Localization System of Group A2

Wu Chengyu 7086cmd@gmail.com

February 4, 2024

Contents

1 Introduction

The map of the contest is a 2D plane with `apriltags` on it. The `apriltags` are asymmetrical and easy to recognize.

Through the camera, we can fetch all the `apriltags` in the view of the camera. Using `OpenCV`, we can get the position of the `apriltags` in the image. Then we can use the position of the `apriltags` to calculate the position of the camera.

The `ROS` (Robot Operating System) provides apis to get the position of the `apriltags` in the image. However, we also need the camera to recognize other blocks, e.g. the "fish" in the contest. So we deprecated the `ROS` and simply use `Python` with `OpenCV` to get the position of the `apriltags` in order to locate.

Also, we need to recognize blocks colored with red, yellow, green, and blue (size: 5cm $\times$ 5cm $\times$ 5cm). We should use `OpenCV` too, so it's a good manner to combine these two parts together.

That's the biggest reason why I strongly recommend navigating with `OpenCV`.

Becuase we don't use `ROS`, so we can test the camera with my laptop. The Apple 1080P camera can help me to test the algorithm. As for the calculation difference, it is not so big.

2 Targets

There are several targets in this project:

2.1 Calibration

- Get the camera matrix and the distortion coefficients with the help of `OpenCV`.
- Store the data, and use it at call time.
- Fetch the external parameters when the robot is on a apriltag and calculate the transform matrixes to get the external parameters.

2.2 Localization & Navigation

NOTE: Finally, this plan is deprecated.

- We need fetch the location of the robot (not only position, but also the orientation) in the ground coordinate system. The ground coordinate system is defined by the `apriltags`.
- We need to adjust the position, making it more accurate. The accuracy should be less than 1cm.
- We need to handle with the emergency situation. If there are less than 3 `apriltags` in the view of the camera, we should use corner of single & double `apriltags` to calculate the position of the camera.
- If there is no `apriltags` in the view of the camera, we should turn around and find the `apriltags`.

NOTE: The Current Plan

We use the `apriltags` with its 4 corners to calculate the position of the camera. We use the PnP algorithm to calculate the position of the camera.

So, in general, even if there is only one `apriltag` in the view of the camera, we can calculate the position of the camera.

2.3 Block Recognition

- We need to recognize the color of the block. The color of the block is red, yellow, green, and blue. If necessary, we need to recognize the orange block too.
- We should get the camera matrix, the distortion coefficients. However, it's not my task.
- We should also get the position of the block (only position) in the ground coordinate system.

2.4 Interprocess Communication

The `CV` is a independent module, and it should communicate with the control kernel. We should use the `socket.io` to communicate between the `CV` and the control kernel.

3 Project Structure (CV)

The project is a `Python` project with modules. It relys on `OpenCV`, `numpy`, `pupil_apriltag`, and `python-socketio`.

The project structure is as follows:

- `calibration`: The calibration methods, including `external` and `internal` calibration.
- `localization`: The localization that can calculate the position of the camera and the vehicle in the ground coordinate system.
- `recognition`: The recognition can calculate the position of the block in the ground coordinate system with the help of the `localization` module.
- `configuration`: The data configuration of `colors` (color range of the block) and `tags` (the position of the `apriltags` in the ground coordinate system).
- `main.py`: The main file of the project. It contains the main function, and is responsible for the communication between the `CV` and the control kernel.

4 Methodological Analysis

In general, positioning requires 6 degrees of freedom. That is to say, we need at least 6 points to calculate the accurate position of the camera. However, the 3 degrees of freedom will remain constant during the robot's motion.

For spatial coordinates, we use the Cartesian coordinate system (x, y, z). For the angular coordinates, we use the Euler angle (α, β, γ). Or we can call it $(\text{roll}, \text{pitch}, \text{yaw})$.

The punctuation of `apriltags` is fixed, therefore, as long as we know the position of `apriltags` in the image and its punctuation, it is possible to correspond the two-dimensional image coordinates to the three-dimensional spatial coordinates. Using PnP algorithm, we can get the position of the camera.

4.1 Calibration

The calibration is divided into 2 parts: `external` and `internal` calibration.

The first step is the `internal` calibration, which is to get the camera matrix and the distortion coefficients. We should use a chess board to get the camera matrix and the distortion coefficients, with different angles and distances.

The internal calibration process is quite simple. We can use the `calibrateCamera` method provided by `OpenCV` to get the camera matrix and the distortion coefficients.

For fun, I provided a simple internal parameter calibration method that can calibrate the camera promptly with the camera. You may turn around to get the accurate calibration data. That's quite fun, or funny.

Then, we should put our robot on a `apriltag`. Of course, you can put the robot anywhere since you know the position of the robot and the camera (using the PnP algorithm is more preferable). Then, we can calculate the transform matrix with the help of the ground coordinate system.

As we know, the transform of 2 coordinate systems is a 4×4 matrix. You can describe it as:

$$\boldsymbol{T}(\texttt{homogeneous\_matrix}) = \begin{pmatrix} \boldsymbol{R}_{ct} & \boldsymbol{t}_{ct} \\ \boldsymbol{0} & 1 \end{pmatrix} \tag{1}$$

And the procedure is (e. g. from `camera` to `vehicle`):

$$\begin{pmatrix} x_v \\ y_v \\ z_v \\ 1 \end{pmatrix} = \begin{pmatrix} \boldsymbol{R}_{ct} & \boldsymbol{t}_{ct} \\ \boldsymbol{0} & 1 \end{pmatrix} \cdot \begin{pmatrix} x_c \\ y_c \\ z_c \\ 1 \end{pmatrix} \tag{2}$$

4.1.1 Mathmetical Model

If you think this part is too boring, you can skip it, or you can take a break and tear up this article.

Transform Relationships We can get the transform matrix through the `camera` – `world` and the `world` – `vehicle` transform matrixes.

Then, the transform matrix from `camera` to `vehicle` is:

$$\boldsymbol{T}_{cv} = \boldsymbol{T}_{cw} \cdot \boldsymbol{T}_{wv} \tag{3}$$

The procedure is following:

1. Through the PnP algorithm (`resolvePnP` in `OpenCV`), we can get the transform matrix from `world` to `camera`:

$$\begin{pmatrix} x_c \\ y_c \\ z_c \\ 1 \end{pmatrix} = \begin{pmatrix} \boldsymbol{R}_{wc} & \boldsymbol{t}_{wc} \\ \boldsymbol{0} & 1 \end{pmatrix} \cdot \begin{pmatrix} x_w \\ y_w \\ z_w \\ 1 \end{pmatrix} \tag{4}$$

2. Then, we can use the vector $\begin{pmatrix}1 & 0 & 0\end{pmatrix}$ to get the transform matrix from `world` to `vehicle`, and the rotate matrix is a 3×3 unit matrix:

   Note: You must confirm that your robot's orientation is the same as the x-axis of the ground coordinate system. If not, you should adjust the orientation of the robot to make it the same as the x-axis of the ground coordinate system.

   That's because when calibrating, there is no way to get the orientation of the robot. Also, the calculation of the orientation is not so accurate. So, we should adjust the orientation of the robot to make it the same as the x-axis of the ground coordinate system. That's more accurate and convenient.

   So, the rotate matrix is a 3×3 unit matrix $\begin{pmatrix}1 & 0 & 0\\0 & 1 & 0\\0 & 0 & 1\end{pmatrix}$, and the position in the vehicle coordinate system is $(0, 0, 0)$. That's because it's the origin of the vehicle coordinate system.

   After getting the x, y coordinate in the world coordinate system, and because the `apriltags` are on the ground, we can easily get 2 transform vectors:

$$\boldsymbol{t}_w = \begin{pmatrix}x_w & y_w & 0\end{pmatrix}, \boldsymbol{t}_v = \begin{pmatrix}0 & 0 & 0\end{pmatrix} \tag{5}$$

   So, the `tvecs` (transform vector) is $\boldsymbol{t}_w - \boldsymbol{t}_v$. The rotate matrix is a 3×3 unit matrix.

   Then, we use `vstack` and `hstack` to get the full transform matrix.

   Note: The result of `solvePnP` is the rotate vector and the transform vector. We can use the `Rodrigues` method to get the rotate matrix.

3. Then, we can replace the matrix $\begin{pmatrix}x_w\\y_w\\z_w\\1\end{pmatrix}$ with the matrix $\boldsymbol{T}_{cw} \cdot \begin{pmatrix}x_c\\y_c\\z_c\\1\end{pmatrix}$ to get the transform matrix from `camera` to `vehicle`:

$$\boldsymbol{T}_{cw} \cdot \begin{pmatrix}x_c\\y_c\\z_c\\1\end{pmatrix} = \boldsymbol{T}_{vw} \cdot \boldsymbol{T}_{cv} \cdot \begin{pmatrix}x_c\\y_c\\z_c\\1\end{pmatrix} \tag{6}$$

   Because the vectors in the left and right are the same, we can get relationships between these matrixes:

$$\boldsymbol{T}_{cw} = \boldsymbol{T}_{vw} \cdot \boldsymbol{T}_{cv} \tag{7}$$

   So, we can get the transform matrix from `camera` to `vehicle` through the transform matrix from `world` to `vehicle` and the transform matrix from `world` to `camera`:

$$\boldsymbol{T}_{cv} = \boldsymbol{T}_{vw}^{-1} \cdot \boldsymbol{T}_{cw} = \boldsymbol{T}_{wv} \cdot \boldsymbol{T}_{cw} \tag{8}$$

4. Then, we can use `numpy.linalg.inv` to get the inverse matrix of $\boldsymbol{T}_{vw}$, and then dot it with $\boldsymbol{T}_{cw}$ to get the transform matrix from `camera` to `vehicle`.

That's quite simple, isn't it?

4.1.2 Basic Parameters

We need the camera's internal reference and distortion coefficients to accurately confirm the conversion.

The camera's internal reference is a 3×3 matrix, which is the camera's focal length and the center of the image:

$$\texttt{camera\_matrix} = \begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix}$$

The distortion coefficients are 5 parameters, which are used to correct the distortion of the image:

$$\texttt{distortion\_coefficients} = \begin{pmatrix} k_1 & k_2 & p_1 & p_2 & k_3 \end{pmatrix}$$

The relevant parameters of the camera are not considered to vary excessively during the process. In other words, we can store these parameters directly at call time.

4.2 Localization

> **Note:** These methods are deprecated. We import the `apriltags`'s corners (range: left – up, right – up, right – down, left – down). Through the range and the width of the tag in the ground tag, we can get 4 or 5 points to calculate points.

4.2.1 6-point Adjustment

Each location of the `apriltags` can define $\dfrac{1}{6}$ of the camera's position. Therefore, we need at least 6 `apriltags` to calculate the position of the camera.

$$\begin{pmatrix} x & y & z \\ \text{roll} & \text{pitch} & \text{yaw} \end{pmatrix}$$

Through the PnP algorithm provided by `OpenCV`, we can calculate the transform vector and the rotation vector of the camera easily.

After getting `tvec` and `rvec`, we can use the `Rodrigues` function to convert the rotation vector to the rotation matrix:

$$\boldsymbol{R}\texttt{\_mtx} = \texttt{Rodrigues}\,(\texttt{rvec})$$

Then, through some simple calculations, we can get the Euler angle.

4.2.2 3-point Localization

The `apriltags`'s center point is accurate enough to calculate the position of the camera. That is to say, through at least 3 `apriltags`, we can calculate the position of the camera through the PnP algorithm.

Knowing z, roll and pitch, we should calculate the "bias" matrix.

Define the original matrixes of rotate and transform through these elements:

$$\boldsymbol{R}_x = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha \\ 0 & \sin\alpha & \cos\alpha \end{pmatrix}, \boldsymbol{R}_y = \begin{pmatrix} \cos\beta & 0 & \sin\beta \\ 0 & 1 & 0 \\ -\sin\beta & 0 & \cos\beta \end{pmatrix} \tag{9}$$

Then, we can get the rotate "bias" matrix:

$$\boldsymbol{R}_{\texttt{bias}} = \boldsymbol{R}_x \cdot \boldsymbol{R}_y = \begin{pmatrix} \cos\beta & 0 & \sin\beta \\ \sin\alpha\sin\beta & \cos\alpha & -\sin\alpha\cos\beta \\ -\cos\alpha\sin\beta & \sin\alpha & \cos\alpha\cos\beta \end{pmatrix} \tag{10}$$

Through `Rodrigues` method, we can get the rotate vector of the "bias" matrix.

Also, the transform vector is easy to get:

$$\boldsymbol{t}_{\texttt{bias}} = \begin{pmatrix} 0 \\ 0 \\ -z \end{pmatrix} \tag{11}$$

4.2.3 1-point Emergency Localization

We can know each corner of the `apriltags` of the map. Therefore, we can calculate the position of the camera through the PnP algorithm.

However, I strongly recommend that we should not use this method. It is not so accurate and it is easy to make mistakes.

If there is less than 3 `apriltags` in the view of the camera, we can calculate corners of the `apriltags` in the image. Then we can calculate the position of the camera through the PnP algorithm.

4.2.4 Turn Around

The robot can never gonna give you up, never gonna let you down. It can never gonna run around and desert you.

So, don't cry, don't say goodbye, don't tell a lie and hurt you.

If there's no `apriltags`, what the robot should do is to turn around and find the `apriltags`. There is no place without `apriltag` around the robot.

4.2.5 Used Algorithm

In order to calculate the position of the vehicle (using the `external parameters`), we can use the multiplication of the `world` – `camera` (PnP) and the `camera` – `vehicle` (external parameter) transform matrixes to get the `vehicle` – `world` transform matrix.

We finally decided to deprecate the Eular Angle and use the Heading Angle instead. I selected the x-axis of the ground coordinate system as the 0° angle, and it is actually the `yaw`.

Then, in order to improve the accurate, I decided to use corners and centers together. Although the $1, 3, 6$ method was deprecated, its idea is still useful. What it inspired me is one point can decide one free degree in average.

4.3 Block Recognition

4.3.1 Color Recognition

Through the `HSV` color space, we can easily recognize the color of the block. The `HSV` color space is a cylindrical coordinate system. The three dimensions represent the hue, saturation, and value, respectively.

We use the opening operation to remove the noise. That's becuase there are white dots on colored blocks and it is not so easy to remove. The opening operation can relatively remove the white dots, making the color recognition more accurate.

The adjustment of target range is a hard work. We should adjust the range of the color recognition to make it more accurate.

4.3.2 Shape Recognition

Recognition Due to the interference of the points in the dice, we cannot determine the outline of each dice very accurately. But the closer the distance, the higher the definition.

We use `HSV` color space to recognize the color of the dice. Before recognizing, we use the opening operation to remove some noise. You can see the action in the next section.

Then, we calculated the transform matrix, and use the `warpPerspective` function to transform the image:

$$\boldsymbol{R}_x = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{pmatrix}, \boldsymbol{R}_y = \begin{pmatrix} \cos\phi & 0 & \sin\phi \\ 0 & 1 & 0 \\ -\sin\phi & 0 & \cos\phi \end{pmatrix}, \boldsymbol{R}_z = \begin{pmatrix} \cos\psi & -\sin\psi & 0 \\ \sin\psi & \cos\psi & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{12}$$

It is the rotate matrix from Eular Angle.

Then, we firstly dot the $\boldsymbol{R}_x$ and $\boldsymbol{R}_y$ matrixes, and then dot the result with $\boldsymbol{R}_z$ matrix. We can get the transform matrix.

$$\boldsymbol{R}_{\texttt{transform}} = \boldsymbol{R}_z \cdot (\boldsymbol{R}_x \cdot \boldsymbol{R}_y) \tag{13}$$

The vertical transform vector is quite easy. Just simply resize it.

Then, we can get the full matrix through `hstack` and `vstack` method. You can see the full algorithm in next section.

Calculation The camera can only capture the two-dimensional image. So we simply use the `N2` algorithm to abstract objects into particles:

$$\begin{pmatrix} x & y \end{pmatrix} = \left(\sqrt{\frac{\sum x_i^2}{n}} \quad \sqrt{\frac{\sum y_i^2}{n}} \right)$$

The reason why the arithmetic mean is not used is because I think it is not elegant enough.

4.4 Communicating

Because the version supportation problems, we finally decide to use `socket.io` to communicate between the contorl kernel and the vision module. The `python-socketio` provides the full package, and we can easily realize it.

The communication can be described as only the `send-to-receive` procedure. The CV core sends the location data, and the control core (maybe `ROS`) receive it and send it to the computing program to calculate the best way.

5 Algorithm Implementation

Using PnP method, we can get the location of the camera easily. The project structure includes `localization` folder, and it is the core of the localization module.

5.1 Kernel: class `Location`

I packed the method into the class `Location`. It contains position, orientation (Euler angle), the camera matrix and the distortion coefficients which is saved in `data/calibration.npz`.

The `x`, `y`, `z`, `roll`, `pitch`, `yaw` are the position and orientation of the camera. The `camera_matrix` and `distortion_coefficients` are the camera matrix and the distortion coefficients.

`x`, `y`, `z`, `roll`, `pitch`, `yaw` are `np.float32` type, and the `camera_matrix` is a 3×3 `np.ndarray` matrix with `np.float32`, and the `distortion_coefficients` is a 5×1 `np.ndarray` matrix with `np.float32`.

The kernel will detect `data/calibration.npz` file, if there's no parameters or the file is not exist, it will raise an error.

Note Because the author is familiar with `HTTP Methods`, the introduction of classes and methods is written in "the form of `HTTP Methods`".

5.1.1 `POST` Methods – Initialization

You need to initialize the class `Location` with the `camera_matrix` and the `distortion_coefficients`. If you don't want to directly write the parameters, you can use the `calibrate` method to calibrate the camera. If there's no parameters, the initialization method will read the parameters from `data/calibration.npz`.

If there's still no parameters, the initialization method will raise an error: `Calibration file does not exist`.

5.1.2 `GET` Methods

The `Location` class provides variety of methods to return the position and orientation of the camera. I don't want to introduce these parameters to you, as I think you are clever enough to know them. Otherwize, you are not suitable to read this document. The methods are:

- `is_adjusted`: Return `True` if the camera is adjusted, otherwise return `False`.
- `get_z`: Return the z-axis value of the camera.
- `get_position`: Return the position of the camera. The return type is (x, y, z).

- `get_orientation`: Return the orientation of the camera. The return type is (roll, pitch, yaw).
- `get_camera_matrix`: Return the camera matrix:

$$\texttt{camera\_matrix} = \begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix}$$

- `get_distortion_coefficients`: Return the distortion coefficients:

$$\texttt{distortion\_coefficients} = \begin{pmatrix} k_1 & k_2 & p_1 & p_2 & k_3 \end{pmatrix}$$

  Note: 2 functions above has been stored since the initialization of the class. It won't change during the process.

- `get_location_matrix`: Return the location matrix. It is a 3×2 matrix. The first line is the position in the ground coordinate system, and the second line is the orientation (Euler angle).

$$\texttt{location\_matrix} = \begin{pmatrix} x & y & z \\ \text{roll} & \text{pitch} & \text{yaw} \end{pmatrix}$$

  In this document, I may regard θ as the roll, ϕ as the pitch, and ψ as the yaw.

5.1.3 `PUT` Methods – Adjustment and Localization

The kernel provides functions to adjust the position and orientation of the camera. Then, you can use less `apriltags` to locate the camera.

Adjustment You can use the `adjust` method to adjust the position and orientation of the camera. The class will store z, roll, and the pitch data.

You need provide at least 6 `apriltags` to adjust the camera. Or the class will raise an error: `At least 6 tags are required to adjust the camera`.

Localization The class will provide a packed method to locate the camera. No matter how many `apriltags` are in the view of the camera, the class will calculate the position and orientation of the camera, (except there's no `apriltags` in the view of the camera.)

The class will use the PnP algorithm to calculate the position and orientation of the camera. If there's no `apriltags` in the view of the camera, the class will raise an error: `No tags in the view of the camera`.

5.2 Other Functions

There are also some implementation functinos. They are not so important, so I won't introduce them to you.

6 Block Recognition

The block recognition is implemented in the `recognition` folder. The core of the block recognition is the `BlockRecognition` class.

6.1 Kernel: class `BlockRecognition`

It is a class that can provide methods to recognize the color and the shape of the block. The class will use the `Location` class to get the position of the camera.

The class will use the `HSV` color space to recognize the color of the block. The class will use the `N2` algorithm to decide wich point to use to calculate the position of the block.

If you call the `recognize`, the class will automatically capture a image to recognize the block. You need to provide the color range (lower, and upper) to recognize it.

After recognition, it will save color and position of the block into `blocks`. The `blocks` is a `tuple` type. You can use `get_blocks` method to get the `blocks`.

6.2 Recognition Method: `detect`

The detector will use the `HSV` color space to recognize the color of the block.

```
1  import cv2
2  import numpy as np
3  from cv2.typing import MatLike
4  from ..configuration.colors import Color
5  from sklearn.cluster import DBSCAN
6
7
8  def dbscan(data: np.ndarray, eps:  float = 2, min_samples:  int = 5):
9      dbscan = DBSCAN(eps=eps, min_samples=min_samples)
10     clusters = dbscan.fit_predict(data)
11     return clusters
12
13
14 def merge_duplicates(contours:  list[np.ndarray]):
15     data = np.array([np.mean(cnt, axis=0) for cnt in contours])
16     clusters = dbscan(data)
17     result = []
18     for i in  range( len(clusters)):
19         result.append(
20             np.concatenate(
21                 [contours[j] for j in  range( len(clusters)) if clusters[j] == i]
22             )
23         )
24     return result
25
26
27 def filter_color(image: MatLike, hsv: MatLike, color: Color):
28     identified_colors:  list[np.ndarray] = []
29     for lower, upper in color. range:
30         mask = cv2.inRange(hsv, lower, upper)
31         result = cv2.bitwise_and(image, image, mask=mask)
32
```

```
        # Draw it
        contours, _ = cv2.findContours(
            mask.copy(), cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE
        )

        # contours = merge_duplicates(contours)

        for cnt in contours:
            area = cv2.contourArea(cnt)
            if area < 1200:
                continue

            identified_colors.append(cnt)

    return color.name, identified_colors


def n2_average(data: np.ndarray):
    if np.shape(data)[2] != 2:
        return None

    x = np.square(data[0, :][:, 0])
    y = np.square(data[0, :][:, 1])

    x_mean = np.mean(x)
    y_mean = np.mean(y)

    return np.sqrt(x_mean), np.sqrt(y_mean)


def detect(image: MatLike, colors:  list[Color]):
    # Change to HSV color space
    hsv = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)

    # Opening operation
    kernel = np.ones((5, 5), np.uint8)
    opening = cv2.morphologyEx(hsv, cv2.MORPH_OPEN, kernel, iterations=2)

    # Filter colors

    result = []

    for color in colors:
        name = color.name
        color, contours = filter_color(opening, hsv, color)

```

```
        if  len(contours) == 0:
            continue

        for cnt in contours:
            # Draw a bounding box around the detected object
            x, y, w, h = cv2.boundingRect(cnt)

            if w * h < 2000:
                continue

            # Get average

            result.append((name, [x + w / 2, y + h / 2]))


    return result
```

6.3 Locate Method: `locate_block`

The locate method provides the location of blocks in the image and the ground coordinate system. It will fetch `camera_matrix` and `distortion_coefficients` from the `Location` class. It will also get the camera position and orientation from the `Location` class.

Then, it will calculate the rotate & transform matrix ($\boldsymbol{R}_{\texttt{transform}}$ and $\boldsymbol{t}_{\texttt{transform}}$) through the position and orientation of the camera. Then, it will use the `undistortPoints` method to transform the image.

It's the core of the block localization, and it's important enough to display the code here.

```
import cv2
import numpy as np
from ..localization.kernel import Location
from .vtoc import restore_from_vehicle

def locate_block(camera_location: Location, block_image_position:  tuple[ float, float], Z_c: np.float32):
    camera_position = camera_location.get_position()
    camera_orientation = camera_location.get_orientation()
    camera_matrix = camera_location.get_camera_matrix()
    dist_coeffs = camera_location.get_dist_coeffs()

    print(block_image_position)

    # Rotation matrix
    R_x = np.array([[1, 0, 0],
                    [0, np.cos(camera_orientation[0]), -np.sin(camera_orientation[0])
                        ],
```

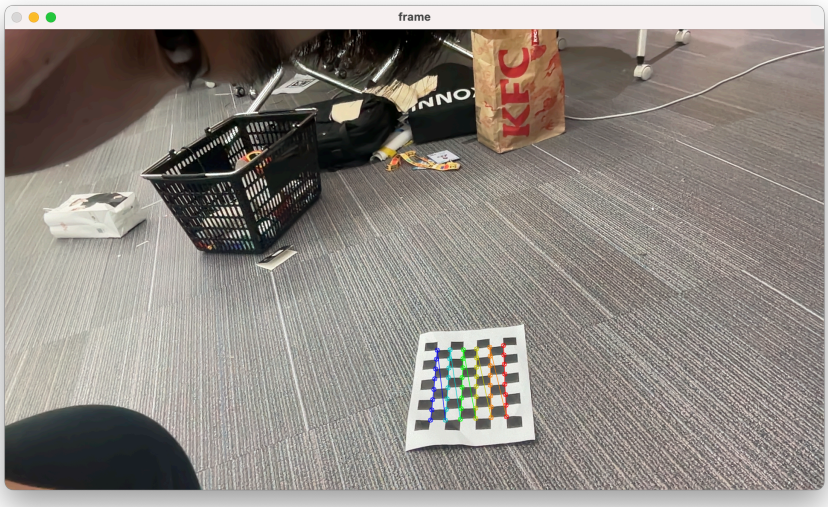
```
                    [0, np.sin(camera_orientation[0]), np.cos(camera_orientation[0])
                        ]])
    R_y = np.array([[np.cos(camera_orientation[1]), 0, np.sin(camera_orientation[1])],
                    [0, 1, 0],
                    [-np.sin(camera_orientation[1]), 0, np.cos(camera_orientation[1])
                        ]])
    R_z = np.array([[np.cos(camera_orientation[2]), -np.sin(camera_orientation[2]),
        0],
                    [np.sin(camera_orientation[2]), np.cos(camera_orientation[2]), 0],
                    [0, 0, 1]])
    R = np.dot(R_z, np.dot(R_y, R_x))
    T = camera_position.reshape(3, 1)
    matrix_vtow = np.hstack((R, T))
    matrix_vtow = np.vstack((matrix_vtow, [0, 0, 0, 1]))
    matrix_ctov = restore_from_vehicle()

    transformation_matrix = np.dot(matrix_ctov, matrix_vtow)

    u, v = block_image_position

    uv_point = np.array([[u, v]], dtype=np.float32)

    undistorted_point = cv2.undistortPoints(uv_point, camera_matrix, dist_coeffs, P=
        camera_matrix)
    X_c = (undistorted_point[0][0][0] - camera_matrix[0, 2]) / camera_matrix[0, 0] *
        Z_c
    Y_c = (undistorted_point[0][0][1] - camera_matrix[1, 2]) / camera_matrix[1, 1] *
        Z_c

    point_camera = np.array([X_c, Y_c, Z_c, 1])
    point_world = np.dot(transformation_matrix, point_camera)
    X_w, Y_w, Z_w = point_world[:3]

    return X_w, Y_w, Z_w
```

According to the code, it will calculate the position of the block as well.

7 Algorithm Effect

The algorithm is quite effective. It can recognize the color of the block and the position of the block.

7.1 Calibration



According to the image, the `OpenCV` can get the camera matrix and the distortion coefficients with chess board.

Then, we can get the data in `data/calibration.npz`.

In this demo, we get the data:

$$\texttt{camera\_matrix} = \begin{pmatrix} 1503.75365 & 0.0 & 899.284676 \\ 0.0 & 1514.5398703 & 367.845474 \\ 0.0 & 0.0 & 1.0 \end{pmatrix}$$

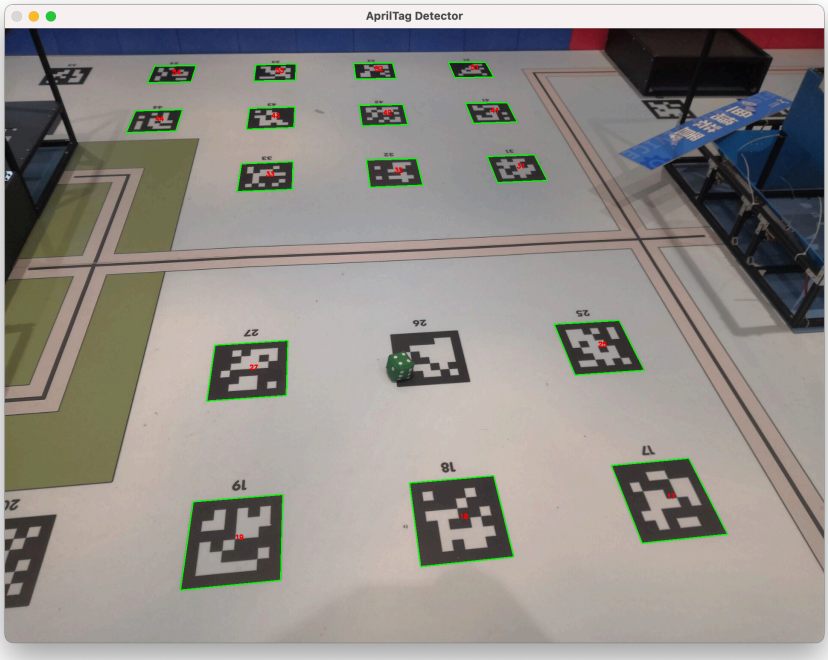
$$\texttt{distortion\_coefficients} = \begin{pmatrix} -0.010567 & 0.302944 & -0.070295 & -0.021696 & -0.445497 \end{pmatrix}$$

Because the image is not flat enough, the calibration is not so accurate. However, it is enough for the algorithm.

Then, we can see the file in `data` folder.

7.2 Localization

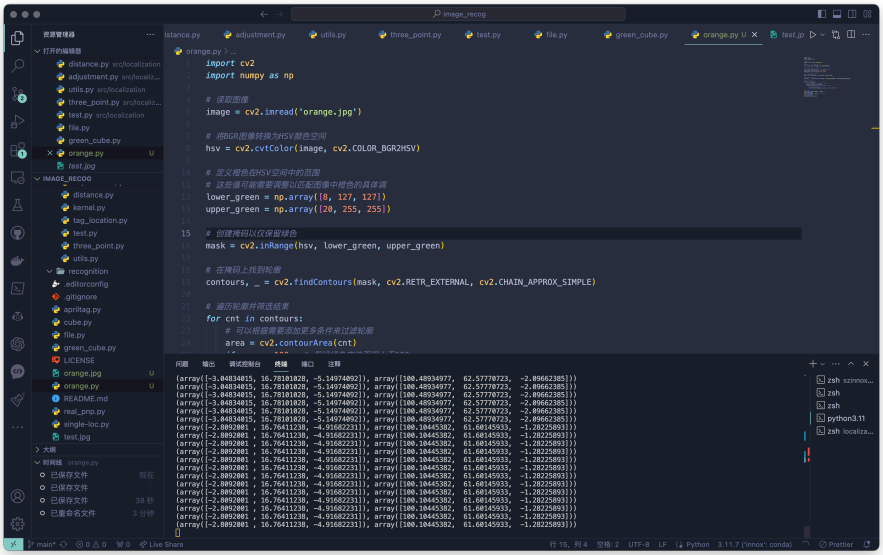
7.2.1 AprilTag Caption



The `OpenCV` and the `pupil_apriltag` package can recognize `package` properly. The picture above can show the effect.

Then, through at laast 6 `apriltags`, we can adjust the position and orientation of the camera.

For example, the position and orientation of the camera (prompt) in the following picture:

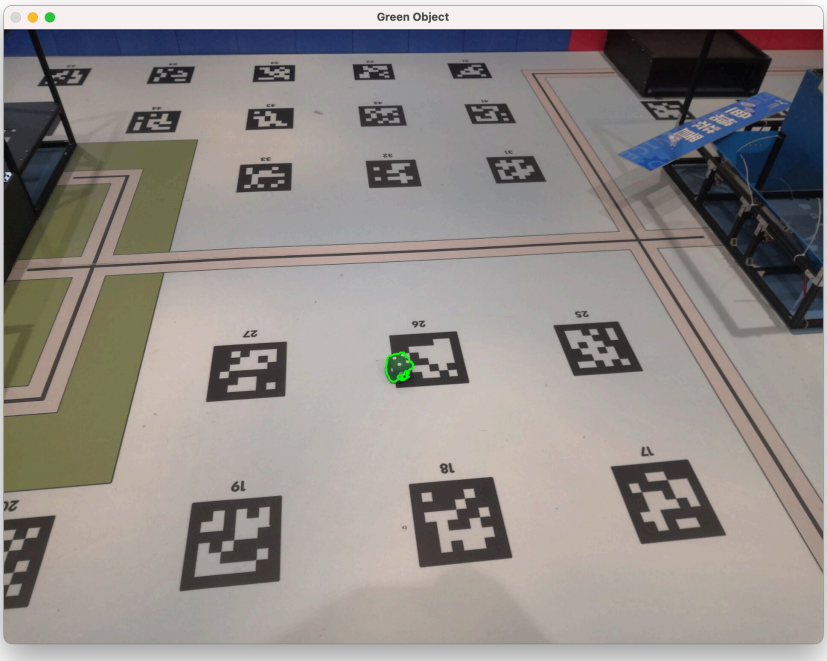


The array is $(x, y, z, \text{roll}, \text{pitch}, \text{yaw})$.

It can display as $\begin{pmatrix} -2.8092001 & 16.76411238 & 4.916822 & 100.10445382 & 61.60145933 & -1.28225893 \end{pmatrix}$ when testing with my laptap.

7.2.2 Block Recognition

The color recognition can work properly. The picture above can show the effect.



Then, adding the mask can make the recognition more accurate. We can recognize the border and then calculate the position of the block.

A Git Repository

The project is stored in the `GitHub` repository. You can visit the repository through the link: `https://github.com/A3-SZInnoX-2024/localization.git`