before = 0.1
        self.turn_delay_after = 0.3
```

And the eject function, that just dropped the package when called.

```
def eject(self):
    arr = ['e', 0, 0]
    self.send_command(arr)
    time.sleep(1)
```

Final optimizations:

When I added the delays to my bot running, a faced a dead end. The program would not display a new frame until it was done looping through all the code. And since I had delays as high as 1 second on my bot, my video feed frame-rate had dropped lower than 1 fps.

This in turn also affected my bot's speed. Since my code wasn't refreshing as fast my bot made more errors and went off-track too often.

I solved this by separating the section of code that sent commands to the bot from the section of code that ran the video feed and running them both parallel. I used the Threading module in python for this.

```
if __name__ == '__main__':
    a = Thread(target=run_video)
    b = Thread(target=run_bot)
    a.start()
    b.start()
```

Conclusion:

And hence my project was finally complete. I ran a few runs and although some of them failed I managed to succeed too. I recorded the runs and uploaded the video files as requested by the competition. I am happy to state that as I am writing this project report the results have come through and I have been promoted to round 2. I managed to transport 4 packages of 20x20x20mm through a distance of 7-8 ft each accurately and automatically.

This project has been a huge learning opportunity for me and I am grateful to the organizers for such an innovative problem statement.