Study report

of the course Information Technology

at the Baden-Wuerttemberg Cooperative State University Mannheim

---

Subject

Robust On-Board Image Recognition for Autonomous Robot-Ball Interaction

---

Rebekka Litzelmann and Michael Ratzel

20.06.2020

Processing period:	14.10.2019 - 20.06.2020
Student id, course:	6079589 6614999, TINF17ITIN
Supervisor:	Prof. Dr. Jochem Poller
Signature of supervisor	

_____

Declaration

We hereby assure you that we have written our study report on the

Subject

Robust On-Board Image Recognition for Autonomous Robot-Ball Interaction

independently and that we have not used any other sources and aids than those indicated.

We also assure you that the electronic version submitted is the same as the printed version.*

* if both versions are required.

Mannheim, 20.06.2020

Mannheim, 20.06.2020

Abstract

In the Small Size League (SSL) soccer competition of the Robot World Cup (RoboCup), the TIGERs Mannheim team uses an image recognition system that recognises the ball directly with cameras on the robots. This reduces the reliance on the league-internal vision system. Based on a previous work, this study report aims at centralising the configuration system and enhancing the detected ball position using undistortion and backprojection. The centralisation avoids deviating configuration states in the multi-robot environment. The former configuration system is modified and extended to the robots' Firmware by introducing a new concept. An undistortion model with few parameters is successfully generated and trained. It combines high performance and accuracy. Additionally, the detected ball position is projected back into the three-dimensional space, which contributes to a more effective control of the robot.

The results of this study report reflect an improvement in the overall robustness of the image recognition, although the new system has yet to prove itself in a competitive tournament environment like RoboCup.

Zusammenfassung

In der Roboterfußballweltmeisterschaft der Small Size League des RoboCup kommt beim Team der TIGERs Mannheim ein Bilderkennungssystem zum Einsatz, das den Ball direkt mit Kameras auf den Robotern erkennt. Dadurch wird die Abhängigkeit von dem ligainternen Bildverarbeitungssystem reduziert. Aufbauend auf einer vorangegangenen Projektarbeit zielt dieser Studienbericht darauf ab, das Konfigurationssystem zu zentralisieren und die erkannte Ballposition durch Entzerrung und Rückprojektion zu verbessern.
Die Zentralisierung vermeidet abweichende Konfigurationszustände in der Multi-Roboter-Umgebung. Das frühere Konfigurationssystem wird modifiziert und durch die Einführung eines neuen Konzepts auf die Roboter-Firmware ausgedehnt. Ein Entzerrungsmodell mit wenigen Parametern wird erfolgreich erzeugt und trainiert. Es kombiniert hohe Leistung und Genauigkeit. Zusätzlich wird die erfasste Ballposition zurück in den dreidimensionalen Raum projiziert, was zu einer effektiveren Steuerung des Roboters beiträgt.
Die Ergebnisse der Projektarbeit spiegeln eine Verbesserung der allgemeinen Robustheit des Bilderkennungssystems wider, wobei das neue System sich erst noch in einem wettbewerbsorientierten Turnierumfeld wie dem RoboCup bewähren muss.

Contents

List of Figures

List of Tables

Listings

Acronyms

AI	Artificial Intelligence
CNN	Convolutional Neural Network
COM	Communication Port
CPU	Central Processing Unit
FOV	Field Of View
GUI	Graphical User Interface
MMAL	Multimedia Abstraction Layer
OpenCV	Open Source Computer Vision Library
RMSE	Root-mean-square Error
SSL	Small Size League
TIGERs	Team Interacting And Game Evolving Robots
UART	Universal Asynchronous Receiver/Transmitter
USB	Universal Serial Bus
YAML	YAML

1. Introduction

The work presented here has been created in the context of a project organised and maintained by students of the Cooperative State University Mannheim. In the project, which is called Team Interacting and Game Evolving Robots (TIGERs), students mainly develop and program robots to participate in the Small Size League (SSL) of RoboCup, a world championship in robot soccer.
The following chapters are an introduction to these topics and provide a general overview necessary for this thesis' conclusions.

1.1. RoboCup

The Robot World Cup Initiative, chiefly referred to as RoboCup, was established in the 1990s as the next big long term challenge in robotics and Artificial Intelligence (AI). Leading scientists declared the following "ultimate goal":

> By the middle of the 21st century, a team of fully autonomous humanoid robot soccer players shall win a soccer game, complying with the official rules of FIFA, against the winner of the most recent World Cup [24].

With soccer as the motivation to push research in robotics and AI, the first international RoboCup was held in 1997 in Nagoya, Japan. Over 40 teams participated in different leagues [2]. RoboCup has grown into a well-established institution in robotics and promotes the development of advanced technologies in robotics ever since. Worldcup competitions and conferences are held annually, with the "dream" mentioned above as constant companion.

In the highly dynamic games usually not all environmental information is available, state information must be gathered by utilizing sensor data [24].

The RoboCup covers several leagues that range from humanoid robot soccer and simulated soccer to non-humanoid robot soccer, and also others like logistics, @Work, @Home and disaster rescue.

1.2. Small Size League

One of the non-humanoid leagues in RoboCup Soccer is the SSL. This league is one of the oldest soccer leagues at RoboCup and is divided into two divisions. Figure 1.1 shows a snapshot of a game in the SSL division A.

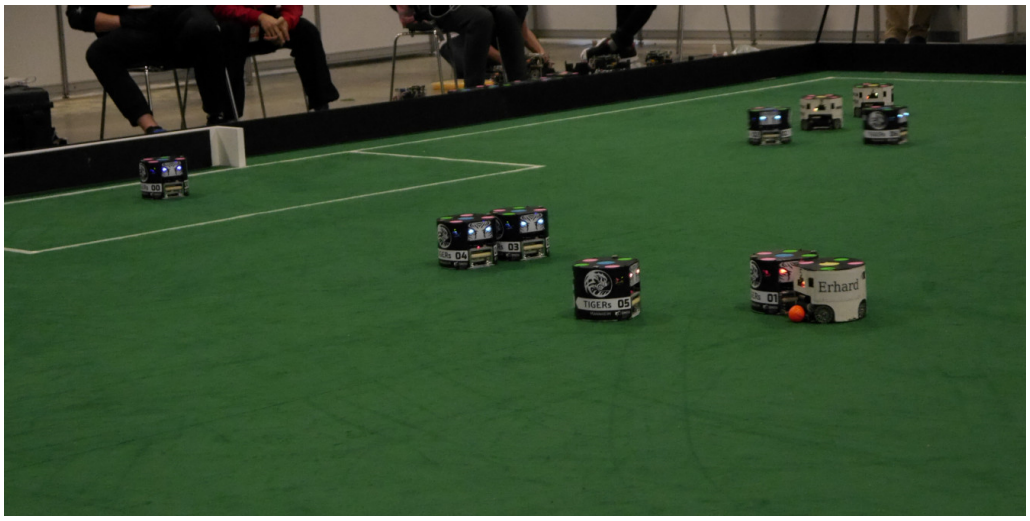


Figure 1.1.: RoboCup Soccer SSL Division A Game

In a game in division A, eleven robots per team play on the 12m x 9m carpeted field. Located above the playing field, a league-internal vision system is used by each team to receive information about the position of the robots, the position of the ball and field lines as well as the colours of the robots to identify the team. The identification is made possible by colour patterns on top of the robots, where either blue or yellow defines the team colour [29]. The data from the vision system is sent to each team's

private computer system next to the playing field, which processes the position data and sensor data of its own robots. Each team develops its own AI to send commands to the robots via wireless communication and control their behaviour.
During the ten minutes of regular playing time, the teams aim to score more goals than the opposing team [29].

The rules for Division B are similar, with minor adjustments such as the number of robots on the field and the field size.

The SSL is the only league in RoboCup Soccer that uses a shared vision and a hybrid centralized/distributed robot control system [27].

1.3. TIGERs Mannheim

TIGERs Mannheim is the robot soccer team of the Cooperative State University Mannheim. The acronym stands for **T**eam **I**nteracting and **G**ame **E**volving **R**obot**s**. The team, which was founded in 2009 by students of information technology, participates in the RoboCup SSL since 2011 annually. By 2014, the team achieved being in the Top 8, the European championship was won in 2016 and the the team celebrated its biggest success in 2018 by being placed 3rd in the worldwide competition and being awarded multiple awards, the Excellence Award for the overall performance and representation of the RoboCup goals being one worth mentioning especially. Other honours include the award for the best Team Description Paper, the Open Source Award and the Technical Challenges. One of the team's robots in the recent generation 2019 can be seen in Figure 1.2.

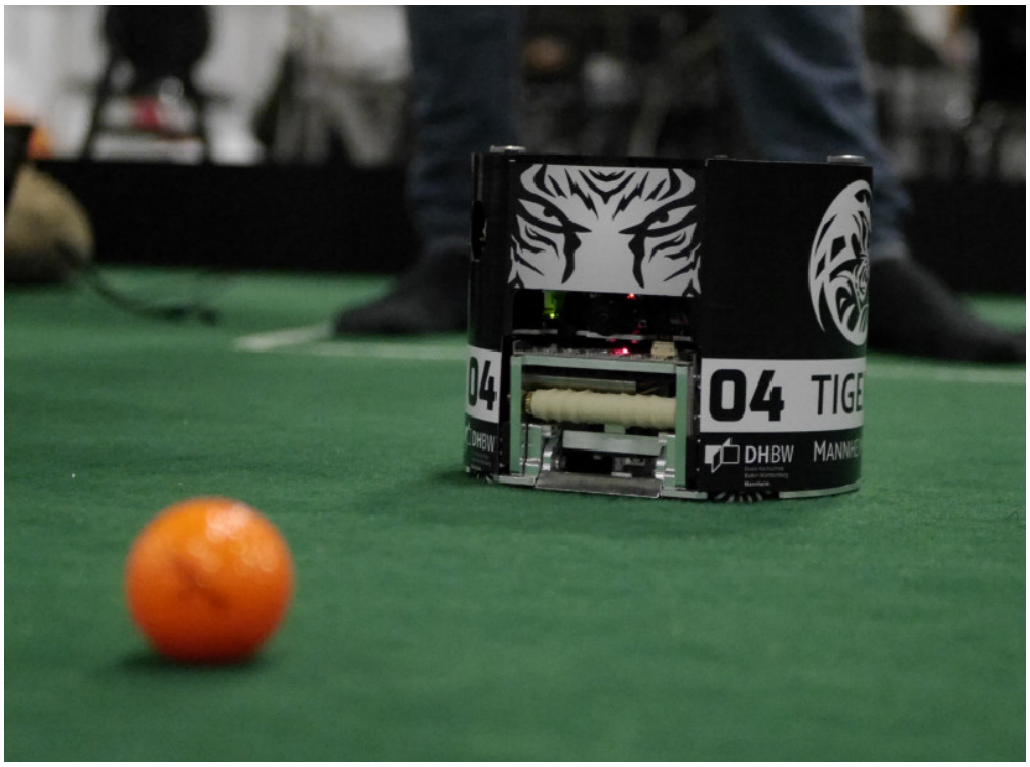


Figure 1.2.: TIGERs robot in generation 2019 at RoboCup 2019 in Sydney, Australia

1.4. Aim of this work

This work focuses on using the Raspberry Pi Camera Module V1 used on the TIGERs robots. A basic approach to detect the ball with this camera was implemented in a previous work. The aim is to refine this software to increase the robustness of the solution. Therefore, typical problems like diverging configurations on multiple bots or distortion errors of the detection itself are tackled. Furthermore, the implementation of a ball backprojection solution is part of this work. It aims to use a three dimensional position and increase the control possibilities.

2. Current Hardware and Software Architecture

The project of centralising the configuration system for the image recognition system focuses on an improvement of the configuration system for the TIGERs ball detection software and therefore is a pure software work. However, many components are touched by the ball detection configurations, so the basics of the hardware as well as the relevant software parts are provided in this chapter.
Current hardware involves the robot in the generation of 2019 with all of its components, including a Raspberry Pi and the Raspberry Pi Camera Module and the TIGERs `Base Station`. The software components contain the central AI, the ball detection software and the robot's Firmware.

2.1. TIGERs Robots

The architectural design of the TIGERs robots in the generation v2019 is shown in Figure 2.1. All of the design and mechanics have been engineered by the team itself and are highly specialized to meet the requirements for robot soccer in the SSL.

Four omnidirectional wheels, a combined kicking and chip-kicking unit as well as a dribble unit are the main parts for the robot to be able to play soccer. A camera located in the front is one of many sensors, which include motor feedback, the infrared barrier of the dribbling unit and a gyroscope [25].
The sensor inputs, including the one for the camera, are connected to the robot's

mainboard, where the robot is controlled from. The processing is based around one main microcontroller, this main processor is an STM32H743 [25].

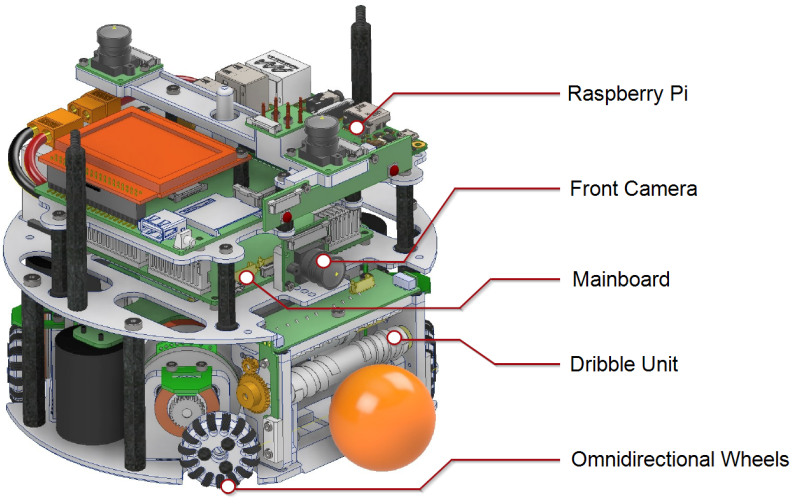


Figure 2.1.: 2019 generation of the TIGERs robot

The on-board front camera provides "local vision information" [32] which is processed on a Raspberry Pi integrated in the robot's design and connected to the mainboard via Universal Asynchronous Receiver/Transmitter (UART).

2.2. Raspberry Pi and Raspberry Pi Camera Module

Raspberry Pi is a "low-cost" [23], single-board Linux computer often used in education [23] and private automation projects as well as Internet-of-Things-applications. The Raspberry Pi 3 A+ contains a single chip system with an ARM-Microprocessor and interfaces for Universal Serial Bus (USB), ethernet, a camera module and a general purpose input/output pin, two of which offer UART functionality [32].

As mentioned above, the robot in its version 2019 has a Raspberry Pi 3 A+ mounted onto it. This single-board computer has the first generation of the Raspberry Pi Camera Module with a wide angle lense connected to it [32]. The camera module was released in 2013 [14] and is specialized for use on Raspberry Pi computers [32].

It is connected to the Raspberry Pi via a flex cable which inserts into the connector situated between the Ethernet and HDMI ports [34], see Figure 2.2.

Figure 2.2.: Raspberry Pi 3 and the Raspberry Pi Camera Module [12]

With a native resolution of 2592 x 1944, the 5MP sensor can also operate in different modes. The mode used is Mode 4, which applies a resolution of 1296 x 972, the aspect ratio 4:3, frame rates from 1-42fps, full field of view (FOV) and 2x2 binning [14].

Communication Raspberry Pi and Mainboard Communication between the mainboard and the Raspberry Pi takes place via a UART interface. This standard serial communication protocol only uses two wires to transmit data between devices [1], which is useful in a rather small robot where not a lot of space is desired for wiring. UART also is a "well documented and widely used method" [1]. The limited data frame size of 9 bits [4] is not a problem for the communication between the Raspberry Pi and the mainboard, as small packages are sent 60 times per second. The packages sent from the mainboard to the Raspberry Pi only contain a shutdown command in the previous version of the `balldetector`, while the package in the opposite direction contains information about so-called `ball candidates`, more about this package of information in Section 2.6.

2.3. Base Station

Another hardware component in the setup used for improving the configuration system of the `balldetector` is the `Base Station`, as seen in Figure 2.3. This device is used for communication between Sumatra, the main AI of the TIGERs, and their robots. Via a self-contained protocol and with a frequency range of 2.300 - 2.555GHz data is sent wirelessly to the robots, while the `Base Station` is connected to a computer or laptop via ethernet (100MBit/s) [30]. Commands like positions where the robots should move to are sent from Sumatra to the robots using the `Base Station`. This makes gameplay possible in the SSL without humans interfering on the field or on any device. The robots only are controlled by the team's AI.

Figure 2.3.: The TIGERs Base Station

2.4. Sumatra

Sumatra is the central software of the TIGERs robot soccer team and is written in the programming language Java. It consists of several modules with different tasks, such as calculating the best moves or processing incoming information from external

sources. One of the external information sources is the robot. Other inputs, for example, include data from the SSL vision system (see Section 1.2). Relevant for the project of developing an improved configuration system for the `balldetector` is the connection to the robot, via the `Base Station` (see Section 2.3) as well as the simulation part of Sumatra. This simulation part consists of a graphical interface where a game can be simulated, different debugging options and options to set values for the algorithms or for the robot are given. The layout graphical interface is depicted in Figure 2.4.

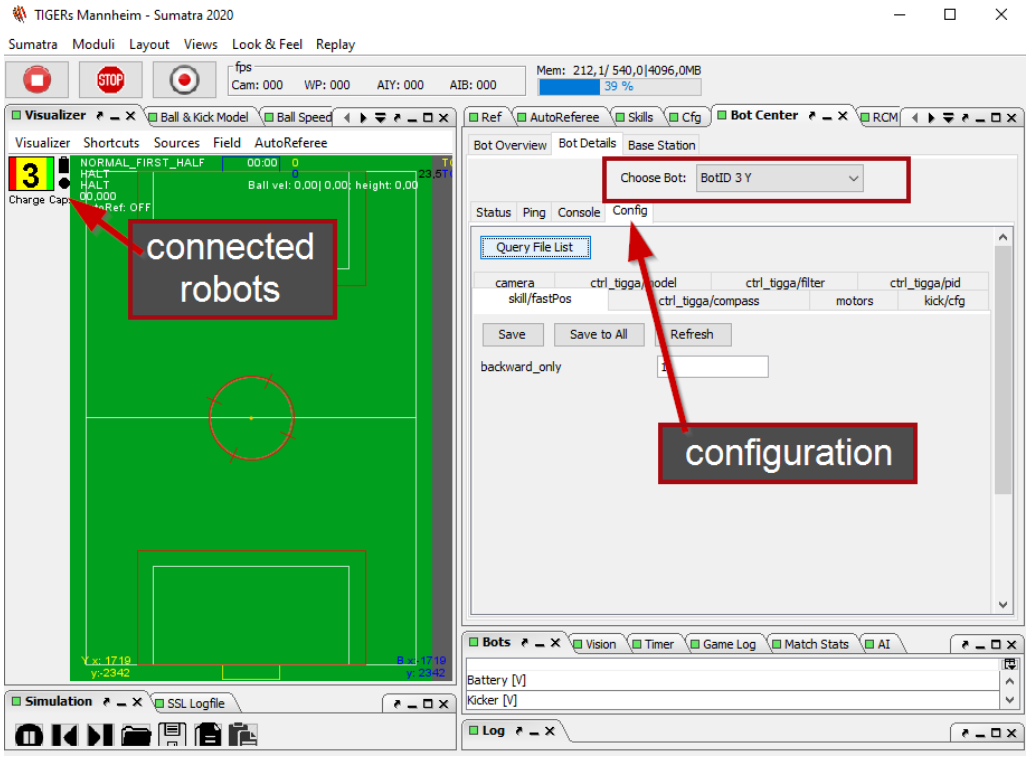


Figure 2.4.: Sumatra - simulation interface

All robots connected to Sumatra via the `Base Station` are shown on the left side of the simulated field, see Figure 2.4. On the right side, a tab "Bot Center" shows all robot-related information and holds the option to change the configurable values for the robot. For changing these configurations, a robot is to be selected ("Choose Bot").

In the tab "Config" several subtabs show the different robot-related configurations, for example for the motors that drive the omnidirectional wheels.

2.5. Firmware

Firmware generally is a "software program or set of instructions programmed on a hardware device." [13] The TIGERs robots need this collection of data and code for communicating with other computer hardware on the robot itself. This includes converting analog sensor signals to digital data or managing communication protocols [10], in the context of the TIGERs robots this may be controlling the motors for the wheels or, more importantly in the case of this report, the Raspberry Pi and its camera module.

The Firmware software itself is written in the high-level programming language C, which in the fast growing market of embedded systems and microcontroller programming has become quite common [19]. The operating system ChibiOS is used [25], which separates different functions into tasks. These tasks are executed independently and priorities can be assigned to influence the scheduling of the respective tasks. In the Firmware, every aspect of the robot is represented by its own task.

In the context of this study report the TIGERs Firmware in its version `main2019` for the mainboard is used. The existing `ExtTask` for communication with external robot exponents like the Raspberry Pi is the most important task for the implementation of an improved `balldetector` configuration system. The command line interface of the `Robot Console` are to be extended by the implementation as well.

2.6. Balldetector: The Image Recognition Software

A previous work based on the TIGERs project has introduced a software component that uses the camera on the robots to detect the standard orange ball used as the robot soccer ball in the SSL. Figure 2.5 shows the robot's view with a ball detected. Moreover, the `balldetector`, as this software is referred to, contributes to the robot intercepting the ball. This manoeuvre is very helpful in a game of robot soccer, where precise ball handling and passes are the key to an optimal game and was part of the SSLs technical challenge at the RoboCup 2019 [8].

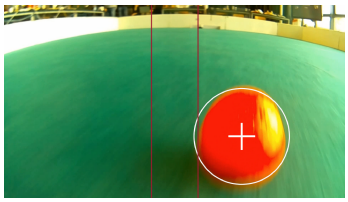


Figure 2.5.: The robot's view and a ball detected

The software, which is written in the programming language C++, analyses the incoming video stream from the Raspberry Pi Camera Module for round orange spheres. A value "confidence" is calculated and represents at which percentage this sphere can be considered an orange golf ball. The processing is perfomed on the Raspberry Pi, which then sends up to ten "ball candidates" and their confidence value to the robot's Firmware via UART [32]. The further handling of this information and interception of the ball is covered by the Firmware and the central AI Sumatra. The general data flow is illustrated in Figure 2.6.

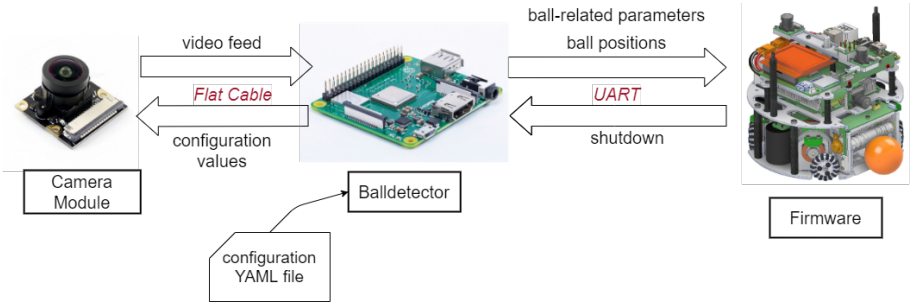


Figure 2.6.: Data flow diagram of the previous configuration system

3. Improved Configuration System

This chapter is dedicated to the improvement of the existing configuration system for the ball detection software. The architecture of the previous system is outlined, the new concept introduced and its implementation as well as the results are presented.

3.1. Previous Balldetector Configuration System

The configurations for the ball detection software were previously handled using YAML files. YAML is a "human-friendly, cross language, Unicode based data serialization languages" [17].

These files contained the configuration in written form and were included in the ball detection software project. Changing a configuration value required making changes to the file and resulted in the software restarting[32]. Several files were defined in the previous version of the `balldetector`, such as a default and a debug configuration file. New files could be defined for additional predefined configuration settings and had to be added in with the other defined files.

3.1.1. YAML File Structure

Three main sections can be defined for the files' structure: `debug`, `camera`, and `detector`. The file in Listing 3.1 shows the values for the default configuration. The first section `debug` includes configurations for debugging the software. stdOut, for example, stores a boolean value which defines if output is shown via the command line on the Raspberry Pi. The other values in this section define whether and how

the camera's frames and information about detected balls are shown to the user. This is the case when the Raspberry Pi is connected via SSH, for example.
The values in the section `camera` characterise how the camera module is initialised. All of the camera configurations are specified by the Raspberry Pi Foundation [14]. Lastly, the `detector` section stored a value for the `balldetector` method, which either can be scanLines or cmVisionScanLines. Configuration values for the two in turn are set in the respective subsections.

For the values exposureMode (Listing 3.1 l.12), awbMode (l.13) and imageEffect (l.21), as well as for the detector method (l.25), linesDirection (l.30), lineDistribution (l.31) and maskBase (l.37) multiple options are available. For the camera-related configurations, these options are specified by Multimedia Abstraction Layer (MMAL), a C library for use with taking videos from the Raspberry Pi [3].

Listing 3.1: YAML file structure

```
1  debug:
2      stdOut: false
3      show: false
4      sendInterval: 1
5      resolution: 720
6      balls: true
7      channel: "uv"
8      lines: true
9  camera:
10   iso: 400
11   shutter_speed: 10000
12   exposureMode:
13   awbMode:
14   awb_gains_b: 2.0
15   awb_gains_r: 1.0
16   brightness: 50
17   saturation: 100
18   sharpness: 0
19   contrast: 0
20   exposureCompensation: 0
21   imageEffect:
22   hflip: true
23   vflip: true
24 detector:
25   method: scanLines  // cmVisionScanLines
26   scanLines:
27     firstLineAt: 1
```

```
28      lastLineAt: 90
29      numLines: 60
30      linesDirection: "up"    //"down"
31      lineDistribution: "quadratic"  //"linear" "log"
32      edgeDetectionThreshold: 35
33      minConfidence: 20
34      minDistBetwBalls: 15
35      shadowFactor: 0.6
36    cmVisionScanLines:
37      maskBase: "color"  //edges
38      colorThresh: 50
39      heightFactor: 0.75
```

3.1.2. Software Workflow

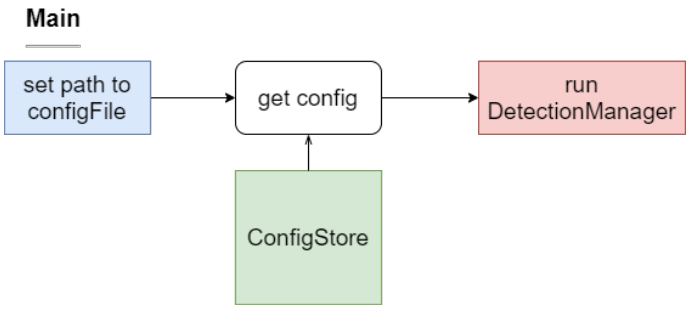


Figure 3.1.: Workflow diagram of the start of the application

The `balldetectors` data workflow can roughly be devided into two workflows. The first, as illustrated in Figure 3.1, includes the process of starting the execution of the software. As a first step, the YAML configuration files are loaded from the `ConfigStore`. This `ConfigStore` also is the method of choice for storing and updating the configuration settings after the YAML file is changed. Next in the workflow, after getting the path to the configuration file, the main program `DetectionManager` is started.

Its workflow includes the main ball detection algorithm and, concentrated on the configuration system, is shown in Figure 3.2. Especially the first components that are called when the `DetectionManager` starts, reference the `ConfigStore`, in the Figure represented by dashed arrows. While a `ChangeListener` permanently "listens" for

changes in any configuration setting, the programm's process continues, sets the debug configurations from the respective part of the YAML file, initializes the camera module and here, also sets the respective configurations. The `DisplayManager` shows debug information to the user.

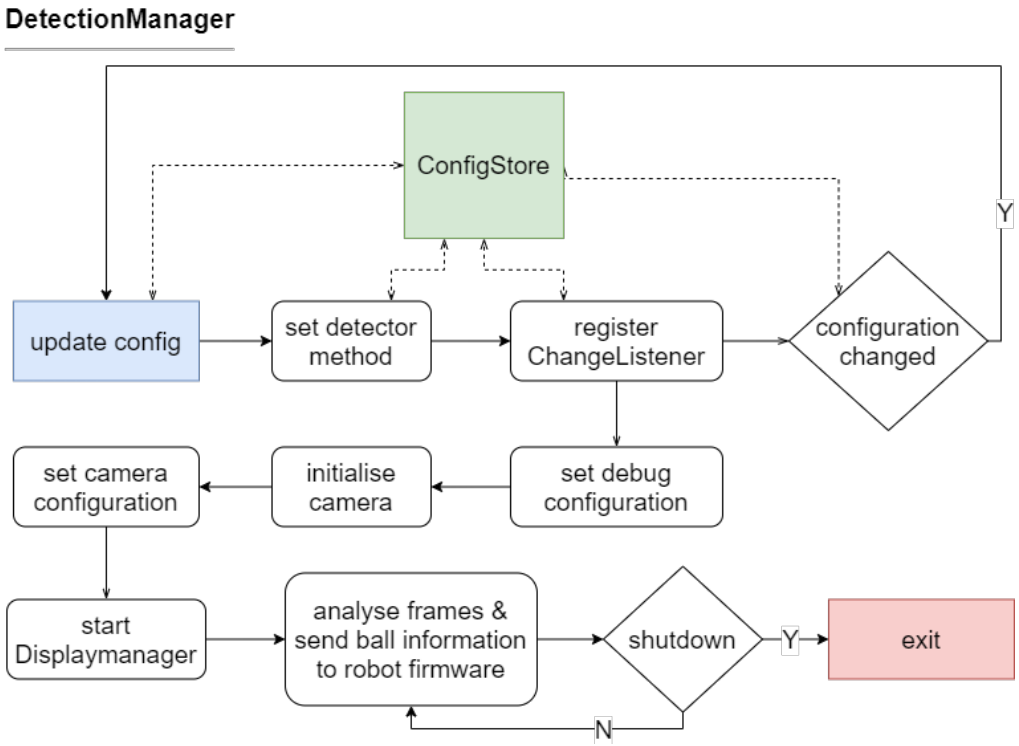


Figure 3.2.: Workflow diagram of the DetectionManager

The ball detection algorithm itself analyses every frame of the incoming video stream from the camera module and initialises the UART packages to be sent to the mainboard. At last, if a shutdown command has been invoked, the software exits the execution.

3.2. Motivation

As mentioned above, in the previous version of the `balldetector` software the configuration system involves YAML files and a `ConfigStore`. This setup generally

allows changing the camera configurations but has one major disadvantage: Each time a change is made, it is necessary to edit the configuration file and restart the software.
The rather complicated process of the previous configuration system is another reason to try and find a simpler and, more importantly, centralised solution: While YAML is "broadly useful for [...] configuration files" [17], in the case of using it in the `balldetector` other aspects need to be considered as well. This includes the other software parts, Sumatra and the Firmware, as well as the usability during testing the robots' behaviour and software, or a game in the SSL.

One major concern when using the `balldetector` during a game of robot soccer is the deployment of the exact same configurations to every robot. With the previous version of the configuration system, this had to be done manually by connecting to each robot, modifying the YAML file, restarting the `balldetector`, then on to the next robot. Scaled up to the 11 robots in a game, this is a task that cannot be done efficiently using the previous `balldetector`s configuration system.

This problem is resolved by the implementation of an improved `balldetector` configuration system.
The improved configuration system should furthermore include the option of easily changing configuration values, also single values, without needing to restart the software and reinitialising all sections of the `balldetector` configurations.

3.3. Concept

Developing a concept is the first step of implementing an improved solution into a software project [22]. For the new configuration system, it was decided to modify the previous structure and extend the configuration system to involve the Firmware. The revised data flow for the `balldetector` is illustrated in Figure 3.3. The components changed are coloured blue. As shown, the use of YAML files is eliminated completely, and the permanent store of the configuration is moved to the Firmware, where a data structure holds all the values. The configuration then is sent to the Raspberry Pi periodically via the existing UART interface, where it is stored locally and here in turn is processed. From here, the camera configurations are sent to the camera module the same way they were in the previous version.

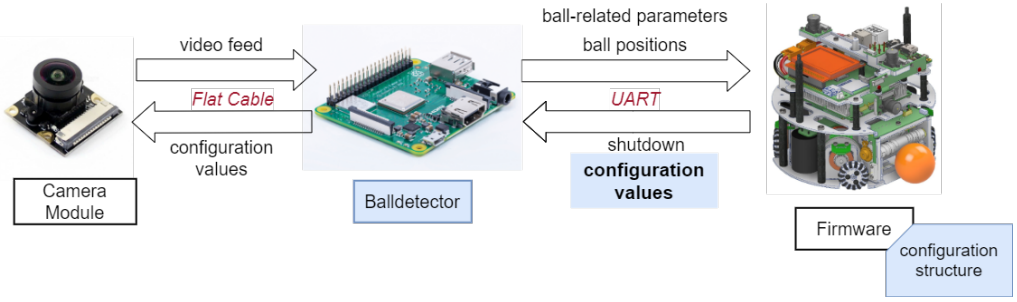


Figure 3.3.: Data flow diagram of the improved configuration system

For making changes to the configuration values, two options are involved in the concept of the new `balldetector` configuration system. One includes the simulation interface of Sumatra, the TIGERs' central AI. Here, the `balldetector` configurations are added to the existing configuration system that connects the Firmware and Sumatra. Values like settings for the kicker unit or the motors' configurations can be modified from the simulation interface in Sumatra. The other option involves the `Robot Console`, which can be accessed by connecting a robot to a computer system, a laptop for example, via a USB cable and using the serial Communication port (COM) interface to access the command line interface. A number of commands are already defined for the same robot-related settings that can be changed in Sumatra,

so this was a preferred and appropriate place for the `balldetector` configuration too.

The following two sections describe the two options of changing values in the improved configuration system in detail and explain their usage. The closing section of this chapter deals with the modifications in the `balldetector` software.

3.3.1. Sumatra

The main advantage of including the `balldetector` configurations in the existing `Config System` in the Firmware is that they are shown and can be modified in Sumatra. From the graphical simulation interface, configuration values can be changed for all robots that are connected to Sumatra via the `Base Station`. In Figure 3.4 the affected components and their interaction, respectively their connections, are displayed. In the new concept, the connection between the Firmware and Sumatra is used for `balldetector` configuration values too. This option is useful when one or multiple robots are connected to the `Base Station`, e.g. in a game of robot soccer or for testing.

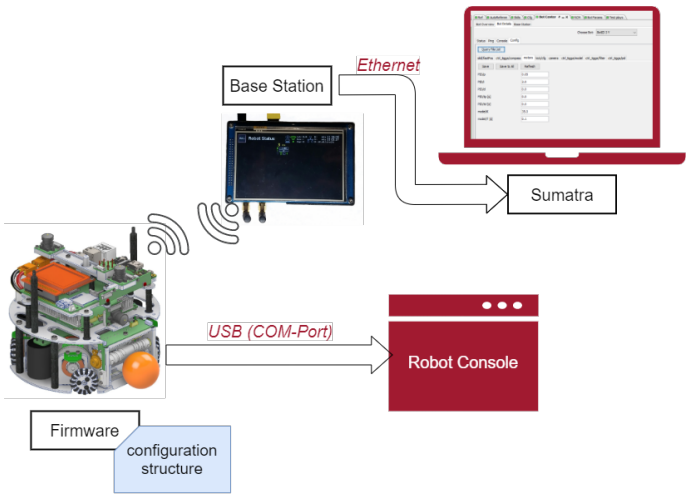


Figure 3.4.: Two options to change the configuration values

3.3.2. Robot Console

Another approach for an option to change configuration settings might seem redundant, given the approach as stated above is implemented correctly and working. However, it has to be considered that for most of the time, the TIGERs robots are not on a field and in a game but rather in the laboratory or in use for tests, debugging, or implementation of another feature. For these use cases, there might not be a `Base Station` involved as not needed. Although, with only the robot as hardware around, `balldetector` settings or settings for the camera module might still need to be changed. Typing a command clearly is not only easier but also time-efficient when needing to make changes quickly, so this approach takes on the implementation of allowing user inputs on the command line to make changes in the ball detection software.

The `Robot Console` can be accessed via the COM interface by connecting the robot to a laptop or computer with a USB cable. This relation is also illustrated in Figure 3.4. The Robot Console and the commands defined here also are part of the existing `Config System` of the Firmware.

3.3.3. Ball Detector

Storing the `balldetector` configurations in a structure in the Firmware and adding it to the Firmware's `Config System`, which allows for changing the values from Sumatra or the `Robot Console` is only half of the concept. The connection from the Firmware to the `balldetector` and the correct processing of the configuration is equally important. Here, the previous configuration system can partly be used. The `ConfigStore` in connection with the `ChangeListener` already offers the functionality that values can be changed on-the-fly. This is kept in the improved version of the configuration system.
What has to be modified however is the use of YAML files and the processing of the values also needs to be adapted to the structures defined in the Firmware. The general workflow of the `DetectionManager` and the `Main` workflow (see Figure 3.1 and Figure 3.2) remains untouched for the most part.

3.4. Implementation

The programming work for the implementation of the concept for the improved and centralised `balldetector` configuration system comes down to changes in the code of the Firmware and the `balldetector`. The following chapter explains the changes to the two and gives code examples.

3.4.1. Firmware

In the Firmware, structures were defined to store the configurations. In fact, for each section of the YAML file, one data structure was created. Structures in the programming language C are used to "group a number of related variables together and refer to them by one overall name" [28]. Listing 3.3 shows the structure for the camera configurations, as an example. Compared to the previous `balldetector` configuration section for the camera in the YAML files (see Listing 3.2), this data structure is identical, except that data types have been added. The data types consider the value range for the respective configuration value.

Listing 3.2: previous balldetector camera configuration YAML file

```
camera:
  iso: 400
  shutter_speed: 10000
  exposureMode:
  awbMode:
  awb_gains_b: 2.0
  awb_gains_r: 1.0
  brightness: 50
  saturation: 100
  sharpness: 0
  contrast: 0
  exposureCompensation: 0
  imageEffect:
  hflip: true
  vflip: true
```

Listing 3.3: new balldetector camera configuration structure in C

```
typedef struct __attribute__((packed)) DetectorCameraConfig
{
   uint16_t iso;
   uint32_t shutterSpeed;
   int8_t exposureMode;
   uint16_t awbMode;
   float awbGainsBlue;
   float awbGainsRed;
   uint8_t brightness;
   int8_t saturation;
   int8_t sharpness;
   int8_t contrast;
   uint16_t exposureCompensation;
   uint16_t imageEffect;
   uint8_t hFlip;    // 1=true, 0=false
   uint8_t vFlip; // 1=true, 0=false
} DetectorCameraConfig;
```

One aspect to highlight here are the configuration settings exposureMode, awbMode, exposureCompensation and imageEffect. These were not assigned a value in the YAML file configuration, and now are assigned to a type of integer in the C structure. This is the case because the camera module's driver MMAL expects values in the form of the respective enum. However, the UART package sent from the firmware to the Raspberry Pi only sends numeric data types. The configuration settings mentioned above are represented by integer values in the data structure and therefore need to be converted to the enum value in the `balldetector` later on. Exemplary, the enum for the awbMode is shown in Listing 3.4.

Listing 3.4: AWB parameters for the Raspberry Pi Camera Module

```
/** AWB parameter modes. */
typedef enum MMAL_PARAM_AWBMODE_T
{
   MMAL_PARAM_AWBMODE_OFF,
   MMAL_PARAM_AWBMODE_AUTO,
   MMAL_PARAM_AWBMODE_SUNLIGHT,
   MMAL_PARAM_AWBMODE_CLOUDY,
   MMAL_PARAM_AWBMODE_SHADE,
   MMAL_PARAM_AWBMODE_TUNGSTEN,
   MMAL_PARAM_AWBMODE_FLUORESCENT,
   MMAL_PARAM_AWBMODE_INCANDESCENT,
   MMAL_PARAM_AWBMODE_FLASH,
```

```
    MMAL_PARAM_AWBMODE_HORIZON,
    MMAL_PARAM_AWBMODE_GREYWORLD,
    MMAL_PARAM_AWBMODE_MAX = 0x7fffffff
} MMAL_PARAM_AWBMODE_T;
```

With the structures defined, they need to be sent to Sumatra via the `Base Station`, to the Raspberry Pi via UART and to the `Robot Console`. The actual algorithm for the communication between those components already exits and is handled in the Firmware's `Config System`, so it has been extended by the newly defined structures for the `balldetector` configurations.

The `Config System` also includes commands for the `Robot Console`. Commands for the `balldetector` configurations have been added to implement this option of modifying the configuration settings. The data structures are updated each time a value is changed in the command line interface of the `Robot Console`.

The implementation of the commands in the Robot Console looks like shown in Listing 3.5. The console input is read and compared whether it equals the command "camera sat <value>" for setting the camera's saturation. If this is the case, in the SetSaturation() method the input value `saturation` is validated for a valid range, the configuration structure is updated and the change confirmed to the user.

Listing 3.5: Implementation of setting the camera's saturation from the Robot Console

```
/*
* get the users input and compare if it equals the command for the saturation
*/
if(ConsoleScanCmd("camera sat %i", &i8) == 1)
   {
      ConsolePrint("setting camera saturation to %hd\r\n", (uint16_t)i8);
      SetSaturation(i8);
   }

/*
* method to set the saturation to the new value
*/
void SetSaturation(int8_t saturation)
{
   if (saturation > 100)
      saturation = 100;
```

```
    detectorCamera.config.saturation = saturation;

    ConfigNotifyUpdate(detectorCamera.pConfigFileDetectorCamera);
    ConsolePrint("camera saturation changed to %hd\r\n", (int16_t)
        detectorCamera.config.saturation);
}
```

Displaying the configurations in Sumatra is covered by defining the structure `ConfigFileDesc`, which is part of the `Config System` in the Firmware. Following the example of the `balldetector` camera settings, the definition for the `ConfigFileDesc` is described in Listing 3.6.

Listing 3.6: Data Structure Declaration and Definition of ConfigFileDesc

```
/*
* the general structure of the ConfigFileDesc data structure
*/
typedef struct _ConfigFileDesc
{
   uint16_t cfgId;
   uint16_t version;
   const char* pName; // 60 characters max
   uint16_t numElements;
   ElementDesc* elements;
} ConfigFileDesc;

/*
* the definition of the ConfigFileDesc for the camera settings
*/
static const ConfigFileDesc configFileDescDetectorCamera =
{ SID_CFG_BALLDETECTOR_CAMERA, 0, "balldetector_camera", 14, (ElementDesc[]) {
   {UINT16, "iso", "100, 200, 400, 800", "Iso"},
   {UINT32, "shutterSpeed", "microsec, max:6s", "ShutterSpeed"},
   {UINT8, "exposureMode", "0 - 13", "ExposureMode"},
   {UINT8, "awbMode", "0 - 11", "AwbMode"},
   {FLOAT, "awbGainsBlue", "1.0-8.0", "AwbGainsBlue"},
   {FLOAT, "awbGainsRed", "1.0-8.0", "AwbGainsRed"},
   {UINT8, "brightness", "0-black,100-white", "Brightness"},
   {INT8, "saturation", "-100-100", "Saturation"},
   {INT8, "sharpness", "-100-100", "Sharpness"},
   {INT8, "contrast", "-100-100", "Contrast"},
   {INT8, "exposureCompensation", "-25 - 25", "ExposureCompensation"},
   {UINT8, "imageEffect", "0 - 26", "ImageEffect"},
   {UINT8, "hFlip", "0-false,1-true", "hFlip"},
   {UINT8, "vFlip", "0-false,1-true", "vFlip"}
```

```
}};
```

The communication to external robot components is handled in the `ExtTask` of the robot. Therefore, the communication to the Raspberry Pi also is integrated into this task of the Firmware. In the previous architecture, a shutdown command is sent to the Raspberry Pi (see Figure 2.6). This connection is reused and extended to include the configuration values (see Figure 3.3.

3.4.2. Ball Detector

In the `balldetector` the counterpart for the structures defined in the firmware are implemented. For this counterpart, data structures identical to those in the Firmware are defined. A new C++ file is created for this. The complete file can be found under Appendix A. The individual structures are combined in a superordinate structure `Config`, see Listing 3.7. This is implemented to better mimic the previous structure of the YAML files and to provide a greater context for the single configuration sections. Also, for future implementations, this modular approach is easily extendible.

Listing 3.7: Config: Thhe superordinate data structure

```
typedef struct _Config
{
  _ExtCameraConfig cameraConfig;
  _ExtDebugConfig debugConfig;
  _ExtDetectorConfig detectorConfig;
} Config;
```

The existing `Config Store` from the previous `balldetector` configuration system is redefined. The greatest part of the work is to update the data types and to deconstruct the connection to the YAML files. Every part of the code, where values for the detector itself, e.g. the method, values for debugging or for the camera are set, the newly implemented data structure is referenced instead of the respective part of the YAML file. Enums are defined for the values of the configuration that cannot be represented by an integer value, as explained in Section 3.4.1.

In the case of the camera module's parameters, the new update function looks as follows in Listing 3.8. A variable of type `Config` is created, which is the representation of the data structure that holds all configuration values received via UART. Each parameter for the camera then is set to the respective value from the configuration data structure.

Listing 3.8: Method updateConfig() for the camera parameters

```
void Camera::updateConfig()
{
  Config * pConfig =  ConfigStore::getConfig();
  parameters.ISO = pConfig->cameraConfig.iso;
  std::cout << "Camera iso set to " << parameters.ISO;
  parameters.shutter_speed = pConfig->cameraConfig.shutterSpeed;
  parameters.exposureMode = static_cast<MMAL_PARAM_EXPOSUREMODE_T>(pConfig->
      cameraConfig.exposureMode); // if set to "off" ignores ISO...
  parameters.awbMode = static_cast<MMAL_PARAM_AWBMODE_T>(pConfig->cameraConfig
      .awbMode);
  parameters.awb_gains_b = pConfig->cameraConfig.awbGainsBlue;
  parameters.awb_gains_r = pConfig->cameraConfig.awbGainsRed;
  parameters.brightness = pConfig->cameraConfig.brightness;
  parameters.saturation = pConfig->cameraConfig.saturation;
  parameters.sharpness = pConfig->cameraConfig.sharpness;
  parameters.contrast = pConfig->cameraConfig.contrast;
  parameters.exposureCompensation = pConfig->cameraConfig.exposureCompensation
      ;
  parameters.imageEffect = static_cast<MMAL_PARAM_IMAGEFX_T>(pConfig->
      cameraConfig.imageEffect);
  parameters.hflip = pConfig->cameraConfig.hFlip;
  parameters.vflip = pConfig->cameraConfig.vFlip;

  if (pCameraComponent)
    raspicamcontrol_set_all_parameters(pCameraComponent, &parameters);

}
```

3.4.3. Tests

Testing the improved `balldetector` configuration system involved the following test cases:

These tests can be seen as Unit Tests and were carried out multiple times during the

Purpose	Test Case Description
Communication between Firmware and Sumatra	Change values in Sumatra
Communication between Firmware and Bot Console	Change values from Bot Console
Data types and input	Enter invalid values in both Sumatra and Bot Console
Camera configuration	Change values and read camera's parameter to verify
Debug configuration	Change values and verify console output on Raspberry Pi
`Balldetector` configuration	Change values and verify changes on Raspberry Pi

Table 3.1.: Test Cases

process of programming. Errors could be eliminated quickly so that at the end of the implementation phase, the tests succeeded and were run multiple times to verify the stability of the implemented configuration system.

3.5. Results

The result of this project is the successful implementation of an improved configuration system for the `balldetector`. The improved version is not limited to the use of components included in the ball detection software but rather centralised the configuration system in the robot's Firmware, where the major advantage is the option to deploy the same configuration settings to multiple robots at once. This option is accessed from the graphical simulation interface integrated in Sumatra, as can be seen in Figure 3.5 with the pre-existing button "Save to All". In addition to that, tabs for the three configuration sections of general `balldetector` configurations, debugging and camera settings are defined.

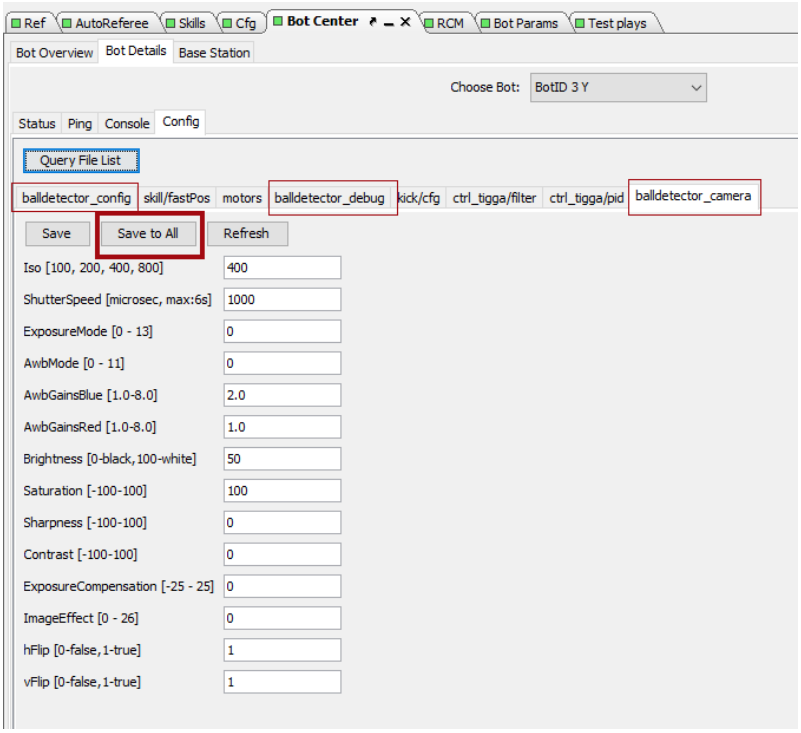


Figure 3.5.: Option to change camera parameters from Sumatra

When a robot is connected to a laptop or computer, the `balldetector` configurations can be changed via the command line interface `Robot Console`. The current settings for the different configuration sections can be viewed with the command "<section> current", as is displayed in Figure 3.6. Here, all commands regarding the camera settings are called with the command "camera current", after all possible commands were called with "help camera". At last, the saturation for the camera module is changed to 90. The command line output confirms the successful change.

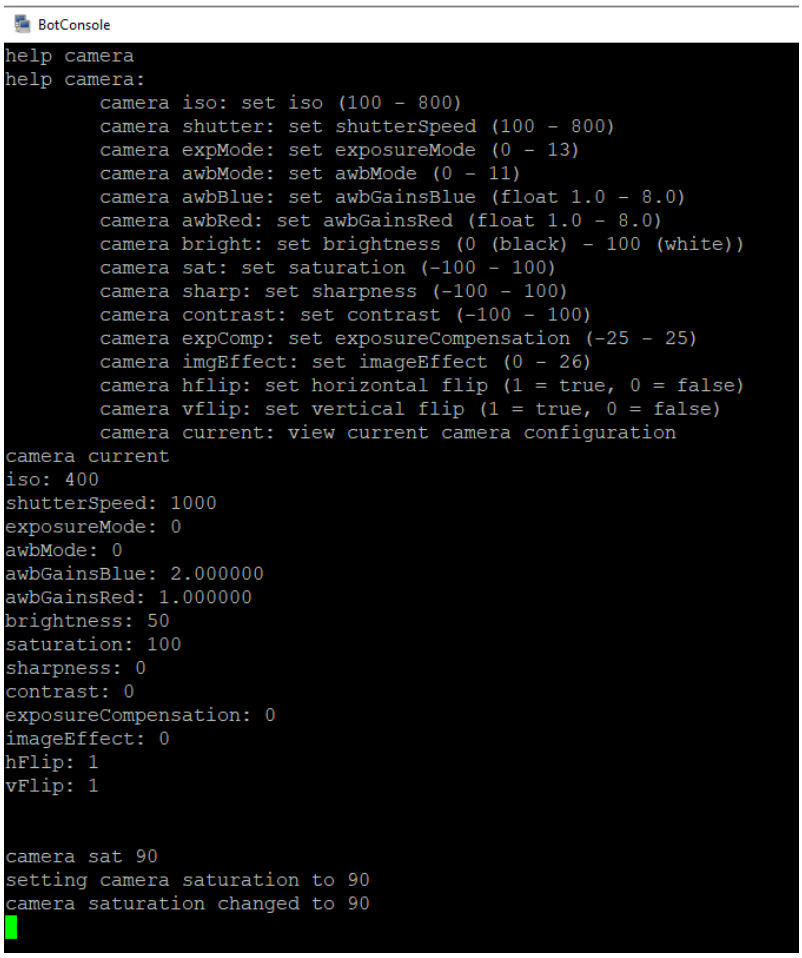


Figure 3.6.: Option to change camera parameters via the Robot Console

In the improved `balldetector` configuration system, the data flow has been redesigned, the permanent configuration storage has been centralized in the Firmware, where it is sent to the Raspberry Pi periodically and from there is processed as before, with small changes to data types and a new structure defined for receiving the configurations from the Firmware.

3.6. Discussion

In the project of improving the `balldetector`s configuration system, many components of the TIGERs software project have been modified. The permanent storage of configuration settings in the Firmware was an approach that has shown great success when it comes to usability when changing values from either the central AI Sumatra or from the `Robot Console`. Regarding the steps of a software development process [22], more testing has to be done to fully determine all implications the changed configuration system has for the TIGERs team, both in internal work and also in a robot soccer game in the SSL. Especially in an event like RoboCup, this implementation is considered very useful and time-saving when it comes to calibrating many robots in a tight schedule.
Maintaining the software, which also is part of a software's development cycle [22], will be covered by future work of the TIGERs team.

Because of the modular structure and the improved framework for sending and storing the `balldetector` configuration that has been developed with the implementation of this work, future additions or substitutions to the improved configuration system can be applied easily.

4. Cross-Compilation

"Embedded computers often lack the necessary resources for developing and compiling software. The Raspberry Pi is rather special in this regard since it already includes the gcc compiler and the needed linking tools (under Raspbian Linux). But while the code can be developed and built on the Raspberry Pi, it may not always be the most suitable place for software development. One reason is the lower performance of the SD card." [15] The limited random access memory (RAM) size of 500MB is another limitation on the used Raspberry Pi 3A+, which slows down the compile process.[5]

Therefore a cross-compiling toolchain is one of the first targets to achieve faster development speed by faster compilation time. Programs build on a normal x86 architecture aren't compatible with the ARM-architecture of the Raspberry Pi processor. A special compiler is necessary which runs on x86 and produces ARM programs, as well as the respective ARM libraries. Crosstool-NG is a generator for such toolchains. It builds the GNU compiler collection (GCC) and the standard libraries necessary to build a standard C/C++ program.[18] With this environment, all dependencies of the final executable could be cross-compiled. In this case, there are over 300 dependencies. The work of building them by hand can be avoided by using the libraries installed on the Raspberry Pi and and to copy them onto the host pc afterwards. An additional advantage is, there are no compatibility issues between different release versions of the handcrafted library and the library installed on the Pi. To configure the build process and use the cross-compiling toolchain CMake is used. It offers the usage of Toolchain Files to configure the cross-compilation while still providing native compilation functionality directly on the Pi.[7] A tutorial to create and use the toolchain can be found on the TIGERs Mannheim GitHub.

5. Camera Model

In principle, cameras project three-dimensional world points onto a two-dimensional image. This is modeled in three consecutive steps. Transforming the world position P_w into the camera space position P_c, projecting the point onto the image plane and applying the intrinsic camera matrix to account for the optical properties of the camera. [20] Equation (5.1) represents this procedure, where $P_w = [X_w, Y_w, Z_w, 1]^T$ is the position in the world, and $p = [u, v, 1]^T$ is the two-dimensional position on the picture in pixels. K is the intrinsic camera matrix and the 3x4 matrix $[R|t]$ represents the extrinsic parameters, while s is an arbitrary scaling factor. Take note both points are represented with homogeneous coordinates, which is not further explained here. To achieve a more readable text, the homogeneous is also often dropped. [16]

$$sp = K[R|t]P_w \tag{5.1}$$

$$s \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \end{bmatrix} \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix}$$

5.1. Transformation Into Camera Space

The extrinsic parameters R and t encode the homogeneous transformation from world space into camera space and compose a 4x4 transformation matrix. R hereby represents a rotation and t a translation. This matrix is multiplied with the point

P_w in world space to transform it into P_c in camera space, see equation (5.2). [20]

$$P_c = \begin{bmatrix} R & t \\ 0 & 1 \end{bmatrix} P_w \tag{5.2}$$

$$\begin{bmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix}$$

5.2. Projection onto the image plane

A 3x4 projection matrix is required to project P_c onto the image plane. In equation (5.3) an additional normalization with $x_1 = X_c/Z_c$ and $y_1 = Y_c/Z_c$ is applied. [20]

$$Z_c \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{bmatrix} \tag{5.3}$$

Equation (5.4) is the combination of the transformation and projection of P_w to calculate the normalized representation of P_c in one single step with the matrix $[R|t]$. [20]

$$Z_c \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \end{bmatrix} \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix} \tag{5.4}$$

5.3. Transformation Within the Image Plane

To account for the intrinsic parameters of the camera, the transformed projected and distorted point is transformed a second time within the image plane by using the matrix K.
The focal length F is the distance between the focus point and the Image plane. That's the distance between the aperture and the image plane in a lens-free pinhole camera model.
The size of the pixels on the sensors in world units (e.g., mm) are p_x, p_y. They can be achieved by dividing the size of the sensor with the number of pixels. And they are often combined with the focal length to create $f_x = F/p_x$, $f_y = F/p_y$.
The parameters c_x, c_y in pixels moves the optical center of the image to the given position in the picture. This shift has extra importance with distortion, due to the influence of the distance to the optical center on distortion. [20]

$$K = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \tag{5.5}$$

5.4. Distortion Model

Distortion is an optional step in this model. It can be applied, to increase the precision. The normalized and projected point P_c is distorted with the model in equation (5.6).

$$\begin{aligned} \begin{bmatrix} x_2 \\ y_2 \end{bmatrix} &= \begin{bmatrix} x_1 \frac{1+k_1 r^2+k_2 r^4+k_3 r^6}{1+k_4 r^2+k_5 r^4+k_6 r^6} + 2p_1 x_1 y_1 + p_2 \left(r^2 + 2x_1^2\right) + s_1 r^2 + s_2 r^4 \\ y_1 \frac{1+k_1 r^2+k_2 r^4+k_3 r^6}{1+k_4 r^2+k_5 r^4+k_6 r^6} + p_1 \left(r^2 + 2y_1^2\right) + 2p_2 x_1 y_1 + s_3 r^2 + s_4 r^4 \end{bmatrix} \\ &= \delta(x_1, y_1) \end{aligned} \tag{5.6}$$

$$r^2 = x_1^2 + y_1^2 \tag{5.7}$$

$$\delta(x,y) = \begin{bmatrix} x\delta^{(r)}(r) + \delta_x^{(t)}(x,y) + \delta_x^{(s)}(x,y) \\ y\delta^{(r)}(r) + \delta_y^{(t)}(x,y) + \delta_y^{(s)}(x,y) \end{bmatrix} \tag{5.8}$$

$$\delta^{(r)}(r) = \frac{1 + k_1 r^2 + k_2 r^4 + k_3 r^6}{1 + k_4 r^2 + k_5 r^4 + k_6 r^6} \tag{5.9}$$

$$\delta_x^{(t)}(x,y) = 2p_1 x_1 y_1 + p_2 \left(r^2 + 2x_1^2\right) \tag{5.10}$$

$$\delta_y^{(t)}(x,y) = p_1 \left(r^2 + 2y_1^2\right) + 2p_2 x_1 y_1 \tag{5.11}$$

$$\delta_x^{(s)}(x,y) = s_1 r^2 + s_2 r^4 \tag{5.12}$$

$$\delta_y^{(s)}(x,y) = s_3 r^2 + s_4 r^4 \tag{5.13}$$

$$\tag{5.14}$$

The presented distortion model in equation (5.8) is composed by three single models, radial distortion, equation (5.9), tangential distortion, equations (5.10) and (5.11), and thin-prism distortion, equations (5.12) and (5.13).
These models represent different physical conditions like an imperfect assembled camera or effects caused by the lense itself, e.g. if the lense and the image sensor aren't perfectly parallel, or the distortion caused by the lense itself. [9]
It's not predetermined if $[x_1, y_1]^T$ or $[x_2, y_2]^T$ is the distorted point or undistorted point. The model can work in both ways, and the usage may differ by the used toolbox and the use case. It's easier to undistort a whole image if the distortion model distorts positions. That's because it's not purposeful to undistort a position of a pixel and get a resulting position in between multiple pixels of the final image. The other way around to calculate from which position the color of the undistorted pixel originates and then interpolate the resulting color from the adjacent pixels is much easier. But as mentioned the requirements for the distortion model change. If the use case is to undistort only one pixel, it is more efficient if the distortion model undistorts positions.[35]

6. TIGERs Camera Calibrator

The TIGERs Camera Calibrator is a collection of python modules and jupyter notebooks, created to easily calibrate the cameras of the TIGERs bots using the Open Source Computer Vision Library (OpenCV). The Camera Calibrator requires images of calibration patterns. At first a detector detects a chessboard calibration pattern and positions are obtained, which would be placed on straight lines on a undistorted image. With the positions detected on multiple images a calibrator approximates the camera and distortion model parameters. In a final step the distortion model is inverted to suit the use case of the on bot vision software. The aim is to create a sufficiently precise distortion model with as few parameters as possible to reduce the performance impact on the bot hardware.

6.1. Pattern Detection Notebook

The pattern detection is handled by OpenCV. It can detect 3 different patterns, a chessboard pattern and two circle patterns, one symmetric and one asymmetric, but only the chessboard pattern is currently supported in the TIGERs Camera Calibrator. [26] The Detector itself is wrapped in an extra python module, which is used in the detection notebook. It can be fully configured, by changing the parameters of a python dictionary. After the detection is applied a TkInter based graphical user interface (GUI) is used to validate the results. The window shows the pictures, where a pattern was detected and highlights the positions of the detected features. It is up to the user to make the decision if the pattern is detected well enough or whether the configuration should be improved further. The notebook structure is designed

for accumulating results from multiple detection runs with different configurations. Therefore patterns with a different amount of chessboard tiles and different detection parameters can be combined. The results are shared between the notebooks via the jupyter storemagic. [33]

6.2. Camera Calibration Notebook

The OpenCV camera calibration module is wrapped in another python module, which is used by the camera calibration notebook. In addition the distortion and camera model is implemented in a python module as well and is used to validate the calibrated Camera Parameters. The implemented distortion model only supports radial, tangential and thin prism distortion. Therefore only these 3 distortions are enabled by the calibration wrapping and the tilted camera model, also provided by OpenCV, is not available. [26] The main feature of this notebook are the plotting and validation capabilities. Which are shown in more detail in Section 6.4 and appendix B.

6.3. Distortion Inversion Notebook

In the next notebook the distortion inversion can be controlled. This is necessary because OpenCV does provide the distortion model parameters in the direction distort an undistorted point into the distorted picture. [26] Which is optimal to undistort whole images, see Section 5.4. Therefore a inversion of the distortion model is necessary to effectively undistort the ball position. For this reason a python module was developed using the scipy library, to optimize the parameters of the inverted Model by using data generated by the OpenCV model.

6.4. TIGERs Camera Calibrator in Use

The following section describes how the created tools are used on a set of images. Multiple Programs and scripts can be found on the internet to take these pictures with the Raspberry Pi, e. g. `raspistill`. It's important that the chosen camera mode matches the used camera mode of the final product. They don't need to be equal, but they must have the same aspect ratio and use the same cutout of the camera FOV.

6.4.1. Detection

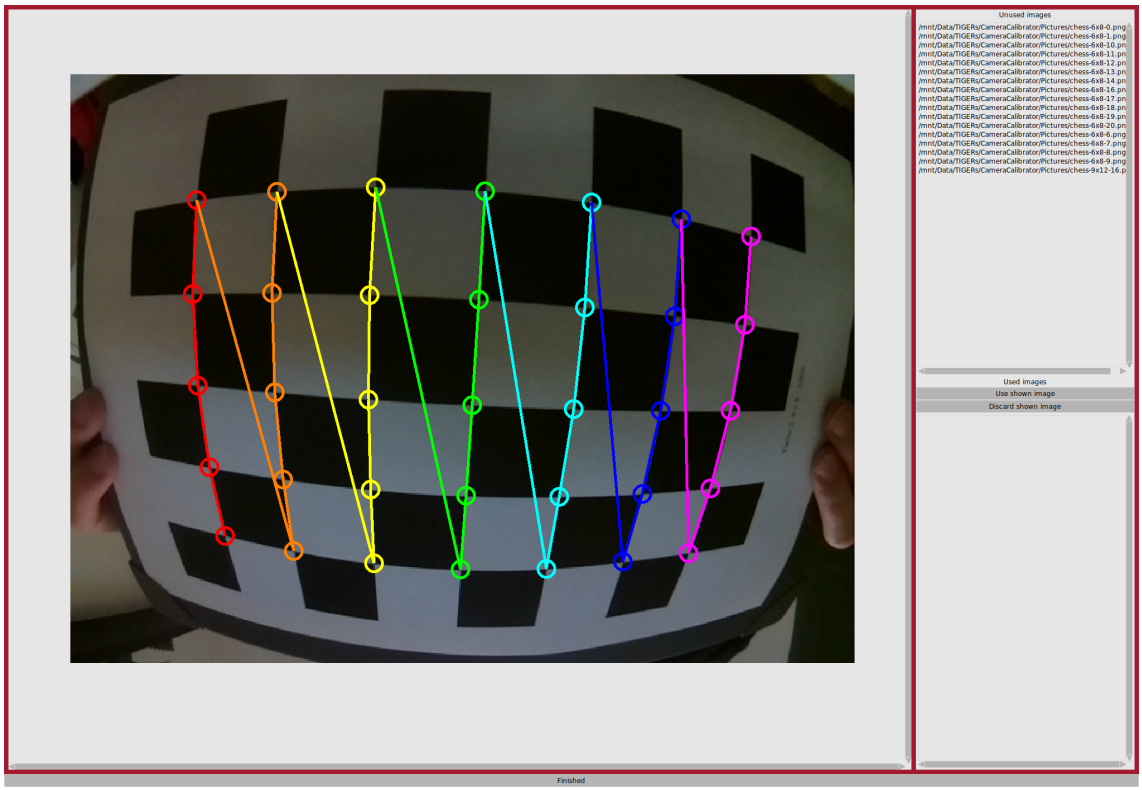


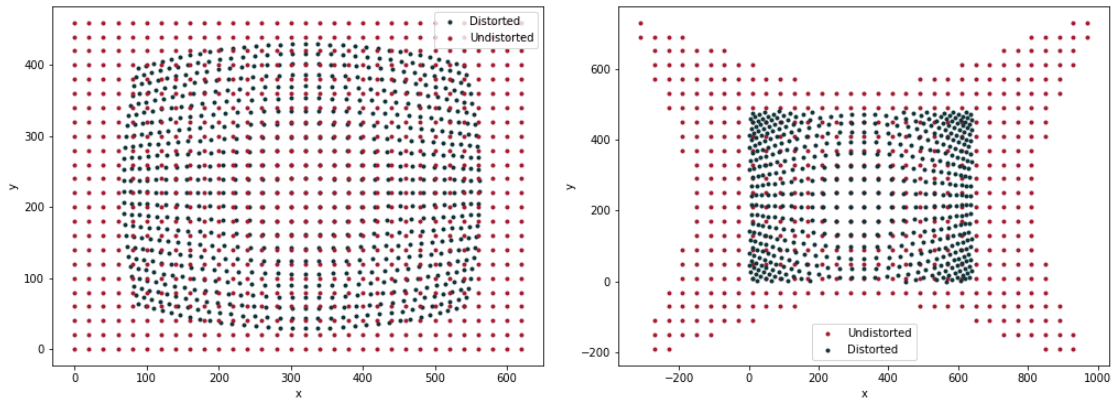
Figure 6.1.: Successful detection of a chessboard pattern

The paths of the resulting images are inserted into the list `all_images` and the detection is configured with the `chess_detect_config` dictionary. Part of the

parameters are used for the `findChessBoardCorners` function of OpenCV. It takes care of the most part of the detection. Note, that the x and y parameters don't specify how many chessboard tiles there are. It specifies how many inner corners with 4 aligning tiles exist, see the detected pattern in Figure 6.1. On a 6x8 chessboard this results into 5x7 corners. [26] With the remaining parameters a subpixel corner detection can be made. It may be used to refine the positions of the detection even further by using the `cornerSubPix` function of OpenCV. [26] Figure 6.1 shows the GUI where the user can zoom into the images and decide if it shall be used in this way.

6.4.2. Calibration

The Camera Calibration notebook is similarly configured via the `calibration_config` dictionary. As mentioned earlier it provides nearly the full features of the `calibrateCameraExtended` OpenCV function by only excluding the tilted camera model. Further information on the dependencies of these parameters can be found in the OpenCV documentation. [26]



(a) Reduced image section **(b)** Full image section

Figure 6.2.: Comparison between the considered image sections

Despite the overall aim to create a model with as few parameters as possible, the focus in this step is on a very precise model. That's because the inverted model is

configured independently, and is created by the data this model creates, so a better model created by OpenCV will yield in a better training environment for the final product. To classify the best model, the root-mean-square error (RMSE) provided by OpenCV during the calibration is not sufficient. Again that's because of the direction of the model. This model is created to undistort a image, and this affects the considered image section of the model. In Figure 6.2a is this reduced section shown. The red dots represent the pixels in the undistorted image and the green ones, the corresponding pixels in the distorted image. The problem is, that the green dots only represent a small amount of the whole image. If the model is used on the whole distorted image area a figure like Figure 6.2b might be the result.

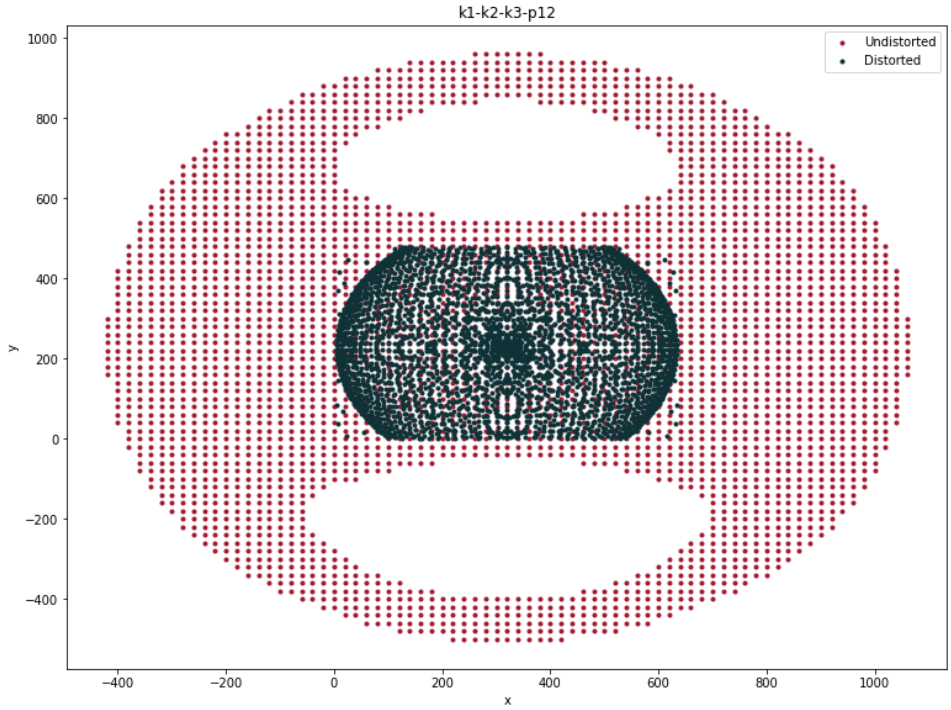


Figure 6.3.: Criteria 1: rectangular distorted shape

As one target of this work is, to undistort a ball position, which can appear on the whole distorted image, the full image section must be considered and the RMSE is not usable alone. But it is still a valuable criteria, because it provides the information how well the model approaches the reality in the reduced image section. The aim is to create a distortion model which is bijective on the whole area. To achieve this

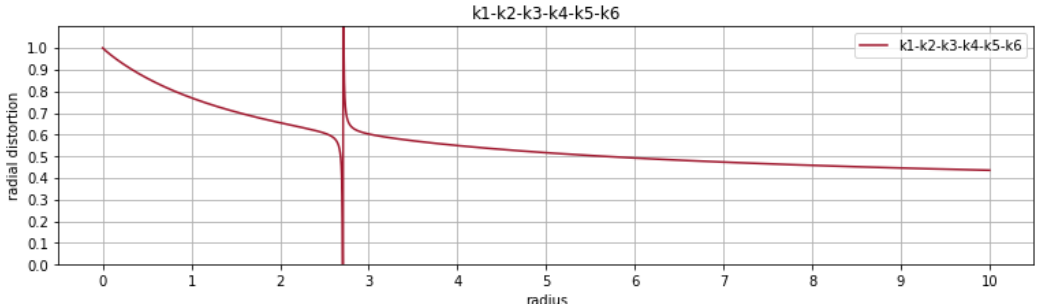


Figure 6.4.: Criteria 2: no singularity in radial distortion

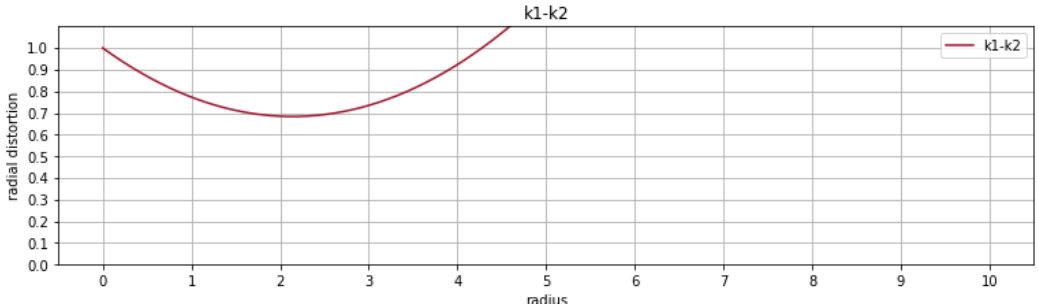


Figure 6.5.: Criteria 3: mostly monotonic radial distortion

four additional criteria are introduced.

Criteria 1 Is the shape of the distorted image in the plots rectangular?

Criteria 2 Has the radial distortion no singularities?

Criteria 3 Is the radial distortion mostly monotonic?

Criteria 4 Has the shape of the undistorted image a clear border?

The first criteria, the shape of the distorted image is relevant because of the way how the figures like Figure 6.2b are created. All points within the 640 by 480 pixel image, and a padding of an additional 1000 pixels around the image are distorted with the distortion model provided by OpenCV. These points are filtered and only if the distorted point is within the 640 by 480 image, the undistorted input is added to the undistorted image. If the model does not behave very well outside of it's designed borders it can result into a shape shown in Figure 6.3. The target is to have a completely filled rectangular shape of the distorted image, so that every possible

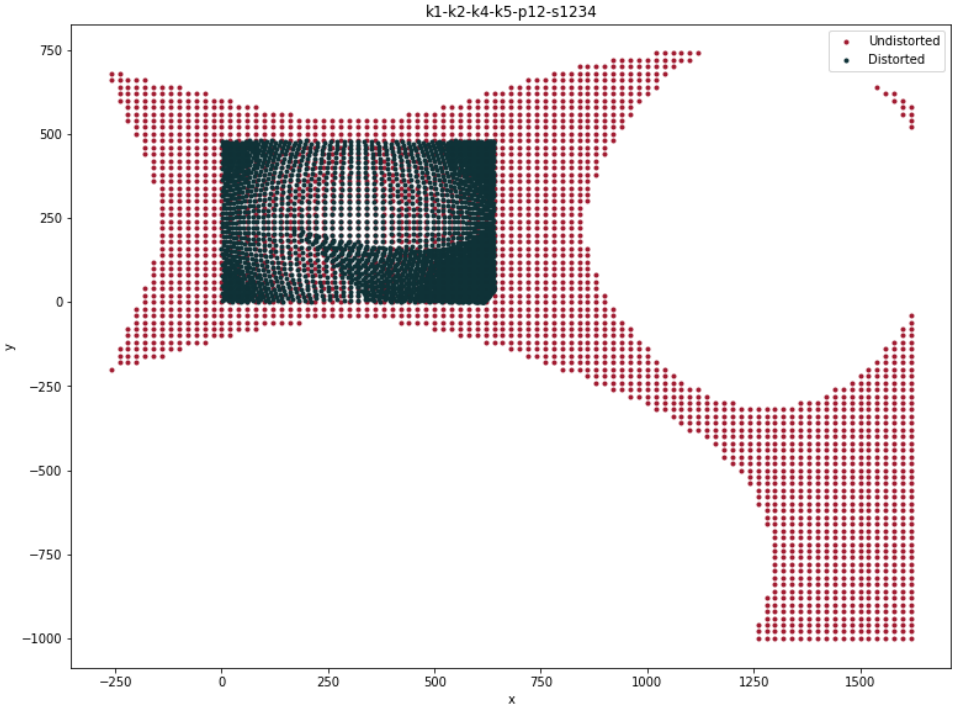


Figure 6.6.: Criteria 4: sharp and wide border around undistorted shape

position in the image has a corresponding undistorted point.
The second and third criteria are based on the radial distortion model $\delta^{(r)}(r)$. In the area close to a singularity the values sometimes tend towards infinity. This causes the model to distort points within this distance very unpredictable. A plot with a singularity is shown in Figure 6.4, and it's affects can be seen in Figures B.4a and B.16a. The area were no green distorted points are shown, is the region where the singularity can be found. This unpredictable distortion of positions does affect the data to train the final model on and therefore should be avoided.
The third criteria, that the radial distortion should be mainly monotonic is based on an empiric observation. The distortion of a lens is stronger the further away from the optical center. Therefore the radial distortion also needs to get bigger, the further away it gets from the optical center, respectively the radius increases. Especially low parameter models like shown in Figures 6.5, B.1, B.7 and B.19 tend to suffer from this problem. [26]

Model name	C. 1	C. 2	C. 3	C. 4	RMSE	Alias
k1-k2	X	X		X	0.3821	
k1-k2-k3		X	X	X	0.2702	
k1-k2-k3-k4-k5	X	X	X		0.2604	
k1-k2-k3-k4-k5-k6	X		X	X	0.2603	
k1-k2-k4		X	X		0.2610	
k1-k2-k4-k5	X	X	X	X	0.2604	A
k1-k2-p12	X	X		X	0.3802	
k1-k2-k3-p12		X	X	X	0.2677	
k1-k2-k3-k4-k5-p12			X		0.2594	
k1-k2-k3-k4-k5-k6-p12	X		X	X	0.2589	
k1-k2-k4-p12		X	X		0.2594	
k1-k2-k4-k5-p12	X	X	X	X	0.2589	B
k1-k2-s1234	X	X		X	0.3703	
k1-k2-k3-s1234		X	X	X	0.2644	
k1-k2-k3-k4-k5-s1234	X	X	X		0.2567	
k1-k2-k3-k4-k5-k6-s1234	X		X	X	0.2565	
k1-k2-k4-s1234		X	X		0.2572	
k1-k2-k4-k5-s1234	X	X	X	X	0.2568	C
k1-k2-p12-s1234	X	X		X	0.3658	
k1-k2-k3-p12-s1234		X	X	X	0.2643	
k1-k2-k3-k4-k5-p12-s1234	X	X	X		0.2553	
k1-k2-k3-k4-k5-k6-p1-s12342	X	X	X		0.2552	
k1-k2-k4-p12-s1234		X	X		0.2561	
k1-k2-k4-k5-p12-s1234	X	X	X		0.2553	

Table 6.1.: Comparison of the different calibrated distortion models

The fourth criteria is the border of the undistorted shape in the plot. This border needs to be sharp and wide. In Figure 6.6 this is not the case. The lower left corner of the undistorted shape continues beyond the expected border. This effect can also be seen in the distorted green points. The green dots look like they wrap around

Parameter	A	B	C
f_x	338.70	338.44	337.65
f_y	338.36	338.13	337.39
c_x	319.15	319.74	319.88
c_y	228.98	229.25	232.14
k_1	7.0490	7.6025	6.4179
k_2	1.0318	1.1167	0.9113
k_4	7.4290	7.9839	0.9113
k_5	3.3603	3.6295	3.0060
p_1	-	-0.0001	-
p_2	-	-0.0003	-
s_1	-	-	-0.0003
s_2	-	-	-0.0002
s_3	-	-	-0.0035
s_4	-	-	0.0015

Table 6.2.: Calibration results

something and continue in another direction. This clearly counteracts the aim to create a bijective distortion model and should therefore be avoided. Even if there is a small gap between the usable part and the outer points, a wide gap is preferred to make sure it's not only the imprecision of the plot.

These four additional criteria are applied to Figures B.1 to B.24 and the results are shown in Table 6.1. Only the three highlighted models A, B and C meet the requirements. All use the k_1, k_2, k_4 and k_5 parameters of the radial model. B uses in addition the tangential and C the thin prism model, with the parameters p_1, p_2 or s_1, s_2, s_3, s_4 respectively (Table 6.2). The results vary on some parameters like k_4, but they don't allow to rank the distortions, even in combination with the RMSE. Therefore all three models are further investigated in Section 6.4.3.

6.4.3. Distortion Inversion

The procedure of inverting the model is at first creating enough data with the previous model and optimize the inverted one with it. The training with a low parameter model usually comes with a loss in precision, but as shown in the RMSE column of Table 6.1, even the models with two or three parameters created decent results in the area they were trained on. Therefore in the inversion process it's important to create the test data set for the whole distorted image area. The data consists of a set of n two-dimensional undistorted Points $P = \{p_1, p_2, ..., p_n\}$ and a set of n two-dimensional distorted points $Q = \{q_1, q_2, ..., q_n\}$. These data sets are created on the same way how the plots are created. Each undistorted point p_i inside an image area with a large padding is distorted and if this distortion is inside the area, the point is added to the data set together with its corresponding position $q_i = \delta(p_i)$. With this data the cost function c can be described as Equation (6.1). The `minimize` of the scipy kit is used to optimize f. [31]

$$c(k_1, ..., k_6, p_1, p_2, s_1, ..., s_4) = \sqrt{\frac{1}{n}\sum_{i=1}^{n}\left([\|\delta^{-1}(k_1, ..., k_6, p_1, p_2, s_1, ..., s_4, q_1) - p_1\|_2]^2\right)} \tag{6.1}$$

To create the best low parameter model, further investigation is necessary. Currently, the models A-C have between four to ten distortion parameters in addition to four parameter of the intrinsic matrix matrix (Table 6.2). The intrinsic parameters can be reduced by combining f_x and f_y into $f = \frac{f_x+f_y}{2}$. The effect of the tangential and thin prism model are very small on the overall distortion especially if it's considered that the k_i parameters are multiplied with up to r^6, while p_i is multiplied with r^2 and s_i with up to r^4, therefore these parameters are ignored for the inversion. This reduces the amount of parameters down to a minimal of four, if a distortion model with only one parameter is sufficient. Multiple test with different combinations of radial parameters were run, and the results are shown in Table 6.3. As expected the highest order model k1-k2-k4-k5 clearly creates the best overall result, but it's unclear if e. g. model k1-k2 is sufficient as well. [35] Another effect, that the results with data created by the distortion model A are the best can be expected. That's because the inversion models don't have the capabilities to represent the tangential

or thin prism distortion used in model B and C.

	Model A	Model B	Model C
k1	13.429	13.54	14.286
k1-k2	1.4644	1.9937	4.1717
k1-k2-k3	1.2211	1.8459	4.0882
k1-k2-k4	1.2644	1.87	4.1047
k1-k2-k4-k5	0.25493	1.4575	3.7171
k1-k4	3.1262	3.3821	5.0797

Table 6.3.: Final cost function value of the inverted distortion models

To decide if k1-k2 is sufficient, the errors are put into relation to the reality. The Raspberry Pi Camera Module has in combination with the used lense a FOV in y direction of ca. 70°. [6] Therefore an error of one pixel on a 640x480 image introduces an error of 0.15°. This might be measurable if the bot stands still, but during the game with the acceleration in multiple dimensions by the omnidirectional drive, or by bot collisions, or just by the roughness of the game field, has an imprecision much higher than 0.15°. Therefore even though the result is worse than the k1-k2-k4-k5 model the k1-k2 model was selected to be implemented on the bots, because of the smaller performance overhead. The question which data set shall be used is hard to answer. As shown on Figures B.6a, B.12a and B.24a the distortion models differ, but no applicable metric was found to tell which one represents the reality better. Therefor a real life test is necessary. The complete results are shown in Appendix C, but in short the model calibrated with the data from Model C created the best results with an average error of -9.6mm ± 15.0mm. Therefore these parameter are used on the bots, spite of the worse theoretical results shown in Table 6.3.

7. Back-Projection

After a successful calibration and distortion removal the detected ball position is still a two dimensional image position, which is helpful but can be improved. A three dimensional position in regards to the bot is much more helpful, because the bot can not only detect if the ball moves from the left side of the image to the right side. It also knows if the ball moves towards the bot or away, or is in the air, therefore a velocity and direction of the ball movement can be calculated and the prediction of the ball position can be improved. E. g. during the task of intercepting a ball, it's not sufficient to move the bot to the left, if the ball is on the left side of the image and vice versa, because if the ball is shot in an angle this might fail. E. g. the ball starts on the left in regard to bot, but the correct interception point would be on the right side, because of the direction of the ball movement. The bot would still start moving to the left and would realize midways that he's on the wrong path, which might be too late to intercept the ball correctly.

To create this three dimensional position, some extra information needs to be extracted from the picture. A position on an image represents a ray into the three dimensional world. But the information where on the ray the ball is, is lost during the process of taking the image. A common approach is to intersect rays of two different

Axis	bot	camera
$+x$	right	right
$+y$	forward	down
$+z$	up	forward

Table 7.1.: Comparison of the axis between bot and camera space

cameras, and get the exact three dimensional position, but this is not possible on the TIGERs Bots, due to the lack of a second camera. Therefore another approach is required, which is to detect the size s of the ball on the image. This size directly corresponds to the distance d between ball and the camera. The relationship is shown in Equation (7.1), with h the height of the ball in world units and f the focal length. [36][37]

$$\frac{h}{d} = \frac{s}{f} \tag{7.1}$$

And with d, the ray is known, as well as the position on the ray and therefore the three dimensional position of the ball can be calculated by Equation (7.2)

$$\begin{bmatrix} X_c \\ Y_c \\ Z_c \end{bmatrix} = \frac{d}{\sqrt{x^2 + y^2 + 1^2}} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{7.2}$$

Another step of the backprojection is to transform the position from the camera space into the more widely used bot space. This allows the information created by the balldetector to be easily used in other parts of the software, because the information is already in the used reference system. The transformation consists of an translation, rotation and swapping axis. The meaning of the different axis can be found in Table 7.1. Note that not only axis y and z need to be swapped, in camera space positive y coordinates represent down and in bot space positive z up. Therefore this axis needs also to be inverted. The camera is only tilted $\varphi = -20°$ downwards, to increase the part of the FOV pointed onto the field and therefore the rotation is only around the x axis. This simplifies the needed rotation matrix. The translation from the bot center to the camera is $t = [0, 0.070, 0.072]^T$. All three

steps combined are shown in Equation (7.3)

$$\begin{aligned}\begin{bmatrix} X_b \\ Y_b \\ Z_b \end{bmatrix} &= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\varphi & -\sin\varphi & t_y \\ 0 & \sin\varphi & \cos\varphi & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X_c \\ Z_c \\ -Y_c \\ 1 \end{bmatrix} \\ &= \begin{bmatrix} X_c \\ \cos\varphi Z_c + \sin\varphi Y_c + t_y \\ \sin\varphi Z_c - \cos\varphi Y_c + t_z) \end{bmatrix}\end{aligned} \tag{7.3}$$

8. Future Work

This chapter is dedicated to future work, which could not make it into this work's projects. They all are related to further improving stability, robustness and performance of the TIGERs' image recognition software `balldetector`.

The software currently uses the Raspberry Pi 3 A+ to do all processing and analysis of the incoming video stream from the camera module. Generally, with the topic of image and object recognition in computer vision, the use of convolutional neural networks (CNNs) comes into play relatively quickly. For the original implementation of the `balldetector` in 2019, the use of CNNs was evaluated and concluded as "impossible without the use of graphics processing units (GPUs), which allow major speed-ups (x10 to x30) compared to Central processing unit (CPU) only processing" [32].

With the machine learning platform TensorFlow Lite, image classification and training as well as optimizing a model is possible on the Raspberry Pi, as done by Johnson, who documented her project online [21]. She used TensorFlow Lite, which is optimized for mobile and edge devices, and the Raspberry Pi 4 (4GB) with the Raspberry Pi Camera in version 2. With the pre-trained model MobileNetV3-SSD she benchmarked the model at "roughly 8 frames per second" [21] and with using the Coral's USB Accelerator the model inference speeds could be accelerated further. The USB Accelerator contains an Edge TPU, is specialized for TensorFlow Lite operations and has set remarkable speed benchmarks. The benchmark by Coral states 7.2 ms for the time per inference for the MobileNetV2-SSD when using a Desktop CPU and the USB Accelerator (USB 3.0) additionaly, compared to 106ms for only using the Desktop CPU [11]. Other model architectures achieved comparable

results. Using Coral's USB Accelerator, Johnson achieved around 24 frames per second.

Despite the different hardware preliminaries, the machine learning approach to detecting the robot soccer balls might shine in a new light by using Coral's USB Accelerator. But achieving 24 frames per second is still not fast enough for the use in the TIGERs project, as robots move as fast as 6 metres per second. The image classification or object detection as well as the Raspberry Pi sending the detected balls and their positions to the robot and the control of the robot to intersect the ball needs to be taken into account for the process.

Upgrading the existing Raspberry Pi 3 A+ or upgrading the camera module might also be helpful for improving precision and robustness for the `balldetector`. However, this is one of the least prioritized approaches, because upgrading 16 robots is not only a matter of time and effort but also the costs to the hardware need to be considered.

Other approaches of enhancing the performance with the given hardware are much more preferred initially. These include measuring the existing performance exactly, benchmarking the delays, for example the communication between the Raspberry Pi and the camera, the delay of MMAL, its driver, the communication delay in the UART communication between the robot's Firmware and the Raspberry Pi and also any delays that occur in the actual ball detection algorithm.

9. Conclusion

The goal of this study report was to develop solutions for robust on-board image recognition and to improve the current implementation of the software to better meet the requirements of the TIGERs project in the context of RoboCup SSL. The work focused on two main developments: an improved, centralised configuration concept and an undistorted backprojection of the ball.

By centralising the configuration systems, diverging configurations on several robots are avoided. In addition, it reduces the effort required to calibrate the robots, so that a better configuration can be achieved in the same time. This is particularly useful for events with a very tight schedule such as RoboCup.
The improved quality and consistency of calibration helps to achieve more robust results. The basic implementation still needs to be tested in such an environment, as so far, the benefits are merely theoretical. Once it proves successful, future implementations may include additional configurations, which can then already use the existing framework to send and store the configuration.

The complete process of calibrating a camera and distortion model was presented, starting with the detection of calibration patterns and ending with the backprojection of the detected ball position into the three dimensional space. A real life test with eight measuring positions successfully confirmed the whole process having a precise prediction with an average error of less than 10mm. The ability to control the robot in relation to three-dimensional ball position improves the overall stability, since unavoidable problems with two-dimensional positions are eliminated. The undistortion of the detected position further increases the stability by improving the detected ball positions.

Bibliography

[1] *8-Bit UART Datasheet: UART V 5.3.* URL: https://www.cypress.com/file/140646/download (visited on 03/05/2020).

[2] *A Brief History of RoboCup.* URL: https://www.robocup.org/a_brief_history_of_robocup (visited on 04/02/2020).

[3] Ian Auty. *What is MMAL?* 2018. URL: https://github.com/techyian/MMALSharp/wiki/What-is-MMAL%3F (visited on 11/25/2019).

[4] *Basics of UART Communication.* 2016. URL: https://www.circuitbasics.com/basics-uart-communication/ (visited on).

[5] *Buy a Raspberry Pi 3 Model A+ – Raspberry Pi.* URL: https://www.raspberrypi.org/products/raspberry-pi-3-model-a-plus/ (visited on 06/19/2020).

[6] *Camera Field of View Calculator (FoV).* URL: https://www.scantips.com/lights/fieldofview.html (visited on 06/19/2020).

[7] *cmake-toolchains(7) — CMake 3.18.0-rc2 Documentation.* URL: https://cmake.org/cmake/help/latest/manual/cmake-toolchains.7.html (visited on 06/19/2020).

[8] RoboCup Soccer Small Size League Technical Committee. *SSL-Vision Blackout Challenge.* 10/16/2018. URL: https://ssl.robocup.org/robocup-2019-technical-challenges/ (visited on 02/23/2020).

[9] Dean H. Brown. "Decentering distortion of lenses". In: (1966).

[10] Abdul Dremali. *Firmware vs embedded software: What's the difference?* 2019. URL: https://www.andplus.com/blog/firmware-vs-embedded-software-what-s-the-difference- (visited on 04/14/2020).

[11] *Edge TPU performance benchmarks.* URL: https://coral.ai/docs/edgetpu/benchmarks/ (visited on 05/23/2020).

[12] core Electronics. *Raspberry Pi Wide Angle Camera Module (Seeed Studio).* URL: https://core-electronics.com.au/raspberry-pi-wide-angle-camera-module-seeed-studio.html (visited on 06/02/2020).

[13] *Firmware: Definition*. 2006. URL: https://techterms.com/definition/firmware (visited on 04/14/2020).

[14] Raspberry Pi Foundation. *Camera Module*. URL: https://www.raspberrypi.org/documentation/hardware/camera/README.md (visited on 12/23/2019).

[15] Warren W. Gay. "Mastering the Raspberry Pi". In: Apress, Berkeley, CA, 2014. Chap. Cross-Compiling. (Visited on 06/17/2020).

[16] Hongbo Li, David Hestenes, and Alyn Rockwood. "Generalized Homogeneous Coordinates for Computational Geometry". In: *Geometric Computing with Clifford Algebras*. 2001.

[17] Brian Ingerson, Clark C. Evans, and Oren Ben-Kiki. *YAML Ain't Markup Language (YAML™) Version 1.2*. 3rd ed. 2009. URL: https://yaml.org/spec/1.2/spec.html (visited on 05/14/2020).

[18] *Introduction*. URL: https://crosstool-ng.github.io/docs/introduction/ (visited on 06/19/2020).

[19] ISO/IEC JTC1 SC22 WG14 N1021. *Information Technology: Programming languages, their environments and system software interfaces: Extensions for the programming language C to support embedded processors*. 09/24/2003. URL: http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1021.pdf (visited on 04/14/2020).

[20] J. Heikkila and O. Silven. "A four-step camera calibration procedure with implicit image correction". In: 1997.

[21] Leigh Johnson. *Real-time Object Tracking with TensorFlow, Raspberry Pi, and Pan-Tilt HAT*. 2019. URL: https://towardsdatascience.com/real-time-object-tracking-with-tensorflow-raspberry-pi-and-pan-tilt-hat-2aeaef47e134 (visited on 05/23/2020).

[22] Ken Lunn. "Software Development with UML". In: 2003. Chap. Software Development Life Cycle.

[23] Jan Newmarch. "Linux Sound Programming". In: 2017. Chap. Raspberry Pi. (Visited on 06/02/2020).

[24] *Objective*. URL: https://www.robocup.org/objective (visited on 04/02/2020).

[25] Nicolai Ommer, Andre Ryll, and Mark Geiger. "Extended Team Description for RoboCup 2019: TIGERs Mannheim". PhD thesis. Mannheim: Cooperative State University Mannheim, 2019. (Visited on 02/14/2020).

[26] *OpenCV: Camera Calibration and 3D Reconstruction*. URL: https://docs.opencv.org/master/d9/d0c/group__calib3d.html (visited on 06/19/2020).

[27] *RoboCupSoccer - Small Size.* URL: `https://www.robocup.org/leagues/7` (visited on 05/04/2020).

[28] Tony Royce. "C Programming". In: Palgrave, London, 1996. Chap. Complex Data Structures.

[29] *Rules of the RoboCup Small Size League.* URL: `https://robocup-ssl.github.io/ssl-rules/sslrules.pdf` (visited on 05/04/2020).

[30] Andre Ryll. *Hardware V6: Bot Hardware & Sicherheit.* 12/2018. (Visited on 02/01/2020).

[31] *scipy.optimize.minimize — SciPy v1.4.1 Reference Guide.* URL: `https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.minimize.html` (visited on 06/19/2020).

[32] Fabio Seel and Sabolc Jut. "On-Board Computer Vision for Autonomous Ball Interception: Implementing the Vision-Blackout Technical Challenge in the Robocup Small Size League". Study Report. Mannheim: Cooperative State University Mannheim, 2019. (Visited on 12/22/2019).

[33] *storemagic — IPython 7.15.0 documentation.* URL: `https://ipython.readthedocs.io/en/stable/config/extensions/storemagic.html` (visited on 06/19/2020).

[34] Eben Upton. *Camera board available for sale!* 2013. URL: `https://www.raspberrypi.org/blog/camera-board-available-for-sale/` (visited on 04/13/2020).

[35] Jason P. de Villiers, F. Wilhelm Leuschner, and Ronelle Geldenhuys. "Centi-pixel accurate real-time inverse distortion correction". In: (2008).

[36] Wikipedia, ed. *Intercept theorem.* 2020. URL: `https://en.wikipedia.org/w/index.php?title=Intercept_theorem` (visited on 06/19/2020).

[37] Wikipedia, ed. *Pinhole camera model.* 2020. URL: `https://en.wikipedia.org/w/index.php?title=Pinhole_camera_model` (visited on 06/19/2020).

Appendix

Appendix A.

Balldetector Configuration

Listing A.1: Data structures for the balldetector configurations in the header file "Commands.h"

```
# pragma once

// superior data structure Config
typedef struct _Config
{
  _ExtCameraConfig cameraConfig;
  _ExtDebugConfig debugConfig;
  _ExtDetectorConfig detectorConfig;
} Config;


// data structures for the different configuration sections
typedef struct _ExtCameraConfig
{
  uint16_t iso = 200;          // 100, 200, 400, 800
  uint32_t shutterSpeed = 10000;   // microseconds, max: 6s
  uint8_t exposureMode = 0;    // 0 - 13
  uint8_t awbMode = 0;         // 0 - 11
  float awbGainsBlue = 2.0;    // 1.0 - 8.0
  float awbGainsRed = 1.0;     // 1.0 - 8.0
  uint8_t brightness = 50;     // 0 = black, 100 = white
  int8_t saturation = 0;       // -100 - 100
  int8_t sharpness = 0;        // -100 - 100
  int8_t contrast = 0;         // -100 - 100
  int8_t exposureCompensation = 0; // -25 - 25
  uint8_t imageEffect = 0;     //0 - 26
  uint8_t hFlip = 0;           // 1 = true, 0 = false
```

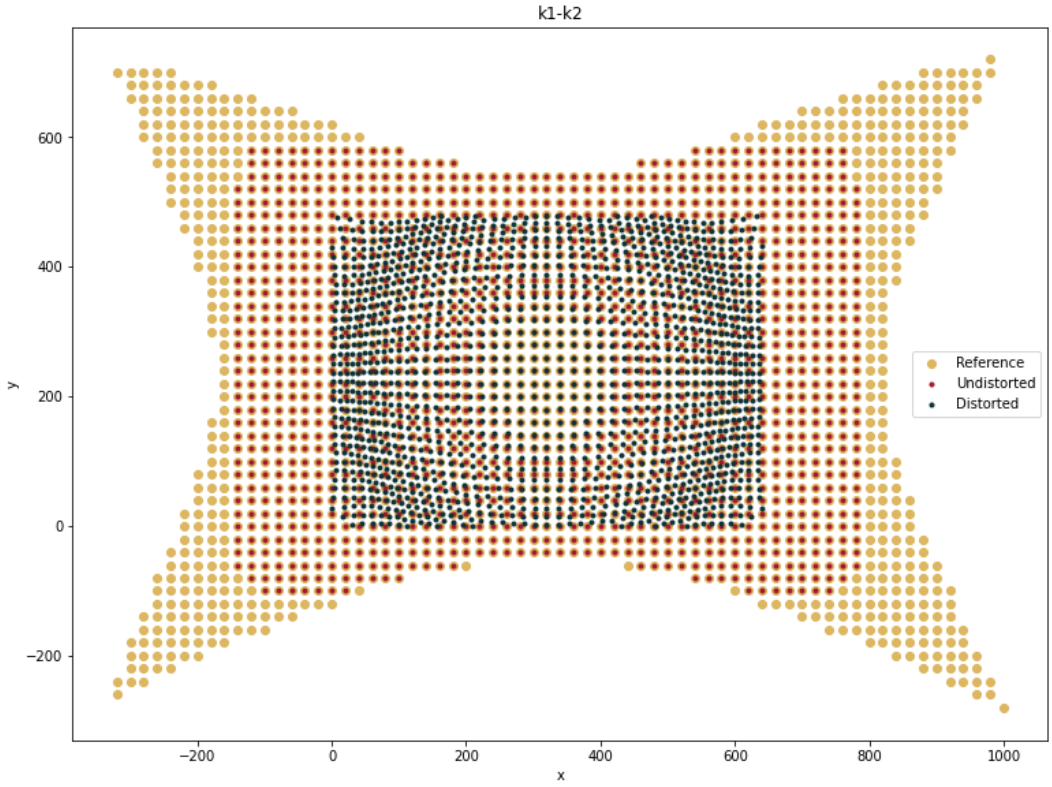
```
  uint8_t vFlip = 0;          // 1 = true, 0 = false
} ExtCameraConfig;

typedef struct _ExtDebugConfig
{
  uint8_t stdOut = 1;  // 1 = true, 0 = false
  uint8_t show = 1;    // 1 = true, 0 = false
  uint8_t balls = 1;
  uint8_t channel = 3; // uv --> Channel [enum]
  uint8_t lines = 1;   // 1 = true, 0 = false
  uint16_t sendInterval = 1;
  uint32_t resolution = 720;
} ExtDebugConfig;

typedef struct _ExtDetectorConfig
{
  uint8_t method = 1;   // "cmVisionScanLines" --> Method [enum]

  struct _ScanLines
  {
    uint16_t firstLineAt = 1;    // Percentage from top
    uint16_t lastLineAt = 90;
    uint16_t numLines = 60;
    uint8_t linesDirection = 0;         // "up" --> ScanLinesLinesDirection [
        enum]
    uint8_t lineDistribution = 0;       // quadratic -->
        ScanLinesLinesDistribution [enum]
    uint16_t edgeDetectionThresh = 35;   // Minimum value for an edge to be
        considered start/end of a ball
    uint16_t minConfidence = 20;
    uint16_t minDistBetwBalls = 15;
    float shadowFactor = 0.6;
  } ScanLines;

  struct _CmVisionScanLines
  {
    uint8_t maskBase = 0;     // "color" --> CmVisionMaskBase [enum]
    int32_t colorThresh = 50; // range at color channel used is -255 - +255
    float heightFactor = 0.75;
  } CmVisionScanLines;
} ExtDetectorConfig;
```

Appendix B.

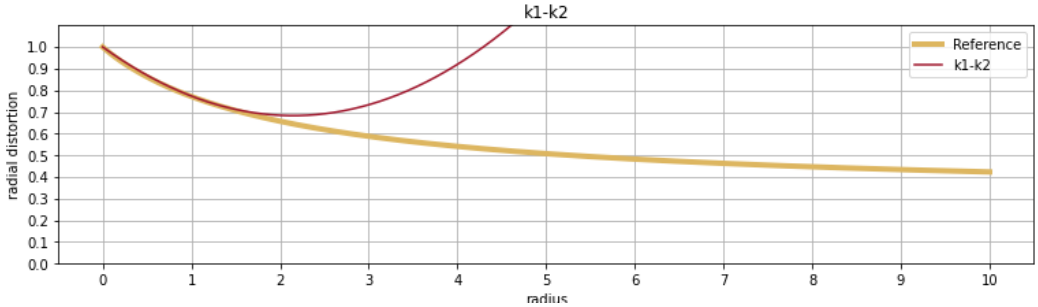
Calibration Plots

The golden points represent a reference model. It is a distortion model with $k_1, k_2, k_4, k5$ and s_1, s_2, s_3, s_4. Therefore no tangential distortion is used. This is the model selected for the final application as described in Chapter 6.
The undistortion plots are created by padding the distorted image with a large border of 1000 pixels. These positions are then distorted with the respective distortion model. The resulting distorted position is checked. Only if it is within the boundaries of the distorted image without the padding, the corresponding undistorted position is added to the undistorted plot.
The radial distortion plots only use the radial distortion part of the model. Although a usual radial distance of an undistorted position isn't much larger than 1.5 after it is projected onto the image plane, the radius is used in up to r^6 in the radial distortion model. Therefore the x-axis expands from $r = 0$ to $r = 10$.

B.1. Without tangential and prism distortion

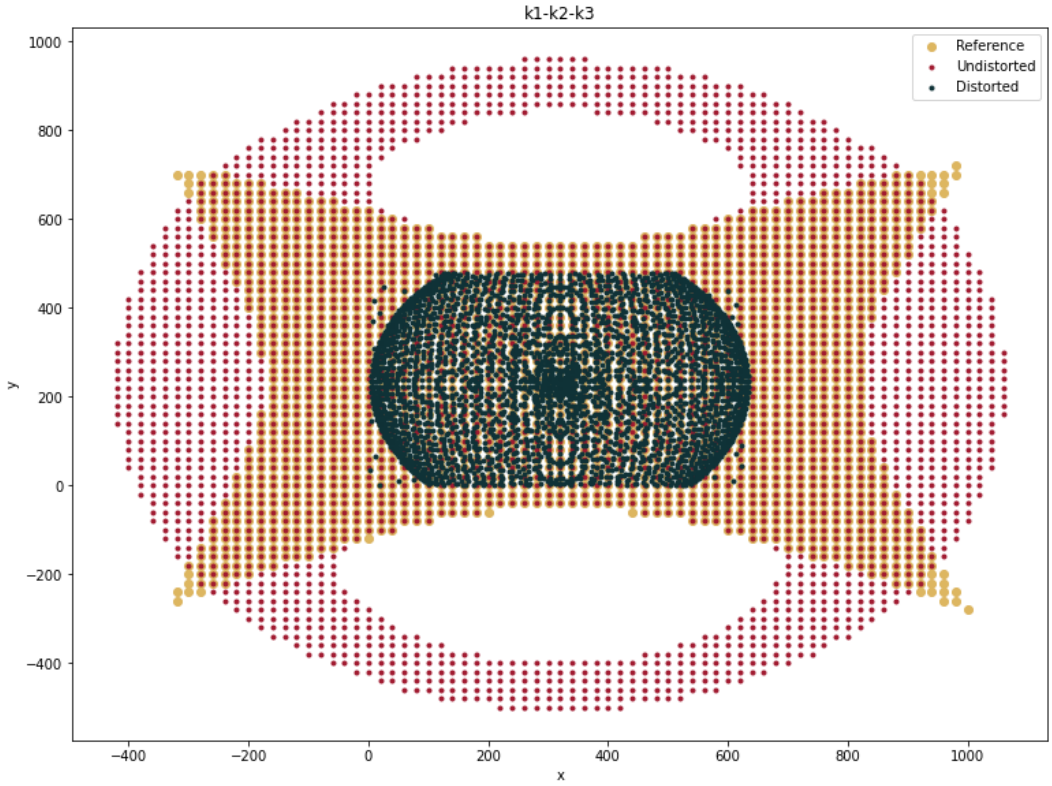


(a) Undistortion of an image

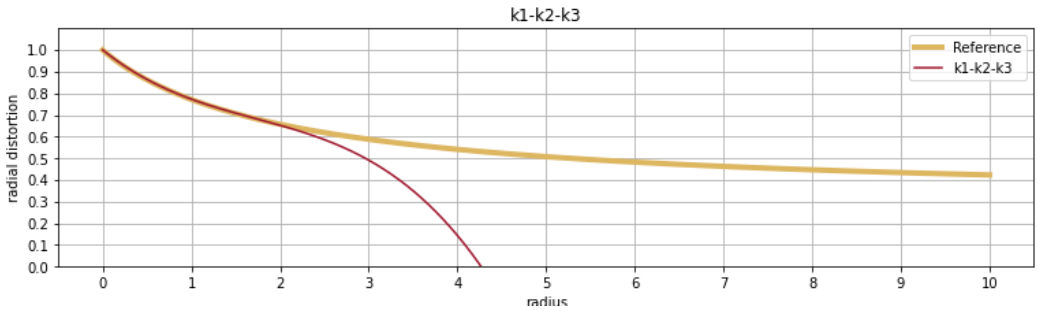


(b) Radial distortion

Figure B.1.: Distortion model with k_1, k_2

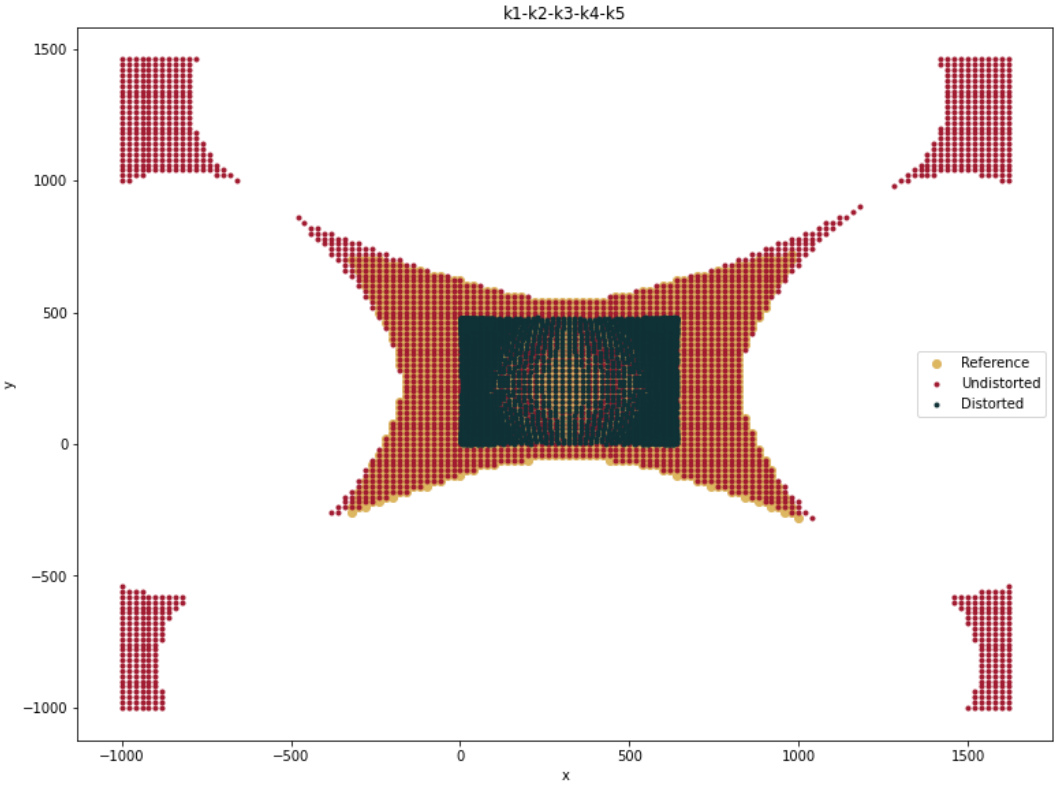


(a) Undistortion of an image

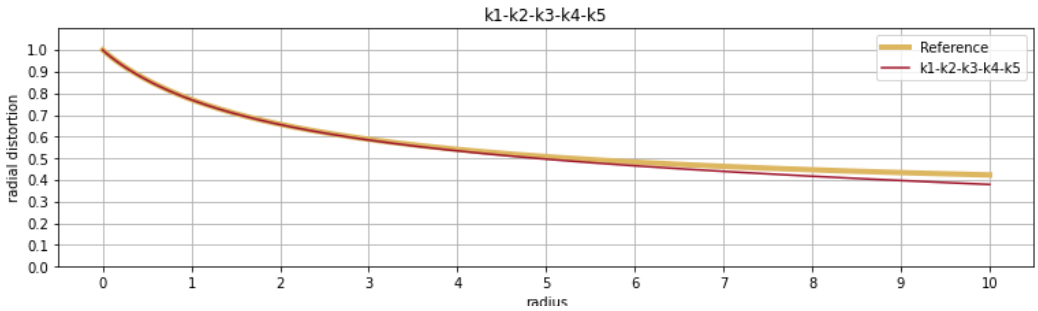


(b) Radial distortion

Figure B.2.: Distortion model with k_1, k_2, k_3

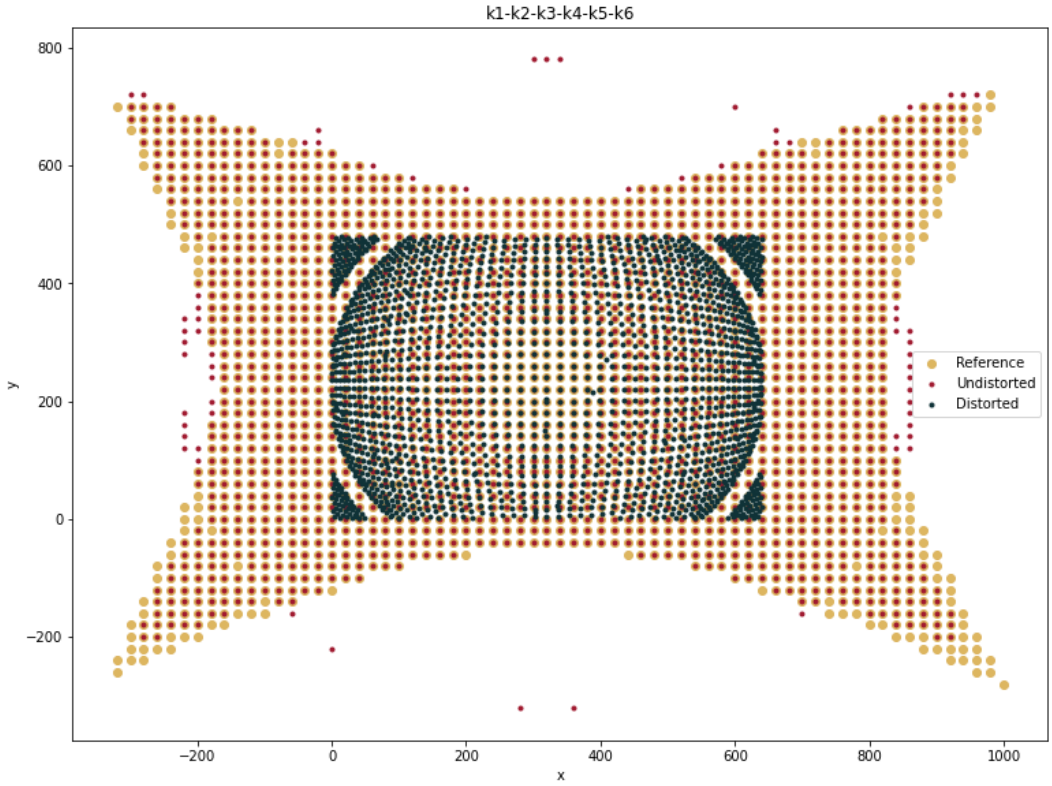


(a) Undistortion of an image

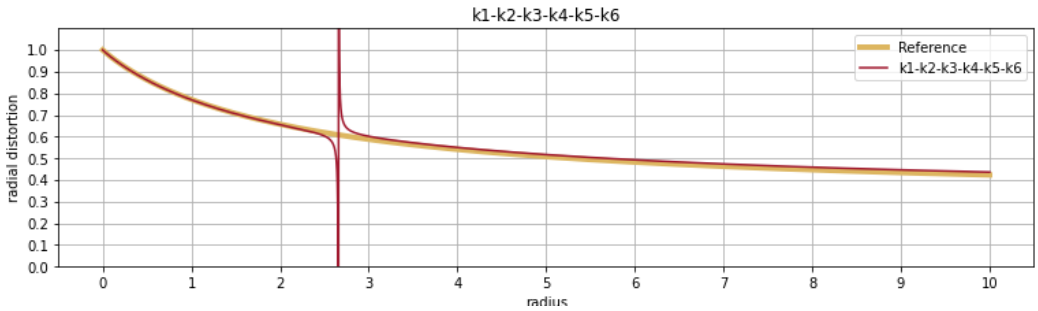


(b) Radial distortion

Figure B.3.: Distortion model with k_1, k_2, k_3, k_4, k_5

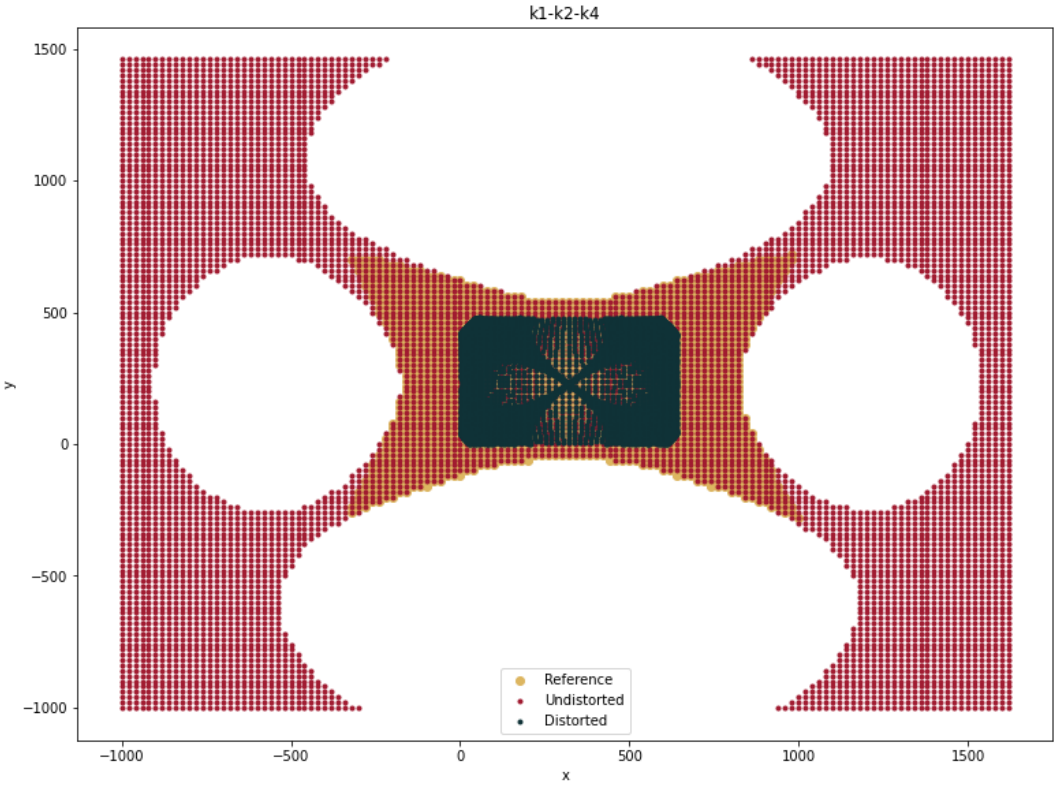


(a) Undistortion of an image

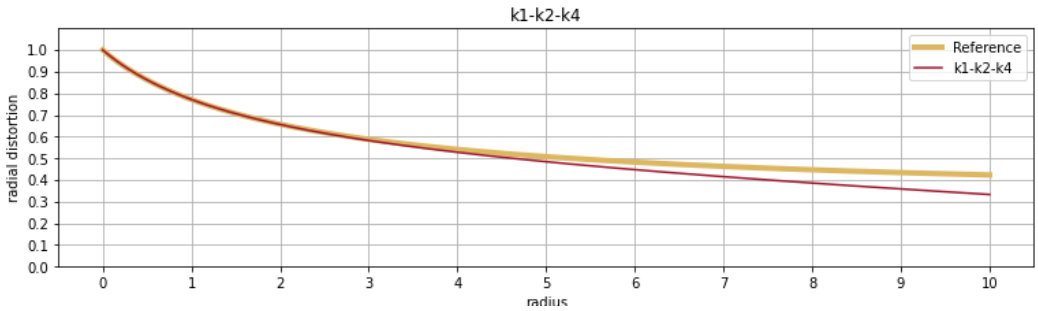


(b) Radial distortion

Figure B.4.: Distortion model with $k_1, k_2, k_3, k_4, k_5, k_6$

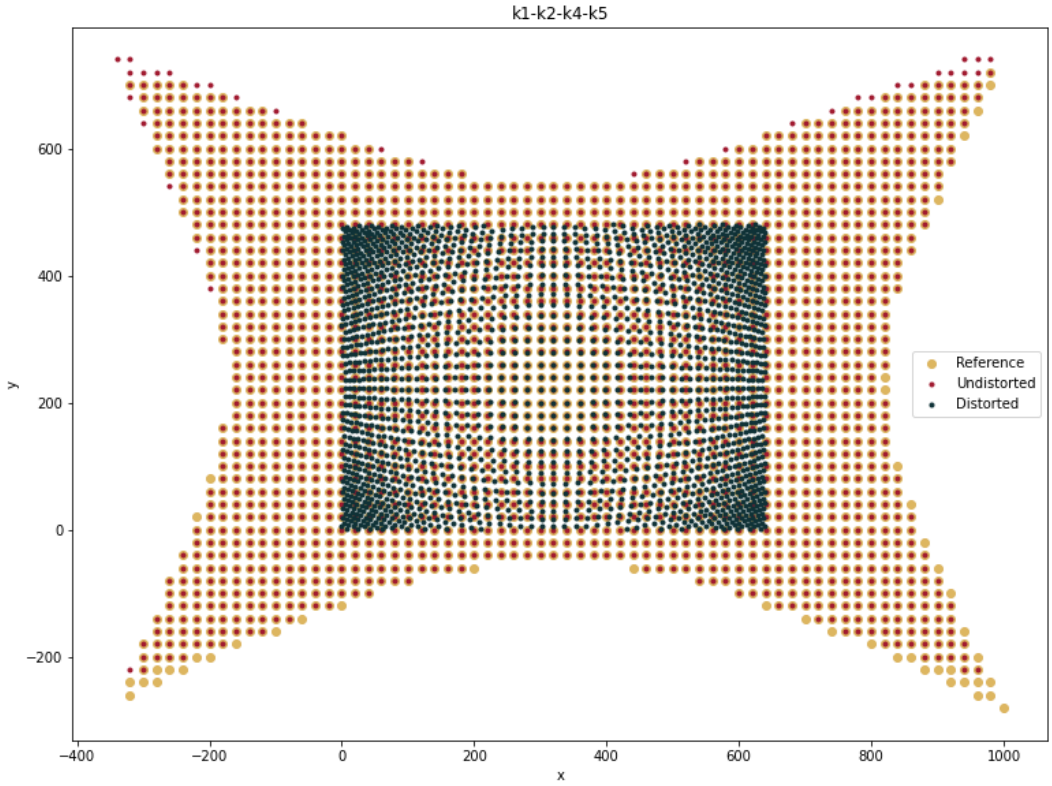


(a) Undistortion of an image

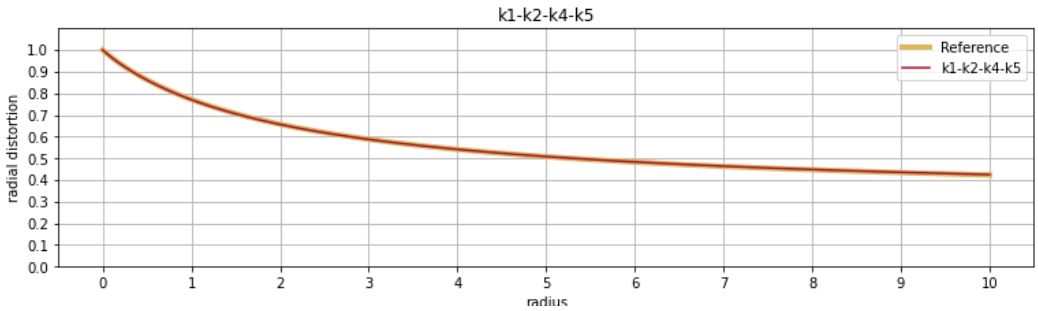


(b) Radial distortion

Figure B.5.: Distortion model with k_1, k_2, k_4



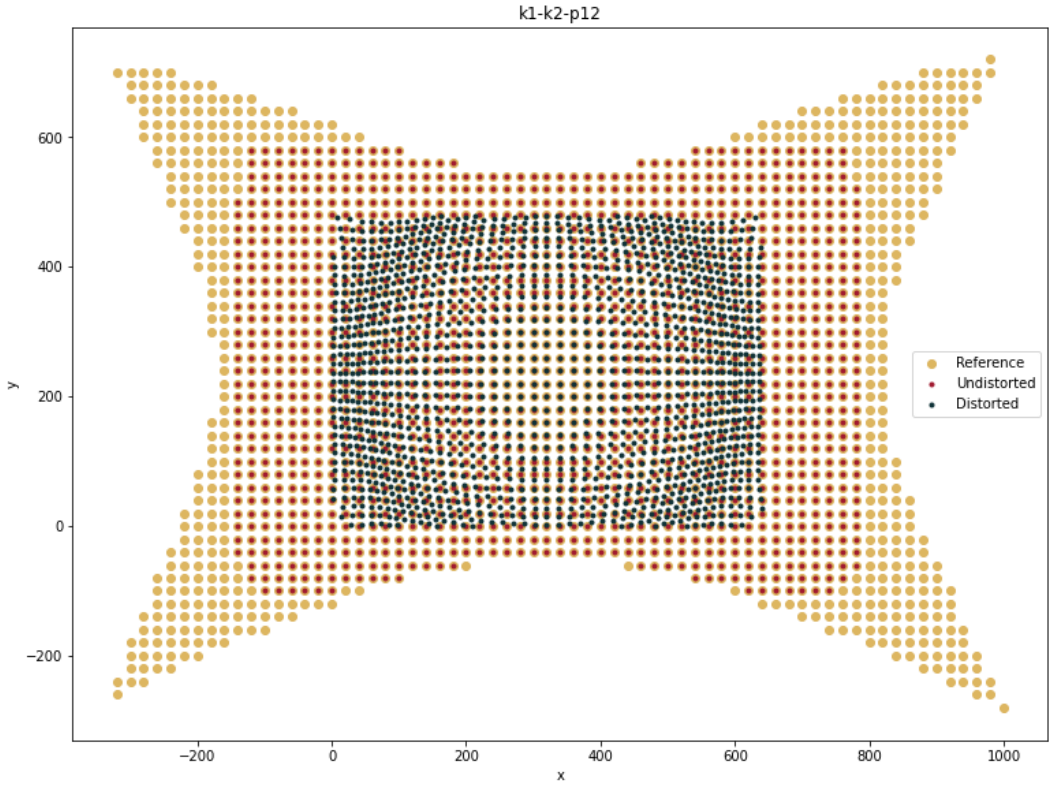
(a) Undistortion of an image



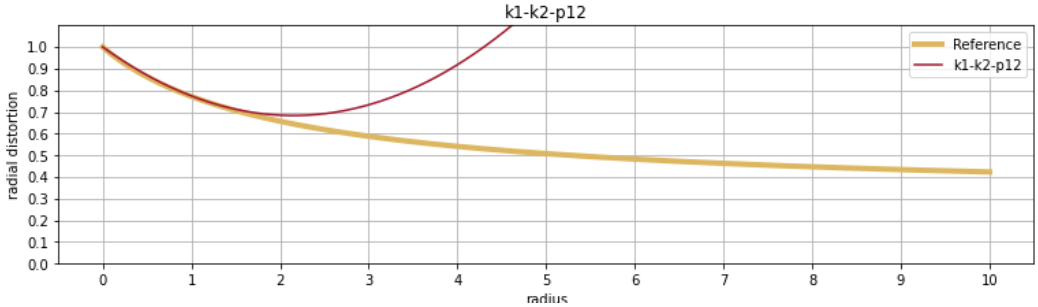
(b) Radial distortion

Figure B.6.: Distortion model with k_1, k_2, k_4, k_5

B.2. With tangential and without prism distortion

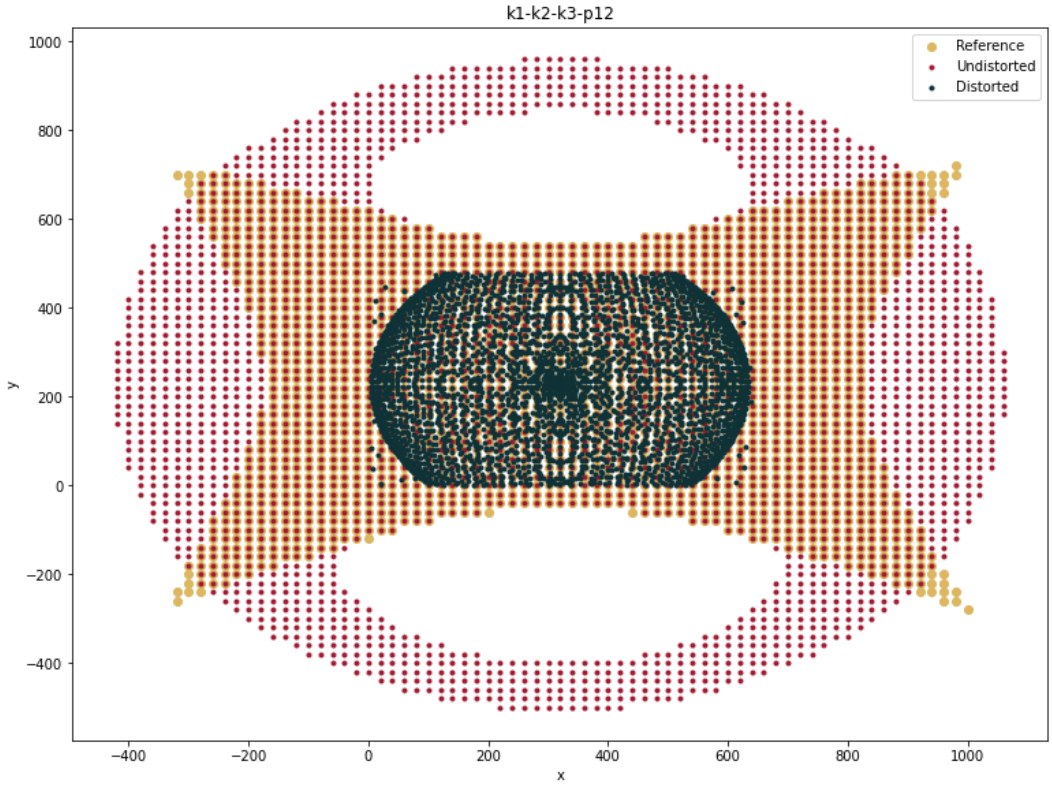


(a) Undistortion of an image

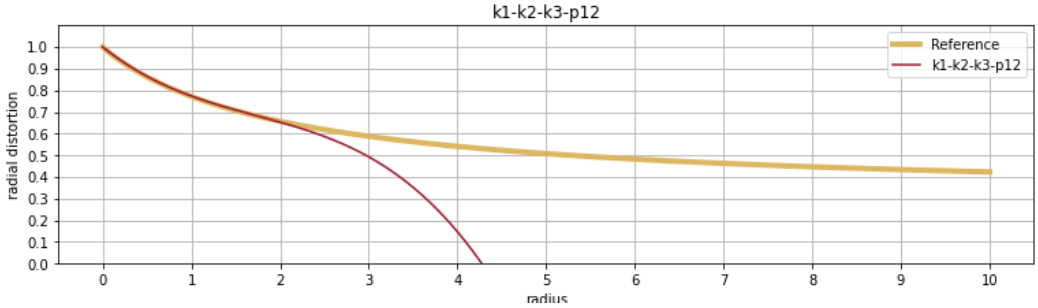


(b) Radial distortion

Figure B.7.: Distortion model with k_1, k_2 and p_1, p_2

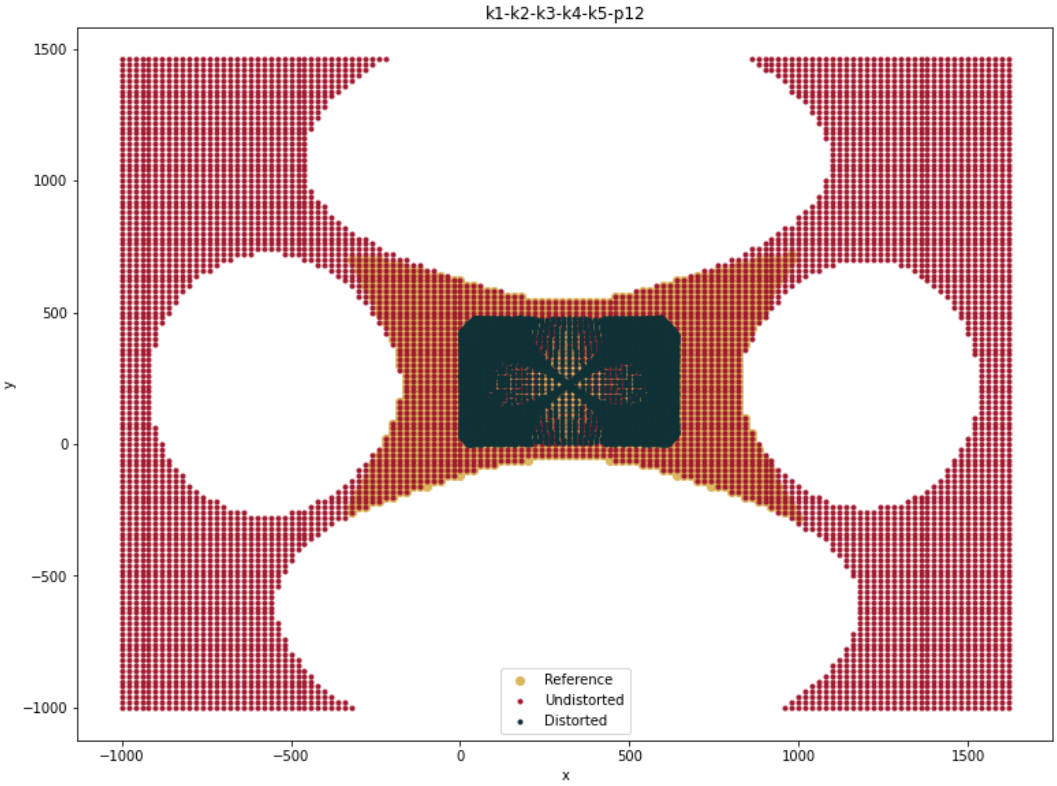


(a) Undistortion of an image

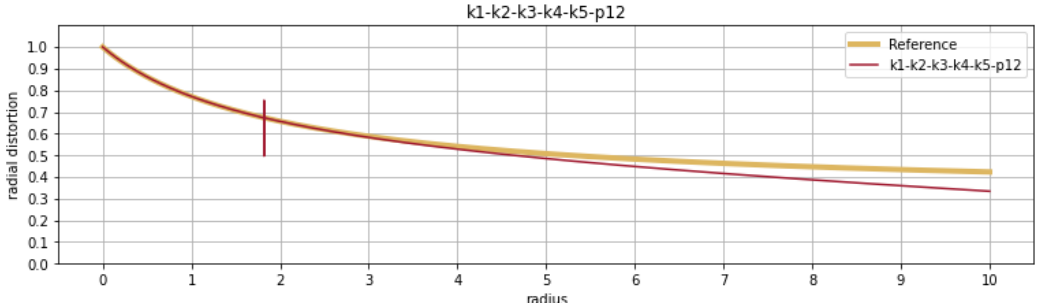


(b) Radial distortion

Figure B.8.: Distortion model with k_1, k_2, k_3 and p_1, p_2

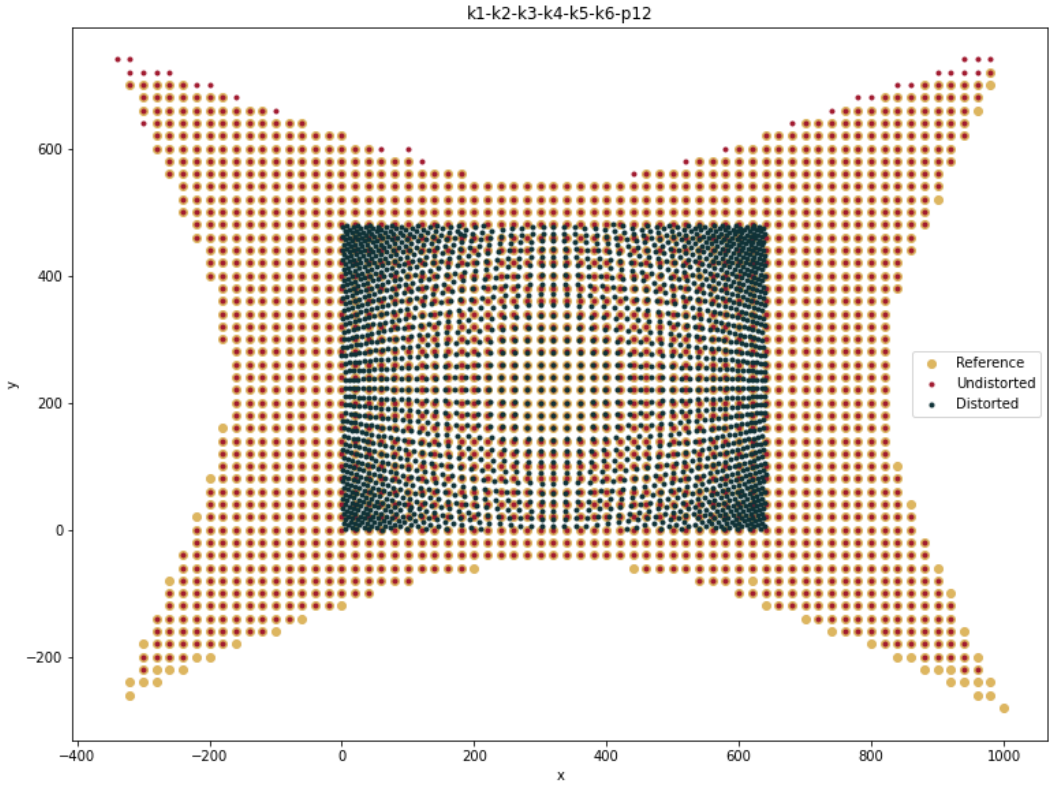


(a) Undistortion of an image

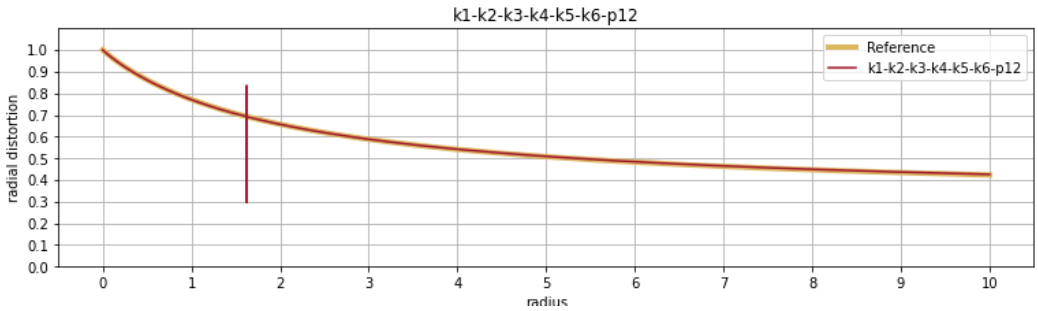


(b) Radial distortion

Figure B.9.: Distortion model with k_1, k_2, k_3, k_4, k_5 and p_1, p_2

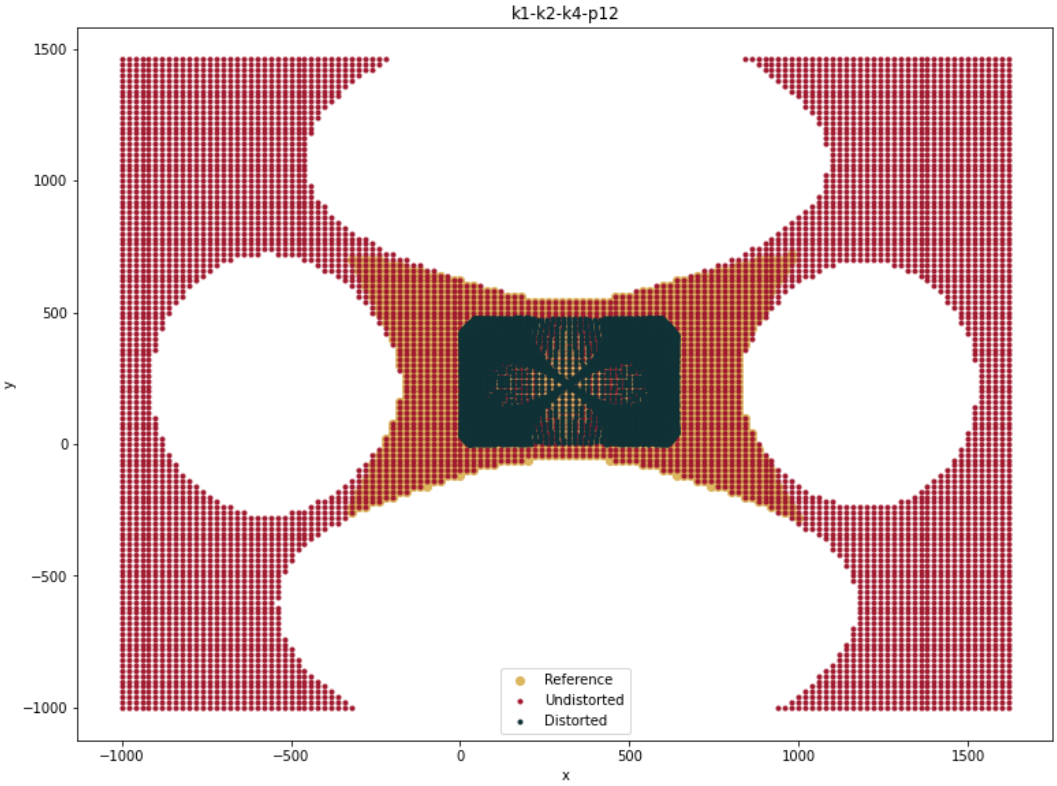


(a) Undistortion of an image

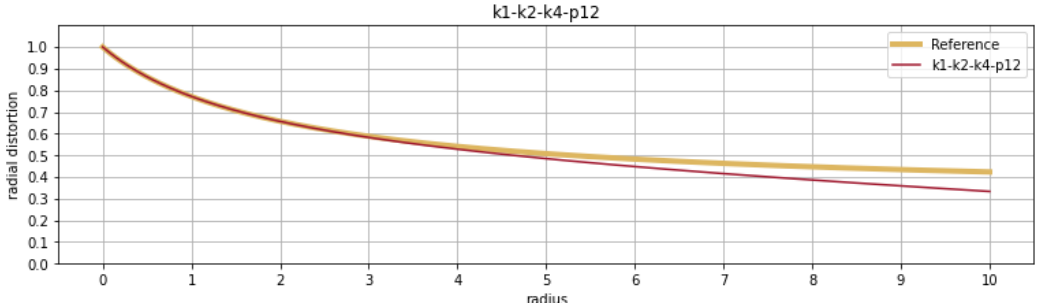


(b) Radial distortion

Figure B.10.: Distortion model with $k_1, k_2, k_3, k_4, k_5, k_6$ and p_1, p_2

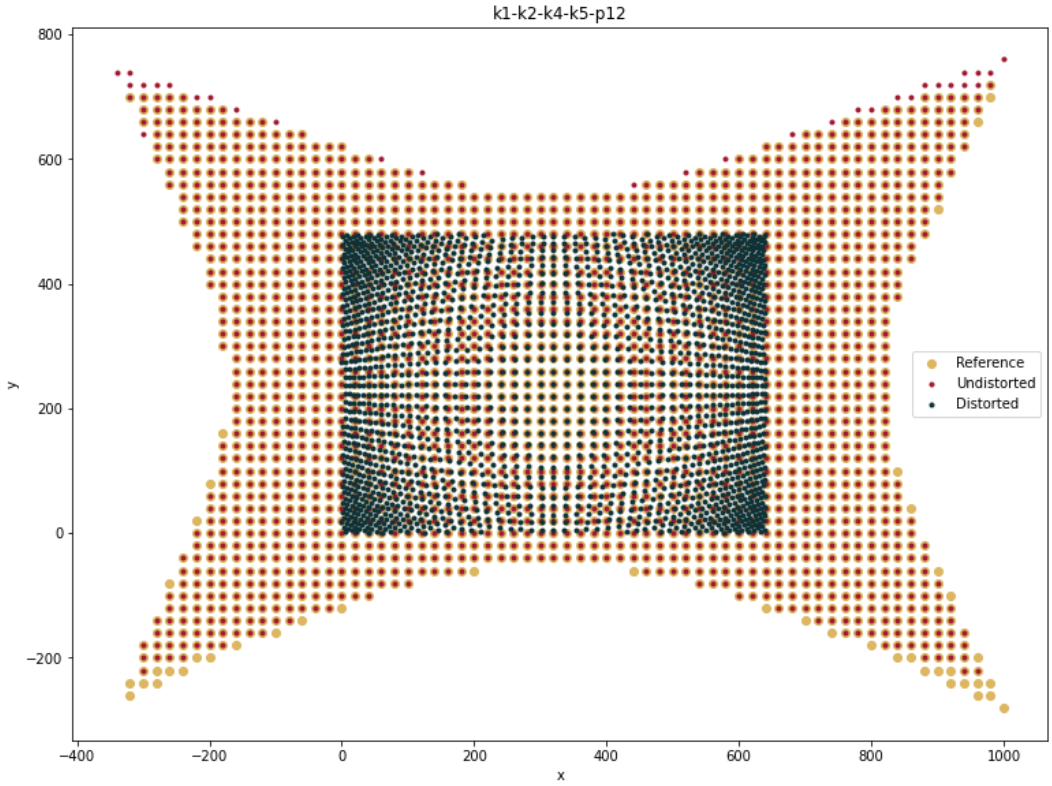


(a) Undistortion of an image

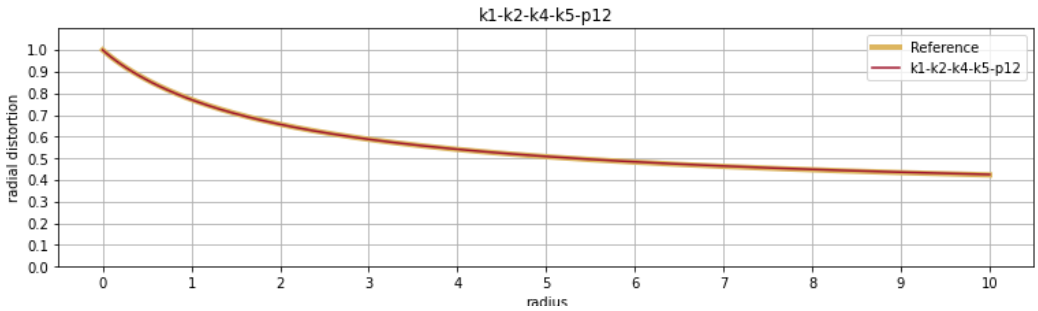


(b) Radial distortion

Figure B.11.: Distortion model with k_1, k_2, k_4 and p_1, p_2



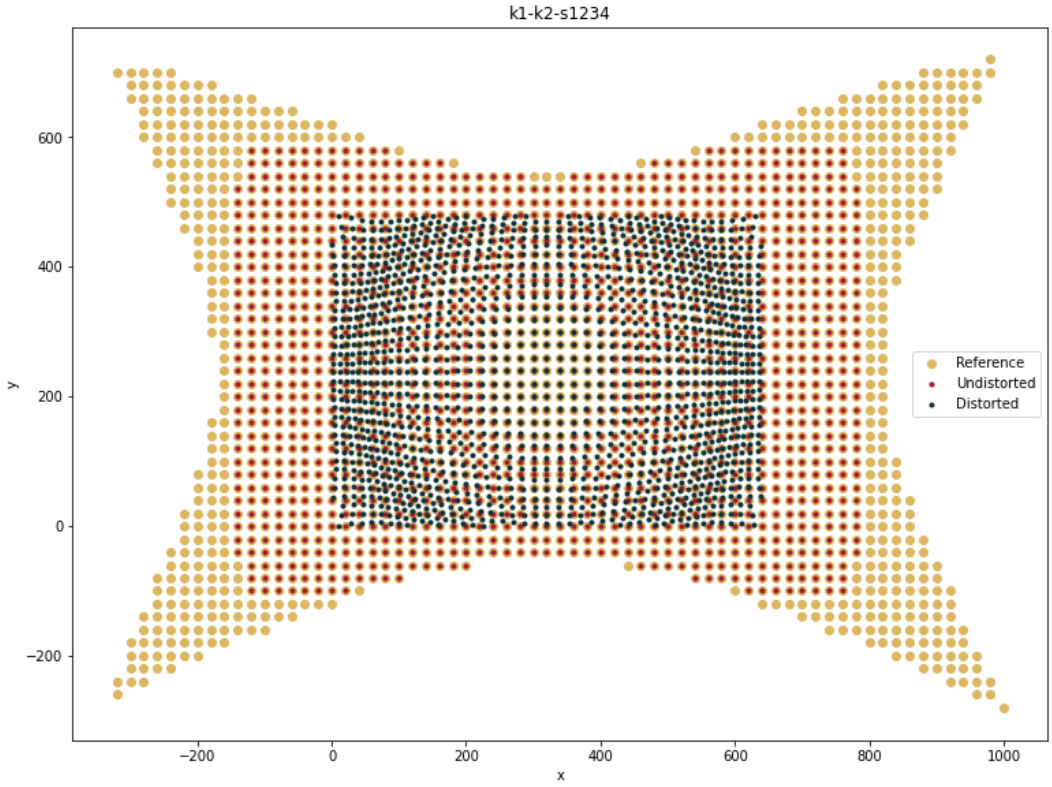
(a) Undistortion of an image



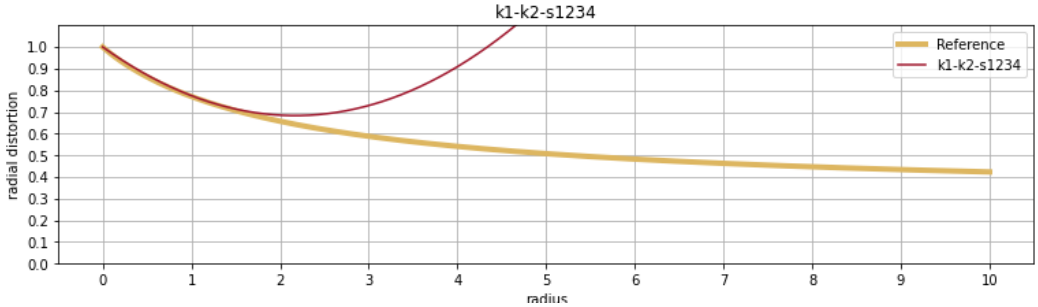
(b) Radial distortion

Figure B.12.: Distortion model with k_1, k_2, k_4, k_5 and p_1, p_2

B.3. Without tangential and with prism distortion

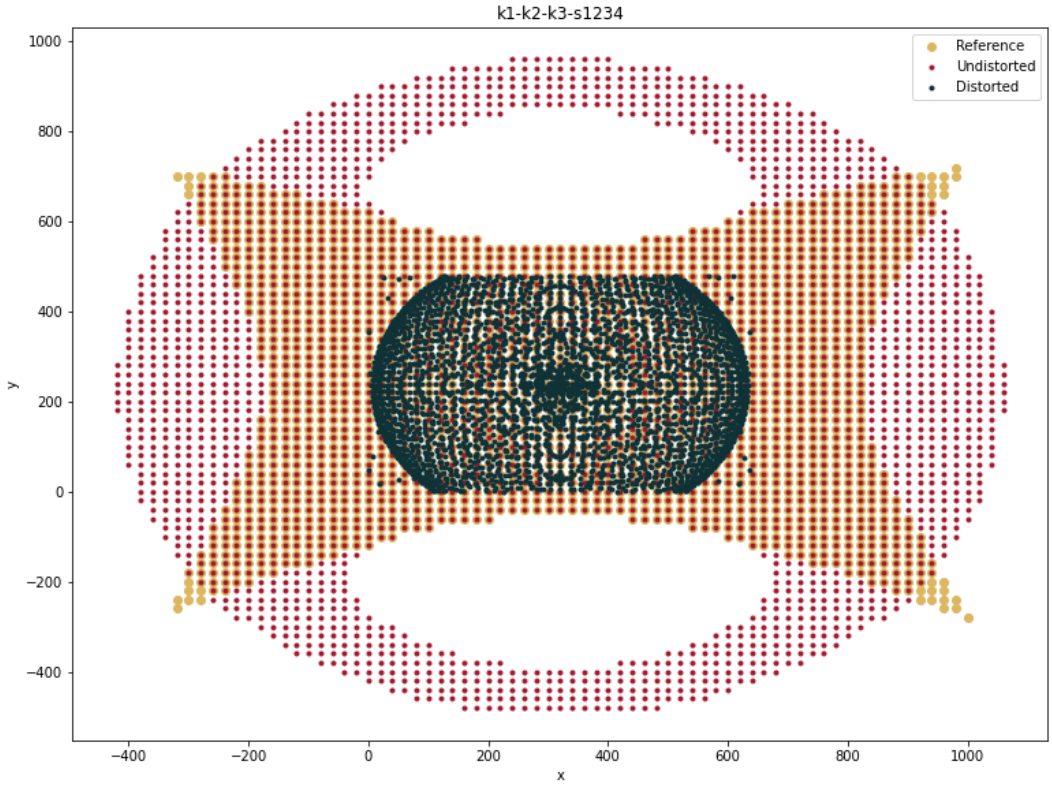


(a) Undistortion of an image

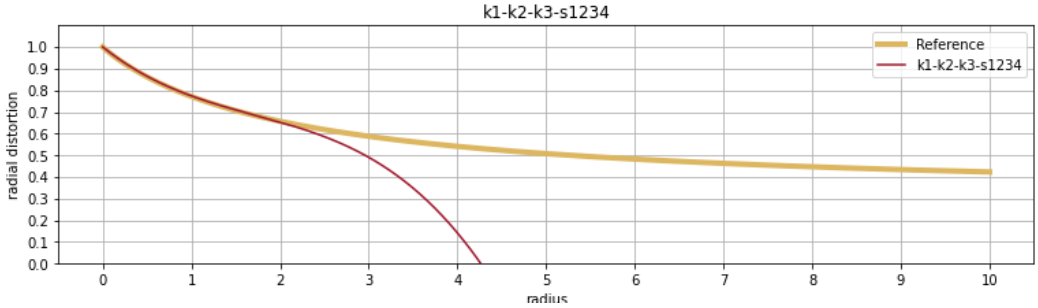


(b) Radial distortion

Figure B.13.: Distortion model with k_1, k_2 and s_1, s_2, s_3, s_4

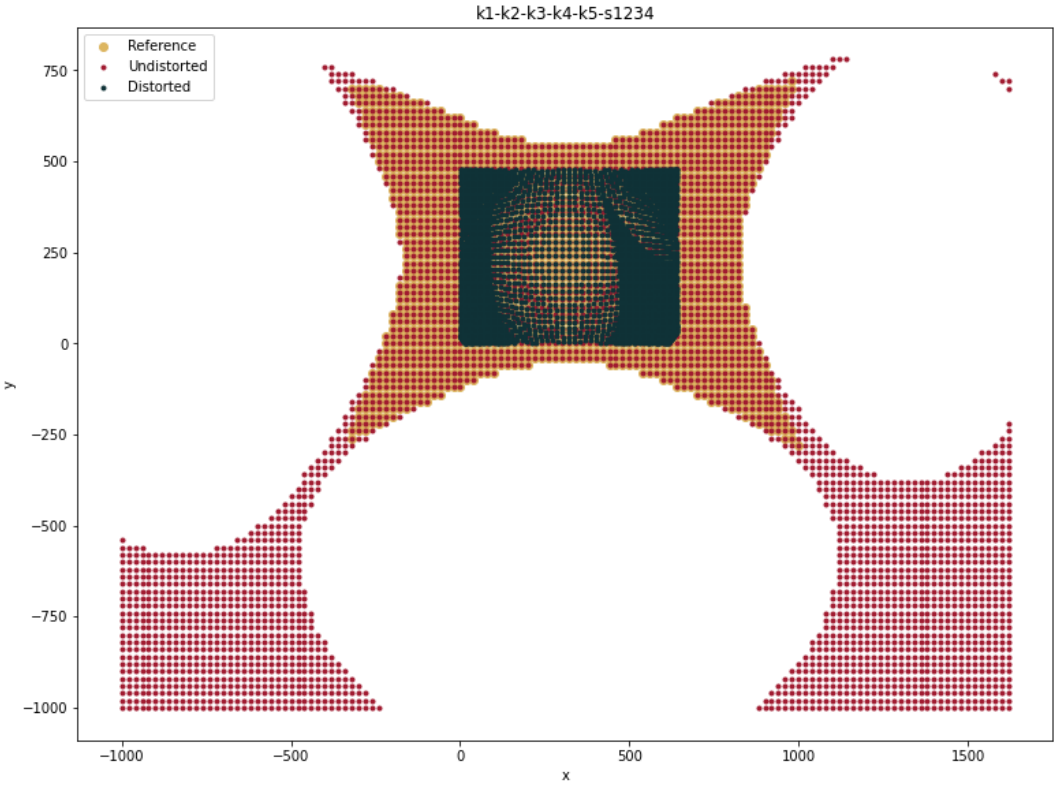


(a) Undistortion of an image

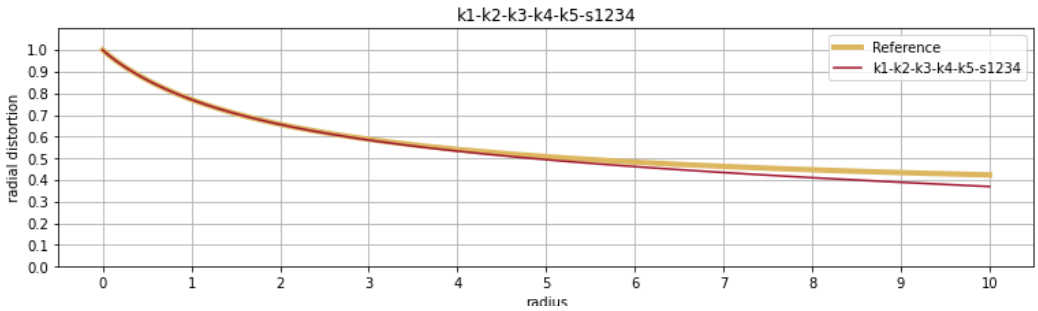


(b) Radial distortion

Figure B.14.: Distortion model with k_1, k_2, k_3 and s_1, s_2, s_3, s_4

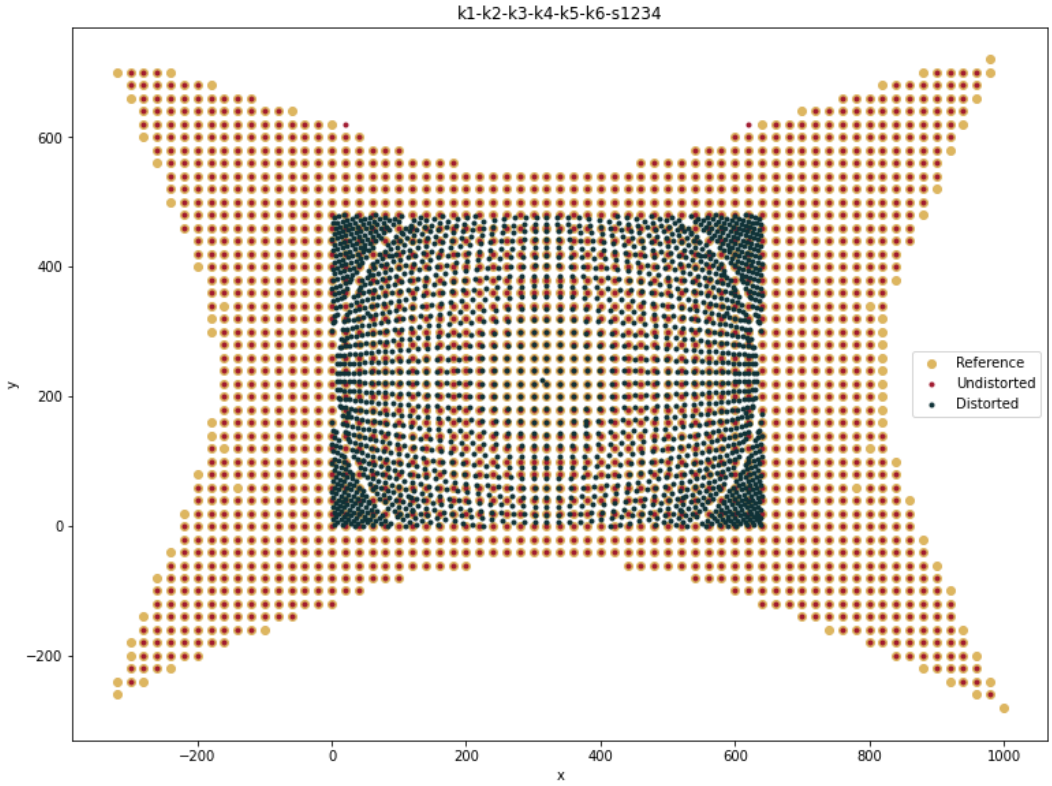


(a) Undistortion of an image

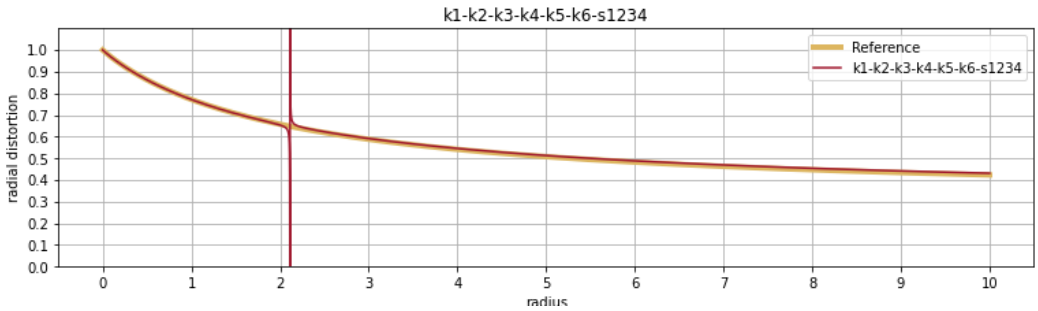


(b) Radial distortion

Figure B.15.: Distortion model with k_1, k_2, k_3, k_4, k_5 and s_1, s_2, s_3, s_4

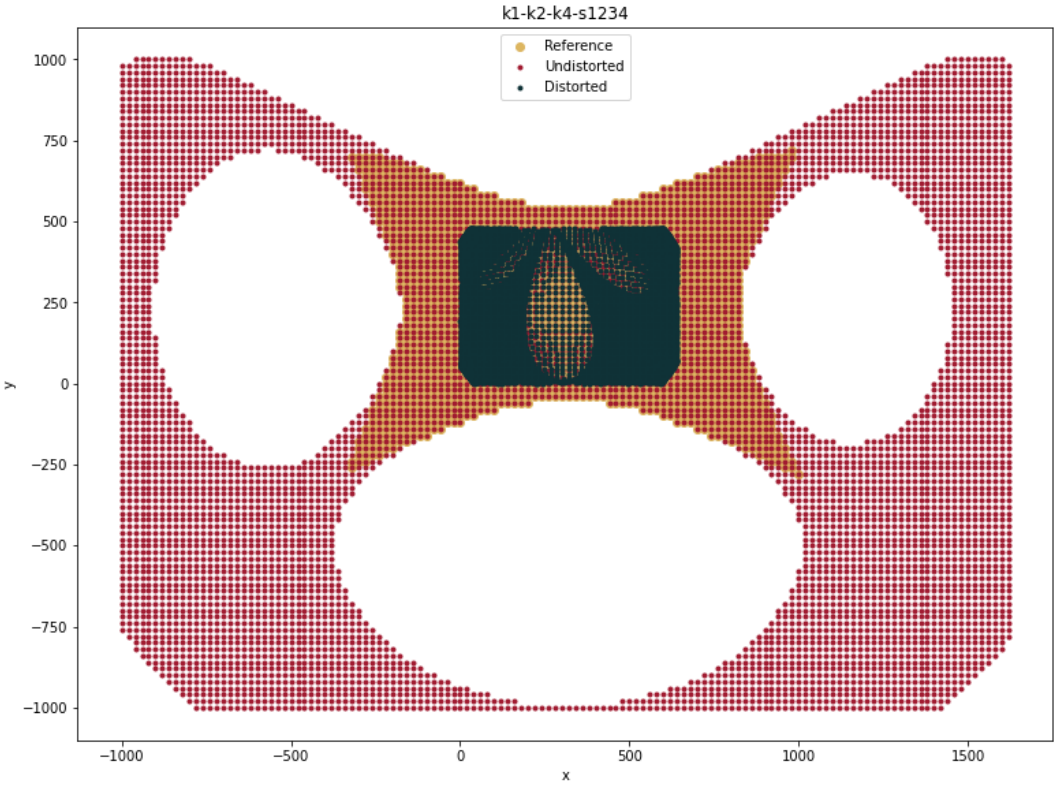


(a) Undistortion of an image

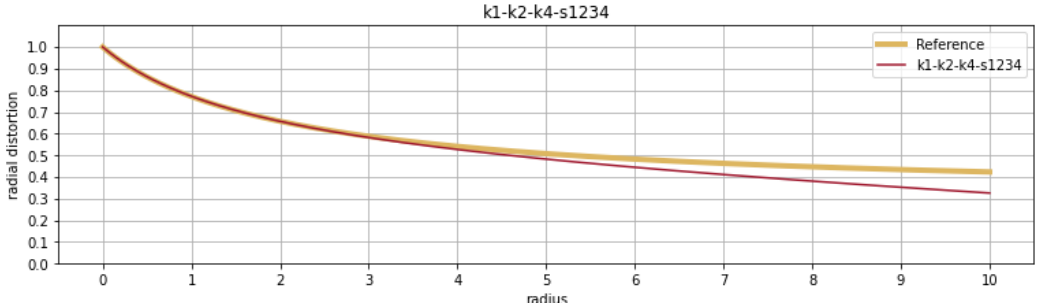


(b) Radial distortion

Figure B.16.: Distortion model with $k_1, k_2, k_3, k_4, k_5, k_6$ and s_1, s_2, s_3, s_4

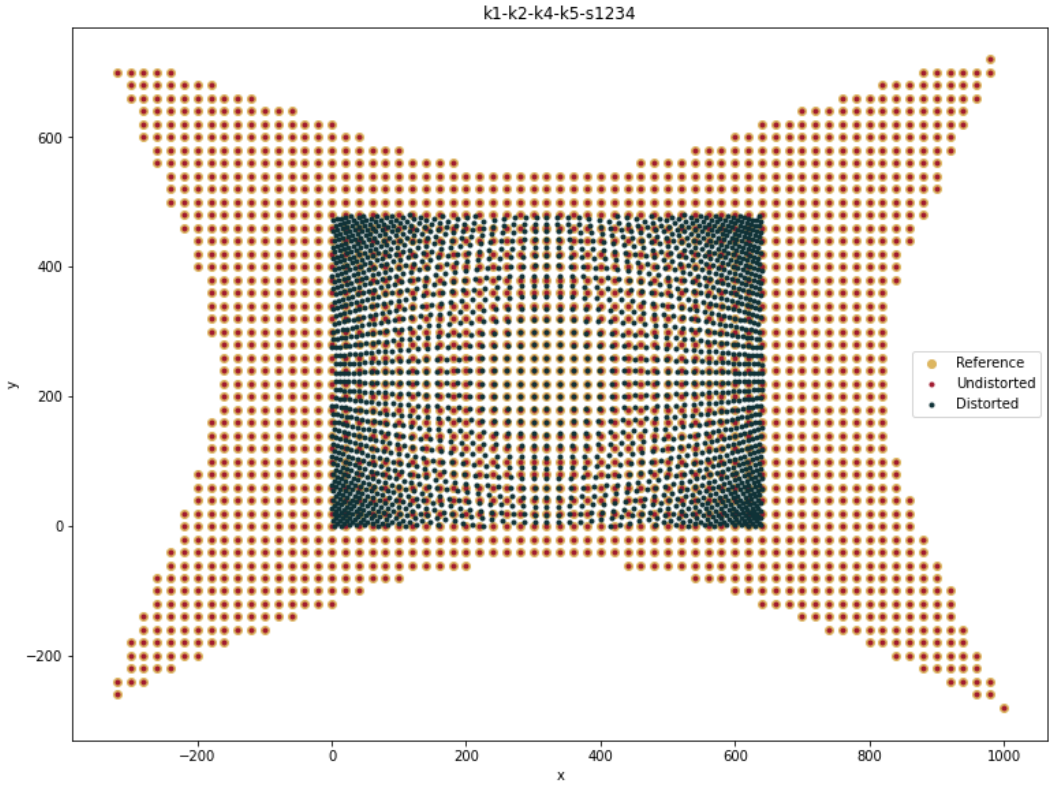


(a) Undistortion of an image

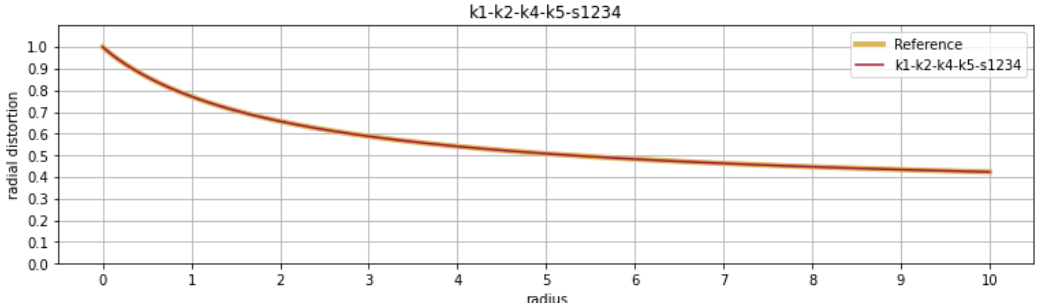


(b) Radial distortion

Figure B.17.: Distortion model with k_1, k_2, k_4 and s_1, s_2, s_3, s_4



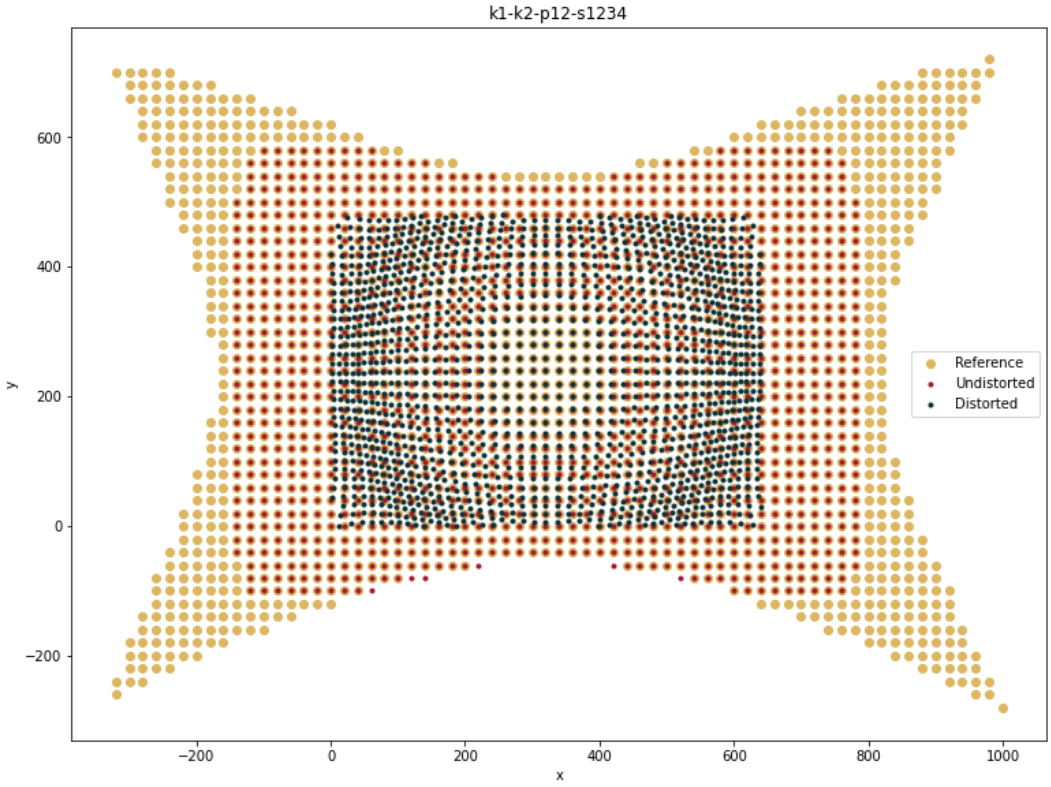
(a) Undistortion of an image



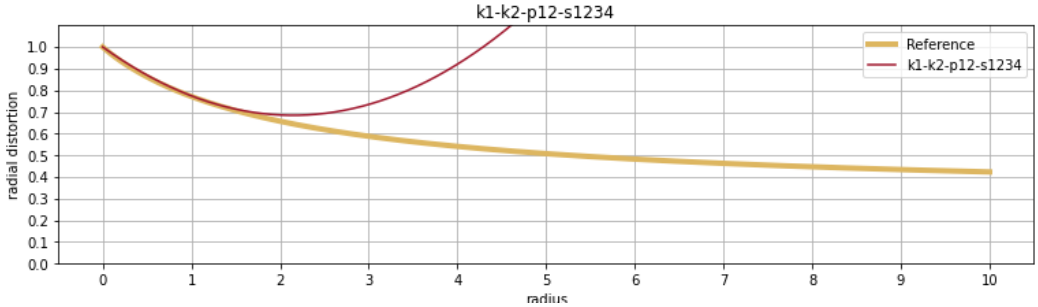
(b) Radial distortion

Figure B.18.: Distortion model with k_1, k_2, k_4, k_5 and s_1, s_2, s_3, s_4

B.4. With tangential and prism distortion

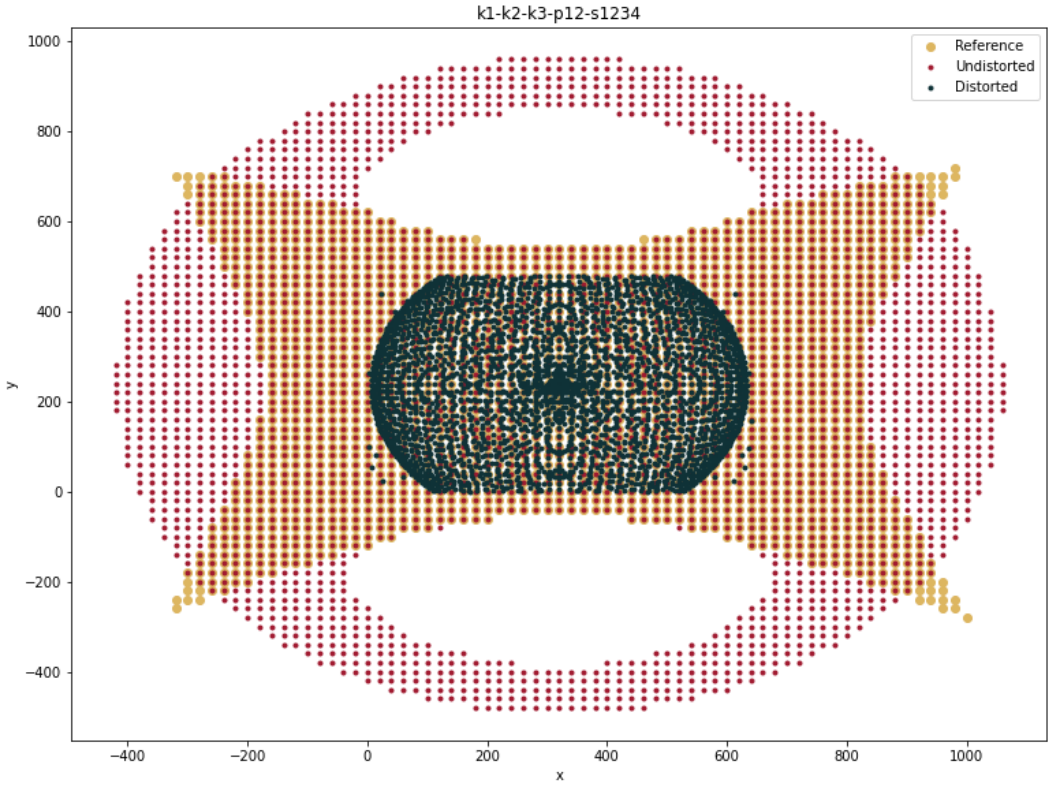


(a) Undistortion of an image

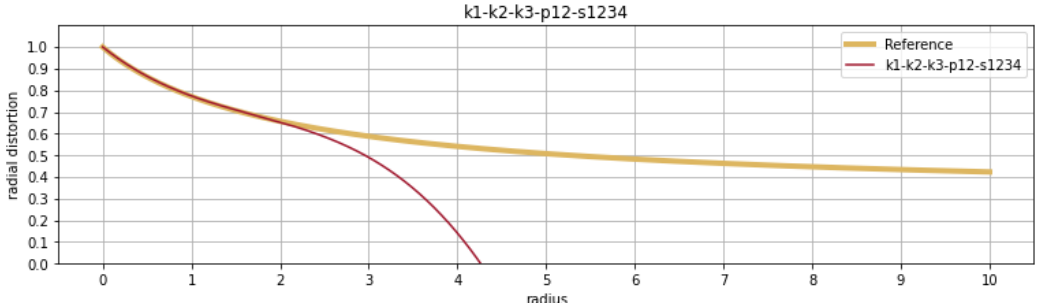


(b) Radial distortion

Figure B.19.: Distortion model with k_1, k_2 and p_1, p_2 and s_1, s_2, s_3, s_4

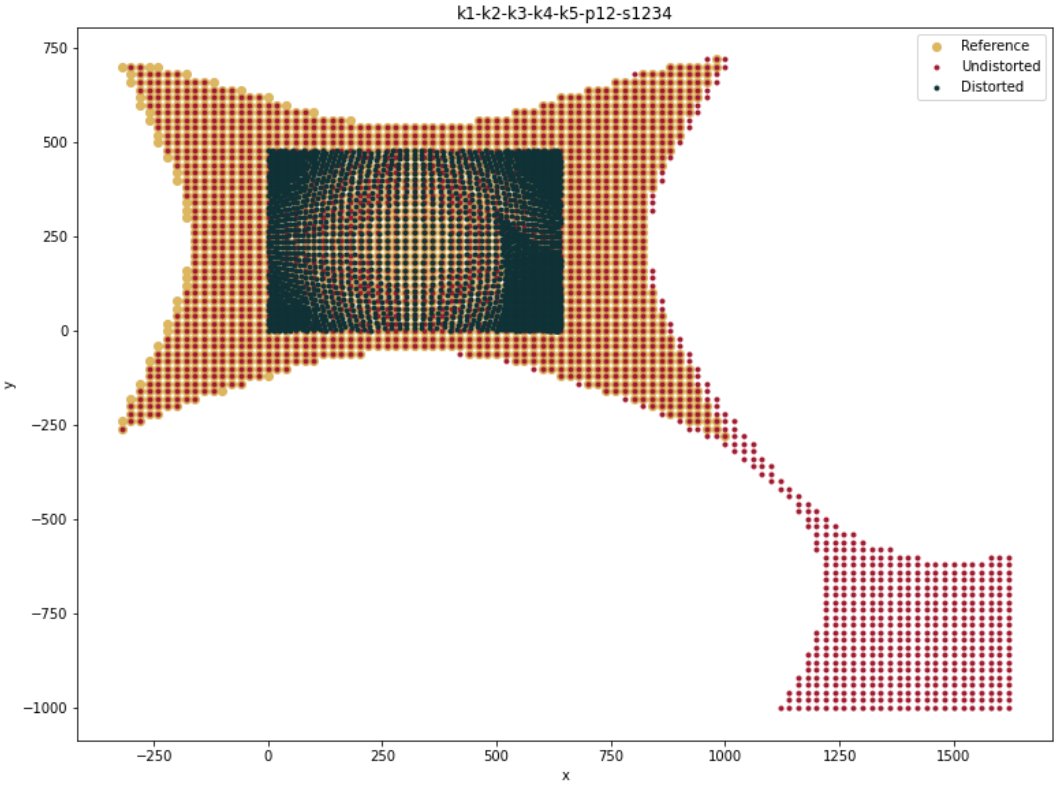


(a) Undistortion of an image

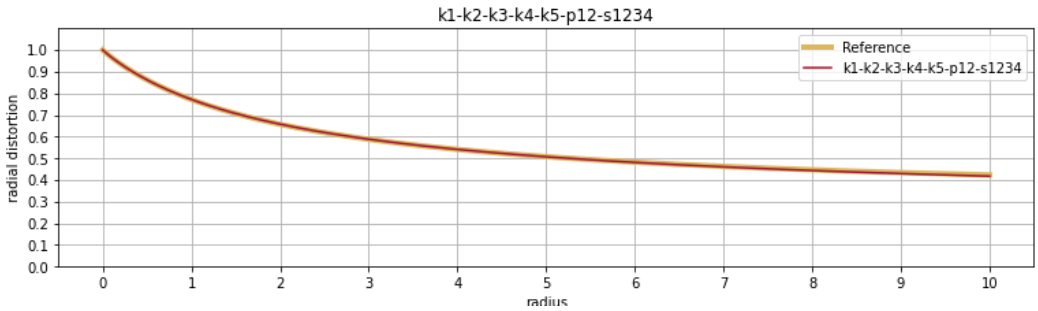


(b) Radial distortion

Figure B.20.: Distortion model with k_1, k_2, k_3 and p_1, p_2 and s_1, s_2, s_3, s_4

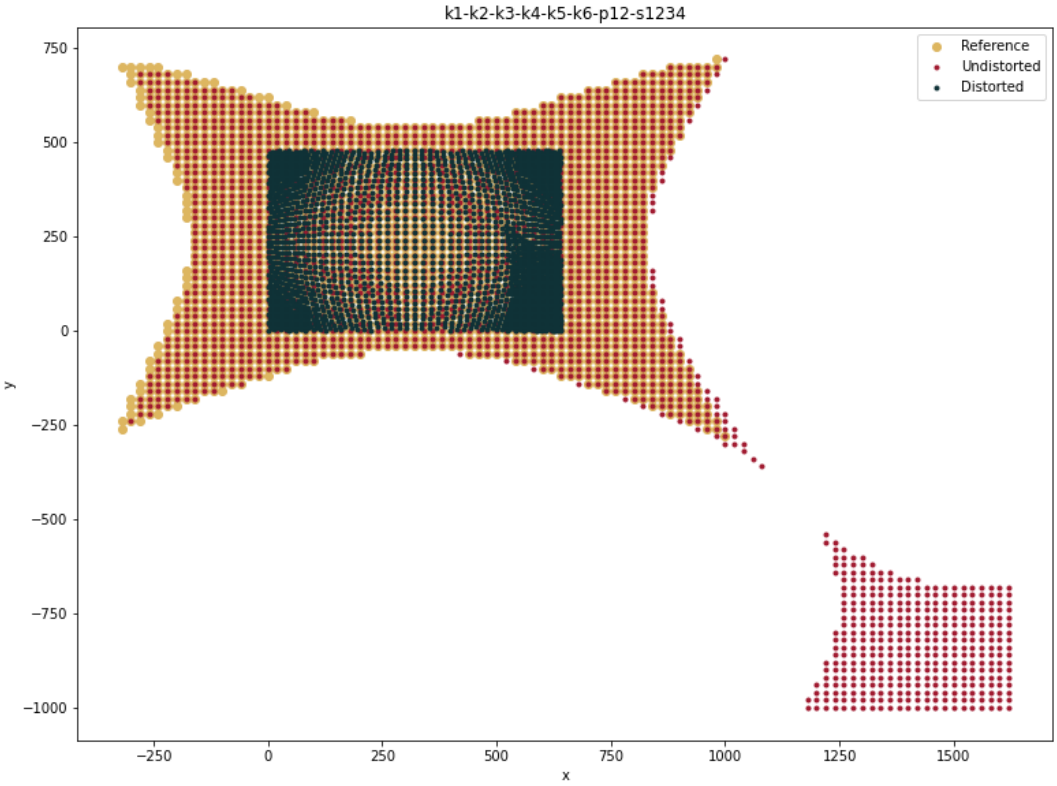


(a) Undistortion of an image

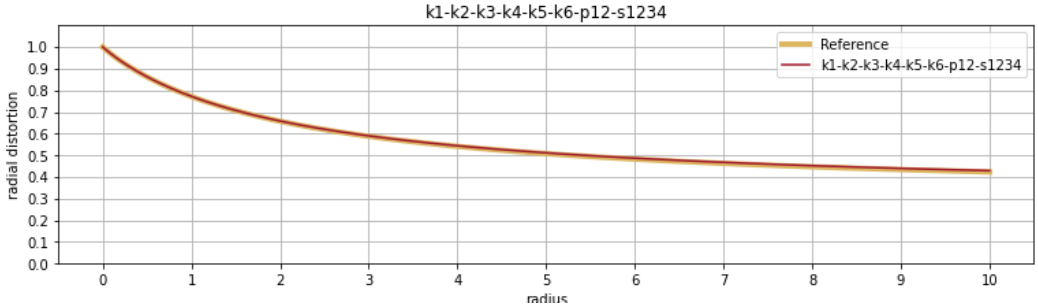


(b) Radial distortion

Figure B.21.: Distortion model with k_1, k_2, k_3, k_4, k_5 and p_1, p_2 and s_1, s_2, s_3, s_4

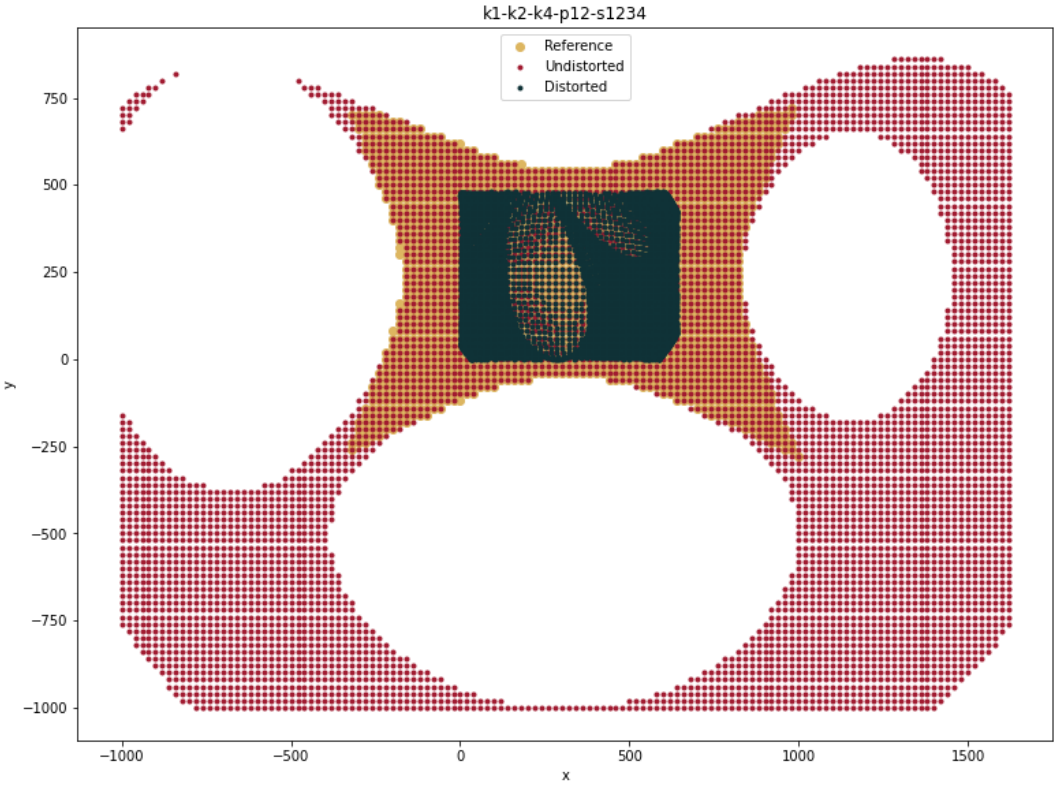


(a) Undistortion of an image

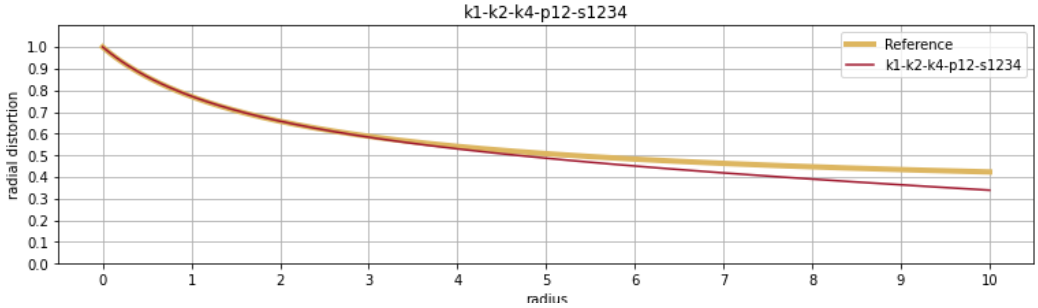


(b) Radial distortion

Figure B.22.: Distortion model with $k_1, k_2, k_3, k_4, k_5, k_6$ and p_1, p_2 and s_1, s_2, s_3, s_4

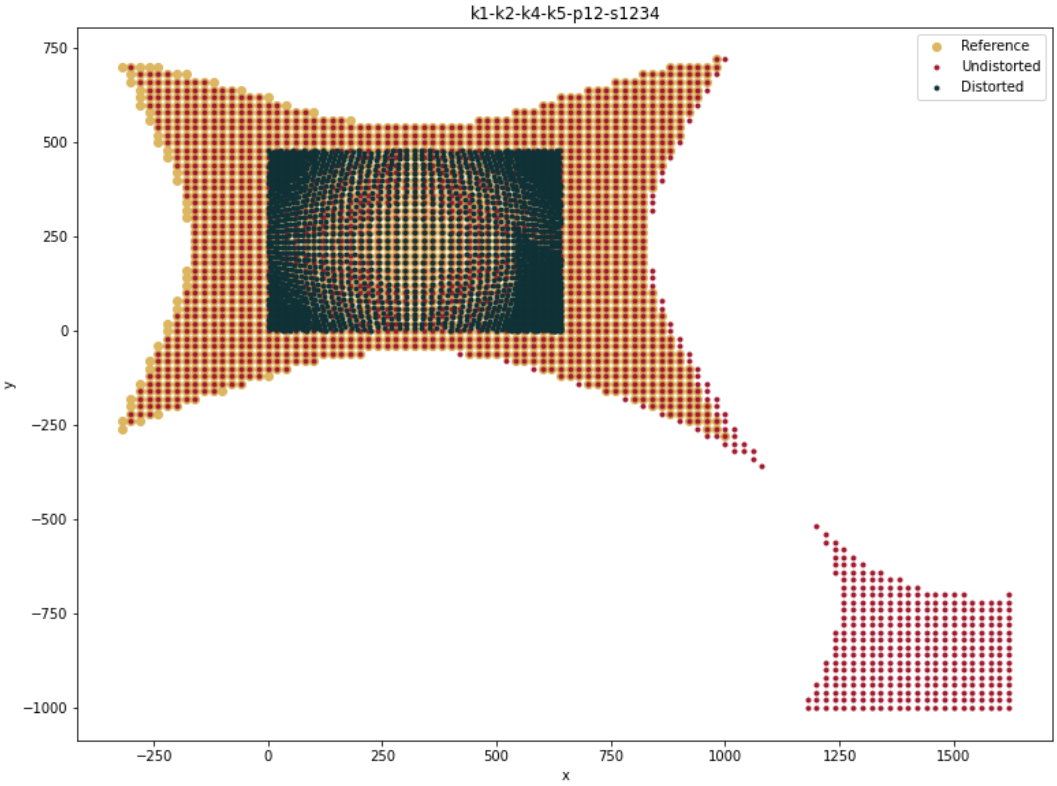


(a) Undistortion of an image

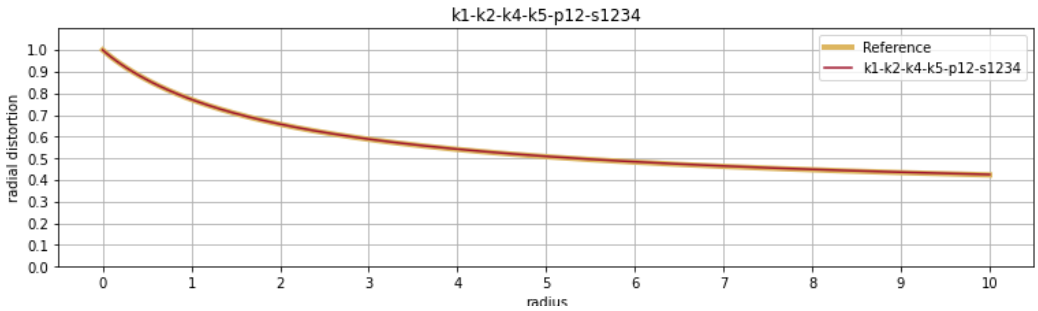


(b) Radial distortion

Figure B.23.: Distortion model with k_1, k_2, k_4 and p_1, p_2 and s_1, s_2, s_3, s_4



(a) Undistortion of an image



(b) Radial distortion

Figure B.24.: Distortion model with k_1, k_2, k_4, k_5 and p_1, p_2 and s_1, s_2, s_3, s_4

Appendix C.

Test of the full balldetector system

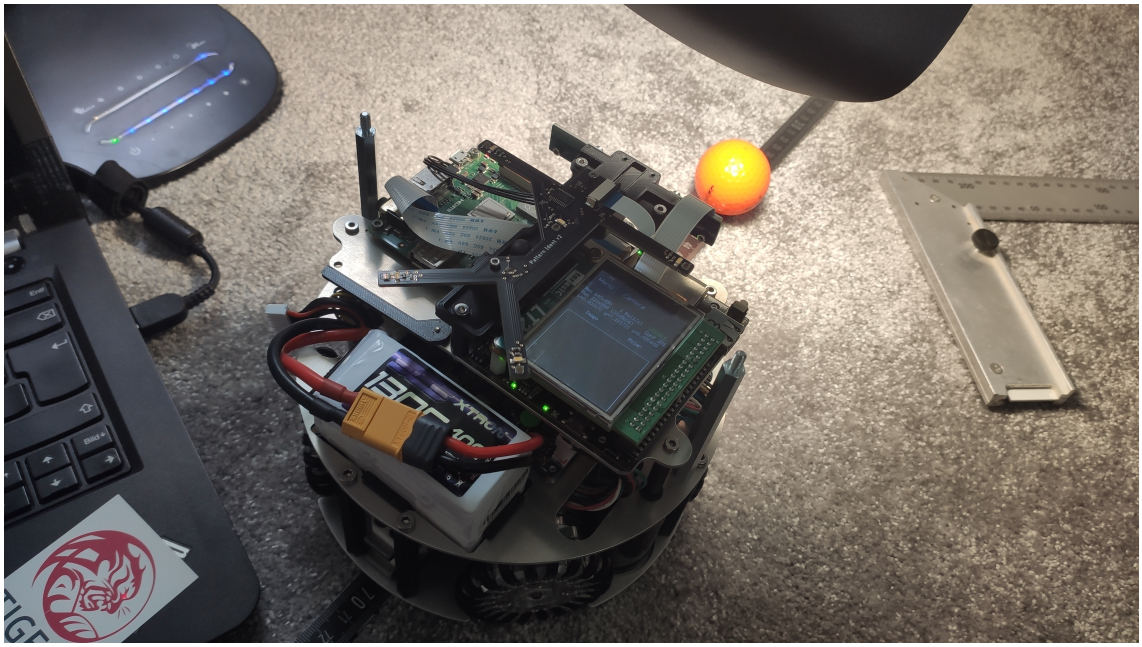


Figure C.1.: Test setup

Eight different positions were chosen to construct a test case for the whole system. The aim is to see how well the calculated three dimensional position matches the actual real world position. Therefore the three created undistortion models created in Section 6.4 are compared. The measurements are shown in Table C.1 and the average errors of each column combined over all three dimensions is for A -13.4mm ± 23.8mm, for B -12.8mm ± 22.3mm and for C -9.6mm ± 15.0mm.

real (x, y, z)			A (x, y, z)		
0.000	0.100	0.030	0.000	0.089	0.028
0.000	0.300	0.030	0.000	0.280	0.004
0.000	0.500	0.030	0.000	0.516	-0.026
0.050	0.100	0.020	0.046	0.098	0.019
-0.050	0.100	0.020	-0.046	0.094	0.016
-0.200	0.300	0.020	-0.192	0.288	-0.018
0.200	0.300	0.020	0.192	0.293	-0.080
0.000	0.300	0.090	-0.001	0.283	0.056
mean error			-0.0001 ± 0.0045	-0.0001 ± 0.0044	-0.0000 ± 0.0044

real (x, y, z)			B (x, y, z)		
0.000	0.100	0.030	0.000	0.089	0.028
0.000	0.300	0.030	0.000	0.280	0.005
0.000	0.500	0.030	0.001	0.516	-0.026
0.050	0.100	0.020	0.047	0.098	0.019
-0.050	0.100	0.020	-0.046	0.095	0.017
-0.200	0.300	0.020	-0.193	0.288	-0.018
0.200	0.300	0.020	0.191	0.293	-0.070
0.000	0.300	0.090	-0.001	0.283	0.056
mean error			-0.0074 ± 0.0104	-0.0072 ± 0.0104	-0.0071 ± 0.0105

real (x, y, z)			C (x, y, z)		
0.000	0.100	0.030	0.000	0.090	0.028
0.000	0.300	0.030	0.001	0.280	0.007
0.000	0.500	0.030	0.001	0.516	-0.022
0.050	0.100	0.020	0.047	0.098	0.019
-0.050	0.100	0.020	-0.046	0.095	0.017
-0.200	0.300	0.020	-0.193	0.288	-0.016
0.200	0.300	0.020	0.191	0.294	-0.005
0.000	0.300	0.090	-0.001	0.282	0.058
mean error			-0.0326 ± 0.0315	-0.0311 ± 0.0290	-0.0217 ± 0.0173

Table C.1.: Balldetector test measurements in m