Univerza *v Ljubljani*
Fakulteta *za strojništvo*

Santiago Leban Parada

Development of a laser tracking system for tracking a selected object

Mechatronic Actuators Project

Ljubljana, 2026

Contents

1. Introduction

For my project in the Mechatronic Actuators course, I chose the proposed topic of laser tracking system using computer vision. This is a field that interests me and about which I knew relatively little so it was an ideal opportunity to learn more about the subject.

1.1 How Computer Vision Works

Computer vision allows a computer to get and analyze information from digital images or videos. The computer treats the captured image as a matrix of pixels, where each pixel contains color and brightness data. Using image processing and pattern recognition algorithms, it extracts object feature and compares them with predefined models or rules. Based on this, it can recognize objects, determine their position, or track their movement.

1.2 Practical Applications

Computer vision is used in many fields. The most familiar to us is probably its use on smartphones for facial unlocking. It is also widely used in industry for product quality control, in medicine for analyzing X-ray images and assisting in disease diagnosis, and on security cameras for facial recognition and detecting suspicious behavior. In agriculture, it is used to monitor plant health and detect pests. It is also integrated into newer cars for road recognition and autonomous driving, and it is increasingly used in stores at self-checkout lines for inventory tracking and customer behavior analysis.

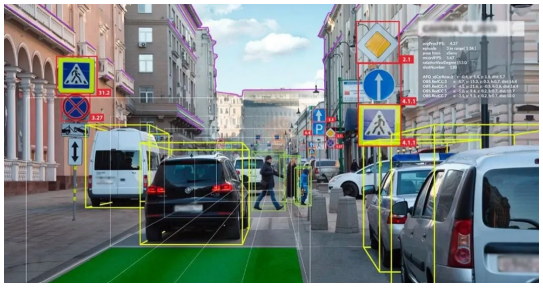


Figure 1.1: Autonomous driving

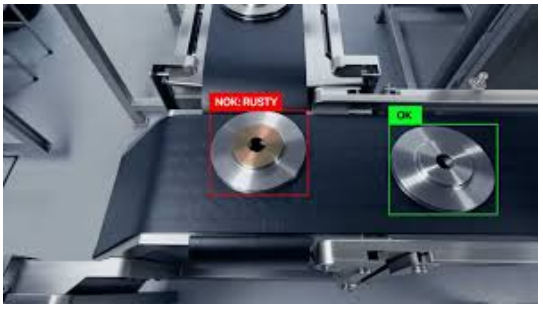


Figure 1.2: Computer vision in industry

2. Problem definition

I had to create a system where a laser pointer of a red color tracks a moving object. I chose a green laser dot as the moving object. The goal of the system is that the red dot tracks the green one, or for the red dot to overlap with the green one if it is stationary.

I devided the system into three parts:

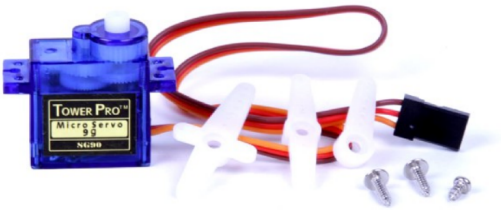
- Physical movment of the laser pointer
- Image capture
- Image analysis

3. System Components

3.1 Physical movment of the laser pointer

3.1.1 SG90 Servomotors

To move the laser, I used two SG90 servo motors – one for movement in the X direction and the other for the Y direction. The SG90 motor is lightweight and quite small in size, which is ideal for my project. It operates on a voltage of 4.8 to 6V, making it suitable for direct connection to the Arduino. It has 2.5 kg·cm of torque, which is quite low, but entirely sufficient for the needs of my system. The most significant limitation of this motor is its refresh frequency, which is 50 Hz. Since the system tracks the object's movement in real time, it is important how fast can the motor react to a change of a laser dot position. With a frequency of 50 Hz, the motor can receive commands for a new move every 20 ms, which is sufficient for solid tracking. If the object would move very fast, a higher frequency would be needed. Another major limitation is accuracy. The motor can move within a 180-degree range, but because the gears are plastic and there is some space between them, the position control accuracy is not very good, though it is acceptable within the scope of my project.



3.1.2 Elegoo UNO R3

To control the servomotors, I used an Elegoo UNO R3 development board, which is based on the ATmega328P microcontroller. This development board is a cheaper version of the Arduino UNO R3, sharing the exact same specifications. Programming the microcontroller is easily done via the Arduino IDE platform.

Figure 3.1: Elegoo UNO R3

3.1.3 Power-supply module

For the stable operation of the servomotors, I connected them to an external power supply module. This module provides a constant voltage of 5V to the servomotors. If I had connected the servomotors directly to the Arduino, a sudden change in current due to a fast movement would cause a voltage drop. This would lead to unstable operation on the servomotors or even reset the Arduino.

Figure 3.2: Power supply module

3.1.4 Wiring of Components

I mounted the servomotors on a L-profile that I made from two wooden sticks with a rectangular cross-section. I glued the sticks together, made a through-hole and mounted a servomotor inside. Each servomotor has wires for power, GND, and a PWM signal. I performed the wiring as follows:

- The power and GND of each motor were connected to the external power-supply module.
- The PWM signal wires were connected to Arduino pins 9 and 10.
- Finally, I connected the GND of the Arduino and the GND of the power supply module together to establish a common ground.

The entire L-profile with the motors was then attached to the top of a cardboard box, and the remaining components were placed inside to keep the system organized.

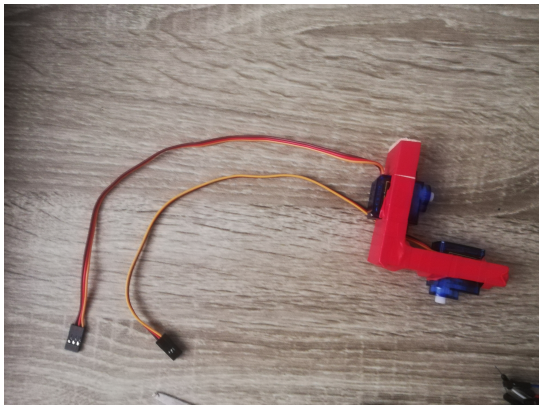


Figure 3.3: L-profile

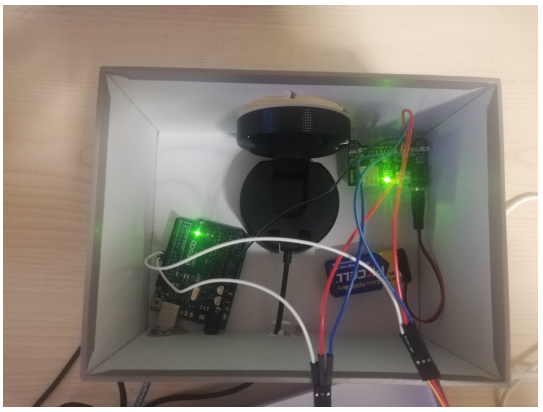


Figure 3.4: Wiring

3.2 Image Capture

To capture images, I used a Razer KIYO USB webcam, which I also placed inside the box alongside the other components. I cut a hole in the box for its lens, allowing it to capture images. Its maximum resolution is 1920×1080 at 30 frames per second (FPS). At resolution of 1280x720, the camera can even output 60 FPS. For this project, high resolution is not required since we are only tracking a simple green laser dot. If the resolution is higher, the processor must examine many more pixels during analysis, which is completely unnecessary in this case. Therefore, I ultimately chose a resolution of 640x480. It is also important to realize that at a higher resolution, the system would be limited by the SG90 servomotors, which are not that precise, making overly precise dot location data pointless. When choosing the refresh rate (FPS), I found that 60 frames per second was unsuitable. At 60 FPS, the Arduino would receive a new image and send a new signal to the servomotor every 16.6 ms, the problem is that the servomotor can only accept a new signal every 20 ms. Therefore, I chose 30 FPS. At this speed a new image is send every 33.3 ms, which aligns nicely with the refresh frequency of the servomotors.

Figure 3.5: Rezer Kiyo camera

3.3 Image Processing

Image analysis is performed by a laptop using Python code and the OpenCV (Open Source Computer Vision Library) library. The computer sees the image from the camera as a matrix of pixels, where each pixel in a color image has 3 values for RGB (red, green, blue). The OpenCV library imported into Python contains ready-made algorithms for processing these numbers. I wrote the entire image analysis code in Python because it allows easy programming and has excellent support for OpenCV.

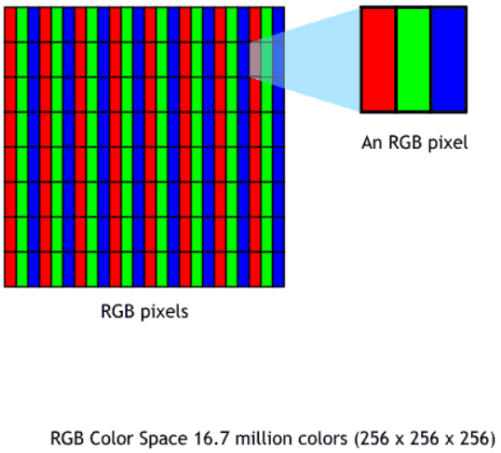


Figure 3.6: RGB

4. System Operation

4.1 Short description of the working principle

The Python program receives the image from the camera and begins analyzing where the green and red dots are located. Once found, it calculates the center of each dot and the distances ex and ey between the centers. The ex and ey data are sent directly to the Arduino via a serial communication.Then the Arduino moves the servomotors appropriately according to the received data. This cycle repeats for every frame every 30ms.

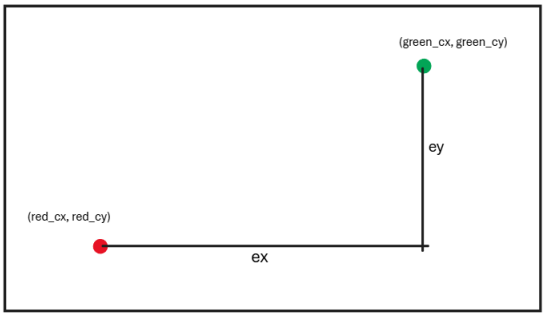


Figure 4.1: Example of an image captured by the camera

4.2 Python code

First, I connected the camera and Python using an OpenCV command. The entire processing then took place inside a while loop. Using the read() command, I converted the image into a matrix of numbers understood by Python. I then converted the image to the HSV (Hue, Saturation, Value) color space, which is more resistant to lighting changes than RGB. This was followed by programming a binary mask, meaning I had to determine which pixels I wanted to detect and convert them to white, while discarding the rest. This created a black-and-white mask where the tracked object is colored white. To track the green laser dot, I defined color thresholds corresponding to a green laser, and additionally, I wrote a mask in the program for very bright pixels (since a laser is very bright) and then combined both masks.

- Code for the green dot mask:

```
value_channel = hsv[:, :, 2]
_, mask_bright = cv2.threshold(value_channel, 220, 255, cv2.
    THRESH_BINARY)
lower_green = np.array([35, 80, 180])
upper_green = np.array([90, 255, 255])
mask_green = cv2.inRange(hsv, lower_green, upper_green)
mask_green = cv2.bitwise_and(mask_green, mask_bright)
```

Initially, I wanted to use only the green mask, extract the coordinates of the green dot from it, and send them to the Arduino, but this turned out to be very poor practice as tracking was highly inefficient. Therefore, I created a red mask as well.

- Code for the red dot mask:

```
lower_red1 = np.array([0, 50, 150])
upper_red1 = np.array([10, 255, 255])
lower_red2 = np.array([170, 50, 150])
upper_red2 = np.array([180, 255, 255])
mask_red1 = cv2.inRange(hsv, lower_red1, upper_red1)
mask_red2 = cv2.inRange(hsv, lower_red2, upper_red2)
mask_red = cv2.bitwise_or(mask_red1, mask_red2)
```

From the masks, Python calculates the coordinates of the centers of the green dot (green_cx, green_cy) and the red dot (red_cx, red_cy) in pixels. From this data, it also calculates the current error (distance) between the dots, ex and ey. This is the information the Arduino needs to move the servomotors appropriately. However, before the program sends this data, it performs a smoothing operation. Due to the constant fluctuations of ex and ey, smoothing ensures that when moving to a new position, the laser also accounts the previous position and only moves by a certain fraction of the error. This makes the system much more stable and prevents stuttering.

- Error calculation and smoothing:

```
ex = red_cx - green_cx
ey = green_cy - red_cy

alpha = 0.3
ex = int(alpha * ex + (1 - alpha) * last_ex)
ey = int(alpha * ey + (1 - alpha) * last_ey)
last_ex, last_ey = ex, ey
```

4.3 Calibration

When I first set up the system, it worked very well; both dots were visible on the camera, and the red one tracked the green one nicely. However, since the red dot must not leave the camera's field of view for proper operation, I had to define the limit angles for the servomotors in the Arduino program. Initially, I determined these values experimentally and put them into the Arduino program. A problem arose when I changed the system's distance from the wall. Although the laser dot remains physically the same size, when the camera is close to the wall, its field of view is smaller. Consequently, the laser dot occupies many more pixels and it can escape from the field of view with even a minimal movement of the motor. The fixed limits in the Arduino thus became unsuitable. I solved this issue by adding an automatic calibration routine to the Python program before the tracking part. Upon every program startup, Python detects the red dot and calculates its center. The servomotors are set to an initial starting position, which is always the same: 90 degrees for the X axis and 110 degrees for the Y axis. This is because the camera and laser are aligned horizontally (X direction), but vertically (Y direction), the laser is physically slightly higher than the camera. By using a larger initial angle (110°), I ensured that the dot appears roughly in the center of the image. Although this offset varies when changing the distance from the wall, the initial settings provide a wide operating range. It is only necessary that the dot is visible in the image in the initial position.

Calibration is then executed in four consecutive while loops:

- Left of the starting position (X-motor)
- Right of the starting position (X-motor)
- Up from the starting position (Y-motor)
- Down from the starting position (Y-motor)

As long as the dot is visible in the image, Python sends commands to Arduino to move the servomotor by one step. When the dot disappears from the field of view, Python saves this last position as a limit, calibration for that direction ends, and the motor returns to the initial starting point. The process is then repeated for the remaining directions.
This movement is managed by the following function in Python:

```
    def scan_limit(command, start, stop, step, label):
    servo = start
    last = servo
    lost_count = 0
    while (step > 0 and servo <= stop) or (step < 0 and servo >= stop):
        arduino.write(f"{command},{servo}\n".encode())
        time.sleep(step_delay)
        ret, frame = cap.read()
        cx, cy, mask = get_red(frame)
        if show_frame(frame, cx, cy, mask):
            print("STOPPED BY USER")
            return last
        print(f"{label} = {servo}, cx = {cx}, cy = {cy}")
        if cx is not None and cy is not None:
            last = servo
            lost_count = 0
        else:
            lost_count += 1
            if lost_count >= 3:
                break
        servo += step
    return last
```

We call this function sequentially for each individual direction. Below is an example for the minimum X value:

```
min_x = scan_limit("SETX", 90, 0, -1, "servoXmin") + border
arduino.write(b"SETX,90\n")
time.sleep(pavza)
print("\n")
```

For safety reasons, I added or subtracted a safety margin of 10 degrees (border) to each retrieved boundary. The calibration results in Python are the final values MIN_X, MAX_X, MIN_Y, and MAX_Y. These values are then sent via serial communication to the Arduino, where they are stored in variables. After calibration, the motors cannot exceed these established limits.

4.4 Arduino program

The Arduino program from the Arduino IDE environment is uploaded to the microcontroller only once at the beginning. After that, all communication happens directly from Python. The Python program is entirely responsible for image processing, while the only task of the Arduino is to convert the received data into angles using proportional gain and send the appropriate PWM signal to the servomotors.

4.4.1 Receiving Data in Arduino

Inside the main loop void loop(), the microcontroller receives data character by character, assembles them, and stores them in the inputString string. When it encounters a newline character '\n', the string is finalized, and the data is passed to the processData function.

```
void loop(){
  while (Serial.available())
  {
    char c = Serial.read();
    if (c == '\n')
    {
      processData(inputString);
      inputString = "";
    }
    else
    {
      inputString += c;
      if (inputString.length() > 40)
      {
        inputString = "";
      }
    }
  }
}
```

4.4.2 Data processing

The processData function has four different scenarios – three are intended for calibration, and one is for tracking itself.
CALIBRATION: If Python sends a string starting with "SETX", code is executed to move the X servomotor by one step. This allows the motor to move during calibration as long as the dot is visible. The identical code is written for the Y axis (string "SETY"). When all edge values are determined, Python sends the string "LIMITS" and Arduino saves the received limit values into the variables MIN_X, MAX_X, MI_Y, and MAX_Y.

```
if (data.startsWith("SETX"))
{
  int v;
  sscanf(data.c_str(), "SETX,%d", &v);
  posX = v;
  servoX.write(posX);
  return;
}

if (data.startsWith("LIMITS")){
  int t_minX, t_maxX, t_minY, t_maxY;
  int parsed = sscanf(data.c_str(), "LIMITS,%d,%d,%d,%d", &t_minX, &t_maxX,
      &t_minY, &t_maxY);
  if (parsed == 4) {
    MIN_X = t_minX;
    MAX_X = t_maxX;
    MIN_Y = t_minY;
    MAX_Y = t_maxY;
  }
  return;
}
```

TRACKING: Once the calibration is completed, we proceed to the tracking section of the function. The program receives the ex and ey data, which represent the error between the position of the red and green dot in pixels (px). First, a dead zone is checked; if the error is extremely small, the program sets it to zero. Such minor discrepancies do not require correction, and acting on them would only cause unnecessary jitter in the system. A crucial part of the code is the proportional controller (P-controller). Since the errors are expressed in pixels and we control the servomotors in degrees, the error must be converted appropriately. This task is handled by the proportional factor Kp, which determines how many degrees the servomotor should move for a given pixel error. I determined the value of the Kp factor experimentally so that the system ensures stable and sufficiently fast tracking. We calculate the new position of the servomotor by adding the product of the error and the corresponding proportional gain to the current position. Before writing to the motors, the values are constrained within the calibrated limits using the constrain command, physically preventing the red laser from escaping the camera's field of view.

```
int commaIndex = data.indexOf(',');
if (commaIndex < 0)
  return;

float ex = data.substring(0, commaIndex).toFloat();
float ey = data.substring(commaIndex + 1).toFloat();

// mrtva cona
if (abs(ex) < 2)
  ex = 0;
if (abs(ey) < 2)
  ey = 0;

// proporcionalni regulator
posX += Kp * ex;
posY += Kp * ey;

// omejitev s kalibriranimi mejami
posX = constrain(posX, MIN_X, MAX_X);
posY = constrain(posY, MIN_Y, MAX_Y);

servoX.write((int)posX);
servoY.write((int)posY);
```

5. System Startup and Testing

As mentioned previously, I mounted all physical components of the system inside a cardboard box. To set up and start the system, the Arduino and webcam must first be connected to the computer via USB cables. Then, the Arduino program is uploaded to the microcontroller, and as a final step, the Python code run on the computer. During testing, I found that my system operates most optimally in moderately lit rooms, capturing images of a flat wall where there are no other objects. Lighting conditions significantly affect operational reliability:

- A room that is too dark causes glare from the lasers, which can confuse the image analysis algorithm. This becomes even more pronounced if there are other objects in the room from which the laser light reflects.
- A room that is too bright causes a drop in contrast, which may result in the system failing to detect the laser dots at all.
- Other red or green objects in the room also have a negative impact on operation, as there is a risk that the program will mistakenly recognize them as the target laser dots.

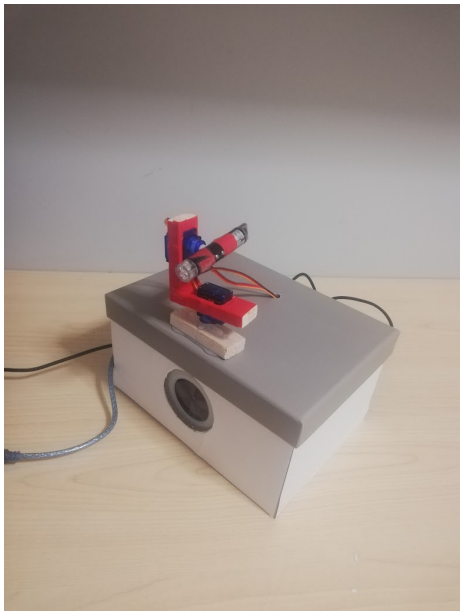


Figure 5.1: Product

Figure 5.2: Complete system

6. Possible Improvements

Despite the successful implementation, I still see plenty of opportunities to improve the system. I consider the unstable construction of the wooden L-profile to be the greatest weakness of my current system. During rapid movements of the servomotors, the entire mount vibrates heavily, introducing significant inaccuracy into the system. This problem could be resolved with a more rigid 3D-printed construction and by choosing advanced, more precise servomotors.

Another drawback that is disruptive during use is the wired connection of the components to the laptop via USB cables. This could be solved by using a wireless Bluetooth camera and installing a Bluetooth module on the Arduino. This would convert the system into a completely independent wireless unit. I am aware that for professional tracking, where both dots would perfectly overlap at all times, much more complex upgrades would be required. Nevertheless, the improvements mentioned above would excellently optimize its performance.

7. Conclusion

I am satisfied with the final outcome of the project work, as I succeeded in creating a functioning mechatronic laser tracking system. The project brought me a lot of new knowledge in the fields of computer vision, programming, and actuator control. Throughout the development process, I regularly encountered challenges and problems for which I had to independently find and implement appropriate solutions, which was an exceptionally beneficial experience for me.

8. Sources

1. http://archive.xaraxone.com/webxealot/workbook35/page_5.htm
2. https://www.csl-computer.com/en/razer-kiyo-webcam.html
3. http://www.ee.ic.ac.uk/pcheung/teaching/DE1_EE/stores/sg90_datasheet.pdf
4. https://epow0.org/~amki/car_kit/Datasheet/ELEGOO%20UNO%20R3%20Board.pdf