CV-Project-2024

Release September 2024

Marco Bonan

Sep 18, 2024

CONTENT:

This is the project documentation, its aim is to make it more understandable and provide documentation for the *CameraUtils* module.

Project repository

Warning

Before executing any script make sure to correctly *setup your environment*.

CHAPTER

ONE

INTRODUCTION

The code present in the `CODE` folder of the repository can be divided into two groups.

1. The `CameraUtils` module contains the definition of classes, methods, and functions written for the project's purpose.
2. The Scripts:
   - `configuration.py` is a Python file that contains user-definable settings that influence the behavior of scripts when they are executed. This approach was chosen for ease of development, as it simplifies the process compared to using command-line arguments.
   - `Calibration_INT.py` is a script that, according to the configuration file:
     - Gets sample frames from chessboard calibration videos and saves them to file OR simply reads calibration images from file if they are already sampled.
     - Performs the intrinsic calibration of the cameras, according to the type of camera (normal or wide-lens).
     - Saves the found parameters to a `.pkl` file for later use.
     - Shows the effect of camera calibration by undistorting the raw images.
   - `Calibration_EXT.py` is a script that reads the intrinsic parameters from the `.pkl` files and launches the `getCorrespondences()` method to allow the user to identify key point correspondences between the real world and the images. Once a sufficient number of correspondences is acquired, the extrinsic calibration is performed, as well as a plane-to-plane mapping between the image and court floor.
   - `PointsHighlighting.py` is a simple graphical user interface to display all camera views and highlight a selected point on all views, using the plane-to-plane mapping found with the previous script.
   - `Plotter.py` is a utility script used to generate plots for the written report.

CHAPTER

TWO

SET UP THE PROJECT

The project was:

- developed and tested on Ubuntu 24.04 LTS.
- tested on Ubuntu 22.04.
- tested on WSL2 - Ubuntu.

Note

IN WSL2, in order to display interactive plots make sure a backend such as *tk* is installed, by running:

```
sudo apt install python3-tk
```

2.1 Virtual environments

The next steps are adapted from this guide.

Warning

According to the latest changes in Debian release, and therefore Ubuntu 24.04, Python packages are not installed or managed via PIP in the external environment/globally. If you use the pip command to manage the Python packages outside the Python environment, you will get the error: **externally managed environment**. The best solution is to create a Python environment for the project and manage all the necessary packages/modules there using **pip**.

Install the `venv` module using `apt`:

```
sudo apt install python3-venv
```

Go to the folder with all the project files and create a virtual environment <ve_name> for Python and activate it using the source command:

```
python3 -m venv <ve_name>
source <ve_name>/bin/activate
```

You can now install the required packages by running:

```
pip install -r requirements.txt
```

> **Note**
>
> In Ubuntu 24.04, both pip and pip3 commands refer to python3-pip, as Python 2 is no longer officially supported.

pip can be installed globally by running:

```
sudo apt install python3-pip
```

If everything is correctly setup, you are ready to launch the scripts.

CHAPTER

THREE

CONFIGURATION

This script contains the necessary configurations for the calibration scripts used in the project. Here there is a brief explanation of the options and parameters that are configurable.

3.1 Overview

1. **Flags**.
2. **Settings**: Specifies camera properties, and their positions.
3. **Folders and File Paths**: Paths used for video input, calibration frames, and outputs.
4. **Court Points and Key Points**: Predefined reference points on the court for calibration correspondences.
5. **Constants**.

3.1.1 Flags

- *SAVE_PARAM*: `bool`

  Controls whether the camera parameters are saved to a file after calibration. If `True`, parameters are saved to *PARAMETER_FOLDER*

- *SHOW_TEST_FRAME*: `bool`

  If `True`, shows before/after intrinsic calibration frames of the camera.

- *GET_SAMPLE*: `bool` (default: False)

  If `True`, scans videos to get sample chessboard frames. Otherwise, uses existing frames stored in *SAMPLE_FOLDER*.

- *READ_PARAM*: `bool`

  Whether to read camera parameters from a pickle file.

- *CROP*: `bool` (default: True)

  If `True`, crops undistorted images after processing.

3.1.2 Settings

- *NUM_OF_SAMPLES*: `int`

  Number of calibration frames to sample.

- *FRAME_SKIP*: `int`

  Number of frames to skip between each calibration frame extraction.

- *SQUARE_SIZE*: `float`

  The size of a chessboard square.

- *CAMS*: `dict`

  A dictionary that stores basic information about each camera. Includes:

  - **number**: The camera's number.
  - **WIDE**: `bool` flag indicating if the camera is a wide-angle camera.
  - **position**: `list` of floats that specify the camera's position in 3D space (in meters).

- *SCALE*: `float` (default: 0.2)

  Scaling factor $0 < \text{SCALE} < 1$ for reducing plot resolution to speed up plotting.

- *TO_CAL*: `list`

  List of cameras to handle, calibrate, and plot during the calibration process.

Chessboard Calibration Setup ~~~~~~~~~~~~~——-

- *CHESSBOARD_SIZES*: `dict`

  Specifies the size of the chessboards used for calibration for each camera. Each entry is a tuple representing the rows and columns of the chessboard (e.g., (5, 7) for 5 rows and 7 columns). Change these values only if you are using different calibration videos.

Folders and File Paths ~~~~~~~~~~~~~-

- *VIDEO_FOLDER*: `str` (default: "./Video")

  Path to the folder containing game videos.

- *FORMAT*: `str`

  The video format used for calibration and game videos.

- *CALIBRATION_FOLDER*: `str` (default: "./Video/Calibration")

  Path to the folder containing calibration videos with chessboards.

- *SAMPLE_FOLDER*: `str` (default: "./Cameras/ChessBoardSamples")

  Destination folder for sampled frames from calibration videos.

- *SAMPLE_PATH*: `str` (default: "./Video/GAME/Samples/")

  Path to the folder where samples from game videos are stored.

- *UNDISTORTED_SAMPLES*: `str` (default: "./Cameras/UndistortedSamples")

  Path to store undistorted image samples after calibration.

- *PARAMETER_FOLDER*: `str` (default: "./Cameras/Parameters")

  Folder to store the camera parameters after calibration.

- *ERROR_FOLDER*: `str` (default: "./Cameras/Errors")

  Folder to store the calibration errors.

- *PLOT_FOLDER*: `str` (default: "./Plots")

  Folder to store plots.

3.1.3 Court and Plotting Settings

- *SHOW_COURT*: `bool` (default: True)

  Whether to display the court in plots.

- *COURT_IMG_LR*, *COURT_IMG_MR*, *COURT_IMG_XL*: `str`

  Paths to court images of varying resolutions (low, medium, and extra-large) for different plotting scenarios.

- *COURT_IMG_QUALITY*: `int` (default: 1)

  Controls the rendering quality of the court image. Lower values correspond to higher quality but slower rendering.

3.1.4 Court Points and Key Points

- *COURT_POINTS*: `np.array`

  A numpy array defining key reference points on the court for calibration purposes. Points are defined for both volleyball and basketball courts, and they correspond to real-world 3D coordinates (in meters). These points are used for correspondence matching in the calibration process.

3.1.5 Calibration Constants

- *MIN_POINTS*: `int` (default: 6)

  The minimum number of points required for external calibration, specifically when using planar 3D points. This ensures compatibility with the calibration system's requirement of using at least 6 points for accurate pose estimation and homography decomposition.

  Note

  On the minimum number of points:

  - *cv2.solvePnP*: Needs at least 6 points for non-planar object points and 4 points for planar objects.
  - *cv2.findHomography*: Requires at least 4 points but uses a minimum of 6 in this code to improve stability and consistency.

CHAPTER

FOUR

CAMERAUTILS

4.1 Classes

class CameraUtils.**Camera**(*camera_number*, *approximate_position*, *WIDE_LENS=False*)

A class used to represent the calibration process of a camera using chessboard images.

camera_number

The identifier number for the camera being calibrated.

Type
int

FISHEYE

A flag indicating if the camera is a WideLens.

Type
bool

chessboard_size

The number of inner corners per chessboard row and column (e.g., (7, 6)).

Type
tuple or None

obj_points

The 3D points in the real-world space.

Type
list

img_points

The 2D points in the image plane.

Type
list

mtx

The camera matrix.

Type
numpy.ndarray or None

dist

The distortion coefficients.

Type
numpy.ndarray or None

`rvecs`
The rotation vectors estimated for each pattern view.

Type
list or None

`tvecs`
The translation vectors estimated for each pattern view.

Type
list or None

`new_mtx`
The refined camera matrix used for undistortion.

Type
numpy.ndarray or None

`INT_CAL`
A flag indicating whether the camera has intrinsic calibration.

Type
bool

`EXT_CAL`
A flag indicating whether the camera has extrinsic calibration.

Type
bool

`img_size`
The size of the images used for calibration (h, w, channels).

Type
list

`roi`
The region of interest for the undistorted images.

Type
tuple or None

`corr_world_points`
A numpy array of points in 3D space, in the court reference frame.

Type
numpy.ndarray or None

`corr_image_points`
A numpy array of points in the 2D image coordinates corresponding to the real-world points (corr_world_points).

Type
numpy.ndarray or None

`ext_mtx`
The matrix representing the camera-to-world transformation.

Type
numpy.ndarray or None

`H_mtx`

The homography matrix between the court and image plane.

Type
numpy.ndarray

`rep_err`

The overall reprojection error after intrinsic calibration, representing the quality of the calibration.

Type
float or None

`pos_err`

A vector representing the positional error in 3D space after extrinsic calibration, initialized to zeros.

Type
numpy.ndarray

`__init__`(*camera_number*, *approximate_position*, *WIDE_LENS=False*)

Initializes the Camera object with a specific camera number, an approximate position in the real world, and whether or not it is WideLens. Sets all other attributes to default values.

`GetSampleFrames`(*video_dir*, *frame_skip*, *out_dir*)

Extracts sample frames from a video to use for camera calibration.

`GetCornersFromSamples`(*sample_folder*)

Reads existing images from a folder of existing samples to use for camera calibration.

`CalibrateCamera`()

Performs intrinsic calibration of the camera and sets the status to calibrated.

`SaveParameters`(*save_dir*)

Saves camera parameters to a .yaml file.

`ReadParameters`(*param_dir*)

Reads camera parameters from a .yaml file.

`NewOptimalCameraMatrix`()

Refines the camera matrix for the undistorsion.

`TestUndistorsion`()

Applies undistorsion to the image and shows the effect on an example.

`GetCorrespondences`(*world_points*, *image_path*)

Obtain correspondences between world points and image points through a manual user interface.

`AddManualCorrespondences`(*world_points*, *image_points*)

Manually add corresponding points to camera attributes.

`ExtCalibration`()

Perform external camera calibration to compute the transformation matrices between the world and camera coordinates, also compute the homography matrix that map the image and court planes.

`PosError`()

Calculate and display the positional error between the approximate and estimated positions

`SaveErrors`(`camera_list`, *path*)

This is a class method. Save the positional and re-projection errors of a list of cameras to a CSV file

`PlotCamera`(*ax*)

Plot the camera's position and direction on a 3D axis. *This method is intended to be called by *PlotMultipleCameras()* class method

`PlotMultipleCameras`(*camera_list*)

This is a class method Plot the positions and orientations of multiple cameras on a 3D plot.

`PlotCamera2D`(*ax*)

Plot the camera's position and direction on an axis. *This method is intended to be called by *PlotMultipleCameras2D()* class method.

`PlotMultipleCameras2D`(*camera_list*)

This is a class method Plot the positions and orientations of multiple cameras on a 2D court image.

`PrintAttributes`(*skip_attributes*)

Prints all the attributes of the CameraInfo instance, except for those in the skip_attributes list. Method useful for quick attributes checks/debugging

`FindHomography`()

Compute the homography matrix from world coordinates (court plane) to image coordinates.

`Court2Image`(*coords*)

Convert court coordinates to image coordinates using the homography matrix

`Image2Court`(*coords*)

Convert image coordinates to court coordinates using the homography matrix.

`AddManualCorrespondences`(*world_points*, *image_points*)

Manually add corresponding points to camera attributes after performing type and NaN checks.

Parameters

- **world_points** (*numpy.ndarray*) – Point coordinates (x, y, z) expressed in a numpy array with shape (n, 3), where *n* is the number of points. Example: *[[x, y, z], ...]*
- **image_points** (*numpy.ndarray*) – Point coordinates (u, v) expressed in a numpy array with shape (n, 2), where *n* is the number of points. Example: *[[u, v], ...]*

Returns

None

Return type

None

Notes

- This method verifies that both input arrays are of type *numpy.ndarray*.
- The input arrays are checked to have the correct shapes: *(n, 3)* for *world_points* and *(n, 2)* for *image_points*.
- NaN values within the input arrays are filtered out before the points are stored.
- If any of these checks fail, appropriate error messages will be printed.

`CalibrateCamera()`

Calibrate the camera and set the status to calibrated.

This method performs the camera calibration process using the collected object points and image points. It computes the camera matrix and distortion coefficients. The calibration process is different for fisheye lenses and standard lenses.

Raises
`RuntimeError` – If there is an error during the calibration process.

Notes

- For fisheye lenses, it uses *cv2.fisheye.calibrate* and sets the fisheye

calibration flags. - For standard lenses, it uses *cv2.calibrateCamera*. - Sets *self.INT_CAL* to True if the calibration is successful. - Prints the calibration status.

`Court2Image(coords)`

Convert court coordinates to image coordinates using the homography matrix.

This method maps 2D coordinates from the court plane to 2D image coordinates using the homography matrix. It is useful for projecting points from the court space onto the image plane.

Parameters
`coords` (*numpy.ndarray*) – A 2D numpy array of shape (n, 2) representing coordinates on the court plane in the format [x, y].

Returns
A 2D numpy array of shape (n, 2) representing the image coordinates in the format [u, v].

Return type
numpy.ndarray

Notes

- The method appends a column of ones to the court coordinates to convert them to homogeneous coordinates.
- The homography matrix *self.H_mtx* must be set before calling this method. If *self.H_mtx* is *None*, the method prints an error message and returns *None*.
- The coordinates are transformed using the homography matrix and normalized to obtain the (u, v) coordinates on the image plane.

See also

Image2Court
Method used to convert image coordinates to court coordinates.

`ExtCalibration()`

Perform external camera calibration to compute the transformation matrices between the world and camera coordinates.

This method calculates the rotation and translation vectors using the *cv2.solvePnP* function, which solves the Perspective-n-Point problem to find the position and orientation of the camera relative to the world coordinate system. It then computes the world-to-camera and camera-to-world transformation matrices and stores them as instance attributes. Additionally, it computes the homography matrix for the plane to plane mapping of the court plane to the image plane.

4.1.1 Returns:

None

4.1.2 Raises:

SystemExit
If the PnP solving process fails, an error message is printed, and the program exits.

4.1.3 Notes:

- **The method sets the following instance attributes:**
  - ***self.W2C_mtx***
    [numpy.ndarray] 4x4 transformation matrix from world coordinates to camera coordinates.
  - ***self.C2W_mtx***
    [numpy.ndarray] 4x4 transformation matrix from camera coordinates to world coordinates.
  - ***self.EXT_CAL***
    [bool] Flag indicating that the external calibration has been successfully completed.
  - ***self.H_mtx***
    [numpy.ndarray] Homography matrix for the court plane to the image plane.
- **This method relies on the instance attributes:**
  - ***self.corr_world_points***
    [list of tuples] List of corresponding points in the world coordinates.
  - ***self.corr_image_points***
    [list of tuples] List of corresponding points in the image coordinates.
  - ***self.new_mtx***
    [numpy.ndarray] Camera matrix after undistorsion.
  - ***self.dist***
    [numpy.ndarray] Distortion coefficients.

`FindHomography()`

Compute the homography matrix from world coordinates (court plane) to image coordinates.

This method calculates the homography matrix using corresponding world and image points on a plane. It utilizes the RANSAC algorithm to robustly estimate the homography.

Notes

- The method requires at least 4 corresponding points in both the world and image coordinate systems to compute the homography matrix. If fewer than 4 points are provided, an error message is printed and the method returns *None*.
- The computed homography matrix *H* is stored in the *self.H_mtx* attribute.
- The RANSAC algorithm is used with a reprojection threshold of 5 and a confidence level of 0.9 to handle outliers in the point correspondences.

Example

To compute the homography matrix, ensure that *self.corr_world_points* and *self.corr_image_points* are properly set with at least 4 corresponding points. Then call:

```
self.FindHomography()
```

where *self* is an instance of the class containing this method.

See also

`cv2.findHomography`
OpenCV function used to compute the homography matrix.

`GetCornersFromSamples`(*sample_folder*)

Reads existing images from a folder of existing samples to use for camera calibration.

This method reads sample images and detects chessboard corners in these frames. It then saves the frames and the detected points to be used for camera calibration.

Parameters
`sample_folder` (*str*) – The directory where the sampled frames are retrieved.

Raises
`FileNotFoundError` – If the specified sample folder does not exist.

Notes

- **The images are expected to be named in a specific format:**
  "out{camera_number}F{format}", where 'camera_number' is the identifier of the camera and 'format' is defined in the configuration (example *out3F.jpg* for camera number 3).
- The method uses a chessboard pattern to detect corners in the frames.
- **It divides the frame into quadrants and saves a specified number of**
  sample frames from each quadrant: - TL: Top-Left - TR: Top-Right - BL: Bottom-Left - BR: Bottom-Right
- **The method stops when the required number of samples from all**
  quadrants have been collected. The required number of steps can be set in the setting script NUM_OF_SAMPLES.

`GetCorrespondences`(*world_points*, *image_path*)

Obtain correspondences between world points and image points through a manual user interface.

4.1.4 Parameters:

world_points
[list of tuples] List of world points (x, y, z) that need to be mapped to the image points (They must lie on the court plane, so z=0!).

image_path
[str] Path to the image file in which correspondences need to be identified.

4.1.5 Returns:

coords
[list of lists] List of image coordinates corresponding to the world points. If acquisition is skipped or terminated, returns an empty list. Each coordinate is either a list of two floats [x, y] or [NaN, NaN] if the point is skipped.

4.1.6 Notes:

This function allows the user to manually select corresponding points in the image using a GUI. The user can interact with the GUI to select points, skip points, or terminate the acquisition process.

If the camera is not calibrated, the function will print a message and return without acquiring any correspondences.

The user can interact with the GUI as follows: - Right-click on the image to select a point. - Press the space bar to skip the current point. - Press 'n' to skip the calibration for the current camera. - Press 't' to terminate the point acquisition process.

This method uses OpenCV for image processing and Matplotlib for the GUI.

4.1.7 Internal Processing:

1. Reads the court plan image and the input image.
2. Undistort the input image based on the camera calibration parameters.
3. Optionally crops the image based on the region of interest (ROI).
4. Transforms world points to court plan coordinates.
5. Initiates a GUI for the user to select corresponding points.
6. Handles user interactions via mouse clicks and keyboard events.
7. Returns the list of acquired coordinates or an empty list if acquisition is skipped/terminated.

`GetSampleFrames`(*video_dir*, *frame_skip*, *out_dir*)

Extracts sample frames from a video to use for camera calibration.

This method reads a video file, samples frames at specified intervals, and detects chessboard corners in these frames. It then saves the frames and the detected points to be used for camera calibration.

Parameters

- `video_dir` (*str*) – The directory where the video file is located.
- `frame_skip` (*int*) – The number of frames to skip between each sampled frame.

- `out_dir` (*str*) – The directory where the sampled frames will be saved.

Raises

- `FileNotFoundError` – If the specified video file does not exist.
- `RuntimeError` – If the video file cannot be opened or read.

Notes

- **The video file is expected to be named in a specific format:**
  "out{camera_number}F{format}", where 'camera_number' is the identifier of the camera and 'format' is defined in the configuration (example 'out3F.mp4' for camera number 3)
- The method uses a chessboard pattern to detect corners in the frames.
- **It divides the frame into quadrants and saves a specified number of**
  sample frames from each quadrant: - TL: Top-Left - TR: Top-Right - BL: Bottom-Left - BR: Bottom-Right
- **The method stops when the required number of samples from all**
  quadrants have been collected. The required number of steps can be set in the setting script NUM_OF_SAMPLES.

`Image2Court`(*coords*)

Convert image coordinates to court coordinates using the homography matrix.

This method transforms 2D image coordinates into coordinates on the court plane by applying the inverse of the homography matrix. It is useful for mapping points from the image space to a corresponding position on the court.

Parameters

`coords` (*numpy.ndarray*) – A 2D numpy array of shape (n, 2) representing image coordinates in the format [u, v].

Returns

A 2D numpy array of shape (n, 2) representing the coordinates on the court plane in the format [x, y].

Return type

numpy.ndarray

Notes

- The method appends a column of ones to the image coordinates to convert them to homogeneous coordinates.
- The homography matrix *self.H_mtx* must be set before calling this method. If *self.H_mtx* is *None*, the method prints an error message and returns *None*.
- The inverse of the homography matrix is used to transform the coordinates to the court plane.
- The resulting coordinates are normalized to obtain the (x, y) coordinates on the court.

See also

FindHomography
: Method used to compute the homography matrix which is required for this conversion.

Court2Image
: Inverse method.

`classmethod LoadCamera`(*param_dir*, *camera*)

Load camera parameters from a .pkl file.

This method deserializes the Camera object from a file in the specified directory.

Parameters

- **param_dir** (*str*) – The directory where the parameters are stored.
- **camera** (*str*) – The identifier for the camera to be loaded.

Returns
The Camera object with the loaded parameters.

Return type
Camera

Raises
FileNotFoundError – If the specified parameter directory does not exist.

Notes

- The method expects the file to be named "Camera_{camera_number}.pkl" where *{camera_number}* is the identifier of the camera.

`NewOptimalCameraMatrix`()

Refine the camera matrix for the undistortion process.

This method computes a new optimal camera matrix based on the current camera matrix and distortion coefficients. It adjusts the camera matrix to improve the undistortion of images.

Depending on whether a fisheye lens is used or a standard lens, the method applies different algorithms to compute the new camera matrix:

- For fisheye lenses, it uses `cv2.fisheye.estimateNewCameraMatrixForUndistortRectify()`.
- For standard lenses, it uses `cv2.getOptimalNewCameraMatrix()`.

The computed camera matrix is stored in *self.new_mtx*. In the case of standard lenses, the method also updates *self.roi* with the region of interest.

Notes

- Ensure that the *self.img_size*, *self.mtx*, and *self.dist* are correctly set before calling this method.
- The *self.FISHEYE* flag determines whether to use the fisheye or standard lens processing approach.

`PlotCamera`(*ax*)

Plot the camera's position and direction on a 3D axis.

This method visualizes the camera's approximate position and estimated position on a 3D plot. It also displays a line indicating the camera's direction vector and a dashed line connecting the approximate and estimated positions.

Parameters

ax (*matplotlib.axes._axes.Axes*) – The 3D axis object on which to plot the camera information.

Notes

This method is intended to bella called by the PlotMultipleCameras() class method.

See also

PlotMultipleCameras

`PlotCamera2D(ax)`

Plot the camera's position and direction on a 2D axis over a court image.

This method visualizes the camera's estimated and approximate positions on a 2D plot with an overlay of a court image. It also shows the direction of the camera with a line and connects the estimated and approximate positions with a dashed line.

Parameters

ax (*matplotlib.axes._axes.Axes*) – The 2D axis object on which to plot the camera information.

Notes

- *This method is intended to bella called by the PlotMultipleCameras2D() class method.*
- The court image is loaded from the path specified by *conf.COURT_IMG_XL*.
- The camera's estimated position is plotted as a circular marker, while the approximate position is plotted as a gray downward-pointing triangle marker.
- A dashed red line connects the estimated position to the approximate position to indicate positional error.
- The camera's direction is shown as a line extending from the estimated position in the direction of the camera's field of view.
- An annotation with the camera number is placed near the approximate position, with a text path effect for better visibility.

See also

PlotMultipleCameras2D

`classmethod PlotMultipleCameras(camera_list)`

Plot the positions and orientations of multiple cameras on a 3D plot.

This class method visualizes all cameras in the provided list on a 3D axis. It includes the following features:

- Plots each camera's approximate and estimated positions.
- Draws direction vectors for each camera.
- Optionally overlays a court image or displays predefined court points.

Parameters

`camera_list` (*list*) – A list of camera objects, where each object should have the following methods and attributes: - *PlotCamera(ax)*: Method to plot a single camera on the 3D axis. - *approximate_position* (list or array-like): The known approximate position of the camera (x, y, z).

- *C2W_mtx* (numpy.ndarray): A 4x4 matrix containing the estimated position in the last column.
- *camera_number* (int or str): An identifier for the camera.
- *pos_err* (float): The positional error of the camera.
- *rep_err* (float): The re-projection error of the camera.

Returns

- *matplotlib.figure.Figure*
- *The figure object containing the 2D plot with camera positions.*

Notes

- The method creates a 3D plot with camera positions, direction vectors, and optional court details.
- Court image is plotted if *conf.SHOW_COURT* is *True*, otherwise, predefined court points are displayed.
- The plot's axes are labeled and ticked to provide clear context for the camera positions.

Example

To plot multiple cameras:

```
fig = CameraUtils.Camera.PlotMultipleCameras(camera_list)
```

where *Camera* is the class containing this method, and *camera_list* is a list of camera objects.

`classmethod PlotMultipleCameras2D(camera_list)`

Plot the positions of multiple cameras on a 2D court image.

This class method creates a 2D plot showing the positions of all cameras in the provided list overlaid on a court image. Each camera's position is plotted, and their respective direction vectors are displayed. The court image serves as a reference for the positions.

Parameters

`camera_list` (*list*) – A list of camera objects, where each object should have the *PlotCamera2D(ax)* method to plot its position on the 2D axis.

Returns

The figure object containing the 2D plot with camera positions.

Return type
matplotlib.figure.Figure

Notes

- The court image is loaded from the path specified by *conf.COURT_IMG_XL*.
- The x and y coordinates are adjusted to match the dimensions of the image.
- Each camera is plotted using the *PlotCamera2D* method, which should be defined in the same class as this method.
- The axis labels are set to "x" and "y" to indicate the coordinate axes.
- The image axis is turned off for a cleaner display of camera positions.

Example

To plot multiple cameras on a 2D image:

```
fig = CameraClass.PlotMultipleCameras2D(camera_list)
```

where *CameraClass* is the class containing this method, and *camera_list* is a list of camera objects.

See also

PlotCamera2D
Method used to plot individual camera positions on a 2D axis.

PosError()

Calculate and display the positional error between the approximate and estimated positions.

This method computes the Euclidean distance between the approximate_position` and the position derived from the *C2W_mtx* matrix. The result is printed and stored in the instance variable *self.pos_err*.

Notes

- *self.approximate_position* should be a list or array-like structure containing the approximate position coordinates (x, y, z).
- *self.C2W_mtx* should be a 4x4 matrix where the last column represents the estimated position in 3D space.
- The Euclidean distance is calculated as the norm of the difference between the approximate and estimated positions.
- The result is printed in meters with three decimal places.

PrintAttributes(*skip_attributes=['obj_points', 'img_points']*)

Prints all the attributes of the CameraInfo instance, except for those in the skip_attributes list.

Parameters
skip_attributes (*list of str, optional*) – A list of attribute names to exclude from printing. Default is: ["obj_points","img_points"]

Input skip_attributes = None to print all attributes

`RepError()`

Calculate the re-projection error given the parameters found in calibration.

Notes

The new refined camera matrix is used.

If the camera is not calibrated (*INT_CAL* is False), the method will print a message indicating that the re-projection error cannot be calculated and return without performing any calculations.

Return type
None

`classmethod SaveErrors(camera_list, path)`

Save the positional and re-projection errors of a list of cameras to a CSV file.

This class method generates a CSV file containing error data for each camera in the provided list. The file is saved at the specified *path* and includes the following columns:

- *CAM*: Camera identifier or number.
- *x, y, z*: Approximate position coordinates of the camera.
- *est. x, est. y, est. z*: Estimated position coordinates from the camera's *C2W_mtx* matrix.
- *distance*: Euclidean distance between the approximate and estimated positions.
- *re-projection error*: Error value representing the difference between the actual and projected positions, resulting from the intrinsic calibration procedure.

The CSV file is formatted with headers and includes error values formatted to two decimal places for positions and distances, and three decimal places for re-projection errors.

Parameters

- **camera_list** (*list*) – A list of camera objects, where each object should have the following attributes: - *approximate_position* (list or array-like): The known approximate position of

  the camera (x, y, z).

  - *C2W_mtx* (numpy.ndarray): A 4x4 matrix containing the estimated position in the last column.
  - *camera_number* (int or str): An identifier for the camera.
  - *pos_err* (float): The positional error of the camera.
  - *rep_err* (float): The re-projection error of the camera.

- **path** (*str*) – The directory path where the CSV file should be saved.

Notes

- The method assumes that each camera object in *camera_list* has the required attributes.
- The CSV file will be created with the name "Errors.csv" in the specified directory.
- Ensure that the *path* provided is a valid directory path.

`SaveParameters`(*save_dir*)

Save camera parameters to a .pkl file.

This method serializes the Camera object and saves it to a file in the specified directory.

Parameters

`save_dir` (*str*) – The directory where the parameters will be saved.

Notes

- If the folder does not exist, it is created.
- The saved file is named "Camera_{camera_number}.pkl" where

{camera_number} is the identifier of the camera.

`TestUndistorsion`()

Test the undistorsion of the camera images.

This method reads a sample image, applies the undistorsion process based on the calibration parameters, and visualizes the results. It also saves the undistorted image to a specified directory.

Notes

- The sample image is read from a predefined folder.
- The method checks if the camera is calibrated before proceeding.
- If the camera uses a fisheye lens, *cv2.fisheye.undistortImage* is used for undistorsion.
- For standard lenses, *cv2.undistort* is used.
- If cropping is enabled, the undistorted image is cropped to the region of interest (ROI).
- The original and undistorted images are displayed side by side for comparison.

Raises

- `RuntimeError` – If the camera is not calibrated.
- `FileNotFoundError` – If the sample image is not found.
- `Saves` –
- `-----` –
- `- The undistorted image is saved to a predefined directory with the filename "Cam{camera_number}.jpg".` –

class CameraUtils.PlotCameras

Handles the plotting of camera views around the court and manages the visualization of points on the court surface across all camera views.

This class is designed to initialize camera views and axes, add new camera views, and set up a plot layout with the court plan in the center and camera views around it.

views

A dictionary mapping camera identifiers (e.g., "CAM1", "CAM2") to instances of the *Camera* class. This holds all the camera views.

Type
dict

axes

A dictionary mapping camera identifiers and "court" to matplotlib axes objects. These axes are used for plotting the camera views and the court plan.

Type
dict

points

An array of shape (n, 2) to store the coordinates of points picked on the court.

Type
numpy.ndarray

court_plan

An image representing the court plan, used as a background for plotting.

Type
numpy.ndarray

color_counter

A counter used to assign different colors to highlighted points.

Type
int

__init__()

Initializes the *PlotCameras* object with empty camera views and axes, and loads the court plan image.

AddView(*camera*)

Adds a new camera view to the *views* dictionary.

InitPlot()

Initializes the plot layout with the court plan in the middle and camera views around it. Configures the layout and sets up the plotting environment.

PlotImages()

Plots the images for all camera views on their respective axes.

ShowViews()

Displays the plotted views and connects mouse and keyboard event handlers.

_on_click(*event*)

Handles mouse click events to record points on the court and save them to the points attribute.

_on_key(event):

Handles keyboard events to clear points when 'c' is pressed.

`DrawPoints()`:

Draws the most recently added point on the images and the court plan.

`ClearPoints()`:

Clears all recorded points from the *points* attribute and removes them from the plot.

`AddView`(*camera*)

Adds a camera view to the *views* dictionary.

Parameters

camera (Camera) – An instance of the *Camera* class that will be added to the *views* dictionary.

Notes

The *camera* parameter should be an instance of the *Camera* class with a *camera_number* attribute that corresponds to the camera identifier (e.g., "CAM1").

`ClearPoints()`

Clears all recorded points from the *points* attribute and removes them from the plot.

This method resets the *points* attribute to an empty array and iterates through all axes to remove plotted points. The figure is then updated to reflect the cleared points.

`DrawPoints()`

Draws the most recently added point on the images and the court plan.

This method plots the last added point on all camera views and the court plan. It projects the point from court coordinates to image coordinates for camera views, and to the court plan for visualization. Points outside the view or court plan are excluded. The color of the points is determined by *color_counter*.

Notes

The method updates the plot for each camera view and the court plan, and then increments the *color_counter* for distinguishing multiple points.

`InitPlot()`

Initializes the plot layout with a court plan in the middle and camera views around it.

This method sets up a matplotlib figure with a grid layout, placing the court plan image in the center and arranging the camera views around it. It also configures the axes for the court plan and each camera view, and sets a figure-wide title with instructions for interacting with the plot.

Notes

The layout consists of a 4x3 grid, with camera views placed in locations reflecting their real distribution around the court, which is displayed in the central subplot.

`PlotImages()`

Plots the images for all camera views on their respective axes.

This method reads images from the specified paths, undistorts them based on calibration parameters, resizes the images to improve plot rendering performance, and then displays them on their respective axes in the matplotlib figure. Each camera view is annotated with its identifier.

The images are rescaled by a factor defined in *conf.SCALE* to enhance plotting performance. Camera identifiers are added as text annotations on the images.

`ShowViews()`

Displays the plotted views and connects mouse and keyboard event handlers.

This method activates the interactive GUI for highlighting points across camera views. It sets up event handlers for mouse clicks and keyboard presses, allowing users to interact with the plot, highlight points, and clear them using specific key commands. The GUI is displayed using *plt.show()*.

4.2 Functions

`CameraUtils.GetFrame(video_folder, cam_number, n)`

Extracts and saves a specific frame from a video file.

This function reads the specified frame from a video file associated with a given camera number and saves it as an image file. If the video file cannot be opened or the frame cannot be read, an error message is printed.

Parameters

- **video_folder** (*str*) – The folder where the video files are located.
- **cam_number** (*int*) – The camera number to identify the video file.
- **n** (*int*) – The frame number to extract from the video.

Returns
None

Return type
None

Raises
ValueError – If the video file cannot be opened or the frame cannot be read.

`CameraUtils.PrintMtx(mtx)`

Print a matrix in human readable format

`CameraUtils.RW2courtIMG(RW_point, scale, RF_Img)`

Transforms a point from the real-world court coordinate system to the image coordinate system for a court plan.

Parameters

- **RW_point** (*np.ndarray*) – A 3D point in the real-world court reference frame, represented as a numpy array in homogeneous coordinates, e.g., np.array([x, y, 1]).
- **scale** (*float*) – The scale factor representing the ratio of pixels to meters. For example, a scale of 50 means 50 pixels correspond to 1 meter.
- **RF_Img** (*list or tuple*) – A list or tuple containing the (u, v) coordinates of the reference frame origin in the image, corresponding to the real-world origin (x, y) of the court.

Returns
A 2D numpy array with the transformed (u, v) coordinates in the image plane.

Return type
np.ndarray

Notes

The function uses a transformation matrix to convert the real-world court coordinates to the image court coordinates, applying scaling and translation to the point.

CameraUtils.**SampleFile**(*folder*)

Randomly selects a file from a specified folder.

This function scans the provided folder for files and randomly selects one from the list of files found. If no files are found, a warning message is printed.

Parameters
folder (*str*) – The path to the folder to scan for files.

Returns
The path of the randomly selected file.

Return type
str

Raises
FileNotFoundError – If the folder does not contain any files.

CameraUtils.**courtIMG2RW**(*img_pnt*, *scale*, *RF_Img*)

Transforms a point from the image coordinate system to the real-world court coordinate system using a homogenous transformation.

Parameters

- **img_pnt** (*np.ndarray*) – A point in the image coordinate system, represented as a numpy array in homogeneous coordinates, e.g., np.array([[u, v, 1]]).
- **scale** (*float*) – The scale factor representing the ratio of pixels to meters. For example, a scale of 50 means 50 pixels correspond to 1 meter.
- **RF_Img** (*list or tuple*) – A list or tuple containing the (u, v) coordinates of the image's reference frame origin, corresponding to the real-world origin (x, y) of the court.

Returns
A 2D numpy array representing the transformed (x, y) coordinates in the real-world court coordinate system, in meters.

Return type
np.ndarray

Notes

The function applies an inverse transformation, converting image coordinates into real-world court coordinates by applying translation and scaling.

CHAPTER

FIVE

RESOURCES

Useful resources used during development of the project.

5.1 Links

Materials:

- Starting point github
- Starting point drive
- Videos
- Calibration videos

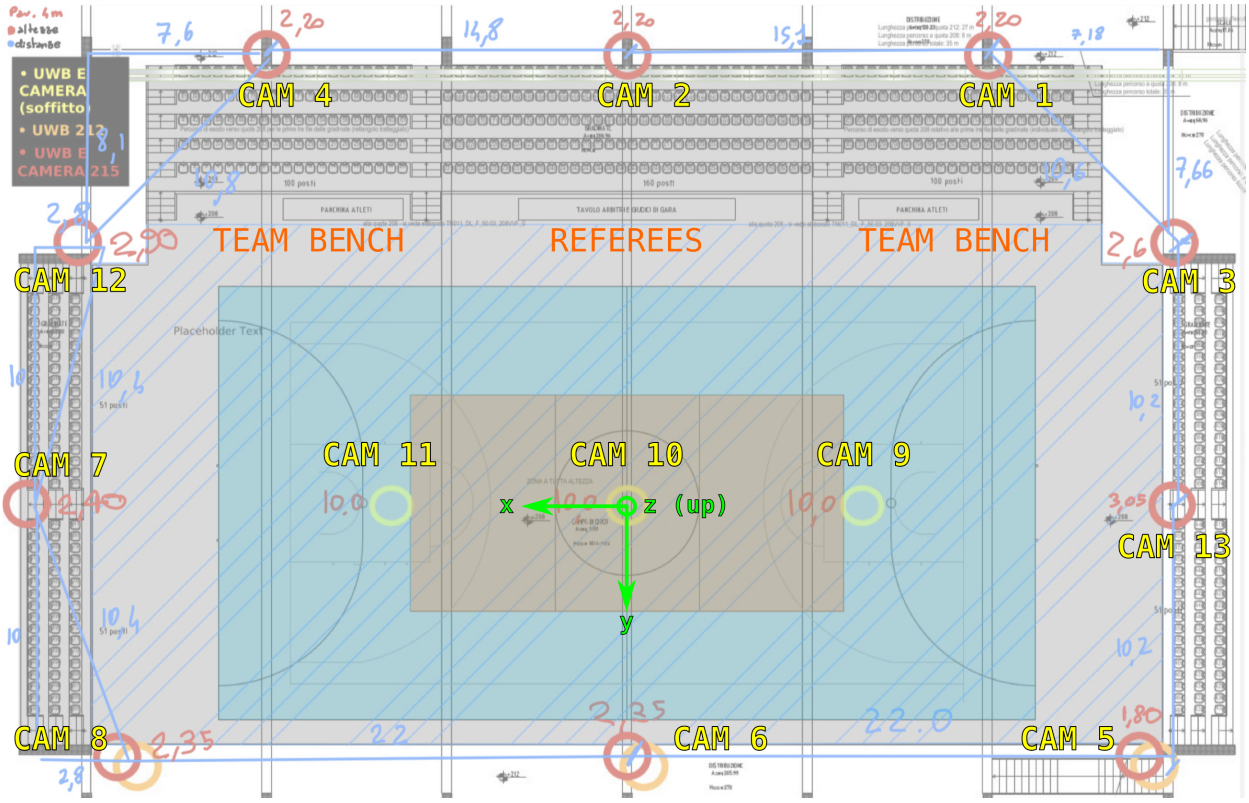
Coding:

- Camera Calibration
- OpenCV 4.10 documentation
- OpenCV Fisheye

Useful links:

- FIBA Rules
- FIVB Rules

5.2 Camera Positions



Camera positions retrieved from drawing

Cam	x	y	z
1	-15.10	-17.90	6.20
2	0.00	-17.90	6.20
3	-22.30	-10.20	6.60
4	14.80	-18.10	6.20
5	-22.00	10.20	6.80
6	0.00	10.20	6.35
7	24.80	0.00	6.40
8	22.00	10.00	6.35
12	24.80	-10.00	6.90
13	-22.00	0.00	7.05

INDEX

R

S

T

V