6. AR Vision

Function package: /home/yahboom/YBAMR-COBOT-EDU-00001/src/yahboom_navrobo_other/yahboom_navrobo_visual

This section runs on NAVROBO, and Jetson is used as an example.

6.1. Overview

Augmented Reality, referred to as "AR", is a technology that cleverly integrates virtual information with the real world. It widely uses multimedia, three-dimensional modeling, real-time tracking and registration, intelligent interaction, sensing and other technical means to simulate computer-generated text, images, three-dimensional models, music, videos and other virtual information, and apply them to the real world. The two types of information complement each other, thereby achieving "enhancement" of the real world.

The AR system has three outstanding characteristics: ①Integration of information between the real world and the virtual world; ②Real-time interactivity; ③Adding and positioning virtual objects in three-dimensional space.

Augmented reality technology includes new technologies and methods such as multimedia, three-dimensional modeling, real-time video display and control, multi-sensor fusion, real-time tracking and registration, and scene fusion.

6.2, Usage

Please do this step before running the program

```
sudo supervisorctl stop ChassisServer #Stop the self-starting chassis service
#Perform this step more for the indoor version of NAVROBO-astra_pro2 camera
/home/yahboom/YBAMR-COBOT-EDU-00001/start/OBColorViewer #Release color stream
video100
#[info][722318][Pipeline.cpp:251] Start streams done!
#[info][722318][Pipeline.cpp:234] Pipeline start done!
#[warning][722318][Pipeline.cpp:327] Wait for frame timeout, you can try to
increase the wait time! current timeout=100
```

When using the AR case, you must have the camera's internal parameters, otherwise it will not work. The internal reference files are in the same directory as the code (under the AR folder of the function package); different cameras correspond to different internal references.

The internal reference calibration can be quickly calibrated using a chessboard. For specific methods, see the [Astra Camera Calibration] lesson.

Start the monocular camera

```
roslaunch usb_cam usb_cam-test.launch
```

Start the calibration node

```
rosrun camera_calibration cameracalibrator.py image:=/usb_cam/image_raw
camera:=/usb_cam --size 9x6 --square 0.02
```

After calibration, move the [calibrationdata.tar.gz] file to the [home] directory.

```
sudo mv /tmp/calibrationdata.tar.gz ~
```

After decompression, open the [ost.yaml] file in the folder, find the camera intrinsic matrix and distortion coefficient and modify them to the corresponding position of the [astra.yaml] file. You only need to modify the contents of two places [data]. For example: the following content.

```
camera_matrix: !!opencv-matrix
  rows: 3
  cols: 3
  dt: d
  data: [615.50506,   0.      , 365.84388,
           0.      , 623.69024, 238.778  ,
           0.      ,   0.      ,   1.     ]
distortion_model: plumb_bob
distortion_coefficients: !!opencv-matrix
  rows: 1
  cols: 5
  dt: d
  data: [0.166417, -0.160106, -0.008776, 0.025459, 0.000000]
```

There are 12 effects in total.

```
["Triangle", "Rectangle", "Parallelogram","WindMill","TableTennisTable", "Ball",
"Arrow", "Knife", "Desk",
"Bench", "Stickman", "ParallelBars"]
```

Startup Command

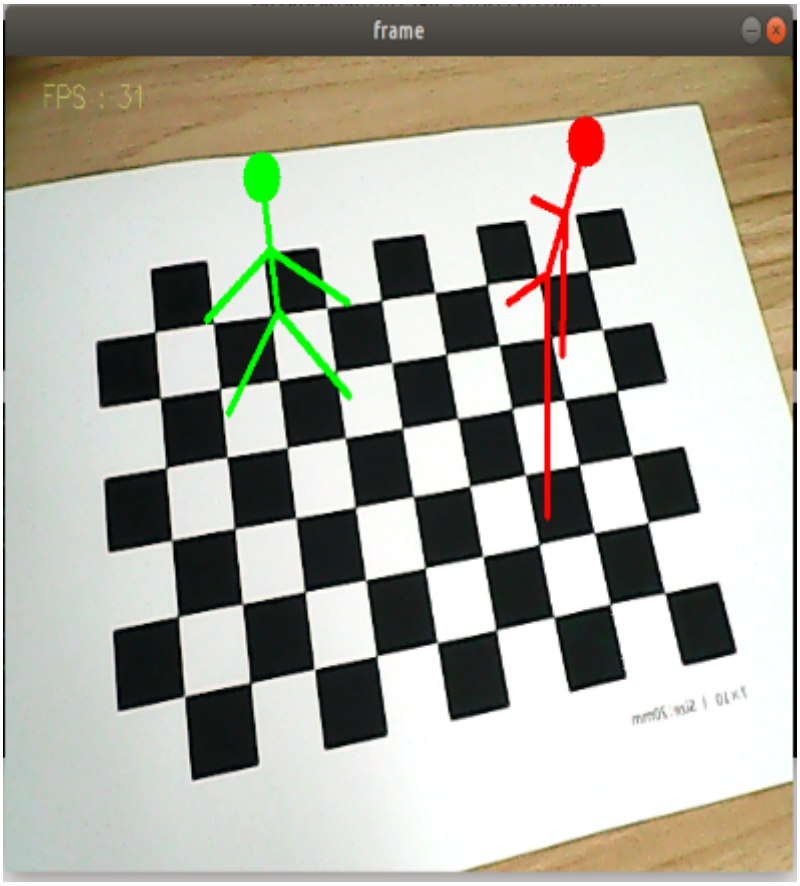
```
roslaunch yahboom_navrobo_visual simple_AR.launch display:=true flip:=false
```

- display parameter: True; local display image window; Fasle, no display.
- flip parameter: whether to switch the screen horizontally, the default is OK.

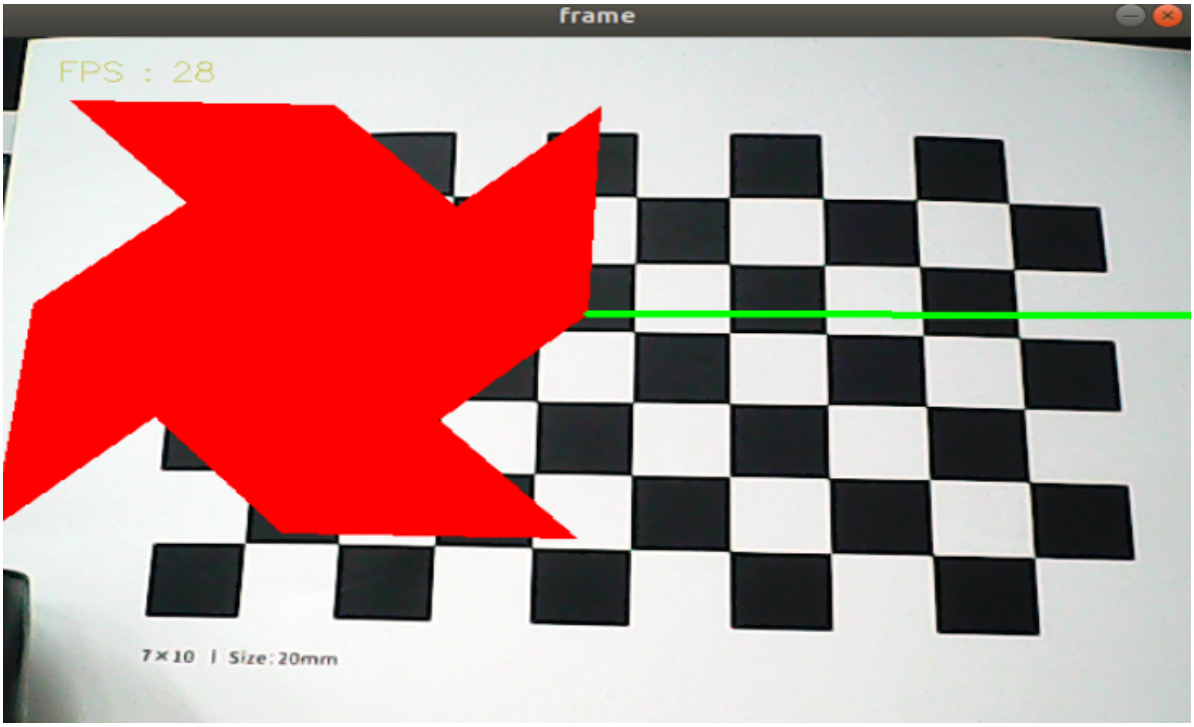
Set parameters according to needs, or modify the launch file directly, so that no additional parameters are required when starting. When the screen is not turned on, you can use network monitoring to view

```
Open the device IP: 8080
```

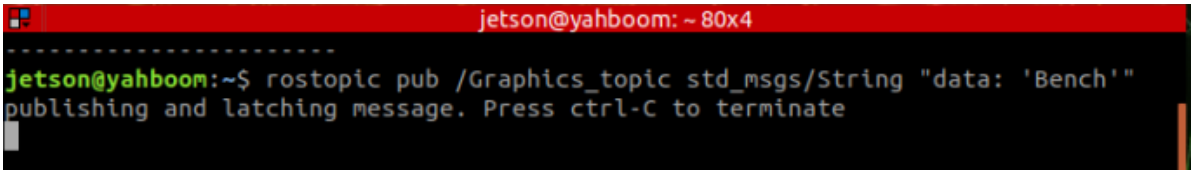
1) When the screen is displayed (that is, display is true), press [q] to exit, press [f] to switch different effects, or use the command line to switch.

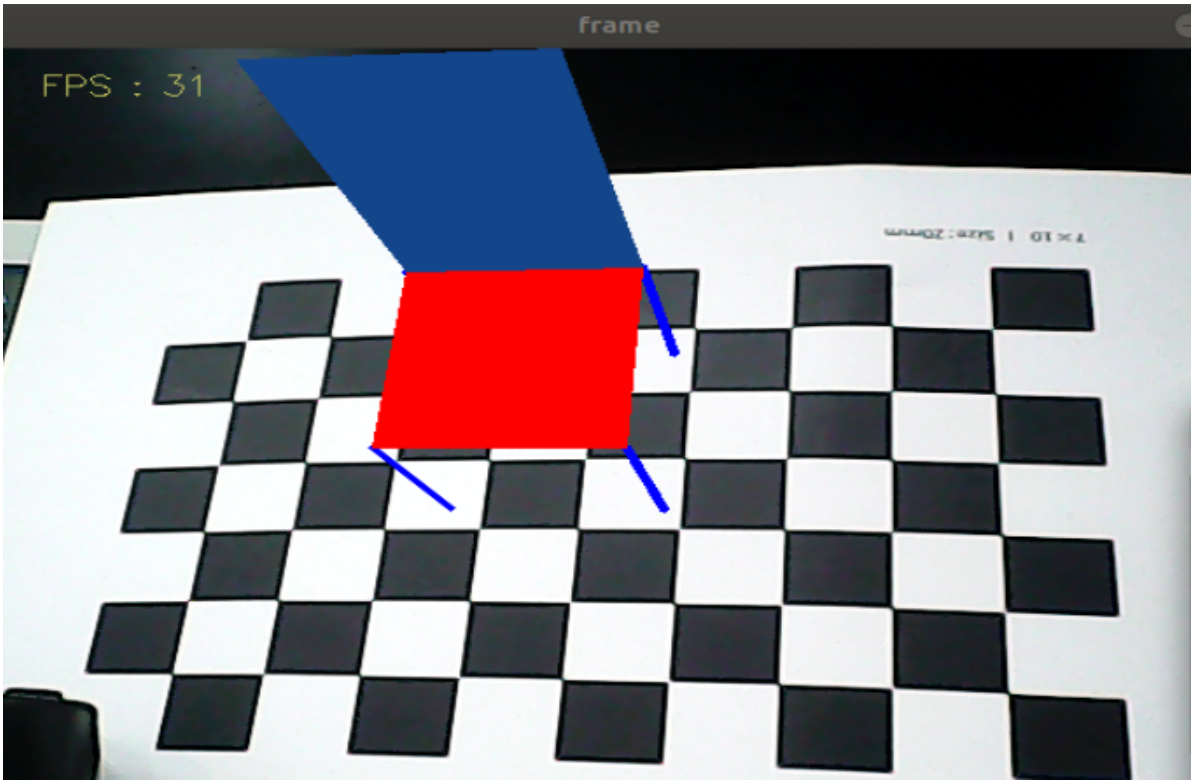


Use [f] or [F] to switch different effects.



2) When the screen is not displayed (i.e. display is false), the effect can only be switched through the command line



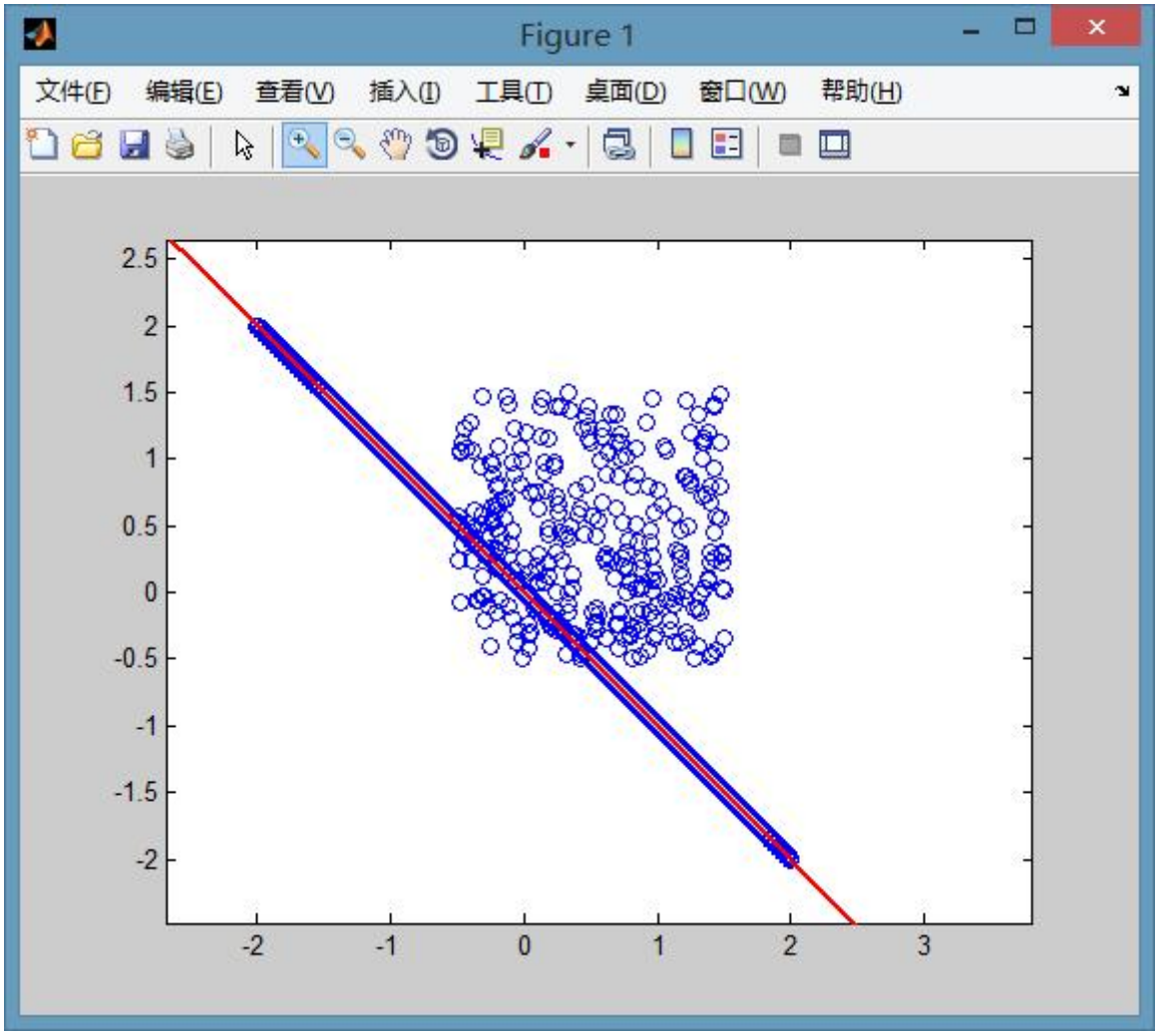


6.3, Source code analysis

6.3.1, Algorithm principle

Use the RANSAC scheme to find the object posture from 3D-2D point correspondence.

The RanSaC algorithm (random sampling consistency) was originally a classic algorithm for data processing. Its function is to extract specific components in an object under a large amount of noise. The following figure is an illustration of the effect of the RanSaC algorithm. Some points in the figure obviously satisfy a certain straight line, and another group of points is pure noise. The goal is to find the equation of a straight line in the presence of a lot of noise, where the amount of noise data is three times the amount of the straight line.



If the least squares method is used, this effect cannot be achieved, and the straight line will be slightly above the straight line in the figure.

The basic assumptions of RANSAC are:

(1) The data consists of "inside points", for example: the distribution of the data can be explained by some model parameters;

(2) "outside points" are data that cannot adapt to the model;

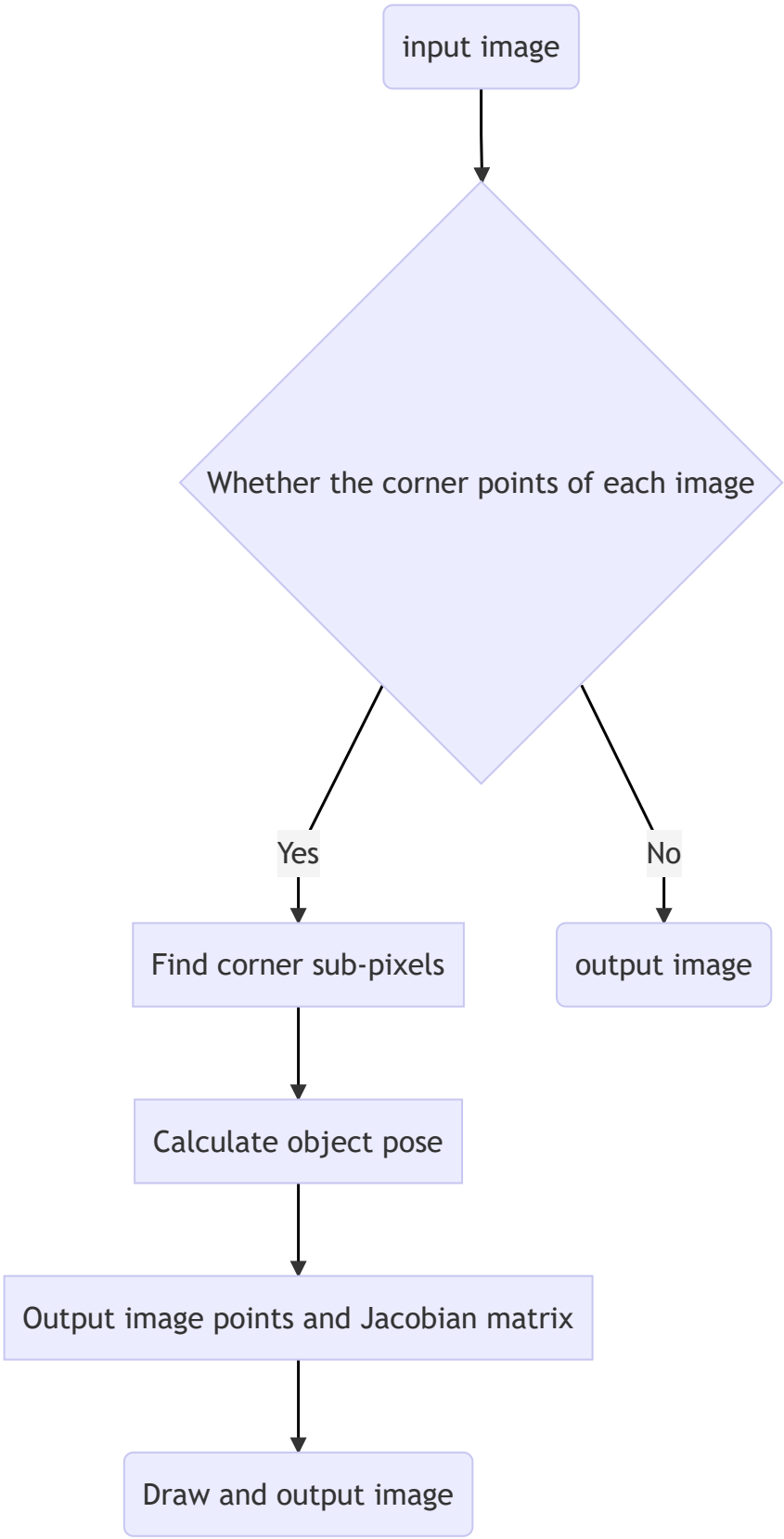
(3) Data other than these are noise.

The causes of outliers include: extreme values of noise; incorrect measurement methods; incorrect assumptions about the data.

RANSAC also makes the following assumptions: given a set of (usually small) inside points, there is a process that can estimate the model parameters; and the model can explain or apply to the inside points.

6.3.2, Core code

Design process:


input image
Whether the corner points of each image
Yes
No
Find corner sub-pixels
output image
Calculate object pose
Output image points and Jacobian matrix
Draw and output image

launch file

```
<launch>
    <arg name="flip" default="False"/>
    <arg name="display" default="False"/>
    <node name="simple_AR" pkg="yahboomcar_visual" type="simple_AR.py"
output="screen" args="$(arg display)">
        <param name="flip" type="bool" value="$(arg flip)"/>
        <remap from="/simpleAR/camera" to="/simpleAR/camera"/>
    </node>
    <!-- web_video_server -->
    <node pkg="web_video_server" type="web_video_server" name="web_video_server"
output="screen"/>
</launch>
```

Python main function

```
    def process(self, img):
        if self.flip == 'True': img = cv.flip(img, 1)
        gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY)
        # Find the corner points of each image
        retval, corners = cv.findChessboardCorners(
            gray, self.patternSize, None,
            flags=cv.CALIB_CB_ADAPTIVE_THRESH + cv.CALIB_CB_NORMALIZE_IMAGE +
cv.CALIB_CB_FAST_CHECK)
        # Find corner sub-pixels
        if retval:
            corners = cv.cornerSubPix(
                gray, corners, (11, 11), (-1, -1),
                (cv.TERM_CRITERIA_EPS + cv.TERM_CRITERIA_MAX_ITER, 30, 0.001))
            # Calculate the object pose solvePnPRansac
            retval, rvec, tvec, inliers = cv.solvePnPRansac(
                self.objectPoints, corners, self.cameraMatrix, self.distCoeffs)
            # Output image points and Jacobian matrix
            image_Points, jacobian = cv.projectPoints(
                self.__axis, rvec, tvec, self.cameraMatrix, self.distCoeffs, )
            # Draw the image
            img = self.draw(img, corners, image_Points)
        return img
```

Key functions

https://docs.opencv.org/3.0-alpha/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html

- findChessboardCorners()

```
def findChessboardCorners(image, patternSize, corners=None, flags=None):
    '''
    Find image corners
    :param image: Input the original chessboard image. The image must be an 8-bit
grayscale or color image.
    :param patternSize: (w,h), the number of interior angles in each row and
column of the chessboard. w = the number of black and white blocks in a row of
the chessboard - 1, h = the number of black and white blocks in a column of the
chessboard - 1.
    For example: for a 10x6 chessboard, (w,h)=(9,5)
    :param corners: array, the output array of detected corner points.
```

```
    :param flags: int, different operation flags, can be 0 or a combination of
the following values:
    CALIB_CB_ADAPTIVE_THRESH Use adaptive thresholding to convert the image to
black and white instead of using a fixed threshold.
    CALIB_CB_NORMALIZE_IMAGE Histogram equalizes the image before binarizing it
with fixed or adaptive thresholding.
    CALIB_CB_FILTER_QUADS Use additional criteria (such as contour area,
perimeter, squareness) to filter out false quadrilaterals extracted during the
contour retrieval phase.
    CALIB_CB_FAST_CHECK Runs a fast check on the image to find checkerboard
corners and returns a quick alert if none are found.
    Can greatly speed up calls in degenerate conditions when no checkerboard is
observed.
    :return: retval, corners
    '''
    pass
```

- cornerSubPix()

We need to use cornerSubPix() to further optimize the detected corner points so that the accuracy of the corner points can reach the sub-pixel level.

```
def cornerSubPix(image, corners, winSize, zeroZone, criteria):
    '''
    Sub-pixel corner detection function
    :param image: input image
    :param corners: pixel corners (both input and output)
    :param winSize: region size is NXN; N=(winSize*2+1)
    :param zeroZone: similar to winSize, but always has a smaller range,
Size(-1,-1) means ignore
    :param criteria: criteria for stopping optimization
    :return: sub-pixel corners
    '''
    pass
```

- solvePnPRansac()

```
def solvePnPRansac(objectPoints, imagePoints, cameraMatrix, distCoeffs,
                   rvec=None, tvec=None, useExtrinsicGuess=None,
iterationsCount=None,
                   reprojectionError=None, confidence=None, inliers=None,
flags=None):
    '''
    Calculating object pose
    :param objectPoints: object point list
    :param imagePoints: corner point list
    :param cameraMatrix: camera matrix
    :param distCoeffs: distortion coefficients
    :param rvec:
    :param tvec:
    :param useExtrinsicGuess:
    :param iterationsCount:
    :param reprojectionError:
    :param confidence:
    :param inliers:
    :param flags:
```

```
    :return: retval, rvec, tvec, inliers
    '''
    pass
```

Finds object pose from 3D-2D point correspondences using the RANSAC scheme. This function estimates the object pose given a set of object points, their corresponding image projections, and the camera matrix and distortion coefficients. This function finds a pose that minimizes the reprojection error, i.e., the re-observation error, which is the sum of the squared distances between the observed pixel point projections imagePoints and the object projections (`projectPoints()`) objectPoints. The use of RANSAC avoids the influence of outliers on the results.

- projectPoints()

```
def projectPoints(objectPoints, rvec, tvec, cameraMatrix, distCoeffs,
imagePoints=None, jacobian=None, aspectRatio=None):
    '''
    Output image points and Jacobian matrix
    :param objectPoints:
    :param rvec: rotation vector
    :param tvec: translation vector
    :param cameraMatrix: camera matrix
    :param distCoeffs: distortion coefficients
    :param imagePoints:
    :param jacobian:
    :param aspectRatio:
    :return: imagePoints, jacobian
    '''
    pass
```