# 🤔Some basic idea when i implement this little project

📷Object detection

This project employs AprilTag as the target for tracking. AprilTag is a visual fiducial system widely utilized in various applications, including augmented reality (AR), robotics, and camera calibration. The AprilTag marker can be easily printed using a standard printer, and the AprilTag detection algorithm enables the precise computation of its 3D position, orientation, and ID relative to the camera. The AprilTag detection process primarily consists of three key steps:

1. **Edge Detection Based on Gradients**: The initial step involves detecting various edges within the image by analyzing gradient changes.
2. **Quadrilateral Identification and Filtering**: The second step focuses on identifying and filtering quadrilateral patterns from the edge-detected image. The AprilTag algorithm rigorously processes the detected edges by first eliminating non-linear edges, then performing an adjacency search among the linear edges. If a closed loop is formed, a quadrilateral is detected. The quadrilateral is subsequently decoded to identify the AprilTag marker.
3. **Determination of the Tracking Point**: The final step involves determining the center point of the quadrilateral, which serves as the 3D coordinate point to be tracked.

The OpenMV platform encapsulates the above steps into a convenient function, allowing users to locate AprilTag markers using the `img.find_apriltags()` function. Furthermore, the 3D coordinates and orientation of the detected AprilTag can be obtained through the function's return values: the x-axis coordinate can be retrieved using `tag.x_translation()`, while `tag.y_translation()` and `tag.z_translation()` provide the y- and z-axis coordinates, respectively, along with the tag's 3D orientation

⏭️ Two axes gimbal tracking control

To enable the gimbal to track the target and keep it centered in the camera's field of view, the OpenMV module provides the x and y coordinates of the target object's center through its target detection function. However, the coordinate system in OpenMV uses the bottom-left corner of the image as the origin, meaning that the obtained x and y coordinates are always positive. This makes it challenging to determine the direction of the target's deviation (left/right or up/down) relative to the image center, which is necessary for effective gimbal tracking. To address this issue, a simple coordinate transformation is applied. The OpenMV module provides functions to retrieve the image dimensions, namely the width (width) and height (height). Given the target's center

coordinates `(x, y)`, the relative position of the target with respect to the image center can be calculated using the following proportional approach:

$$y1 = y/height - 0.5$$
$$x1 = x/width - 0.5$$

With the relative offsets $x_1$and $y_1$ obtained as the latest values, their ranges are both $[-0.5, 0.5]$ To ensure the gimbal continuously tracks the target, these normalized coordinates must be converted into rotation angles for the servo motors of the 2-DOF gimbal, as illustrated in Figure 1.1. The lower servo controls the yaw angle, which corresponds to the camera's x-axis, while the upper servo controls the pitch angle, which corresponds to the camera's y-axis. The mechanical rotation range of the yaw servo is $[0^\circ, 180^\circ]$, where an angle of $0^\circ$ positions the servo to the right, and an angle of $180^\circ$ positions it to the left. The mechanical rotation range of the pitch servo is $[90^\circ, 180^\circ]$ , where an angle of $90^\circ$keeps the platform horizontal, and an angle of $180^\circ$ positions the platform perpendicular to the horizontal plane.

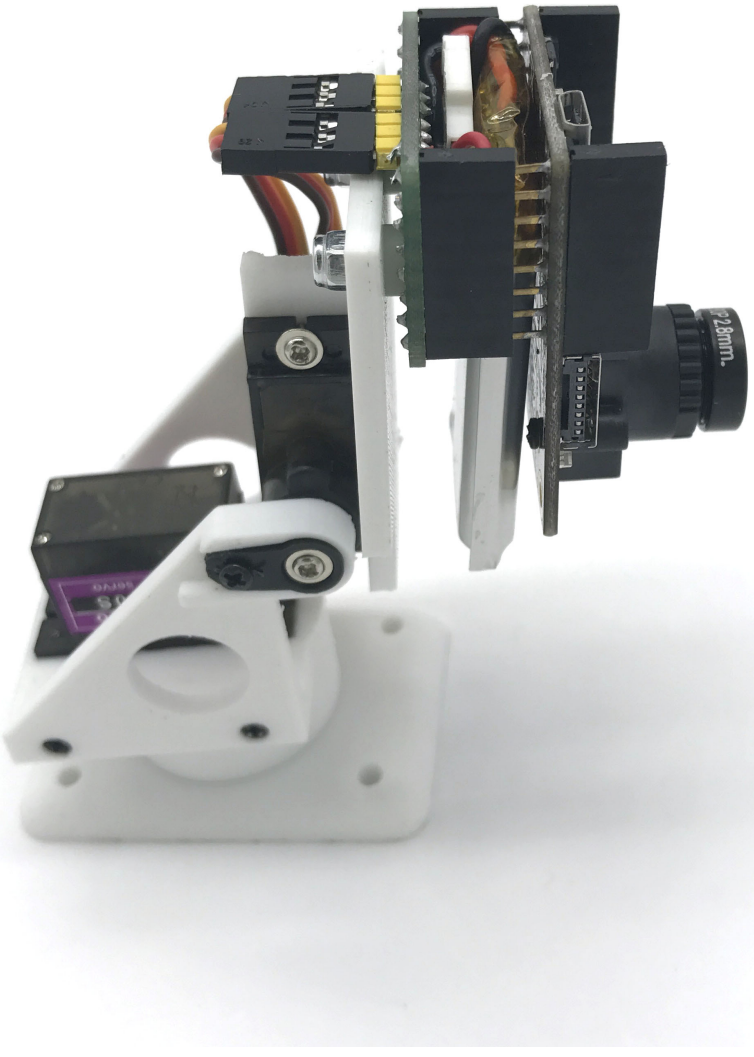


**Figure1.1**

To achieve smooth and stable gimbal tracking, a PD control algorithm is implemented, as PD control is well-suited for servo motors due to its stability. Given the current normalized coordinates of the target $(x_1, y_1)$, the objective is to move the target to the center of the camera's field of view, i.e., the target position $(0, 0)$. The error along the x-axis and y-axis are defined as `error_x` and `error_y`, respectively. Let the previous error be `error_x_last` and `error_y_last`, and the current servo angles be `yaw_now` and `pitchnow`. A simplified incremental PD control algorithm is applied as follows

$$output_x = Kp * （error_x） + Kd * (error_x - error_(x_{last}))$$
$$output_y = Kp * (error_y) + Kd * (error_y - error_(y_{last}))$$
$$yaw_{now} += output_x$$
$$pitch_{now} += output_y$$

## 📏 Straight-line distance to the target

The target selected for recognition in this project is an AprilTag. In the OpenMV ecosystem, the image.find_apriltags() function is utilized to detect AprilTags within the captured image. For detailed usage, refer to the OpenMV documentation: image — Machine Vision — MicroPython 1.9.2 Documentation. The specific attributes of the returned AprilTag object can be found in the documentation: image.apriltag — Machine Vision — MicroPython 1.9.2 Documentation.

For the purposes of this project, only the x, y, and z coordinates of the AprilTag relative to the camera's center are required, which correspond to indices 12, 13, and 14 of the returned object, respectively. The units of these coordinates are not explicitly defined in the documentation but are proportional to the actual distance from the camera to the AprilTag. This proportional relationship can be expressed using a scaling factor$K$, where the value of $K$ varies depending on the physical size of the AprilTag used

$$diatance = K * (\sqrt{(x^2 + y^2 + z^2)})$$

How to compute $K$. Since the OpenMV module employs a monocular camera, direct distance measurement through stereo vision is not feasible. Instead, distance estimation is achieved by utilizing a reference object, as shown in Figure 1.2

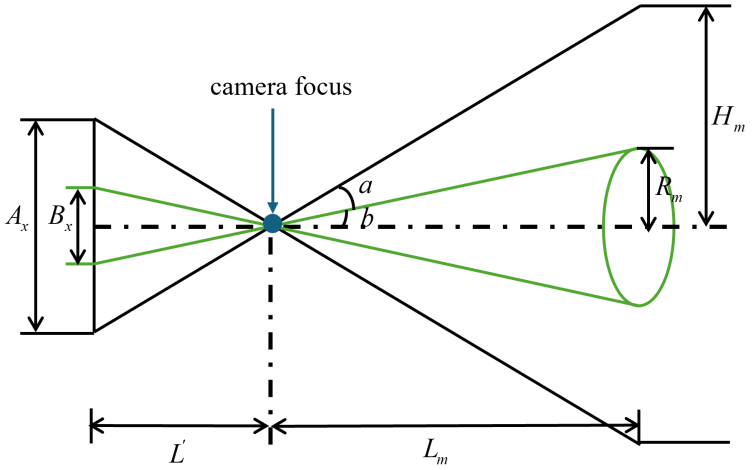


**Figure1.2**

$L_m$ : The distance from the object to the camera. : The

$L'$ focal length of the camera.

$B_x$: The diameter of the object in the image (in pixels).

$A_x$: The diameter of the image (in pixels).

$R_m$: The actual radius of the object (physical size).

$H_m$: The radius of the camera's field of view (physical size).

$$\tan a = \frac{A_x}{2L'} = \frac{H_m}{L_m}$$
$$\tan b = \frac{B_x}{2L'} = \frac{R_m}{L_m}$$
$$\frac{\tan a}{\tan b} = \frac{A_x}{B_x} = \frac{H_m}{R_m}$$

From expression of $\tan a$ and $\tan b$, we can get $L_m = \frac{2L'R_m}{B_x}$, $L' = \frac{A_x}{2\tan a}$, thus we have $L_m = \frac{2L'R_m}{B_x} = \frac{A_x R_m}{B_x \tan a}$, let $K = \frac{A_x R_m}{\tan a}$, the value of $A_x$, $R_m$ and $\tan a$ are constant, $L_m = \frac{K}{B_x}$so we can obtain $K$.

A method to determine the constant $K$ is as follows: First, position the object at a distance of $10cm$ from the camera and record the pixel value of the diameter as observed in the camera. Then, multiply the pixel value by the distance to obtain the value of $K$, the pixel value

We have obtained the straight-line distance to the target. However, due to the presence of a yaw angle, this distance is not necessarily the horizontal distance. By incorporating the servo motor angle previously utilized, we can calculate the horizontal distance and transmit it to the STM32,

Currently, the state of the vehicle can be obtained as shown in Figure 1.3 (a). The distance of the vehicle is decomposed, with Figure 1.3 (b) providing a schematic illustration of the distance decomposition. The principle is relatively straightforward: it can be observed that the distance should first be projected onto the horizontal plane, effectively performing a spatial projection onto the horizontal plane, thereby yielding the horizontal distance, denoted as $\mathrm{distance}_{\mathrm{horizontal}}$.

$$distance_{horizontal} = distance * cos(pitch)$$

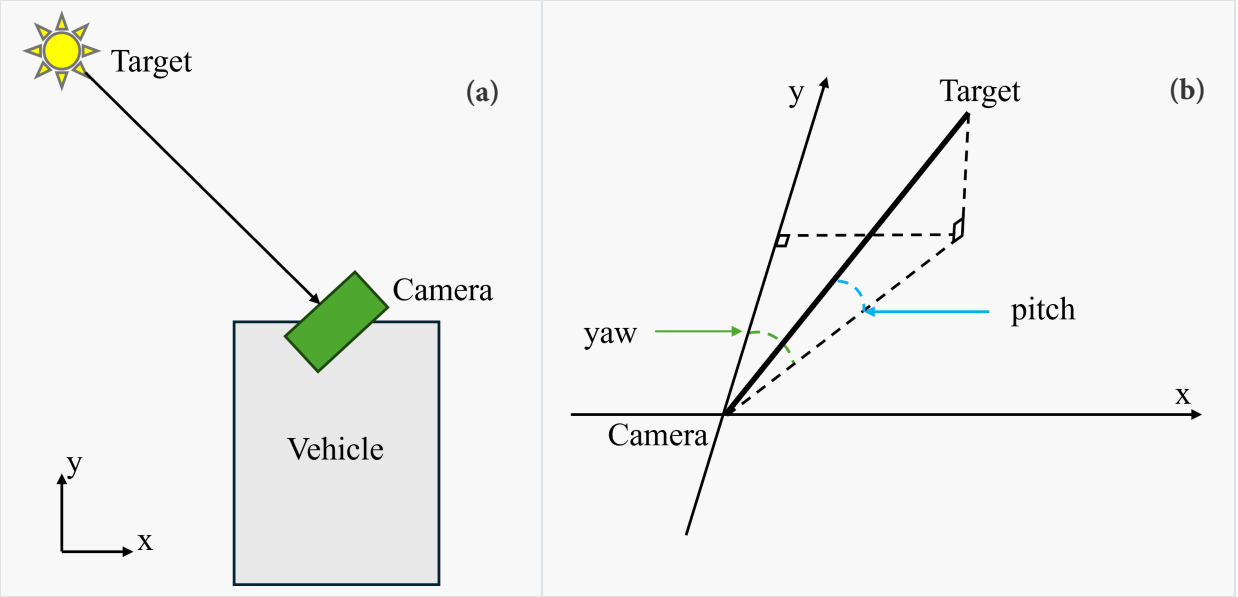


**Figure1.3**

Using the horizontal distance, we can determine the coordinates in the x x x- and y y y-directions, with the coordinates being

$$(distance_{horizontal} * sin(yaw), distance_{horizontal} * cos(yaw))$$

It should be noted that the yaw here is not $\mathrm{yaw}_{\mathrm{now}}$, and the pitch is not $\mathrm{pitch}_{\mathrm{now}}$ . Both $\mathrm{yaw}_{\mathrm{now}}$ and $\mathrm{pitch}_{\mathrm{now}}$ represent the current rotation angles of the servo motor. Therefore, we determine these based on the mechanical and geometric properties of the system (for further details, refer to **Two axes gimbal tracking control**).

$$pitch = 180 - pitch_{now}$$
$$yaw = yaw_{now} - 90$$
$$(x, y) = [distance * cos(pitch) * sin(yaw), distance * cos(pitch) * cos(yaw)]$$
$$= [-distance * cos(pitch_{now}) * cos(yaw_{now}), -distance * cos(pitch_{now}) * sin(yaw_{now})]$$

With this, the tasks of the OpenMV are essentially completed, and the obtained x and y represent the spatial coordinates of the object. Next, the information needs to be transmitted via the serial port.

# ℹ️ Transmission and reception of  serial port messages

## (1) Hardware connection

- For STM32. The USART2 of the STM32 is selected to connect with the USART3 of the OpenMV. Specifically, for the STM32's USART2, as shown in the figure below, PA2 serves as the signal transmission port, while PA3 serves as the reception port.

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| L2 | J2 | 16 | 25 | 36 | PA2 | I/O | | PA2 | USART2_TX(6)/ TIM5_CH3/ADC123_IN2/ TIM2_CH3 (6) |
| M2 | K2 | 17 | 26 | 37 | PA3 | I/O | | PA3 | USART2_RX(6)/ TIM5_CH4/ADC123_IN3 TIM2_CH4(6) |

- For OpenMV. In the figure 1.1, the OpenMV is equipped with an OV7725 camera, whereas the camera we purchased is an OV5640. For the USART3 of the OpenMV, it corresponds to the P4 and P5 pins, where the P4 pin serves as the transmission port, and the P5 pin serves as the reception port.
- Connection. The transmission port of the STM32 is connected to the reception port of the OpenMV, and the reception port of the STM32 is connected to the transmission port of the OpenMV. Specifically, the PA2 pin of the STM32 is connected to the P5 pin of the OpenMV, and the PA3 pin of the STM32 is connected to the P4 pin of the OpenMV.

## (2) Software implementation

- For OpenMV.

### Openmv send

Enable UART3 for serial transmission and set the baud rate to 115200`uart = UART(3, 115200)`. The `uart.write` function is a pre-implemented function in the UART library of OpenMV, which can be directly invoked. Here, `struct.pack` is used to pack the information, where each letter in pack corresponds to a specific data type:

| Format Letter | variable | Data Type | Byte Length |
|---|---|---|---|
| x | pad byte | no value | 1 |
| c | char | string of length 1 | 1 |
| b | signed char | integer | 1 |
| B | unsigned char | integer | 1 |
| ? | _Bool | bool | 1 |

| Format Letter | variable | Data Type | Byte Length |
|---|---|---|---|
| h | short | integer | 2 |
| H | unsigned short | integer | 2 |
| i | int | integer | 4 |
| I | unsigned int | integer or long | 4 |
| L (小写l) | long | integer | 4 |
| L | unsigned long | long | 4 |
| q | long long | long | 8 |
| Q | unsilong long | long | 8 |
| f | float | float | 4 |
| d | double | float | 8 |
| s | char[] | string | 1 |
| p | char[] | string | 1 |
| P | void * | long | |

Like `uart.write(struct.pack('<BBBhh', 0xFF, 0xF1, Bool_s, int(get_x), int(get_y)))`, the format string 'BBBhh' represents: byte + byte + byte + short integer + short integer.

- '0xFF, 0xF1': Two frame headers
- 'Bools': Indicates whether the target is detected
- 'get_x, get_y': Transmits the x x x- and y y y-coordinates of the target

**Stm32 receive**

At the STM32 end, upon receiving data, the system will enter an interrupt handler function. The interrupt handler function name in STM32 is fixed and must be used in accordance with its specified requirements. Since the USART2 serial port is utilized in this experiment, the corresponding interrupt handler function is:

```
void USART2_IRQHandler(void);
```

Each invocation of the interrupt handler receives one byte of data. This means that the received data needs to be stored. After a complete frame of data is acquired, the frame header is verified. In this experiment, the frame header is defined as 0xFF, 0xF1. Only after the frame header is identified can the remaining data be processed. The received data is stored as static variables to facilitate access and processing by external functions.

## Control  for vehicle

The obtained xxx and yyy coordinates are referenced to the camera of the vehicle as the origin. Therefore, to reach the target point, the current positional error of the vehicle in the xxx and yyy directions corresponds directly to the xxx and yyy coordinates. The output along the xxx-axis represents the incremental change in the PWM signal, relative to the baseline increment required to induce motion in the motor.

```
output_x=Kp_x*(error_x_now)+Kd_x*(error_x_last-error_x_now);
```

The output along the y-axis follows the same principle. Regarding the distance to the target, a certain tolerance for error should be allowed to prevent collisions. Within this permissible error range, the output should be adjusted to zero to ensure the vehicle comes to a stop as required.

If $\mathrm{output}_x$ is ultimately determined to be zero while $\mathrm{output}_y$ remains nonzero, the vehicle should move in a straight line forward (where the x-axis represents lateral movement and the y-axis represents forward motion).

If $\mathrm{output}_x$ is nonzero, the vehicle must undergo rotational adjustment. Continuous turning while moving may result in target loss due to speed constraints. Therefore, the vehicle should be controlled via differential drive to reduce $\mathrm{output}_x$ to zero, ensuring that it can continue moving forward in the correct direction.

## End

This concludes the overview of this small project. If you have any questions, feel free to leave a comment for discussion. Although this project is relatively simple, looking back after pursuing graduate studies, I realize that I have gained valuable insights from this demo while also recognizing its limitations. The time I spent away was dedicated to studying control theory, and I am now close to completing it. Moving forward, I will continue to work and learn in the field of robotics!