Improvement of Self-Localization via Pose Correction Procedure in RoboCup

Semester Thesis

Simon Zimmermann
simonzi@ethz.ch

Computer Vision Laboratory
Department of Information Technology and Electrical Engineering
ETH Zürich

Advisors:
Louis Lettry
Alex Locher

Supervisor:
Prof. Dr. Luc van Gool

29th August 2017

Abstract

In RoboCup, humanoid NAO robots are used to perform 5-vs-5 soccer matches as a part of the Standard Platform League (SPL). In order to carry out soccer tasks like kicking the ball into the opponent goal or passing it to another team player, the robots need to know their position on the field. One of the few static objects on the soccer field are the field lines, which is why they are used as features for self-localization. The former implementation for detecting these lines was found to be unstable and noisy, which is why the robot's position could only be insufficiently updated. This led to the problem that the robots often lost track of their position while walking.
In the work presented in this thesis, the Random Forest machine learning algorithm is used to improve the detection of the field lines. The resulting line patch detections in combination with a prior robot pose are used to perform a Pose Correction Procedure. This method does not determine the robot's position out of the blue, but uses an estimate of it, which is modified in order to find the correct pose. The correction factors used for this modification are determined by applying the Direct Linear Transformation method.
Furthermore, a verification scheme that collaborates with an already existing particle filter is presented. It allows to keep track of the robot's position while it is moving. As this scheme provides a systematic criteria when the robot is lost, it can be well-combined with other methods that find a prior pose.
Test results show that the robot's position can be estimated more accurately and robustly in comparison to the former implementation, meaning that the robot does not lose track of its position so easily and quickly anymore. As a consequence the robot can kick and pass the ball more precisely to a target.

Contents

Acknowledgements

First of all I would like to thank Prof. Dr. Luc van Gool for the opportunity to take part in this project.

My sincerest thanks go to my advisors Louis Lettry and Alex Locher. Our meetings were of great value and helped me in understanding and solving the problems. I have learned a lot from the both of you.

Furthermore I would like to express my thanks to the teaching assistant Simon Flückiger, Simon Maurer and Maximilian Wulf, who were very supportive and assisted me patiently through the problems with the robots and the framework. I would also like to thank Marc Naumann for covering our backs at the tournament so that we could focus on coding.

Lastly I would like to further my thanks to the members of the NomadZ team for the smooth collaboration, especially Patrick Zhao, who has always been around in the Lab when there were open questions, Gerald Zhang for the good teamwork in Nagoya and Martin Kälin for his assistance with the Random Forest.

List of Figures

List of Tables

Chapter 1

Introduction

RoboCup is an international initiative that facilitates research in robotics and artificial intelligence through several different competitions. One of these is the RoboCup *Standard Platform League* (SPL) [1], where humanoid robots are used to perform 5-vs-5 soccer games. All participating teams have to compete with the same kind of robots, which means that the robot platform for each team is given and no hardware modifications are allowed. The robots have to operate fully autonomously during a soccer game without external control, neither by humans nor by computers. The rules of the competition are changed every year in order to make the technical problems more challenging.
In 2012, ETH Zurich started its own RoboCup soccer team as a joint effort of the *Automatic Control Laboratory* (IfA) [2] and the *Computer Vision Laboratory* (CVL) [3] at the *Department for Information Technology and Electrical Engineering* [4]. This team is currently named *NomadZ* [5].

One of the general challenges in the Standard Platform League is that the provided robots only have limited computational power, which is why it is very important to write efficient algorithms. The main challenge in solving the self-localization task is that all detectable objects on the soccer field look very similar through the robot's cameras. This is due to the high amount of whiteness that can be found in these objects such as the field lines, the ball, the goal, parts of opponent robots, even the shirts of the referees walking over the field. Objects lying around the field border can also disturb the robot. This makes it difficult to find reliable features which can be used for localization. Another point is that the soccer field is symmetric, which can cause the robot to become disoriented and play against its own field side.
Looking into the current framework of the NomadZ team, one can see that the information of the robot's position is used or required by nine different modules during a game. This makes it one of the most used pieces of information in the whole framework. The most important task the robot pose is used for is to align the robot behind the ball such that it can kick the ball in the direction of a team mate for a pass or the opponent goal to score points for its team.

The main problem with the previous self-localization implementation is that the performance of the line detection is quite noisy and unstable, which is why not enough features for a proper localization can be detected. This causes the robot to lose track of its position. More information will be given in section 5.1. The main goal of this semester thesis is to improve the robot's existing self-localization procedure in such a way that it is more accurate and robust in a sense that the robot doesn't get lost easily. In order to achieve this goal, the field lines are used as features for localization, as they are one of the few static objects on the soccer field.

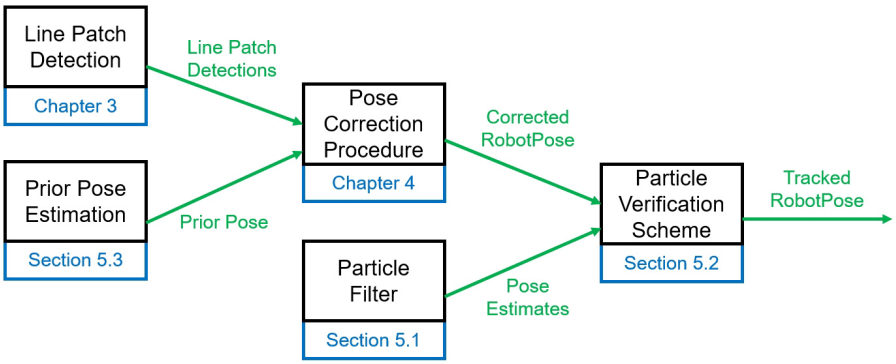

Figure 1.1: Chapter Overview

An overview of the chapters of this thesis is given in figure 1.1. In order to detect parts of the field lines, a method using the *Random Forest* machine learning algorithm is presented in chapter 3. As a next step, the resulting line patch detections and a prior of the robot's position are used to perform a *Pose Correction Procedure*, which is explained in chapter 4. This means that the method uses the prior pose as an initial estimate, which is then refined using the detected parts of the field lines. The prior position can be determined using a different approach developed by former student projects, which uses external features around the soccer field in combination with PnP pose estimation. It will be introduced in section 5.3. In a further step, the resulting corrected robot pose in combination with an already existing particle filter (see section 5.1) is used for keeping track of the robot's position while it is walking. This *Particle Verification Scheme* is presented in chapter 5. Additionally, an introduction to the used material is given in chapter 2. Finally, the results are presented in chapter 6 and a conclusion and a prospect of possible future work are given in chapter 7.

Figure 1.2: Image of a Standard Platform League game of the RoboCup World Championship 2017 in Nagoya (Japan)

Chapter 2

Materials and Methods

2.1 NAO Robots

The hardware currently used in the Standard Platform League are the so-called *NAO* robots, which have been engineered by the French robotics company *Aldebaran Robotics* and is nowadays enhanced and distributed by the Japanese company *SoftBank Robotics* [6]. The NAO (depicted in figure 2.1) is a 58 cm tall, 4.3 kg heavy autonomous, programmable humanoid robot equipped with a single core 1.60 GHz Intel ATOM Z530 CPU [7]. The NAO model used for RoboCup has 21 degrees of freedom for movement. For sensing it is equipped with several sensors, such as accelerometer, gyroscope, sonars and contact and tactile sensors [8]. Furthermore it has two video cameras in its head with one pointing downward and the other one straight ahead, which means that they don't have an overlapping field of view. When newly ordered, the NAO is delivered with its own software package, which includes its own specialized Linux-based operating system called *NAOqi* [9] and a SDK [10] including drivers and examples. This package delivers the basic functionalities in order to control the NAO. To include further applications to perform soccer tasks, most RoboCup teams develop their own software platform on top of the NAOqi. Further and more detailed information can be found in the NAO Documentation [11].

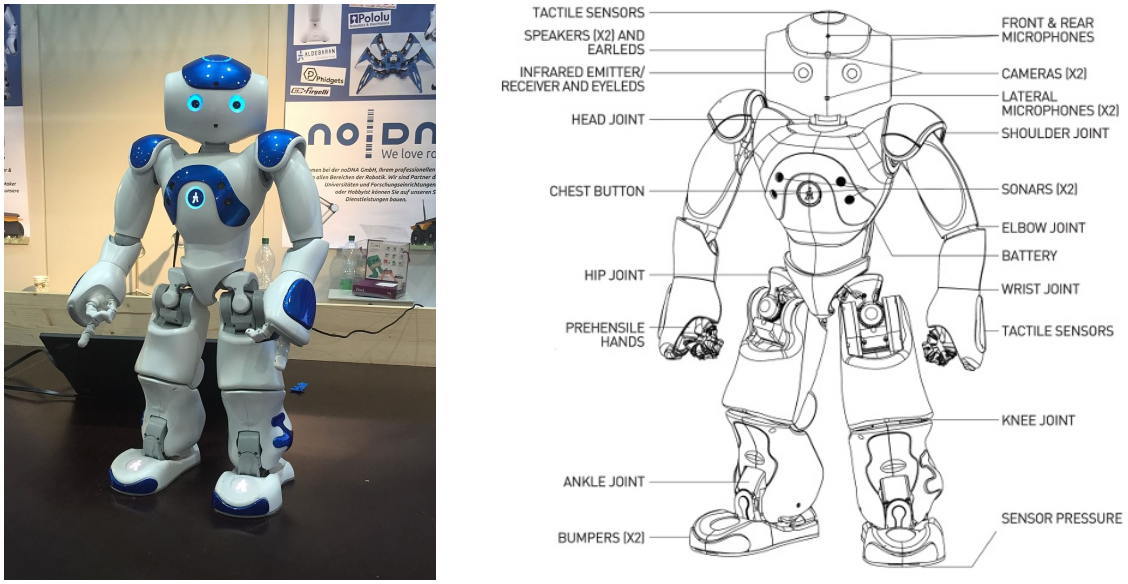

Figure 2.1: Left side: Picture of the NAO robot at RoboCup 2016, Source: [12]; Right side: Sensor and joint overview of the NAO (body type H25), Source: [13]

2.2 B-Human Framework

One essential rule of RoboCup is that the four best teams of the year have to publish a stripped down version of their code. The idea is that new teams get a faster entry into the tournament as they can focus on their own high level implementations and don't have to invest a lot of work into low-level calls and drivers for sensors and actuators. As like many other younger RoboCup teams, NomadZ relies on the so-called *B-Human* framework developed by the team of the University of Bremen in Germany [14]. NomadZ uses the B-Human code release of 2013 [15] as a basic framework and has implemented some of its own algorithms since then.
The B-Human framework is programmed in C++. One of its most important features is its modularity, which means that functionalities that belong together can be summarized into so-called *Modules*. Due to this kind of architecture, the programmer can benefit from simple extensibility and reusability of the code. As the NAO is only equipped with a single core processing unit, it does not support parallel processing. In order to guarantee a Module integration without errors, the B-Human framework contains a scheduler that handles the sequence of the different Modules. The exchange of data between Modules can be implemented by so-called *Representations*. These kind of classes are serializable, which means that they can be transformed into bit streams that are sent and stored over a transmission channel. For additional simplification of the implementation of Modules and Representations, the B-Human framework provides several *macros*.
The *Behavior* of the NAO decides the order of the tasks that have to be fulfilled. To handle this the framework provides a behavior engine that implements hierarchical state machines. It consists of a programming language called CABSL, which provides four main features in order to facilitate the development of the state machines: *Options*, which implement finite state machines and can include other options; *States*, which can contain transitions to other states and actions; *Transitions*, which are used to get from one state to another within one option; *Actions*, which can call upon other options or execute C++ code.
Another very important feature of the B-Human framework is the software *SimRobot*, which runs on the computer. It fulfills two main tasks: On one hand it is the interface to communicate with the NAO robots, on the other hand it can be used as a simulation tool. The program provides a logging functionality that can record a sequence of robot actions and the corresponding sensor data and save it in a log file. It can be replayed at a later point in time in order to test different code implementations and parameters on the exact same setup. Figure 2.2 shows an example of the graphical user interface of SimRobot. The connection from computer to robot can be done either by Ethernet or by Wifi.
Further and more detailed information can be found in the B-Human team report and code release 2013 [15].

2.3 Self-Localization

Self-localization of the NAO is a central part of the robot's *cognition process*. It contains the process used to determine the position of the robot on the field. Its location is defined by three degrees of freedom, which are described by two translational coordinates and one rotational coordinate. Therefore, the robot's field position can be described by the 3-tuple (α, x, y). They are measured relative to the fixed field coordinate system, which is located at the middle point of the soccer field. How this coordinate system is defined is depicted in figure 2.3.
In the B-Human framework, the self-localization procedure is performed in the module *SelfLocator*. The resulting information of the robot's position is streamed to the other modules using the representation *robotPose*.

Figure 2.2: Example screenshot of the SimRobot user interface: (1) Scene Graph to select required data; (2) Console for commands; (3) WorldState to track positions of objects on the soccer field; (4) Images of robot's upper and lower camera; (5) Data of representations

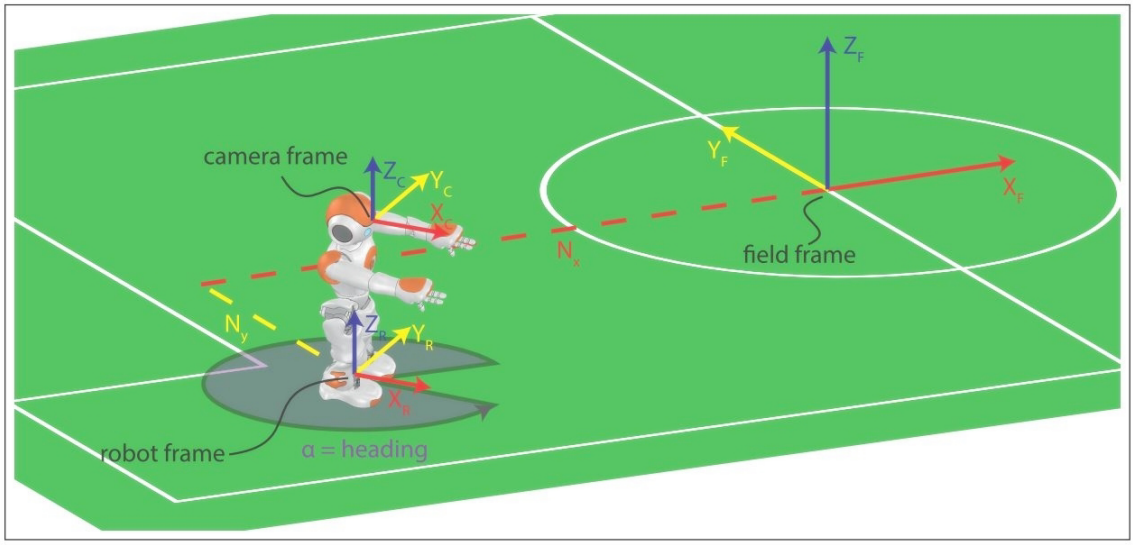

Figure 2.3: Definition of the different coordinate systems: Field frame, camera frame and robot frame. The parameters α, N_x and N_y expressed in field frame are used to describe the robotPose; Source: [16]

Chapter 3

Line Patch Detection

In this chapter, the detection of parts of the field lines with the help of the Random Forest algorithm is presented. The resulting line patch detections can be used in a next step as features for localization. The general goal of this method is to classify whether a patch consisting of a certain pixel size of an image received from one of the robot's cameras contains a field line or not.
Originally, the Random Forest method was implemented in the NomadZ framework for ball detection in a former student project [17]. Some additional work has been done in the thesis presented here in order to adapt the procedure to detect the field lines. Random Forest has been found to be the right classification method for RoboCup as it has a good runtime performance. This is essential as it has to run online during a match on the computationally restricted NAO robots.
In the first section 3.1, the training procedure of the Random Forest is explained together with the used parameters for the tree growing. The second section 3.2 is about the creation of the data set used for this training.
Further information about the Random Forest method in general and for ball detection can be found in [17] and [18].

3.1 Random Forest Training

Random Forest is a supervised machine learning algorithm which can be used to solve classification and regression tasks. The method consists of a set of decision trees, where the samples that are used to grow these trees are randomly selected from a training data set. Each sample contains the features of the allocated class one wants to identify. A scheme of how a structure of a Random Forest tree looks like is depicted in figure 3.1.
A tree is grown from the root node using the entire set of training samples. At each tree node, split functions for splitting the samples into two subsets are randomly generated several times. Each of these split functions executes a binary test of one or more randomly selected features out of the training samples arrived at the corresponding node. These selected low level features are values of one of the three image channels in YCbCr colorspace. The implemented split functions generally work by comparing the pixel value difference of two of these image features. The performed binary test decides whether a sample is sent to the right or the left child node. The split function with the best performance is selected for the current split node. It is the one that leads to the lowest entropy of the sub nodes after splitting the set of samples. This procedure is repeated for each child node until a stopping criteria is fulfilled. Three stopping criteria can be set via input parameters of the Random Forest trainer. These are the maximal depth of the tree, the minimum number of samples needed for a split and the minimum number of samples arriving at a leaf node after the split. An additional criteria is when a node has an entropy of zero. This means that it contains only samples of one single class, which is why no more splitting is necessary.

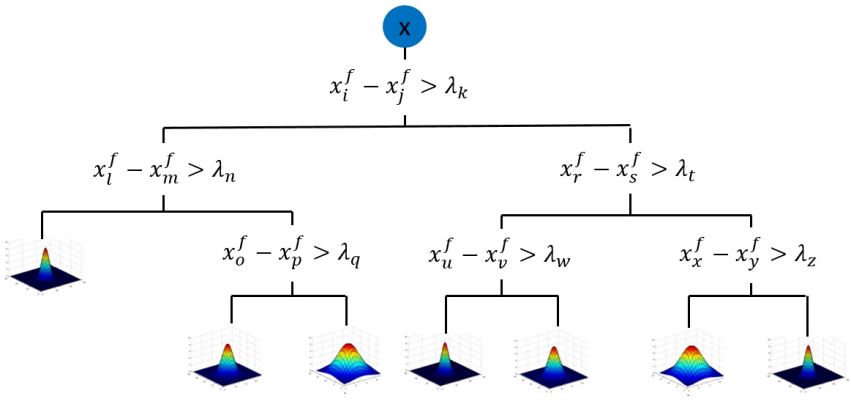

Figure 3.1: Scheme of a Random Forest tree with binary tests at each split

Nodes for which no more splitting is possible according to one of these stopping criteria are called *leaf nodes* and they store the probabilities of each class based on the samples it has received during the training procedure.
The high amount of randomness in this procedure allows the handling of a large amount of training data without overfitting, which would lead to a poorer classification performance.
After the trees have been generated, they can be used for identification of the classes that have been trained. To achieve this, a chosen sample is passed through each tree of the forest, starting at the corresponding root nodes. At each split node, the chosen split function decides whether the sample is sent to the left or the right sub node. Once the sample has arrived at a leaf node of each tree, the probabilities of the classes stored in these leaf nodes are averaged. The sample gets classified with the class belonging to the highest averaged probability.
The performance of the trees can be evaluated using a test set containing samples which are similar to the ones used for training. These test samples are fed to the generated trees for classification. Based on the results, a *confusion matrix* is created where *recall* and *precision* for each class and the overall *accuracy* of the matrix are calculated. Recall is the fraction of samples that are correctly detected from all samples belonging to the corresponding class, while precision is the fraction of samples correctly detected out of all predicted samples of that class. The overall accuracy is the average of all precision and recall values. Examples of such confusion matrices can be found in the tables 3.1 and 3.3. They indicate how well the generated trees perform in general and which classes are often confused.
The Random Forest trainer used by the NomadZ team to generate and test the trees is a console application implemented in C++ and has been developed during the semester project of [17]. One adaption of this tool has been made during the thesis presented here: The training set and the test set have been clearly separated from each other and are given to the function as two different inputs. The former implementation randomly extracted 5% of the training set for the test set. Therefore the two sets were very similar to each other, which led to a statistic that was much better than the reality. With the implemented change, the training and test set can be generated from different images, which makes the results of the confusion matrix more reliable.

3.1.1 Training Parameters for Line Trees

The parameters used to grow the set of decision trees are important for the Random Forest procedure, as they have direct impact on the performance of the algorithm. The choice of these parameters have been inspired by the work of [17] and have been further tuned in order to optimize the detection of the field lines. The input parameters which have been found to be the best to generate trees for line patch detection are:

- **Number of training patches for each class:** For the training set, approximately 15'000 patches for each class have been randomly extracted from the annotations. Additionally, about 750 patches (this has been chosen empirically to be 5% of the training patches) have been extracted from different annotations for the test set.

- **Number of trees:** The number of trees have been chosen to be nine based on the performance evaluation of [17].

- **Maximum depth of trees:** The maximum tree depth has been chosen to be eleven. This means that the training set gets split at maximum $2^{10} = 1024$ times, which leads to at least $15000/1024 \approx 15$ patches per leaf node.

- **Minimum patches at split and leaf nodes:** These numbers have been set to 20 for split nodes and to 10 for leaf nodes, respectively, based on the results derived in [17].

- **Number of random tests:** This parameter decides over the number of randomly generated split functions for each node. Setting this number to 1000 has shown reasonable results.

- **Patch size:** The best patch size for the detection of the field lines have been found to be 32x32 px (see section 3.2.2).

In order to guarantee a diverse training data set in order to avoid overfitting, between 500 and 800 images of the soccer field were collected and annotated. Images have been taken from the robot's lower and upper camera. It is important that the images are recorded from different viewing angles and distances to the objects that we want to classify in order to ensure a robust detection during the various situations which can occur during a game. It has also been found to be a good idea to merge different training sets from different locations and lightning conditions together, as this increases the robustness of the detection a lot.

3.2 Creation of Data Set

3.2.1 Annotation and Patch Extraction

As Random Forest is a supervised machine learning algorithm, we have to manually teach the system how a specific class we want to identify looks like. We call this process *annotation*. To do that, the NomadZ team uses a MATLAB tool where figures can be drawn into images collected from the soccer field in order to frame a certain object. Afterwards, a specific predefined class can be assigned to these objects. An example of how this looks like is shown in figure 3.2. Each color represents a different class we want to train. Any image area without annotation will be assigned automatically to the background / negative class.
The MATLAB tool has been developed by [17] and further evolved during this thesis. For example, the ability to draw polygons and ellipses and not only rectangles into the image has been added. This enables proper annotation of the field lines and also of the other classes with more precision. Furthermore, one can merge several training sets together in order to get a larger set for the tree growing. The number of training samples taken from a certain training set in comparison to another set can be chosen at will. This offers the possibility of tuning the content of the training set of the trees for a specific use.
For acquiring images from the soccer field, the module *ImageAcquisition* developed by [17] can be used. It saves images from the robot's cameras together with the corresponding camera matrix in regular time intervals.

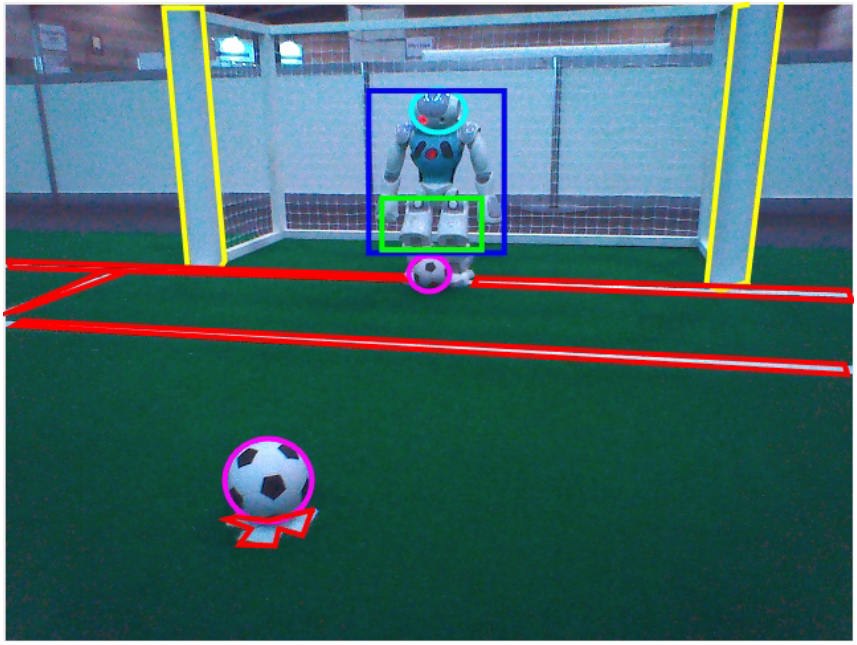

Figure 3.2: Example of an annotated image using NomadZ' MATLAB annotation tool

After all the images have been annotated and the objects have been assign to the corresponding classes, they can be used to create a training set for the learning process. This training set consists of small image patches that are generated by randomly extracting patches with a certain pixel size from the annotated objects. An example of how this looks like can be seen in figure 3.3 based on an annotated ball. Before this extraction process is executed, one can decide how many patches should be extracted from the annotation boundary and how many from the object's inside. This percentage number can be tuned for each class separately depending on where the most class specific features are.

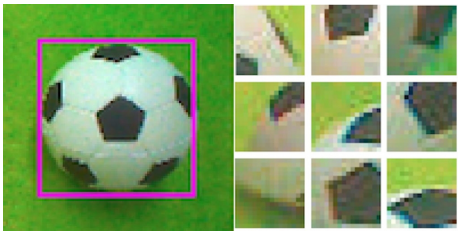

Figure 3.3: Example of patch extraction from a ball; Left: Annotated ball; Right: Resulting randomly extracted patches with a patch size of 16x16 px; Source: [17]

3.2.2 Line Classes and Patch Size

From the former project [17], we already have three classes to identify using Random Forest: *ball*, *robot* and *goal*. Additionally, there is a class *negative* covering all the background things appearing in the robot's image which we don't want to identify, for example the green carpet, the walls and the commercials around the soccer field. As already mentioned, the class we want to identify in order to improve the self-localization is the field lines. This is why we added the following four classes: The straight *lines*, the center *circle*, the line *crossings* and the *penalty* points. An overview of these classes is given in figure 3.4.

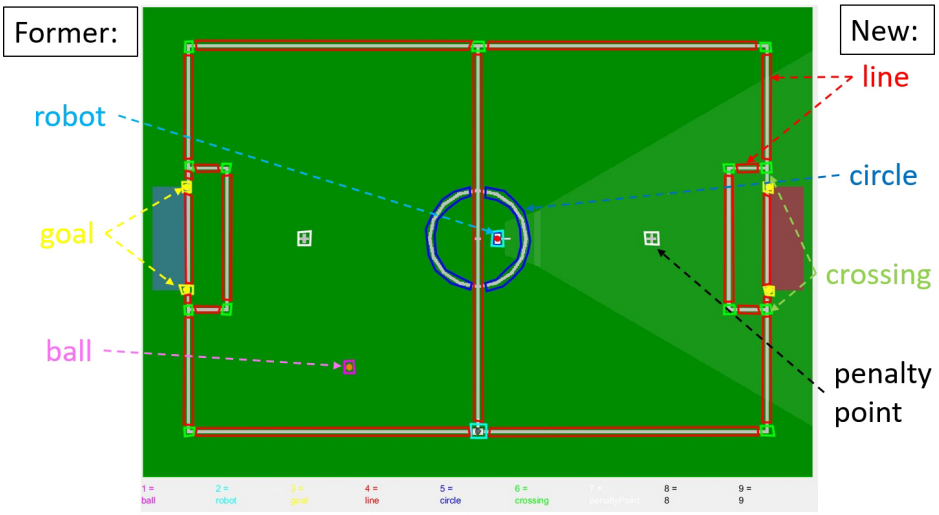

Figure 3.4: Overview of classes for Random Forest detection

Field lines are separated into four different categories because we are interested in how well the Random Forest can distinguish between these very similar looking classes. Additionally, three different patch sizes have been tested for line patch detection: $16 \times 16\ px$, which is currently used for ball detection and two larger patch sizes $24 \times 24\ px$ and $32 \times 32\ px$, as the expectation was that a larger area would present more line specific characteristics for the Random Forest.
The results of training the trees with these eight different classes and running a test set over them are representatively shown in table 3.1, which presents the generated confusion matrix. The results have been averaged over ten runs using patch size 32. The parameters used to train these trees are the ones presented in section 3.1.1 except for two values: The number of training patches for each class was set to only 6'000 as it was not possible to extract more random patches out of the small annotations corresponding to the classes line crossing and penalty point. As a consequence, the maximum depth of trees was adapted to ten.
When looking at the values of table 3.1, we can see that there is a lot of confusion between the different line classes, mainly between line and circle, line and crossing and penalty point and crossing. Also, the two classes crossing and penalty point are mixed up with the background a lot. Having a look at table 3.2 where precision, recall and accuracy for all classes and patch sizes are summarized, we can see that this behavior is similar for all tested patch sizes. What we can also see on this table based on the results of the accuracies is that the best results have been found using the largest tested patch size 32.
As the training of these eight different classes did not provide very satisfying results, we decided to merge the classes together so that we ended up with the following two classes:

- Negative = Negative + Ball + Robot + Goal
- Line = Line + Circle + Crossing + Penalty Point

New trees were trained and tested with these two classes using the same three different patch sizes. The corresponding results are presented in the tables 3.3 and 3.4. The overall results proved positive, with an accuracy over 90% when testing the newly generated trees directly on the robot. As a consequence, we decided to go with these two classes for line patch detection. Additionally, based on the results presented in table 3.4, patch size $32 \times 32\ px$ was found to be the best in order to detect the field lines.

Table 3.1: Confusion matrix of all classes with patch size 32x32 px

	Ground Truth								
Prediction	Negative	Ball	Robot	Goal	Line	Circle	Crossing	Penalty	**Precision**
Negative	232	7	36	19	9	3	33	73	**0.563**
Ball	8	246	44	20	9	1	5	3	**0.735**
Robot	20	25	144	20	8	1	5	2	**0.638**
Goal	13	3	33	221	10	1	2	0	**0.781**
Line	12	7	23	11	188	24	11	4	**0.673**
Circle	4	4	2	1	65	253	5	10	**0.737**
Crossing	11	6	17	9	25	4	231	34	**0.688**
Penalty	2	2	1	0	7	13	8	174	**0.840**
Recall	**0.772**	**0.819**	**0.480**	**0.735**	**0.589**	**0.843**	**0.769**	**0.580**	–
Accuracy: 0.703									

Table 3.2: Overview of precision, recall and accuracy values for all classes

	Patch size 16		Patch size 24		Patch size 32	
Classes	**Precision**	**Recall**	**Precision**	**Recall**	**Precision**	**Recall**
Negative	0.472	0.800	0.509	0.782	0.563	0.772
Ball	0.587	0.764	0.707	0.783	0.735	0.819
Robot	0.518	0.296	0.592	0.458	0.638	0.480
Goal	0.720	0.688	0.761	0.743	0.781	0.735
Line	0.554	0.573	0.633	0.587	0.673	0.589
Circle	0.665	0.725	0.743	0.796	0.737	0.843
Crossing	0.565	0.441	0.639	0.613	0.688	0.769
Penalty	0.682	0.386	0.782	0.502	0.840	0.580
Accuracy	**0.584**		**0.658**		**0.703**	

Table 3.3: Confusion matrix of merged classes with patch size 32x32 px

	Ground Truth		
Prediction	Negative	Line	**Precision**
Negative	578	54	**0.914**
Line	39	563	**0.936**
Recall	**0.937**	**0.912**	–
Accuracy: 0.925			

Table 3.4: Overview of precision, recall and accuracy values for merged classes

	Patch size 16		Patch size 24		Patch size 32	
Classes	**Precision**	**Recall**	**Precision**	**Recall**	**Precision**	**Recall**
Negative	0.834	0.915	0.880	0.923	0.914	0.937
Line	0.906	0.817	0.919	0.874	0.936	0.912
Accuracy	**0.866**		**0.899**		**0.925**	

Chapter 4

Pose Correction Procedure

In this chapter, the Pose Correction Procedure will be presented. It takes a prior robot pose and the line patch detections in order to get a corrected robot pose. The procedure will be explained step by step in section 4.1. As the performance of the algorithm is highly dependent on the calibration of the cameras, a possible online camera calibration procedure will be discussed in section 4.2.

4.1 Pose Correction Procedure Step-by-Step

The Pose Correction Procedure will be explained in five different steps. Furthermore, it is presented based on a specific example, namely when the robot is standing on the outer line in the middle of the field, looking into the field. The results of each step based on this example are shown in figure 4.2.

0. Starting Point

All pictures in figure 4.2 (with the exception of number two) show the image of the robot's upper camera in the situation of the example. What we can do now is project the real-world field lines the robot should see at its currently estimated position into the image. The start and end point of each line is saved in a class *LineTable*. The lines are transformed from the 3D robot frame into the 2D image frame. How such a transformation is done is explained in step two of the procedure. Only field lines which intersect with the image plain are considered for the projection. If the projected field lines fit the real field lines in the image, the robotPose contains the correct position values. In figure 4.2 image 0, this is the case for the black projection. The corresponding pose is therefore our target position. What often happens due to uncertainty and calibration errors is that the robotPose is slightly off the correct position. This is represented by the projection in orange and the corresponding pose is our prior position. The goal of the Pose Correction Procedure is to find correction factors to modify the prior robotPose in such a way that the orange projection fits the black one.

1. Line Patch Detection

In order to achieve this goal, the procedure starts with applying the Random Forest detection on the image as explained in chapter 3 for detecting parts of the real field lines in white. The resulting detections are depicted in figure 4.2 image 1 by the orange patches. In order to get these detections for the presented example, the complete image has been randomly sampled. Doing that during a game would be computationally too expensive. This is why we had to come up with a different

sampling strategy. We decided to apply a different module called *Edge Detector*, which was developed by a member of the NomadZ team. It uses a real-time Hough transform method in order to detect all straight lines in the image (more information can be found in [19]). The resulting lines are used as a prior for sampling the field lines with Random Forest. 32 $pixels$ was chosen as the distance between the sampling points. This corresponds to the patch size we use for line detection. An example of how the result looks like can be seen in figure 4.1.
After patches have been sampled from the image they are given to the Random Forest classifier. For positive detections of a certain class, the probability needs to be higher than a threshold, which is a trade-off between recall and precision. A higher threshold leads to a lower false positive rate, but also to a lower true positive rate. As we have a good sampling strategy and the Pose Correction Procedure includes a step where we can get rid of the false positive detections (see step four), the threshold can be set to the quite low value of 50%.
This step of the procedure is implemented in the module *RandomForestDetectorProvider*. The resulting line patch detections are given to the representation *RandomForestDetector* which provides the information to the LineLocator module, where the rest of the procedure is handled.

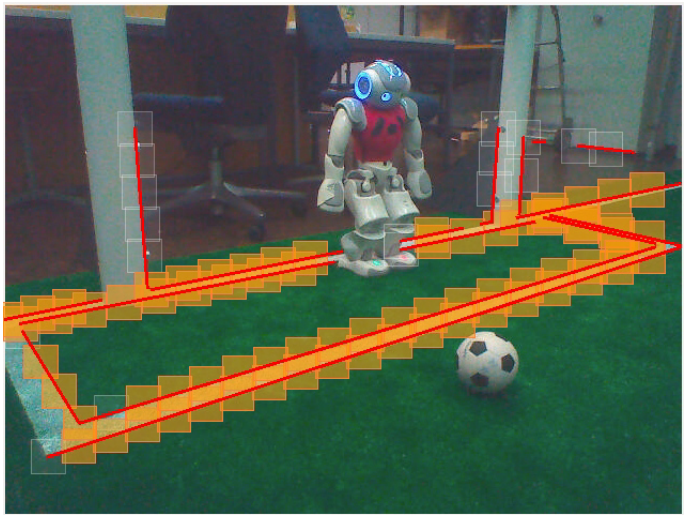

Figure 4.1: Example of sampling the field lines with edge detection as prior. The red lines depict the edge detections. The orange patches are the line detections and the white patches represent samples where no lines were found.

2. Point Conversion

As a next step, we convert the received line patch points (corresponding to the center of the line patches) from the 2D *image* frame into the 3D *robot* frame. The reason for this is explained in the next point of the procedure. The conversion is done in two steps: First, we go from camera to image frame using the intrinsics expressed by matrix K plus a rotation matrix R_{IC}. K contains the intrinsic camera parameters, which are the focal length, the image sensor format and the principal point of the camera. They are defined in the representation *theCameraInfo*. R_{IC} is needed for the passive intrinsic rotation, which aligns the axis of the image frame u_I and v_I and the axis of the camera frame x_C and y_C. This constant matrix can be calculated using:

$$R_{IC} = R_z\left(\frac{\pi}{2}\right) \cdot R_y\left(-\frac{\pi}{2}\right), \tag{4.1}$$

where $R_z(\frac{\pi}{2})$ and $R_y(-\frac{\pi}{2})$ represent the unit rotation matrices around the local z and y axes, respectively, which perform a rotation by $\frac{\pi}{2}$ and $-\frac{\pi}{2}$, respectively.

As a second step, we go from robot to camera frame using the extrinsic parameters. These are defined in the representation *theCameraMatrix* and are dependent on the robot's camera calibration. This homogeneous transformation matrix is denoted by M_{CR}.
As we want to convert the point coordinates from image frame (denoted by $\mathbf{x}_I$) to robot frame (denoted by $\mathbf{x}_R$) and not the other way round, we need to take the inverse of the matrices presented above. This leads to the following transformation:

$$\mathbf{x}_R = M_{CR}^{-1} \cdot R_{IC}^{-1} \cdot K^{-1} \cdot \mathbf{x}_I = M_{RC} \cdot R_{CI} \cdot K^{-1} \cdot \mathbf{x}_I = M_{RI} \cdot \mathbf{x}_I. \tag{4.2}$$

Note that in order to adjust the dimensions of the matrices and calculate their inverses, rows and columns of zeros and ones might have to be added to some matrices. An overview over the transformation matrices can be seen in figure 4.3. Additionally, a scheme where the different coordinate systems are located is depicted in figure 4.2 image 2.

3. Closest Matching

The first part of the third procedure step is to collect the line parameters of the projected field lines corresponding to the prior pose (depicted in orange). This means that for all straight projected lines visible in the image, we collect the start and end points and for the center circle the circle center point and the circle radius, directly in robot frame. This can be done using the same *LineTable* class as for the projection. Then we use these parameters and the converted line patch points in robot frame to perform point-to-line segment (and point-to-circle, respectively) *closest matching* in an euclidean sense. The results are depicted for our example in figure 4.2 image 3 by the yellow transition lines. As an output we collect the closest points on the projected field lines in robot frame corresponding to each of the line patches. We call these points *line samples* in further explanations.
The reason for the transformation from the 2D image to the 3D robot frame explained in the last step can be seen in some closest matching examples in the area around the center circle (marked in figure 4.2 image 3 as violet rectangles). Patch detections in these areas clearly belong to the center circle. If we were to apply the closest matching procedure in image frame instead, the patches could be wrongly matched to the straight lines which are closer in this space. This issue is resolved when applying the matching in 3D space.

4. Pose Hypothesis

To explain this step of the procedure, the underlying transformation method will be presented first. Afterwards it will be shown how this method is used in the Pose Correction Procedure.

Direct Linear Transformation: This step of the procedure needs a transformation method in order to determine the correction factors for the prior robotPose. We decided to use the *Direct Linear Transformation (DLT)* method, which is an algorithm that solves for a set of variables from a set of similarity relations. The relation we want to solve in our problem is:

$$P = T \cdot S \Leftrightarrow \begin{pmatrix} x_{p,1} & \dots & x_{p,n} \\ y_{p,1} & \dots & y_{p,n} \\ 0 & \dots & 0 \\ 1 & \dots & 1 \end{pmatrix} = \begin{pmatrix} cos(\Delta\alpha) & -sin(\Delta\alpha) & 0 & \Delta x \\ sin(\Delta\alpha) & cos(\Delta\alpha) & 0 & \Delta y \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x_{s,1} & \dots & x_{s,n} \\ y_{s,1} & \dots & y_{s,n} \\ 0 & \dots & 0 \\ 1 & \dots & 1 \end{pmatrix}. \tag{4.3}$$

In order to find our robotPose correction factors, we are interested in three parameters: $\Delta\alpha$ to correct the rotation, and Δx and Δy to correct the translation. That means that we want to solve for

the homogeneous transformation matrix T. This matrix consists of a rotation matrix around the z axis in robot space which rotates about the angle $\Delta\alpha$ and a translation vector that translates by Δx and Δy. The similarity relations are given by the matrices P and S, which consists of coordinates of n line patch points $(x_{p,i}, y_{p,i})$ and the corresponding matched line samples $(x_{x,i}, y_{s,i})$ for $i = 1, ..., n$, respectively, stacked together into matrices. In order to be well-defined, the transformation method needs at least three points which do not lie on the same straight line. We can simplify the equation by getting rid of some lines and columns which are not giving any additional information:

$$\begin{pmatrix} x_{p,1} & \dots & x_{p,n} \\ y_{p,1} & \dots & y_{p,n} \end{pmatrix} = \begin{pmatrix} cos(\Delta\alpha) & -sin(\Delta\alpha) & \Delta x \\ sin(\Delta\alpha) & cos(\Delta\alpha) & \Delta y \end{pmatrix} \cdot \begin{pmatrix} x_{s,1} & \dots & x_{s,n} \\ y_{s,1} & \dots & y_{s,n} \\ 1 & \dots & 1 \end{pmatrix}. \tag{4.4}$$

Afterwards we can solve for the matrix T using the Moore-Penrose pseudo-inverse of the matrix S:

$$T = P \cdot S^T (S \cdot S^T)^{-1}. \tag{4.5}$$

The needed parameters Δx and Δy can be directly extracted from the resulting matrix. In order to get $\Delta\alpha$, we apply:

$$\Delta\alpha = tan^{-1}\left(\frac{sin(\Delta\alpha)}{cos(\Delta\alpha)}\right). \tag{4.6}$$

The fourth step of the procedure: In the fourth step of the Pose Correction Procedure, the algorithm performs a *RANSAC (RANdom SAmple Consensus)* loop, which is basically an iterative outlier detection method. The purpose of this step is to sort out the false positive line patch detections the Random Forest can generate. The loop consists of the following repeating steps:

1. **Random point selection:** The loop starts by selecting three points randomly from all detected line patches. It has to be made sure that the selected points do not lie on the same line in order to guarantee that the next step of the loop works.

2. **DLT:** As a next step we use the three randomly selected point pairs (line patches and samples) in order to apply the Direct Linear Transformation method as presented above. Then we use the resulting pose correction factors $\Delta\alpha$, Δx and Δy to correct our prior position temporarily.

3. **Inlier patches:** Afterwards we find all patches whose euclidean shortest distance to the projected field lines corresponding to this temporarily corrected pose is below a certain threshold. This threshold has been empirically determined to be $150\ mm$ (in 3D space). The resulting patches are collected and are called *inlier patches*.

4. **Find best solution:** Finally, we compare the number of inlier patches with the last solutions of the loop and memorize the corrected pose and the coordinates of the inlier patches of the solution corresponding to the highest number of inlier patches.

The runtime of one iteration of this loop has been found to be around $0.2\ ms$. This is why we decided to take 50 RANSAC iterations per frame update.
As a result of this step, we get a hypothesis what our correct robotPose could approximately be and we get rid of the false positive line patch detections of the Random Forest. These results are depicted in figure 4.2 image 4 by the red projected field lines representing our hypothesis pose and the green and red line patches showing the inlier and outlier patches, respectively.

5. Pose Refinement

In order to get the final corrected robot pose, we perform a further refinement step. We do so by using all the inlier patches received at the former procedure step and the corresponding matched line samples to calculate the final pose correction factors again with the DLT method. This process solves a local optimization problem that results in the least squares solution. The resulting corrected robotPose (depicted in 4.2 image 5 in blue) should be very close to the optimal solution.

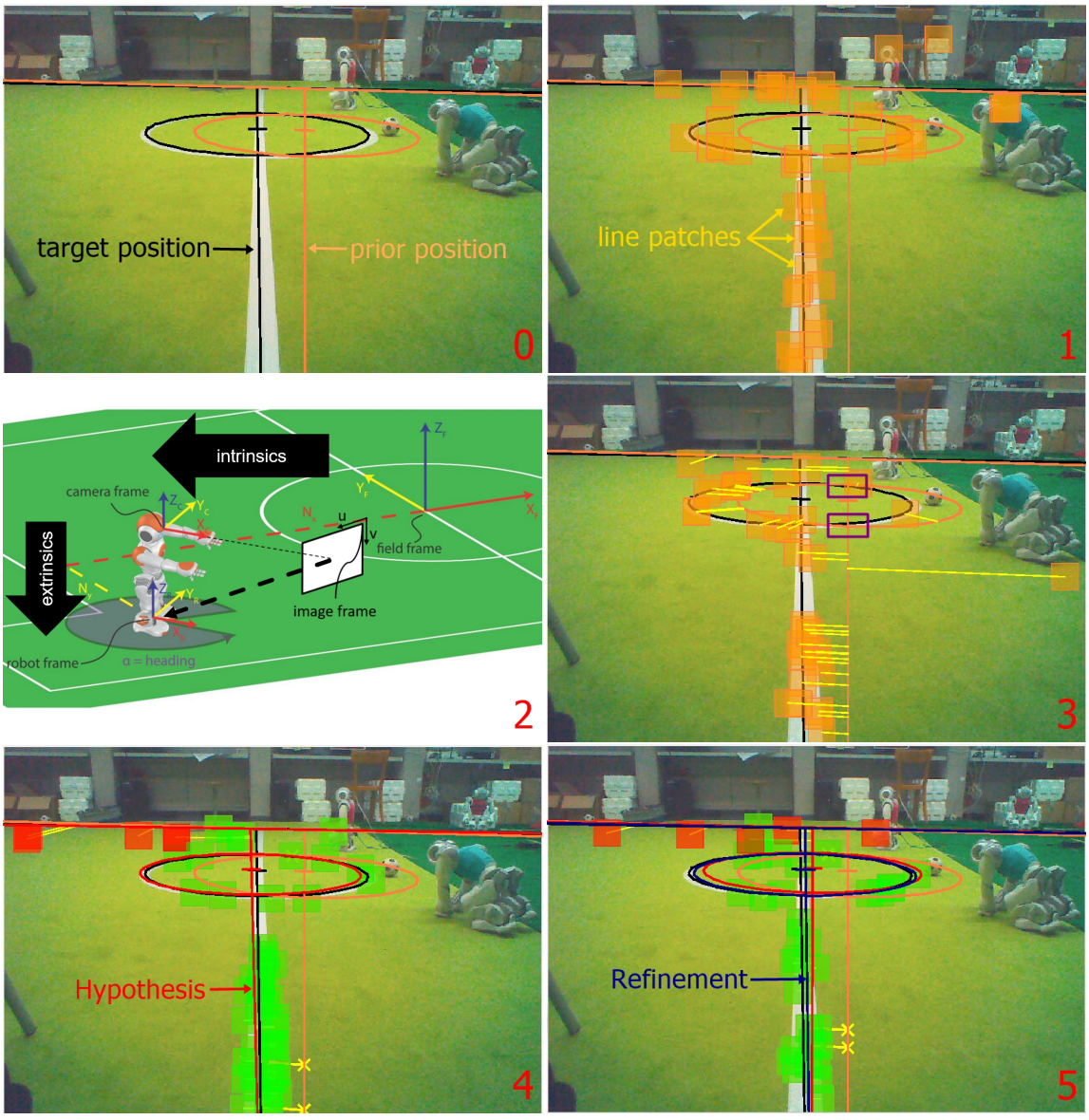

Figure 4.2: Example images of Pose Correction Procedure: (0) Starting point; (1) Line patch detection; (2) Point Conversion; (3) Closest Matching; (4) Pose Hypothesis; (5) Pose Refinement

4.1.1 General Comments

The Pose Correction Procedure can of course only be applied if there are field lines visible for the robot. An important point to mention is that in order to correct all three robotPose coordinates, the robot needs to see at least two different field lines which are not collinear. Otherwise the transformation of the DLT method will not be well-defined as there is too little information to modify the three degrees of freedom. In the case that only one field line is detectable, we decided to only correct the rotation coordinate of the robotPose, and not the two translations. This is due to the fact that the robot's rotation is more important for tasks like aligning behind the ball than its translation on the field. In some special situations on the soccer field, it can nevertheless happen that the transformation is not well-defined and therefore the resulting correction factors don't make sense at all. To make sure that this does not confuse the algorithm, a sanity check has been added which checks if the pose correction factors are below certain thresholds. These thresholds have been empirically found to be $0.2\ rad$ for rotation and $200\ mm$ (in 3D space) for translation. If one of the determined correction factors is above its threshold, then the correction is not applied. This does not restrict the algorithm much as we are only looking for small corrections in order to track the robotPose.

4.2 Online Camera Calibration Procedure

Another important point to mention is that the performance of the Pose Correction Procedure is highly dependent on the calibration of the robot's cameras. This is due to the transformation between the image and the robot frame presented in step two of the procedure. This transformation uses the camera's extrinsic parameters, which are directly dependent on the camera calibration. During a match it is possible that the robot could fall down, which can disturb its calibration. Because of this dependency, we have started working on an online camera calibration procedure using the projective transformation method *Homography*.
According to [20], a general homography is a non-singular, line preserving, projective mapping $h : P^n \rightarrow P^n$ which is represented by a square $(n+1)$-dimensional matrix with $(n+1)^2 - 1$ degrees of freedom. For our purpose, we need mapping between planes, which is why the homography H is a (3×3) matrix:

$$\lambda \mathbf{x}' \propto H \cdot \mathbf{x} \Leftrightarrow \lambda \begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}. \tag{4.7}$$

This means that the homography H maps each point in P^2 represented by a vector $\mathbf{x}$ to another point $\mathbf{x}'$ in P^2 with a certain scale λ. In order to solve for the matrix H, we need at least four 2D to 2D point correspondences as the matrix has 8 degrees of freedom (it has 9 parameters, but the scale of the matrix is arbitrarily).
An overview of the different frames together with the needed transformation matrices to get from one frame to the other are given in the flow chart in figure 4.3. As explained in section 4.1 under point two, the matrix M_{CR} which is the transformation from robot to camera frame is dependent on the camera calibration, as it consists of the camera's extrinsic parameters. Therefore, the goal of the online camera calibration procedure is to find a correction matrix when going from camera to robot frame ΔM_{RC} in order to correct M_{CR} in case it has been disturbed:

$$M_{RC}^{corr} = \Delta M_{RC} \cdot M_{RC}^{dist}. \tag{4.8}$$

In order to achieve this, we can use the Homography method to calculate a transformation matrix M_{IF}^{homo} which directly transforms points from field to image frame.

When performing the procedure correctly, this matrix is independent of the camera calibration errors. In order to derive this matrix, we need to have the line sample points matched to the Random Forest line patch detections in field space. We can do that by converting the points collected in robot frame $\mathbf{x}_F^S$ as described in section 4.1 at point two into field frame using:

$$\mathbf{x}_F^S = M_{FR} \cdot \mathbf{x}_R^S, \tag{4.9}$$

where the homogeneous transformation matrix M_{FR} can be derived from the robotPose coordinates. In order for the calibration method to work, we have to make sure that the field lines projected in the image corresponding to these line sample points fit the real field lines, so that we get the right correction matrix.
Afterwards we use these line sample points in field frame and the corresponding line patch points in image frame to derive the homography matrix H. It is calculated using the *OpenCV* function *findHomography*, which is described in [21]. In order to get the corresponding transformation matrix M_{IF}^{homo}, the received matrix H needs to be normalized and decomposed. This can be done with the OpenCV function $decomposeHomographyMat$, which can also be found in [21]. This function is unfortunately only supported by the OpenCV version 3.0. As the NomadZ framework works with OpenCV 2.4, the function had to be programmed. A detailed description of the implementation can be found in chapter 4 of [22].
As an output of this analytical method for homography decomposition, we get the following four possible solutions for our transformation: $\{R_a, t_a, n_a\}$, $\{R_a, -t_a, -n_a\}$, $\{R_b, t_b, n_b\}$ and $\{R_b, -t_b, -n_b\}$, where R_i represents the rotation matrix, t_i the translation vector, n_i the normal vector of the transformed plane and $i \in \{a, b\}$ stands for two different solutions. In order to find the correct one among these possible solutions, we apply the following two sanity checks:

- $n_i(3) < 0$: We check if the z component of the vector n_i is smaller than zero, which would mean that the robot is standing upside down, rotated around its x axis (which is pointing forward between its feet) by $180°$.
- $feetHeight < -cameraHeight$: If this is true it would mean that the robot is standing at a point below the floor. $feetHeight$ is the z component of the vector $offset_{RF} = -R_{RF}^T \cdot t_{RF}$, where R_{RF}^T and t_{RF} are the rotation and translation from field to robot frame, respectively, extracted from $M_{RF} = M_{IR}^{-1} \cdot M_{IF}^{test}$ and M_{IF}^{test} consists of the possible solution $\{R_i, t_i, n_i\}$ that is tested. Similarly, $cameraHeigth$ can be calculated from the z component of the vector $offset_{CR} = -R_{CR}^T \cdot t_{CR}$.

If one or both of these criteria is true for one of the four options, the corresponding solution is not the correct one. We select the remaining option as our solution to construct the homogeneous transformation matrix we are looking for:

$$M_{IF}^{homo} = \begin{pmatrix} R_i & t_i \\ 0_{1\times 3} & 1 \end{pmatrix}. \tag{4.10}$$

As we want to have a correction from camera to robot frame in order to fix the calibration issues, we apply the following transformation using the transformation matrices as depicted in figure 4.3:

$$M_{RC}^{homo} = M_{RF} \cdot \left(M_{IF}^{homo}\right)^{-1} \cdot M_{IC}. \tag{4.11}$$

Then the correction matrix we are looking for can be calculated by:

$$\Delta M_{RC} = M_{RC}^{homo} \cdot (M_{RC})^{-1}, \tag{4.12}$$

where $(M_{RC})^{-1}$ is the inverse of the matrix containing the camera's extrinsic parameters.

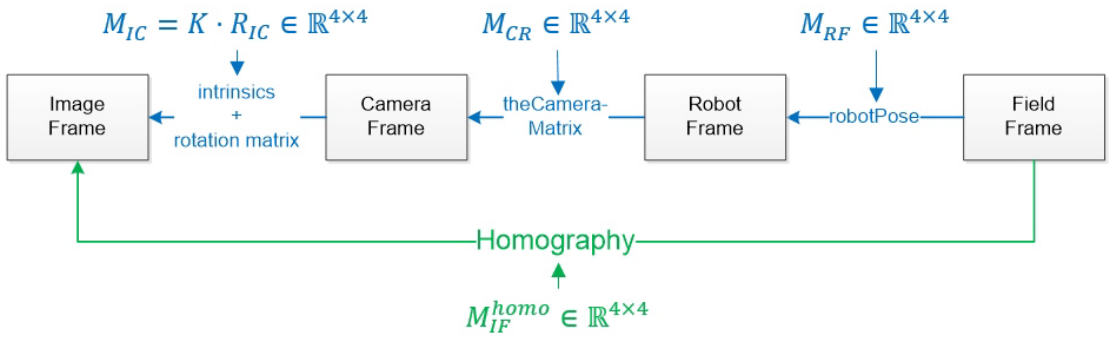

Figure 4.3: Flow chart of the different frames with transformation matrices

Testing this procedure has shown very promising results, but some further thoughts about the implementation have to be made before this can be used in combination with the Pose Correction Procedure. The runtime has been found to be about $5\ ms$ per frame, which is why the correction matrix cannot be calculated at each frame update. This is why one has to think about a criteria for when its best to apply the procedure and how to make sure that it delivers the correct results. So far, the Pose Correction Procedure has been used without this additional online camera calibration.

Chapter 5

Particle Verification Scheme

In section 5.2 of this chapter, a novel verification scheme will be presented, which uses the corrected robot pose received from the Pose Correction Procedure in order to keep track of the robots position while it is walking. The pose tracking is performed using the already existing particle filter of the SelfLocator module. Some more information about its design are given in section 5.1. The verification scheme is used in order to decide whether the position each particle holds is a good estimation for the robotPose. In the final section 5.3 of this chapter, one possible approach of how to find a prior robot pose in case the robot is lost is shortly introduced.

5.1 Design of SelfLocator Module

In this section, the basic implementation and functionality of the SelfLocator module in the B-Human Code Release 2013 will be presented. Further and more detailed information can be found in [15] and [23].
The SelfLocator module, which provides the robotPose representation is designed as a particle filter with an Unscented Kalman Filter (UKF) running on top of each particle. The UKF performs the state estimation while the particle filter executes the hypothesis management and sensor resetting. Due to the limited computational power of the NAO robots, the module only works with 16 particles. Each particle holds a possible robotPose estimation and a variance that indicates the uncertainty of the estimation.
A Kalman filter generally consists of two main parts: The motion update, driven by the odometry of the robot plus some random noise to model uncertainties, and the sensor update, which uses features detected by the cameras to verify the possible positions. These features are the detected field lines and goal posts, which the SelfLocator compares to the respective transformed field model corresponding to the position of a particle. If the detection and the model fit, the particle's variance shrinks. Therefore the variance gives the information of how probable a possible robotPose is. The robotPose is calculated by an averaging procedure which is dependent on these variance values of the particles. An example of how the particle filter in general looks like in SimRobot's WorldState can be seen in figure 5.1, where the particles are depicted by the yellow boxes and the selected robotPose by the blue box.
The *Unscented* Kalman Filter is an improvement of the *Extended* Kalman Filter used for recursive state estimation of nonlinear systems. The main difference between these two filters is that the UKF uses a deterministic sampling strategy to select a minimal set of sample points around the mean. These points are then propagated through the nonlinear process functions. For certain systems, this results in more accurate estimates of mean and variance. More information about the UKF can be found in [24].

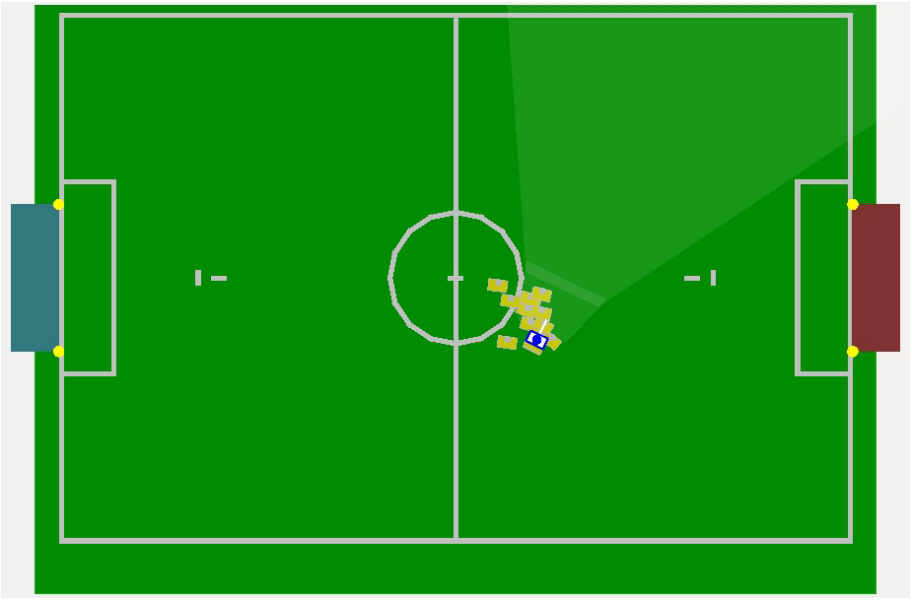

Figure 5.1: Example of particle filter in WorldState

The main problem with this former implementation is that the performance of the line and the goal detection is noisy and unstable, which is why a line only gets detected inconsistently. This often leads to an insufficient sensor update. Therefore, the variance of all particles keeps rising and the particle's position estimates spread out due to the random noise in the motion update step. Another problem is that due to small calibration errors of the robots odometry, the particles can have a rotation bias which is why they can drift to one side, away from the correct robotPose.
Based on these findings we decided to keep the well-working motion update step and just replace the sensor update step by the Particle Verification Scheme presented in the next chapter.

5.2 Particle Verification and Correction

We want to have a verification strategy that indicates whether a particle is a good estimate of the robot position. We use the so-called *inlier ratio* of each particle as an indicator of whether it is close to the correct robotPose. The verification process consists of several steps:

1. **Inlier ratio:** As a first step, we calculate the inlier ratio for each particle. This is done in a similar way as presented in section 4.1 at point four for finding the inlier patches: We project the field lines corresponding to the pose a particle is holding into robot frame and count all line patches (also in robot frame) which are within a certain distance from this projection. The resulting number of inlier patches is divided by the total number of line detections in order to get the inlier ratio.

2. **Best particle:** In order to find the best pose among all particles, we determine the particle with the highest inlier ratio and set this *best particle* as our new robotPose.

3. **Bad particles:** Due to the drifting problem explained in the previous section, particles can drift away from the correct robotPose while the robot is walking. That is why we also need to keep track of particles that have gone "bad". These particles are the ones whose inlier ratio are below a certain threshold, which have been empirically found to be 20%.

4. **Corrected particle:** To correct the particle drift, we apply the Pose Correction Procedure presented in chapter 4 on the best particle in order to get a corrected robotPose. This generates a *corrected particle*.

5. **Particle replacement:** To get rid of the bad particles, we replace them with the corrected particle.

5.2.1 Comments and Restrictions

To complete this section, some general remarks regarding the presented verification scheme must be noted:

- **Enough line patch detections:** The complete verification scheme can only bring reasonable results when there are enough line patch detections. This is why the algorithm has been restricted to the case when the robot detects at least ten line patches.

- **Delay in replacement:** Through testing we found that it is better to not immediately replace the bad particles, instead only after they have been found to be bad at least three consecutive frame updates. This number has been found empirically. The same is valid for the selection of a new robotPose. This prevents the robotPose from getting too jumpy, which could result in unstable walking when the robot is approaching a target such as the ball.

- **Runtime:** It would give us the most accurate estimates if we could apply the Pose Correction Procedure to all particles at each frame update. But as the runtime is an important criteria, we decided to only apply the procedure on the best particle. Additionally, the procedure is only performed when bad particles have been detected and not at each frame update. This decreases the performance of the algorithm, but improves the runtime by a significant amount.

- **Systematic lost criteria:** An additional advantage of the presented Particle Verification Scheme is that it provides a systematic criteria for when the robot is lost, which is the case when all particles have repeatedly been found to be bad for about 30 seconds (empirically determined). In this case, the robot goes into a *lost state* where a different approach is applied in order to regain a prior pose. This is explained in more detail in section 5.3.

5.3 Prior Pose Estimation

During a match, it may happen that a robot loses track of its position. This is the case in the following situations:

- **Falling:** The robot can fall down due to collision with another robot, unstable walking or kicking. The robot has the tendency to turn a little when getting up, which is why the robotPose rotation might no longer be accurate.

- **Penalization:** The robot is penalized by the referee for pushing an opponent player and is manually replaced at the field border.

- **Lack of features:** The robot is at a position where it sees to few features for localization and therefore loses track of its position due to uncertainties.

Figure 5.2: Examples for external features at a RoboCup tournament which can be used to find a prior robot pose; Source: [16]

For these situations, another method has been developed in the former student's projects [16], [25] and [26] in order to find a prior position. It takes external features around the field as landmarks to derive the robot's position. Examples for such features can be seen in figure 5.2. To enable this, the method consists of two different phases:

1. **Offline Phase:** Before a match, a 3D model of the environment based on *Structure from Motion (SfM)* from images recorded from the corresponding field is constructed.

2. **Online Phase:** The model is loaded onto the NAO and used in conjunction with the live image from the robot's upper camera to estimate the robot's position using the algorithm *PnP (Perspective-n-Point)* pose estimation.

As a result, an estimate of the robot's position (α, x, y) together with a certainty factor is received. This approach has the great benefit that it solves the symmetry problem we have on the soccer field. The disadvantage it holds is that it is computationally very demanding for the NAO hardware, which is why we can only use it occasionally. As the Particle Verification Scheme provides a systematic criteria when the robot is lost (as explained in section 5.2.1), we can combine these two methods very well. In case the verification scheme provides the information that the robot is lost, then the PnP pose estimation procedure is applied in order to find the prior again. To increase the chance to retrieve track of the robotPose, the particles are randomly spread a bit around the received prior pose based on the corresponding certainty factor, meaning that the lower the certainty factor (which is the case when the pose estimation procedure is less certain about how accurate the estimated position is), the more the particles are spread. This helps the verification scheme to regain track of the robot's position.

Chapter 6

Results

The Particle Verification Scheme with Pose Correction Procedure (but without the online camera calibration procedure) has been tested in several situations such as when the robot is walking over the field. The results presented in this thesis are based on a specific representative example of when the robot is walking across the field. The path the robot is taking is depicted in figure 6.1.
In order to compare the estimate to the correct robotPose, the robot's ground truth position has been measured by an external camera system called *Vicon*. In our case, this system consists of four infrared cameras which track retroreflective surfaces. In order to track the NAO robot, reflective markers are placed on its body. More detailed information can be found in [27] and [28].
The resulting values are plotted in figure 6.2. We compare the results of the former implementation (left) to the ones of the new implementation (right). As the estimation is not deterministic, the results have been averaged over ten runs.
When looking at the estimate of the robotPose plotted in red of the new implementation on the right side, we can conclude the following: The rotation (plotted on top) is not perfectly accurate but follows the progress of the ground truth. For the translation in x direction (plotted in the middle), a small almost constant offset between the ground truth and the estimate is visible. This can be explained by an offset in the camera calibration, which could be fixed when adding the online camera calibration procedure presented in section 4.2. The translation in y direction (plotted below) fits the correct position during the complete run almost perfectly. This shows that the motion update of the Kalman filter cooperates well with the Particle Verification Scheme.
In addition to the robotPose estimate, the mean position of all particles is plotted in green. For all three degrees of freedom it is almost identical to the pose estimation. Another result is visible when looking at the plots on the right side of figure 6.3, which show the variance of the particles during the experiment. The variance of the rotation parameter (plotted on top) stays below 0.12 rad and the variance of the translation parameters (plotted below) do not overstep the boundary of 100 mm in this particular example. This leads to the conclusion that the particles stay together in one cluster around the estimated robotPose and do not spread out very much.
When we view the results of the same experiment but with the code of the former implementation (shown in the plots on the left side), we can see that the initial estimate is quite accurate, but the longer the robot is walking, the more the particles spread out. This is due to the fact that the line detection of this implementation is quite unstable, which is why the sensor update which happened only occasionally is insufficient. This is why the particle filter can only rely on the motion update step. Due to the added random noise to cover the uncertainties, the particles spread out to both sides of the pose estimate, which leads to high variances and an inaccurate pose estimate.

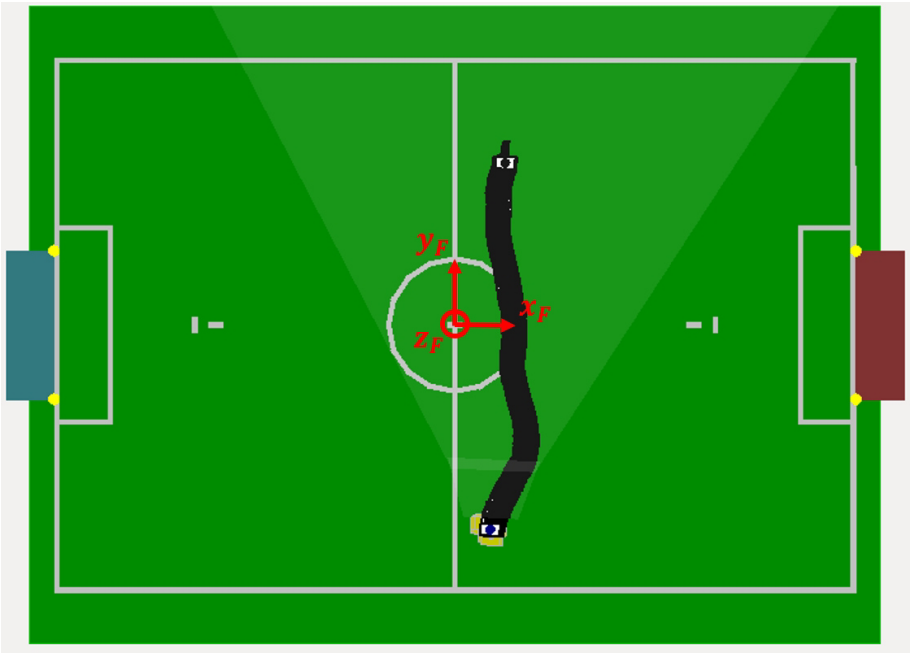

Figure 6.1: Path the robot is walking in the test run depicted in the WorldState of SimRobot

An example of how the situation looks like in the WorldState of the simulation software SimRobot at the end of the run for both implementations is depicted in figure 6.4. Additionally, the mean error between ground truth and estimate for all three robotPose coordinates and for both implementations have been calculated using the formula:

$$ME = \frac{1}{N} \sum_{i=1}^{N} |\hat{v}_i - v_i|, \tag{6.1}$$

where $i = 1, ..., N$ are the enumeration of the frames of the test run, $\hat{v}_i$ are the pose estimates and v_i are the ground truth values at frame i. The resulting values are presented in table 6.1. They reflect the same results as presented above.

Table 6.1: Comparison of mean error of former and new implementation

	Former	New
rotation [rad]	0.40	0.09
x translation [mm]	424.62	69.46
y translation [mm]	139.63	39.43

The average runtime of the newly implemented procedure on the robot has been found to be around 5 ms per frame. The average overall runtime of all modules running on the robot during a soccer game should not exceed the value of 17 ms as this would slow down all the processes. Therefore, the result of 5 ms can be considered efficient.
The derived procedures have also been applied and tested at the RoboCup World Championship 2017 in Nagoya Japan. The results of the robotPose estimation have been found to be quite accurate and robust and the implementation on the NAO robot has been proven to work reliably.

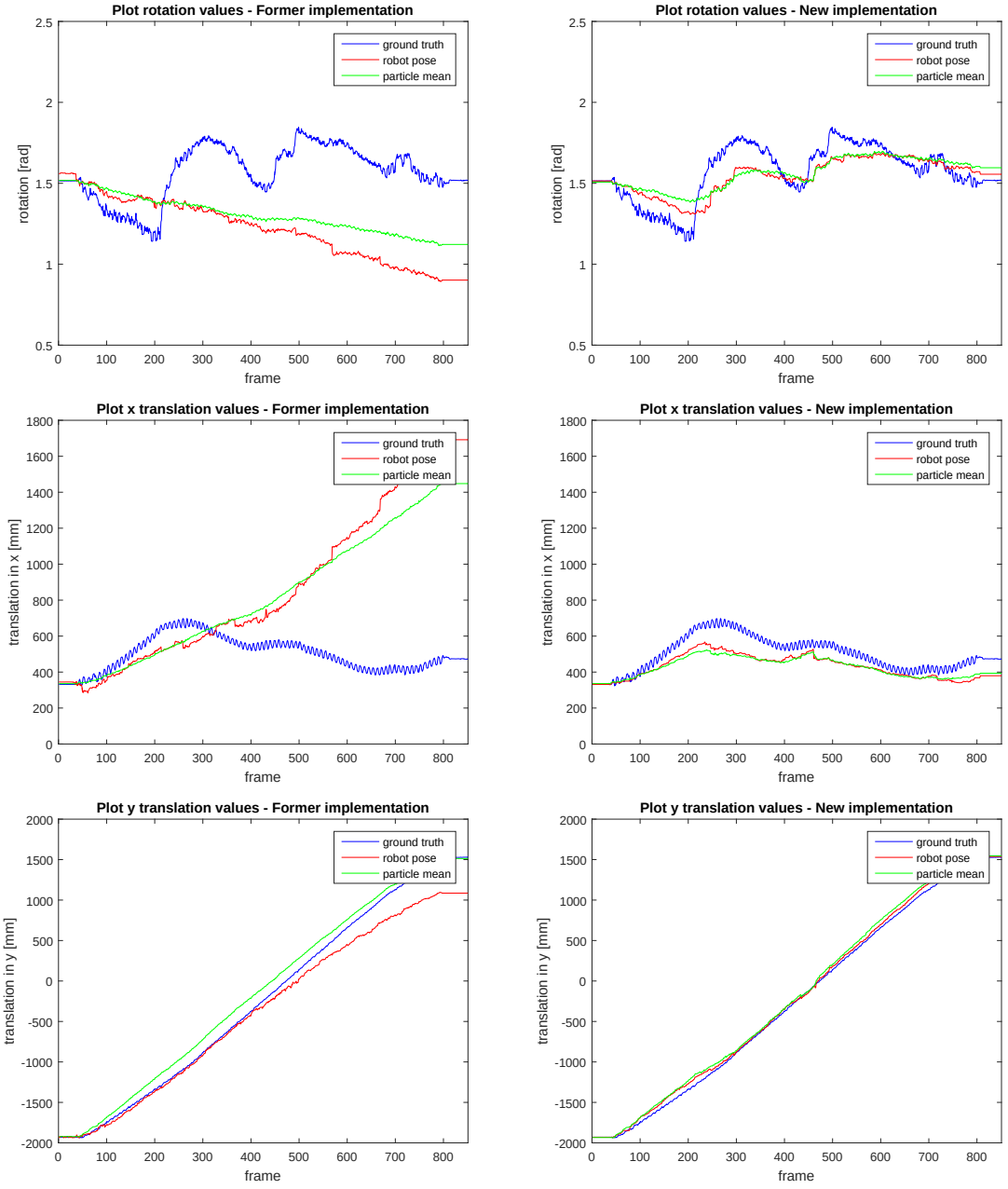

Figure 6.2: Plots of resulting robotPose estimate (rotation and translation in x and y direction) during the test run; Left: Former implementation; Right: New implementation; Blue: Ground truth robotPose; Red: Estimated robotPose; Green: Mean of all particles

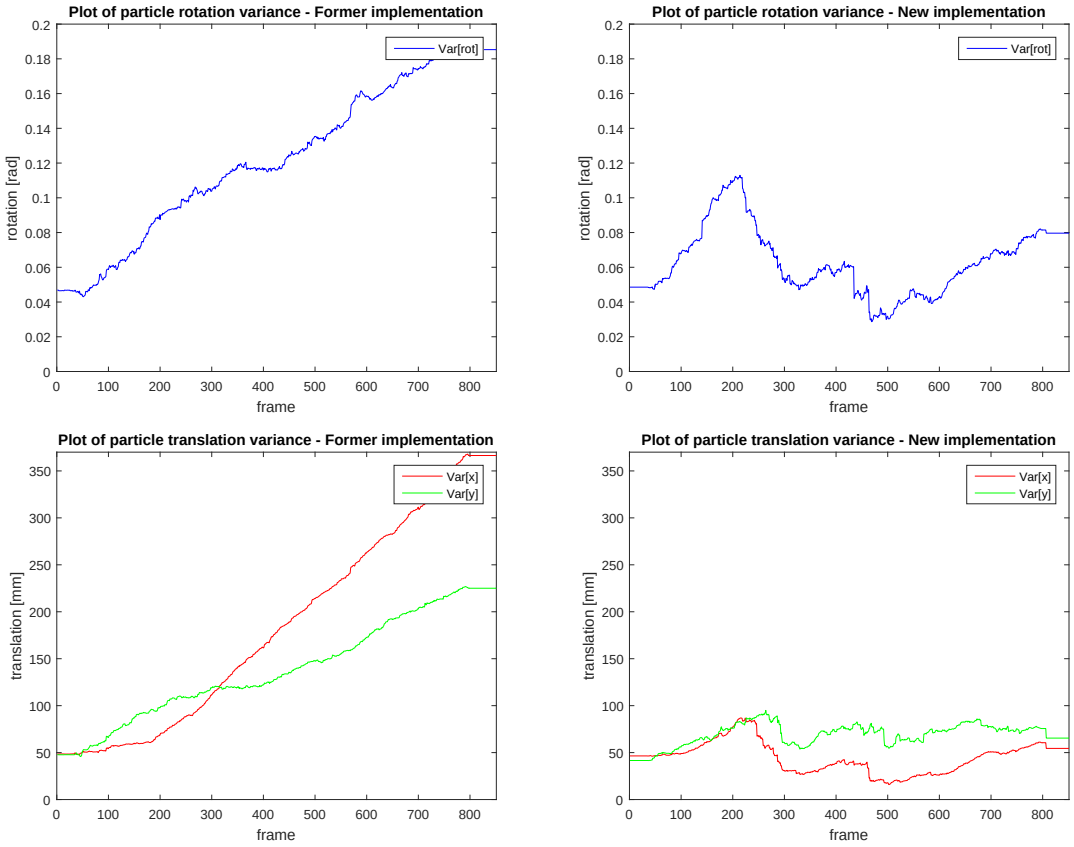

Figure 6.3: Plots of resulting particle variance during the test run; Left: Former implementation; Right: New implementation; Blue: Variance of rotation; Red: Variance of translation in x direction; Green: Variance of translation in y direction

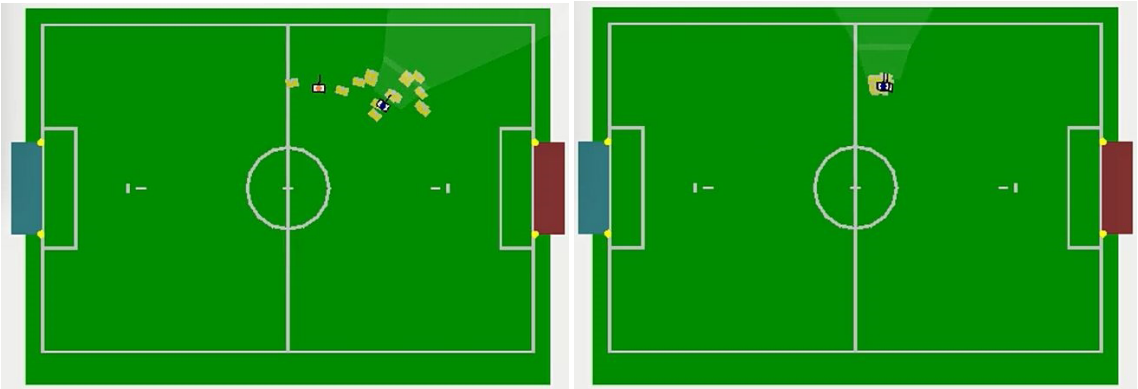

Figure 6.4: Example images of particle spread and robot pose at the end of the test run; Left: Former implementation; Right: New implementation; Orange rectangle: Ground truth robot pose; Blue rectangle: Robot pose estimate; Yellow rectangles: Particles

Chapter 7

Conclusion and Future Work

As a conclusion, we can say in general that the self-localization of the NAO robots on the soccer field has improved in comparison to the former implementation as the resulting robotPose is more accurate. Furthermore we have better position certainty, as the particles stay together in one cluster around the robotPose and do not spread out so much anymore. Additionally, no sudden particle jumping can occur, which had been an issue when the detected field line was wrongly matched to the field model. Due to the fact that the presented Particle Verification Scheme provides a systematic criteria when the robot has lost track of its position, the method can be combined very well with other methods that estimate a prior pose. The restrictions that have been made in order to decrease the runtime to about 5 ms do restrict the performance of the procedure, but still allows satisfactory results. The method is still limited in the case where the robot doesn't detect any or too few field lines.

The presented methods could be improved by taking the images of the lower camera into account. At the moment, the line patch detection and the corresponding field line projection matching works only with the upper camera. Additionally using the lower camera could be beneficial in some situations on the soccer field, but as the corresponding images are processed alternately and not in the same frame update, the combination of both camera feeds could pose a tricky issue.
Due to the high dependency of the performance of the presented methods on the calibration of the cameras, we have started working on an online camera calibration procedure as presented in section 4.2. This method has shown promising results, but some more careful testing on the robot itself has to be done in order to make sure that it works in each situation on the field. Furthermore, a criteria should be set for when the method should be applied during a game as it is computationally quite demanding and could therefore not be performed at each frame update. An example for its application could be for each time the robot falls, as this invalidates the calibration the most. This approach would be very helpful in general as it could also increase the performance of other modules which depend on a proper camera calibration.
Another opportunity for possible future work could be to use the well-working line patch detection to derive an automatic *offline* camera calibration procedure. In the current method, the field lines in the image must be manually selected, which takes a lot of time before a game at a tournament. This could be automated using Random Forest to detect the field lines.
Additionally, the goal detection with Random Forest could be further optimized as this classification has shown promising results as well. Knowing where the goal is could help the robot with localization and kicking the ball in the right direction.
In another student project, the *Multiple Forest* approach has been implemented on the coach robot. This method uses trees with different patch sizes. When an object is in the upper part of the robot's camera image, it is sampled with a smaller patch size. This improves the detection of ob-

jects which are further away. The approach should be tried on the field players in order to improve the Random Forest detection in general.
Furthermore, more methods should be developed in order to determine a prior pose. One possibility has been shown in section 5.3 using external features from the surroundings of the soccer field. Unfortunately, generating the offline map with the external features takes a lot of time at tournaments. Another promising approach which has been implemented is using a combination of the Edge Detector and Random Forest for line detection. The Edge Detector provides all lines visible in a camera image. The Random Forest is used in order to verify if such a detected line is a field line or not. With this strategy, the penalty box and the center circle can successfully be detected and the pose can be estimated using these features. Additionally, larger line features like corners and the T-structure where the middle line crosses the border lines could also be used for detection. This would bring additional information in order to determine a prior pose.
One example of how the presented methods could be used in order to regain track of the robotPose after the robot has fallen is through ”educated guesses” around the last position before the fall by slightly changing the rotation parameter of the pose. This idea comes from the fact that the robot mainly loses track of its orientation and not of its translation after getting up again. These guesses could be verified and corrected using the presented verification scheme.

Bibliography

[1] RoboCup, "Standard Platform League," http://spl.robocup.org, Mai 2017.

[2] Automatic Control Laboratory (IfA) - ETH Zürich, "Welcome to IFA," http://control.ee.ethz.ch/, Mai 2017.

[3] Computer Vision Laboratory (CVL) - ETH Zürich, "About the Lab," https://www.vision.ee.ethz.ch/en/, Mai 2017.

[4] Department of Information Technology and Electrical Engineering - ETH Zürich, "D-ITET," https://www.ee.ethz.ch/, Mai 2017.

[5] Automatic Control Laboratory & Computer Vision Laboratory, "RoboCup ETH Zürich - Nomadz," https://robocup.ethz.ch/, Mai 2017.

[6] Softbank Robotics, "Who is Nao?" https://www.ald.softbankrobotics.com/en/cool-robots/nao, Mai 2017.

[7] Aldebaran Robotics, "Aldebaran Documentation - Motherboard," http://doc.aldebaran.com/2-1/family/robots/motherboard_robot.html, Mai 2017.

[8] ——, "Aldebaran Documentation - NAO - Actuator and Sensor list," http://doc.aldebaran.com/2-1/family/nao_dcm/actuator_sensor_names.html, Mai 2017.

[9] ——, "NAO Software 1.14.5 documentation - NAOqi Framework," http://doc.aldebaran.com/1-14/dev/naoqi/index.html, Mai 2017.

[10] ——, "NAO Software 1.14.5 documentation - C++ SDK," http://doc.aldebaran.com/1-14/dev/cpp/index.html, Mai 2017.

[11] ——, "Aldebaran Documentation - NAO Documentation," http://doc.aldebaran.com/2-1/home_nao.html, Mai 2017.

[12] Wikipedia, "Nao (robot)," https://en.wikipedia.org/wiki/Nao_(robot), Mai 2017.

[13] Aldebaran Robotics, "Aldebaran Documentation - NAO - Versions and Body Type," http://doc.aldebaran.com/2-1/family/body_type.html, Mai 2017.

[14] University of Bremen - B-Human, "Welcome," https://www.b-human.de/, Mai 2017.

[15] ——, "Team Report and Code Release 2013," https://www.b-human.de/downloads/publications/2013/CodeRelease2013.pdf, January 2014.

[16] S. Flückiger, "Self-Localization on NAO robots based on external features: PnP pose estimation," Computer Vision Lab - ETH Zürich, Semester Project, Sep. 2016.

[17] F. Amstutz, "Ball Detection with Random Forest Classifier," Computer Vision Lab - ETH Zürich, Semester Project, Feb. 2017.

[18] Juergen Gall, Nima Razavi, and Luc Van Gool, "An Introduction to Random Forests for Multi-class Object Detection," Computer Vision Laboratory - ETH Zurich, Tech. Rep.

[19] John Gordon Morrison, "Implementing a Real-Time Hough Transform on a Mobile Robot," Bowdoin College, An Honors Paper for the Department of Computer Science, 2011.

[20] Christiano Gava and Gabriele Bleser, "2d projective transformations (homographies)," Technische Universität Kaiserslautern, Lecture Slides.

[21] opencv dev team, "Camera Calibration and 3D Reconstruction," http://docs.opencv.org/3.0-beta/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html, November 2014.

[22] Ezio Malis and Manuel Vargas, "Deeper understanding of the homography decomposition for vision-based control," INRIA Sophia Antipolis, Tech. Rep., 2007.

[23] M. Walter, "Revision of the NAO Robot Self-Localization," Computer Vision Lab - ETH Zürich, Semester Thesis, 2016.

[24] E. A. Wan and R. V. D. Merwe, "The unscented Kalman filter for nonlinear estimation," in *Proceedings of the IEEE 2000 Adaptive Systems for Signal Processing, Communications, and Control Symposium (Cat. No.00EX373)*, 2000, pp. 153–158.

[25] S. Maurer, "Self-localization on NAO robots based on external features: Offline-model generation (SfM) and Online feature matching," Computer Vision Lab - ETH Zürich, Semester Project, Sep. 2016.

[26] M. Wulf, "Self-localization on NAO robots based on external features: Implementation," Computer Vision Lab - ETH Zürich, Semester Project, Sep. 2016.

[27] VICON - Intelligence in Motion, "Vicon Object Tracking," https://www.vicon.com/motion-capture/engineering, August 2017.

[28] N. Evirgen, "Ground-trouth Robotic Tracking and Feedback System for RoboCup," Department of Information Technology and Electrical Engineering - ETH Zürich, Semester Project, Feb. 2017.