Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Conditional Random Fields for joint ball and opponent detection, and self-localization

Object-Localization for the RoboCup

Bachelor's Thesis

Hans Hardmeier
Department of Computer Science, ETH Zrich

Advisor: Xavier Boix Gemma Roig
Supervisor: Prof. Dr. Luc van Gool

September 16, 2013

Abstract

In the Robocup, a crucial basis for the use of complex strategies is the understanding of the current situation of the scene. In a previous semester project, Martin Utz presented in [9] an approach to self-localization of robots using Conditional Random Fields (CRFs), which claimed to be accurate and fast. In this thesis, the CRF is expanded to include the detection of the ball and the opponents. We show that the joint modeling of the self-localization and the detection of the ball and the opponents has an important positive impact on the overall estimation. Comparing the new algorithm with the algorithm in [9], we observe that the mean error of the estimated rotation of the robot improved, while recovering from being kidnapped stays under a 0.3 sec. mark. The computational costs of MAP inference in the CRF was reduced to the half and can be executed on the robot with 60 Hz. In a series of experiments we show that the achieved accuracy is satisfactory.

Acknowledgements

This project would not have been possible without the excellent assistance of Xavier Boix and Gemma Roig. I would also like to thank Beat Kueng for his help regarding the Nao Operating System and Martin Utz for letting me base this thesis on his work. Last but not least, I would like to express my appreciation to Prof. Dr. Luc van Gool for giving me the possibility to be part of such a great project like the RoboCup.

Contents

List of Figures

List of Tables

Acronyms

BH B-Human

BHM Ball Heatmap

CRF Conditional Random Field

EKF Extended Kalman Filter

ERPHM EnemyRobotPoseHeatMap

HMM Hidden Markov Model

Chapter 1

Introduction

For a normal soccer player, several elements have to be tracked to be able to construct a consistent global state of the game field. Some of those elements are the player itself, the ball and the opponents. The team from the University of Bremen and German Research Center for Artificial Intelligence (B-Human (BH)) was able to win the RoboCup World Championship four times in a row using several Kalman- and Particle-Filters to calculate the position and odometry of the robot itself, the ball and the opponents. For the model of the ball, BH used twelve multivariate Gaussian probability distributions to predict the movement of the ball [4]. Thus, extended Kalman-filters were calculated in parallel and separately from each other to achieve different results, for which the best one was chosen. For the opponents, BH used a very simple model [4]:

> ...we use an unscented Kalman filter for each detected robot .... Actually it is not a real Kalman filter, since we did not implement a motion model yet. So we are currently assuming that all recognized robots do not move but we add a significant noise in each cycle so that we can react faster to the robots movements.

This filters, however, have two major disadvantages: the Bayes assumption for measurement vectors which states that an assumption have to be made to reduce the complexity of the filter and the fact that distributions are only approximated which leads to calculation errors [7]. Martin Utz replaced those filters for the self-localisation of the Robot using CRF in his semester project at the ETH Zurich [9]. CRFs are widely used in the field of computer vision and speech processing, due their combination of conditional models and global normalization of random fields. They are able to handle arbitrary dependencies between observations, sensor noises and sample-based approximations.

1.1 Focus of this Work

The focus of this work is to improve the CRF model proposed by Martin Utz by expanding it. The new model will be able to improve the user-robot position and track also the ball, the opponents and their movements, which can be used on the NAO robots for the RoboCup competition for advanced strategies. Having knowledge of the absolute position of the ball and of the opponent robots, help the user-robot to repositioning, if one of those elements is being percepted, and if the robot is correctly positioned, the estimation is more accurate. We will focus on the estimation of the models of the ball and of the opponents through the CRF. The challenges of fully understanding an unfamiliar code, expanding the CRF in a mathematically correctly and computationally efficent way, and being accurate enough to be useful, make this project both challenging and worthwhile. The RoboCup team at ETH Zurich will utilise the B-Human framework to incorporate future work. Therefore this framework is also used to develop the implementation of the CRF algorithm. The perfomance tests will be based on comparision with the old CRF.

Chapter 2

Related Work

In the following section, the methods used by the B-Human Team to locate the ball and the oppenents will be presented. Afterwards, an introduction for CRF will be given, followed by a presentation of the work of Martin Utz -Self-Localisation based on Conditional Random Field [9]- , on which this thesis is based. At the end, a brief discussion about the expansion of the CRF will help us understand the motivation for this work.

2.1 Ball- and Opponentlocation in B-Human

BH uses EKF to derive the motion of the ball, because of their ability to operate recursively on streams of noisy input data to produce a statistically optimal estimate of the underlying system state. Kalman filters basically remove the noise produced by different errors of measurements. They keep track of the predicted state of the system and the variance of the prediction. The following material was taken from the master course *Recursive Estimation* taught by Prof. Dr. Raffaello D'Andrea at ETH Zurich [5] and from the Semester thesis of Martin Utz at the ETH in the fall semester 2012 [9].

2.1.1 Notation for EKF

In this chapter we assume that our system can be modelled with non-linear equations in the form of:

$$x(k) = q_k(x(k-1)\,, u(k)\,, v(k)) \tag{2.1a}$$

$$y(k) = h_k(x(k), u(k), w(k)) \tag{2.1b}$$

Where $x(k)$ is the state vector, $u(k)$ known control input, $v(k)$ process noise, $w(k)$ sensor noise and $y(k)$ is the measurement. q_k is the non-linear, time variant system update function and h_k the non-linear time variant output function. k is the discretised time. The aim is now to estimate the state $\hat{x}(k)$ based on the measurement $y(k)$. To do that the variables $x(k)$, $v(k)$ and $w(k)$ are treated as independent random variables with variance $P(k)$, $Q(k)$ and $R(k)$ respectively.

The EKF involves two steps:

1. A priori prediction step

    (a) Prior state update

$$\hat{x}_p(k) = q_k(\hat{x}_m(k-1), u(k), v(k) = 0) \tag{2.2}$$

(b) Linearise dynamics for variance update

$$A(k) = \frac{\partial q_k}{\partial x}\left(\hat{x}_m(k-1), u(k), v(k) = 0\right) \tag{2.3a}$$

$$L(k) = \frac{\partial q_k}{\partial v}\left(\hat{x}_m(k-1), u(k), v(k) = 0\right) \tag{2.3b}$$

(c) Update prior variance

$$P_p(k) = A(k)\, P_m(k-1)\, A^{\top}(k) + L(k)\, Q(k)\, L^{\top}(k) \tag{2.4}$$

2. A posteriori update step

(a) Linearise non-linear measurement equation

$$H(k) = \frac{\partial h_k}{\partial x}\left(\hat{x}_p(k), w(k) = 0\right) \tag{2.5}$$

(b) Measurement update

$$P_m(k) = \left(P_p^{-1}(k) + H^{\top}(k)\, R^{-1}(k)\, H(k)\right)^{-1} \tag{2.6a}$$

$$\hat{x}_m(k) = \hat{x}_p(k) + P_m(k)\, H^{\top}(k)\, R^{-1}(k)\, (y(k) - h_k(\hat{x}_p(k), w(k) = 0)) \tag{2.6b}$$

2.1.2 Balllocation

Now, having explained the unimodal probability distribution calculation by a Kalman Filter, BH uses twelve multivariate Gaussian probability distributions to represent the belief concerning state of the ball. Those distributions are divided in two equally big subsets. Each one of these subsets represent one of the possible states of the ball: $\mathcal{V} = \{\text{Rolling}, \text{Stationary}\}$. Each distribution is handled in isolation from eachother, which means that in each frame twelve Kalman filters are running. Having the result of each one of the filters, only one is used at the end to generate the ball model. The choice criteria are the variance of the distribution and the fitting of the measurement.

2.1.3 Opponentlocation

For the prediction of the opponents, BH uses a special variation of a Kalman filter. Actually it is not a real Kalman filter, as they did not implement a motion model. Since the motion of the opponents is not included, they added a significant amount of noise, to be able to get a result in case the robot changes location. To be able to use this approach, recognized robots should be matched with the perceived robots. For this they calculated the Mahalanobis distance as developed in [8]:

$$d(\mathbf{x}, \mathbf{y}) = \sqrt{\langle (\mathbf{x}-\mathbf{y})^T \Sigma^{-1} (\mathbf{x}-\mathbf{y}) \rangle}. \tag{2.7}$$

The problem here, is that the field is *partially* −not *fully*− observable. The perception layer almost never recognizes every robot in the field at the same time. Having the Mahalanobis distance of a perceived robot, it is difficult to find out whether it correlates with a robot in the model or it is a new robot. To solve this problem BH simply put an euclidean distance as a threshold to accept a detected robot as a previously modelled robot one or not. Robots in the model, which have not been seen in a long time will not be tracked any longer if the approximated probability of an area around the mean is below a certain threshold.

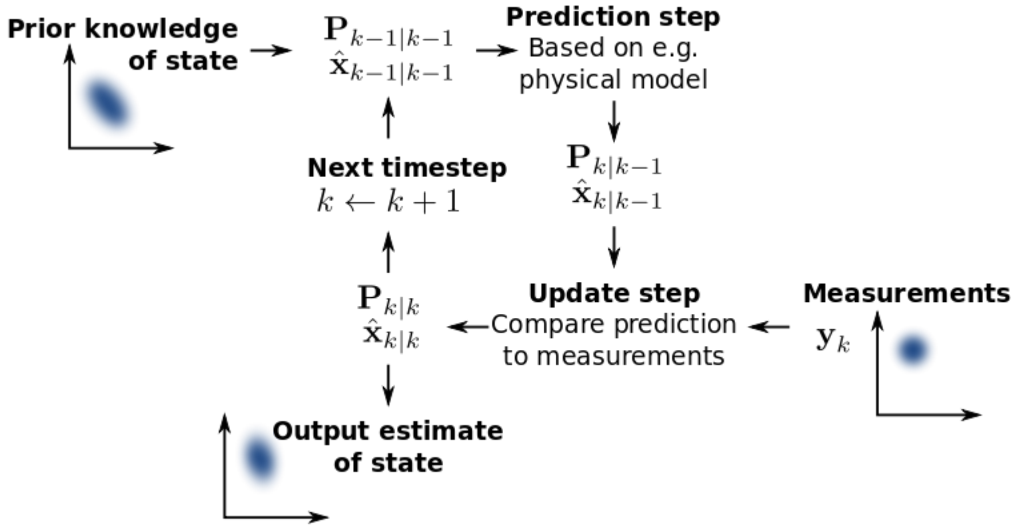


Figure 2.1: Graphical representation of a Kalman filter. $\hat{x}_{k|k-1}$ represents the prediction of the system's state at time k. y_k denotes the measurements, which serve as a parallel input at time step k for the a posteriori update step. Source: Wikimedia

2.1.4 Disadvantages

Therea are several problems with EKF. The first one is that it is not an optimal estimator. The estimated covariance matrix only underestimate the actual matrix. This may lead to inconsistency, if no other stabilization methods are used. A second problem is, that EKF may diverge, if the initial estimate is wrong and the third problem is that EKF is a naive Bayes classifier. It assumes that the presence or absence of a features are unrelated to the presence or absence of other features. The argument that each feature contributes indenpendenly to the probability of the state, is questionable [7].

2.2 Conditional Random Fields

Conditional Random Fields CRF are a class of statistical modelling method used for structured prediction in pattern recognition and machine learning. The main idea behind this model is to encode known relationships between the different elements to derive a consistent prediction of those elements. Lafferty, McCallum and Pereira defined a CRF on observations and random variables as follows [6] :

> Let $G = (V, E)$ be a graph such that $\mathbf{Y} = (\mathbf{Y}_v)_{v \in V}$, so that $\mathbf{Y}$ is indexed by the vertices of G. Then $(\mathbf{X}, \mathbf{Y})$ is a conditional random field in case, when conditioned on $\mathbf{X}$, the random variables $\mathbf{Y}_v$ obey the Markov property with respect to the graph: $p(\mathbf{Y}_v|\mathbf{X}, \mathbf{Y}_\omega, \omega \neq v) = p(\mathbf{Y}_v|\mathbf{X}, \mathbf{Y}_\omega, \omega \sim v)$, where $\omega \sim v$ means that ω and v are neighbours in G

Applying this model to our situation with a 2D game field, we can imagine the CRF as a first-order chain of random Variables, which each one of them describing the global state of the field and a global Variable X, which describes the global restrinctions (i.e. exactly one ball in the field). A graphical representation can be found in Fig. 2.2.

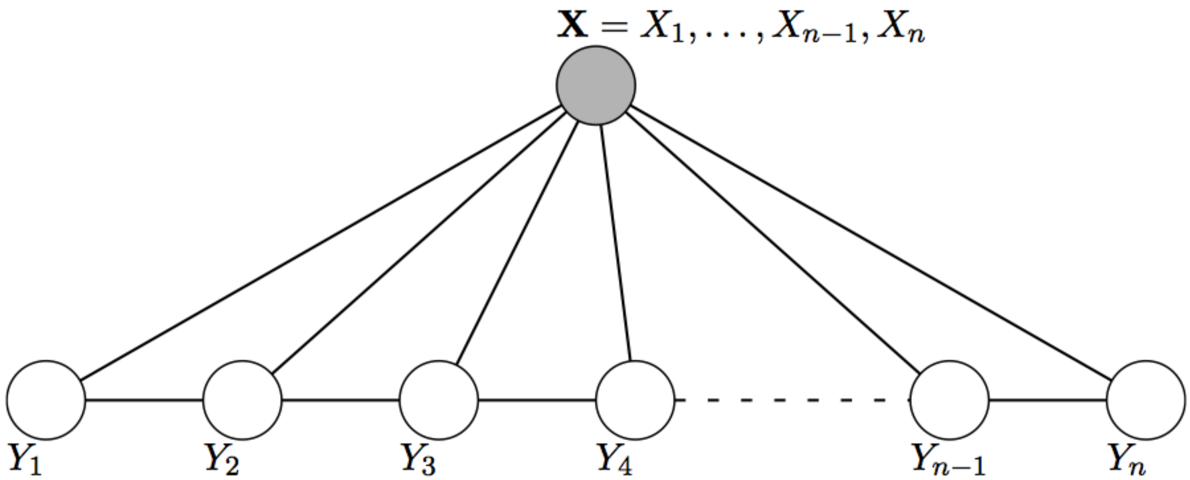


Figure 2.2: Representation of a chain-structured CRF. The variables corresponding to grey nodes are *not* generated by the model

2.2.1 Potential Functions

Having such a chain of variables connected through the Markov property, we may ask, what will the difference be between a Hidden Markov Model (HMM), which is in essence a Markov-chain, and a CRF. The main difference is that Random Fields are a distribution over *potentials* which are just any strictly positive real valued functions defined over a *clique* of variables. In HMM, we have a graph structure over probability distributions instead. We define a *potential function* as:

$$\psi_C(y_{i-1}, y_i, \mathbf{x}) \tag{2.8}$$

A potential function per se does not possess an interpretation, instead it represents a constraint between the random variables. The joint probability distribution of $\mathbf{y}$ can be calculated as the product of the potential functions:

$$p(\mathbf{y}|\mathbf{x}) = \frac{1}{Z(X,Y)} p(\mathbf{y}, \mathbf{x}) \tag{2.9}$$

,where $Z(X,Y) = \sum_y p(\mathbf{y}, \mathbf{x})$

$$p(\mathbf{y}, \mathbf{x}) = \prod_i \psi_C(y_{i-1}, y_i, \mathbf{x}) \tag{2.10}$$

Defining the potential functions to have an exponential form, we set:

$$\psi_C(\mathbf{y}, \mathbf{x}) = \exp\left(-\,\mathrm{E}\left(\mathbf{y}, \mathbf{x}\right)\right) \tag{2.11}$$

Here, we will call $\mathrm{E}()$ an *energy function*. Due to the fact that we are only interested in the maximum of the joint distribution, we do not have to calculate $\frac{1}{Z(X,Y)}$. Transforming the Equation 2.10 with 2.11 by taking the logarithm we get a sum of energy functions. In this case, the maximun of the joint distribution means the minimum of the energy function.

2.3 Self-Localisation using CRF

In the semester thesis of Martin Utz [9], he defined the energy function as a function of the state vector $\mathbf{x} \in \mathbf{X}$. Where $\mathbf{x}$ is a possible instantiation of the random variable $\mathbf{X}$ that means a possible localisation of the robot. Each element of the state vector $x_i^t \in \mathbf{x}$ takes a label from the set $\mathcal{L} = \{l_r, l_\varnothing\}$. Each x_i^t corresponds to a certain time t, where $0 \leq t \leq t_{max} = |\mathcal{T}|$ and a certain position i, which includes position in the 2D field and a discretised rotation.

Having defined the variables, the energy function was defined as follows:

$$\mathrm{E}(\mathbf{x}) = \underbrace{\sum_{i,\,t} \phi_i^t\big(x_i^t, \mathcal{F}_t\big)}_{\text{unary}} + \underbrace{\sum_{(i,j)\in\mathcal{N},\,t} \chi_i^t\Big(x_j^{t+1}, x_i^t, \Delta o_{t+1}\Big)}_{\text{temporal consistency}} + \underbrace{\sum_t \psi^t\big(\mathbf{x}^t\big)}_{\text{label restriction}} \tag{2.12}$$

The *unary potential* is based on the perceptions of the robot. Depending on which features ($\mathcal{F}_t$) or landmarks the robot sees, a *HeatMap* is calculated with the probabilities over each position and rotation. After all probabilities are known, a *flatlist* is created,which is basically a 2D grid with the best pose over all the rotation from each position.

The *temporal consistency* is a constraint, which penalises a prediction, if the odometry data of the proprioceptive sensors of robot suggest very differently. Furthermore, the *label restriction* is another constraint, which allows the robot to be at only one position at time.

2.3.1 Inference

The complexity to solve this minimization problem is usually NP-hard. In [9], iterative random sampling is used to solve this problem. The algorithm takes randomly a pose p_i^t,which is composed out of a rotation and a position in the field, and labels the corresponding state $x_i^t = r$ and evaluates the energy $\mathrm{E}(\mathbf{x}^t)$. If the energy is lower than the previous candidate, then state x_i^t takes label r.

2.3.2 Expansion of model

Taking this model as a basis and knowing that CRF's are quite flexible, several other labels could be added to $\mathcal{L} = \{r, l_\varnothing\}$. Some labels could represent the ball position i.e. $x_i^t = b$ or the different positions of the opponent robots in the field $\mathcal{L}_{op} = \{r_1 \dots r_4\}$. In this new model we have to take care that the ball does not possess a *rotation*. In addition to the unary potential, the *temporal potential* represents also a new challenge regarding the opponent robots. The algorithm should be able to differentiate the percepted robots and compute their temporal potential. Due to the fact, that for the self-Localisation only landmarks and the odometry of the robot were used, the influence of this new information in the localisation of the robot itself should be investigated. Furthermore, the computational cost of expanding the model should not exceed the current solution and their results should be in an accurate and usable margin.

Chapter 3

CRF for Situation Awareness

Having the CRF for self-localization, we will change the energy function in such a way that it includes the information of the ball and the opponents. In this chapter, we will present the theory behind the implementation of the new CRF.

3.1 Energy Function

Because this work is an extension of the CRF for self-localization, the energy function, which has to be minimized, remains a function of the state vector $\mathbf{x} \in \mathbf{X}$. Where $\mathbf{x}$ is a possible instantiation of the random variable $\mathbf{X}$. In this case, each element of the state (x_i^t) takes a label from a set $\mathcal{L} = \{r, b, r_1, r_2, r_3, r_4, l_\varnothing\}$. $l_1 = r$ denotes that there is a robot present, $l_2 = b$ denotes that there is a ball present, r_1, r_2, r_3 and r_4 denote that there is an opponent present and $l_\varnothing$ denotes that there is nothing of interest.

Considering that different labels mean different elements, we have to take care of some details regarding the model. The basis of the model remains as described in [9] on p.9 :

> An element $x_i^t \in \mathbf{x}$ has two indexes, one for the pose (position and orientation of the element) $0 \leq i \leq i_{max} = |\mathcal{P}_t|$ and another one for the time $0 \leq t \leq t_{max} = |\mathcal{T}|$. Where $\mathcal{P}_t$ denotes the set of poses at time t and $|\mathcal{P}_t|$ is the number of poses at time t. $\mathcal{T}$ is the set of time frames which is handled by the CRF and corresponds to the temporal memory of the algorithm.

The difference is that the projection to a specific pose depends on the label of the state. If $x_i^t = r$ then the projection remains the same:

$$\mathbf{p}_i^t = \begin{pmatrix} x_i^t \\ y_i^t \\ \theta_i^t \end{pmatrix} \in \mathcal{P}_t \tag{3.1}$$

where θ is a discretised angle between $[-\pi, \pi]$ But if $x_i^t = b \vee r_l$ the projection is to a 2D-Pose, due to the fact that the rotation of the ball and the rotation of the enemy robots is irrelevant. To achieve this, the rotation for this element is constant:

$$\mathbf{p}_i^t = \begin{pmatrix} x_i^t \\ y_i^t \\ c \end{pmatrix} \in \mathcal{P}_t \tag{3.2}$$

We will see in 3.1.1 and 3.1.2 that this small change will help us to reach what we seek. This leads us to the formal definition of the energy function:

$$\mathrm{E}(\mathbf{x}) = \underbrace{\sum_{i,\,t} \phi_i^t(x_i^t, \mathcal{F}_t)}_{\text{unary}} + \underbrace{\sum_{(i,j)\in\mathcal{N},\,t} \chi_i^t\Big(x_j^{t+1}, x_i^t, \Delta o_{t+1}\Big)}_{\text{temporal consistency}} + \underbrace{\sum_t \psi^t(\mathbf{x}^t)}_{\text{label restriction}} \tag{3.3}$$

Where $(i, j) \in \mathcal{N}$ is the set of indexes which represent neighbouring poses in the graphical model.

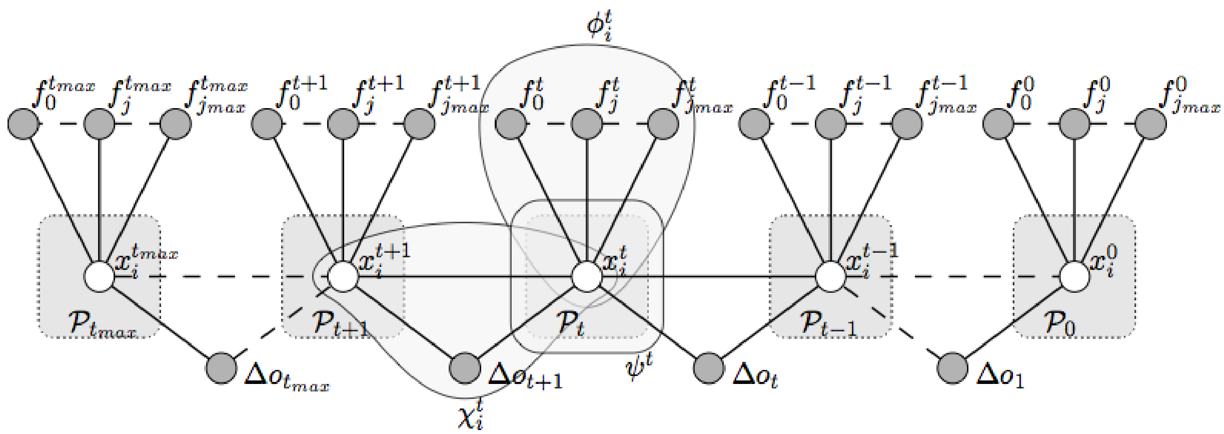


Figure 3.1: Graphical representation of the model. Overview of how the different potential $\psi^t(\mathbf{x}^t)$,$\chi_i^t\Big(x_j^{t+1}, x_i^t, \Delta o_{t+1}\Big)$ and $\phi_i^t(x_i^t, \mathcal{F}_t)$ interact to generate the new state at $\mathcal{P}_t$. f_i^t denotes here the features, from which the unary potentials are created.

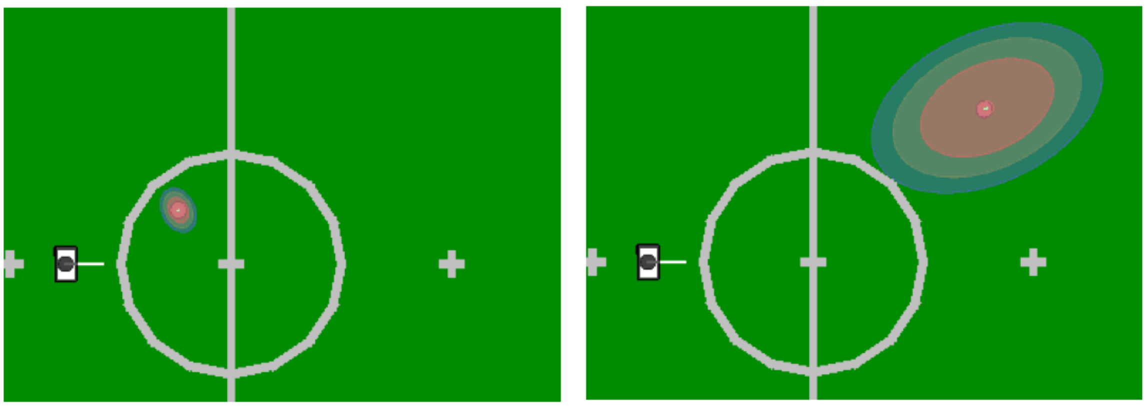
3.1.1 Unary Potentials

Unlike the unary potential of the robot, the potentials of the ball and of the opponents are relative to the one of the robot. In [9], the unary of the robot is explained as follows:

> The unary term ϕ_i^t is based on the scores from the feature detector. The image obtained from the robot camera is analysed and certain features are extracted and used to calculate the unary potential. The features are $\mathbf{f}^t \in \mathcal{F}_t \subset \mathcal{F} = \{$field lines, goal posts, field line corners, center circle$\}$. Each pose gets a score $s_f(p_i^t, f_j^t)$ regarding the feature measurements.

Regarding the ball, we also have to take errors of the measurements into account. We do this by applying an un-normalized multivariate gaussian distribution over the perceived location. This distribution depends on the euclidean distance d_j^t between the percepted location and the position of the robot. Fig. 3.2(a) and Fig. 3.2(b) show some examples. It's un-normalized with a 1 at maximum, because we multiply it by the probability of the robot being in p_j^t, to get the actual probability. The calculation of the covariance is taken from the BH implementation. Due to the fact that the ball does not possess a rotation, we can merge all the results in a 2D map by taking the maximum over each position. Writing the unary potential in a more formal way, we get:

$$\phi_i^t(x_i^t, \mathcal{F}_t) = \begin{cases} \omega_i^t \left(1 - \prod_{f_j^t \in \mathcal{F}_t} s_f\left(p_i^t, f_j^t\right)\right) & \text{if } x_i^t = r \\ \omega_i^t * (1 - gauss(p_j^t, p_i^t, d_j^t) * \phi_i^t(x_j^t, \mathcal{F}_t)) & \text{if } x_i^t = b \vee r_{l \in \{1 \ldots 4\}} \\ 0 & \text{otherwise} \end{cases} \tag{3.4}$$



(a) Covariance of Ball (near)

(b) Covariance of Ball (far)

Figure 3.2: Graphical representation of the distributions, which depends on the euclidean distance of the ball to the robot.

3.1.2 Temporal Consistency Potentials

The temporal consistency potential is our smoothness factor. It connects the global states in a realistic way. To achieve this, we analyze each element of a candidate state isolated from the others.

We can calculate the potential by taking a state from the past x_j^{t+1} and adding the odometry difference $\Delta o_t = o_t - o_{t+1}$ to their respective elements and see how much it differs from the perceived state using the euclidean distance. Here o_t represents the total change of the pose from the start until time t accumulated only by the proprioceptive sensors.

This applies for elements whose odometry is known or accurate enough to be calculated (i.e. Ball). In the case of the opponent robots, which are not seen most of the time, we face another problem. We have to match candidate robots in the new state with robots in the past state. As we have seen before, this is not a trivial problem, due to the fact that the game field is partially observable. We define the smoothness potential of the opponents as the global minimun distance between the robots at time t and those at time t+1 in the model. This is a linear assignment problem which can be solved using a combinatorial optimization algorithm. The formal definition of the assignment problem is: Given two sets, $\mathcal{R}_{t+1}$ and $\mathcal{R}_t$ together with a weight function ω : $\mathcal{R}_{t+1}$ x $\mathcal{R}_t \rightarrow \mathcal{T}$. Find a bijection f : $\mathcal{R}_{t+1} \rightarrow \mathcal{R}_t$ such that the cost function:

$$\sum_{p \in \mathcal{R}_{t+1}}^{N} \omega * dist(p, f(p)) \tag{3.5}$$

is minimized. We define $dist(p, f(p))$ to be the euclidean distance between the robot p and $f(p)$So, the temporal consistency potential is defined formally like,

$$\chi_i^t\left(x_j^{t+1}, x_i^t, \Delta o_{t+1}\right) = \begin{cases} \mathbf{v}_i^{t\top} \cdot \left|p_j^{t+1} + \Delta o_{t+1} - p_i^t\right| & \text{if } x_j^{t+1} = x_i^t = r \\ \frac{\omega_i^t}{|\mathcal{R}_t| * |\mathcal{R}_{t+1}|} * \min_f \sum_{p \in \mathcal{R}_{t+1} \wedge f(p) \in \mathcal{R}_t} dist(p, f(p)) & \text{if } x_j^t \in \mathcal{R}_t \wedge x_j^{t+1} \in \mathcal{R}_{t+1} \\ 0 & \text{otherwise} \end{cases} \tag{3.6}$$

The division by $\mid \mathcal{R}_t \mid * \mid \mathcal{R}_{t+1} \mid$ is justified by the fact that there will be exactly $\mid \mathcal{R}_t \mid * \mid \mathcal{R}_{t+1} \mid$ cases, where

$x_j^t \in \mathcal{R}_t \wedge x_j^{t+1} \in \mathcal{R}_{t+1}$. We will simplify this behavior in the algorithm just by adding

$$\min_f \sum_{p \in \mathcal{R}_{t+1} \wedge f(p) \in \mathcal{R}_t} dist(p, f(p)) \tag{3.7}$$

to the potential.

3.1.3 Label Restriction Potentials

The label restriction potentials enforces that exactly one position has the label r, one position has the label b and at most four positions the label $r_1, \ldots r_4$, by adding infinite energy to the energy function if something different happens.

$$\psi^t\left(\mathbf{x}^t\right) = \begin{cases} 0 & \text{if } \sum_i \mathbb{I}_{\left(x_i^t = r\right)} = 1 \wedge \sum_i \mathbb{I}_{\left(x_i^t = b\right)} = 1 \wedge \sum_i \mathbb{I}_{\left(x_i^t = r_{l \in 1 \ldots 4}\right)} \leq 4 \\ \infty & \text{otherwise} \end{cases} \tag{3.8}$$

$\mathbb{I}$ is the indicator function which evaluates to 1 if the condition in the brackets is true.

Chapter 4

Implementation

This section is dedicated to the technical part of the thesis. Once the theorical basis is stated, the implementation focus on the technique used to achieve performance and correctness. The preliminaries will give a short overview of the environment of the development. Afterwards, we will focus on some changes done from the old CRF to the new CRF and last , but not least, the specific implementation is explained.

Fig Fig. 4.1 shows the modules and representations, which are used to implement the CRF for scene understanding. In BH, *BallLocator* used twelve Kalman filters to generate the *BallModel* and *RobotLocatorUKF* was responsable for the creation of the *RobotsModel*. Both modules were replaced by the newCRF.

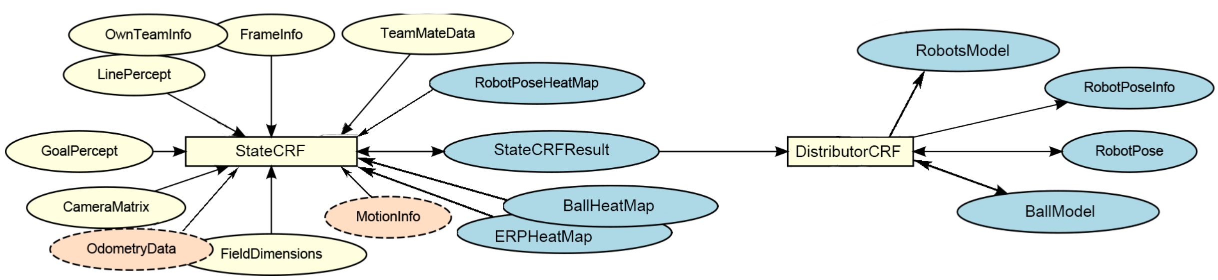


Figure 4.1: Modules and representations for CRF scene understanding. The blue circles stand for representations created by by/for the CRF. Yellow circles are perceptions used by the algorithm. Orange circles are measurements of the odometry. The rectangles are the modules.

4.1 Environment

4.1.1 Software Platform

At the beginning of this work, the idea was to develop the new CRF on the NAO H21 roboter itself, which run bhuman OS2012. The original CRF code for selflocalisation was written for bhuman OS2011. After two months of developing, it became clear, that to be able to use the CRF on the roboters, the whole code has to be written from scratch, because several basic features either disappear or changed drastically in the new version of the interface. Several flags, features and variables which the CRF code uses, took another form in the new operating system. To be able to transpose the code from OS2011 to OS2012, the programmer need knowledges of both systems, modules, representations and APIs to be able to transcript the whole code. In addition, he has to understand the concept of the CRF,reproduce it and handle some new problems i.e. the goal posts have in the version 2012 the same colors. This alone would take several hours of work. Nevertheless, BH 2011's framework provides a fully functional robot operating system, several tools for debugging/testing and a simulation program to the test the code without using the hardware of the robots. More details about the bhuman Operating System and Framework can be found on the official website of BH [3].

4.1.2 Hardware Platform

The robot use the Head version V4.0, which comes with an Atom Z530 CPU with 1.6GHz and 1 GiB RAM. More technical details can be found at the Aldebaran website [1]. The hardware used to run the robot simulator was a MacBookPro7,1 Intel Core 2 Duo 2.4 GHz 4GB DDR3 running Ubuntu 10.10.

4.2 Representations

In the BH framework, representations are the way of passing information from module to module. They can be seen as input/output elements. In this section we will present the different representations, which had to be added for the new CRF.

4.2.1 BallHeatMap (BHM)

The BHM is the result of the unary potential of the ball. It has a vector of the probability for each possible position in the field. As explained in the section 3.1.1, it is relative to the probabilities of the position and rotation of the robot. The BHM has similar parameter as the RoboPoseHeatMap. *xWidth* and *yWidth*, which can be changed by the module *ballHeatMapper*, define the number of cells of the discretised field. In this HeatMap, we also store some information of the *BallPercept*, which will become useful later for the calculation of the CRF. Namely, we store the relative position of the percepted Ball. Some of the most important variables of this class contain can be seen in A.9.

To get a graphical representation of the HeatMap, we can draw each cell over the field with a color representing the probability of the ball being there. If the cell was not updated or the probability is zero, then it is drawn in red, otherwise the colour is determined with the formula:

$$r = 255\frac{s}{s_{max}} \tag{4.1a}$$

$$g = 255\frac{s}{s_{max}} \tag{4.1b}$$

$$b = 64 - 64\frac{s}{s_{max}} \tag{4.1c}$$

If we translate this equations in words, we see that the colour gradient goes from dark blue to yellow as shown in Fig. 4.2.

4.2.2 EnemyRobotPoseHeatMap (ERPHM)

The ERPHM is basically the same as BHM, the only difference, is that we keep track of how many robots have been percepted with the *robotseen* variable and for each robot seen, a *layer* of the HeatMap is initial-

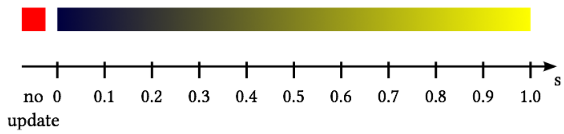


Figure 4.2: Color coding of the probability over a cell

ized. In this section, we have also to mention, that although there are four enemy robots on the field, this variable almost never exceess the number of 2. This is an important observation, which we will follow in 5.

4.2.3 StateCRFResult

This representation in A.13 is the smallest, because it has only one purpose: The synchronisation between the next modules and the CRF. Instead of updating each one of the correspondingly representations at different states of the calculation in the CRF, it is best practice to separate the generation of the models and their distribution.

4.3 Modules

Modules take the data of the representations and uses it to generate other representations. As seen before, we will replace the modules *BallLocator* and *RobotLocatorUKF* with *StateCRF*. A helper module *DistributorCRF* will distribute the results of the CRF to the corresponding representations . In addition, we have to created other modules, which will create *BHM* and *ERPHM*, these are *BallHeatMapper* and *ERPHeatMapper*.

4.3.1 BallHeatMapper

In this module, we iterate over all the positions in the *RoboPoseHeatmap* and, accordinly to 3.1.1, together with the relative perception of the ball from *BallPercept*, we create the distribtutions over the field. It's heavily inspired by the *RoboPoseHeatMapper*. Taking advantage of the fact that the rotation is not important, we initialize the BallHeatMap as a 2D field instead of a 3D like the RoboPoseHeatMap.

Parameters

Like *RobotPoseHeatMap*, this module has parameters which define the behavior of the algorithm. Using the BH framework, those parameters can be changed during a real time simulation using the *set* and *get* commands in the simulator.

The module has several parameters which influences its behaviour and the *RobotPoseHeatMap*. They can be changed during run time from the simulator by the *set* command. As all parameters have to be addressed with the *set* command and there is quite a number of them, it is advised to use first the *get* command which gives back the current parameter values and a prepared *set* statement, as shown in the next listing **??**. A brief description of the parameters is given in table 4.1

```
get module:BallHeatMapper:parameter
```

Algorithm 1 Compute probabilities in BallHeatMap

$ballHeatMap.init()$
$P_b = ballPercept.relPos$ {Relative position of ball to robot}
for each robot orientation θ_r **do**
  for each robot position ${}_W\mathbf{p}_r$ **do**
    $Cov = getCovOfPixelInWorld({}_W\mathbf{p}_r, \theta_r, P_b)$ {Covariance of Percepted Pixel in World}
    $\mathbf{Gp} = getPositions({}_W\mathbf{p}_r, \theta_r, P_b, Cov)$ {Position from the calculated BallCandidates with Covariance}
    for each position $g\mathbf{p} \in \mathbf{Gp}$ **do**
      $ballHeatMap[g\mathbf{p}] = max(ballHeatMap[g\mathbf{p}], getGaussProb(g\mathbf{p},_W \mathbf{p}_r, \theta_r, P_b))$ {Get Prob of certain position given the gauss distribution}
    end for
  end for
end for

Parameter	Name	Description
1	xWidth	Size along x-Coord. of a cell
2	yWidth	Size along y-Coord. of a cell
3	stepSizeDraw	This parameter is used to reduce the number of cells drawn on the field to speed up the process
4	maxProbThreshold	Threshold for cell to be ignore while scanning *RobotPoseHeatMap* to speed up the process
5	robotRorationDeviation.x	Used to calculate the Covariance Matrix
6	robotRorationDeviation.y	Used to calculate the Covariance Matrix

Table 4.1: Parameter of BallHeatMapper

```
set module:BallHeatMapper:parameter xWidth = 70; yWidth = 70;stepSizeDraw = 3;
    maxProbThreshold = 0.1;robotRotationDeviation.x=0.02f;
    robotRotationDeviation.y=0.08f;
\label{GetSet}
```

Listing 4.1: Full *get* statement and the resulting *set* statement to change the parameters of *BallHeatMapper*

4.3.2 EnemyRoboPoseHeatMapper

The *ERPHeatMapper* uses the same algorithm as the *BallHeatMapper* for each robot seen by *RobotsPercept* and almost the same parameters. It creates a new *layer* of the Heatmap depending of the number of robot seen. For those layer a new parameter *NumOfRobots* is included in the class, which is pulled from *RobotPercept*.

4.3.3 StateCRF

This module is the core of the whole algorithm. It approximates energy function $\mathbf{x}^\star = \arg\min_\mathbf{x} \mathrm{E}(\mathbf{x})$ as mentioned in 2.3.1. Basically, *StateCRF* is *RoboPoseCRF*, but extended. Several new features for model generation were added, a complete refactoring of the class was performed and the CRF was extended.

Parameters

The parameters used in the *StateCRF* are almost the same as for *RobotPoseCRF*. The only difference is $getCandidateMode = 3$ and $candidateThreshold = 0.3$ A list of the parameters and of their default values can be seen in 4.2:

Model generation

In *StateCRF*, after the CRF have been executed and the results are stored locally. To use the results later, we create the models of the needed objects, which will be needed afterwards by the framework. Those model are passed to the representation *StateCRFResult*. Afterwards *DistributorCRF* is in charge of distributing them accordingly. We do this, because the request of the models have to wait, until the CRF finishes.

The odometry of the ball is approximated, by calculating the translation between the last CRF result and the current one and dividing by the time difference. Here we make the assumption, that the ball continuous with a slighty less speed, due to the friction of the ball with the ground. The models and the correnponding methods are:

- RobotPose - void createRobotPose();
- RobotPoseInfo - void createRobotPoseInfo();
- BallModel - void createBallModel();
- RobotsModel - void createRobotsModel();

Changes

In this subsection, we will address some of the changes done to the code of *RoboPoseCRF* in *StateCRF*. Some of the changes may seem to lead to a performance loose, if we ignore the fact, that the temporal and unary potentials of the ball and of the robot are also calculated.

- As mentioned in 2.3, one of the optimizations was to merge the three dimensional *RobotPoseHeatMap* into a 2D *flatlist* with the poses with most probabilities over each position. For the new CRF, the whole *RobotPoseHeatMap* is used, not only the *flatlist*. Intuitively, this appears to be a loose in performance. This is ,however, not the case. For more details, we refer to the section **??**.
- Refactoring. One of the most important qualities of a code, is its ability to be readable. A method beyond 100 lines of code is basically unreadable. This has to do with the human short term memory. The limited duration of it quickly suggests that its contents spontaneously decay over time. The decay assumption is part of many theories of short-term memory, the most notable one being Baddeley's model of working memory [2]. That is the reason why large methods in the class *RobotPoseCRF* were divided into many much shorter ones.

4.3.4 LogWriter

This small module allows to extract information to the system and store it in an external .csv file for other purposes. One of this purposes, is the evaluation of the data generated by *StateCRF*. For more details, we refer to 5

4.3.5 Hungarian Algorithm

To implement the KuhnMunkres algorithm again would be a good practical excercise for computer science, but unusable for performance critical modules. In out case, we prefer to use an optimazed version created by John Weaver as an open source project. The project can be found in GitHub.

Parameter	Description	Default Value
bufferSize	Number of time frames stored in the buffer. Maximum value is 150.	5
numberOfIterations	Number of iterations through the whole buffer.	50
numberOfCandidateIterations	Maximum number of how often a new candidate within the same time frame is selected.	500
candidateIterationsAsFraction	Defines how often a new candidate within the same time frame is selected, if *candidateIterationAsFraction* times *numberOfCandidates* is smaller than *numberOfCandidateIterations*.	0.2
frameStep	Determines how many frames should be left out before the next one is saved in the buffer. Defines the temporal resolution and together with the frame rate (30Hz) and *buffersize* the temporal length of the memory.	10
getCandidateMode	Defines which method should be used to generate the candidate list. (whole heatMap, flatMap, threshold).	3
candidateThreshold	The threshold for *getCandidateMethod* = 3.	0.3
alphaUnary	Weight of the currently processed candidate unary potential, except for the last entry (most current) in buffer.	30.0
alphaX alphaY alphaRot	Weight of odometry potentials based on past frame. Propagates pose forward in time.	1.0 1.0 1.0
betaUnary	Weight for unary potential of last (most current) entry in buffer.	30.0
betaX betaY betaRot	Weight of odometry potentials based on future frame. Propagates pose backward in time.	1.0 1.0 1.0

Table 4.2: Parameters of StateCRF

Chapter 5

Experiments and Results

Each experiment reflects an important property of the developed model. After defining their differing test setups, the experiment are evaluated. Observations are noted as well as several aspects regarding the difference between EKF, CRF in [9] and our model. On the first half, a direct comparision of the accuracy with different parameters will reveal, if the behavior of the new CRF is similar to the old one. Afterwards, the new elements, i.e. ball positioning, ball error and opponent influence will be evaluated. This leads up to the analysis of the experiements in 6.

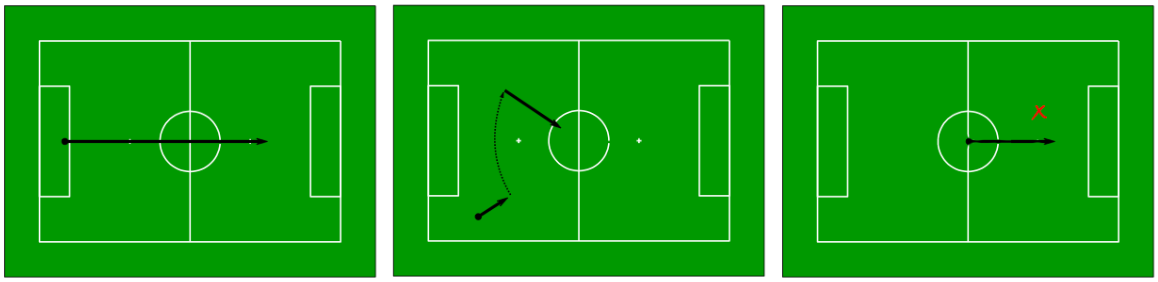
5.1 Test Setups

To be able to compare the results with the old CRF, the exact same scenarios have to be used. They are defined in chapter - Experiments and Result - in [9]. The directly quotation is:

> The first test scenario 5.1(a) is used to determine the accuracy in a well defined, optimal environment. Therefore the robot is placed at the coordinates $(2000, 0)$ which is in front of the opposite goal. Then it is directed to walk in a straight line towards his own goal. The measurement of interest here is the distance between the robot pose from the self-localisation algorithm and the ground truth.
>
> The second scenario 5.1(b) is to find out how well the robot can recover from kidnapping. The robot is placed at $(2000, 1500, 225)$ and directed to walk towards the centre of the field. As soon as it reaches the coordinate $(1490, 1170)$ it is manually replaced to $(1500, -1000, 135)$ and resumed walking towards the centre. To measure the performance of recovering from kidnapping, the time is measured from the replacement until the estimated robot pose is within 30cm of the ground truth robot pose.

The camparasion between the two CRF will show the improvement of the self localization using the data of the ball and opponents. In addition to those cases a third experiment well proof, that the temporal potential of the ball has a significant impact on the self localozation, by measuring only the error of the *rotation* of the robot in a similiar scenario as the first one 5.1(a). The robot is placed again at the coordinates $(2000, 0)$. Then it is directed to walk in a straight line towards his own goal visualizing the ball at the entrance of his own goal.

A forth scenario is to find out, if the improvement of the selflocalization using the temporal consistency potential of the opponents is worth the computational costs. For this, we will set the robot at $(0, 0)$ and the ball again at the entrace of his goal at $(-2000, 0)$. Because the perception layer can track at most two



(a) Accuracy (b) Kidnapping (c) Opponent

Figure 5.1: Camparative Test Cases

opponents at the same time, we will put only one opponent at $(-1500, -1000)$. Comparing the errors and the times, will give us a result.

5.2 Simulator

The Nao robots now a days run the operating system of BH v.2012, which is different as the one used to develop the code for the CRF. This is the reason why most of the test were conducted with the simulator *SimRobot*. The simulator is part of the framework of BH and it has a lot of usable feature for developping and debugging. Sadly, there is no API which allows *SimRobot* to interact with other software i.e. for evaluation. The module *LogWriter* allows us to write the output of the console in a file format called *.CSV*, which can be used as a bridge between the simulator and other software. The logging module currently exports following data:

- **time**: theFrameInfo.time/1000.0f
- **groundTruthX**: theGroundTruthRobotPose.translation.x
- **groundTruthY**: theGroundTruthRobotPose.translation.y
- **groundTruthR**: theGroundTruthRobotPose.rotation
- **robotPoseX**: theRobotPose.translation.x
- **robotPoseY**: theRobotPose.translation.y
- **robotPoseR**: theRobotPose.rotation
- **ballGroundTruthX**: theGroundTruthBallModel.estimate.position.x
- **ballGroundTruthY**: theGroundTruthBallModel.estimate.position.y
- **ballPoseX**: theBallModel.estimate.position.x
- **ballPoseY**:theBallModel.estimate.position.y

The reason why the opponent robots data is not being exported, is because most of the time its empty. The perception layer doesn't recognize a robot if it is far away. They are recognized only if the robots are relatively near to the robot. For this special case, we refer to the experiment 5.4.2.

For practical reasons, printing debug messages on the console during the test have be disabled. This can be done by writting following commands in the console:

```
msg off
msg log <fileName>.csv
```

Listing 5.1: Switch off printing of messages before starting the logger

If not mentioned otherwise, then for the experiments the default settings in Appendix B are used.

5.3 Accuracy

With this extension, the impact of the parameters may have changed. For that reason, we investigated the influece of different parameters on the accuracy of the algorithm.

5.3.1 Unary vs. Temporal Consistency Potentials

Unary and temporal potentials have different magnitudes. To regulate the behavior of the CRF, weights have to be integrated in the model. The figure 5.2 plots an diagram with the influece of the weights and the total error of the BallModel and RobotPoseModel. We have to mention here, that the error of the BallModel is relative, not absolute. :

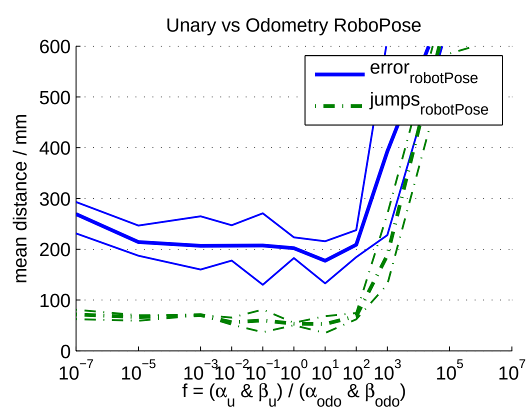
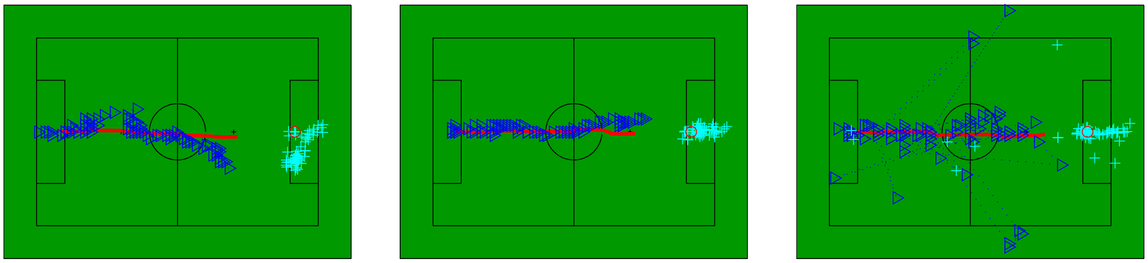


Figure 5.2: Influence of unary potentials versus temporal consistency potentials in the error of the robot in scenario 5.1(a).

We can clearly see, how the different weights affect the algorithm. In 5.3(a) the odometry has more impact in the decicion of the CRF then the unary. We can cleary see, how the errors of the measurements affect the accuracy of the algorithm, while in 5.3(c) the algorithm takes the position with highest probability regardless of the smoothness of the solution.

5.3.2 Kidnapping

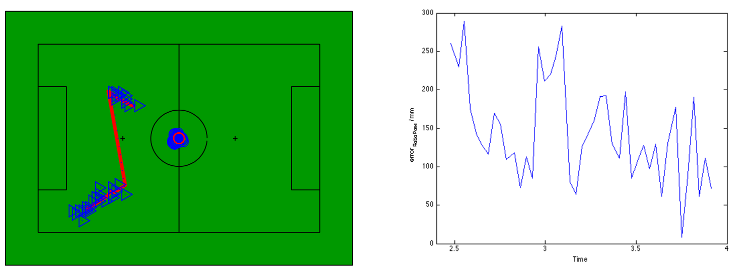
For the kidnapping problem, we can see that the correction of the error happens approximately within a 0.33s window. In 5.4(a), the error never excess the 30cm threshold, which means that the robot always knew pretty



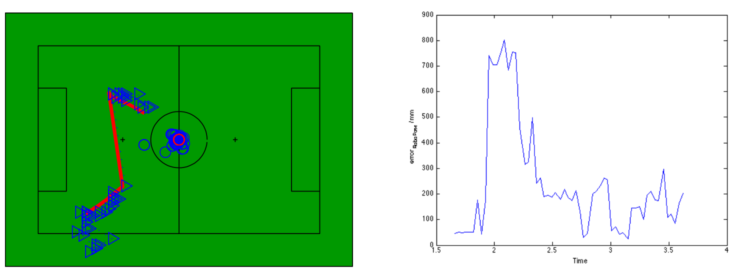
(a) Ratio = 1 / 10000 (b) Ratio = 10/ 1 (c) Ratio = 10000 / 1

Figure 5.3: Example of the walking route with parameInfluence of unary potentials versus temporal consistency potentials in scenario 5.1(a). the red elements are the GroundTruth, while the blue elements are the estimates of the CRF

accurately, where he was. A more common scenario is 5.4(c). The robot realized, at 2 sec. mark, the he was probably wrong. He was able to correct his position around 0.4 sec later.



(a) Kidnapping Scenario 1 (b) Error of 5.4(a)



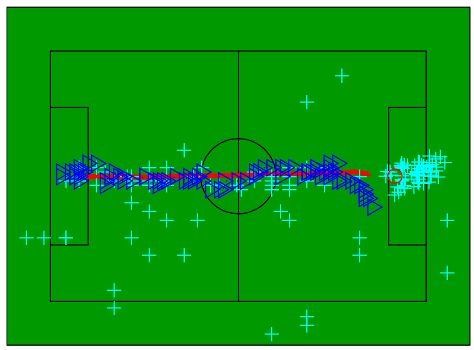
(c) Kidnapping Scenario 2 (d) Error of 5.4(d)

Figure 5.4: Different outcomes of the kidnapping experiment

The influence of the ball into the recovering time is very difficult to state, due to the fact that the times are very similar.

5.3.3 Flatlist

On the first runs of the experiments, a strange behavior of the ball indicated to an important difference in the new algorithm. The old CRF uses different strategies for performance optimation. One of those strategies is called *flatlist*. This list, reduces the complexity of the textitRoboPoseHeatMap from a $185 * 135 * 8$ element list to a $185 * 135$ list by choosing over each position the candidate with most probability as a representant of the cell. Created for perfomance improvement, the new CRF seems to work better and faster without it. The plot 5.5 shows how many ball jumps accour.



(a) Ball Jumps

Figure 5.5: Examples of jumps of the ball marked as blue crosses on the map, because of flatlist.

Such jumps should be avoided thank the temporal concistency potencial of the ball. Later, we realized that the algorithm worked fine, but the correct rotation of the robot was deleted from the map, because another rotation on the same cell had a slighly better unary potential. Therefore, the new CRF acted correctly by choosing the best wrong state. Using the whole *RoboPoseHeatMap* with a certain threshold instead of the *flatlist* implices that there are much more position candidates to consider then before. At the same time, the parameter *numberOfCandidateIterations* in *StateCRF* remains unchanged. This means that it doesnt matter that our pool of candidates is bigger, the CRF will execute the same amount of calls. Fig. 5.6 shows even, that the time to execute *executeCRF* is even much less then before and the timing of *getCandidates* did not increase much.

Module	Min	Max	Avg
StateCRF:executeCRF	70.824	121.184	109.927
StateCRF:getCandidates	0.550	1.001	0.642

(a) Timing with Flatlist

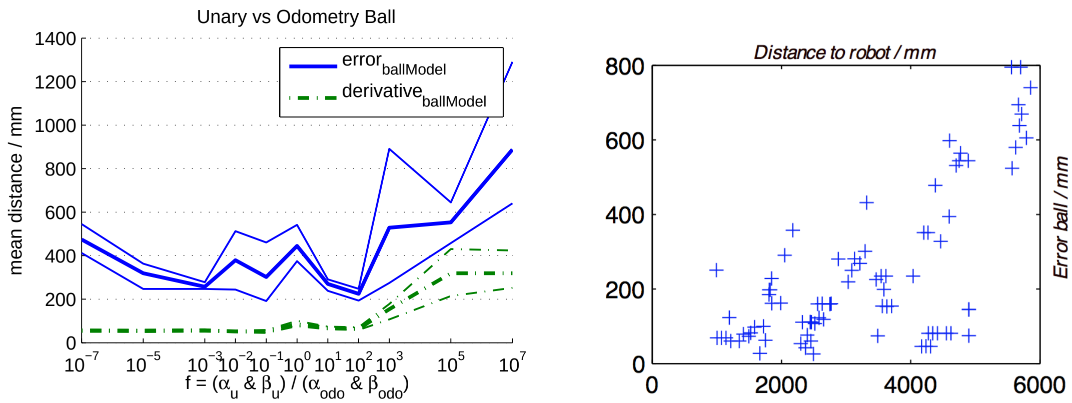
Module	Min	Max	Avg
StateCRF:executeCRF	1.100	123.983	63.466
StateCRF:getCandidates	0.916	30.545	6.829

(b) Timing without Flatlist. Threshold= 0.3

Figure 5.6: Execution times of the CRF with and without *flatlist*.

The observed time reduction in the average execution of the CRF is significant, due to the use of a thresholded HeatMap instead the flatlist. By eliminating the flatlist and chosing only positions above a certina threshold, we improve the average quality of the candidates. By improving the average quality of the candidates, the possibility to get an minor energy than the last execution is less and thus a lot of memory write operations are avoided. This leads to the improve of the algorithm.

5.4 Ball and Opponents in CRF



(a) Error of the Ball at different parameters

(b) Error of the Ball vs Distance to Robot with parameter ratio =100

Figure 5.7: Different ball error plot for scenario 5.1(a).

After the evaluation of the experiments, the error of the ball seemed to be relative high. The next plot shows that the error of the ball in the new CRF is compared to the Kalman-filter greater. This let us suspect, that the implementation do not correspond to the developed model. By examining the code again, we came to the conclusion that the generation of the ball candidates is not optimal. Due to time aspects, this issue is a suggestion of improvement

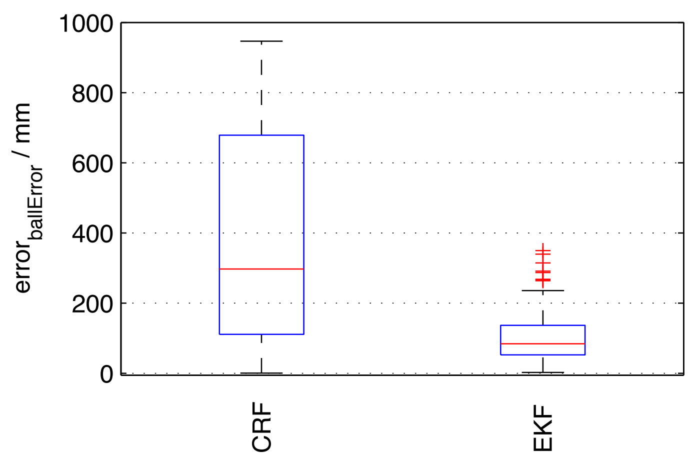
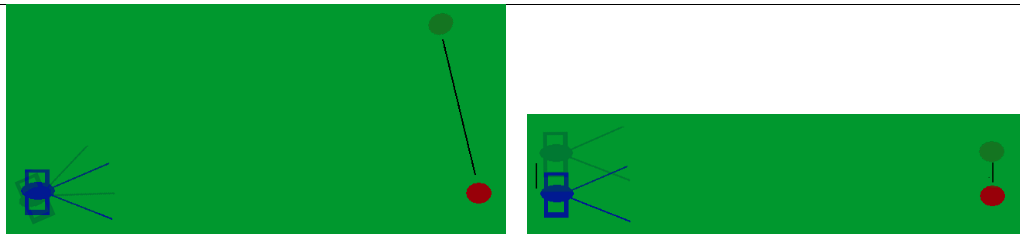


Figure 5.8: The error of the CRF is dramatically high then the EKF.

5.4.1 Improvement of RoboPose through Ball

Having knowledge of the position of the ball and of the opponents, can help the robot in positioning himself. One way of how it can improve the positioning, is through the temporal consistency of the ball. If there is



(a) Correct place, wrong rotation (b) Slightly wrong place, correct rotation

Figure 5.9: Black line shows the increase of the temporal potential.

information about the ball, which is a certain distance, a wrong rotation of the robot, will have a hugh impact in the smoothness of the ball. In other words, fot the robot it is better to be slighly mispositioned and having the correct rotation, then be at the perfect position and miss the angle. Running the scenario 5.1(a) a few times without the smoothness and with the smoothness of the ball, we get a boxplot which shows an improvement in the error of the rotation of the robot.

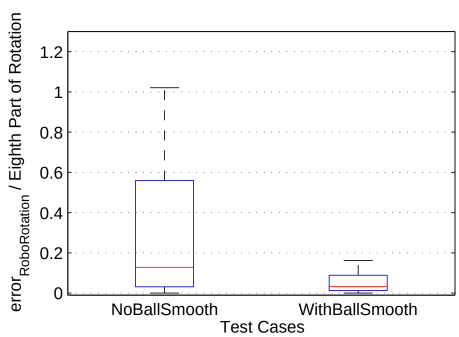
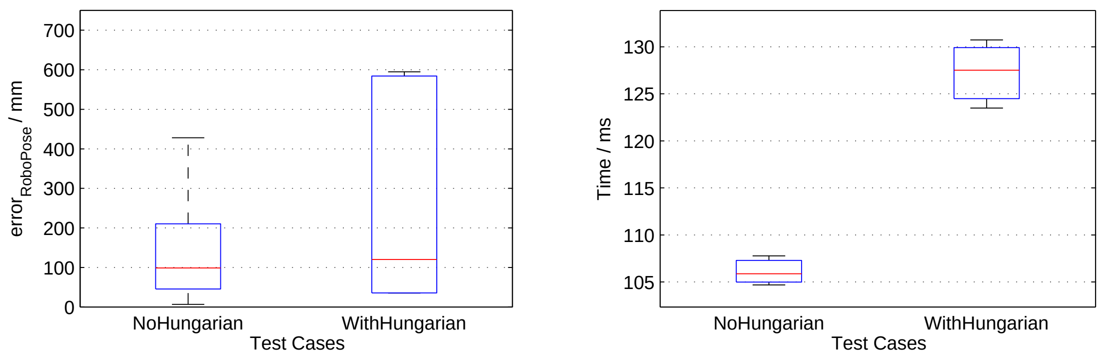


Figure 5.10: Average Error in 8^{th} of rotation

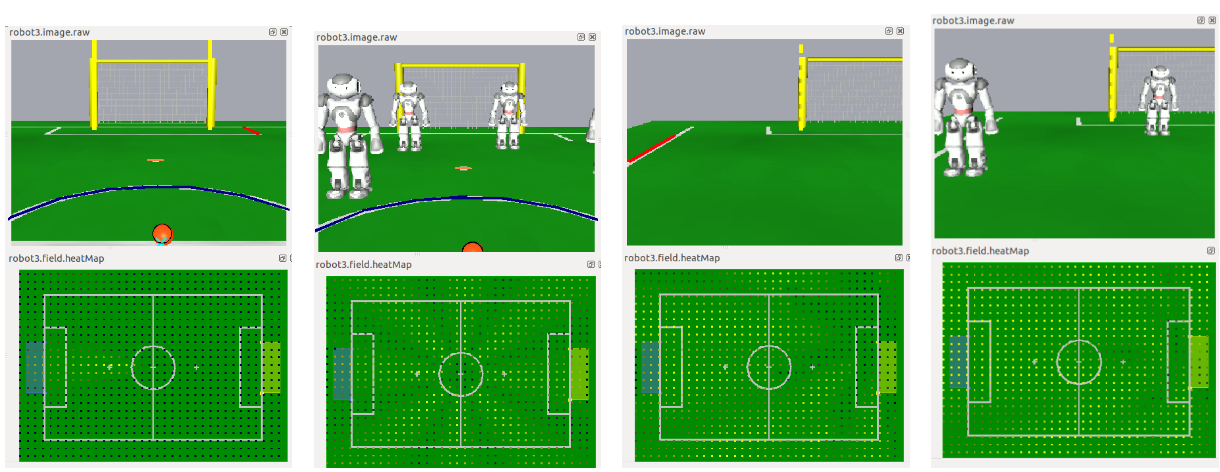
5.4.2 Hungarian Algorithm

The assignment problem of the opponent robot is a major topic. Although it can be solved in polynomial time $O(n^3)$ with the hungarian algorithm, this experiment showed, that the numbers of memory access increases the time of calculation significantly. The Hungarian Algorithm is very efficient for large n. Because there are at total of only four enemy robots, the repeatedly reads and write to the different matrices in the algorithm make this solution unusable for the RoboCup. On the other side, although the time increased significantly by 21.75ms (5.11(b)), the average error did not decrease (5.11(a)). The greater variance and the same average error in 5.11(a) is due to the fact, that the robot partially covers lines which are needed for the unary potential. This leads to a worse heatmap as we can see in the following image:



(a) Error (b) Time

Figure 5.11: Graphical representation of the errors and timing in scenario 5.1(c) with and without the temporal consistency potential of the opponent robots



(a) Case 1 No Robots (b) Case 1 With Robots (c) Case 2 No Robots (d) Case 2 With Robots

Figure 5.12: Different situation where the opponents worsens the RoboPoseHeatMap

Chapter 6

Discussion

The results of the experiments did not only confirm the theory, they also showed a misbehavior in the implementation and helped us to uncover a feature with a wrong funtionality, which after correction lead us to an improvement in accuracy and performance. In this section, we will discuss this results in detail.

6.1 Accuracy

A big difference between the new CRF and the old one, is the influence of the temporal concistency potential on the algorithm. In Fig. 5.2, we can see that the error with high weights in the temporal potential is not that severe as expected. As a matter of fact, the error is three times less than the old CRF. This is the case, because an error in the position is not only penalized by the robot, but by the ball and the opponents at the same time. So, although the robot follows his odometry, which is not perfect due to miss calibration of the joints and some other errors, he autocorrects the trajectory slightly using the ball as a reference. On the other side, the influence of the unary potentials was as expected. The greater the weights at the unary potentials, the unsmoother gets the trajection.

The result of the error of the ball in 5.7(a) showed a miss bahavior in the implementation of the algorithm. Although the ball was stationary, the error was relative high. The direct comparition in **??** strengthened the conclusion that the ball prediction didnt work as expected. By verifying the code it became clear, that the estimation of the ball position was highly dependend on the unary potential. This problem can be solved, if the ball prediction candidates would be chosen more randomly.

At a early experiment stage, the sporadically jumps of the ball indicate to another anomaly in the implementation. The culpit was a performace feature called *flatlist*,which improved the selflocalization in the old CRF. Accidentally, the *flatlist* erased states with the lowest energy, by overriding different rotations on the same cell based on the unary potential. 5.6 and 5.3(b) not only proofed that without the *flatlist* those jumps disappeared, they also proofed that the new algorithm is in average almost twice as fast as the old one.

6.2 Kidnapping

The repositioning of the robot was on the old CRF excellent. The results of 5.4(b) and 5.4(d) confirmed that the performace was still the same. In the new CRF, the rotation estimation improved due to the information of the positioning of the ball in the field, as 5.10 showed. A wrong rotation was penalized by the temporal potential of the ball. This had the side effect, that the robot prefered to be slighty in a wrong position and having the correct angle, then otherwise. This effect had no significantly global influece.

6.3 Temporal Potential of Opponents

Although the theoretical assigment problem of opponent robots is solvable in polynomial time with the Hungarian Algorithm, the size of the problem in our case and the time requirements make this solution unusable. In 5.11(b) we can see that the time increased significantly, while the error stayed in average the same. This is due to the fact, that robots visually block some important landmarks (i.e. lines, goal posts), which the algorithm needs to create the RoboPoseHeatMap. 5.11(a) shows that this trade off is not worth. Together with the fact, that the perception layer never recognizes more than 2 robots in the same frame, we conclude that the ball has a much bigger influece in the algorithm then the opponents.

6.4 Computational Cost

The cost of execution the CRF was reduced by the half, which is relativly high, but this doesnt change the overall effort. The computation of the RoboPoseHeatMap is still the most expensive operation in the whole algorithm. You can claim, that the new CRF doesnt use twelve EKF for the prediction of the ball and 1 EKF fot the opponent Robots . This is true, but, if we take the values from [9], the difference between the calculation of 100 particle in 1 ms and $185 * 135 * 8 \approx 200000$ cells in around 1000ms is so big, the influence of that improvement is minimal.

Chapter 7

Conclusion

The experiments proofed that the implementation of the CRF in an embedded system like the NAO robots is factible. The self-localization of the robot is as accurate as the other investigated methods. We could also proof, that the parameters for the temporal concistency stabilazed the error for extrem values. After some changes in the parameters, we were able to half the time of calculation of the CRF without losing accuracy. In addition, the global ball position helped to improve the rotation estimation of the robot and the repositioning. The time needed to recover after being kidnapped remained below the 0.3 sec mark, which is almost 16 times faster then the BH approach. In conclusion, we can claim, that the CRF is a very good alternative to Kalman filters and with further optimations it can be even better.

Appendix A

Appendix Code

```
/**
 * @file ERPHeatMapper.cpp
 * Creates the HeatMap for the Enemy Robots.
 * @author Hans Hardmeier
 * @date 24/04/13
 * Copyright (c) 2013 __ETHZ__. All rights reserved.
 */

#include "ERPHeatMapper.h"  //Include Header


ERPHeatMapper::ERPHeatMapper():
    parameter(new HMERParameter)
{
    initdone=false;
}

// Destructor
ERPHeatMapper::~ERPHeatMapper()
{
    delete parameter;
}

void ERPHeatMapper::init(ERPHeatMap& erpHeatMap)
{
    erpHeatMap.resize(parameter->NumOfRobots, parameter->xWidth, parameter->
        yWidth);
    initdone=true;
}



/**
Get Covariance Matrix of a Pixel in the World using global parameters
*/
```

```
Matrix2x2f ERPHeatMapper::getCovOfPixelInWorld(const Vector2<>&
    correctedPointInImage, float pointZInWorld) const
{
// From B-Human BallLocator
//Problems: Independet of Movement or State

    const Vector3<> unscaledVectorToPoint(theCameraInfo.focalLength,
        theCameraInfo.opticalCenter.x - correctedPointInImage.x, theCameraInfo
        .opticalCenter.y - correctedPointInImage.y);
    const Vector3<> unscaledWorld = theCameraMatrix.rotation *
        unscaledVectorToPoint;
    const float h = theCameraMatrix.translation.z - pointZInWorld;
    const float scale = h / -unscaledWorld.z;
    Vector2<> pointInWorld(unscaledWorld.x * scale, unscaledWorld.y * scale);
    const float distance = pointInWorld.abs();
    const float angle = pointInWorld.angle();
    const float c = cos(angle), s = sin(angle);
    Matrix2x2f rot(Vector2f(c, s), Vector2f(-s, c));
    Matrix2x2f cov(Vector2f(sqr(h / tan((distance == 0.f ? pi_2 : atan(h /
        distance)) - parameter->robotRotationDeviation.x) - distance), 0.f),
                   Vector2f(0.f, sqr(tan(parameter->robotRotationDeviation.y)
                       * distance)));
    return rot * cov * rot.transpose();
}

/**
Given the RoboPosition and rel Ball Position, calculate global Pos Position
@param relBallPos Relative Ball Position to the Robot's View Position
@param xView x-axis direction relation of View direction;
@return Global Pos of Ball in mm
*/

Vector2f ERPHeatMapper::getPossibleGlobalRoboPos(Vector2<int> roboPos,Vector2
    <> relBallPos,
        Vector2f xView,Vector2f yView)
{
    Vector2f result = (xView*relBallPos.x)+(yView*relBallPos.y);
    result.x+=roboPos.x;
    result.y+=roboPos.y;
    return result;
}

/**
Calculate the Probability (using a Covariance Distribution) of pos relative to
    mean
*/

float ERPHeatMapper::getUnscaledProbabilityAt(const Vector2f& mean, const
    Matrix2x2f& cov, const Vector2f& pos) const
{
    Vector2f diff = pos - mean;
```

```
    float exponent = diff * (cov.invert() * diff);
    float p = exp(-0.5f * exponent);
    return max(p, 0.01f);
}

void ERPHeatMapper::update(ERPHeatMap& erpHeatMap)
{
    if(!initdone)
    {
        init(erpHeatMap);
    }
    MODIFY("module:ERPHeatMapper:parameter", *parameter);
    // reset weights
    erpHeatMap.reset();
    erpHeatMap.robotsSeen = theRobotPercept.robots.size();
    erpHeatMap.stepDraw = parameter->stepSizeDraw;
        erpHeatMap.maxProb = 0.0f;
        erpHeatMap.updated = false;
        erpHeatMap.flatListIsUpdated=false;

    if(!theRobotPercept.robots.empty() && theRobotPoseHeatMap.updated)
    {
        updateHeatMap(erpHeatMap);
    }
  //  OUTPUT(idText,text,"Pos: "<<erpHeatMap.robotsSeen);
}

void ERPHeatMapper::updateHeatMap(ERPHeatMap& erpHeatMap)
{
    roboPositionThreshold = theRobotPoseHeatMap.maxProb - parameter->
        maxProbThreshold;
    if(roboPositionThreshold<0)
    {
        roboPositionThreshold=0;
    }

    int i=0; // Index in HeatMap for Robot r
    for(RobotPercept::RCIt r(theRobotPercept.robots.begin()); r !=
        theRobotPercept.robots.end(); r++)
    {
     //    if((*r).teamRed != parameter->enemyteamRed)
     //        continue;
        //Calculate if Robot on Field
        Vector2<> relPosOnField;
        if(!Geometry::calculatePointOnField(theImageCoordinateSystem.
            toCorrected((*r).lowestPx), theCameraMatrix, theCameraInfo,
            relPosOnField))
        {
            continue;
        }
        for(int rot = 0; rot < theRobotPoseHeatMap.maxIndex[0]; ++rot)
```

```
        {
            roboPerceptCov = getCovOfPixelInWorld(theImageCoordinateSystem.
                toCorrected((*r).lowestPx),0.f);
            // Calculate relative Coordinate System
            float viewAngle = 2*pi*((float)rot/(float)theRobotPoseHeatMap.
                maxIndex[0]);
            Vector2f xView(cos(viewAngle),sin(viewAngle));
            Vector2f yView(-xView.y,xView.x);

            for(int x = 0;x < theRobotPoseHeatMap.maxIndex[1];++x)
            {
                for(int y = 0; y < theRobotPoseHeatMap.maxIndex[2]; ++y)
                {
                    long index=theRobotPoseHeatMap.getIndex(rot,x,y);
                    if(theRobotPoseHeatMap.heatMap[index]>=
                        roboPositionThreshold)
                    {
                        //Global Robot Position in question:
                        Vector2<int> roboPosition = erpHeatMap.getPos(x,y);

                        //Possible Global Robo Position
                        Vector2f enemyRoboPosition =getPossibleGlobalRoboPos(
                            roboPosition,relPosOnField,xView,yView);

                        if(isNotInRange(erpHeatMap,enemyRoboPosition))
                            continue; //Continue if roboPosition out ouf Range

                        //Calculate start of Covariance Field
                        float radius=max(sqrt(roboPerceptCov.c[0][0]),sqrt(
                            roboPerceptCov.c[1][1]));
                        int cellSize=min(erpHeatMap.cell[0],erpHeatMap.cell
                            [1]);
                        int radiusCell=radius/cellSize;
                        Vector2<int> roboCoords = erpHeatMap.getCoords((int)
                            enemyRoboPosition.x,(int)enemyRoboPosition.y);
                        Vector2<int> start(max(roboCoords.x-radiusCell,0),max
                            (0,roboCoords.y-radiusCell));
                        Vector2<int> end(min(roboCoords.x+radiusCell,
                            erpHeatMap.maxIndex[1]),
                                    min(roboCoords.y+radiusCell,
                                        erpHeatMap.maxIndex[2]));
                        //Start update Neighbours in HeatMap
                        for(int covX=start.x; covX<=end.x; covX++)
                        {
                            for(int covY=start.y; covY<=end.y; covY++)
                            {
                                Vector2<int> temp = erpHeatMap.getPos(covX,
                                    covY);
                                Vector2f neighbour((float)temp.x,(float)temp.y
                                    );
```

```
                                        float prob=getUnscaledProbabilityAt(
                                            enemyRoboPosition,roboPerceptCov,neighbour
                                            );
                                        long indexBHM = erpHeatMap.getIndex(i,covX,
                                            covY);
                                        if(erpHeatMap.heatMap[indexBHM]<(
                                            theRobotPoseHeatMap.heatMap[index]*prob))
                                        {
                                             erpHeatMap.heatMap[indexBHM]=
                                                 theRobotPoseHeatMap.heatMap[index]*
                                                 prob;
                                             if(erpHeatMap.maxProb<theRobotPoseHeatMap.
                                                 heatMap[index]*prob)
                                             {
                                                  erpHeatMap.maxProb=
                                                      theRobotPoseHeatMap.heatMap[index
                                                      ]*prob;
                                             }
                                        }
                                   }
                              }
                         }
                    }
               }
          }


     erpHeatMap.robots[i].relPosOnField=relPosOnField;
     erpHeatMap.robots[i].teamRed = (*r).teamRed;
     erpHeatMap.robots[i].standing = (*r).standing;
     Matrix2x2<> temp(Vector2<>(roboPerceptCov.c[0].x,roboPerceptCov.c[0].y),
                      Vector2<>(roboPerceptCov.c[1].x,roboPerceptCov.c[1].y));
     erpHeatMap.robots[i].covariance =temp;
     erpHeatMap.robots[i].timeStamp=theFrameInfo.time;
     i++;
     }
     erpHeatMap.updated=true;
}

bool ERPHeatMapper::isNotInRange(ERPHeatMap& bhm, Vector2f pos)
{
     return (pos.x<-bhm.carpet[0]/2 || pos.y<-bhm.carpet[1]/2 || pos.x>bhm.
         carpet[0]/2 || pos.y > bhm.carpet[1]/2);
}

MAKE_MODULE(ERPHeatMapper, Modeling)
```

Listing A.1: ERHeatMapper.cpp

```
/**
 * @file ERPHeatMapper.h
 * Creates the HeatMap for the Enemy Robots.
```

```
 * @author Hans Hardmeier
 * @date 24/04/13
 * Copyright (c) 2013 __ETHZ__. All rights reserved.
 */

#pragma once        //Makes sure that the source file is only included once in
    the compilation
#include "Tools/Module/Module.h"

#include "Representations/Modeling/ERPHeatMap.h"

#include "Representations/Perception/BallPercept.h"

#include "Representations/Perception/RobotPercept.h"

#include "Representations/Modeling/RobotPoseHeatMap.h"
#include "Representations/Configuration/FieldDimensions.h"
#include "Representations/Perception/CameraMatrix.h"
#include "Representations/Infrastructure/CameraInfo.h"
#include "Representations/Perception/ImageCoordinateSystem.h"
#include "Representations/Modeling/BallHeatMap.h"
#include "Representations/Infrastructure/FrameInfo.h"


#include "Modules/Modeling/ERPHeatMapper/HMERParameter.h"
#include "Tools/Debugging/Debugging.h"
#include "Tools/Math/Vector2.h"
#include "Tools/Math/Vector.h"
#include "Tools/Math/Matrix.h"
#include "Tools/Math/Matrix2x2.h"
#include "Tools/Math/Common.h"
#include "Tools/Math/Geometry.h"

MODULE(ERPHeatMapper)
  REQUIRES(BallPercept)
  REQUIRES(RobotPoseHeatMap)
  REQUIRES(RobotPercept)
//   REQUIRES(OdometryData)
  REQUIRES(FrameInfo)
  REQUIRES(FieldDimensions)
  REQUIRES(CameraMatrix)
  REQUIRES(CameraInfo)
  REQUIRES(ImageCoordinateSystem)
//   REQUIRES(TorsoMatrix)
//   REQUIRES(RobotModel)
//   REQUIRES(RobotDimensions)
//   REQUIRES(MotionInfo)
//   REQUIRES(TeamMateData)
 PROVIDES_WITH_MODIFY_AND_DRAW(ERPHeatMap)
END_MODULE

```

```
class ERPHeatMapper : public ERPHeatMapperBase
{
private:

 // BallHeatMap m_ballHeatMap;
 HMERParameter* parameter;
 Matrix2x2f roboPerceptCov;


 void update(ERPHeatMap& erpHeatMap);
 void init(ERPHeatMap& erpHeatMap);
 Matrix2x2f getCovOfPixelInWorld(const Vector2<>& correctedPointInImage, float
     pointZInWorld) const;
 float getUnscaledProbabilityAt(const Vector2f& mean, const Matrix2x2f& cov,
    const Vector2f& pos) const;
 Vector2f getPossibleGlobalRoboPos(Vector2<int> roboPos,Vector2<> relBallPos,
    Vector2f xView,Vector2f yView);
 void updateHeatMap(ERPHeatMap& erpHeatMap);
 bool isNotInRange(ERPHeatMap& bhm, Vector2f pos);
 Vector2<> getPointInWorld(const Vector2<>& correctedPointInImage);

 float roboPositionThreshold;
 bool initdone;

public:
  /**
  * Constructor
  */

 ERPHeatMapper();

  /**
  * Destructor
  */
  ~ERPHeatMapper();
};


//Creates the module thus allowing it to be instantiated
//the second parameter defines the module as part of the category Modeling
```

Listing A.2: ERPHeatMepper.h

```
#include "HMERParameter.h"
#include "Platform/BHAssert.h"
#include "Tools/Streams/InStreams.h"
#include "Tools/Global.h"

#include "Tools/Math/Vector2.h"

HMERParameter::HMERParameter()
{
```

```
  load("HMenemies.cfg");
}

void HMERParameter::load(const std::string& fileName)
{
  InConfigMap file(Global::getSettings().expandLocationFilename(fileName));
  ASSERT(file.exists());
  file >> *this;
}
```

Listing A.3: HMERParameter.cpp

```

#pragma once

#include "Tools/Math/Pose2D.h"
#include "Tools/Settings.h"
#include "Tools/Range.h"
#include "Tools/Math/Vector2.h"
#include "Tools/Debugging/Debugging.h"

/**
* A collection of all parameters of the module.
*/
class HMERParameter : public Streamable
{
private:
  /**
  * The method makes the object streamable.
  * @param in The stream from which the object is read
  * @param out The stream to which the object is written
  */
  virtual void serialize(In* in, Out* out)
  {
    STREAM_REGISTER_BEGIN();
    STREAM(robotRotationDeviation);
    STREAM(maxProbThreshold);
    STREAM(stepSizeDraw);
    STREAM(xWidth);
    STREAM(yWidth);
    STREAM(NumOfRobots);
    STREAM(enemyteamRed);
    STREAM_REGISTER_FINISH();
  }

public:
  HMERParameter();
  void load(const std::string& fileName);
  Vector2<> robotRotationDeviation;
  int stepSizeDraw;
  int xWidth; // mm
  int yWidth; //
```

```
  float maxProbThreshold;
  int NumOfRobots;
  bool enemyteamRed;
  };
```

Listing A.4: HMERParameter.h

```
/**
 * @file BallHeatMapper.cpp
 * Creates the HeatMap for the Ball.
 * @author Hans Hardmeier
 * @date 24/04/13
 * Copyright (c) 2013 __ETHZ__. All rights reserved.
 */

#include "BallHeatMapper.h"  //Include Header


BallHeatMapper::BallHeatMapper():
parameter(new HMBallParameter)
{
    initdone=false;
}

// Destructor
BallHeatMapper::~BallHeatMapper()
{
  delete parameter;
}

void BallHeatMapper::init(BallHeatMap& ballHeatMap)
{
 //OUTPUT ( idText , text ,"INIT BALLHEATMAP" );
    ballHeatMap.resize(parameter->xWidth, parameter->yWidth);
    initdone=true;
}


/**
Get Covariance Matrix of a Pixel in the World using global parameters
*/
 Matrix2x2f BallHeatMapper::getCovOfPixelInWorld(const Vector2<>&
    correctedPointInImage, float pointZInWorld) const
{
// From B-Human BallLocator
//Problems: Independet of Movement or State

  const Vector3<> unscaledVectorToPoint(theCameraInfo.focalLength,
      theCameraInfo.opticalCenter.x - correctedPointInImage.x, theCameraInfo.
      opticalCenter.y - correctedPointInImage.y);
  const Vector3<> unscaledWorld = theCameraMatrix.rotation *
      unscaledVectorToPoint;
```

```
  const float h = theCameraMatrix.translation.z - pointZInWorld;
  const float scale = h / -unscaledWorld.z;
  Vector2<> pointInWorld(unscaledWorld.x * scale, unscaledWorld.y * scale);
  const float distance = pointInWorld.abs();
  const float angle = pointInWorld.angle();
  const float c = cos(angle), s = sin(angle);
  Matrix2x2f rot(Vector2f(c, s), Vector2f(-s, c));
  Matrix2x2f cov(Vector2f(sqr(h / tan((distance == 0.f ? pi_2 : atan(h / distance)) - parameter->robotRotationDeviation.x) - distance), 0.f),
                 Vector2f(0.f, sqr(tan(parameter->robotRotationDeviation.y) * distance)));
  return rot * cov * rot.transpose();
}


/*
Vector2f BallHeatMapper::getPossibleGlobalBallPos(Vector2<int> roboPos, Vector2<> relBallPos,
                                                  Vector2f xView, Vector2f yView){
    Vector2f result = (xView*relBallPos.x)+(yView*relBallPos.y);
    result.x+=roboPos.x;
    result.y+=roboPos.y;
    return result;
}

*/
/**
Calculate the Probability (using a Covariance Distribution) of pos relative to mean
*/
float BallHeatMapper::getUnscaledProbabilityAt(const Vector2f& mean, const Matrix2x2f& cov, const Vector2f& pos) const
{
  Vector2f diff = pos - mean;
  float exponent = diff * (cov.invert() * diff);
  float p = exp(-0.5f * exponent);
  return max(p, 0.01f /*0.0000001f*/);
}

void BallHeatMapper::update(BallHeatMap& ballHeatMap)
{
  if(!initdone)
  {
    init(ballHeatMap);
  }

  MODIFY("module:BallHeatMapper:parameter", *parameter);

  DEBUG_RESPONSE_ONCE("BallHeatMap:maxProbIndex", OUTPUT(idText, text, "Ball Position: " << ballHeatMap.maxProbIndex[0]<<" "<<ballHeatMap.
```

```
      maxProbIndex[1]););

   // reset weights
   ballHeatMap.reset();

   if(theBallPercept.ballWasSeen && theRobotPoseHeatMap.updated){

    // calculate variance of the percept
   ballPerceptCov = getCovOfPixelInWorld(
                         theImageCoordinateSystem.toCorrected(theBallPercept.
                             positionInImage),
                         float(theFieldDimensions.ballRadius));

   ballHeatMap.stepDraw = parameter->stepSizeDraw;
   ballHeatMap.maxProb = 0.0f;
   ballHeatMap.updated = false;

   updateHeatMap(ballHeatMap);
   }else{
     // Do nothing
   }

    DEBUG_RESPONSE_ONCE("BallHeatMapPos", OUTPUT(idText, text, "Ball Position:
        " << ballHeatMap.relBallPos.x<<" "<<ballHeatMap.relBallPos.y););


}

void BallHeatMapper::updateHeatMap(BallHeatMap& ballHeatMap)
{
   ballHeatMap.relBallPos=theBallPercept.relativePositionOnField;
   roboPositionThreshold = theRobotPoseHeatMap.maxProb - parameter->
      maxProbThreshold;
   if(roboPositionThreshold<0)
   {
     roboPositionThreshold=0;
   }

   for(int rot = 0; rot < theRobotPoseHeatMap.maxIndex[0]; ++rot)
         {
           // Calculate relative Coordinate System
           float viewAngle = 2*pi*((float)rot/(float)theRobotPoseHeatMap.maxIndex
               [0]);
           Vector2f xView(cos(viewAngle),sin(viewAngle));
           Vector2f yView(-xView.y,xView.x);

           for(int x = 0; x < theRobotPoseHeatMap.maxIndex[1]; ++x)
           {
             for(int y = 0; y < theRobotPoseHeatMap.maxIndex[2]; ++y)
             {
               long index=theRobotPoseHeatMap.getIndex(rot,x,y);
```

```
            if(theRobotPoseHeatMap.heatMap[index]>=roboPositionThreshold){

                //Global Robot Position in question:
                Vector2<int> roboPosition = ballHeatMap.getPos(x,y);

                //Possible Global Ball Position
                Vector2f ballPosition =ballHeatMap.getPossibleGlobalBallPos(
                    roboPosition,ballHeatMap.relBallPos,xView,yView);
                if(ballHeatMap.isNotInRange(ballPosition))
                     continue; //Continue if ballPosition out ouf Range

                //Calculate start of Covariance Field
                float radius=max(sqrt(ballPerceptCov.c[0][0]),sqrt(
                    ballPerceptCov.c[1][1]));
                int cellSize=min(ballHeatMap.cell[0],ballHeatMap.cell[1]);
                int radiusCell=radius/cellSize;
                Vector2<int> ballCoords = ballHeatMap.getCoords((int)
                    ballPosition.x,(int)ballPosition.y);
                Vector2<int> start(max(ballCoords.x-radiusCell,0),max(0,
                    ballCoords.y-radiusCell));
                Vector2<int> end(min(ballCoords.x+radiusCell,ballHeatMap.
                    maxIndex[0]),
                                 min(ballCoords.y+radiusCell,ballHeatMap.
                                     maxIndex[1]));
                //Start updateLocalHeatMap
                for(int covX=start.x;covX<=end.x;covX++)
                {
                    for(int covY=start.y;covY<=end.y;covY++)
                     {
                          Vector2<int> temp = ballHeatMap.getPos(covX,covY);
                          Vector2f neighbour((float)temp.x,(float)temp.y);
                          float prob=getUnscaledProbabilityAt(ballPosition,
                              ballPerceptCov,neighbour);
                          long indexBHM = ballHeatMap.getIndex(covX,covY);
                          if(ballHeatMap.heatMap[indexBHM]<(theRobotPoseHeatMap.
                              heatMap[index]*prob))
                          {
                               ballHeatMap.heatMap[indexBHM]=theRobotPoseHeatMap.
                                   heatMap[index]*prob;
                               if(ballHeatMap.maxProb<theRobotPoseHeatMap.heatMap
                                   [index]*prob)
                               {
                                   ballHeatMap.maxProb= theRobotPoseHeatMap.
                                       heatMap[index]*prob;
                                    ballHeatMap.maxProbIndex[0]=covX;
                                    ballHeatMap.maxProbIndex[1]=covY;
                               }

                          }
                     }
                }
```

```
                }
            }
        }
    }
//  ballHeatMap.maxProb=theRobotPoseHeatMap.maxProb;
    ballHeatMap.updated=true;



}

MAKE_MODULE(BallHeatMapper, Modeling)
```

Listing A.5: BallHeatMapper.cpp

```
/**
 * @file BallHeatMapper.h
 * Creates the HeatMap for the Ball.
 * @author Hans Hardmeier
 * @date 24/04/13
 * Copyright (c) 2013 __ETHZ__. All rights reserved.
 */

#pragma once        //Makes sure that the source file is only included once in
    the compilation
#include "Tools/Module/Module.h"

//#include "Representations/Modeling/ERPHeatMap.h"

#include "Representations/Perception/BallPercept.h"
#include "Representations/Modeling/RobotPoseHeatMap.h"
#include "Representations/Configuration/FieldDimensions.h"
#include "Representations/Perception/CameraMatrix.h"
#include "Representations/Infrastructure/CameraInfo.h"
#include "Representations/Perception/ImageCoordinateSystem.h"
#include "Representations/Modeling/BallHeatMap.h"




#include "Modules/Modeling/BallHeatMapper/HMBallParameter.h"
#include "Tools/Debugging/Debugging.h"
#include "Tools/Math/Vector2.h"
#include "Tools/Math/Vector.h"
#include "Tools/Math/Matrix.h"
#include "Tools/Math/Common.h"

MODULE(BallHeatMapper)
  REQUIRES(BallPercept)
  REQUIRES(RobotPoseHeatMap)
//  REQUIRES(OdometryData)
//  REQUIRES(FrameInfo)
  REQUIRES(FieldDimensions)
```

```
  REQUIRES(CameraMatrix)
  REQUIRES(CameraInfo)
  REQUIRES(ImageCoordinateSystem)
//   REQUIRES(TorsoMatrix)
//   REQUIRES(RobotModel)
//   REQUIRES(RobotDimensions)
//   REQUIRES(MotionInfo)
//   REQUIRES(TeamMateData)
 PROVIDES_WITH_MODIFY_AND_DRAW(BallHeatMap)
END_MODULE

class BallHeatMapper : public BallHeatMapperBase
{
private:

 // BallHeatMap m_ballHeatMap;
 HMBallParameter* parameter;
 Matrix2x2f ballPerceptCov;


 void update(BallHeatMap& ballHeatMap);
 void init(BallHeatMap& ballHeatMap);
 Matrix2x2f getCovOfPixelInWorld(const Vector2<>& correctedPointInImage, float
      pointZInWorld) const;
 float getUnscaledProbabilityAt(const Vector2f& mean, const Matrix2x2f& cov,
     const Vector2f& pos) const;
 Vector2f getPossibleGlobalBallPos(Vector2<int> roboPos,Vector2<> relBallPos,
     Vector2f xView,Vector2f yView);
 void updateHeatMap(BallHeatMap& ballHeatMap);
 bool isNotInRange(BallHeatMap& bhm, Vector2f pos);

 float roboPositionThreshold;
 bool initdone;

public:
  /**
  * Constructor
  */

 BallHeatMapper();

  /**
  * Destructor
  */
  ~BallHeatMapper();
};


//Creates the module thus allowing it to be instantiated
//the second parameter defines the module as part of the category Modeling
```

Listing A.6: BallHeatMapper.h

```
#include "HMBallParameter.h"
#include "Platform/BHAssert.h"
#include "Tools/Streams/InStreams.h"
#include "Tools/Global.h"

#include "Tools/Math/Vector2.h"

HMBallParameter::HMBallParameter()
{
  load("HMball.cfg");
}

void HMBallParameter::load(const std::string& fileName)
{
  InConfigMap file(Global::getSettings().expandLocationFilename(fileName));
  ASSERT(file.exists());
  file >> *this;
}
```

Listing A.7: HBBallParameter.cpp

```

#pragma once

#include "Tools/Math/Pose2D.h"
#include "Tools/Settings.h"
#include "Tools/Range.h"
#include "Tools/Math/Vector2.h"
#include "Tools/Debugging/Debugging.h"

/**
* A collection of all parameters of the module.
*/
class HMBallParameter : public Streamable
{
private:
  /**
  * The method makes the object streamable.
  * @param in The stream from which the object is read
  * @param out The stream to which the object is written
  */
  virtual void serialize(In* in, Out* out)
  {
    STREAM_REGISTER_BEGIN();
    STREAM(robotRotationDeviation);
    STREAM(maxProbThreshold);
    STREAM(stepSizeDraw);
    STREAM(xWidth);
    STREAM(yWidth);
    STREAM_REGISTER_FINISH();
  }
```

```
public:
  HMBallParameter();
  void load(const std::string& fileName);
  Vector2<> robotRotationDeviation;
  int stepSizeDraw;
  int xWidth; // mm
  int yWidth; //
  float maxProbThreshold;
  };
```

Listing A.8: HMBallParameter.h

```
/**
 * @file BallHeatMap.h
 * B-Human //
 * @author Hans Hardmeier
 * @date 24/04/13
 * Copyright (c) 2013 __ETHZ__. All rights reserved.
 */



#pragma once

#include "Tools/Math/Vector.h"
#include "Tools/Math/Vector2.h"
#include "Tools/Streams/Streamable.h"
#include "Tools/Debugging/DebugDrawings.h"
#include "Representations/Modeling/RobotPoseHeatMap.h"
#include <vector>
#include <limits.h>

#define FIELD_Y 4000
#define FIELD_X 6000
#define CARPET_Y 5400
#define CARPET_X 7400
#define CELL_X 40
#define CELL_Y 40

class BallCandidate : public Streamable
{
public:
  Vector2<int> position;//Absolute Postion in field
  float probability;

  BallCandidate()
  {
    probability = 0.0f;
  }
private:
  /**
```

```
    * The method makes the object streamable.
    * @param in The stream from which the object is read.
    * @param out The stream to which the object is written.
    */
    virtual void serialize(In* in, Out* out)
    {
      STREAM_REGISTER_BEGIN();
      STREAM(position);
      STREAM(probability);
      STREAM_REGISTER_FINISH();
    }
};



class BallHeatMap : public Streamable
{

private:
  void serialize(In* in,Out* out)
  {
    STREAM_REGISTER_BEGIN();
    STREAM(carpet[0]);
    STREAM(carpet[1]);
    STREAM(field[0]);
    STREAM(field[1]);
    STREAM(cell[0]);
    STREAM(cell[1]);
    STREAM(maxIndex[0]);
    STREAM(maxIndex[1]);
    for (int x = 0; x < maxIndex[0]; ++x)
      for (int y = 0; y < maxIndex[1]; ++y)
        STREAM(heatMap[x*maxIndex[1]+y]);
    STREAM(stepDraw);
    STREAM_REGISTER_FINISH();
  }
  //All the serialize function does, is to open a stream, load the variable
      counterS into it and close it again.

public:
  int carpet[2];        /**< Carpet dimension including area around soccer field
      . units: mm */
  int field[2];
  int cell[2];          /**< Cell dimension */
  int maxIndex[2]; /**< Maximum Index of the Probability Field (0->x &< 1->y)
      */
  std::vector<float> heatMap; /**< 2Dimensional probability map (x, y) */
  float maxProb; /** Maximum probability in the heatMap */
  int maxProbIndex[2]; /**< Most likely position of the ball (index in heatMap
      ) */
  bool updated;
```

```
int stepDraw;

Vector2<> relBallPos; /** rel Ball Position in mm to Ball*/


BallHeatMap() :
heatMap(CARPET_X/CELL_X * CARPET_Y/CELL_Y)
{
  carpet[0] = CARPET_X;
  carpet[1] = CARPET_Y;

  field[0]=FIELD_X;
  field[1]=FIELD_Y;

  cell[0] = CELL_X;
  cell[1] = CELL_Y;

  maxIndex[0] = CARPET_X/CELL_X;
  maxIndex[1] = CARPET_Y/CELL_Y;

  stepDraw = 3;

 // initialize heatMap
  for (int x = 0; x < maxIndex[0]; ++x)
  {
    int idx_x = x*maxIndex[1];
    for (int y = 0; y < maxIndex[1]; ++y)
    {
        int index = idx_x+y;
        heatMap[index] = 0.0f;
    }
  }

  maxProb = 0.0f;

  // init "Location" of maxProb
  maxProbIndex[0] = maxIndex[0] / 2; // x
  maxProbIndex[1] = maxIndex[1] / 2; // y
  updated = false;

} // end constructor


/** Returns the field coordinates af a cell in mm
* @param x x-index of heatMap
* @param y y-index of heatMap
* @return Vector2<int> with field coordinates in mm
*/
inline Vector2<int> getPos(int x, int y) const
{
```

```
    return Vector2<int>(x * cell[0] - (carpet[0] - cell[0]) / 2, y * cell[1] -
        (carpet[1] - cell[1]) / 2);
  }

  /** Returns the index of the 1D-Array
  * @param rotIndex rotational Index [0..15]
  */
  inline long getIndex(int xIndex, int yIndex) const
  {
    return xIndex*maxIndex[1]+yIndex;
  }

  /** Draws the robot pose to the field view*/
  void draw();
  /** Resize the Map*/
  void resize(int xWidth, int yWidth);
  /** Reset Map- All entries 0.f*/
  void reset();

  /** Given a Robo Pos give the Candidate for Ball Position*/
  BallCandidate getCandidate(Vector2<int> roboPos, float viewAngle) const;

  /** Calculates if Position in field*/
  bool isNotInRange(Vector2f pos) const;

  /**
 Given the RoboPosition and rel Ball Position, calculate global Pos Position
 @param relBallPos Relative Ball Position to the Robot's View Position
 @param xView x-axis direction relation of View direction;
 @return Global Pos of Ball in mm
 */

  Vector2f getPossibleGlobalBallPos(Vector2<int> roboPos,Vector2<> relBallPos,
                                    Vector2f xView,Vector2f yView)
                                    const;

  /** Transform real coordinates into x,y of heatmap*/
  inline Vector2<int> getCoords(int x, int y) const
  {
       return Vector2<int>((x+(carpet[0] - cell[0]) / 2)/cell[0], (y+(carpet
          [1] - cell[1]) / 2)/cell[1]);
  }

  inline int getDistance()
  {
    return abs(sqrt(relBallPos.x*relBallPos.x+relBallPos.y*relBallPos.y));
  }



};
```

Listing A.9: BallHeatMap.h

```
/**
* @file BallHeatMap.cpp
*
* contains the implementation of BallHeatMap
*/

#include "BallHeatMap.h"

void BallHeatMap::draw()
{
    //updateFlatList();

    DECLARE_DEBUG_DRAWING("representation:BallHeatMap", "drawingOnField");

    COMPLEX_DRAWING("representation:BallHeatMap",
    {
        unsigned short r, g, b;
        ColorRGBA color;

        int tempXY;
        float prob;
        for (int x = 0; x < maxIndex[0]; x+=stepDraw)
        {
            for (int y = 0; y < maxIndex[1]; y+=stepDraw)
            {

                tempXY = maxIndex[1] * x + y;
                prob =heatMap[tempXY];
                //prob = flatList[tempXY];


                // calculate center of cell in mm
                float xField = x * cell[0] - (carpet[0] - cell[0]) / 2;
                float yField = y * cell[1] - (carpet[1] - cell[1]) / 2;

                // draw a dot in the field
                if (maxProb != 0.0f && updated)
                {
                    r = 255 * prob / maxProb;
                    b = 64 - prob * 64 / maxProb;

                    if (r > 255)
                        r = 255;
                    else if (r < 0)
                        r = 0;

                    if (b > 64)
                        b = 64;
```

```
                    else if (b < 0)
                        b = 0;

                    g = r;
                }
                else
                {


                    r = 255;
                    g = 0;
                    b = 0;
                }

                color.r = r;
                color.g = g;
                color.b = b;

                //if(index_prob > 1)
                LARGE_DOT("representation:BallHeatMap", xField, yField, color,
                    color);
            }
        }
    }); // END OF COMPLEX_DRAWING

}

void BallHeatMap::reset()
{
    for(int x = 0; x < maxIndex[0]; ++x)
        for(int y = 0; y < maxIndex[1]; ++y)
            heatMap[getIndex(x, y)] = 0.0f;

    maxProb=0.0f;
}

void BallHeatMap::resize(int xWidth, int yWidth)
{
    cell[0] = xWidth;
    cell[1] = yWidth;
    maxIndex[0] = carpet[0]/xWidth;
    maxIndex[1] = carpet[1]/yWidth;

    heatMap.resize(maxIndex[1] * maxIndex[0]);

    //DEBUG_RESPONSE("representation:RobotPoseHeatMap:resize",
// OUTPUT(idText, text, "Resizing BallHeatMap: ");
// OUTPUT(idText, text, "cell_x: " << cell[0]);
// OUTPUT(idText, text, "cell_y: " << cell[1]);
// OUTPUT(idText, text, "max Index x: " << maxIndex[0]);
// OUTPUT(idText, text, "max Index y: " << maxIndex[1]);
```

```
    //);
}

BallCandidate BallHeatMap::getCandidate(Vector2<int> roboPos, float viewAngle)
    const
{
    BallCandidate cand;

    Vector2f xView(cos(viewAngle),sin(viewAngle));
    Vector2f yView(-xView.y,xView.x);

    Vector2f temp = getPossibleGlobalBallPos(Vector2<int>(roboPos.x,roboPos.y)
        ,relBallPos,xView,yView);
    if(isNotInRange(temp))
    {
        //BallCandidate outside of Field
        //Not possible if Ball was seen
        cand.position= Vector2<int> (temp.x,temp.y);
        cand.probability=-10000.f;
    }
    else
    {
        cand.position= Vector2<int> (temp.x,temp.y);
        Vector2<int> coords = getCoords(temp.x,temp.y);
        cand.probability=heatMap[getIndex(coords.x,coords.y)];
    }
    return cand;
}


Vector2f BallHeatMap::getPossibleGlobalBallPos(Vector2<int> roboPos,Vector2<>
    relBallPos,
        Vector2f xView,Vector2f yView) const
{
    Vector2f result = (xView*relBallPos.x)+(yView*relBallPos.y);
    result.x+=roboPos.x;
    result.y+=roboPos.y;
    return result;
}

bool BallHeatMap::isNotInRange(Vector2f pos) const
{
    return (pos.x<-carpet[0]/2 || pos.y<-carpet[1]/2 || pos.x>carpet[0]/2 ||
        pos.y >carpet[1]/2);
}
```

Listing A.10: BallHeatMap.cpp

```
/**
 * @file ERPHeatMap.h
 * B-Human //
 * @author Hans Hardmeier
```

```
 * @date 24/04/13
 * Copyright (c) 2013 __ETHZ__. All rights reserved.
 */



#pragma once

#include "Tools/Math/Vector.h"
#include "Tools/Math/Vector2.h"
#include "Tools/Streams/Streamable.h"
#include "Tools/Debugging/DebugDrawings.h"
#include "Representations/Modeling/RobotsModel.h"
#include <vector>
#include <limits.h>

#define FIELD_Y 4000
#define FIELD_X 6000
#define CARPET_Y 5400
#define CARPET_X 7400
#define CELL_X 40
#define CELL_Y 40
#define ROBOTS 8

class ERPHeatMap : public Streamable
{

private:
  void serialize(In* in,Out* out)
  {
    STREAM_REGISTER_BEGIN();
    STREAM(carpet[0]);
    STREAM(carpet[1]);
    STREAM(cell[0]);
    STREAM(cell[1]);
    STREAM(maxIndex[0]);
    STREAM(maxIndex[1]);
    STREAM(maxIndex[2]);
    for (int rot = 0; rot < maxIndex[0]; ++rot)
      for (int x = 0; x < maxIndex[1]; ++x)
        for (int y = 0; y < maxIndex[2]; ++y)
          STREAM(heatMap[(rot * maxIndex[1] + x) * maxIndex[2] + y]);
    STREAM(stepDraw);
    STREAM_REGISTER_FINISH();
  }
  //All the serialize function does, is to open a stream, load the variable counterS into it and close it again.

public:
  int carpet[2];      /**< Carpet dimension including area around soccer field. units: mm */
```

```
  int cell[2];        /**< Cell dimension */
  int maxIndex[3]; /**< Maximum Index of the Probability Field (0->robots 1->x
      & 2->y)*/
  std::vector<float> heatMap; /**< 3Dimensional probability map (x, y) */
  float maxProb; /** Maximum probability in the heatMap */
 // int maxProbIndex[2]; /**< Most likely position of the robot (index in
    heatMap) */
  bool updated;
  int stepDraw;
  int robotsSeen;

/**Flat list stuff*/

bool flatListIsUpdated;
std::vector<float> flatList; /**< List with a flattened candidate list (for
    each pos in field 1 candidate) */


/** Stuff for RobotsModel*/

std::vector<RobotsModel::Robot> robots;

  ERPHeatMap() :
  heatMap(ROBOTS * CARPET_X/CELL_X * CARPET_Y/CELL_Y), flatList(CARPET_X/
      CELL_X * CARPET_Y/CELL_Y),robots(ROBOTS)
  {
    carpet[0] = CARPET_X;
    carpet[1] = CARPET_Y;

    cell[0] = CELL_X;
    cell[1] = CELL_Y;

    maxIndex[0] = ROBOTS;
    maxIndex[1] = CARPET_X/CELL_X;
    maxIndex[2] = CARPET_Y/CELL_Y;

    stepDraw = 3;
    robotsSeen = 0;

   // initialize heatMap
    for (int rot = 0; rot < maxIndex[0]; ++rot)
      for (int x = 0; x < maxIndex[1]; ++x)
        for (int y = 0; y < maxIndex[2]; ++y)
        {
          int index = (rot * maxIndex[1] + x) * maxIndex[2] + y;
          heatMap[index] = 0.0f;
        }

    maxProb = 0.0f;

    // init "Location" of maxProb
```

```
      // init "Location" of maxProb
//     maxProbIndex[0] = 0; // rotation
  //   maxProbIndex[1] = maxIndex[1] / 2; // x
    //maxProbIndex[2] = maxIndex[2] / 2; // y
    updated = false;

  } // end constructor


  /** Returns the field coordinates af a cell in mm
  * @param x x-index of heatMap
  * @param y y-index of heatMap
  * @return Vector2<int> with field coordinates in mm
  */
  inline Vector2<int> getPos(int x, int y) const
  {
    return Vector2<int>(x * cell[0] - (carpet[0] - cell[0]) / 2, y * cell[1] -
        (carpet[1] - cell[1]) / 2);
  }

  /** Returns the index of the 1D-Array
  * @param rotIndex rotational Index [0..15]
  */
  inline long getIndex(int rotIndex, int xIndex, int yIndex) const
  {
    return (rotIndex * maxIndex[1] + xIndex) * maxIndex[2] + yIndex;
  }

  /** Draws the robot pose to the field view*/
  void draw();
  /** Resize the Map*/
  void resize(int rot,int xWidth, int yWidth);
  /** Reset Map- All entries 0.f*/
  void reset();
  /** Flat list of Enemy Robots*/
  void updateFlatList();
  /** Transform real coordinates into x,y of heatmap*/
  inline Vector2<int> getCoords(int x, int y) const
  {
      return Vector2<int>((x+(carpet[0] - cell[0]) / 2)/cell[0], (y+(carpet
          [1] - cell[1]) / 2)/cell[1]);
  }
  /** Given a Robo Position and rotation, return RoboCandidates according
     HeatMap*/
  std::vector<RobotsModel::RobotCandidate> getCandidates(Vector2<int> roboPos,
      float viewAngle) const;

  /** Is pos out of field?*/
  bool isNotInRange(Vector2f pos) const;

  /**Calculate global position*/
```

```
    Vector2f getPossibleGlobalPos(Vector2<int> roboPos, Vector2<> relPos,
            Vector2f xView, Vector2f yView) const;

};
```

Listing A.11: ERPHeatMap.h

```
/**
* @file ERPHeatMap.cpp
*
* contains the implementation of BallHeatMap
*/

#include "ERPHeatMap.h"

std::vector<RobotsModel::RobotCandidate> ERPHeatMap::getCandidates(Vector2<int
    > roboPos, float viewAngle) const
{
    RobotsModel::RobotCandidate cand;
    std::vector<RobotsModel::RobotCandidate> result;

    Vector2f xView(cos(viewAngle),sin(viewAngle));
    Vector2f yView(-xView.y,xView.x);

    for(int index=0; index<robotsSeen; index++)
    {
        //To implement
        Vector2f temp = getPossibleGlobalPos(Vector2<int>(roboPos.x,roboPos.y)
            ,robots[index].relPosOnField,xView,yView);
        if(isNotInRange(temp)) //TODO Iplemente
        {
            //RobotCandidate outside of Field
            continue;

        }
        else
        {
            cand.position= Vector2<int> (temp.x,temp.y);
            Vector2<int> coords = getCoords(temp.x,temp.y);
            cand.probability=heatMap[getIndex(index,coords.x,coords.y)];
            cand.odometry = Vector2<int> (temp.x,temp.y);
            cand.relPosOnField = robots[index].relPosOnField;
            cand.teamRed=robots[index].teamRed;
            cand.standing=robots[index].standing;
            cand.covariance=robots[index].covariance;
            cand.timeStamp=robots[index].timeStamp;
        }
        result.push_back(cand);

    }

```

```
    return result;
}



void ERPHeatMap::draw()
{

    updateFlatList();

    DECLARE_DEBUG_DRAWING("representation:ERPHeatMap", "drawingOnField");

    COMPLEX_DRAWING("representation:ERPHeatMap",
    {
        unsigned short r, g, b;
        ColorRGBA color;

        int tempXY;
        float prob;
        for (int x = 0; x < maxIndex[1]; x+=stepDraw)
        {
            for (int y = 0; y < maxIndex[2]; y+=stepDraw)
            {

                tempXY = maxIndex[2] * x + y;
                //prob = flatList[tempXY];
                prob=heatMap[getIndex(0,x,y)];
                // calculate center of cell in mm
                float xField = x * cell[0] - (carpet[0] - cell[0]) / 2;
                float yField = y * cell[1] - (carpet[1] - cell[1]) / 2;

                // draw a dot in the field
                if (maxProb != 0.0f && updated)
                {
                    r = 255 * prob / maxProb;
                    b = 64 - prob * 64 / maxProb;

                    if (r > 255)
                        r = 255;
                    else if (r < 0)
                        r = 0;

                    if (b > 64)
                        b = 64;
                    else if (b < 0)
                        b = 0;

                    g = r;
                }
                else
                {
```

```
                    r = 255;
                    g = 0;
                    b = 0;
                }

                color.r = r;
                color.g = g;
                color.b = b;

                //if(index_prob > 1)
                LARGE_DOT("representation:ERPHeatMap", xField, yField, color,
                    color);
            }
        }
    }); // END OF COMPLEX_DRAWING

}


void ERPHeatMap::reset()
{
    for(int rob = 0; rob < maxIndex[0]; ++rob)
        for(int x = 0; x < maxIndex[1]; ++x)
            for(int y = 0; y < maxIndex[2]; ++y)
                heatMap[getIndex(rob,x, y)] = 0.0f;

    maxProb=0.0f;
}


void ERPHeatMap::resize(int robots, int xWidth, int yWidth)
{
    cell[0] = xWidth;
    cell[1] = yWidth;
    maxIndex[0] = robots;
    maxIndex[1] = carpet[0]/xWidth;
    maxIndex[2] = carpet[1]/yWidth;

    heatMap.resize(robots * maxIndex[1] * maxIndex[2]);
    flatList.resize( maxIndex[1] * maxIndex[2]);

}



void ERPHeatMap::updateFlatList()
{

    float tempCandidate;
    int maxXY = maxIndex[1] * maxIndex[2];
    int temp;
```

```
    int tempXY;

    // make sure the list is updated before it is drawn, but not updated if it
        is already up-to-date
    if(!flatListIsUpdated)
    {
        if(flatList.size() != (unsigned int)(maxXY))
            flatList.resize(maxXY);

        for (int x = 0; x < maxIndex[1]; ++x)
        {
            //  tempCandidate.x = x * cell[0] - (carpet[0] - cell[0]) / 2;
            for (int y = 0; y < maxIndex[2]; ++y)
            {
                //   tempCandidate.y = y * cell[1] - (carpet[1] - cell[1]) /
                    2;
                float prob = 0.0f;

                tempXY = maxIndex[2] * x + y;
                for (int robot = 0; robot < maxIndex[0]; ++robot)
                {
                    temp = maxXY *robot + tempXY;
                    if(heatMap[temp] > prob)
                    {
                        prob = heatMap[temp];
                        //  tempCandidate.rotation = normalize((float)rot /
                            maxIndex[0] * pi2);
                        tempCandidate = prob;
                    }
                }
                flatList[tempXY] = tempCandidate;
            }
        }
    }
    flatListIsUpdated = true;


}


bool ERPHeatMap::isNotInRange(Vector2f pos) const
{
    return (pos.x<-carpet[0]/2 || pos.y<-carpet[1]/2 || pos.x>carpet[0]/2 ||
        pos.y >carpet[1]/2);
}


Vector2f ERPHeatMap::getPossibleGlobalPos(Vector2<int> roboPos,Vector2<>
    relPos,
        Vector2f xView,Vector2f yView) const
{
```

```
    Vector2f result = (xView*relPos.x)+(yView*relPos.y);
    result.x+=roboPos.x;
    result.y+=roboPos.y;
    return result;
}
```

Listing A.12: ERPHeatMap.cpp

```
/**
 * @file StateCRFResult.h
 * This is the Result of the CRF
 * @author Hans Hardmeier
 * @date 19/06/13
 * Copyright (c) 2013 __ETHZ__. All rights reserved.
 */

//For some reason bhuman likes to define whole classes in the header files:
    this is done here too.

#pragma once
#include "Representations/Modeling/RobotPose.h"
#include "Representations/Modeling/BallModel.h"
#include "Representations/Modeling/RobotsModel.h"


class StateCRFResult : public Streamable {

private:
  void serialize(In* in,Out* out)
  {
    STREAM_REGISTER_BEGIN();
    STREAM(robotPose);
    STREAM(ballModel);
    STREAM(robotPoseInfo);
    STREAM(robotsModel);
    STREAM(unaryPot);
    STREAM(smoothPot);
    STREAM_REGISTER_FINISH();
  }
public:
  RobotPose robotPose;
  BallModel ballModel;
  RobotPoseInfo robotPoseInfo;
  RobotsModel robotsModel;
  float unaryPot;
  float smoothPot;

  StateCRFResult():
  robotPose(),
  ballModel(),
  robotPoseInfo(),
  robotsModel()
```

```
  {
  }
};
```

Listing A.13: StateCRFResult.h

```
/**
* @file StateCRF.cpp
* The Implementation of the module that calculates the state based on the
   heatMap.
* @author Hans Hardmeier, Martin Utz
*/

#include <limits>
#include <iostream>
#include <iterator>
#include "StateCRF.h"
#include "Tools/Team.h"

using namespace std; //vector<>

MAKE_MODULE(StateCRF, Modeling)

// Constructor
StateCRF::StateCRF() : parameter(new StateCRFParameter)
{
    /*
    CRFData dummy;
    for(int i = 0; i < bufferSize; ++i)
    {
      data.add(dummy);
    }
    */
    /*
    sumUnaryPot = 0.0f;
    sumXPot = bufferSize * theRobotPoseHeatMap.carpet[0];
    sumYPot = bufferSize * theRobotPoseHeatMap.carpet[1];
    sumRotPot = bufferSize * pi;
    */

    bufferSize = BUFFERSIZE;
    numberOfIterations = 60;
    numberOFCandidateIterations = 100; // how often a new candidate is
       selected within the same frame
    candidateIterationsAsFraction = 0.2;
    frameStep = 1;
    counter = 0;
    xCarpet = 7400;
    yCarpet = 5400;

    alphaUnary = 0.1f;
    alphaX = 1.0f;
```

```
    alphaY = 1.0f;
    alphaRot = 0.5f;

    betaUnary = 0.1f;
    betaX = 1.0f;
    betaY = 1.0f;
    betaRot = 0.5f;

    getCandidateMode = 2; // flatList
    candidateThreshold = 0.2f;

    energy.resize(numberOfIterations);
    selectedCandidateList.resize(numberOfIterations *
        numberOFCandidateIterations);

    //energy = alphaUnary * (bufferSize - sumUnaryPot) + alphaX * sumXPot /
        divXPot + alphaY * sumYPot / divYPot + alphaRot * sumRotPot /
        divRotPot;
}

// Destructor
StateCRF::~StateCRF()
{
    delete parameter;
}

void StateCRF::init()
{

}

void StateCRF::update(StateCRFResult& result)
{
    CRFData currentData;
    PoseCandidate currentCandidate, previousCandidate;
    BallCandidate previousBallCandidate, currentBallCandidate;
    Vector3f smoothPot;

    // increase counter of frameStep
    ++counter;
    result.robotPose.validity = 0.01;


    if(counter >= frameStep)
    {
        assignParams();
        prepareCRFData(currentData);

    DEBUG_RESPONSE_ONCE("StateCRF:Time",
                              OUTPUT(idText, text,"Now: "<<theFrameInfo.time);
                              OUTPUT(idText, text,"Tolerance: "<<timeTolerance);
```

```
                    OUTPUT(idText,text,"Ball Time: "<<
                        lastBallPerceptTimeStamp);

                    );
        //Choose Position and Ball Candidates
        currentCandidate = currentData.candidates[currentData.
            selectedCandidate];
        currentBallCandidate = getBallCandidate(currentCandidate);
        currentData.ballCandidate= currentBallCandidate;

        //ERP
        currentData.robots= getRobotsCandidates(currentCandidate);
        // Start Calculating CRF Starting Point
        currentData.unaryPot = getUnaryPot(currentCandidate,
            currentBallCandidate,currentData.robots);


        if(data.getNumberOfEntries() != 0)
        {
            predictedPose=getPredictedPose(0,-1); //Predicting Pose from
                Percetion and last CRFModelEntry

            previousBallCandidate = data[0].ballCandidate;
            predictedBallPose=getPredictedBallPose(previousBallCandidate,0,-1,
                currentBallCandidate);

            smoothPot = getSmoothnessPot(predictedPose,currentCandidate,
                                         predictedBallPose,
                                             currentBallCandidate);
            currentData.xPot=smoothPot.x;
            currentData.yPot=smoothPot.y;
            currentData.rotPot=smoothPot.z;

        }
        else
        {
            //initialise with high values
            currentData.xPot = xCarpet*10;
            currentData.yPot = yCarpet*10;
            currentData.rotPot = pi*2;
        }

        // add the new data set to the buffer / Overrides oldest data in
            Ringbuffer
        data.add(currentData);

        // start CRF
        STOP_TIME_ON_REQUEST("StateCRF:executeCRF", executeCRF(); );

        DEBUG_RESPONSE_ONCE("StateCRF:energy", outputEnergy(); );
```

```
        //Updating Result
        createRobotPose();
        result.robotPose=crfRobotPose;

        createRobotPoseInfo();
        result.robotPoseInfo=crfRobotPoseInfo;

        createBallModel();
        result.ballModel = crfBallModel;

        createRobotsModel();
        result.robotsModel = crfRobotsModel;

        result.unaryPot=data[0].unaryPot;
        result.smoothPot=data[0].xPot+data[0].yPot+data[0].rotPot;

        TEAM_OUTPUT_FAST(idTeamMateRobotPose, bin, RobotPoseCompressed(
            crfRobotPose));

        counter = 0;
    }

} // END OF UPDATE
//
    ********************************************************************************

void StateCRF::executeCRF()
{
    //CRFData currentData, previousData, nextData;
    int bufferStep = 1;
    PoseCandidate currentCandidate, previousCandidate, nextCandidate;
    BallCandidate currentBallCandidate, previousBallCandidate,
        nextBallCandidate;

    std::vector<RobotsModel::RobotCandidate> currentRobots;
    float unaryPot, xPot, yPot, rotPot, xPotNext, yPotNext, rotPotNext;
    Vector3f smoothPot;
    int selectedCandidate;
    float currentEnergy, currentEnergyBeta, newEnergyAlpha, newEnergyBeta;
    Pose2D odometryOffset, ballOdometryOffset;

    // go through the whole buffer several times
    for(unsigned int i = 0; i < numberOfIterations; ++i)
    {
        energy[i] = 0;

        // this is the only place where betaUnary has an influance on the
            result
```

```
        currentEnergy = betaUnary * data[0].unaryPot + alphaX * data[0].xPot +
            alphaY * data[0].yPot + alphaRot * data[0].rotPot;

        // first entry in buffer
        if((bufferStep < data.getNumberOfEntries()) && (bufferStep <
           bufferSize))
        {

            predictedPose = getPredictedPose(bufferStep,0);
            predictedBallPose = getPredictedBallPose(bufferStep,0);

            for(unsigned int k = 0; k < numberOFCandidateIterations; ++k)
            {
                selectedCandidate = rand() % data[0].candidates.size();
                currentCandidate = data[0].candidates[selectedCandidate];
                currentBallCandidate = getBallCandidate(currentCandidate);
                currentRobots = getRobotsCandidates(currentCandidate);

                DEBUG_RESPONSE_ONCE("RobotsEnemy",
                                         OUTPUT(idText,text,"How Many? : "<<
                                             currentRobots.size());
                                         OUTPUT(idText,text,"PosRobo? : "<<
                                             currentCandidate.x<<" "<<
                                             currentCandidate.y);


                                         if(currentRobots.size()>0)
            {
                OUTPUT(idText,text,"Pos: "<<currentRobots[0].position.x<<" "<<
                    currentRobots[0].position.y);
                    OUTPUT(idText,text,"rel Pos : "<<currentRobots[0].
                        relPosOnField.x<<" "<<currentRobots[0].relPosOnField.y
                        );
                    OUTPUT(idText,text,"Prob: "<<currentRobots[0].probability)
                        ;

                }
                                        );


                //ROBOTS

                unaryPot = getUnaryPot(currentCandidate,currentBallCandidate,
                    currentRobots);

                smoothPot = getSmoothnessPot(predictedPose,currentCandidate,
                                              predictedBallPose,
                                                 currentBallCandidate);
                //TODO
                //smoothPot+= getSmoothnessPotRobots(data[bufferStep].robots,
                    currentRobots);
```

```
                xPot= smoothPot.x;
                yPot=smoothPot.y;
                rotPot=smoothPot.z;

                newEnergyAlpha = betaUnary * unaryPot + alphaX * xPot + alphaY
                    * yPot + alphaRot * rotPot;

                if(newEnergyAlpha < currentEnergy)
                {
                    data[0].selectedCandidate = selectedCandidate;
                    data[0].unaryPot = unaryPot;
                    data[0].xPot = xPot;
                    data[0].yPot = yPot;
                    data[0].rotPot = rotPot;
                    data[0].ballCandidate = currentBallCandidate;
                    data[0].robots = currentRobots;
                    currentEnergy = newEnergyAlpha;
                }
                selectedCandidateList[i*numberOFCandidateIterations + k] =
                   selectedCandidate;
            }
        }
        else // bufferStep is to big to take odometry into account
        {
            for(unsigned int k = 0; k < numberOFCandidateIterations; ++k)
            {
                selectedCandidate = rand() % data[0].candidates.size();
                currentCandidate = data[0].candidates[selectedCandidate];

                currentBallCandidate = getBallCandidate(currentCandidate);
                currentRobots = getRobotsCandidates(currentCandidate);

                unaryPot = getUnaryPot(currentCandidate,currentBallCandidate,
                   currentRobots);

                newEnergyBeta = betaUnary * unaryPot;

                if(newEnergyBeta < currentEnergy)
                {
                    data[0].selectedCandidate = selectedCandidate;
                    data[0].unaryPot = unaryPot;
                    data[0].xPot = xCarpet*10;
                    data[0].yPot = yCarpet*10;
                    data[0].rotPot = pi*2;
                    data[0].ballCandidate = currentBallCandidate;
                    data[0].robots = currentRobots;

                    currentEnergy = newEnergyBeta;
                }
```

```
                selectedCandidateList[i*numberOFCandidateIterations + k] =
                    selectedCandidate;
            }
        }

        energy[i] = currentEnergy;

        // go back in time / Middle Iterative Actions
        int t = bufferStep;
        for(; (t < data.getNumberOfEntries()-bufferStep) && (t < bufferSize-
            bufferStep); t += bufferStep) // last entry is special again
        {
            // potentials from current time frame
            currentEnergy = alphaUnary * data[t].unaryPot + alphaX * data[t].
                xPot + alphaY * data[t].yPot + alphaRot * data[t].rotPot;
            // plus potentials from next time frame, unary has no influance
                her, stays constant
            currentEnergyBeta = betaX * data[t-bufferStep].xPot + betaY * data
                [t-bufferStep].yPot + betaRot * data[t-bufferStep].rotPot;

            currentEnergy += currentEnergyBeta;

            //Robo
            predictedPose = getPredictedPose(t+bufferStep,t); // Predicted
                Pose in time (t-bufferStep)
            nextCandidate = data[t-bufferStep].candidates[data[t-bufferStep].
                selectedCandidate]; // next in time = future


            //Ball
            nextBallCandidate = data[t-bufferStep].ballCandidate;
            predictedBallPose = getPredictedBallPose(t+bufferStep,t);

            for(unsigned int k = 0; k < numberOFCandidateIterations; ++k)
            {
                selectedCandidate = rand() % data[t].candidates.size();
                currentCandidate = data[t].candidates[selectedCandidate];
                currentBallCandidate = getBallCandidate(currentCandidate);
                currentRobots = getRobotsCandidates(currentCandidate);

                //Calculate Potentials
                unaryPot = getUnaryPot(currentCandidate,currentBallCandidate,
                    currentRobots);
                smoothPot = getSmoothnessPot(predictedPose,currentCandidate,
                                              predictedBallPose,
                                                 currentBallCandidate);
                //TODO Smooth Robots
                //smoothPot+= getSmoothnessPotRobots(data[t+bufferStep].robots
                    ,currentRobots);

                xPot= smoothPot.x;
```

```
                yPot=smoothPot.y;
                rotPot=smoothPot.z;

                // potential in next candidate changes too
                predictedNextPose=getPredictedPose(currentCandidate,t,t-
                    bufferStep);
                predictedNextBallPose = getPredictedBallPose(
                    currentBallCandidate,t,t-bufferStep);

                smoothPot = getSmoothnessPot(predictedNextPose,nextCandidate,
                                             predictedNextBallPose,
                                                 nextBallCandidate);


                xPotNext= smoothPot.x;
                yPotNext=smoothPot.y;
                rotPotNext=smoothPot.z;

                // potentials from current time frame
                newEnergyAlpha = alphaUnary * unaryPot + alphaX * xPot +
                    alphaY * yPot + alphaRot * rotPot;
                // plus potentials from next time frame
                newEnergyBeta = betaX * xPotNext + betaY * yPotNext + betaRot
                    * rotPotNext;

                if((newEnergyAlpha + newEnergyBeta) < currentEnergy)
                {
                     data[t].selectedCandidate = selectedCandidate;
                     data[t].unaryPot = unaryPot;
                     data[t].xPot = xPot;
                     data[t].yPot = yPot;
                     data[t].rotPot = rotPot;
                     data[t].ballCandidate = currentBallCandidate;
                     data[t].robots=currentRobots;

                     data[t-bufferStep].xPot = xPotNext;
                     data[t-bufferStep].yPot = yPotNext;
                     data[t-bufferStep].rotPot = rotPotNext;
                     data[t-bufferStep].ballCandidate = nextBallCandidate;
                     currentEnergy = newEnergyAlpha + newEnergyBeta;
                }
           }

           energy[i] += currentEnergy;
      }

      //last entry in buffer
      if((t < data.getNumberOfEntries()) && (t < bufferSize))
      {

```

```
            nextCandidate = data[t-bufferStep].candidates[data[t-bufferStep].
                selectedCandidate];
            nextBallCandidate = data[t-bufferStep].ballCandidate;

            // potentials from current time frame
            currentEnergy = alphaUnary * data[t].unaryPot;
            // plus potentials from next time frame
            currentEnergyBeta = betaX * data[t-bufferStep].xPot + betaY * data
                [t-bufferStep].yPot + betaRot * data[t-bufferStep].rotPot;
            currentEnergy += currentEnergyBeta;

            for(unsigned int k = 0; k < numberOFCandidateIterations; ++k)
            {
                selectedCandidate = rand() % data[t].candidates.size();
                currentCandidate = data[t].candidates[selectedCandidate];
                currentBallCandidate = getBallCandidate(currentCandidate);
                currentRobots = getRobotsCandidates(currentCandidate);

                // potential in next candidate changes too
                predictedNextPose = getPredictedPose(currentCandidate,t,t-
                    bufferStep);
                predictedNextBallPose = getPredictedBallPose(
                    currentBallCandidate,t,t-bufferStep);


                //Calculate Potentials
                unaryPot = getUnaryPot(currentCandidate,currentBallCandidate,
                    currentRobots);
                smoothPot = getSmoothnessPot(predictedNextPose,nextCandidate,
                                             predictedNextBallPose,
                                                nextBallCandidate);
                //TODO Smooth Robots
               //smoothPot+= getSmoothnessPotRobots(data[t].robots,
                  currentRobots);

                xPotNext= smoothPot.x;
                yPotNext=smoothPot.y;
                rotPotNext=smoothPot.z;

                // potentials from current time frame
                newEnergyAlpha = alphaUnary * unaryPot;
                // plus potentials from next time frame
                newEnergyBeta = betaX * xPotNext + betaY * yPotNext + betaRot
                    * rotPotNext;

                if((newEnergyAlpha + newEnergyBeta) < currentEnergy)
                {
                    data[t].selectedCandidate = selectedCandidate;
                    data[t].unaryPot = unaryPot;
                    data[t].ballCandidate = currentBallCandidate;
                    data[t].robots=currentRobots;
```

```
                        data[t-bufferStep].xPot = xPotNext;
                        data[t-bufferStep].yPot = yPotNext;
                        data[t-bufferStep].rotPot = rotPotNext;
                        data[t-bufferStep].ballCandidate =nextBallCandidate;

                        currentEnergy = newEnergyAlpha + newEnergyBeta;
                        //currentEnergyBeta = newEnergyBeta;
                    }
                }
                energy[i] += currentEnergy; // + betaUnary * data[t-bufferStep].
                    unaryPot;
            }
        }
        DEBUG_RESPONSE("StateCRF:drawSelectedCandidates", drawSelectedCandidates()
            ; );

}


void StateCRF::prepareCRFData(CRFData& currentData)
{

    int selectedCandidate, numOfCandidates;

    STOP_TIME_ON_REQUEST("StateCRF:getCandidates",
                        currentData.candidates = theRobotPoseHeatMap.
                            getCandidateList(getCandidateMode,
                            candidateThreshold);
                        );
    currentData.odometry = theOdometryData;

    numOfCandidates = currentData.candidates.size();
    selectedCandidate = rand() % numOfCandidates;

    DEBUG_RESPONSE_ONCE("StateCRF:numOfCandidates", OUTPUT(idText, text, "#
        Candidates: " << numOfCandidates); );
    PLOT("StateCRF:numOfCandidates", numOfCandidates);

    if(numberOFCandidateIterations > parameter->candidateIterationsAsFraction
        * numOfCandidates)
    {
        numberOFCandidateIterations = parameter->candidateIterationsAsFraction
            * numOfCandidates;
    }


    if(energy.size() != numberOfIterations)
    {
        energy.resize(numberOfIterations);
    }
```

```

    if(selectedCandidateList.size() != numberOfIterations * numberOFCandidateIterations)
    {
        selectedCandidateList.resize(numberOfIterations * numberOFCandidateIterations);
    }
    currentData.selectedCandidate = selectedCandidate;
}

void StateCRF::assignParams()
{
    DECLARE_DEBUG_DRAWING("module:StateCRF:selectedCandidates", "drawingOnField");

    MODIFY("module:StateCRF:parameter", *parameter);

    bufferSize = parameter->bufferSize;
    numberOfIterations = parameter->numberOfIterations;
    numberOFCandidateIterations = parameter->numberOFCandidateIterations; // how often a new candidate is selected within the same frame
    candidateIterationsAsFraction = parameter->candidateIterationsAsFraction;
    frameStep = parameter->frameStep;

    getCandidateMode = parameter->getCandidateMode;
    candidateThreshold = parameter->candidateThreshold;

    alphaUnary = parameter->alphaUnary;
    alphaX = parameter->alphaX/theRobotPoseHeatMap.cell[0];
    alphaY = parameter->alphaY/theRobotPoseHeatMap.cell[1];
    alphaRot = parameter->alphaRot*theRobotPoseHeatMap.maxIndex[0]/(pi*2);

    betaUnary = parameter->betaUnary;
    betaX = parameter->betaX/theRobotPoseHeatMap.cell[0];
    betaY = parameter->betaY/theRobotPoseHeatMap.cell[1];
    betaRot = parameter->betaRot*theRobotPoseHeatMap.maxIndex[0]/(pi*2);

    timeTolerance = parameter->timeTolerance;

    if(bufferSize > data.getMaxEntries())
    {
        bufferSize = data.getMaxEntries();
        OUTPUT(idText, text, "max buffer size is " << data.getMaxEntries() );
    }

}


void StateCRF::createRobotPoseInfo()
{
    crfRobotPoseInfo.timeLastPoseReset = theFrameInfo.time;
```

```
}

void StateCRF::createBallModel()
{

    PoseCandidate currentCandidate=data[0].candidates[data[0].
        selectedCandidate];
    BallCandidate currentBallCandidate=data[0].ballCandidate;

    BallState currentBallState; //Relative to Robot;

    currentBallState.position = getRelPos(currentBallCandidate.position,
        currentCandidate);

    float timeScale = 1.f / ((theFrameInfo.time - lastBallPerceptTimeStamp) *
        0.001f);

    Vector2<float> speed(data[0].ballOdometry.translation.x*timeScale,data[0].
        ballOdometry.translation.y*timeScale);

    //TODO
    currentBallState.velocity = speed ;

    crfBallModel.estimate = currentBallState;

    crfBallModel.endPosition = currentBallState.position;
    if(theBallPercept.ballWasSeen)
    {
        crfBallModel.timeWhenLastSeen = theFrameInfo.time;
        crfBallModel.lastPerception.position=theBallPercept.
            relativePositionOnField;
        crfBallModel.lastPerception.velocity=Vector2<>();
        crfBallModel.lastSeenEstimate = crfBallModel.estimate;
        lastBallPerceptTimeStamp=theFrameInfo.time;
    }


DEBUG_RESPONSE_ONCE("BallModel",
                                OUTPUT(idText,text,"Ball Pose: "<<crfBallModel.
                                    estimate.position.x<<" "<<crfBallModel.estimate.
                                    position.y);
                                OUTPUT(idText,text,"Speed: "<<crfBallModel.estimate.
                                    velocity.x<<" "<<crfBallModel.estimate.velocity.y)
                                    ;
                                OUTPUT(idText,text,"EndPos: "<<crfBallModel.
                                    endPosition.x<<" "<<crfBallModel.endPosition.y);
                                OUTPUT(idText,text,"Last Seen Estimate: "<<
                                    crfBallModel.timeWhenLastSeen);
                                OUTPUT(idText,text,"TimeLastStampPercept: "<<
                                    lastBallPerceptTimeStamp);
                               );
```

```
}

void StateCRF::createRobotPose()
{
    PoseCandidate currentCandidate;
    crfRobotPose.ownTeamColorForDrawing = theOwnTeamInfo.teamColor ==
        TEAM_BLUE ? ColorRGBA(0, 0, 255) : ColorRGBA(255, 0, 0);

    currentCandidate = data[0].candidates[data[0].selectedCandidate];
    crfRobotPose.rotation = currentCandidate.rotation;
    crfRobotPose.translation = Vector2<>(currentCandidate.x, currentCandidate.
        y);
    crfRobotPose.validity = 0.02;
    crfRobotPose.deviation = 1 - currentCandidate.probability;

}



void StateCRF::createRobotsModel()
{

    if(data.getNumberOfEntries()!=0)
    {
        PoseCandidate currentCandidate=data[0].candidates[data[0].
            selectedCandidate];
        if(crfRobotsModel.robots.size()!=data[0].robots.size())
            crfRobotsModel.robots.resize(data[0].robots.size());
        for(unsigned idx=0; idx<data[0].robots.size(); idx++)
        {
            RobotsModel::RobotCandidate robo = data[0].robots[idx];

            crfRobotsModel.robots[idx].relPosOnField=getRelPos(robo.position,
                currentCandidate);
            crfRobotsModel.robots[idx].teamRed=data[0].robots[idx].teamRed;
            crfRobotsModel.robots[idx].standing=data[0].robots[idx].standing;
            crfRobotsModel.robots[idx].covariance=data[0].robots[idx].
                covariance;
            crfRobotsModel.robots[idx].timeStamp=data[0].robots[idx].timeStamp
                ;
        }
    }
}

Vector2<> StateCRF::getRelPos(Vector2<int> coords, PoseCandidate robo)
{
    Vector2f temp(coords.x- robo.x, coords.y-robo.y);

    float viewAngle = robo.rotation;
    //rotation Matrix
```

```
    const float c = cos(viewAngle), s = sin(viewAngle);
    Matrix2x2f rot(Vector2f(c, -s), Vector2f(s, c)); //Vector are column
        vectors
    // Rotation of global vector to rel to Robo vector (Coord System of Robot)
    temp = rot*temp;
    Vector2<> temp2((int)temp.x,(int)temp.y);
    return temp2;


}


void StateCRF::outputEnergy()
{
    OUTPUT(idText, text, "Energy for each iteration:");

    for(unsigned int i =  0; i < numberOfIterations; ++i)
    {
        OUTPUT(idText, text, "#" << i << ": " << energy[i]);
    }

    OUTPUT(idText, text, "Unary" <<": " << data[0].unaryPot);
    OUTPUT(idText, text, "Smooth" <<": x:" << data[0].xPot << " y:"<<data[0].
        xPot);
}

std::vector<RobotsModel::RobotCandidate> StateCRF::getRobotsCandidates(
    PoseCandidate currentCandidate)
{
    vector<RobotsModel::RobotCandidate> newRobots;
    Vector2<int> roboPos(currentCandidate.x,currentCandidate.y);


    if(theERPHeatMap.robotsSeen!=0)
    {
        newRobots = theERPHeatMap.getCandidates(roboPos,currentCandidate.
            rotation);
    }
    else
    {
        if(!data[0].robots.empty() && data[0].robots[0].timeStamp>
            theFrameInfo.time-timeTolerance)
        {
            newRobots= data[0].robots;
        }
    }

    return newRobots;
}
```

```
std::vector<RobotsModel::RobotCandidate> StateCRF::unify(std::vector<
    RobotsModel::RobotCandidate> src1, std::vector<RobotsModel::RobotCandidate
    > src2)
{
     //TODO: FINISH
     vector<RobotsModel::RobotCandidate> result;
     vector<RobotsModel::RobotCandidate> other;
     bool isResultnewer=false;
     if(src1.size()<src2.size())
     {
          result=src2;
          other=src1;
     }
     else
     {
          //Equal big, take Perception
          result=src1;
          other=src2;
          isResultnewer=true;
     }

    //   RobotsModel::RobotCandidate* minDistanceRobot;

     for(std::vector<RobotsModel::RobotCandidate>::iterator it = other.begin()
         ; it != other.end(); ++it)
     {
          for(std::vector<RobotsModel::RobotCandidate>::iterator it2 = result.
              begin() ; it2 != result.end(); ++it2)
          {

               int minDistance=1000000;

               if(distance(it,it2)<minDistance)
               {
          //          RobotsModel::RobotCandidate* minDistanceRobot(it2);

               }

          }
          if(!isResultnewer)
          {
        //        (minDistanceRobot*)=(RobotsModel::RobotCandidate)(*it);
          }
     }



     return result;
}

BallCandidate StateCRF::getBallCandidate(PoseCandidate currentCandidate)
```

```
{
    if(theBallPercept.ballWasSeen)
    {
        Vector2<int> roboPos(currentCandidate.x,currentCandidate.y);
        return theBallHeatMap.getCandidate(roboPos,currentCandidate.rotation);
    }else{

        BallCandidate temp;
        if(data.getNumberOfEntries()!=0)// && lastBallPerceptTimeStamp>
            theFrameInfo.time-timeTolerance)// -> BallCandidate Default
        {
        BallCandidate lastEstimate = data[0].ballCandidate;
     //  Pose2D ballOdometry=data[0].ballOdometry;
     //   PoseCandidate pose = data[0].candidates[data[0].selectedCandidate];
      //  temp.probability=0;
      //  temp.position.x=lastEstimate.position.x;//+ballOdometry.translation.
         x;
       // temp.position.y=lastEstimate.position.y;//+ballOdometry.translation.
          y;
       temp=lastEstimate;
        }
     //   return temp;
     return temp;
    }

}

//Position Predictions

Pose2D StateCRF::getPredictedPose(int bufferOld, int bufferNew)
{
    PoseCandidate previousCandidate;
    previousCandidate = data[bufferOld].candidates[data[bufferOld].
        selectedCandidate];

    return getPredictedPose(previousCandidate,bufferOld,bufferNew);
}

Pose2D StateCRF::getPredictedPose(PoseCandidate& previousCandidate,int
    bufferOld, int bufferNew)
{
    Pose2D momOdometry;
    if(bufferNew==-1)
    {
        momOdometry=theOdometryData;

    }
    else
    {
        momOdometry=data[bufferNew].odometry;
    }
```

```
    Pose2D odometryOffset;
    odometryOffset = momOdometry − data[bufferOld].odometry;
    Vector2<> odoPrev(odometryOffset.translation.x, odometryOffset.translation.y);
    odoPrev.rotate(previousCandidate.rotation); // change to world coordinate frame

    Pose2D predictedPose;
    predictedPose.translation.x = previousCandidate.x + odoPrev.x;
    predictedPose.translation.y = previousCandidate.y + odoPrev.y;
    predictedPose.rotation = previousCandidate.rotation + odometryOffset.rotation;

    return predictedPose;
}

Pose2D StateCRF::getPredictedBallPose(int bufferOld, int bufferNew)
{
    BallCandidate previousCandidate;
    previousCandidate = data[bufferOld].ballCandidate;
    return getPredictedBallPose(previousCandidate,bufferOld,bufferNew);
}

Pose2D StateCRF::getPredictedBallPose(BallCandidate& previousCandidate,int bufferOld, int bufferNew)
{
    BallCandidate nullCandidate;
    nullCandidate.probability=−1;
    return getPredictedBallPose(previousCandidate,bufferOld,bufferNew,nullCandidate);
}



Pose2D StateCRF::getPredictedBallPose(BallCandidate& previousCandidate,int bufferOld,
                                      int bufferNew,BallCandidate& currentCandidate)
{
    Pose2D momOdometry;
    Pose2D ballOdometryOffset;
    Pose2D predictedPose;
    if(bufferNew==−1 && currentCandidate.probability==−1)
    {
        Pose2D temp(0,0);
        OUTPUT(idText,text,"Warning: Calling getPredictegBallPose with bufferNew −1 and currentCan null");
        OUTPUT(idText,text,"Setting momOdometry to (0,0)");
        momOdometry = temp;
    }
```

```
    else if(bufferNew==-1 && currentCandidate.probability!=-1)
    {
        momOdometry = currentCandidate.position;
    }
    else
    {
        momOdometry=data[bufferNew].ballOdometry;
    }

    ballOdometryOffset = momOdometry-data[bufferOld].ballOdometry;
    Vector2<> odoPrev(ballOdometryOffset.translation.x, ballOdometryOffset.
        translation.y);
    predictedPose.translation.x = previousCandidate.position.x + odoPrev.x;
    predictedPose.translation.y = previousCandidate.position.y + odoPrev.y;
    return predictedPose;
}


// CRF Potential Calculations

float StateCRF::getUnaryPot(PoseCandidate pose, BallCandidate ball,std::vector
   <RobotsModel::RobotCandidate> robots)
{
    float robotsUnary =1;
    for(unsigned i=0; i<robots.size(); i++)
    {
        robotsUnary*=(1-robots[i].probability);
    }
    return (1.f - pose.probability)*(1.f-ball.probability)*robotsUnary; //ball
        .probability implicit in pose.probability due getBallCandidate
        Gaussian
}

Vector3f StateCRF::getSmoothnessPotRobots(std::vector<RobotsModel::
    RobotCandidate> oldRobots, std::vector<RobotsModel::RobotCandidate>
    newRobots)
{


    Vector3f result;

    int oldR = (int)oldRobots.size();
    int newR =(int)newRobots.size();

    if(oldR==0 || newR==0)
        return result;


    MunkresMatrix<double> dist(oldR,newR);
    for (int i = 0; i < oldR; i++)
     for (int j = 0; j < newR; j++)
```

```
            dist(i,j)=getDistance(oldRobots[i],newRobots[j]);

    //Solve using Kuhn-Munkres
    Munkres m;
    m.solve(dist);



    for (int i = 0; i < oldR; i++){
      for (int j = 0; j < newR; j++){
        if(dist(i,j)==0){
            result.x += abs(oldRobots[i].position.x-newRobots[j].position.x);
            result.y += abs(oldRobots[i].position.x-newRobots[j].position.y);
            break;
        }
      }
    }
    result.z=0;



    return result;
}

int StateCRF::getDistance(RobotsModel::RobotCandidate rob1, RobotsModel::
    RobotCandidate rob2)
{
    int xQuo=(rob1.position.x-rob2.position.x);
    int yQuo=(rob1.position.y-rob2.position.y);

    return sqrt(xQuo*xQuo + yQuo*yQuo);
}



Vector3f StateCRF::getSmoothnessPot(Pose2D prediction1,PoseCandidate
    comparison1,
                                    Pose2D prediction2,BallCandidate
                                        comparison2
                                   )
{
    Vector3f result;
    result.x = abs(prediction1.translation.x - comparison1.x)
               +abs(prediction2.translation.x - comparison2.position.x)
               ;
    result.y= abs(prediction1.translation.y - comparison1.y)
               +abs(prediction2.translation.y - comparison2.position.y)
              ;
    result.z = abs(normalize(prediction1.rotation - comparison1.rotation));


```

```
    return result;
}

// DRAWING UTILITY

void StateCRF::drawSelectedCandidates()
{
    DECLARE_DEBUG_DRAWING("module:StateCRF:selectedCandidates", "drawingOnField");

    COMPLEX_DRAWING("module:StateCRF:selectedCandidates",
    {
        int r, g, b;
        ColorRGBA color;
        int x, y;
        float prob;

        for (unsigned int i = 0; i < selectedCandidateList.size(); ++i)
        {
            x = data[0].candidates[selectedCandidateList[i]].x;
            y = data[0].candidates[selectedCandidateList[i]].y;
            prob = data[0].candidates[selectedCandidateList[i]].probability;

            r = 255 * prob;
            b = 64 - prob * 64;

            if (r > 255)
                r = 255;
            else if (r < 0)
                r = 0;

            if (b > 64)
                b = 64;
            else if (b < 0)
                b = 0;

            g = r;

            color.r = (unsigned short)r;
            color.g = (unsigned short)g;
            color.b = (unsigned short)b;

            // draw a dot in the field


            LARGE_DOT("module:StateCRF:selectedCandidates", x, y, color, color);
            //OUTPUT(idText, text, "Candidate: " << selectedCandidateList[i] << "; x: " << x << "; y: " << y);

        }
```

```
    }); // END OF COMPLEX_DRAWING
}
```

Listing A.14: StateCRF.cpp

```
/**
* @file StateCRF.h
* Declares a class that calculates the current State using Conditional Random
   Fields (CRF)
* @author Hans Hardmeier, Martin Utz
*/

#pragma once

#include "Tools/Module/Module.h"
#include "Tools/Debugging/Stopwatch.h"
#include "Tools/Debugging/DebugDrawings.h"
#include "Representations/Infrastructure/CameraInfo.h"
#include "Representations/Infrastructure/FrameInfo.h"
#include "Representations/Infrastructure/TeamInfo.h"
#include "Representations/Infrastructure/TeamMateData.h"
#include "Representations/Configuration/FieldDimensions.h"
#include "Representations/Configuration/DamageConfiguration.h"
#include "Representations/Perception/CameraMatrix.h"
#include "Representations/MotionControl/SpecialActionRequest.h"
#include "Representations/MotionControl/MotionInfo.h"
#include "Representations/Perception/LinePercept.h"
#include "Representations/Perception/BallPercept.h"
#include "Representations/Perception/GoalPercept.h"
#include "Representations/Modeling/RobotPose.h"
#include "Representations/Modeling/RobotPoseHeatMap.h"
#include "Representations/Modeling/BallHeatMap.h"
#include "Representations/Modeling/ERPHeatMap.h"
#include "Representations/Modeling/BallModel.h"
#include "Representations/Modeling/RobotsModel.h"
#include "Representations/Modeling/FallDownState.h"
#include "Representations/Modeling/StateCRFResult.h"
#include "Representations/Sensing/GroundContactState.h"
#include "Representations/MotionControl/OdometryData.h"
#include "Tools/Math/Matrix.h"
#include "Tools/RingBuffer.h"
#include "Tools/Optimization/KuhnMunkres/munkres.h"
#include "Tools/Optimization/KuhnMunkres/MunkresMatrix.h"
#include "StateCRFParameter.h"
#include <vector>

#define BUFFERSIZE 150

MODULE(StateCRF)
REQUIRES(RobotPoseHeatMap)
REQUIRES(BallHeatMap)
REQUIRES(ERPHeatMap)
```

```
//REQUIRES(CameraMatrix)
//REQUIRES(FieldDimensions)
//REQUIRES(DamageConfiguration)
//REQUIRES(CameraInfo)
REQUIRES(BallPercept)
REQUIRES(OwnTeamInfo)
REQUIRES(MotionInfo)
REQUIRES(OdometryData)
//REQUIRES(GoalPercept)
//REQUIRES(GroundContactState)
//REQUIRES(FallDownState)
REQUIRES(FrameInfo)
REQUIRES(TeamMateData) // for TEAM_OUTPUT_FAST
//  PROVIDES_WITH_MODIFY_AND_OUTPUT_AND_DRAW(RobotPose)
//  REQUIRES(RobotPose)
//  REQUIRES(BallModel)
//  PROVIDES(BallModel)
//  PROVIDES_WITH_MODIFY(RobotPoseInfo)
PROVIDES(StateCRFResult)
END_MODULE


/**
* @class CRFData
* Dataset which contains all candidates and the odometry
*/
class CRFData
{
public:
    std::vector<PoseCandidate> candidates;
    Pose2D odometry;
    float unaryPot;
    float xPot;
    float yPot;
    float rotPot;
    int selectedCandidate;

    //Data for BallCRF
    BallCandidate ballCandidate;
    Pose2D ballOdometry;

    std::vector<RobotsModel::RobotCandidate> robots;

    CRFData() : odometry(),ballCandidate(),ballOdometry(),robots()
    {
        unaryPot = 1.0f;
        xPot = 1.0f;
        yPot = 1.0f;
        rotPot = 1.0f;
        selectedCandidate = 0;
    }
```

```
};



/**
* @class StateCRF
* A module that determines the validity of a robot pose.
*/
class StateCRF : public StateCRFBase
{
public:
    /**
    * Default constructor.
    */
    StateCRF();

    /**
    * Destructor
    */
    ~StateCRF();


private:

    StateCRFParameter* parameter;
    int bufferSize; /**< not the real buffer size, but the maximum # of frames
         if frameStep is set to 1 */
    unsigned int numberOfIterations; /**< how many time to go through the
        ringbuffer during CRF */
    unsigned int numberOFCandidateIterations; /**< how often a new candidate
        is selected within the same frame */
    float candidateIterationsAsFraction; /**< how often a new candidate is
        selected within the same frame as a fraction of the number of
        candidates */
    int frameStep; /**< if != 1 then not every frame is processed */
    int getCandidateMode; /**< used o determine the mode in getCandidateList(
        mode, threshold); 1=whole heatMap, 2=flatList, 3=threshold */
    float candidateThreshold; /** (relative) Threshold to be reached for a
        cell to be a candidate for CRF */
    int counter;

    float xCarpet;
    float yCarpet;

    Pose2D lastOdometryData; /**< OdometryData of the previous iteration. */
    Pose2D predictedPose; /**< predicted pose (previous candidate +
        odometryOffset) */
    Pose2D predictedNextPose; /** predicted pose (current candidate +
        odometryOffset)*/
```

```
    Pose2D lastBallOdometryData; /**< OdometryData of the previous iteration.
        */
    Pose2D predictedBallPose; /**< predicted pose (previous candidate +
        odometryOffset) */
    Pose2D predictedNextBallPose; /** predicted pose (current candidate +
        odometryOffset)*/



    /*float sumUnaryPot;
    float sumXPot;
    float sumYPot;
    float sumRotPot;*/
    float alphaUnary;
    float alphaX;
    float alphaY;
    float alphaRot;

    float betaUnary;
    float betaX;
    float betaY;
    float betaRot;

    float timeTolerance;

    std::vector<float> energy;
    std::vector<int> selectedCandidateList;

    float lastBallPerceptTimeStamp;


    BallModel crfBallModel;
    RobotPose crfRobotPose;
    RobotPoseInfo crfRobotPoseInfo;
    RobotsModel crfRobotsModel;


    RingBuffer<CRFData, BUFFERSIZE> data;

    /**
    * Initializes the module.
    */
    void init();


    /**
    * Updates the validated robot pose provided from this module.
    * @param robotPose The robotPose updated from the module
    */
    void update(StateCRFResult& result);
```

```
    /**
    * Updates further information about the robot pose.
    * @param robotPose The robotPoseInfo updated from the module
    */
    void update(RobotPoseInfo& robotPoseInfo);

//  void update(BallModel& ballModel);


    void executeCRF();
    //void executeCRFold();

    void outputEnergy();

    void drawSelectedCandidates();

    /** Create BallModel for DistributionCRF*/
    void createBallModel();

    /** Create RobotPose for DistributionCRF*/
    void createRobotPose();

    /** Create RobotPoseInfo for DistributionCRF*/
    void createRobotPoseInfo();

    /** Create RobotsModel for DistributionCRF*/
    void createRobotsModel();

    /** Assign new Parameters of CRF*/
    void assignParams();

    /** Prepare CRF Data*/
    void prepareCRFData(CRFData& currentData);

    /** Given a PoseCandidate return repons. BallCandidate */
    BallCandidate getBallCandidate(PoseCandidate currentCandidate);
    /** Given a PoseCandidate return repons. RobotsCandidate*/
    std::vector<RobotsModel::RobotCandidate> getRobotsCandidates(PoseCandidate
         currentCandidate);


    /** Small feature to get Unary Potential*/
    float getUnaryPot(PoseCandidate pose, BallCandidate ball,std::vector<
        RobotsModel::RobotCandidate> robots);

    /** Predicted Pose at time bufferNew*/
    Pose2D getPredictedPose(int bufferOld, int bufferNew);
    Pose2D getPredictedPose(PoseCandidate& previousCandidate,int bufferOld,
        int bufferNew);
```

```
    /** Predicted BallPose at time bufferNew*/
    Pose2D getPredictedBallPose(int bufferOld, int bufferNew);
    Pose2D getPredictedBallPose(BallCandidate& previousCandidate,int bufferOld
        , int bufferNew);
    Pose2D getPredictedBallPose(BallCandidate& previousCandidate,int bufferOld
        ,
                                            int bufferNew,BallCandidate&
                                                currentCandidate);

    /** Smoothness*/
    Vector3f getSmoothnessPot(Pose2D prediction1,PoseCandidate comparison1,
                                            Pose2D prediction2,BallCandidate
                                                comparison2
                                            );
    Vector3f getSmoothnessPotRobots(std::vector<RobotsModel::RobotCandidate>
        oldRobots,
                                            std::vector<RobotsModel::RobotCandidate>
                                                newRobots);



                /** Utility: Calculation rel Pos to Robot*/
    Vector2<> getRelPos(Vector2<int> coords, PoseCandidate robo);

    /**Update the Perception and CRF DATA of RobotsModel at current Time */
    std::vector<RobotsModel::RobotCandidate> unify(std::vector<RobotsModel::
        RobotCandidate> src1, std::vector<RobotsModel::RobotCandidate> src2);


    /** Get Distance between robots*/
    int getDistance(RobotsModel::RobotCandidate rob1, RobotsModel::
        RobotCandidate rob2);

};
```

Listing A.15: StateCRF.h

```

#pragma once

#include "Tools/Math/Pose2D.h"
#include "Tools/Settings.h"
#include "Tools/Range.h"

/**
* A collection of all parameters of the module.
*/
class StateCRFParameter : public Streamable
{
private:
  /**
  * The method makes the object streamable.
```

```
  * @param in The stream from which the object is read
  * @param out The stream to which the object is written
  */
  virtual void serialize(In* in, Out* out)
  {
    STREAM_REGISTER_BEGIN();
    STREAM(bufferSize);
    STREAM(numberOfIterations);
    STREAM(numberOFCandidateIterations);
    STREAM(candidateIterationsAsFraction);
    STREAM(frameStep);
    STREAM(getCandidateMode);
    STREAM(candidateThreshold);
    STREAM(alphaUnary);
    STREAM(alphaX);
    STREAM(alphaY);
    STREAM(alphaRot);
    STREAM(betaUnary);
    STREAM(betaX);
    STREAM(betaY);
    STREAM(betaRot);
    STREAM(timeTolerance);
    STREAM_REGISTER_FINISH();
  }

public:
  StateCRFParameter();
  void load(const std::string& fileName);
  int bufferSize;
  unsigned int numberOfIterations;
  unsigned int numberOFCandidateIterations; // how often a new candidate is
      selected within the same frame
  float candidateIterationsAsFraction;
  int frameStep;
  int getCandidateMode;
  float candidateThreshold;

  // tuning parameters for potentials
  float alphaUnary;
  float alphaX;
  float alphaY;
  float alphaRot;

  float betaUnary;
  float betaX;
  float betaY;
  float betaRot;

  float timeTolerance;

};
```

Listing A.16: StateCRFParameter.h

```cpp
#include "StateCRFParameter.h"
#include "Platform/BHAssert.h"
#include "Tools/Streams/InStreams.h"
#include "Tools/Global.h"

StateCRFParameter::StateCRFParameter()
{
  load("stateCRF.cfg");
}

void StateCRFParameter::load(const std::string& fileName)
{
  InConfigMap file(Global::getSettings().expandLocationFilename(fileName));
  ASSERT(file.exists());
  file >> *this;
}
```

Listing A.17: StateCRFParameter.cpp

```cpp
/**
 * @file DistributorCRF.h
 * Distributes the Result of the CRf to the Models
 * @author Hans Hardmeier
 * @date 19/06/19
 * Copyright (c) 2013 __ETHZ__. All rights reserved.
 */

#pragma once        //Makes sure that the source file is only included once in
    the compilation
#include "Tools/Module/Module.h"
#include "Representations/Modeling/StateCRFResult.h"
#include "Representations/Modeling/RobotPose.h"
#include "Representations/Modeling/BallModel.h"
#include "Representations/Modeling/RobotsModel.h"


#include "Tools/Debugging/Debugging.h"

MODULE(DistributorCRF)
  REQUIRES(StateCRFResult)
  PROVIDES_WITH_MODIFY_AND_OUTPUT_AND_DRAW(RobotPose)
  PROVIDES_WITH_MODIFY(RobotPoseInfo)
  PROVIDES_WITH_MODIFY_AND_OUTPUT_AND_DRAW(BallModel)
  PROVIDES_WITH_MODIFY_AND_OUTPUT_AND_DRAW(RobotsModel)
END_MODULE

class DistributorCRF : public DistributorCRFBase
{
```

```
private:

void update(RobotPose& robo);
void update(BallModel& ball);
void update(RobotPoseInfo& info);
void update (RobotsModel& robotsModel);

};

MAKE_MODULE(DistributorCRF, Modeling)
```

Listing A.18: DistributorCRF.h

```
/**
 * @file DistributorCRF.cpp
 * B-Human //
 * @author Hans Hardmeier
 * @date 19/06/13
 * Copyright (c) 2013 __ETHZ__. All rights reserved.
 */

#include "DistributorCRF.h"  //Include Header



void DistributorCRF::update(RobotPose& robo)
{
     robo=theStateCRFResult.robotPose;
     DEBUG_RESPONSE_ONCE("RoboCRF",
                              OUTPUT(idText,text,"Updating RobotPose");
                              OUTPUT(idText,text,"Place: "<<robo.translation.x<<" "
                                  <<robo.translation.y);
                             );


}

void DistributorCRF::update(BallModel& ball)
{

     ball=theStateCRFResult.ballModel;
     DEBUG_RESPONSE_ONCE("BallCRF",
                              OUTPUT(idText,text,"Updating BallModel");
                              OUTPUT(idText,text,"Place: "<<ball.estimate.position.x
                                  <<" "<<ball.estimate.position.y);
                             );
}

void DistributorCRF::update(RobotPoseInfo& info)
{
     info=theStateCRFResult.robotPoseInfo;
}
```

```
void DistributorCRF::update(RobotsModel& robotsModel)
{DEBUG_RESPONSE_ONCE("RobotsCRF",
                        OUTPUT(idText,text,"Updating RobotsModel");
                        OUTPUT(idText,text,"Rel Place: "<<robotsModel.robots
                            [0].relPosOnField.x<<" "<<robotsModel.robots[0].
                            relPosOnField.y);
                        OUTPUT(idText,text,"Time: "<<robotsModel.robots[0].
                            timeStamp);
                       );
    robotsModel=theStateCRFResult.robotsModel;
}
```

Listing A.19: DistributorCRF.cpp

```
/*
 *   Copyright (c) 2007 John Weaver
 *
 *   This program is free software; you can redistribute it and/or modify
 *   it under the terms of the GNU General Public License as published by
 *   the Free Software Foundation; either version 2 of the License, or
 *   (at your option) any later version.
 *
 *   This program is distributed in the hope that it will be useful,
 *   but WITHOUT ANY WARRANTY; without even the implied warranty of
 *   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 *   GNU General Public License for more details.
 *
 *   You should have received a copy of the GNU General Public License
 *   along with this program; if not, write to the Free Software
 *   Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307 USA
 */

#include "munkres.h"

#include <iostream>
#include <cmath>

bool
Munkres::find_uncovered_in_matrix(double item, int &row, int &col) {
  for ( row = 0 ; row < matrix.rows() ; row++ )
    if ( !row_mask[row] )
      for ( col = 0 ; col < matrix.columns() ; col++ )
        if ( !col_mask[col] )
          if ( matrix(row,col) == item )
            return true;

  return false;
}

bool
```

```
Munkres::pair_in_list(const std::pair<int,int> &needle, const std::list<std::
    pair<int,int> > &haystack) {
  for ( std::list<std::pair<int,int> >::const_iterator i = haystack.begin() ;
      i != haystack.end() ; i++ ) {
    if ( needle == *i )
      return true;
  }

  return false;
}

int
Munkres::step1(void) {
  for ( int row = 0 ; row < matrix.rows() ; row++ )
    for ( int col = 0 ; col < matrix.columns() ; col++ )
      if ( matrix(row,col) == 0 ) {
        bool isstarred = false;
        for ( int nrow = 0 ; nrow < matrix.rows() ; nrow++ )
          if ( mask_matrix(nrow,col) == STAR ) {
            isstarred = true;
            break;
          }

        if ( !isstarred ) {
          for ( int ncol = 0 ; ncol < matrix.columns() ; ncol++ )
            if ( mask_matrix(row,ncol) == STAR ) {
              isstarred = true;
              break;
            }
        }

        if ( !isstarred ) {
          mask_matrix(row,col) = STAR;
        }
      }

  return 2;
}

int
Munkres::step2(void) {
  int rows = matrix.rows();
  int cols = matrix.columns();
  int covercount = 0;
  for ( int row = 0 ; row < rows ; row++ )
    for ( int col = 0 ; col < cols ; col++ )
      if ( mask_matrix(row,col) == STAR ) {
        col_mask[col] = true;
        covercount++;
      }

```

```
  int k = matrix.minsize();

  if ( covercount >= k ) {
    return 0;
  }
  return 3;
}

int
Munkres::step3(void) {
  /*
  Main Zero Search

   1. Find an uncovered Z in the distance matrix and prime it. If no such zero
       exists, go to Step 5
   2. If No Z* exists in the row of the Z', go to Step 4.
   3. If a Z* exists, cover this row and uncover the column of the Z*. Return
       to Step 3.1 to find a new Z
  */
  if ( find_uncovered_in_matrix(0, saverow, savecol) ) {
    mask_matrix(saverow,savecol) = PRIME; // prime it.
  } else {
    return 5;
  }

  for ( int ncol = 0 ; ncol < matrix.columns() ; ncol++ )
    if ( mask_matrix(saverow,ncol) == STAR ) {
      row_mask[saverow] = true; //cover this row and
      col_mask[ncol] = false; // uncover the column containing the starred
          zero
      return 3; // repeat
    }

  return 4; // no starred zero in the row containing this primed zero
}

int
Munkres::step4(void) {
  int rows = matrix.rows();
  int cols = matrix.columns();

  std::list<std::pair<int,int> > seq;
  // use saverow, savecol from step 3.
  std::pair<int,int> z0(saverow, savecol);
  std::pair<int,int> z1(-1,-1);
  std::pair<int,int> z2n(-1,-1);
  seq.insert(seq.end(), z0);
  int row, col = savecol;
  /*
  Increment Set of Starred Zeros

```

```
  1. Construct the ``alternating sequence'' of primed and starred zeros:

        Z0 : Unpaired Z' from Step 4.2
        Z1 : The Z* in the column of Z0
        Z[2N] : The Z' in the row of Z[2N-1], if such a zero exists
        Z[2N+1] : The Z* in the column of Z[2N]

     The sequence eventually terminates with an unpaired Z' = Z[2N] for some
        N.
*/
bool madepair;
do {
  madepair = false;
  for ( row = 0 ; row < rows ; row++ )
    if ( mask_matrix(row,col) == STAR ) {
      z1.first = row;
      z1.second = col;
      if ( pair_in_list(z1, seq) )
        continue;

      madepair = true;
      seq.insert(seq.end(), z1);
      break;
    }

  if ( !madepair )
    break;

  madepair = false;

  for ( col = 0 ; col < cols ; col++ )
    if ( mask_matrix(row,col) == PRIME ) {
      z2n.first = row;
      z2n.second = col;
      if ( pair_in_list(z2n, seq) )
        continue;
      madepair = true;
      seq.insert(seq.end(), z2n);
      break;
    }
} while ( madepair );

for ( std::list<std::pair<int,int> >::iterator i = seq.begin() ;
    i != seq.end() ;
    i++ ) {
  // 2. Unstar each starred zero of the sequence.
  if ( mask_matrix(i->first,i->second) == STAR )
    mask_matrix(i->first,i->second) = NORMAL;

  // 3. Star each primed zero of the sequence,
  // thus increasing the number of starred zeros by one.
```

```
    if ( mask_matrix(i->first,i->second) == PRIME )
      mask_matrix(i->first,i->second) = STAR;
  }

  // 4. Erase all primes, uncover all columns and rows,
  for ( int row = 0 ; row < mask_matrix.rows() ; row++ )
    for ( int col = 0 ; col < mask_matrix.columns() ; col++ )
      if ( mask_matrix(row,col) == PRIME )
        mask_matrix(row,col) = NORMAL;

  for ( int i = 0 ; i < rows ; i++ ) {
    row_mask[i] = false;
  }

  for ( int i = 0 ; i < cols ; i++ ) {
    col_mask[i] = false;
  }

  // and return to Step 2.
  return 2;
}

int
Munkres::step5(void) {
  int rows = matrix.rows();
  int cols = matrix.columns();
  /*
  New Zero Manufactures

   1. Let h be the smallest uncovered entry in the (modified) distance matrix.
   2. Add h to all covered rows.
   3. Subtract h from all uncovered columns
   4. Return to Step 3, without altering stars, primes, or covers.
  */
  double h = 0;
  for ( int row = 0 ; row < rows ; row++ ) {
    if ( !row_mask[row] ) {
      for ( int col = 0 ; col < cols ; col++ ) {
        if ( !col_mask[col] ) {
          if ( (h > matrix(row,col) && matrix(row,col) != 0) || h == 0 ) {
            h = matrix(row,col);
          }
        }
      }
    }
  }

  for ( int row = 0 ; row < rows ; row++ )
    if ( row_mask[row] )
      for ( int col = 0 ; col < cols ; col++ )
        matrix(row,col) += h;
```

```
  for ( int col = 0 ; col < cols ; col++ )
    if ( !col_mask[col] )
      for ( int row = 0 ; row < rows ; row++ )
        matrix(row,col) -= h;

  return 3;
}

void
Munkres::solve(MunkresMatrix<double> &m) {
  // Linear assignment problem solution
  // [modifies matrix in-place.]
  // matrix(row,col): row major format assumed.

  // Assignments are remaining 0 values
  // (extra 0 values are replaced with -1)

  double highValue = 0;
  for ( int row = 0 ; row < m.rows() ; row++ ) {
    for ( int col = 0 ; col < m.columns() ; col++ ) {
      if ( m(row,col) != INFINITY && m(row,col) > highValue )
        highValue = m(row,col);
    }
  }
  highValue++;

  for ( int row = 0 ; row < m.rows() ; row++ )
    for ( int col = 0 ; col < m.columns() ; col++ )
      if ( m(row,col) == INFINITY )
        m(row,col) = highValue;

  bool notdone = true;
  int step = 1;

  this->matrix = m;
  // STAR == 1 == starred, PRIME == 2 == primed
  mask_matrix.resize(matrix.rows(), matrix.columns());

  row_mask = new bool[matrix.rows()];
  col_mask = new bool[matrix.columns()];
  for ( int i = 0 ; i < matrix.rows() ; i++ ) {
    row_mask[i] = false;
  }

  for ( int i = 0 ; i < matrix.columns() ; i++ ) {
    col_mask[i] = false;
  }

  while ( notdone ) {
    switch ( step ) {
```

```
      case 0:
        notdone = false;
        break;
      case 1:
        step = step1();
        break;
      case 2:
        step = step2();
        break;
      case 3:
        step = step3();
        break;
      case 4:
        step = step4();
        break;
      case 5:
        step = step5();
        break;
    }
  }

  // Store results
  for ( int row = 0 ; row < matrix.rows() ; row++ )
    for ( int col = 0 ; col < matrix.columns() ; col++ )
      if ( mask_matrix(row,col) == STAR )
        matrix(row,col) = 0;
      else
        matrix(row,col) = -1;

  m = matrix;

  delete [] row_mask;
  delete [] col_mask;
}
```

Listing A.20: munkres.cpp

```
/*
 *   Copyright (c) 2007 John Weaver
 *
 *   This program is free software; you can redistribute it and/or modify
 *   it under the terms of the GNU General Public License as published by
 *   the Free Software Foundation; either version 2 of the License, or
 *   (at your option) any later version.
 *
 *   This program is distributed in the hope that it will be useful,
 *   but WITHOUT ANY WARRANTY; without even the implied warranty of
 *   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 *   GNU General Public License for more details.
 *
 *   You should have received a copy of the GNU General Public License
 *   along with this program; if not, write to the Free Software
```

```
 *    Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307 USA
 */

#if !defined(_MUNKRES_H_)
#define _MUNKRES_H_

#include "Tools/Optimization/KuhnMunkres/MunkresMatrix.h"

#include <list>
#include <utility>

class Munkres {
public:
    void solve(MunkresMatrix<double> &m);
private:
    static const int NORMAL = 0;
    static const int STAR = 1;
    static const int PRIME = 2;
    inline bool find_uncovered_in_matrix(double,int&,int&);
    inline bool pair_in_list(const std::pair<int,int> &, const std::list<std::
        pair<int,int> > &);
    int step1(void);
    int step2(void);
    int step3(void);
    int step4(void);
    int step5(void);
    int step6(void);
    MunkresMatrix<int> mask_matrix;
    MunkresMatrix<double> matrix;
    bool *row_mask;
    bool *col_mask;
    int saverow, savecol;
};

#endif /* !defined(_MUNKRES_H_) */
```

Listing A.21: munkres.h

```
/*
 *    Copyright (c) 2007 John Weaver
 *
 *    This program is free software; you can redistribute it and/or modify
 *    it under the terms of the GNU General Public License as published by
 *    the Free Software Foundation; either version 2 of the License, or
 *    (at your option) any later version.
 *
 *    This program is distributed in the hope that it will be useful,
 *    but WITHOUT ANY WARRANTY; without even the implied warranty of
 *    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 *    GNU General Public License for more details.
 *
 *    You should have received a copy of the GNU General Public License
```

```
 *    along with this program; if not, write to the Free Software
 *    Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307 USA
 */

#ifndef Included_Munkres_H
#define Included_Munkres_H


#include "MunkresMatrix.h"

#include <cassert>
#include <cstdlib>
#include <algorithm>

/*export*/ template <class T>
MunkresMatrix<T>::MunkresMatrix() {
    m_rows = 0;
    m_columns = 0;
    m_matrix = NULL;
}

/*export*/ template <class T>
MunkresMatrix<T>::MunkresMatrix(const MunkresMatrix<T> &other) {
    if ( other.m_matrix != NULL ) {
        // copy arrays
        m_matrix = NULL;
        resize(other.m_rows, other.m_columns);
        for ( int i = 0 ; i < m_rows ; i++ )
            for ( int j = 0 ; j < m_columns ; j++ )
                m_matrix[i][j] = other.m_matrix[i][j];
    } else {
        m_matrix = NULL;
        m_rows = 0;
        m_columns = 0;
    }
}

/*export*/ template <class T>
MunkresMatrix<T>::MunkresMatrix(int rows, int columns) {
    m_matrix = NULL;
    resize(rows, columns);
}

/*export*/ template <class T>
MunkresMatrix<T> &
MunkresMatrix<T>::operator= (const MunkresMatrix<T> &other) {
    if ( other.m_matrix != NULL ) {
        // copy arrays
        resize(other.m_rows, other.m_columns);
        for ( int i = 0 ; i < m_rows ; i++ )
            for ( int j = 0 ; j < m_columns ; j++ )
```

```
                m_matrix[i][j] = other.m_matrix[i][j];
    } else {
        // free arrays
        for ( int i = 0 ; i < m_columns ; i++ )
            delete [] m_matrix[i];

        delete [] m_matrix;

        m_matrix = NULL;
        m_rows = 0;
        m_columns = 0;
    }

    return *this;
}

/*export*/ template <class T>
MunkresMatrix<T>::~MunkresMatrix() {
    if ( m_matrix != NULL ) {
        // free arrays
        for ( int i = 0 ; i < m_rows ; i++ )
            delete [] m_matrix[i];

        delete [] m_matrix;
    }
    m_matrix = NULL;
}

/*export*/ template <class T>
void
MunkresMatrix<T>::resize(int rows, int columns) {
    if ( m_matrix == NULL ) {
        // alloc arrays
        m_matrix = new T*[rows]; // rows
        for ( int i = 0 ; i < rows ; i++ )
            m_matrix[i] = new T[columns]; // columns

        m_rows = rows;
        m_columns = columns;
        clear();
    } else {
        // save array pointer
        T **new_matrix;
        // alloc new arrays
        new_matrix = new T*[rows]; // rows
        for ( int i = 0 ; i < rows ; i++ ) {
            new_matrix[i] = new T[columns]; // columns
            for ( int j = 0 ; j < columns ; j++ )
                new_matrix[i][j] = 0;
        }

```

```
        // copy data from saved pointer to new arrays
        int minrows = std::min<int>(rows, m_rows);
        int mincols = std::min<int>(columns, m_columns);
        for ( int x = 0 ; x < minrows ; x++ )
            for ( int y = 0 ; y < mincols ; y++ )
                new_matrix[x][y] = m_matrix[x][y];

        // delete old arrays
        if ( m_matrix != NULL ) {
            for ( int i = 0 ; i < m_rows ; i++ )
                delete [] m_matrix[i];

            delete [] m_matrix;
        }

        m_matrix = new_matrix;
    }

    m_rows = rows;
    m_columns = columns;
}

/*export*/ template <class T>
void
MunkresMatrix<T>::identity() {
    assert( m_matrix != NULL );

    clear();

    int x = std::min<int>(m_rows, m_columns);
    for ( int i = 0 ; i < x ; i++ )
        m_matrix[i][i] = 1;
}

/*export*/ template <class T>
void
MunkresMatrix<T>::clear() {
    assert( m_matrix != NULL );

    for ( int i = 0 ; i < m_rows ; i++ )
        for ( int j = 0 ; j < m_columns ; j++ )
            m_matrix[i][j] = 0;
}

/*export*/ template <class T>
T
MunkresMatrix<T>::trace() {
    assert( m_matrix != NULL );

    T value = 0;
```

```
    int x = std::min<int>(m_rows, m_columns);
    for ( int i = 0 ; i < x ; i++ )
        value += m_matrix[i][i];

    return value;
}

/*export*/ template <class T>
MunkresMatrix<T>&
MunkresMatrix<T>::transpose() {
    assert( m_rows > 0 );
    assert( m_columns > 0 );

    int new_rows = m_columns;
    int new_columns = m_rows;

    if ( m_rows != m_columns ) {
        // expand matrix
        int m = std::max<int>(m_rows, m_columns);
        resize(m,m);
    }

    for ( int i = 0 ; i < m_rows ; i++ ) {
        for ( int j = i+1 ; j < m_columns ; j++ ) {
            T tmp = m_matrix[i][j];
            m_matrix[i][j] = m_matrix[j][i];
            m_matrix[j][i] = tmp;
        }
    }

    if ( new_columns != new_rows ) {
        // trim off excess.
        resize(new_rows, new_columns);
    }

    return *this;
}

/*export*/ template <class T>
MunkresMatrix<T>
MunkresMatrix<T>::product(MunkresMatrix<T> &other) {
    assert( m_matrix != NULL );
    assert( other.m_matrix != NULL );
    assert ( m_columns == other.m_rows );

    MunkresMatrix<T> out(m_rows, other.m_columns);

    for ( int i = 0 ; i < out.m_rows ; i++ ) {
        for ( int j = 0 ; j < out.m_columns ; j++ ) {
            for ( int x = 0 ; x < m_columns ; x++ ) {
                out(i,j) += m_matrix[i][x] * other.m_matrix[x][j];
```

```
            }
        }
    }

    return out;
}

/*export*/ template <class T>
inline T&
MunkresMatrix<T>::operator ()(int x, int y) {
    assert ( x >= 0 );
    assert ( y >= 0 );
    assert ( x < m_rows );
    assert ( y < m_columns );
    assert ( m_matrix != NULL );
    return m_matrix[x][y];
}
#endif // Included_Munkres_H
```

Listing A.22: MunkresMatrix.cpp

```
/*
 *   Copyright (c) 2007 John Weaver
 *
 *   This program is free software; you can redistribute it and/or modify
 *   it under the terms of the GNU General Public License as published by
 *   the Free Software Foundation; either version 2 of the License, or
 *   (at your option) any later version.
 *
 *   This program is distributed in the hope that it will be useful,
 *   but WITHOUT ANY WARRANTY; without even the implied warranty of
 *   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 *   GNU General Public License for more details.
 *
 *   You should have received a copy of the GNU General Public License
 *   along with this program; if not, write to the Free Software
 *   Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307 USA
 */

#if !defined(_MATRIX_H_)
#define _MATRIX_H_

template <class T>
class MunkresMatrix {
public:
    MunkresMatrix();
    MunkresMatrix(int rows, int columns);
    MunkresMatrix(const MunkresMatrix<T> &other);
    MunkresMatrix<T> & operator= (const MunkresMatrix<T> &other);
    ~MunkresMatrix();
    // all operations except product modify the matrix in-place.
    void resize(int rows, int columns);
```

```
    void identity(void);
    void clear(void);
    T& operator () (int x, int y);
    T trace(void);
    MunkresMatrix<T>& transpose(void);
    MunkresMatrix<T> product(MunkresMatrix<T> &other);
    inline int minsize(void) {
        return ((m_rows < m_columns) ? m_rows : m_columns);
    }
    inline int columns(void) {
        return m_columns;
    }
    inline int rows(void) {
        return m_rows;
    }
private:
    T **m_matrix;
    int m_rows;
    int m_columns;
};

#ifndef USE_EXPORT_KEYWORD
#include "MunkresMatrix.cpp"
//#define export /*export*/
#endif

#endif /* !defined(_MATRIX_H_) */
```

Listing A.23: MunkreMatrix.h

Appendix B

Appendix Parameters

```
xWidth = 70; // mm of Cell
yWidth = 70; // mm of Cell
stepSizeDraw = 3; // Draw Step size
robotRotationDeviation.x=0.02f; //Deviation of Rotation
robotRotationDeviation.y=0.08f; // Deviation of Rotation
maxProbThreshold = 0.1f; // Threshold below maxProb of HeatMap totake in
    account
```

Listing B.1: HMball.cfg

```
xWidth = 70; // mm of Cell
yWidth = 70; // mm of Cell
stepSizeDraw = 3; // Draw Step size
robotRotationDeviation.x=0.02f; //Deviation of Rotation
robotRotationDeviation.y=0.08f; // Deviation of Rotation
maxProbThreshold = 0.1f; // Threshold below maxProb of HeatMap totake in
    account
NumOfRobots = 4; // Max Number of Enemy Robots
enemyteamRed = true; // Is Enemy Team red?
```

Listing B.2: HMenemies.cfg

```
bufferSize = 5; // number of frames stored in the buffer
numberOfIterations = 10; // how manytimes the algorithm goes throug the buffer
numberOFCandidateIterations = 500; // how often a new candidate is selected
    within the same frame
candidateIterationsAsFraction = 0.2; // how often a new candidate is selected
    within the same frame, as a fraction of the total number of candidates
frameStep = 10; // every frameStep-th frame is taken for the CRF
getCandidateMode = 3;
candidateThreshold = 0.3;
alphaUnary = 300.0f;
alphaX = 1.0f;
alphaY = 1.0f;
alphaRot = 1.0f;
betaUnary = 300.0f;
betaX = 1.0f;
```

```
betaY = 1.0f;
betaRot = 1.0f;
```

Listing B.3: stateCRF.cfg

Bibliography

[1] Website for documention - aldebaran.

[2] *Baddeley, A.D.; Wilson, B. A. "Prose recall and amnesia: implications for the structure of working memory"*. 2002.

[3] B-Human. Offical website of bhuman.

[4] B-Human-Team. B-human team report and code release. 2011.

[5] Raffaello D'Andrea. Recursive estimation, 2012.

[6] John Lafferty, Andrew Mccallum, and Fernando Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data'. *http://www.cs.cmu.edu/ lafferty/pub/crf.ps.*, 18th International Conf. on Machine Learning.:282–289, 2001.

[7] Benson Limketkai, Dieter Fox, and Lin Liao. Crf-filters: Discriminative particle filters for sequential state estimation. Technical report, Department of Computer Science and Engineering University of Washington, 2007.

[8] Institute of Sciences of India. Prasanta chandra mahalanobis. on the generalised distance in statistics. *Proceedings of the National Institute of Sciences of India*, 1936.

[9] Martin Utz. Selflocalisation based on conditional random fields. 2012.