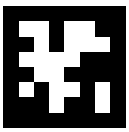
Localization System of Group A2

Wu Chengyu 7086cmd@gmail.com

February 1, 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.
- `OPTIONAL`: Calibrate the camera promptly.

2.2 Localization & Navigation

- 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`.

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.

3 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.

3.1 Calibration

The first step of localization, navigation and block recognition is to calibrate the camera, knowing the camera's internal reference and distortion coefficients.

The 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 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.

3.1.1 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.

3.2 Localization

3.2.1 6-point Adjustment

Each location of the `apriltags` can define $\frac{1}{6}$ 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.

3.2.2 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:

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

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

$$\boldsymbol{R}_{\texttt{bias}} = \boldsymbol{R}_x \cdot \boldsymbol{R}_z = \begin{pmatrix} \cos\gamma & -\sin\gamma & 0 \\ \sin\gamma\cos\alpha & \cos\gamma\cos\alpha & -\sin\alpha \\ \sin\gamma\sin\alpha & \cos\gamma\sin\alpha & \cos\alpha \end{pmatrix} \tag{2}$$

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

Also, the transform vector is easy to get:

$$\boldsymbol{t}_{\texttt{bias}} = \begin{pmatrix} 0 \\ 0 \\ -z \end{pmatrix} \tag{3}$$

3.2.3 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.

3.2.4 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.

3.3 Block Recognition

3.3.1 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.

3.3.2 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:

$$\boldsymbol{R}_x = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{pmatrix}, \boldsymbol{R}_y = \begin{pmatrix} \cos\phi & 0 & \sin\phi \\ 0 & 1 & 0 \\ -\sin\phi & 0 & \cos\phi \end{pmatrix}, \boldsymbol{R}_z = \begin{pmatrix} \cos\psi & -\sin\psi & 0 \\ \sin\psi & \cos\psi & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{4}$$

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.

$$\boldsymbol{R}_{\texttt{transform}} = \boldsymbol{R}_z \cdot (\boldsymbol{R}_x \cdot \boldsymbol{R}_y) \tag{5}$$

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:

$$\begin{pmatrix} x & y \end{pmatrix} = \begin{pmatrix} \sqrt{\frac{\sum x_i^2}{n}} & \sqrt{\frac{\sum y_i^2}{n}} \end{pmatrix}$$

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

3.4 Communicating

Finally, we select `ROS` to communicate with the robot. It is not implemented yet.

4 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.

4.1 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`".

4.1.1 `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`.

4.1.2 `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.

4.1.3 `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.)