UNIVERSITY OF MANITOBA

Coordination of Simulated Soccer Agents Using Visual Field Graphs

by

Matthew Woelk

and

Benjamin Bergman

September 2010

UNIVERSITY OF MANITOBA

Coordination of Simulated Soccer Agents Using Visual Field Graphs

by

Matthew Woelk

and

Benjamin Bergman

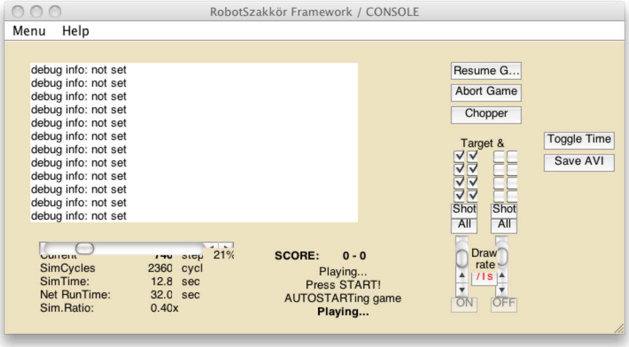
A thesis submitted in partial fulfillment for the
degree of B.Sc. Computer Engineering

in the
Faculty of Engineering
Department of Electrical and Computer Engineering

Faculty Supervisor
Dr. István Harmati, Budapest University of Technology and Economics

September 2010

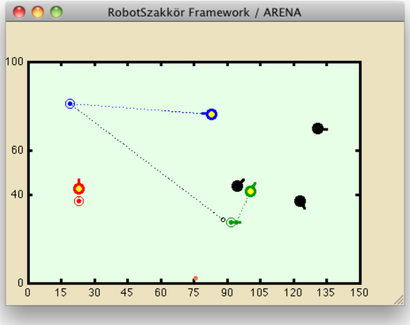
xiii + 125 = 138

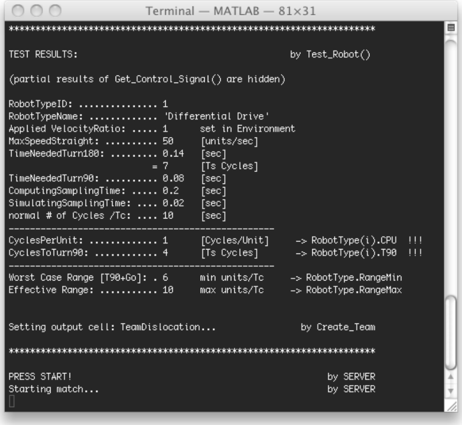


```
debug info: not set
debug info: not set
debug info: not set
debug info: not set
debug info: not set
debug info: not set
debug info: not set
debug info: not set
debug info: not set
debug info: not set
debug info: not set
debug info: not set
```

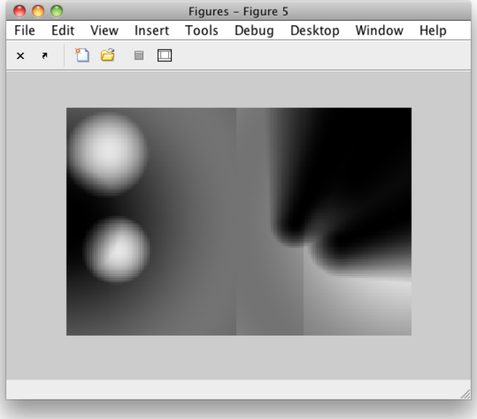
```
Current          740   step   21%
SimCycles        2360  cycl
SimTime:         12.8  sec
Net RunTime:     32.0  sec
Sim.Ratio:       0.40x
```

```
SCORE:     0 - 0
    Playing...
   Press START!
AUTOSTARTing game
    Playing...
```





```
***********************************************************************

TEST RESULTS:                                          by Test_Robot()

(partial results of Get_Control_Signal() are hidden)

RobotTypeID: ............... 1
RobotTypeName: ............. 'Differential Drive'
Applied VelocityRatio: ..... 1      set in Environment
MaxSpeedStraight: .......... 50     [units/sec]
TimeNeededTurn180: ......... 0.14   [sec]
                           = 7      [Ts Cycles]
TimeNeededTurn90: .......... 0.08   [sec]
ComputingSamplingTime: ..... 0.2    [sec]
SimulatingSamplingTime: .... 0.02   [sec]
normal # of Cycles /Tc: .... 10     [sec]
---------------------------------------------------
CyclesPerUnit: ............. 1      [Cycles/Unit]     -> RobotType(i).CPU  !!!
CyclesToTurn90: ............ 4      [Ts Cycles]       -> RobotType(i).T90  !!!
---------------------------------------------------
Worst Case Range [T90+Go]: . 6      min units/Tc     -> RobotType.RangeMin
Effective Range: ........... 10     max units/Tc     -> RobotType.RangeMax


Setting output cell: TeamDislocation...                by Create_Team

***********************************************************************

PRESS START!                                                 by SERVER
Starting match...                                            by SERVER
```



Abstract

Multi-agent environments are an area of great interest. Autonomous agents must be able to interact with each other while still being able to handle unknown influences. While any number of example fields could have been used, this project focuses on robot soccer as an example of collaborative and competitive autonomous actions.

This report describes a strategy for a simulated soccer team which is based on visual field graphs. The strategy is implemented in a simulated soccer team which competes with other, independently created teams. The team ran at full speed within the simulator, which ran at real-time speeds.

The three main components of the system are goalie positioning, determining kick target locations, and determining agent placement in the field. These three components are dynamically linked to provide accurate prediction of players' movements, which allowed for a high degree of coordination, and the ability for players to switch roles on the fly.

This project was successful in beating all of the other independently created teams. This shows that using a graphical approach to solve the problem of team coordination can be successful under real-time conditions.

Contributions

This project demonstrates an effective method of role assignment and target selection in an environment containing both system-coordinated agents as well as agents not coordinated by the system. In this specific project, that environment is a soccer field, but similar concepts could be used in any environment involving many agents that must collaborate as well as agents whose intended actions are not known by the system (eg. robots on an assembly line along side human workers).

The project was divided into a few major tasks which were either assigned to each team member from the start or picked up as needed. All aspects of the project were worked on by both team members, but many were primarily coordinated by one team mate.

Benjamin Bergman

- Study of provided soccer simulator, tactical planner, and the interface the team must connect through.
- Researched coordination strategies such as potential fields, coordination graphs, q-learning, and utile coordination.
- Developed ball prediction functions as well as supplemental functions used by the team.

Matthew Woelk

- Researched coordination strategies such as fuzzy logic systems and coordination graphs.
- Developed high level team programming including goalie behaviour and setting states.
- Developed the mirroring and shadow mapping functionality.

Common tasks

- Report authoring was shared. Each author worked primarily on sections developed by them, but both authors worked on all sections.
- Analysis of teams via simulator tournaments.

- Code structure and version control system.

The soccer simulator used was provided by Mr. Péter Gasztonyi. The tactical planner used was provided by Mr. Gergely Bódi.

Acknowledgements

We would like to thank the following people for their assistance in making this project a success:

- Dr. Witold Kinsner
- Dr. István Harmati
- Mr. J.P. Burak
- Dr. Nariman Sepehri
- Dr. Cyrus Shafai
- Mr. Gergely Bódi
- Mr. Péter Gasztonyi

Contents

List of Figures

List of Tables

Abbreviations

BME	Budapest University of Technology and Economics
C&CofMARS	**C**ontrol and **C**oordination of **M**ulti-**A**gent **R**obotics **S**ystems
FIFO	**F**irst **I**n **F**irst **O**ut
HLS	**H**igh **L**evel **S**trategy
TP	**T**actical **P**lanner

Nomenclature

Throughout this document, the word *agents* refers to any agent on the field, whether they are on the team which is controlled using the program described by this report, or are opponents to that team. The term *players* refers to any agent which is on the team described by this report, and *opponents* refers to any agent playing on a different team, in opposition to the *players*. *The team* refers to the group of *players*, and *The opposition* refers to the group of *opponents*.

In the body of this report, *italics* are often used to denote variables and `block text` is used to denote method names. Vectors are denoted as variables with an arrow printed above them such as $\vec{V}$.

The terms *Graph* and *Matrix* are used interchangeably.

List of Symbols

symbol	name	unit
θ	angle	rads
$\angle X$	angle at point X	rads
(x, y)	a point at location x,y	field units

Chapter 1

Introduction

The coordination of multiple autonomous agents in a dynamic environment is something that is seeing an increase of importance in industry. The ability to coordinate teams of robots is valuable in countless fields such as agriculture and surveillance. Every year more and more automated machines are found performing tasks that had never before seen computer control. Many of these uses of robots involve a multitude of machines working together for a common goal. The coordination of these machines such that they are able to function in a dynamic environment without interfering with each other can be a complicated problem to solve.

The area of robotic soccer is an excellent test bed for multi-agent control strategies. Robotic soccer players must work together towards a common goal without interfering with each other. They must also be able to handle the dynamic, unknown actions of the opposing team. These different aspects of robotic soccer strategies could be useful in industrial applications.

The goal of this project was to develop a high-level strategy for a robotic soccer team that could coordinate formation, role assignment, motion, and shooting the ball. This strategy was designed to defeat all other teams that were previously designed as a part of the Control and Coordination of Multi-Agent Robotics Systems program (C&CofMARS) inter-university program[1].

[1] http://ensc.sfu.ca/research/erl/EU-CANADA/site/, August 2010

1.1 Background

At the Budapest University of Technology and Economics (BME) through the C&CofMARS, previous students had built a robotic soccer simulator as well as several teams using different strategies. The aim of the project was to create a superior team to these and test it out in real robot hardware. At the start of this project, physical robotic soccer players were under development, but were not completed in time for the testing of this newly developed team. All testing was performed in software simulation.

1.2 Strategy

Using layers of matrices (graphs) and finding their high points.

1.3 Other Strategies Considered

When researching how to design a robot soccer team, many possible solutions were examined. A brief overview of each one is given in the following sections.

1.3.1 Coordination Graphs

Coordination graphs are a way of organizing when agents need to communicate with each other. By determining the states ahead of time in which agents need to communicate, agents can spend less time influenced on what other agents are doing and more on how to optimize their own actions. Upon arrival in Hungary, it was discovered that the simulator used a ”hive-mind” control of the agents, meaning that all agents are controlled through one function. While this could have been broken into individual agents, it was deemed simpler to allow a single function to coordinate all units.[1]

1.3.2 Q-learning

Q-learning is a way of dynamically learning and optimizing coordination graphs. Agents can actively decide optimal times at which to communicate, thus minimizing communication times. This strategy was very interesting, but as it was a distributed system and required many parameters to be provided by the programmer, this idea was abandoned.

1.3.3 Fuzzy Logic

Fuzzy logic systems are a way to dynamically change the output of a function, based on its many inputs. The strengths of fuzzy logic systems are being able to give a steady output with imprecise inputs, and being able to take many inputs and logically lay them out to produce a consistent and useful output. The system creates membership functions for each of its inputs, and then combines them to produce a smooth function graph. [2]

1.3.4 Potential Fields

Potential fields are a way of planning paths for agents in order to have them navigate through a set of obstacles while avoiding collisions. This method works similar to the physics of point charges in an electrical potential field. The soccer field is broken down into pixels and each pixel is assigned a charge. The target is assigned the highest potential and obstacles are assigned very low potentials. There are a few methods for achieving potential field path planning. The primary way to do this with a short path length and maximum distance from obstacles is the “roadmap” method. This involves “tagging” the pixels around obstacles with a value similar to the value of the obstacle. The further from the obstacle the pixel is, the higher it’s value will be. When the tagged pixels of two objects meet, this is the midpoint between the two obstacles and thus the farthest the agent can be from either obstacle if it were to pass between them. Once all of these mid points have been discovered, a roadmap will have been created that shows all the places between obstacles that the agent can go. This road map will be given a very high potential and the potential will increase as proximity to the target increases. In this way, the agent will tend to move onto the road map and towards the target.[3][4]

The problem with using this method was that there were so many obstacles that were constantly in motion it was deemed unnecessary to try to move through these potential fields that are changing at an extremely high rate.

The potential field method did, however, inspire the final design of this strategy. Various targets are laid out on the soccer field with high values and the opponents are mapped onto the field with low values. This mapping then allows a good target to be selected out of many. A very similar method is used to determine where to kick the ball. This strategy is explained in more detail in Chapters 7 and 8.

1.4 Summary

Having researched various concepts for robotic soccer coordination strategies, no one method became a perfect match for this project. Using some basic concepts from these strategies as well as some new ideas, the visual field method was developed. The overall system concept is described in greater detail in the following chapter.

Chapter 2

System Overview

2.1 Outline

The main `.m` file that calls all of the others is `HLS.m`, which is called every ten cycles to make decisions, store information, and tell players how to act. This code is split into seven different conceptual sections. These are shown in Figure 2.1. Each of these sections are given a brief introduction in this chapter.

2.2 Initialization

The initialization portion of the code is run first, and initializes all of the persistent variables of the system as well as creating the unchanging matrices that will be used later. Much of this code is only run once, the first time that `HLS.m` is called. The rest is called whenever a team scores a goal.

2.3 Setting States

The Setting States section sets all of the states that are used in the later sections to determine how to act. It takes the information from the field as well as information that was

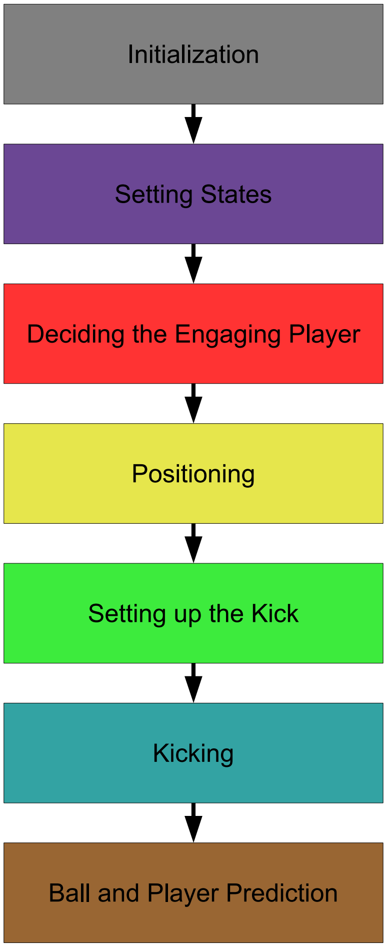


Figure 2.1: A block diagram outlining the major components of the system.

stored when `HLS.m` was previously called and abstracts them into ideas. The inputs and outputs of this section are detailed below:

Inputs:

- Player FIFOS: $Fifo\{n\}$ (discussed in Chapter 3)
- Ball Position: *Ball.Pos*
- Ball Prediction: *BallPrediction* (discussed in Chapter 5)
- Field Size: *FieldX* and *FieldY*

- Player Positions: *TeamOwn{n}.Pos(1:2)*
- Player Velocities: *TeamOwn{n}.Pos(3:4)*
- Engaging Player: *engagingPlayer*

Outputs:

- Whether a player is engaging the ball: *isPlayerEngaging*
- Whether the ball is a safe distance from the team's goal: *safeDistanceAway*
- Whether the ball has been moved from its predicted location: *BallInterrupted*
- Whether the engaging player's position has been interrupted: *PlayerInterrupted*
- If the current run of *HLS.m* is the first in the engaging player's windup to kick the ball: *firstCalculation*
- *Fifo* may be cleared for the engaging player.

2.4 Deciding the Engaging Player

The purpose of this section is to use the states that were set to decide who should chase the ball and become the *engaging player*. There is always one engaging player. This is discussed in detail in Chapter 10.

2.5 Positioning

The Positioning section is responsible for telling all of the non-engaging players where to position themselves. One main state called *hasPossession* determines whether players move up or down the field, and the Goalie has special instructions to determine where it should be placed. The goalie's positioning is discussed further in Chapter 4 while the other players' positioning is discussed in Chapter 7.

2.6 Setting up the Kick

This section does all of the calculating to decide where would be the best position to kick the ball to. This is based on the opponents' positions, the team's positions, and *firstCalculation*. This is discussed in Chapter 8.

2.7 Kicking

The Kicking section tells the engaging player to kick the ball to the location which was calculated in the Setting up the Kick section. If the player cannot kick the ball, they are told to either go to where they can intersect the ball or, if they are the goalie, to position themselves. This is discussed in Chapter 8.

2.8 Ball and Player Prediction

This section predicts where the ball and the players will be the next time `HLS.m` is called, ten cycles in the future. This is discussed in Chapter 5.

2.9 Summary

In summary, the main file called `HLS.m` is divided into seven logical sections, the last six of which are called every time that `HLS.m` is run. These sections each serve a unique role, and they build off of each other in order to direct the players.

Chapter 3

The FIFO

3.1 Purpose

In developing this soccer team, it was determined that a certain level of memory was required. In order for a player to successfully perform a kick, there are usually more required motions than can be sent to the simulator. The *FIFO* was implemented as a queue of future moves. When a string of successive moves were generated, they were stored in the *FIFO* in a first-in-first-out style so that the first move in the list is the next one to be executed.

3.2 Implementation

The *FIFO* is constructed much in the same way as the *ControlSignal*. In fact, they are mostly interchangeable. The primary difference between the two is the number of signals stored in each. The *FIFO* can contain any number of signals. The *ControlSignal*, on the other hand, will always contain the number of signals requested by the simulator, a number that stays constant throughout the simulation. Each player of the team has a *FIFO* assigned to them and stored in a persistent cell array in `HLS.m`.

3.3 `isPlayerEngaging`

When the strategy needs to determine the next cycle's set of actions, the availability of the agents is needed. A player's availability can be found by looking at the *FIFO* of that player. At the start of each cycle, the strategy checks to see if the player who was previously trying to engage the ball still has actions queued up in it's *FIFO*. This check is performed using the function `isKicking.m` and the result is stored in the boolean variable *isPlayerEngaging*. In order to keep this system working smoothly, any time a kick is interrupted (generally when the goalie needs to return to net or when it is determined that the previous action is no longer feasible due to some disturbance) the *FIFO* must be cleared. This system was used because extra boolean variables for saving the kicking state were deemed undesirable. Since the *FIFO* is only used for queuing a kick, this strategy worked well.

3.4 `Kick.m`

If it has been determined that the player's currently queued commands should be executed, the player's *FIFO* is passed into the `Kick.m` function. This function takes the player's *FIFO* and generates a control signal that is properly formatted and truncated for the simulator. A trimmed down copy of the *FIFO* with all previously executed signals removed is also returned. The $HLSTraj$ and $BallTraj$ variables for the kick are also set with this function, as discussed in Chapter 8.

3.5 Summary

The *FIFO* functionality provides a level of memory to the players that the simulator does not actively provide. It allows a complex set of actions needed to perform a kick to be retained and executed far into the future. This was useful so that the team did not need to make new decisions every single simulation cycle, making the team more efficient.

The idea of using the contents of the *FIFO* as a way of determining the kicking state of the players worked well for the team that was developed, but did not provide much flexibility to future versions. For example, it might have been nice to store complicated navigation-related control signals in the *FIFO* in addition to the kicking-related control signals if a more advanced navigation layer were added to the team.

The concept of the *FIFO* worked very well for players that were actively kicking the ball. The player that was actively playing the role of the goalie needed special consideration to allow rapid reaction to aggressive opponents. This is discussed in detail in the following chapter.

Chapter 4

Goalie

The goalie is a dynamically chosen player whose job it is to keep the opponents from being able to score on the goal. There are three states that the goalie can be in: positioning, blocking, and chasing. The `Goalie.m` file determines where the goalie should be positioned when positioning. `isBallGoingForOurGoal.m` determines whether the goalie should be in the blocking state as opposed to the positioning state with `DistanceToLine.m` determining where to go in this state. `canGetThereFirst.m` determines whether the goalie should chase the ball or choose one of the other two states.

4.1 Chasing State and `canGetThereFirst.m`

The `canGetThereFirst.m` function uses `Intersection.m` (discussed in Chapter 11) to determine whether or not the inputted player will be able to engage the ball before an opponent. In the Setting States section of `HLS.m`, if the goalie has been chosen to engage the ball, `canGetThereFirst.m` is called. If it returns true then the goalie will engage the ball. If not then another player will be chosen and the goalie will choose between the positioning and blocking states instead of the chasing state.

4.2 Blocking State and `isBallGoingForOurGoal.m`

During the Positioning section of the code, if the goalie is not the current engaging player, `isBallGoingForOurGoal.m` is used to determine whether the ball is heading toward the team's goal. If it is, then the goalie will move to the spot where it can intersect the ball's path. This position is calculated using `DistanceToLine.m`, which is discussed at length in Section 7.3.

To calculate whether the ball is headed toward the team's goal, `isBallGoingForOurGoal.m` first calculates the slope of the ball's velocity vectors as follows:

$$slope = \frac{BV_y}{BV_x} \tag{4.1}$$

where BV_y is the ball's vertical velocity, and BV_x it's horizontal velocity. From there the intersection with the end of the field is calculated using

$$wallIntersection = slope * (-B_x) + B_y \tag{4.2}$$

where B_x and B_y are the ball's x and y positions, respectively. If this intersection is between the two goal posts (whose locations are environmental constants), then the function returns *true*.

4.3 Positioning State and `Goalie.m`

If no other state has been chosen, the goalie goes into its positioning state. The `Goalie.m` file determines where to position the goalie in this state. To do this, it uses the ball's current position, as well as the opponents' positions.

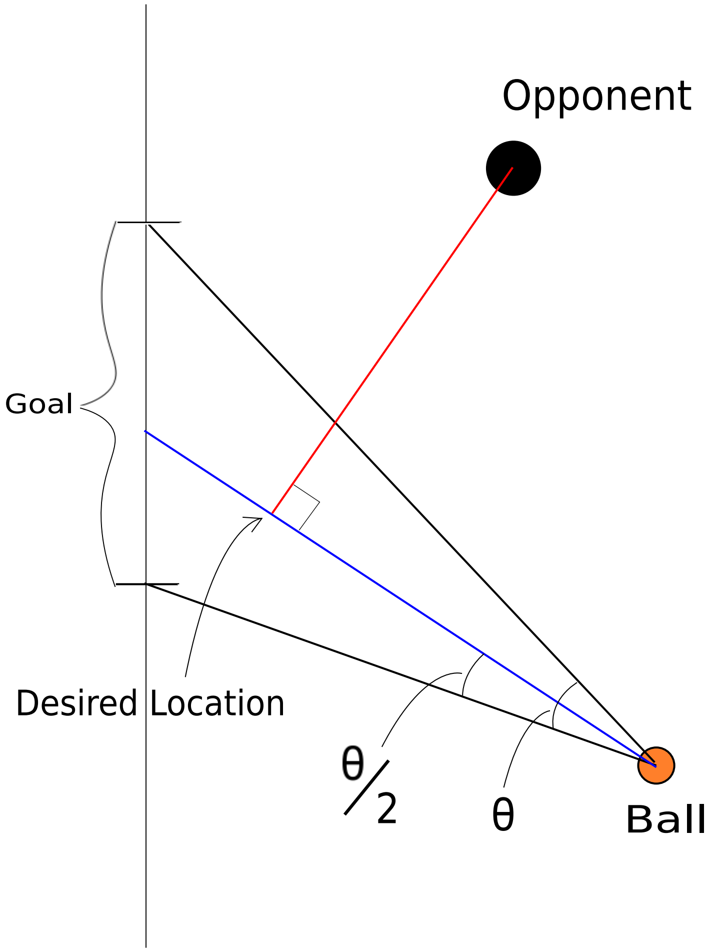


FIGURE 4.1: The `Goalie.m` calculation process and objects.

4.3.1 Overview of `Goalie.m`

Figure 4.1 shows the calculation process that `Goalie.m` goes through in determining where the goalie should be positioned. The goalie will always stay on the blue line, which intersects the ball, and is in the center of the angle θ between the two lines which intersect the ball and the goalposts. Lines perpendicular to the blue line and intersecting each of the opponents are calculated. Whichever of these lines intersects with the blue line closest to the net is the one used in the calculation. The goalie will be sent to the location on the blue line which is either 1/3 the distance between the goal and the ball, or where the opponent's intersection is; whichever of these two options is closest to the net.

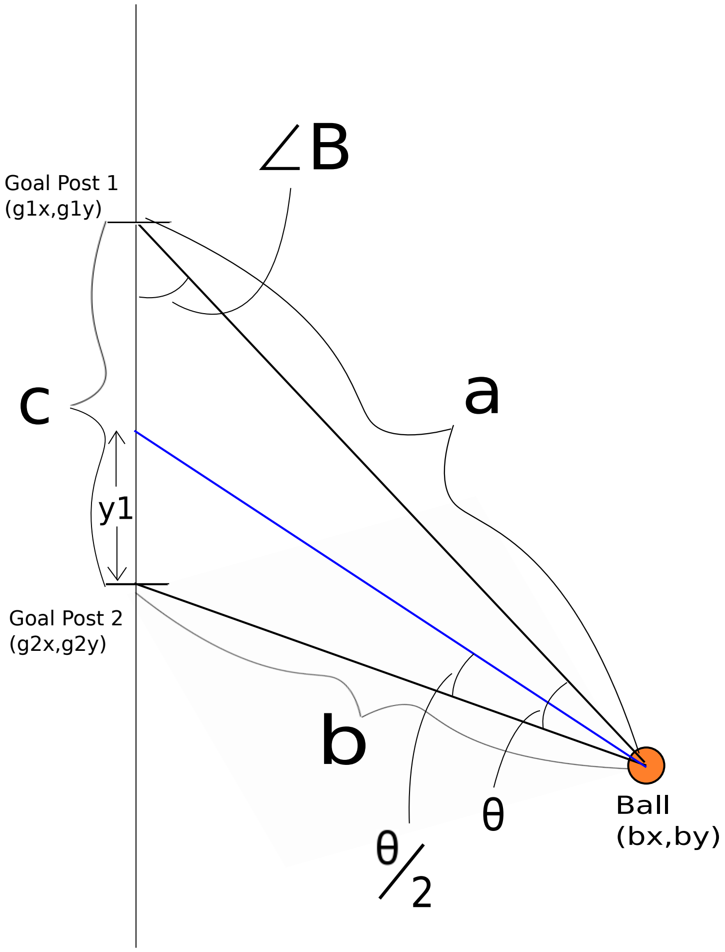


FIGURE 4.2: `Goalie.m` calculations diagram.

4.3.2 Calculations

To calculate angle θ, a triangle is formed by the two goal posts and the ball. Figure 4.2 shows this triangle, which is formed by the points $GoalPost1$, $GoalPost2$, and $Ball$. Length c is given as a constant in the simulator, while lengths a and b are calculated using Equation 4.3.

$$length = \sqrt{(x2 - x1)^2 + (y2 - y1)^2} \tag{4.3}$$

where $(x1, y1)$ and $(x2, y2)$ are the points between the length. Cos law is then used to determine θ, shown in Equation 4.4.

$$\theta = acos(\frac{a^2 + b^2 - c^2}{2ab}) \tag{4.4}$$

where *acos* is the arccosine function. $\angle B$ is calculated similarly with Equation 4.5.

$$\angle B = acos(\frac{a^2 + c^2 - b^2}{2ac}) \tag{4.5}$$

Sin law is then used in Equation 4.6 to determine $y1$.

$$y1 = \frac{sin(\theta/2) * b}{sin(\angle B)} \tag{4.6}$$

Now we have the blue line described by the points *Ball* and the point which is $y1$ above *GoalPost*2. To find which opponent's perpendicular intersection with the blue line is closest to the team's goal, each opponent's position is given to `DistanceToLine.m` along with the two points which describe the blue line, and the returned distances are compared to find the minimum. This final distance is compared with 1/3 of the distance between the goal and *Ball*, and the minimum value is used on the blue line as the final location.

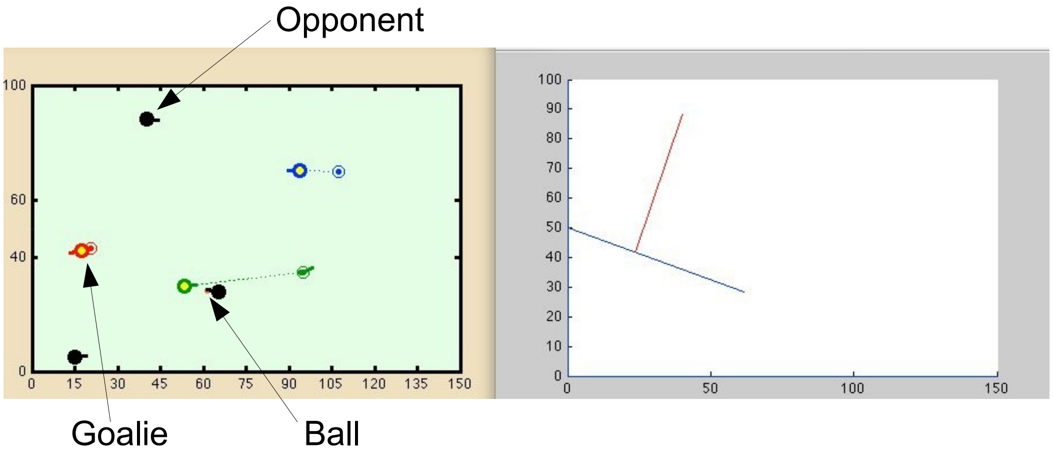


Figure 4.3: The `Goalie.m` generated output based on a field position.

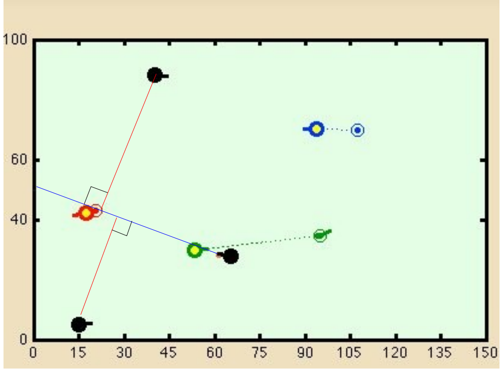


FIGURE 4.4: An overlay of `Goalie.m`'s calculations onto the field.

4.3.3 Examples and Screenshots

Figure 4.3 shows an example of a situation where the goalie's target location has been calculated. The left side is a labeled field, and the right side is a generated graph of the opponent's line, and the goalie's line. Figure 4.4 shows an overlay of a representation of the calculation over a field situation. You can see in this diagram that each player's intention is shown as a lighter circle of their same colour, and you can see that the red goalie is going to move to the calculated position.

4.4 Determining the Goalie and `ClosestToNet.m`

Since the goalie is a dynamically chosen player, it is necessary to reevaluate who it should be. This happens when the current goalie is chosen to be the engaging player, as shown in Figure 10.1 on Page 46. The function used to determine who should tend the net is called `ClosestToNet.m`.

`ClosestToNet.m` is a simple function which uses Equation 4.7 on each player, with chosen player being the one with the lowest *distance*.

$$distance = \sqrt{x^2 + (y - \left(\frac{FieldY}{2}\right)^2} \tag{4.7}$$

where (x, y) is the player's location, and $FieldY$ is an environmental variable, given by the simulator which stores the height of the field.

4.5 Summary

The goalie's three states, chasing, blocking, and positioning, do an exceptional job of preventing goals on the team's net. The chasing state lets the goalie intercept the ball if it is able to do so, the blocking state gives the goalie the ability to intercept shots on net, and the positioning state permits the goalie to get into a good position for blocking shots. Since the goalie is dynamically assigned, the net is always well protected from the opponents. It allows for goalies who have left the net, with the aim of intercepting the ball or otherwise drawn out, to be replaced by nearby teammates. This dynamic process is determined greatly by the ball and kick prediction, which is discussed in the following chapter, which is integral in determining whether it is safe for the goalie to chase the ball or not.

Chapter 5

Ball and Kick Prediction

5.1 Overview

One major aspect of this team strategy is the ability to predict the future state of the field and perform actions which may span several calls of `HLS.m`. One problem encountered early in testing was that these longer actions, often calculated using the provided tactical planner, were not aware of future predictions and would not react to unexpected changes in the state of the field. Two major instances of this latter failure were:

- The ball becomes disrupted resulting in the player performing an action that often did not even contact the ball
- The player becomes disrupted resulting in the player performing an action in a different position than was originally calculated

In order to overcome the above problems, a system for predicting future states of the field and verifying these predictions was required. The primary information used for these predictions is the ball's position and velocity as well as the queue of moves the player needs to execute in order for there to be a successful kick.

5.2 Underlying methods

5.2.1 Ball Prediction

The function `BallPrediction.m` was developed in order to get a highly accurate expectation of the ball's future position and velocity. The inputs to the function are the ball's current position and velocity, as well as the number of cycles into the future that we want to predict. The function then returns a matrix containing all of the expected ball positions and velocities.

The ball is expected to slow by a certain damping factor, *qDamp*, each cycle. This damping factor is provided by the simulator. The expected velocity of the ball in the next cycle can be found by:

$$\vec{V}_{next} = \vec{V}_{current} * qDamp \tag{5.1}$$

where $\vec{V}$ is the velocity vector. The expected position of the ball in the next cycle can be found with

$$\vec{P}_{next} = \vec{P}_{current} + \vec{V}_{current} \tag{5.2}$$

where $\vec{P}$ is the position vector.

Once the expected next position and velocity is known, it must be checked to see if this would cause a collision with a wall. The ball's position is determined at the center of the ball, therefor the radius of the ball is critical when calculating collisions with the walls. Because there are four walls in the field, there are four conditions that must be checked.

If the ball's predicted x position is *less* than 0, the leftmost x coordinate of the field, plus the balls radius, then:

$$\begin{cases} x_{mirror} = 2 * BallRadius - x \\ v_{x\_mirror} = -v_x \ . \end{cases} \tag{5.3}$$

If the ball's predicted x position is *greater* than $FieldX - BallRadius$, the fields rightmost x position minus the radius of the ball, then:

$$\begin{cases} x_{mirror} = 2 * (FieldX - BallRadius) - x \\ v_{x\_mirror} = -v_x \ . \end{cases} \tag{5.4}$$

Similarly, if the ball's predicted y position is less than $0 + BallRadius$, then:

$$\begin{cases} y_{mirror} = 2 * BallRadius - y \\ v_{y\_mirror} = -v_y \end{cases} \tag{5.5}$$

and if the y position is greater than $FieldY - BallRadius$,

$$\begin{cases} y_{mirror} = 2 * (FieldY - BallRadius) - y \\ v_{y\_mirror} = -v_y \ . \end{cases} \tag{5.6}$$

These checks are performed for each cycle that has been requested through the provided input. Each iteration uses the previously predicted value to determine the next.

During the testing of this function, it was discovered that the true maximum y coordinate of the field was not exactly equal to the $FieldY$ variable provided by the simulator. Testing showed that the ball rebounded as if the true maximum y value of the field were $FieldY - 0.3$. As such, the `BallPrediction.m` function has been constructed to take this anomaly into account.

The primary usage of this function is to determine if a player should continue a kick they are currently executing. The function that generates the control signals for the kick do not take into account rebounds off of other agents and so these rebounds have also been omitted from the `BallPrediction.m` function.

5.2.2 Kick Prediction

The function `PlayerPrediction.m` was developed to give a highly accurate prediction of the player's future coordinates. This function uses the player's current position and orientation, their queue of moves, the number of cycles that are to be predicted, and the number of the next cycle to be executed.

As the only way of determining where a player will be in the future is by looking at the control signal, the $FIFO$ of the player, an extended version of the standard control signal, is used for prediction purposes. The simulator is designed for use with robots utilizing differential drive systems. As such, the control signals for each cycle contain either an amount to turn or an amount to drive in a straight line. Currently the simulator is not designed to handle driving in an arc as would occur if one wheel turns at a different speed than the other.

For each cycle for which a prediction has been requested, the control signal uses the next control signal in the FIFO to determine the next cycle's position. If the control signal is a driving signal, it is a simple matter of multiplying the player's orientation, a unit vector, by the length of the move. The result is as follows, where P is the position vector, O is the orientation vector, and m is the amount the player is to move during that cycle:

$$\vec{P}_{next} = \vec{P} + \vec{O} * m \,. \tag{5.7}$$

If the control signal is a turning signal, the orientation of the player is converted from a vector to an angle, θ. Function 5.8 shows how the Cartesian vector is converted to an angle in the range of $[-\pi, \pi]$[5]:

$$\theta = \begin{cases} \arctan(\frac{y}{x}) & \text{if } x > 0 \\ \arctan(\frac{y}{x}) + \pi & \text{if } x < 0 \text{ and } y \geq 0 \\ \arctan(\frac{y}{x}) - \pi & \text{if } x < 0 \text{ and } y < 0 \\ \frac{\pi}{2} & \text{if } x = 0 \text{ and } y > 0 \\ -\frac{\pi}{2} & \text{if } x = 0 \text{ and } y < 0 \\ 0 & \text{if } x = 0 \text{ and } y = 0\,. \end{cases} \tag{5.8}$$

Once the angle of the player has been obtained, the angle listed in the $FIFO$ is added to produce the new predicted angle:

$$\theta_{new} = \theta + \phi \tag{5.9}$$

where ϕ is the angle defined in the $FIFO$.

The resulting angle is then converted back into an orientation using

$$\vec{O}_{new} = [cos(\theta_{new}), sin(\theta_{new})]\,. \tag{5.10}$$

After these calculations, the new position and orientation information is stored in the prediction matrix. Each subsequent calculation uses the information predicted in the previous iteration.

In some cases, the number of cycles requested is more than are able to be predicted due to the length of the provided $FIFO$. In order to account for this, it is assumed that the player will stop moving after the signals in their $FIFO$ are executed. The remainder of the prediction matrix will therefor return the last predicted position and orientation.

The player's prediction does not consider the player's collisions with the walls or other agents as these collisions are not considered in the tactical planner.

5.3 Usage of Predictions

At the end of each `HLS.m` call, the `BallPrediction.m` and `PlayerPrediction.m` functions are called and the results are stored in a persistent variable. At the start of each `HLS.m` call, the predicted values are compared to the true values. If the predicted values and the true values disagree by more than a certain threshold, it is assumed that there has been some sort of disruption. If the ball is not where it was predicted to be, then another agent has likely made contact with it and sent it off course. If the player is not where predicted, it has likely collided with a wall or another agent. If either of these cases occur, there is no point in having the player continue their current kick and so it is abandoned. At this point, a new kick is attempted and if this fails, alternative measures are taken.

5.4 Summary

So the ball and kick predictions are calculated in order to determine whether an opponent has had contact with the ball or one of our players. They are calculated by utilizing the fact that the simulator does not contain any randomization, and performs in a predictable manner. If our player prediction for the current time is not accurate to the current situation, then we know our player has run into another, and we know to halt a kick. Similarly, both the player and ball predictions help to determine whether we have possession, as described in the following chapter.

Chapter 6

HasPossession

The entire team is influenced by two states which are whether or not the team has possession of the ball. This is stored in the boolean variable *hasPossession*. This state is a very abstract idea, with a better description being whether or not the team is in control of the current situation.

hasPossession is stored in a persistent variable, so it remains unchanged unless it is explicitly changed (as opposed to being re-initialized every time `HLS.m` is run). The conditions in which the state is changed are described in the following sections of this report.

6.1 Unpredicted Contact

If the ball is in a position other than the prediction stored in the *BallPrediction* variable, then the ball has had contact with an opponent. The process of calculating `BallPrediction.m` is described in detail in Chapter 5. This means that the team has lost control of the ball, and so $hasPossession = false$.

6.2 Failed Pass

If a player is engaging the ball, `canGetThereFirst.m` (4.1) is called to see if they can reach the ball before any opponent. If they cannot, the pass has failed and so $hasPossession = false$.

6.3 Failed Kick

When a player is attempting a kick, `canKick.m` (8.2) is called to determine whether the engaging player will be able to intercept the ball and kick it to the desired location at a speed which is within the target range. If not then there is no way to control the ball, so $hasPossession = false$.

6.4 Safe Distance Away

There are only two situations in which $hasPossession = true$. The first is when the ball is on the opponent's half of the field, and moving away from the team's goal. In this situation, the variable $safeDistanceAway = true$ and $hasPossession = true$.

6.5 Continued Engagement

The second situation is when one player was engaging the ball during the previous `HLS.m` call without any ball or player interruptions (referred to in Chapter (5)). If that player is continuing to engage the ball without any of these interruptions then $hasPossession = true$, and if not, $hasPossession = false$.

6.6 Summary

There are five situations in which the *hasPossession* boolean variable is changed. They are when the ball prediction is different than where the ball currently is, when a player's pass fails, a player's kick fails, the ball is a safe distance away from the goal, and when contact between two teammates is successful.

The *hasPossession* variable affects where the players are sent, as described in the following chapter. If the team has possession, then the team will move forward, if not, then they will move backward.

Chapter 7

Positioning

The main method used to determine where players should be positioned is by creating different matrices using the information gathered in the Setting States, as well as information from the simulator. Each of these matrices is the size of the field, and represents a different layer of information. An entry-wise, or "Hadamard"[1] product of these matrices is taken to produce a final matrix, and the position of the highest value in that matrix is used as the location to send the player. The following sections go in depth about each of the multiplied matrices.

7.1 Static Field Mapping

The base layer is the static field matrix. Figure 7.2 shows this matrix when *hasPossession* is *true*. The matrix is flipped left-to-right when *hasPossession* (as described in Chapter 6) is $false$. The purpose of this matrix is to guide players away from their goal and toward the opponents', or to go back to their goal to defend, depending on the circumstances.

The equation used to generate this matrix is as follows. First, a matrix is created which contains a value for each x and y location on the field, which is the distance between that point and the closest goal, whether that be the team's goal, or the opponents' using Equation 7.1, for each point.

[1] July 5, 2010 - `http://planetmath.org/encyclopedia/HadamardProduct.html`

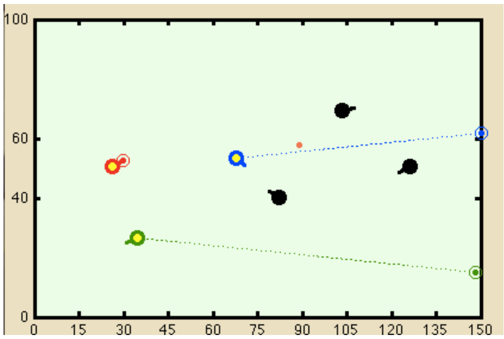


FIGURE 7.1: The current state of the field where the blue player's future position is being determined.

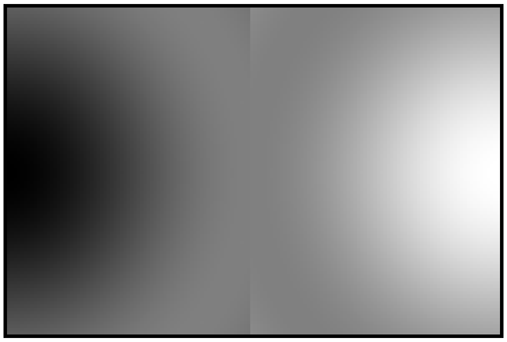


FIGURE 7.2: Static field mapping when *hasPossession* is *true*.

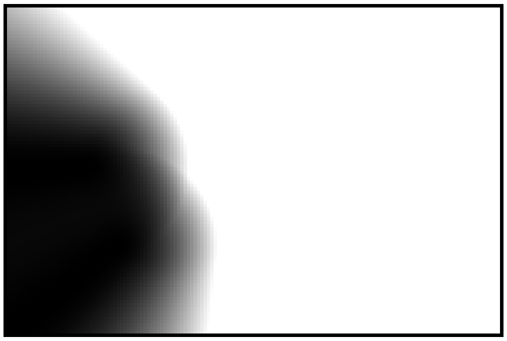


FIGURE 7.3: Negative areas determined by the location of the two other players.

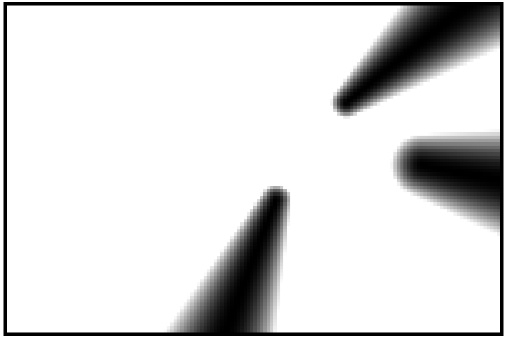


FIGURE 7.4: Shadows determined by the location of opponents in relation to the ball.

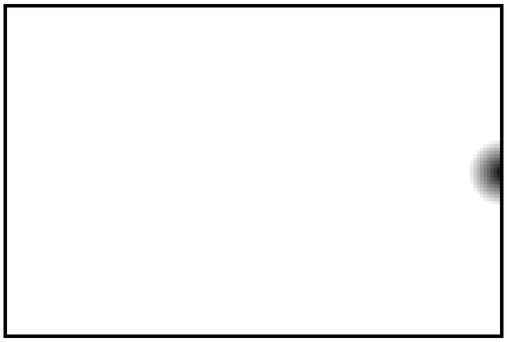


FIGURE 7.5: Mapping to prevent camping.

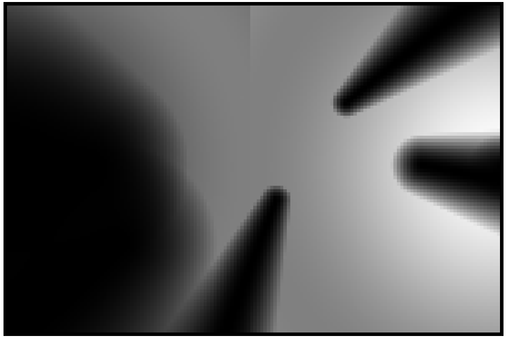


FIGURE 7.6: The result of a entry-wise product of the previous four matrices. The highest value in this matrix determines where the player will go.

$$distance = min \left[\sqrt{x^2 + (y - (\frac{FieldY}{2})^2)} \ , \sqrt{(x - FieldX)^2 + (y - (\frac{FieldY}{2})^2)} \right] \tag{7.1}$$

where $min[a, b]$ takes the minimum value between a and b, and $FieldX$ and $FieldY$ are environmental variables which are the size of the field's width and height, respectively. From there, the result matrix is calculated by running Equation 7.2 for each set of coordinates on the left half of the field, and Equation 7.3 for each on the right.

$$matrix_L = \frac{sin\left(\frac{distance. * pi}{\frac{FieldX}{2}} - \frac{pi}{2}\right) + 1}{4} \tag{7.2}$$

$$matrix_R = \frac{sin\left(\frac{\left(\frac{FieldX}{2} - distance\right) . * pi}{\frac{FieldX}{2}} - \frac{pi}{2}\right) + 1}{4} + 0.5 \tag{7.3}$$

This produces the gradual sinusoidal gradient in the static field matrix, with values from 0 to 1. In addition, all of the values in the rightmost column are given a solid value of 1. This is because this field matrix is also used for kicking, and a value of 1 supersedes any other, so the shooter will shoot for the net, as opposed to make a pass, when the opponent's goal is unobstructed.

7.2 Player Mapping

The matrix in Figure 7.3 is a matrix whose purpose is to keep our players from all going to the same position. This matrix maps a gradient circle around each player other than the current player. The circles are created using Equation 7.4 on each position of the field for each player, where the min of the calculations for each player is taken. The darkness of any point has a parabolic relationship to the distance from that point to the player.

$$value = max[1 - \frac{1}{30^2} * distance^2, 0] \tag{7.4}$$

where $max[a, b]$ is the highest value between a and b, and $value$ is the value at point (x, y), and $distance$ is the distance from that point to the player. The whole resulting matrix is subtracted from 1 before multiplied by the rest of the matrices to invert it so that it resembles Figure 7.3.

7.3 Shadow Mapping

The purpose of the matrix in Figure 7.4 is to disallow the players to move near opponents, or behind opponents in relation to the ball's position. This is to limit the ability of the opponents to intercept the ball if the engaging player were to pass it to the player.

Figure 7.7 shows the objects used to calculate this matrix where $p1$ to $p3$ are three example points whose value in the shadow matrix is being determined, and $d1$ to $d3$ are the shortest distances between the opponent and a line segment which goes from that point to the ball.

Further away opponents have larger-radius shadows, because by the time the ball could get to them, they would be able to move a further distance. Their future position could be anywhere in that larger radius.

For every point on the field the distance from the ball to the aforementioned line segment is found and stored in the *distance* variable using the `DistanceToLine2.m` function. Then a field-size matrix for each opponent is created using Equation 7.5.

$$matrix_{xy} = max[1 - \frac{1}{(k * sin(0.5))^2} * distance_{xy}^2, 0] \tag{7.5}$$

where k is the distance between the ball and the opponent. The matrices for each opponent are then subtracted from 1 to invert them, and then are multiplied together to produce the final shadow matrix.

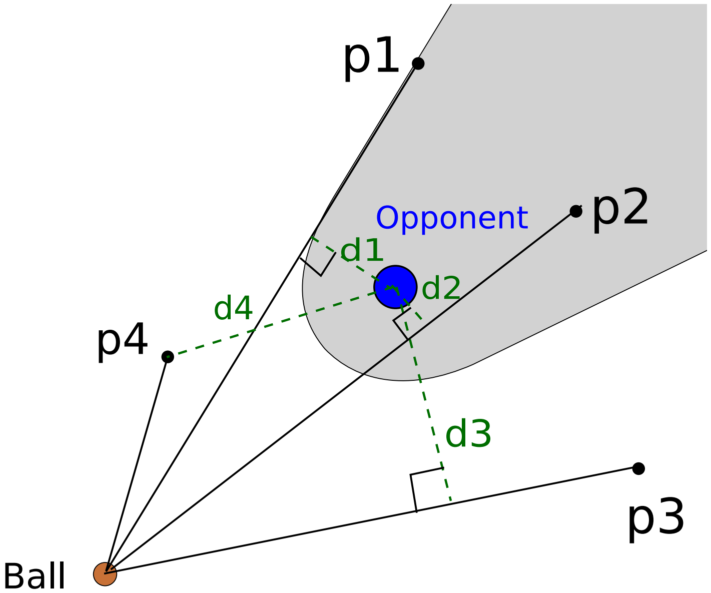


Figure 7.7: Diagram showing the objects used in creating shadow matrices.

7.4 Anti-Camping Mapping

Figure 7.5's matrix has the purpose of preventing the players from sitting inside the opponent's net while waiting for the ball. Before this matrix was implemented, there was a problem where the players would block the opponents' net while waiting for a pass. This would result in the players blocking their teammates' shots. This matrix fixed this problem.

The matrix is created with the `GraphDontCamp.m` function. For each point in the field (x, y), Equation 7.6 is run, with the results stored in the resulting matrix.

$$matrix_{xy} = min\left[1, \frac{\sqrt{(x - FieldX)^2 + (y - \left(\frac{FieldY}{2}\right)^2}}{10}\right] \tag{7.6}$$

where $FieldX$ and $FieldY$ are environmental constants given by the simulator.

7.5 Summary

In summary, the positioning of the team is done by using the *hasPossession* state which has been previously set while using a combination of four different mappings of the field. These mappings are as follows: A static field mapping which is flipped right-to-left when *hasPossession* is switched, a player mapping which prevents the players from standing near each other, a shadow mapping which prevents the players from standing near or behind opponents, and an anti-camping mapping which is to keep the players from sitting in the opponent's goal, or their own goal when *hasPossession* is switched.

This positioning determines where the players go, and this information is used to help determine where the kicker should kick the ball. This is described in the following chapter of this report.

Chapter 8

Kicking

8.1 Overview

The ability of a player to accurately kick the ball is critical to the success of this team. The tactical planner (TP) provided by Gergely Bódi provided a strong base for this strategy. The tactical planner function `TP_Kick.m` uses the player's position, the ball's position, the target of the kick, and the minimum and maximum desired speed of the kick and returns a set of control signals that the player can use to perform the desired kick. This function also sets the $HLSTraj$ and $BallTraj$ variables that the simulator uses to display trajectory information, shown in Figure 8.1.

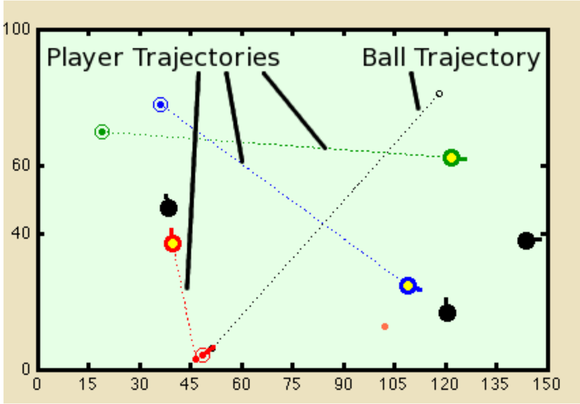


FIGURE 8.1: Dotted lines show the players' destinations and where they intend to kick the ball

This functionality is useful if the returned kick is always to be followed through and only one agent will attempt a kick, but in this team that is not always the case. In order to provide better functionality, a set of wrapper functions were created for the tactical planner. These functions allow the tactical planner to be used as a check to see if a kick can be performed without necessarily performing the kick.

8.2 The `canKick.m` Function

In order to provide the desired functionality, the `canKick.m` function was developed. The primary purpose of this function is to determine if a kick is possible under the current conditions. If so, it will also return the control signals required to perform the kick as well as the old trajectory information used for visual output. The old trajectory information is returned in case this kick is ignored and the previous one is selected. Later, if this kick is chosen to be performed, the $FIFO$ containing the control signals and the trajectory information is sent to the `Kick.m` function. This function formats the $FIFO$ into the proper control signal required by the simulator, removes previously executed signals from the $FIFO$, and sets the trajectory information for the kicking player and the ball if it is provided.

8.3 Performing a Kick

Each time `HLS.m` is called, the team tries to assign a player to kick the ball. Often a player is already in the middle of performing a kick. Before performing a kick, any existing kick that has been started is checked for interruption as discussed in Chapter 5 on page 19. If a player is already engaging the ball and it has not been inturrupted, then the $isPlayerEngaging$ state is set to true.

If a player is already engaging the ball, the kick will be performed if the completion of the kick is less than 31 cycles in the future, as can be seen in Figure 8.2. If there are more than 31 cycles before the completion of the kick, a new kick is calculated. This is done so that a kick that is near completion can be followed through properly, but a kick that is not very

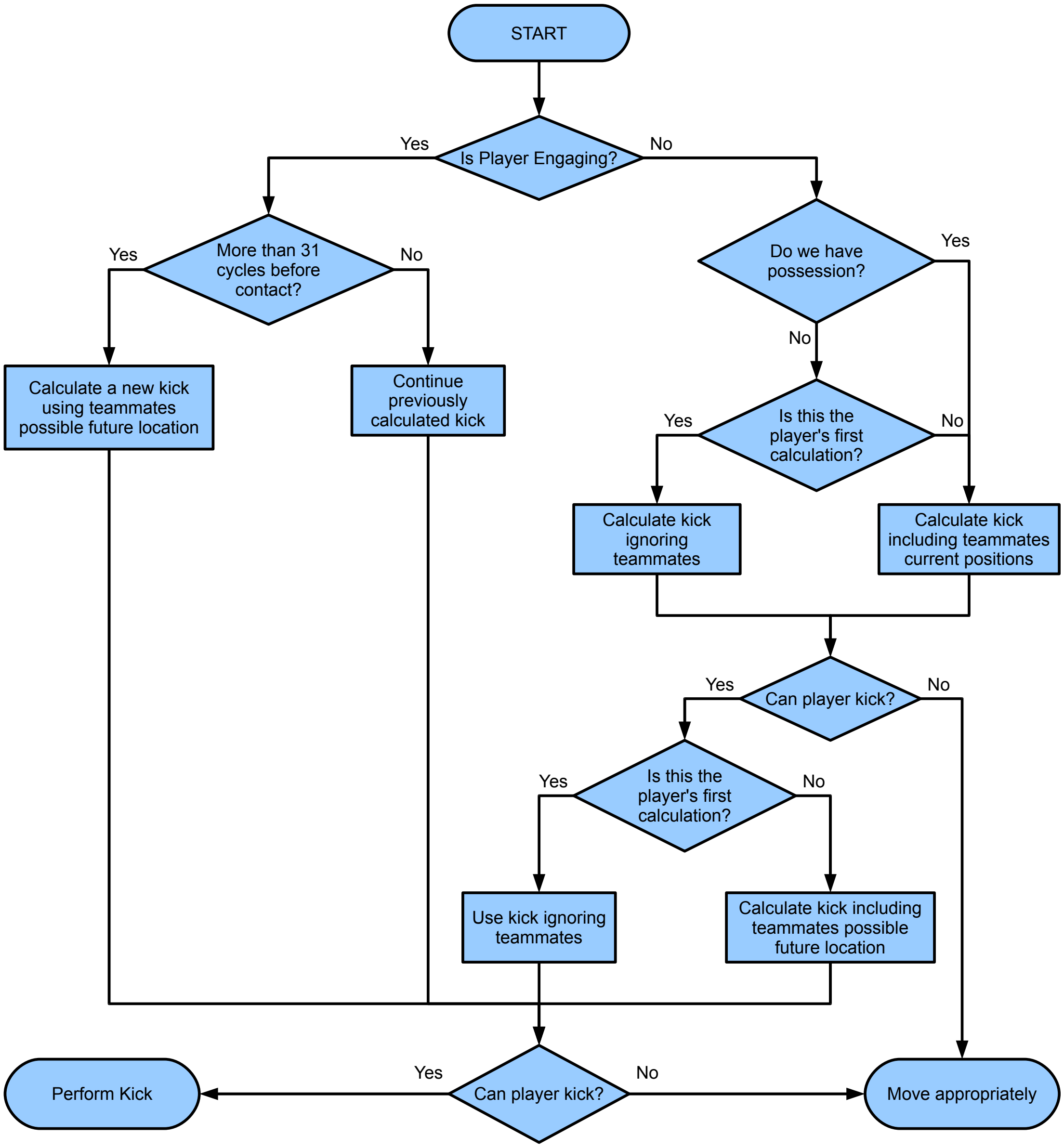

START
Is Player Engaging?
Yes
No
More than 31 cycles before contact?
Yes
No
Calculate a new kick using teammates possible future location
Continue previously calculated kick
Do we have possession?
Yes
No
Is this the player's first calculation?
Yes
No
Calculate kick ignoring teammates
Calculate kick including teammates current positions
Can player kick?
Yes
No
Is this the player's first calculation?
Yes
No
Use kick ignoring teammates
Calculate kick including teammates possible future location
Perform Kick
Yes
Can player kick?
No
Move appropriately

FIGURE 8.2: A flowchart depicting the creation of a 'kick'

near completion can be refined. Refinement may be necessary as the prediction used for the original kick may or may not have come true.

If there is not a player currently engaging the ball, a new kick is setup for the engaging player. How the kick is created is dependant on whether or not we have possession as well as whether or not this is the engaging player's first attempt in a string of kick calculations. If the team does not have possession and this is the player's first calculation, a kick is attempted that ignores the positions of all teammates. A goalie who is trying to clear the ball from the net will usually not have much time to perform a kick. The goalie generally wants to avoid passing to any teammates who are also near their net. This first calculation gives prioritization to the clearing of the ball. If this is not the engaging player's first calculation, the teammates' current positions are included.

This initial kick calculation is used to determine if a kick from the player's current position is possible. If the `canKick.m` function determines that a kick can be performed, a true kick is performed. If this is the player's first calculation, the kick that ignores the teammates is used, but if it is not the first calculation, a new kick is generated. This new kick uses the contact point from the previous kick to make a prediction of the state of the field. A quick calculation is run to check where teammates will be able to receive a pass. Their current trajectories are passed to the `IntersectPoints.m` function, discussed in Chapter 12 on page 51, which will return the locations where the engaging player can kick the ball to such that the teammates will easily intercept it. These position are then used in the creation of a new kick target matrix. These locations are all valid at different times, but they will each be valid when then ball will get to them. This allows for highly accurate leading passes.

After this kick calculation is performed, the system checks to see if the second, more intelligent kick can be performed. If so, the kick is executed as described in Section 3.4. If the `canKick.m` function deems a kick impossible, the system attempts to move the engaging player to an appropriate position.

8.4 Positioning When Player Cannot Kick

When the engaging player is unable to perform an intelligent kick, the player is moved to an advantageous position. The positioning process has two different branches based on whether or not the engaging player is also the current goalie, as shown in Figure 8.3.

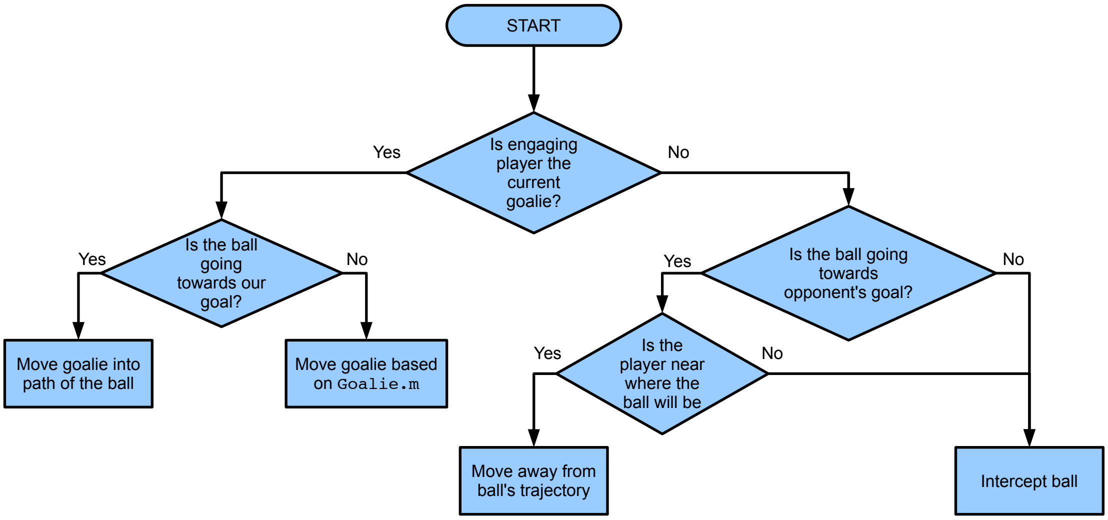

START
Is engaging player the current goalie?
Yes
No
Is the ball going towards our goal?
Yes
No
Move goalie into path of the ball
Move goalie based on Goalie.m
Is the ball going towards opponent's goal?
Yes
No
Is the player near where the ball will be
Yes
No
Move away from ball's trajectory
Intercept ball

FIGURE 8.3: A flowchart depicting the positioning of the engaging player if a kick cannot be performed

If the engaging player is the goalie, then the player should try to defend the goal. If the ball is already moving towards the goal, the goalie is sent to intercept the ball's trajectory as fast as possible. The `isBallGoingForOurGoal.m` function checks to see if the ball is moving towards the goal and will return the location in the net that the ball is moving towards. Using the ball's position and the position in the net where the ball is moving towards, an imaginary line is drawn. The `DistanctToLine.m` function is then used to find the point on the line nearest to the goalie. The goalie will then move to this point.

If the ball is not moving towards the goal, the goalie will return to following the motions dictated by the `Goalie.m` function as discussed in Chapter 4 on page 12.

When the engaging player is not the current goalie, a similar set of actions is performed. If the ball is moving towards the opponent's goal, determined by the `isBallGoingForGoal.m` function, the engaging player should avoiding blocking a potential goal. If the player is near to where the ball is predicted to be 15[1] cycles in the future, the player will move away from the ball's trajectory. Otherwise, the player will attempt to position itself in such a way that it might be able to make a kick the next time `HLS.m` is run.

8.5 Calculating Kicking Location

Determining where players should kick the ball is done very similarly to Positioning (as detailed in chapter 7). The only different is that the Player Mapping (from Section 7.2) is inverted because we want to kick toward our players instead of avoid them, and the Anti-Camping Mapping (from Section 7.4) is flipped right-to-left so that our players never kick the ball into our own net. The only addition is a slight dimming which happens to the section of the field between the engaging player and the player's goal. This helps to decrease the chance of the player passing backwards. It is a slight heuristic addition that was made due to a lack of shots being taken on the opponent's goal. An example of the final outputted matrix is in Figure 8.4.

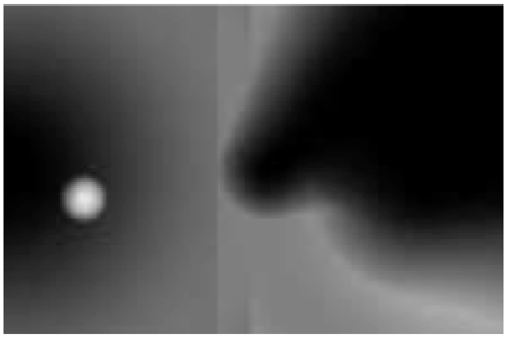


Figure 8.4: An example matrix showing how the player chooses a kicking target

8.6 Summary

Despite being provided a very good tactical planner, developing a highly accurate kicking system still proved challenging in the dynamic environment of a competitive soccer field.

[1] 15 cycles was chosen because there are 10 cycles between calls to `HLS.m` and it may take some additional time for an agent to rotate before it can drive out of the way.

Though the original concept for performing kicks was quite simple, the one that actually proved itself to perform well involved many small heuristics which added undesired complexity to the system.

One improvement to the kicking part of the team is how kicking is performed. Wrapping the provided tactical planner worked, but it was clumsy feeling. Ideally the tactical planner would be re-written to provide a more generic interface to the team. Even though a working wrapper was developed, this wrapper could also be improved. One such improvement would be to have `canKick.m` return all the information required to perform a kick without changing anything such as *HLSTraj* or *BallTraj*. This would make ignoring suggested kicks easier and the code much easier to follow, though perhaps at a very minor cost in execution speed.

While developing the kicking system, it was noticed that players were not taking advantage of rebounds off of walls. A new matrix system was then developed for kicking and positioning that took these rebound shots into account. This system is described in the next chapter of this report.

Chapter 9

Mirroring

One key feature of this team strategy is the ability to utilize rebounds off of the top and bottom of the field when determining where to kick the ball. To achieve this, some layers of the field matrix, which are used to determine where to kick the ball, are mirrored before being calculated, and some are mirrored afterward. The following sections describe each matrix layer and how its calculations are done to take into account edge rebounds.

9.1 `GraphShadowsMir.m`

`GraphShadowsMir.m` calculates a field-size matrix which has shadows around and behind opponents to keep the players from moving and kicking to a location where the opponents' chance of getting to the ball is higher than the players' chance. This matrix is calculated using the same algorithm as described in Section 7.3, but before that the players' locations are mirrored above and below the field.

Figure 9.1 shows an example of a situation which is being mirrored this way. When the normal, non-mirroring calculations are done on it, the result is Figure 9.2. To calculate the rebound opportunities, first the opponents positions are copied and mirrored above and below the field as indicated in Figure 9.4. This new, larger field is then sent through the aforementioned shadow-generating algorithm, and the result is shown in Figure 9.3.

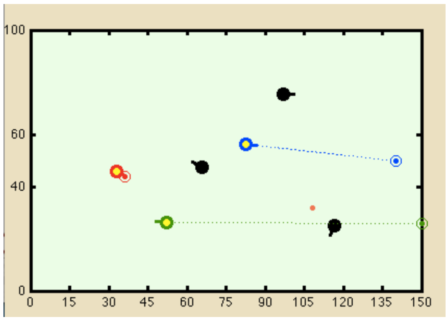


Figure 9.1:

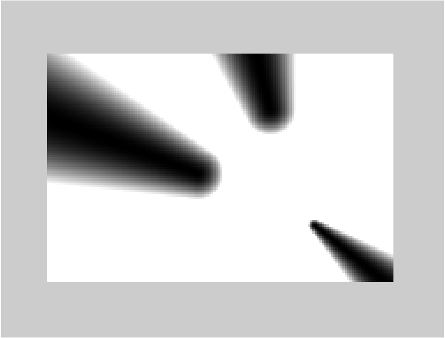


Figure 9.2:

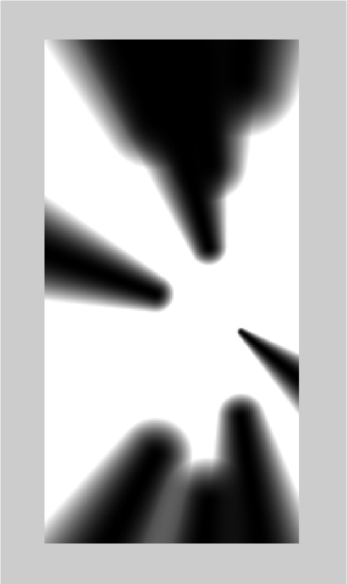


Figure 9.3:

The reason this works is because the simulator supports telling the players to kick to locations which are outside of the field's bounds. When this happens, they simply kick in that direction as if the field did continue on, and the ball simply bounces off the wall.

9.2 `GraphPlayerPositionsMir.m`

`GraphPlayerPositionsMir.m` performs the same way that `GraphShadowsMir` does, but using `GraphPlayerPositions.m` as described in Section 7.2, instead of `GraphShadows.m`.

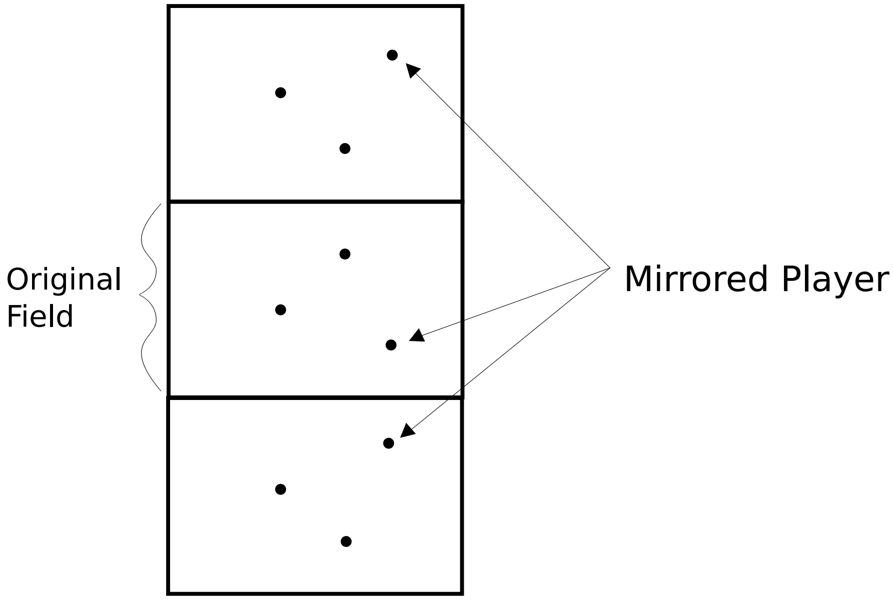


FIGURE 9.4:

9.3 GraphMirror.m

For convenience, the function `GraphMirror.m` was created. It takes in any field-size matrix and outputs one three times the size with mirrored duplicates over and under the original. This function is used with every matrix used in planning kicks (Chapter 8) except for the matrices already mentioned in this chapter, which are the player position matrix, and the shadow matrix.

9.4 Advantages

The major advantage of using these mirrored matrices is the ability of the players to essentially kick the ball around opponents by using rebounds off of the wall. This opens up more opportunities to both pass and shoot.

9.5 Speed Issues

The biggest problem with this code is that doing the same calculations as before on matrices which are three times the size slows down the program on every machine that it was tested on, up to Intel i7 processors running with 4GB of ram.

9.6 Necessary Offset

Because the MATLAB language only supports matrices with positive indices, it is necessary to subtract one field height from the positions in the mirrored matrices when using them in order to access the data that represents the actual field, as opposed to the mirrored data.

9.7 Summary

Mirroring a strategy employed to take advantage of times when the ball bounces off of the top and bottom of the field. It is done by duplicating all opponents into mirrored fields on top and below the field, then doing the normal mapping calculations on this new, larger field. This slowed down the simulator visibly, but provided better results.

Chapter 10

Determine Chaser

One core strategy implemented by this program is to always have one of the players chase after the ball, and try to intercept, kick, or pass it. Choosing this player is done in the Deciding the Engaging Player section of the program, of which this chapter will give an explanation. Figure 10.1 shows a flowchart detailing the process of making this decision.

A superseding situation is the kick-off situation. A Kick-off happens when the game has just begun, and after each goal is scored. When this happens, all of the players line up on their sides of the field, with player 3 being closest to the ball. It is for this reason that player 3 is the designated kicker, and will always be chosen to engage the ball when there is a kick-off.

The next condition is whether or not a player is already engaging the ball. If the Setting States section of code has deemed that a player is successfully chasing the ball, it sets the $isPlayerEngaging$ variable. When this is the case, the Determine Chaser section does not change who this player is.

The next two conditions involve the goalie. If the goalie cannot get to the ball before an opponent, as checked using `canGetThereFirst.m` (as described in Chapter 4), then we want the goalie to behave defensively. When this is the case, the goalie is not taken into consideration when deciding the chaser.

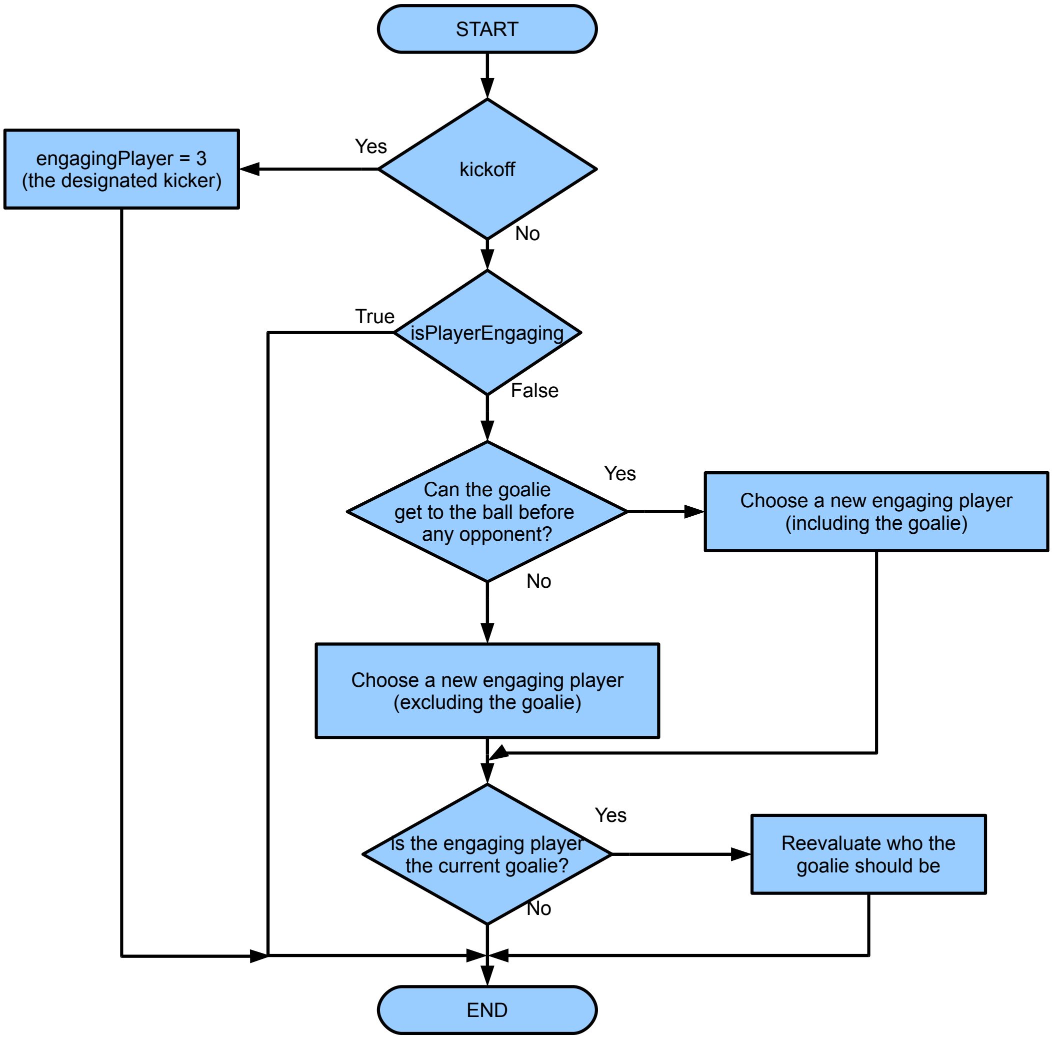


Figure 10.1: Flowchart describing the process of deciding which player should engage the ball

Next, a new engaging player is calculated using `ChooseChaser2.m` as described in Section 10.1.

If the now chosen player is the same as the current goalie, the goalie is reevaluated and chosen with `ClosestToNet.m` as described in Section 4.4.

10.1 ChooseChaser2

ChooseChaser2.m calls Intersection.m (as detailed in Chapter 11 of this report) for each of the players who are being considered as the chaser. Intersection.m returns the amount of time until an inputted player can contact the ball. The values for each player are compared, and the player with the least time until contact is returned by ChooseChaser2.m.

10.2 Summary

Determining the chaser is done by seeing if a player from the previous HLS.m call is still engaging the ball, seeing whether the goalie can get to the ball before every opponent, choosing the player who is in the best position, and then switching who the goalie is if the goalie is chosen to engage the ball. From here, the chaser is told to kick the ball according to the kicking section of code as described in Chapter 8.

Chapter 11

Determining A Player's Intersection With The Ball

11.1 Purpose

The function `Intersection.m` is used to determine when and where a specific agent could contact the ball based on its current trajectory. This function is dependent on the `BallPrediction.m` function to determine where the ball will be in the future.

11.2 Implementation

As input, the `Intersection.m` function takes the agent's current position and orientation, their *Type* attributes, the balls current position and velocity, and lastly an offset value. Sometimes it is necessary for the intersection to be calculated following a planned disruption of the ball (ie. a kick) and so a future ball position and velocity could be used. The offset tells the function how many control cycles in the future the provided ball information is for.

For each cycle between the provided offset and the statically defined maximum number of cycles in the future to check, the function looks at the ball's expected position. Based on this position, the function checks how far the agent can move towards that point in the number of cycles available. If the agent will be able to contact the ball in the current cycle, the function returns the player's location and the number of cycles until the collision will occur. While this does take into account the time it takes the agent to turn to face the collision point, it does expect the agent to move at full speed directly to the point of collision. The players are controlled by this code, so this is a highly accurate approximation for them.

11.3 Application

The `Intersection.m` function is used in several instances. The primary use is to determine which player should be chosen to chase the ball. The player who is first capable of contacting the ball is selected to chase the ball while the other players will perform other duties.

The second usage is to find the location where the kicking player can contact the ball soonest. This allows for rapid kicks as the player is not sent chasing the balls current position or cutting off it's trajectory too far in the future.

The third usage is as part of the function `canGetThereFirst.m`. This function determines if a specific player can reach the ball before any of the opponents. This is primarily useful for when the goalie is considering kicking the ball. During initial testing it was found that without this function the goalie would reposition itself for a kick which allowed an opponent to take a clear shot on net.

11.4 Summary

The `Intersection.m` function is very accurate, but can be processor intensive. Future versions might be improved to provide faster execution, even if it means slightly less accurate results as this can be afforded. Another potential improvement would be to clean up the function's interface so that it is more intuitive for programmers. The idea of generating a hypothetical future ball object and providing offsets is admittedly clumsy feeling. A simpler interface might take in player *FIFO*s and generate all of the prediction information automatically. This could even be done in a wrapping function to minimize needed restructuring.

The `Intersection.m` function fills a specific need allowing players to know where they can intercept the ball on it's current trajectory. A similar function that fills a nearly opposite need is described in the next chapter.

Chapter 12

IntersectPoints

The `IntersectPoints.m` function is used to find places on the field where an agent can pass the ball to a teammate. For each potential receiver, the function calculates where along their current trajectory would be the best place to pass the ball and a best location for each player is then returned. This calculation uses the `PassIntersection.m` function described below as a major component of it's calculation.

12.1 PassIntersection.m

The `PassIntersection.m` function is used to determine where and when the ball can intersect the path of a player. The future position of the player uses a FIFO to determine where the player will be in the future. This function is provided an expected ball speed, a position where the ball is expected to start from, and an offset that tells the function which cycle in the future the ball will have these expected parameters.

Using the provided inputs, the function then systematically looks at each position where the receiving player will be in the future, starting with the offset and ending at the statically set maximum prediction cycle count. The player's expected position is found by the '`PlayerPrediction.m`' function. The function then creates a hypothetical ball variable with the provided ball coordinates and uses the provided speed and the direction to the player's position to find the ball's velocity vector. This information is then passed to '`BallPrediction.m`.' If the ball's predicted position at the point in time currently being inspected will intersect with the player's predicted position, an intersection point has been found. This position, as well as the number of cycles into the future that the collision occurs, is passed back to the calling function.

12.2 Summary

In it's current state, '`PassIntersection.m`' calculates the minimum time it would take for the ball to intersect the player's path. There are no provisions to adjust timing to take into account how the receiving player needs to adjust in order to kick the ball nor how the original player needs to adjust in order to kick at the proper angle. As this function is currently only used as a rough approximation it was deemed adequate, but if higher precision is required, these issues will need to be addressed.

Much like how real life soccer players look at the approximate trajectories of their teammates to perform leading passes, this function provides the team with similar abilities. While the results of this function can be very approximate as agents may at any time change their course of action, the results are accurate enough to provide a high level of team coordination.

Chapter 13

Conclusions

As this project has shown, using layered mappings of the field to establish movement and kicking targets for agents in a simulated soccer environment can provide a high level of coordination between agents. The system described in this report was successfully able to beat all other teams which had been previously designed in the C&CofMARS program at BME, the results of which are shown in Appendix B. The final score for the most recent revision of the team was 39, over 32, which was the highest score out of all of the other students' teams.

This strategy used role assignments based on field conditions to give tasks to players. These roles were dynamically assigned and provided a foundation for the rest of the strategy. A dynamic approach was taken as an alternative to a static approach in order to give the players a wider range of possible actions.

This strategy also used a visual, modular approach as opposed to a monolithic, hidden or complex approach to determine where players should move and kick the ball. This allowed code and calculations to be reused, and provided an intuitive way to check the validity of output quickly. It also gave visual information which showed how players' decisions were being made while the game was happening, which was integral to understanding and fine-tuning the decision making process.

While this system was successful in what it set out to do, there are a number of improvements that could be included in future versions of the team strategy. The code developed in this project runs, but it is not as efficient as the previously developed teams. A more efficient version of this team would reduce the system requirements.

The agents also currently avoid opponents when they are far away, but do little to avoid collisions when players are tightly packed. While the system is able to easily recover from collisions, the team strategy would be improved if collisions between agents were reduced.

This project was done at the Budapest University of Technology and Economics via a University of Manitoba mobility program. The cultural and educational experience was as big a part of this program as this project was, and details are included in Appendix D.

Bibliography

[1] Matthijs T. J. Spaan Jelle R. Kok and Nikos Vlassis, "Multi-Robot Decision Making using Coordination Graphs." Article provided by Dr. Istvan Harmati. 2006, 6.

[2] Anton Ayzikovsky, "C&C of MARS EU Mobility Program Special Project Final Report." Article provided by Dr. Istvan Harmati. 2008, 42.

[3] John Anderson and Jacky Baltes, "An Agent-Based Approach to Introductory Robotics Using Robotic Soccer.", Article provided by Dr. Istvan Harmati. 2006, 20.

[4] Stefan Baert, "Motion Planning Using Potential Fields," *GameDev.net*, July 2000, 5.

[5] Wikipedia, "Polar coordinate system — Wikipedia, The Free Encyclopedia," May 2010, [Online]. Available: `http://en.wikipedia.org/w/index.php?title=Polar_coordinate_system&oldid=360221872`. [Accessed 12-May-2010].

Vita

Name:	Matthew Woelk
Place of Birth:	Winnipeg, Canada
Year of Birth:	1988
Secondary Education:	Mennonite Brethren Collegiate Institute, 1999-2006
Honours & Awards:	University of Manitoba entrance scholarship

Name:	Benjamin Bergman
Place of Birth:	Winnipeg, Canada
Year of Birth:	1988
Secondary Education:	Mennonite Brethren Collegiate Institute, 1999-2006
Honours & Awards:	University of Manitoba entrance scholarship
	Dean's Honour List
	Dr. Kwan Chi Kao Scholarship in Computer Engineering

Appendix A

Budget Summary

Description	Cost (CAD)	Note
Simulation Hardware	$ 0.00	Used our personal computers and computers at BME
Matlab Software	$ 0.00	Was already installed on all computers used prior to the start of the project
Transportation	$ 2322.00	Funding was provided by the C&CofMARS internship program
Accommodations	$ 4750.00	Funding was provided by the C&CofMARS internship program
Food and other expenses	$ 2800.00	Funding was provided by the C&CofMARS internship program

Table A.1: Budget

Appendix B

Tournament Results

This appendix holds the results of the tournament used to evaluate the final performance of the team. The tables and charts are displayed on the following pages. Teams including the words *Matt* and *Ben* are various revisions of the strategy described in this report, with *MergedBranches* being the final revision.

Legend: y's points scored against x

Scores	Anton10.16.2	AntonAltPassTri1	AntonAltPassTri2	AntonDemoPot	AntonDemoWin1	AntonDemo Win2	AntonFinal1	AntonFinal3	KT1	Team_Gyuszi_06061 _mod_by_Peat	Team_RandomPos+1 ouchKick	BensBranch	MattsBranch_Rebou ds	MattsBranch_NoRebo unds	MergedBranches_Re ounds	MergedBranches_No Rebounds	MattAndBen_Root
Anton10.16.2	0	1	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0
AntonAltPassTri1	1	0	1	0	0	0	0	0	0	0	2	0	1	0	0	0	0
AntonAltPassTri2	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	-
AntonDemoPot	34	34	20	0	1	2	1	1	1	3	6	0	0	0	0	0	0
AntonDemoWin1	34	41	0	0	0	0	2	1	0	4	1	0	0	0	0	0	1
AntonDemo Win2	0	3	7	0	0	0	1	0	0	1	1	0	0	0	0	0	1
AntonFinal1	5	5	5	1	1	1	0	1	0	7	4	0	0	1	1	0	2
AntonFinal3	5	5	6	1	1	0	2	0	1	0	6	0	0	0	0	0	1
KT1	7	5	0	2	0	1	1	3	0	0	1	0	1	0	0	0	1
Team_Gyuszi_060617_mod_by _Peat	1	4	0	1	0	0	0	0	0	0	0	0	0	0	0	0	2
Team_RandomPos+1touchKick	1	3	2	1	0	0	1	0	0	0	0	0	1	0	0	1	0
BensBranch	10	10	3	4	6	3	6	6	1	7	8	0	1	0	0	0	2
MattsBranch_Rebounds	16	16	-	3	2	3	1	2	1	6	6	1	0	1	0	0	0
MattsBranch_NoRebounds	11	11	6	3	3	3	4	1	0	11	11	1	0	0	1	1	0
MergedBranches_Rebounds	11	11	6	12	2	0	2	11	2	11	10	0	1	0	0	1	1
MergedBranches_NoRebounds	11	11	6	12	2	0	2	11	2	7	10	0	1	0	0	0	1
MattAndBen_Root	4	4	-	4	2	1	0	2	0	7	3	0	0	1	1	1	0

TOTALS	
Anton10.16.2	2
AntonAltPassTri1	4
AntonAltPassTri2	1
AntonDemoPot	103
AntonDemoWin1	83
AntonDemo Win2	13
AntonFinal1	30
AntonFinal3	27
KT1	20
Team_Gyuszi_060617_mod_by _Peat	6
Team_RandomPos+1touchKick	8
BensBranch	64
MattsBranch_Rebounds	56
MattsBranch_NoRebounds	64
MergedBranches_Rebounds	78
MergedBranches_NoRebounds	74
MattAndBen_Root	27

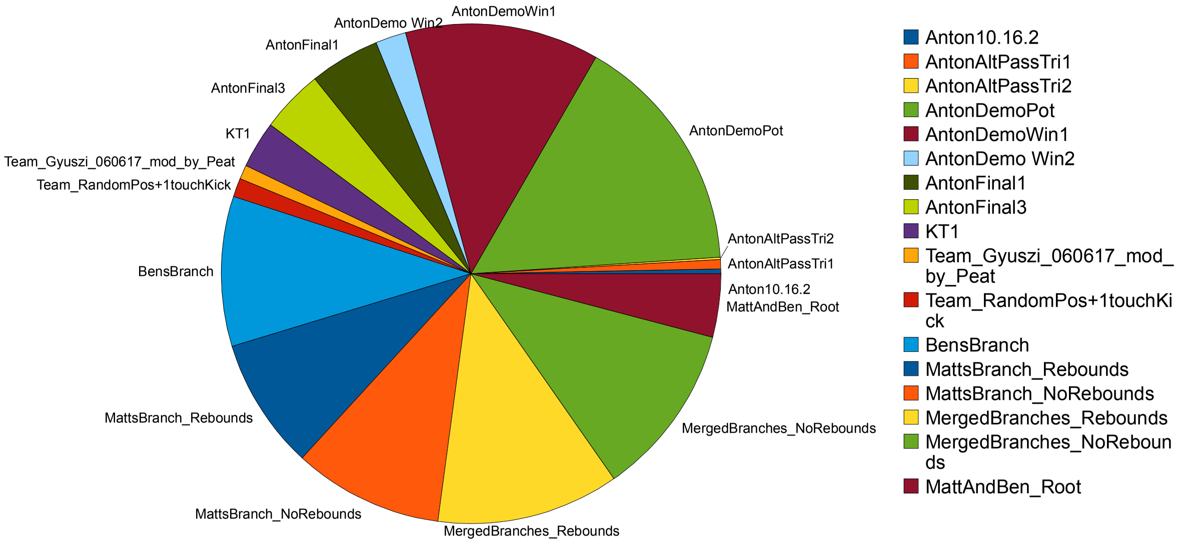


FIGURE B.1: The goals scored in each match of the tournament.

Legend: 3 = won 1 = tied 0 = lost

Who won	Anton10.16.2	AntonAltPassTri1	AntonAltPassTri2	AntonDemoPot	AntonDemoWin1	AntonDemo Win2	AntonFinal1	AntonFinal3	KT1	Team_Gyuszi_0606 7_mod_by_Peat	Team_RandomPos+ touchKick	MattAndBen_BensB anch	MattAndBen_MattsB anch_Rebounds	MattAndBen_MattsB anch_NoRebounds	MattAndBen_Merge Branches_Rebounds	MattAndBen_Merge Branches_NoRebou ds	MattAndBen_Root
Anton10.16.2	0	1	1	0	0	1	0	0	0	1	0	0	0	0	0	0	0
AntonAltPassTri1	1	0	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0
AntonAltPassTri2	1	0	0	0	1	0	0	0	1	1	0	0	1	0	0	0	1
AntonDemoPot	3	3	3	0	3	3	1	1	0	3	3	0	0	0	0	0	0
AntonDemoWin1	3	3	1	0	0	0	3	1	1	3	3	0	0	0	0	0	0
AntonDemo Win2	1	3	3	0	0	0	1	1	0	3	3	0	0	0	1	0	1
AntonFinal1	3	3	3	1	0	1	0	0	0	3	3	0	0	0	0	0	3
AntonFinal3	3	3	3	1	1	1	3	0	0	1	3	0	0	0	0	0	0
KT1	3	3	1	3	1	3	3	3	0	1	3	0	1	1	0	3	3
Team_Gyuszi_060617_mod_by _Peat	1	3	1	0	0	0	0	1	1	0	1	0	0	0	0	0	0
Team_RandomPos+1touchKick	3	3	3	0	0	0	0	0	0	1	0	0	0	0	0	0	0
MattAndBen_BensBranch	3	3	3	3	3	3	3	3	3	3	3	0	1	0	1	0	3
MattsBranch_Rebounds	3	3	1	3	3	3	3	3	1	3	3	1	0	3	3	0	1
MattsBranch_NoRebounds	3	3	3	3	3	3	3	3	1	3	3	3	0	0	3	1	0
MergedBranches_Rebounds	3	3	3	3	3	1	3	3	3	3	3	1	3	0	0	3	1
MergedBranches_NoRebounds	3	3	3	3	3	1	3	3	3	3	3	1	3	0	0	0	1
MattAndBen_Root	3	3	1	3	3	1	0	3	0	3	3	0	1	3	0	3	0

TOTALS	
Anton10.16.2	4
AntonAltPassTri1	4
AntonAltPassTri2	6
AntonDemoPot	23
AntonDemoWin1	18
AntonDemo Win2	17
AntonFinal1	20
AntonFinal3	19
KT1	32
Team_Gyuszi_060617_mod_by _Peat	8
Team_RandomPos+1touchKick	10
MattAndBen_BensBranch	38
MattAndBen_MattsBranch_Reb ounds	37
MattAndBen_MattsBranch_NoR ebounds	38
MattAndBen_MergedBranches_ Rebounds	39
MattAndBen_MergedBranches_ NoRebounds	36
MattAndBen_Root	30

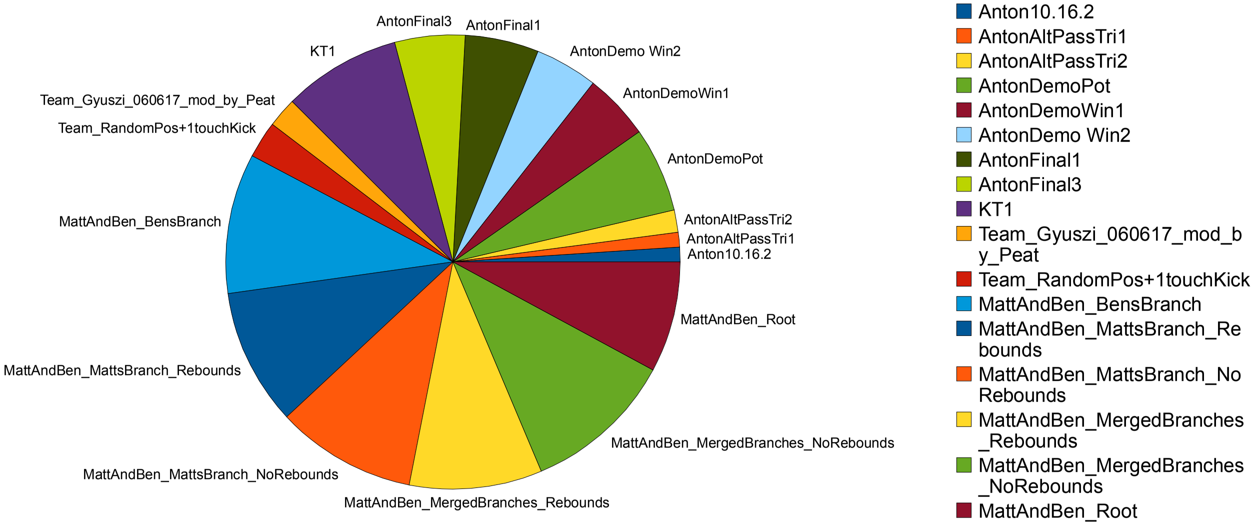


Figure B.2: The wins, losses, and ties, converted into point values. These were used to determine the ranking of the teams in the tournament.

Appendix C

Code

C.1 HLS.m

```
%=% The HLS function is the High-Level Strategy.
%=% This function receives all the environment parameters and returns the control signals for the team.


% Comments starting with %=% were written by Benjamin Bergman
% Comments starting with %-% were written by Matthew Woelk


function ControlSignal = HLS( TeamOwn, TeamOpp, Ball, GameMode, TeamCounter )
global FUN Score
global Environment Team M FieldX FieldY qDamp
global BallTraj HLSTraj                          %=% These variables are used to draw the ball and players' trajectories
global CycleBatch

%=% These are used for the tactical planner.
global qDampRec qDampMRec qDampLogRec

%=% We want to remember the previous queued up commands
persistent Fifo BallPrediction PlayerPrediction

persistent kickoff %-% Whether we are starting a new round.
persistent isPlayerEngaging %-% tells whether a player is currently in the process of kicking the ball.
persistent engagingPlayer %-% tells which player is currently going after the ball.
persistent hasPossession %-% a boolean that states whether we are in possession state or not.
persistent currentGoalie %-% the player which is currently acting as the goalie
persistent matrixField %-% An unchanging matrix of values for the field.
persistent matrixFieldMir %-% An unchanging matrix of values for the field (including top & bottom mirrors)
persistent BallTrajBackup %-% A backup of BallTraj
persistent PlayerTrajBackup %-% A backup of PlayerTraj
persistent PlayerTargets %-% An array of where players want to go.
persistent engagePosition %-% Stores where the kicker is going to contact the ball.
persistent canKick %-% Stores whether a player can kick the ball or not.
persistent kickertarget %-% Stores the spot where the kicker is going to kick the ball.
persistent matrixMoveOut  %-% A black semi−circle in our net.
persistent matrixDontCamp %-% A black semi−circle in the opponent's net.
persistent matrixPlayersGoStatic %-% A field matrix for movement calculations.
persistent firstCalculation %-% so that the first kick calculation for each kick is to clear the ball.
persistent rebounds %-% VERY IMPORTANT: This sets the ability for players to calculate rebound when taking shots. (slows down the game when rebounds are on)
persistent predictCycles %=% The number of cycles in the future we want to predict for the ball's position

persistent dimmer %=% When discouraging backwards kicks, we multiple the field behind the kicker by this




CycleBatch = GameMode(4) + GameMode(5);

qDamp  = 1−Environment.BallDampingFactor;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%=%                        SIMULATION INITIALIZATION                    %=%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
if GameMode(1) == 0

  Fifo = cell(1,M);
  PlayerPrediction = cell(1,M);

  %=% These lines are needed to run the tactical planner
  qDampRec = 1 / qDamp;
  qDampMRec = 1 / (1 - qDamp);
  qDampLogRec = 1 / log(qDamp);

  %-% Initialize all our persistent variables
  isPlayerEngaging = false;
  engagingPlayer = 2;
  hasPossession = false;
  currentGoalie = 1;
  matrixField = FUN.GraphField();
  matrixFieldMir = FUN.GraphMirror(matrixField);
  BallTrajBackup = [];
  PlayerTrajBackup = [];
  PlayerTargets{1} = [];
  matrixMoveOut = FUN.GraphMoveOut();
  matrixDontCamp = FUN.GraphDontCamp();
  matrixSides = FUN.GraphSides();
  matrixPlayersGoStatic = (1-matrixField).*matrixMoveOut.*matrixSides;
  firstCalculation = true;
  rebounds = true; %-% VERY IMPORTANT: This sets the ability for players to calculate
     rebounds when taking shots. (slows down the game when rebounds are on)

  %=% When discouraging backwards kicks, we multiply the field behind the kicker by
    this
  %=% the lower this number (between 0 and 1) the less likely we are to kick
    backwards
  dimmer = 0.9;
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%=%                     KICKOFF INITIALIZATION                       %=%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
if GameMode(2) == 2

  TeamOwn = FUN.HLS_SetUp(TeamOwn);
  ControlSignal=cell(1,Team.NoofTeamMember);
  for i= 1:Team.NoofTeamMember
    ControlSignal{i} = FUN.TP_HARD( TeamOwn, TeamOpp, CycleBatch, i );
    ControlSignal{i} = [GameMode(1) + (1:CycleBatch)', ControlSignal{i}(:,:)]; %
    timestamps?
    Fifo{i} = [];
    PlayerPrediction{i} = repmat(zeros(1,4), 10, 1);
  end
  %-% BallTraj is necessary to draw what the players are going to do.
  BallTraj{TeamCounter} = [-1 -1];
  for i=1:Team.NoofTeamMember
    HLSTraj{TeamCounter}{i}.data  = [TeamOwn{i}.Pos; TeamOwn{i}.Target];
    HLSTraj{TeamCounter}{i}.tst   = 0;
    HLSTraj{TeamCounter}{i}.index = 1;
  end

  kickoff = true;

  %=% This initializes the ball prediction variable
  predictCycles = 20;
  BallPrediction = repmat([FieldX/2, FieldY/2, 0, 0], predictCycles, 1);

  return
end








%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%=%                    BEGINNING OF STRATEGY                         %=%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%



%=% MinKickVel = 1.6; %-% These are currently nearly arbitrary.
%=% MaxKickVel = 1.6; %-% They represent the speed which all our kicks will be.

%=% 1.6 works decently but a dynamic choice for kick speed allows greater versatility
%=% TP_Kick (and thus canKick, our wrapper for it) will find the first working kick within a range, but it is slow and not necessarily the best kick speed
%=% we want to use the ball's current speed and use that to map a new kicking velocity on a different range

RangeKickVel = 0; %=% If we want TP_Kick to try a wider range of speeds, turn this value up
ballSpeed = norm(Ball.Pos(3:4));
MidKickVel = ballSpeed*0.35 + 1.3;
MinKickVel = MidKickVel - RangeKickVel/2;
MaxKickVel = MidKickVel + RangeKickVel/2;

```

```


for i = 1:M %% This is just to format TeamOpp{i}.Pos to play nice with GraphShadows
  OpponentTargets{i} = TeamOpp{i}.Pos;
end
matrixShadow = FUN.GraphShadows(OpponentTargets,Ball.Pos,false,1);
if rebounds
  matrixShadowMir = FUN.GraphShadowsMir(OpponentTargets,Ball.Pos,false,1);
end
%%FUN.DisplayMatrix(FUN.GraphShadowsMir(OpponentTargets,Ball.Pos,false,1),4);





%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Set states. (BallInterrupted, PlayerInterrupted, safeDistanceAway) %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% set isPlayerEngaging to true if the kicker is in the process of kicking the ball.
isPlayerEngaging = FUN.isKicking(Fifo{engagingPlayer});
dontPickGoalie = false;

safeDistanceAway = false;
threshold = 0.8;
%% This ignores the different in the ball's velocity
if norm(Ball.Pos(1:2) - BallPrediction(10,1:2)) > threshold
  BallInterrupted = true; %% An opponent or teammate (or a goalpost) has contacted the ball.
  firstCalculation = true;
else
  BallInterrupted = false;
  if Ball.Pos(3) > 0 && Ball.Pos(1) > FieldX/2
    %% If the ball is far away and has been moving away predictably then set hasPossession.
    safeDistanceAway = true;
  end
end

threshold = 0.01;

if norm(TeamOwn{engagingPlayer}.Pos(1:2) - PlayerPrediction{engagingPlayer}(10, 1:2)) > threshold || ...
```

```
    norm(TeamOwn{engagingPlayer}.Pos(3:4) - PlayerPrediction{engagingPlayer}(10, 3:4)) > threshold
  %=% The engaging player has been disrupted by another player (friend or foe) or possibly run into a wall. As a result, the kick he was trying to perform will not work as expected.
  PlayerInterrupted = true;
else
  PlayerInterrupted = false;
end

if ~FUN.canGetThereFirst(TeamOpp,TeamOwn{engagingPlayer}.Pos,TeamOwn{engagingPlayer}.Type,Ball.Pos,17)
  hasPossession = false;
  if engagingPlayer == currentGoalie
    isPlayerEngaging = false;
    Fifo{currentGoalie} = [];
    BallTraj{TeamCounter} = [-1 -1];
  end
end



%-% If someone is already engaging the ball, see if they're still being successful:
if isPlayerEngaging
  if (~BallInterrupted && ~PlayerInterrupted)
    [ControlSignal{engagingPlayer}, Fifo{engagingPlayer}] = FUN.Kick( Fifo{engagingPlayer}, TeamCounter, engagingPlayer, GameMode);
    hasPossession = true;
  else
    Fifo{engagingPlayer} = [];
    BallTraj{TeamCounter} = [-1 -1];

    isPlayerEngaging = false;
    hasPossession = false;
  end
end






%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%-% Decide who should engage the ball %-%
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

if ~isPlayerEngaging
  %-% Keep state after someone kicks the ball until it is interrupted.
  if BallInterrupted
    hasPossession = false;
  end

  previousEngagingPlayer = engagingPlayer;
  if ~FUN.canGetThereFirst(TeamOpp, TeamOwn{currentGoalie}.Pos, TeamOwn{currentGoalie}.Type, Ball.Pos, 17) %=% if the goalie cannot get to the ball before an opponent, do not let him leave the net
    engagingPlayer = FUN.ChooseChaser2(Ball,TeamOwn,currentGoalie); %-% figure out who should kick the ball
  else
    engagingPlayer = FUN.ChooseChaser2(Ball,TeamOwn); %-% figure out who should kick the ball
  end

  if previousEngagingPlayer ~= engagingPlayer
    firstCalculation = true;
  end

  if engagingPlayer == currentGoalie && ~hasPossession
    %-% Define a new goalie. Maybe.
    currentGoalie = FUN.ClosestToNet(M,TeamOwn,FieldY);
    %-% NB: Set back when he's done kicking? (just a little kick-out; maybe no change needed)
  end

  %=% NB: should be fixed to account for larger/smaller teams. I think this was done as a crude patch for when the ball is not moving which happens primarily at kickoff. Since this could happen at times other than kickoff, this might want to be accounted for as well.
  if kickoff
    engagingPlayer = 3;
    kickoff = false;
  end
end


if safeDistanceAway
  hasPossession = true;
end
```

```






%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%-% Tell the players where to position themselves. %-%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

if hasPossession
  %-% FOR PLAYERS IN POSSESSION—STATE (who aren't going for the ball)
  for inc = 1:M
    if inc ~= engagingPlayer && inc ~= currentGoalie %-% Engaging Player is going
    after the ball.
      %-% NB: we want to not go through other players to get somewhere.
        %=% ^Perhaps a dim shadow behind other players could fix this?
      %-% Currently matrixGo gets a little murky when two players are beside each
    other, but that's probably okay.
      matrixPlayerGo = FUN.GraphShadowsStatic(TeamOwn,inc,false,1);
      matrixGo = matrixField.*matrixShadow.*matrixPlayerGo;
      matrixGo = matrixGo.*matrixDontCamp;

      [highPoint,xVal,yVal] = FUN.FindHighestValue(matrixGo);

      %-% Send player to highPoint (coord: xVal, yVal)
      garbage = []; %-% Do not use the Fifo that GoHere gives us.
      [ControlSignal{inc}, garbage] = FUN.GoHere(inc,[xVal yVal], TeamOwn, GameMode,
    CycleBatch, TeamCounter);
      PlayerTargets{inc} = [xVal yVal];
    end
  end
else
  %-% FOR PLAYERS IN NON—POSSESSION—STATE (who aren't going for the ball)
  for inc = 1:M
    if inc ~= engagingPlayer && inc ~= currentGoalie
      %-% NB: We should have players go between opponents if we want to intercept
    passes.
      matrixPlayerGo = FUN.GraphShadowsStatic(TeamOwn,inc,false,1);
      matrixGoN = matrixPlayersGoStatic.*matrixPlayerGo.*matrixDontCamp;
      [highPoint,xVal,yVal]= FUN.FindHighestValue(matrixGoN);

      %-% Send player to highPoint (coord: xVal, yVal)
      garbage = []; %-% Do not use the Fifo that GoHere gives us.
      [ControlSignal{inc}, garbage] = FUN.GoHere(inc,[xVal yVal], TeamOwn, GameMode, CycleBatch, TeamCounter);
      PlayerTargets{inc} = [xVal yVal];
    end
  end
end




%-% Tell the goalie to move to an ideal spot on the field
if engagingPlayer ~= currentGoalie
  %-% if the ball is on the way to the net, get in the way!
  [onTheWay wallIntersection] = FUN.isBallGoingForOurGoal(Ball);
  if onTheWay
    %-% Find out where the ball will intersect our net    %=% <-- I think this comment should read: Find out where the goalie can intercept the ball quickest
    intersectionPoint = FUN.DistanceToLine(Ball.Pos(1),Ball.Pos(2),0,wallIntersection,TeamOwn{currentGoalie}.Pos(1),TeamOwn{currentGoalie}.Pos(2),false);
    goalieTarget = intersectionPoint(1:2);
    %=% NB: might want goalie to start moving along the ball trajectory once it is on the line (intersectionPoint(3) < ball radius, for instance)
    %=%      move towards ball to get it away from our net? move towards our goal to slow the ball upon contact? attempt a slower/faster kick? only intercept if ball is close?
  else
    goalieTarget = FUN.Goalie(Ball,TeamOpp);
  end
  garbage = []; %-% Do not use the Fifo that GoHere gives us.
  [ControlSignal{currentGoalie}, garbage] = FUN.GoHere(currentGoalie,goalieTarget, TeamOwn, GameMode, CycleBatch, TeamCounter);

  PlayerTargets{currentGoalie} = goalieTarget;
end




%%%%%%%%%%%%%%%%%%%%%%%%
%-% Setting up the kick %-%
%%%%%%%%%%%%%%%%%%%%%%%%

if ~isPlayerEngaging
  if ~hasPossession && firstCalculation
```

```
    %-% the first calculated place to kick doesn't take into account our players.
    %-% - This makes kicks that are not able to be calculated more than once "clear the ball"
    %-% - rather than kick to a place where our players are going defensively.
    %-% - hasPossession is used because we will only need to do this when we don't have ball control.
    %if rebounds
    %  matrixKick = matrixFieldMir .* matrixShadowMir .* FUN.GraphMirror(matrixMoveOut);
    %else
      matrixDimmer = FUN.GraphDimmer(TeamOwn{engagingPlayer}.Pos, dimmer);
      matrixKick = matrixField .* matrixShadow .* matrixMoveOut .* matrixDimmer;
    %end
  else
    %if rebounds
    %  matrixPlayer = FUN.GraphPlayerPositionsMir(PlayerTargets,Ball.Pos,false,1,engagingPlayer);
    %  matrixKick = max(matrixFieldMir,1-matrixPlayer) .* matrixShadowMir .* FUN.GraphMirror(matrixMoveOut);
    %else
      matrixPlayer = FUN.GraphPlayerPositions(PlayerTargets,Ball.Pos,false,1,engagingPlayer);
      matrixDimmer = FUN.GraphDimmer(TeamOwn{engagingPlayer}.Pos, dimmer);
      matrixKick = max(matrixField,1-matrixPlayer) .* matrixShadow .* matrixMoveOut .* matrixDimmer;
    %end
  end
  [highPoint,xVal,yVal] = FUN.FindHighestValue(matrixKick);
  yVal = yVal - FieldY; %-% This is because graphs cannot have negative indices, so the mirrored graphs are one field-height too high.

  %-% This canKick is to get an estimate of how long it will take to engage the ball
  [canKick, FifoTemp, BallTrajBackup, PlayerTrajBackup]=FUN.canKick(MinKickVel, MaxKickVel, TeamOwn{engagingPlayer}, [xVal,yVal], Ball.Pos, TeamCounter, engagingPlayer, GameMode);

  if canKick
    %-% instead of using Ball's position, use the position where the player will engage the ball.
    timeUntilContact = FUN.timeLeftInKick(FifoTemp,GameMode);
    engagePositionMatrix = FUN.BallPrediction(Ball.Pos,timeUntilContact,false);
    %=% NB: these two lines can probably be replaced to improve speed; use "end" instead of "1" and no flip is needed
    engagePositionMatrix = flipud(engagePositionMatrix);
```

```
engagePosition = engagePositionMatrix(1,:);

%-% PlayerFuture gives the positions where the ball will be able to meet up with the players when kicked.
PlayerFuture = FUN.IntersectPoints(TeamOwn, PlayerTargets, engagePosition, MaxKickVel, timeUntilContact, engagingPlayer, Fifo, GameMode);
if rebounds
  matrixPlayer = FUN.GraphPlayerPositionsMir(PlayerFuture, engagePosition, false, 1, engagingPlayer);
  matrixShadow2 = FUN.GraphShadowsMir(OpponentTargets, engagePosition, false, 2);
else
  matrixPlayer = FUN.GraphPlayerPositions(PlayerFuture, engagePosition, false, 1, engagingPlayer);
  matrixShadow2 = FUN.GraphShadows(OpponentTargets, engagePosition, false, 2);
end
if firstCalculation
  if rebounds
    matrixDimmer = FUN.GraphMirror(FUN.GraphDimmer(TeamOwn{engagingPlayer}.Pos, dimmer));
    matrixKick = matrixFieldMir .* matrixShadow2 .* FUN.GraphMirror(matrixMoveOut) .* matrixDimmer;
  else
    matrixDimmer = FUN.GraphDimmer(TeamOwn{engagingPlayer}.Pos, dimmer);
    matrixKick = matrixField .* matrixShadow2 .* matrixMoveOut .* matrixDimmer;
  end
else
  if rebounds
    matrixDimmer = FUN.GraphMirror(FUN.GraphDimmer(TeamOwn{engagingPlayer}.Pos, dimmer));
    matrixKick = max(matrixFieldMir, 1-matrixPlayer) .* matrixShadow2 .* FUN.GraphMirror(matrixMoveOut) .* matrixDimmer;
  else
    matrixDimmer = FUN.GraphDimmer(TeamOwn{engagingPlayer}.Pos, dimmer);
    matrixKick = max(matrixField, 1-matrixPlayer) .* matrixShadow2 .* matrixMoveOut .* matrixDimmer;
  end
end
[highPoint, xVal, yVal] = FUN.FindHighestValue(matrixKick);
if rebounds
  yVal = yVal - FieldY; %-% This is because graphs cannot have negative indices, so the mirrored graphs are one field-height too high.
end

%-% This canKick is used to create the player's Fifo.
```

```
    [canKick, FifoTemp, BallTrajBackup, PlayerTrajBackup]=FUN.canKick(MinKickVel, MaxKickVel, TeamOwn{engagingPlayer}, [xVal,yVal], Ball.Pos, TeamCounter, engagingPlayer, GameMode);
    kickertarget = [xVal,yVal];
  end
else
  %-% A player is engaging, so we tell them to reevaluate until they're 30 cycles from their kick.
  timeUntilContact = FUN.timeLeftInKick(Fifo{engagingPlayer},GameMode);
  if timeUntilContact > 31
    %-% PlayerFuture gives the positions where the ball will be able to meet up with the players when kicked.
    PlayerFuture = FUN.IntersectPoints(TeamOwn, PlayerTargets, engagePosition, MaxKickVel, timeUntilContact, engagingPlayer, Fifo, GameMode);
    if rebounds
      matrixPlayer = FUN.GraphPlayerPositionsMir(PlayerFuture, engagePosition, false, 1, engagingPlayer);
      matrixShadow2 = FUN.GraphShadowsMir(OpponentTargets, engagePosition, false, 2);
      matrixDimmer = FUN.GraphMirror(FUN.GraphDimmer(TeamOwn{engagingPlayer}.Pos, dimmer));
      matrixKick = max(matrixFieldMir,1-matrixPlayer) .* matrixShadow2 .* FUN.GraphMirror(matrixMoveOut) .* matrixDimmer;
    else
      matrixPlayer = FUN.GraphPlayerPositions(PlayerFuture, engagePosition, false, 1, engagingPlayer);
      matrixShadow2 = FUN.GraphShadows(OpponentTargets, engagePosition, false, 2);
      matrixDimmer = FUN.GraphDimmer(TeamOwn{engagingPlayer}.Pos, dimmer);
      matrixKick = max(matrixField,1-matrixPlayer) .* matrixShadow2 .* matrixMoveOut .* matrixDimmer;
    end
    [highPoint, xVal, yVal] = FUN.FindHighestValue(matrixKick);
    if rebounds
      yVal = yVal - FieldY; %-% This is because graphs cannot have negative indices, so the mirrored graphs are one field-height too high.
    end

    %-% This canKick is used to create the player's Fifo.
    [canKick, FifoTemp, BallTrajBackup, PlayerTrajBackup]=FUN.canKick(MinKickVel, MaxKickVel, TeamOwn{engagingPlayer}, [xVal,yVal], Ball.Pos, TeamCounter, engagingPlayer, GameMode);
    kickertarget = [xVal,yVal];
  else
    FifoTemp = Fifo{engagingPlayer};
  end
```

```
end



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%-% Kicking (and moving if unable to kick) %-%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%-% If the engaging player can kick, we tell them to. If not, we tell them to chase the ball.
if canKick
  firstCalculation = false;
  [ControlSignal{engagingPlayer}, Fifo{engagingPlayer}] = FUN.Kick( FifoTemp, TeamCounter, engagingPlayer, GameMode );
end
if ~canKick
  %-% Tell the goalie to move to an ideal spot on the field
  if engagingPlayer == currentGoalie
    %-% if the ball is on the way to the net, get in the way!
    hasPossession = false;
    [onTheWay wallIntersection] = FUN.isBallGoingForOurGoal(Ball);
    if onTheWay
      %-% Find out where the ball will intersect our net
      intersectionPoint = FUN.DistanceToLine(Ball.Pos(1),Ball.Pos(2),0, wallIntersection,TeamOwn{currentGoalie}.Pos(1),TeamOwn{currentGoalie}.Pos(2), false);
      goalieTarget = intersectionPoint(1:2);
    else
      goalieTarget = FUN.Goalie(Ball,TeamOpp);
    end
    garbage = []; %-% Do not use the Fifo that GoHere gives us.
    [ControlSignal{currentGoalie}, garbage] = FUN.GoHere(currentGoalie,goalieTarget, TeamOwn, GameMode, CycleBatch, TeamCounter);

    PlayerTargets{currentGoalie} = goalieTarget;
  else
    %-% Reset the Fifo and BallTraj:
    Fifo{engagingPlayer} = [];
    BallTraj{TeamCounter} = [-1 -1];

    %-% Tell player to intersect the ball and block it UNLESS the ball is headed toward the opposition's net.
    %=% if a player is blocking a shot and is within N cycles of contact with the ball, move over slightly
    pointOfContact = BallPrediction(15,1:2);

    %=% NB: the condition for making a player move out of the way might be able to use some improvement still. Players should be moved to an intelligent location.
    if FUN.isBallGoingForGoal(Ball) && (norm(pointOfContact - TeamOwn{engagingPlayer}.Pos(1:2)) < 20)
      xpos = FieldX.*0.9;
      if Ball.Pos(2) > TeamOwn{engagingPlayer}.Pos(2)
        ypos = FieldY.*0.1;
      else
        ypos = FieldY.*0.9;
      end
    else
      [xpos, ypos, cyc] = FUN.Intersection(TeamOwn{engagingPlayer}.Pos,TeamOwn{engagingPlayer}.Type,Ball.Pos,0);
    end
    garbage = []; %-% Do not use the Fifo that GoHere gives us.
    [ControlSignal{engagingPlayer}, garbage] = FUN.GoHere(engagingPlayer, [xpos,ypos],TeamOwn, GameMode, CycleBatch, TeamCounter);
  end
end
PlayerTargets{engagingPlayer} = [];







%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%-% Set up Ball and Player Prediction for the next 'predictCycles' cycles %-%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%=% This establishes a prediction for the future state of the ball and any kicking player.
%=% These values are used in the next HLS call to determine if a kick has been interrupted.
BallPrediction = FUN.BallPrediction(Ball.Pos,predictCycles);
%-% plan for the kicker's contact with the ball as well.
if canKick
  timeUntilContact = FUN.timeLeftInKick(Fifo{engagingPlayer},GameMode);
  if timeUntilContact <= predictCycles
    %-% The purpose of this section is to correct BallPrediction to account for when our players kick the ball.
```

```
        engagePositionMatrix = FUN.BallPrediction(Ball.Pos,timeUntilContact,false);
        engagePositionMatrix = flipud(engagePositionMatrix);
        engagePosition = engagePositionMatrix(1,:);
        maxvel = MinKickVel; %-% Now that we're using a range of possible speeds, this doesn't work.
        delx =  engagePosition(1) - kickertarget(1);
        dely =  engagePosition(2) - kickertarget(2);
        velx = sqrt(maxvel.^2./(1+(dely.^2/delx.^2)));
        vely = sqrt(maxvel.^2 - velx.^2);
        %-% This doesn't make the velocities completely correctly, but it's okay because we don't check for them.
        targetVector = [-velx -vely];
        fakeBall.Pos = [engagePosition(1:2) targetVector];
        reflectPre = FUN.BallPrediction(fakeBall.Pos,predictCycles+1-timeUntilContact);
        reflectPreSize = size(reflectPre);
        BallPrediction = [BallPrediction(1:(timeUntilContact-1),:);reflectPre(1:reflectPreSize(1),:)];
    end
end

for i=1:M
    PlayerPrediction{i} = FUN.PlayerPrediction( TeamOwn{i}.Pos, Fifo{i}, 10, GameMode );
end





%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%-% Notes for possible further improvements %-%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%-% Players need much bigger radii so that they don't go near each other. (Maybe? Maybe not.)
%-% If a player is going to kick the ball AND no opponent can get there first, THEN change state. (not currently a visible issue.)
%-----------------------------------------%
%-% Make the players actually get out of the way when the ball is heading toward their net. (It's close right now.)

%-% Just moving to where the ball is when we can't kick it is turning into a very dangerous (and stupid) thing.
%-%  - players will go toward the ball even when it means scoring on ourselves.
%-%  - maybe have players run between the ball and the net when they're near our goal. (might be more difficult than it sounds)
%-%  - maybe have players position themselves instead when ball's near our goal. (not a true fix)

%-% Perhaps include a timeout for how long between passes the ball is still "in our control" (so that dumb teams won't affect us as much).
%-% Using rebounds off of the sides of the field when determining how to shoot.

%-% Increase goalie winduptime delay
%-% Get players' default positions to be closer to the center of the field.
%-% Players need much bigger radiuses so that they don't go near eachother.
%-% Current biggest folly: positioning.

%-% If a player is going to kick the ball AND no opponent can get there first, THEN change state






%   2010
% Benjamin Bergman - ben.bergman@gmail.com
% Matthew Woelk - umwoelk@cc.umanitoba.ca
% This document is subject to the Creative Commons 3.0 Attribution Non-Commercial Share Alike license.
% http://creativecommons.org/licenses/by-nc-sa/3.0/
```

C.2 Supplementary Functions

C.2.1 `BallPrediction.m`

```
function matrix = BallPrediction(BallPos,cycles,displayOutput)

%-% Predict where the ball will be "cycles" number of cycles in the future.
%=% NB: possible future improvement: for easier debugging, the code repitition should be removed; all appropriate code should be in the loop.

global qDamp FieldX FieldY

```

```
%-% Now it takes into account bounces off the wall.
ballradius = 1;

%-% This is a workaround to fix rebounding off top:
ballradiustop = 1.3;

matrix = zeros(cycles,4);

prevX = BallPos(3)*qDamp; %-% predicted velocity of the ball.
prevY = BallPos(4)*qDamp;
preX = BallPos(1) + prevX; %-% predicted position of the ball.
preY = BallPos(2) + prevY;

%-% To consider bouncing off of walls:
if preX < (0 + ballradius)
  preX = 2*ballradius - preX;
  prevX = - prevX;
end

if preX > (FieldX - ballradius)
  preX = 2*(FieldX-ballradius) - preX;
  prevX = - prevX;
end

if preY < (0 + ballradius)
  preY = 2*ballradius - preY;
  prevY = - prevY;
end

if preY > (FieldY - ballradiustop)
  preY = 2*(FieldY-ballradiustop) - preY;
  prevY = - prevY;
end

matrix(1,:) = [preX preY prevX prevY];

for i = 2:cycles
  prevX = prevX*qDamp; %-% predicted velocity of the ball.
  prevY = prevY*qDamp;
  preX = preX + prevX; %-% predicted position of the ball.
  preY = preY + prevY;

  %-% To consider bouncing off of walls:
  if preX < (0 + ballradius)
```

```
    preX = 2*ballradius - preX;
    prevX = - prevX;
  end

  if preX > (FieldX - ballradius)
    preX = 2*(FieldX-ballradius) - preX;
    prevX = - prevX;
  end

  if preY < (0 + ballradius)
    preY = 2*ballradius - preY;
    prevY = - prevY;
  end

  if preY > (FieldY - ballradiustop)
    preY = 2*(FieldY-ballradiustop) - preY;
    prevY = - prevY;
  end

  matrix(i,:) = [preX preY prevX prevY];
end


if exist('displayOutput', 'var') && displayOutput
  figure(4);
  clf;
  hold on;
  set(gcf,'Position',[500 30 490 300]);

  xlim([0 150]);
  ylim([0 100]);

  line([BallPos(1) BallPos(1)],[BallPos(2) BallPos(2)],'Marker','o','Color','black');
  line([preX preX],[preY preY],'Marker','o','Color','blue');
end

%    2010
% Benjamin Bergman - ben.bergman@gmail.com
% Matthew Woelk - umwoelk@cc.umanitoba.ca
% This document is subject to the Creative Commons 3.0 Attribution Non-Commercial Share Alike license.
% http://creativecommons.org/licenses/by-nc-sa/3.0/
```

C.2.2 canGetThereFirst.m

```
function canGetThereFirst = canGetThereFirst(TeamOpp,PlayerPos,PlayerType,BallPos, offset)

%-% This function calculates whether our player will be able to get to the ball before any opponent.
%-% offset: roughly how long it takes to set up a shot. (17 seems like a good choice)
%=% NB: this offset is an approximation. If the ball is moving quickly, this approximation may fail. This might be better to incorporate with the Intersection function.

global FUN M

%-% the x position and y position are not used
[garbage1, garbage2, timeTillGKick] = FUN.Intersection(PlayerPos,PlayerType, BallPos, 0);
for i = 1:M
  [xpos, ypos, timeTillOKick(i)] = FUN.Intersection(TeamOpp{i}.Pos,PlayerType,BallPos,0);
end
%-% we assume the opponent needs no time to wind-up.
if any(timeTillOKick < timeTillGKick + offset)
  canGetThereFirst = false;
else
  canGetThereFirst = true;
end

%     2010
% Benjamin Bergman - ben.bergman@gmail.com
% Matthew Woelk - umwoelk@cc.umanitoba.ca
% This document is subject to the Creative Commons 3.0 Attribution Non-Commercial Share Alike license.
% http://creativecommons.org/licenses/by-nc-sa/3.0/
```

C.2.3 canKick.m

```
function [CanKick, Fifo, BallTrajBackup, PlayerTrajBackup] = canKick( MinKickVel, MaxKickVel, Player, Target, BallPos, TeamCounter, agentIndex, GameMode);
%CANKICK Generates a set of control signals for the provided player
% and determines whether that player is able to perform a kick from
% their current location on the field.
%
```

```
% Outputs:
% CanKick:              Boolean. This will be 0 if we are not able to
%                       initiate kick from the current pos.
% Fifo:                 The set of control signals for the new kick.
%                       This is the fifo for the current player only.
% BallTrajBackup:       This is a backup of the previous BallTraj.
%                       The use of TP_KICK overwrites the previous
%                       value which is needed for properly displaying
%                       a continued kick. This needs to be passed to
%                       KICK if continuing a previous kick.
%                 NOTE: There is only one BallTraj per team, so be
%                       careful if multiple agents are running the
%                       canKick function.
% PlayerTrajBackup:     Similar to BallTrajBackup.
%
% Inputs:
% MinKickVel:           This is the minimum speed we want the ball to
%                       have when we kick it.
% MaxKickVel:           This is the maximum speed we want the ball to
%                       have when we kick it.
%                 NOTE: Having a wide range of values between the
%                       above inputs will cause TP_KICK to be quite
%                       slow. Ideally the same value should be passed
%                       for both of these inputs.
% Player:               This is the player we want to make the kick.
%                       For instance, pass TeamOwn{agentIndex}.
% Target:               This is the X and Y value of our kick target.
%                       For example, if we want to kick into the goal,
%                       use Target = [150, 50].
% BallPos:              This is the ball's position and velocity.
% TeamCounter:          This is the team number of the current player.
% agentIndex:           This is the agent number of the current player.
% GameMode:             This is the GameMode variable passed from the
%                       simulator.


global FUN

global BallTraj HLSTraj

BallTrajBackup = BallTraj{TeamCounter};
PlayerTrajBackup = HLSTraj{TeamCounter}{agentIndex};

```

```
[Fifo, BallPosPlanned, CyclePlanned] = FUN.TP_Kick(MinKickVel, MaxKickVel, Player, Target, BallPos, TeamCounter, agentIndex);

if isempty(Fifo)
  CanKick = 0;
else
  CanKick = 1;
end

Fifo = FUN.U_TimeStamp(Fifo, GameMode(1));

%    2010
% Benjamin Bergman - ben.bergman@gmail.com
% Matthew Woelk - umwoelk@cc.umanitoba.ca
% This document is subject to the Creative Commons 3.0 Attribution Non-Commercial Share Alike license.
% http://creativecommons.org/licenses/by-nc-sa/3.0/
```

C.2.4 ChooseChaser2.m

```
function ind = ChooseChaser2(Ball,TeamOwn,NotThisOne)

%-% The purpose of this function is to determine which player should chase after the ball.
%-% Whichever player can get to the ball first is chosen.
%-% If NotThisOne is set, that player will not be chosen.

global FUN M

if exist('NotThisOne','var')
  omit = NotThisOne;
else
  omit = -1;
end

for i = 1:M
  if i ~= omit
    %-% the x position and y position are not used
    [garbage1, garbage2, timeTillKick(i)] = FUN.Intersection(TeamOwn{i}.Pos,TeamOwn{i}.Type,Ball.Pos,0);
    %-% 0 is chosen because we assume the opponent needs no time to wind-up.
  else
    timeTillKick(i) = inf;
  end
end

[value ind] = min(timeTillKick);

%     2010
% Benjamin Bergman - ben.bergman@gmail.com
% Matthew Woelk - umwoelk@cc.umanitoba.ca
% This document is subject to the Creative Commons 3.0 Attribution Non-Commercial Share Alike license.
% http://creativecommons.org/licenses/by-nc-sa/3.0/
```

C.2.5 `ClosestToNet.m`

```
function lowestPlayer = ClosestToNet(M,TeamOwnSave,FieldY,NotThisOne)

%-% Choose who will be goalie (not including NotThisOne)
%-% This is done by determining which player is closest to the center of our goal.

%-% If no current goalie is specified, all players are considered.
if ~exist('NotThisOne', 'var')
  NotThisOne = -1;
end

distanceFromNet = @(x,y) sqrt((x - 0).^2 + (y - FieldY/2).^2);
lowestValue = 999; %-% Arbitrarily high

if NotThisOne ~= 1
  lowestValue = distanceFromNet(TeamOwnSave{1}.Pos(1),TeamOwnSave{1}.Pos(2));
  lowestPlayer = 1;
end
for inc = 2:M
  if distanceFromNet(TeamOwnSave{inc}.Pos(1),TeamOwnSave{inc}.Pos(2)) < lowestValue && inc ~= NotThisOne
    lowestPlayer = inc;
  end
end

%     2010
% Benjamin Bergman - ben.bergman@gmail.com
% Matthew Woelk - umwoelk@cc.umanitoba.ca
% This document is subject to the Creative Commons 3.0 Attribution Non-Commercial Share Alike license.
% http://creativecommons.org/licenses/by-nc-sa/3.0/
```

C.2.6 DisplayMatrix.m

```
function noOutput = DisplayMatrix(matrix,whichFigure)

%-% This function displays matrix on figure whichFigure.

if exist('whichFigure','var')
  figure(whichFigure);
else
  figure(4);
end
imshow(flipud(matrix));

noOutput = true;

%     2010
% Benjamin Bergman - ben.bergman@gmail.com
% Matthew Woelk - umwoelk@cc.umanitoba.ca
% This document is subject to the Creative Commons 3.0 Attribution Non-Commercial Share Alike license.
% http://creativecommons.org/licenses/by-nc-sa/3.0/
```

C.2.7 DistanceToLine.m

```
function inter = DistanceToLine (x1, y1, x2, y2, x3, y3, OnLineSegment) %inter = [x, y, dist]

%function returns the distance to line (point1 to point 2) and the third point and the intersection point

%-% adapted from:
%-% http://www.gamedev.net/community/forums/topic.asp?topic_id=444154
%-% accessed on March 5, 2010
%-% made by oliii (GDNet+ Member)
%-%   posted on 17/4/2007


A = [x1,y1];
B = [x2,y2];
```

```
P = [x3,y3];

AP = P-A;
AB = B-A;
ab2 = AB(1)*AB(1) + AB(2)*AB(2);
apab = AP(1)*AB(1) + AP(2)*AB(2);
tee = apab/ab2;
if OnLineSegment
  if tee < 0.0
    tee = 0.0;
  elseif tee > 1.0
    tee = 1.0;
  end
end
closest = A + AB*tee;

inter = [closest,Distance([x3,y3],closest)];



%     2010
% Benjamin Bergman - ben.bergman@gmail.com
% Matthew Woelk - umwoelk@cc.umanitoba.ca
% This document is subject to the Creative Commons 3.0 Attribution Non-Commercial Share Alike license.
% http://creativecommons.org/licenses/by-nc-sa/3.0/
```

C.2.8 DistanceToLine2.m

```
function inter = DistanceToLine2 (x1, y1, x2, y2, x3, y3, OnLineSegment) %inter = [x, y, dist]

%-% function returns the distance to line (point1 to point 2) and the third point and the intersection point
%-% This function takes in matrices of points instead of just points.

%-% adapted from:
%-% http://www.gamedev.net/community/forums/topic.asp?topic_id=444154
%-% accessed on March 5, 2010
%-% made by oliii (GDNet+ Member)
%-%   posted on 17/4/2007


```

```
sizex = size(x1);
sizey = size(y1);

Ax = x1;
Ay = y1;
Bx = ones(sizex(1),sizex(2))*x2;
By = ones(sizey(1),sizey(2))*y2;
Px = ones(sizex(1),sizex(2))*x3;
Py = ones(sizey(1),sizey(2))*y3;

APx = Px-Ax;
APy = Py-Ay;
ABx = Bx-Ax;
ABy = By-Ay;
ab2 = ABx.*ABx + ABy.*ABy;
apab = APx.*ABx + APy.*ABy;
tee = apab./ab2;
if OnLineSegment
  tee = max(tee,0.0);
  tee = min(tee,0.999);
end
closestx = Ax + ABx.*tee;
closesty = Ay + ABy.*tee;

inter = sqrt((x3-closestx).^2 + (y3 - closesty).^2);

%    2010
% Benjamin Bergman - ben.bergman@gmail.com
% Matthew Woelk - umwoelk@cc.umanitoba.ca
% This document is subject to the Creative Commons 3.0 Attribution Non-Commercial Share Alike license.
% http://creativecommons.org/licenses/by-nc-sa/3.0/
```

C.2.9 `FindHighestValue.m`

```
function [sparseMatrix,xVal,yVal] = FindHighestValue(matrix)

%-% This function returns the x and y location of the highest
%-% point in matrix.
%-% It also gives a sparse matrix which contains a single non-zero
%-% where the high point in matrix is.

[maxes,yVals] = max(matrix);
```

```
[maxVal,xVal] = max(maxes);
yVal = yVals(xVal);
sizeOfMatrix = size(matrix);
sparseMatrix = zeros(sizeOfMatrix(1),sizeOfMatrix(2));
sparseMatrix(yVal,xVal) = 0.9999;

%     2010
% Benjamin Bergman - ben.bergman@gmail.com
% Matthew Woelk - umwoelk@cc.umanitoba.ca
% This document is subject to the Creative Commons 3.0 Attribution Non-Commercial Share Alike license.
% http://creativecommons.org/licenses/by-nc-sa/3.0/
```

C.2.10 `Goalie.m`

```
function target = Goalie(Ball,TeamOpp)

%-% This function defines the position where the goalie should go
%-% when the goalie in defensive mode.

persistent boundingRadius myline1
global  FieldY Environment M FUN

if ~isempty(boundingRadius)
  boundingRadius = TeamOwn{1}.Type.BoundingRadius
end

%-% Define Care zone
distBtwnPnts = @(x1,y1,x2,y2) (sqrt((x2-x1).^2 + (y2-y1).^2));
%-%See which opponent is closest to the center of the net, and return that distance.

%-%Calculate the angle formed by two lines that go from the goalposts to the ball:
%-%distance between goalposts:
c = Environment.GoalSize;
%-%distance from both goalposts to the ball:
a = distBtwnPnts(Ball.Pos(1),Ball.Pos(2),0,Environment.FieldSize(2)/2+Environment.GoalSize/2);
b = distBtwnPnts(Ball.Pos(1),Ball.Pos(2),0,Environment.FieldSize(2)/2-Environment.GoalSize/2);

%-%Cos Law:
%-%C = acos((a.^2+b.^2-c.^2)/(2*a*b));
angleC = acos((a.^2+b.^2-c.^2)/(2*a*b));
```

```

%-%Finding the point in the net where a line that is halfway between those two lines (in terms of angle) intersects the goal line (xpos, ypos)
angleB = acos((a.^2+c.^2-b.^2)/(2*a*c));
%-%3 angles of a triangle add up to pi
intermediateAngle = pi - angleC/2 - angleB;

%-%Sin Law:
%-% sin(A)/a = sin(B)/b = sin(C)/c
yposAboveNet = sin(angleC/2)*b / sin(angleB);

ypos = yposAboveNet + Environment.FieldSize(2)/2 - Environment.GoalSize/2;


%-%Find opponent with the closest intersection,it's the one we care about
care = [];
olddist = 999; %-%Arbitrarily large

for i=1:M
  %-%Find the intersection at the closest point between the location of the opponent and the line made by the ball and the calculated point in the net (ypos).
  intersection = FUN.DistanceToLine(0,ypos,Ball.Pos(1),Ball.Pos(2),TeamOpp{i}.Pos(1),TeamOpp{i}.Pos(2),false);

  newdist = intersection(1);

  if newdist < olddist
    care = i;
    olddist = newdist;
  end
end
intersection = FUN.DistanceToLine(0,ypos,Ball.Pos(1),Ball.Pos(2),TeamOpp{care}.Pos(1),TeamOpp{care}.Pos(2),false);


%-%Put the goalie between the calculated spot and the ball, but not further away from the net as 'intersection'
if intersection(1) > Ball.Pos(1)/3
  target = [Ball.Pos(1)/3, ypos - (ypos-Ball.Pos(2))/3];
elseif intersection(1) < boundingRadius
  %-%This is the case where the intersection is behind the net.
  %-%Put the goalie on the goal-line at ypos
  target = [0,ypos];
else
```

```
  target = [intersection(1), ypos - (ypos-Ball.Pos(2))/3];
end

%-% SHOW LIMITING LINES: --v
%-%figure(3)
%-%cla
%-%line([0,150],[0,0]);
%-%line([0,0],[0,100]);
%-%myline1 = line([0,Ball.Pos(1)],[ypos,Ball.Pos(2)],'LineStyle','-','Color','blue');
%-%myline2 = line([intersection(1),TeamOpp{care}.Pos(1)],[intersection(2),TeamOpp{care}.Pos(2)],'LineStyle','-','Color','red');
%-% --^

%     2010
% Benjamin Bergman - ben.bergman@gmail.com
% Matthew Woelk - umwoelk@cc.umanitoba.ca
% This document is subject to the Creative Commons 3.0 Attribution Non-Commercial Share Alike license.
% http://creativecommons.org/licenses/by-nc-sa/3.0/
```

C.2.11 GoHere.m

```
function [ControlSignalForPlayer, FifoForPlayer] = GoHere(agentIndex, Target, TeamOwn, GameMode, CycleBatch, TeamCounter)

%-% This function tells a player to go to a location specified in Target.

global FUN HLSTraj

DesiredSpeedTime = 1;
TeamOwn{agentIndex}.Target=[Target 0 0];
[CS,TeamOwn{agentIndex}.Target,TeamOwn{agentIndex}.TargetSpeedTime]=...
    FUN.moveTo(agentIndex,TeamOwn,DesiredSpeedTime);

FifoForPlayer = FUN.U_TimeStamp(CS, GameMode(1)); %=% adds timestamps to controlsignals
[ControlSignalForPlayer, FifoForPlayer] = ...
    FUN.U_Shorten(FifoForPlayer, GameMode(1), CycleBatch);

HLSTraj{TeamCounter}{agentIndex}.data  = TeamOwn{agentIndex}.Target;
HLSTraj{TeamCounter}{agentIndex}.tst   = TeamOwn{agentIndex}.TargetSpeedTime;
HLSTraj{TeamCounter}{agentIndex}.index = 1;

```

```
%     2010
% Benjamin Bergman - ben.bergman@gmail.com
% Matthew Woelk - umwoelk@cc.umanitoba.ca
% This document is subject to the Creative Commons 3.0 Attribution Non-Commercial
    Share Alike license.
% http://creativecommons.org/licenses/by-nc-sa/3.0/
```

C.2.12 GraphDimmer.m

```
function matrix = GraphDimmer (PlayerPos, Dimmer)
%% Uses the Dimmer value to create a matrix that dims the field between the player
    and the player's goal.

global FieldX FieldY

matrix = [ones(FieldY-1, floor(PlayerPos(1)))*Dimmer, ones(FieldY-1, FieldX-floor(
    PlayerPos(1)))];

%     2010
% Benjamin Bergman - ben.bergman@gmail.com
% Matthew Woelk - umwoelk@cc.umanitoba.ca
% This document is subject to the Creative Commons 3.0 Attribution Non-Commercial
    Share Alike license.
% http://creativecommons.org/licenses/by-nc-sa/3.0/
```

C.2.13 GraphDontCamp.m

```
function matrix = GraphDontCamp()

%% This function maps a deadzone into the opponents goal so that we don't
    accidentally block shots on their goal.
%% The matrix returned from this function should be multiplied by the matrix used to
    choose player's destination.

global FUN Score
global Environment Team M FieldX FieldY

matrix = zeros(FieldY,FieldX);

distFromYAxis = repmat([1:FieldX], FieldY-1, 1);
distFromXAxis = repmat([1:FieldY-1]', 1, FieldX);
```

```

distToMidOppGoal = sqrt((distFromYAxis - FieldX).^2 + (distFromXAxis - FieldY/2).^2);
matrix = min(ones(size(distToMidOppGoal)), distToMidOppGoal/10.0);

%     2010
% Benjamin Bergman - ben.bergman@gmail.com
% Matthew Woelk - umwoelk@cc.umanitoba.ca
% This document is subject to the Creative Commons 3.0 Attribution Non-Commercial Share Alike license.
% http://creativecommons.org/licenses/by-nc-sa/3.0/
```

C.2.14 GraphField.m

```
function matrix = GraphField()

%-% This function maps the field in terms of the best locations to pass the ball to, not taking into account player positions, ball positions, or opponent positions.

global FUN Score
global Environment Team M FieldX FieldY

matrix = zeros(FieldY,FieldX);

%=% NB: Could probably replace for loops with repmat command for improved performance. Not deemed strictly necessary as this function should only be run once. See "GraphDontBlock" for example on how to replace with repmat.

ecks = [];
eck = 1:FieldX;
for n = 1:FieldY-1
  ecks = [ecks;eck];
end

why = [];
wh = (1:FieldY-1)';
for n = 1:FieldX
  why = [why wh];
end

distance = min(sqrt((ecks - 0).^2 + (why - FieldY/2).^2),...
                 sqrt((ecks - FieldX).^2 + (why - FieldY/2).^2));
resultMatrix = max((ecks < FieldX/2).*(sin((distance.*pi)./(FieldX/2) - pi./2) + 1)./4,...
                    (ecks >= FieldX/2).*((sin(((FieldX/2 -distance).*pi)./(FieldX/2) - pi./2) + 1)./4 + 0.5));
matrix = resultMatrix;


%-% Make a few spots in the center of the net always bright white:
matrix(FieldY/2+Environment.GoalSize/2 - 1:FieldY/2-Environment.GoalSize/2 + 1,FieldX) = 1;
%-% The value 1 is the ball radius.

%    2010
% Benjamin Bergman - ben.bergman@gmail.com
% Matthew Woelk - umwoelk@cc.umanitoba.ca
% This document is subject to the Creative Commons 3.0 Attribution Non-Commercial Share Alike license.
% http://creativecommons.org/licenses/by-nc-sa/3.0/
```

C.2.15 GraphMirror.m

```
function matrix = GraphMirror(inputmatrix)

%-% This function takes in a field-sized graph and
%-% outputs a graph that is the correct size to be
%-% used when calculating mirrored kicks.

sizes = size(inputmatrix);

matrix = zeros(sizes(1)*3,sizes(2));
height = sizes(1);
width = sizes(2);

matrix(1:height,1:width) = flipud(inputmatrix);
matrix(height+1:2*height,1:width) = inputmatrix;
matrix(2*height+1:3*height,1:width) = flipud(inputmatrix);

%    2010
% Benjamin Bergman - ben.bergman@gmail.com
% Matthew Woelk - umwoelk@cc.umanitoba.ca
% This document is subject to the Creative Commons 3.0 Attribution Non-Commercial Share Alike license.
% http://creativecommons.org/licenses/by-nc-sa/3.0/
```

C.2.16 GraphMoveOut.m

```
function matrix = GraphMoveOut()

%=% This function maps a deadzone into our goal so that we don't accidentally pass into our net and so we don't crowd the goalie.
%=% The matrix returned from this function should be multiplied by the matrix used to choose player's destination.

global FUN Score
global Environment Team M FieldX FieldY

matrix = zeros(FieldY,FieldX);

distFromYAxis = repmat([1:FieldX], FieldY-1, 1);
distFromXAxis = repmat([1:FieldY-1]', 1, FieldX);

distToMidOurGoal = sqrt((distFromYAxis).^2 + (distFromXAxis - FieldY/2).^2);
matrix = min(ones(size(distToMidOurGoal)), distToMidOurGoal/25.0);



%     2010
% Benjamin Bergman - ben.bergman@gmail.com
% Matthew Woelk - umwoelk@cc.umanitoba.ca
% This document is subject to the Creative Commons 3.0 Attribution Non-Commercial Share Alike license.
% http://creativecommons.org/licenses/by-nc-sa/3.0/
```

C.2.17 GraphPlayerPositions.m

```
function matrix = GraphPlayerPositions(PlayerPositions, Pos, displayOutput, radiusMultiplier, ignorePlayer)

%-% This function graphs a field where its values are dependent on the players' positions.

global FUN Environment Team M FieldX FieldY qDamp

ELLIPSEcircle = false; %-% Ellipse mode is not yet working properly.

matrix = ones(FieldY,FieldX)*0.5;

bx = Pos(1);
```

```
by = Pos(2);

point2 = []; %-% This is only used for Ellipse mode.
             %-% It is the point with b distance from the opponent's position, between the opponent and the goal. It is one of the two focal points of the ellipse. (The other is the opponent's position)

ecks = [];
eck = 1:FieldX;
for n = 1:FieldY-1
  ecks = [ecks;eck];
end

why = [];
wh = (1:FieldY-1)';
for n = 1:FieldX
  why = [why wh];
end

%-% For our team:
for inc = 1:M
  if inc ~= ignorePlayer
    px = PlayerPositions{inc}(1);
    py = PlayerPositions{inc}(2);

    r = 0.25*radiusMultiplier;
    k = FUN.Distance([bx,by],[px,py]);
    b = k.*sin(r);

    doubleradius = sqrt((ecks - px).^2 + (why - py).^2);
    multiplier = 1/b^2;
    if ELLIPSEcircle
      slope = (FieldY-by)/(FieldX/2-bx);
      i = b/sqrt(slope^2 + 1);
      j = i*slope;
      %abovenet = (py > FieldY/2)*2 - 1; %-% -1 if below the net, 1 if above
      point2 = [px-abovenet*i,py-abovenet*j];
      %point2 = [px-i,py-j];
      doubleradius = doubleradius + sqrt((ecks - point2(1)).^2 + (why - point2(2)).^2);
      multiplier = multiplier.*0.1
    end
    resultMatrix{inc} = max(1 - multiplier.*doubleradius.^2,0.0);
  end
```

```
end

if ignorePlayer ~= 1 && ignorePlayer ~= 2
  resultMatrix2 = (1-resultMatrix{1}).*(1-resultMatrix{2});
elseif ignorePlayer ~= 1
  resultMatrix2 = (1-resultMatrix{1});
else
  resultMatrix2 = (1-resultMatrix{2});
end
for ink = 3:M
  if ink  ~= ignorePlayer
    resultMatrix2 = resultMatrix2.*(1-resultMatrix{ink});
  end
end

if displayOutput
  figure(4);
  imshow(flipud(resultMatrix2));
end
matrix = max(0.001,resultMatrix2);

%     2010
% Benjamin Bergman - ben.bergman@gmail.com
% Matthew Woelk - umwoelk@cc.umanitoba.ca
% This document is subject to the Creative Commons 3.0 Attribution Non-Commercial Share Alike license.
% http://creativecommons.org/licenses/by-nc-sa/3.0/
```

C.2.18 GraphPlayerPositionsMir.m

```
function matrix = GraphPlayerPositionsMir(PlayerPositions, Pos, displayOutput, radiusMultiplier, ignorePlayer)

%-% This function graphs a field where its values are dependent on the players' positions.
%-% It takes into account rebounds when calculating where to kick.
%-% It also runs very slowly

global FUN Environment Team M FieldX FieldY qDamp

ELLIPSEcircle = false; %-% Ellipse mode is not yet working properly.

matrix = ones(FieldY*3,FieldX)*0.5;
```

```

bx = Pos(1);
by = Pos(2) + FieldY;

point2 = []; %-% This is the point with b distance from the opponent's position, between the opponent and the goal. It is one of the two focal points of the ellipse. (The other is the opponent's position)
            %-% It is only used for Ellipse mode.

ecks = [];
%eck = (1:FieldX/3)*3;
%for n = 1:FieldY/3-1
eck = 1:FieldX;
for n = 1:FieldY*3-1
  ecks = [ecks;eck];
end

why = [];
%wh = ((1:FieldY/3-1)*3)';
%for n = 1:FieldX/3
wh = (1:FieldY*3-1)';
for n = 1:FieldX
  why = [why wh];
end

ignoreNumbers = zeros(M*3);
if ignorePlayer > 0
  ignoreNumbers((ignorePlayer-1)*3 + 1) = 1;
  ignoreNumbers((ignorePlayer-1)*3 + 2) = 1;
  ignoreNumbers((ignorePlayer-1)*3 + 3) = 1;
end

howbig = size(PlayerPositions);
%-% set up doubles of the players:
for i = 1:howbig(2)
  if ~isempty(PlayerPositions{i})
    positions{(i-1)*3+1}(1) = PlayerPositions{i}(1);
    positions{(i-1)*3+1}(2) = PlayerPositions{i}(2) + FieldY;
    positions{(i-1)*3+2}(1) = PlayerPositions{i}(1);
    positions{(i-1)*3+2}(2) = FieldY - PlayerPositions{i}(2) + FieldY + FieldY;
    positions{(i-1)*3+3}(1) = PlayerPositions{i}(1);
    positions{(i-1)*3+3}(2) = FieldY - PlayerPositions{i}(2) - FieldY + FieldY;
  end
end
```

```

sizeofpos = size(positions);

%-% For our team:
for inc = 1:sizeofpos(2)
  if ~ignoreNumbers(inc) && ~isempty(positions{inc})
    px = positions{inc}(1);
    py = positions{inc}(2);

    r = 0.25*radiusMultiplier;
    k = FUN.Distance([bx,by],[px,py]);
    b = k.*sin(r);

    doubleradius = sqrt((ecks - px).^2 + (why - py).^2);
    multiplier = 1/b^2;
    if ELLIPSEcircle
      slope = (FieldY-by)/(FieldX/2-bx);
      i = b/sqrt(slope^2 + 1);
      j = i*slope;
      %abovenet = (py > FieldY/2)*2 - 1; %-% -1 if below the net, 1 if above
      point2 = [px-abovenet*i,py-abovenet*j];
      %point2 = [px-i,py-j];
      doubleradius = doubleradius + sqrt((ecks - point2(1)).^2 + (why - point2(2)).^2);
      multiplier = multiplier.*0.1
    end
    resultMatrix{inc} = max(1 - multiplier.*doubleradius.^2,0.0);
  else
    resultMatrix{inc} = [];
  end
end

resultMatrix2 = ones(FieldY*3-1,FieldX);

for ink = 1:sizeofpos(2)
  if ~ignoreNumbers(ink) && ~isempty(resultMatrix{ink})
    resultMatrix2 = resultMatrix2.*(1-resultMatrix{ink});
  end
end

if displayOutput
  figure(4);
  imshow(flipud(resultMatrix2));
end
```

```
matrix = max(0.001,resultMatrix2);
%-% The following is an easy work-around
matrix = matrix(1:FieldY*3-3,:);

%     2010
% Benjamin Bergman - ben.bergman@gmail.com
% Matthew Woelk - umwoelk@cc.umanitoba.ca
% This document is subject to the Creative Commons 3.0 Attribution Non-Commercial Share Alike license.
% http://creativecommons.org/licenses/by-nc-sa/3.0/
```

C.2.19 GraphShadows.m

```
function matrix = GraphShadows(PlayerPositions, Pos, displayOutput, radiusMultiplier)

%-% This function displays shadows behind opponents, which are the areas where the ball should not be passed.

global FUN Score
global Environment Team M FieldX FieldY


%-% This makes it so the last part of the code is the only part that is run!!!!!!!!
%-% (The last part is the only good part, and the rest can probably be deleted)
drawShadowFunction = false;
  drawShadowFunctionPolar = false;
  drawShadowFunctionCartesian = false;
    drawShadowFunctionCartesianPlayers = false;
drawShadowValues = false;
drawEfficientShadowValues = true;





if drawShadowFunction %-% This section generates a plot that shows the good and bad locations to shoot to based on the opponents' positions.
  ballx = Pos(1);
  bally = Pos(2);

  r = 0.25; %-% The radius of the semicircle in the polar coordinate system.
  b = 0; %-% The radius of the semicircle in the Cartesian plane.
```

```
h = 0;  %-% h is the angle in relation to the ball. It has the range: [-pi/2,+3pi/2)
k = 0; %-% k is the distance from the ball to the player.
numberOfValues = 300;
x = linspace(-pi,2*pi,numberOfValues); %-% x is a range of angles that will be cut down to [-pi/2,3pi/2).

for i = 1:M
  k(i) = sqrt((PlayerPositions{i}(2) - bally).^2 + (PlayerPositions{i}(1) - ballx).^2);
  b(i) = k(i).*sin(r);
  if PlayerPositions{i}(1) - ballx >= 0
    if PlayerPositions{i}(2) - bally == 0
      h(i) = 0;
    else
      h(i) = asin((PlayerPositions{i}(2) - bally)/k(i));
    end
  elseif PlayerPositions{i}(1) - ballx < 0
    h(i) = pi - asin((PlayerPositions{i}(2) - bally)/k(i));
  end
end

y = min(... %-% Combine the graphs of two of the opponents together.
      (-sqrt(((b(1)).^2).*(1 - ((x - h(1)).^2)/r.^2)) + k(1)).*... %-% Graph a semicircle
      (((1-heaviside(x - (h(1)-r)))+heaviside(x - (h(1)+r))).*499 + 1),... %-% Use the step function to isolate the semicircle
      (-sqrt(((b(2)).^2).*(1 - ((x - h(2)).^2)/r.^2)) + k(2)).*... %-% Graph a semicircle
      (((1-heaviside(x - (h(2)-r)))+heaviside(x - (h(2)+r))).*499 + 1));
for inc = 3:M %-% Combine the rest of the players too.
  y = min(y,...
        (-sqrt(((b(inc)).^2).*(1 - ((x - h(inc)).^2)/r.^2)) + k(inc)).*... %-% Graph a semicircle
        (((1-heaviside(x - (h(inc)-r)))+heaviside(x - (h(inc)+r))).*499 + 1));
end

y = real(y); %-% Ignore the imaginary parts of the plot.
y = max(y,(zeros(1,length(y)))); %-% This will get rid of negative values.

%-% Now we want the arcs from one angle to wrap around to the other side.
temp = min(y(1:numberOfValues/3),y(numberOfValues*2/3+1:numberOfValues)); %-% Take the minimum of the regions that wrap around.
y(1:numberOfValues/3) = temp; %-% Replace the x values with the new minimums
```

```
  y(numberOfValues*2/3+1:numberOfValues) = temp;


  figure(4)
  clf;
  hold on;
  set(gcf,'Position',[500 30 490 300])

  if drawShadowFunctionCartesian %-% This will plot in Cartesian coordinates.
    %-% Now we convert it to Cartesian coordinates.
    f{1} = y.*cos(x) + ballx;
    f{2} = y.*sin(x) + bally;
    plot(f{1},f{2});
    %plot3(x,f{1},f{2}); %-% Plot it in Cartesian coordinates.
    xlim([0 150]);
    ylim([0 100]);
  end

  if drawShadowFunctionCartesianPlayers %-% This plots the opponents, the ball, and our players on the Cartesian graph.
    for i = 1:M
      line([PlayerPositions{i}(1) PlayerPositions{i}(1)],[PlayerPositions{i}(2) PlayerPositions{i}(2)],'Marker','o','Color','black')
    end
    line([TeamOwnSave{1}(1) TeamOwnSave{1}(1)],[TeamOwnSave{1}(2) TeamOwnSave{1}(2)],'Marker','o','Color','red')
    line([TeamOwnSave{2}(1) TeamOwnSave{2}(1)],[TeamOwnSave{2}(2) TeamOwnSave{2}(2)],'Marker','o','Color','green')
    line([TeamOwnSave{3}(1) TeamOwnSave{3}(1)],[TeamOwnSave{3}(2) TeamOwnSave{3}(2)],'Marker','o','Color','blue')
    line([Pos(1) Pos(1)],[Pos(2) Pos(2)],'Marker','o','Color',[0.5 0.5 0.5])
    xlim([0 150]);
    ylim([0 100]);
  end

  if drawShadowFunctionPolar %-% This will plot in polar coordinates.
    plot(x,y);
    xlim([-pi/2 3*pi/2]);
    ylim([0 200]);
  end
end



```

```
if drawShadowValues %-%Calculate and display a matrix of coordinates that represent good passing spots.
  ballx = Pos(1);
  bally = Pos(2);
  r = 0.25; %-% The radius of the semicircle in the polar coordinate system.
  b = 0; %-% The radius of the semicircle in the Cartesian plane.
  h = 0;  %-% h is the angle in relation to the ball. It has the range: [-pi/2,+3pi/2)
  k = 0; %-% k is the distance from the ball to the player.
  for inc = 1:M
    k = sqrt((PlayerPositions{inc}(2) - bally).^2 + (PlayerPositions{inc}(1) - ballx).^2);
    b = k.*sin(r);
    if PlayerPositions{inc}(1) - ballx >= 0
      if PlayerPositions{inc}(2) - bally == 0
        h = 0;
      else
        h = asin((PlayerPositions{inc}(2) - bally)/k);
      end
    elseif PlayerPositions{inc}(1) - ballx < 0
      h = pi - asin((PlayerPositions{inc}(2) - bally)/k);
    end

    for i = 1:2:Environment.FieldSize(1)
      for j = 1:2:Environment.FieldSize(2)
        R = sqrt((j - bally).^2 + (i - ballx).^2);
        if i - ballx >= 0
          %%%%h) = asin((PlayerPositions{i}(2) - bally)/k(i));
          theta = asin((j - bally)./R);
        elseif i - ballx == 0 && j - bally == 0
          theta = 0
        else
          theta = pi - asin((j - bally)/R);
        end

        %-%This is to take care of angle wrapping:
        if j-bally<0 && i-ballx>0 && PlayerPositions{inc}(1)-ballx<0
          h2 = h - 2*pi;
        elseif j-bally<0 && i-ballx<0 && PlayerPositions{inc}(1)-ballx>0
          h2 = h + 2*pi;
        else
          h2 = h;
        end

        quant{inc}(j,i) = R > (-sqrt(((b).^2).*(1 - ((theta - h2).^2)/r.^2)) + k).*... %-% Graph a semicircle
                                  (((1-heaviside(theta - (h2-r)))+heaviside(theta - (h2+r))).*499 + 1);
      end
    end
  end

  quant2 = max(quant{1},quant{2});
  for ink = 3:M
    quant2 = max(quant2,quant{ink});
  end

  if drawShadowValues %-% This will graph the data as an upside-down image.
    figure(4);
    imshow(flipud(quant2));
    figure(5)
    imshow(flipud(quant{1}))
    figure(6)
    imshow(flipud(quant{2}))
    figure(7)
    imshow(flipud(quant{3}))
  end
end








if drawEfficientShadowValues %-%Calculate and display a matrix of coordinates that represent good passing spots, efficiently.
  matrix = zeros(FieldY,FieldX);

  ecks = [];
  eck = 1:FieldX;
  for n = 1:FieldY-1
    ecks = [ecks;eck];
  end

  why = [];
  wh = (1:FieldY-1)';
  for n = 1:FieldX
    why = [why wh];
  end

  bx = Pos(1);
  by = Pos(2);
  r = 0.25*radiusMultiplier; %-% The radius of the semicircle in the polar coordinate system.
  b = 0; %-% The radius of the semicircle in the Cartesian plane.
  %h = 0;  %-% h is the angle in relation to the ball. It has the range: [-pi/2,+3pi/2)
  k = 0; %-% k is the distance from the ball to the player.
  for inc = 1:M
    px = PlayerPositions{inc}(1);
    py = PlayerPositions{inc}(2);
    k = FUN.Distance([bx,by],[px,py]);
    b = k.*sin(r);

    multiplier = 1/b^2;

    distance = FUN.DistanceToLine2(ecks,why,bx,by,px,py,true);
    %resultMatrix{inc} = b > distance;
    resultMatrix{inc} = max(1 - multiplier.*distance.^2,0.0);
  end

  resultMatrix2 = (1-resultMatrix{1}).*(1-resultMatrix{2});
  for ink = 3:M
    resultMatrix2 = resultMatrix2.*(1-resultMatrix{ink});
  end

  if (displayOutput)
    figure(5);
    imshow(flipud(resultMatrix2));
  end
  matrix = resultMatrix2;
end


%    2010
% Benjamin Bergman - ben.bergman@gmail.com
% Matthew Woelk - umwoelk@cc.umanitoba.ca
% This document is subject to the Creative Commons 3.0 Attribution Non-Commercial
    Share Alike license.
% http://creativecommons.org/licenses/by-nc-sa/3.0/
```

C.2.20 GraphShadowsMir.m

```
function matrix = GraphShadowsMir(PlayerPositions, Pos, displayOutput,
    radiusMultiplier)

%-% This function displays shadows behind opponents, which are the areas where the
    ball should not be passed.
%-% It also takes into account rebounds when calculating where to kick.

%-% This currently doesn't take into account the radius of the ball.
%-% - (But we don't need that much accuracy for what we're doing.)
global FUN Score
global Environment Team M FieldX FieldY



matrix = zeros(FieldY*3-2,FieldX);

ecks = [];
eck = 1:FieldX;
for n = 1:FieldY*3-1
  ecks = [ecks;eck];
end

why = [];
wh = (1:FieldY*3-1)';
for n = 1:FieldX
  why = [why wh];
end

%-% set up doubles of the opponents:
for i = 1:M
  positions{(i-1)*3+1}(1) = PlayerPositions{i}(1);
  positions{(i-1)*3+1}(2) = PlayerPositions{i}(2) + FieldY;
  positions{(i-1)*3+2}(1) = PlayerPositions{i}(1);
  positions{(i-1)*3+2}(2) = FieldY - PlayerPositions{i}(2) + FieldY + FieldY;
  positions{(i-1)*3+3}(1) = PlayerPositions{i}(1);
  positions{(i-1)*3+3}(2) = FieldY - PlayerPositions{i}(2) - FieldY + FieldY;
end
```

```


bx = Pos(1);
by = Pos(2) + FieldY;
r = 0.25*radiusMultiplier; %-% The radius of the semicircle in the polar coordinate system.
b = 0; %-% The radius of the semicircle in the Cartesian plane.
%h = 0;  %-% h is the angle in relation to the ball. It has the range: [-pi/2,+3pi/2)
k = 0; %-% k is the distance from the ball to the player.
for inc = 1:M*3
  px = positions{inc}(1);
  py = positions{inc}(2);
  k = FUN.Distance([bx,by],[px,py]);
  b = k.*sin(r);

   multiplier = 1/b^2;

   distance = FUN.DistanceToLine2(ecks,why,bx,by,px,py,true);
  %resultMatrix{inc} = b > distance;
   resultMatrix{inc} = max(1 - multiplier.*distance.^2,0.0);
end

resultMatrix2 = (1-resultMatrix{1}).*(1-resultMatrix{2});
for ink = 3:M*3
   resultMatrix2 = resultMatrix2.*(1-resultMatrix{ink});
end

if (displayOutput)
  figure(5);
  imshow(flipud(resultMatrix2));
end
%-% The following is an easy work-around
matrix = resultMatrix2(1:FieldY*3-3,:);

%    2010
% Benjamin Bergman - ben.bergman@gmail.com
% Matthew Woelk - umwoelk@cc.umanitoba.ca
% This document is subject to the Creative Commons 3.0 Attribution Non-Commercial Share Alike license.
% http://creativecommons.org/licenses/by-nc-sa/3.0/
```

C.2.21 `GraphShadowsStatic.m`

```
function matrix = GraphShadowsStatic(TeamOppSave, CurrentPlayer, displayOutput, radiusMultiplier)

%-% This function displays shadows behind opponents, which are the areas where the players should not move.
%-% The shadows are circles, and their radii don't change size.

global FUN Score
global Environment Team M FieldX FieldY



%-%Calculate a matrix of coordinates that represent good passing spots.
matrix = zeros(FieldY,FieldX);

ecks = [];
eck = 1:FieldX;
for n = 1:FieldY-1
  ecks = [ecks;eck];
end

why = [];
wh = (1:FieldY-1)';
for n = 1:FieldX
  why = [why wh];
end

b = 30*radiusMultiplier; %-% The radius of the semicircle in the Cartesian plane.

for inc = 1:M
  px = TeamOppSave{inc}.Pos(1);
  py = TeamOppSave{inc}.Pos(2);

  multiplier = 1/b^2;

  %-%distance = FUN.DistanceToLine2(ecks,why,bx,by,px,py,true);
  %-%distance = sqrt((ecks-px).^2 + (why-py).^2);
  distance = FUN.DistanceToLine2(ecks,why,TeamOppSave{CurrentPlayer}.Pos(1), TeamOppSave{CurrentPlayer}.Pos(2),px,py,true);
  %resultMatrix{inc} = b > distance;
  resultMatrix{inc} = max(1 - multiplier.*distance.^2,0.0);
end


```

```
if CurrentPlayer == 1
  resultMatrix2 = 1-resultMatrix{2};
elseif CurrentPlayer == 2
  resultMatrix2 = 1-resultMatrix{1};
else
  resultMatrix2 = (1-resultMatrix{1}).*(1-resultMatrix{2});
end

for ink = 3:M
  if ink ~= CurrentPlayer
    resultMatrix2 = resultMatrix2.*(1-resultMatrix{ink});
  end
end

if (displayOutput)
  figure(5);
  imshow(flipud(resultMatrix2));
end
matrix = resultMatrix2;


%     2010
% Benjamin Bergman - ben.bergman@gmail.com
% Matthew Woelk - umwoelk@cc.umanitoba.ca
% This document is subject to the Creative Commons 3.0 Attribution Non-Commercial Share Alike license.
% http://creativecommons.org/licenses/by-nc-sa/3.0/
```

C.2.22 `GraphSides.m`

```
function matrix = GraphSides()

%-% This function makes a graph that has dimmed edges
%-% It is used to make players stay more toward the center of the field.

global FUN Score FieldX FieldY

%-% This makes a nice curve that keeps players away from the edges of the field.
fandle = @(x) (1.17 - 1./(x./4+0.8));

distFromYAxis = repmat([1:FieldX], FieldY-1, 1);
distFromXAxis = repmat([1:FieldY-1]', 1, FieldX);

```

```
distFromSide = min(fandle(distFromXAxis),fandle(FieldY - distFromXAxis));
distFromSide = min(distFromSide,fandle(distFromYAxis));
distFromSide = min(distFromSide,fandle(FieldX - distFromYAxis));

matrix = min(1,distFromSide);
matrix = max(0,distFromSide);


%    2010
% Benjamin Bergman - ben.bergman@gmail.com
% Matthew Woelk - umwoelk@cc.umanitoba.ca
% This document is subject to the Creative Commons 3.0 Attribution Non-Commercial Share Alike license.
% http://creativecommons.org/licenses/by-nc-sa/3.0/
```

C.2.23 IntersectPoints.m

```
function locations = IntersectPoints(TeamOwn,PlayerTargets,engagePosition,MaxKickVel,offset,engagingPlayer,Fifo,GameMode)

%-% This function calculates where the ball would be able to contact each player
%-% based on where the player is currently going.
%-% also based on the ball's max speed being MaxKickVel.

global FUN M

locations = [];

for i = 1:M
  %-% Create a fake ball with the speed that we will kick it at.
  if i ~= engagingPlayer
    kickertarget = PlayerTargets{i};
    maxvel = MaxKickVel;
    delx =  engagePosition(1) - kickertarget(1);
    dely =  engagePosition(2) - kickertarget(2);
    velx = sqrt(maxvel.^2./(1+(dely.^2/delx.^2)));
    vely = sqrt(maxvel.^2 - velx.^2);
    targetVector = [-velx -vely];
    fakeBall.Pos = [engagePosition(1:2) targetVector];

    [xpos,ypos,cycles] = FUN.PassIntersection(TeamOwn{i}.Pos,TeamOwn{i}.Type,Fifo{i},GameMode,engagePosition,MaxKickVel,offset);
    %-%[xpos,ypos,cycles] = FUN.Intersection(TeamOwn{i}.Pos,TeamOwn{i}.Type,engagePosition,offset);
    locations{i} = [xpos ypos];
    if locations{i} < 0
      locations{i} = PlayerTargets{i};
    end
  else
    locations{i} = [];
  end
end



%     2010
% Benjamin Bergman - ben.bergman@gmail.com
% Matthew Woelk - umwoelk@cc.umanitoba.ca
% This document is subject to the Creative Commons 3.0 Attribution Non-Commercial Share Alike license.
% http://creativecommons.org/licenses/by-nc-sa/3.0/
```

C.2.24 Intersection.m

```
function [ xpos, ypos, cycles ] = Intersection ( PlayerPos, Type, BallPos, Offset )

%INTERSECTION Returns the position and number of cycles in the future
%  where the provided player can intersect the ball.
%  If the agent cannot intercept the ball within PredictCycles, the pos
%  returned is [-1, -1] and cycles is inf.
%  Note that this function calculates the absolute minimum time for an
%  agent to contact the ball, not the time it will take to make a
%  calculated kick.


%-% Offset will most often be the number of cycles before the player kicks the ball.

global FUN

BallRadius = 1;
PredictCycles = 70;
BallPredict = FUN.BallPrediction(BallPos, PredictCycles);

if ~exist('Offset', 'var')
  Offset = 0;
```

```
end

for i = Offset+1:PredictCycles
  j = i - Offset;
  distVector = (BallPredict(j, 1:2) - PlayerPos(1:2));
  unitVector = distVector/norm(distVector);


  Count = FUN.U_Count( 0, U_TurnAngle( asin(unitVector(2)) ), Type, (Type.MaxSpeed/Type.Parameters(2)) );



  PlayerLoc = unitVector.*Type.MaxSpeed.* (i - Count) + PlayerPos(1:2);
  if norm(PlayerLoc - BallPredict(j,1:2)) < (Type.BoundingRadius + BallRadius)
    %=% collide near here
    xpos = PlayerLoc(1);
    ypos = PlayerLoc(2);
    cycles = i;
    return
  end
end

%=% Player is unable to contact ball.
PlayerLoc = [-1 -1];
xpos = PlayerLoc(1);
ypos = PlayerLoc(2);
cycles = inf;


%     2010
% Benjamin Bergman - ben.bergman@gmail.com
% Matthew Woelk - umwoelk@cc.umanitoba.ca
% This document is subject to the Creative Commons 3.0 Attribution Non-Commercial Share Alike license.
% http://creativecommons.org/licenses/by-nc-sa/3.0/
```

C.2.25 isBallGoingForGoal.m

```
function isBallGoingForGoal = isBallGoingForGoal( Ball )

%-% This function determines whether the ball is heading for the opponent's goal.
%=% It does not take into account rebounds off of the walls.
```

```

global FieldX FieldY Environment

%Received parameters
Bx=Ball.Pos(1);
By=Ball.Pos(2);
BVx=Ball.Pos(3);
BVy=Ball.Pos(4);

isBallGoingForGoal = 0;

if BVx > 0
  if BVx || BVy ~= 0
    slope = BVy/BVx;
    ua = slope*(FieldX - Bx) + By;

    if ua < FieldY/2+Environment.GoalSize/2 && ua > FieldY/2-Environment.GoalSize/2
      isBallGoingForGoal = 1;
    end
  end
end


%     2010
% Benjamin Bergman - ben.bergman@gmail.com
% Matthew Woelk - umwoelk@cc.umanitoba.ca
% This document is subject to the Creative Commons 3.0 Attribution Non-Commercial Share Alike license.
% http://creativecommons.org/licenses/by-nc-sa/3.0/
```

C.2.26 `isBallGoingForOurGoal.m`

```
function [isBallGoingForGoal wallIntersection] = isBallGoingForOurGoal( Ball )


%-% This function determines whether the ball is heading for our goal.
%=% This function does not take into account rebounds off of the walls.

global FieldX FieldY Environment

%Received parameters
Bx=Ball.Pos(1);
By=Ball.Pos(2);
```

```
BVx=Ball.Pos(3);
BVy=Ball.Pos(4);

isBallGoingForGoal = false;
wallIntersection = 0;

if BVx < 0
  slope = BVy/BVx;
  wallIntersection = slope*(-Bx) + By;

  if wallIntersection < FieldY/2+Environment.GoalSize/2 && wallIntersection > FieldY/2-Environment.GoalSize/2
    isBallGoingForGoal = true;
  end
end


%     2010
% Benjamin Bergman - ben.bergman@gmail.com
% Matthew Woelk - umwoelk@cc.umanitoba.ca
% This document is subject to the Creative Commons 3.0 Attribution Non-Commercial Share Alike license.
% http://creativecommons.org/licenses/by-nc-sa/3.0/
```

C.2.27 isKicking.m

```
function isKicking = isKicking( Fifo );
%ISKICKING Returns a boolean telling us if the agent is in the middle
% of a kick. The input is the Fifo for the agent in question.
% NOTE: The fifos for each agent should be stored as a persistent
% variable, likely in the HLS function. When a kick is to be abandoned
% the fifo for the kicking player should be emptied, ie. set to [].

isKicking = ~isempty(Fifo);
%=% NB: should this do any truncating?


%     2010
% Benjamin Bergman - ben.bergman@gmail.com
% Matthew Woelk - umwoelk@cc.umanitoba.ca
% This document is subject to the Creative Commons 3.0 Attribution Non-Commercial Share Alike license.
% http://creativecommons.org/licenses/by-nc-sa/3.0/
```

C.2.28 Kick.m

```
function [CS, Fifo] = Kick( Fifo, TeamCounter, agentIndex, GameMode, BallTrajBackup, PlayerTrajBackup );
%KICK Generates a control signal for the provided player using the
% provided Fifo. The returned CS and Fifo are properly timestamped
% and have all previously executed commands cut out.
% Player's HLSTraj information is set during KICK. It should only be
% manually set if you assign different commands to the robot.
%
% Outputs:
% CS:                     This is the properly formatted and truncated
%                         control signal needed by the simulator
% Fifo:                   This is the fifo of all future moves needed
%                         to complete the current kick. This fifo is
%                         only for the current player.
%
% Inputs:
% Fifo:                   This is the fifo of moves we want to format
%                         for use by the simulator. This should only
%                         be the fifo of the current player
% TeamCounter:            This is the number of the current player's
%                         team.
% agentIndex:             This is the agent number of the current player.
% GameMode:               This is the GameMode variable passed from the
%                         simulator.
% BallTrajBackup:         This is the backup of the BallTraj used in a
%                         continued kick. If a new kick is being
%                         performed, this does not need to be set as it
%                         was set with CANKICK.
%                   NOTE: There is only one BallTraj per team, so be
%                         careful if multiple agents are running the
%                         CANKICK function.
% PlayerTrajBackup:       Similar to BallTrajBackup.

global FUN

global M
global BallTraj HLSTraj
global CycleBatch

```

```
if exist('BallTrajBackup', 'var') && exist('PlayerTrajBackup')
  BallTraj{TeamCounter} = BallTrajBackup;
  HLSTraj{TeamCounter}{agentIndex} = PlayerTrajBackup;
end

%=% Add time stamping and create control signal
Fifo = FUN.U_TimeStamp(Fifo, GameMode(1));
[CS, Fifo] = FUN.U_Shorten(Fifo, GameMode(1), CycleBatch);


%     2010
% Benjamin Bergman - ben.bergman@gmail.com
% Matthew Woelk - umwoelk@cc.umanitoba.ca
% This document is subject to the Creative Commons 3.0 Attribution Non-Commercial Share Alike license.
% http://creativecommons.org/licenses/by-nc-sa/3.0/
```

C.2.29 moveTo.m

```
function [ControlSignal,Target,Time]=moveTo(agentIndex,TeamOwn,DesiredSpeedTime)

% This function provides control signal for moving the agent to a desired target
% (TeamOwn{agentIndex}.Target)

global FUN

[U, Ori2] = FUN.Get_Control_Signal(TeamOwn{agentIndex}.Pos, TeamOwn{agentIndex}.Target, TeamOwn{agentIndex}.Type, DesiredSpeedTime);
ControlSignal=U;
Target=TeamOwn{agentIndex}.Target;
[s,o]=size(ControlSignal);
Time=s;

%     2010
% Benjamin Bergman - ben.bergman@gmail.com
% Matthew Woelk - umwoelk@cc.umanitoba.ca
% This document is subject to the Creative Commons 3.0 Attribution Non-Commercial Share Alike license.
% http://creativecommons.org/licenses/by-nc-sa/3.0/
```

C.2.30 PassIntersection.m

```
function [ xpos, ypos, cycles ] = PassIntersection ( PlayerPos, PlayerType, PlayerFifo, GameMode, BallPos, BallVel, Offset )

%PASSINTERSECTION Returns the position and number of cycles in the
%  future where the provided player and ball can collide. This point
%  is based on the provided Fifo of moves for the player and the
%  speed of the ball. The offset is the time until we can kick the
%  ball at the provided speed. The BallPos should be the position
%  at that same kick time.
%  Note that this function calculates the absolute minimum time for
%  the ball to get to the agent's path. This does not consider the
%  receiving agent slowing down for a kick, nor does it consider how
%  the passing angle affects the first agent's wind up.


global FUN

BallRadius = 1;
PredictCycles = 70;
PlayerPredict = FUN.PlayerPrediction( PlayerPos, PlayerFifo, PredictCycles, GameMode );
if ~exist('BallVel', 'var') || BallVel == 0
  BallVel = norm(BallPos(3:4));
end
if ~exist('Offset', 'var')
  Offset = 0;
end

for i = Offset+1:PredictCycles
  j = i - Offset;
  distVector = PlayerPredict(i, 1:2) - BallPos(1:2);
  unitVector = distVector/norm(distVector);

  FakeBall = [BallPos(1:2), BallVel*unitVector(1:2)];
  BallPredict = FUN.BallPrediction(FakeBall, PredictCycles); %% NB: don't need this many cycles, but this will always be sufficient

  if (norm(BallPredict(j, 1:2) - PlayerPredict(i, 1:2)) < (PlayerType.BoundingRadius + BallRadius)) && ~(norm(PlayerPredict(i, 1:2) - [0 0]) < 0.1)
    %% there is a collision

    xpos = PlayerPredict(i, 1);
    ypos = PlayerPredict(i, 2);
    cycles = i;
    return
  end
end



%% Player is unable to contact ball.
xpos = -1;
ypos = -1;
cycles = inf;


%     2010
% Benjamin Bergman - ben.bergman@gmail.com
% Matthew Woelk - umwoelk@cc.umanitoba.ca
% This document is subject to the Creative Commons 3.0 Attribution Non-Commercial Share Alike license.
% http://creativecommons.org/licenses/by-nc-sa/3.0/
```

C.2.31 `PlayerPrediction.m`

```
function PredictedCoordinates = PlayerPrediction( PlayerPos, PlayerFifo, cycles, GameMode )

%PLAYERPREDICTION This function predicts where the player will
%  be based on a given set of control signals stored in the
%  fifo. This function will return an array of length 'cycles'
%  containing the predicted future coordinates for the agent
%  in each cycle. The fifo passed to PLAYERPREDICTION must be
%  timestamped.

global FUN

[CS, Fifo] = FUN.U_Shorten( PlayerFifo, GameMode(1), 20 ); %% 20 is an arbitrary value. It is only used to construct CS which we don't care about right now.

PredictedCoordinates = zeros(cycles, 4);

cycles = min(cycles, size(Fifo, 1));

for i=1:cycles
  if Fifo(i, 2)   %% we are moving forward
    %% use the current angle vector to find x and y to add to position
    MoveVector = PlayerPos(3:4) * Fifo(i, 2);
    NewPos = PlayerPos(1:2) + MoveVector;
    PredictedCoordinates(i, :) = [ NewPos, PlayerPos(3:4) ];
    PlayerPos = PredictedCoordinates(i, :);
  elseif Fifo(i, 3)   %=% we are turning
    %=% use the rotation amount and the current angle vector to find a new angle vector

    x = PlayerPos(3);
    y = PlayerPos(4);

    %=% adapted from wikipedia's polar coordinates article
    %=% use of the atan2 function might make this code simpler
    if x > 0
      angle = atan(y/x);
    elseif x < 0 && y >=0
      angle = atan(y/x) + pi;
    elseif x < 0 && y < 0
      angle = atan(y/x) - pi;
    elseif x == 0 && y > 0
      angle = pi/2;
    elseif x == 0 && y < 0
      angle = -pi/2;
    elseif x == 0 && y == 0
      angle = 0;
    end

    NewAngle = angle + Fifo(i, 3);

    NewDir = [ cos(NewAngle), sin(NewAngle) ];

    PredictedCoordinates(i, :) = [ PlayerPos(1:2), NewDir ];
    PlayerPos = PredictedCoordinates(i, :);
  else
    if i ~= 1
      %=% if the number of moves in the fifo is shorter than the requested cycles, assume agents stops here
      PredictedCoordinates(i, :) = PredictedCoordinates(i-1, :);
      PlayerPos = PredictedCoordinates(i, :);
    else
      PredictedCoordinates(i, :) = PlayerPos;
    end
  end
end
```

```


% 2010
% Benjamin Bergman - ben.bergman@gmail.com
% Matthew Woelk - umwoelk@cc.umanitoba.ca
% This document is subject to the Creative Commons 3.0 Attribution Non-Commercial Share Alike license.
% http://creativecommons.org/licenses/by-nc-sa/3.0/
```

C.2.32 `timeLeftInKick.m`

```
function cycles = timeLeftInKick( PlayerFifo, GameMode )

% It is assumed that this function will be called before the Fifo
% gets modified by HLS.

global FUN

[CS, Fifo] = FUN.U_Shorten(PlayerFifo, GameMode(1), 20); %=% 20 is an arbitrary value. It is only used to construct CS which we don't care about right now

cycles = size(Fifo, 1);


% 2010
% Benjamin Bergman - ben.bergman@gmail.com
% Matthew Woelk - umwoelk@cc.umanitoba.ca
% This document is subject to the Creative Commons 3.0 Attribution Non-Commercial Share Alike license.
% http://creativecommons.org/licenses/by-nc-sa/3.0/
```

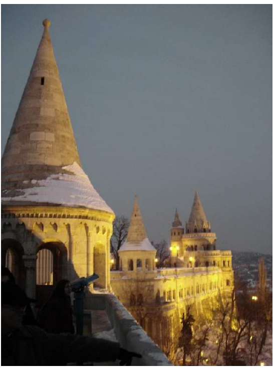
Appendix D

Experience Reports

This appendix contains two experience reports, one for each Benjamin and Matthew. These reports discuss the non-technical portion of the mobility program in which the project discussed in this report was done.

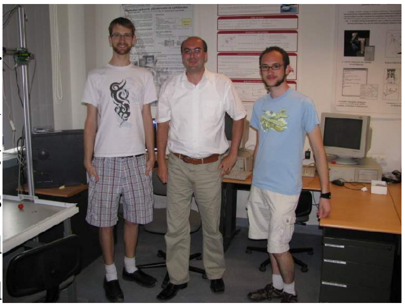
D.1 Benjamin Bergman's Experience report

Thanks to the C&C of MARS international studies program, as well as the great professors at the University of Manitoba and Budapest University of Technology and Economics, I was able to spend five months working on my final year group design project abroad in Budapest, Hungary. There I worked with a fellow University of Manitoba student, Matthew Woelk, on developing a set of high level strategies for a robotic soccer simulation.



I first learned of this program in an email sent out to the engineering mailing list at the University of Manitoba. At first I thought there would be no way my application would be accepted, but it looked like such a good experience, I decided to apply anyway. Low and behold, a couple of weeks later I was told I had been accepted! After discussing the program with the Computer Engineering department heads, we were told that we could use the project as our Final Year Group Design project. This was the first time anyone had travelled as part of their project, so we were extremely excited.

One year later, I found myself on a plane flying high above the Atlantic Ocean on my way to Central Europe. The weeks leading up to the flight, and even the flight itself, felt like a dream. It did not become a reality until we set foot on Hungarian soil and realized that we now had to find a way to get to our hotel. As my first time living away from home, this was intimidating for the first few days, but I very quickly adapted to my new environment and all of its challenges.



Once we had found a flat and settled in, we set to work on our project. This was my first time working on a project of this kind of scale in a group of just two. It was both extremely challenging and rewarding. I learned a lot about managing large projects (breaking them into smaller sections, researching both broad and specific projects, organizing large collections of work and

research, etc.), working with a single parter on a project (coordinating efforts, data synchronization, compromising on opinions, etc.), and working in a foreign environment where not everyone can speak the same language as I do.

While our work kept us extremely busy, working upwards of 7 hours per day, 5 days a week, our schedule was flexible enough that we were able to take some time to have fun in and around Budapest. There were lots of international students from all over the globe studying in BME's English exchange program. We were invited to join in with them at all of their events which were a lot of fun. There were many parties, trips to other towns and cities, and activities within Budapest. I think the most fun party was the international party where all the people from each country got together and made customary dishes from their home country. Since Canada is a fairly new country which is very multicultural, we had a hard time deciding what to bring. In the end the 3 of us living on the one flat made a big batch of poutine while the two guys in the other flat brought pirogi pizza. Having a chance to try some authentic foods from across the world, all in one night, was a real treat.

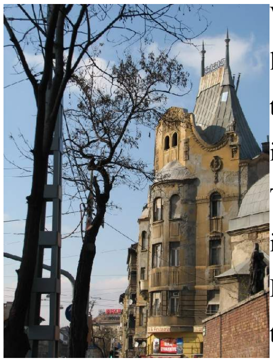
As for trips, the international students went to many places. All of these trips were great. We went to Eger, Hungary for a “get to you know” event. Eger is known for its wines and we got to take part in a wine tasting. We also went to Bratislava, Slovakia, and Krakow, Poland. In Poland we also went to Auschwitz concentration camp. While I cannot say this was a fun experience, it was definitely a good one. It was very sobering to see real evidence of what happened in WWII.

I think that my favourite activity was when we went caving through the Buda hills. We donned coveralls, hardhats, and headlamps and set out through some old caves. There were some massive rooms, and there were also lots of small passages where you had to contort your body in odd ways to squeeze through. It was lots of fun. We also toured a science and inventions museum featuring all sorts of things that originated in

Hungary. There are lots of amazing things that have and will come out of Hungary.

Following our time working on the project, some family came out to visit us and all of use took a few weeks to travel around Europe. We were able to see Croatia (Zagreb), Austria (Vienna and Salzburg), Germany (Munich and Bamburg), and the Czech Republic (Prague).

All the food I got to try all over Europe was amazing, especially within Hungary. I managed to find many dishes, mostly deserts, that were made from cottage cheese, and I loved every one of them. The Hungarian sausage, lángos, goulash, and pretty much everything else I tried was also very good.

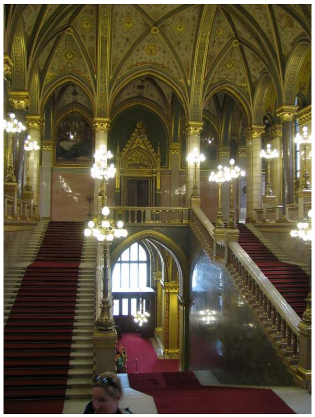


Working on this project abroad was a very rewarding experience. I was able to learn about many topics which I had not previously known, I learned some of the intricacies of organizing large projects in a group dynamic, and most importantly, I was able to interact and connect with people all over the world. The Budapest University of Technology and Economics did a great job involving us in all of the activities for the European exchange students. Having the chance to study in Europe allowed me to truly appreciate things that other cultures just take as the norm. While this alone would have been worth the trip, being away from home for so long and learning as much as I did about these other cultures made me really grateful for the things I take for granted living in Canada.

All in all, I have absolutely no regrets about doing this program. If I had to do it all again, I would ask to do it 10 times more. If anyone has any doubts about doing this program, I highly recommend giving it a try. It truly is a once in a lifetime experience.

My father, myself, my brother, Matt, and Matt's father, in the courtyard of Neuschwanstein Castle, just outside Munich, Germany

St. Steven's Basilica in the heart of Budapest



Inside the extravagant Hungarian Parliament building

Szechenyi Spa in the centre of City Park, Budapest

"Small" side garden at Schönbrunn Palace, Vienna

Gelért Hill near the Budapest University of Technology and Economics

Horse races around Hero Square during a national festival

One of many beautiful views in the gorgeous town of Bamburg, Germany

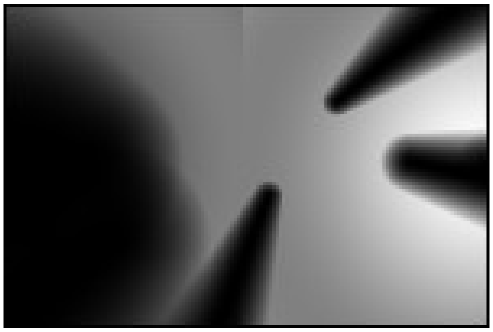
One of the many serene lakes in the picturesque Plitvice Lakes National Park, Croatia

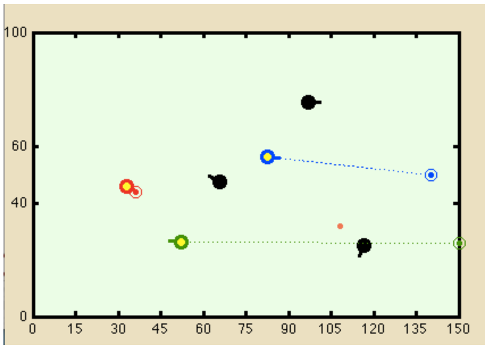
D.2 Matthew Woelk's Experience report

Through the C&CofMARS (Control and Coordination of Multi-Agent Robotics Systems) inter-University mobility program, my undergraduate final year design project was able to be done in Budapest, Hungary at the Budapest University of Technology and Economics. A bursary was supplied by the program to help pay for the costs of flights, food, and accommodation for the trip.

The project that Benjamin Bergman and I worked on was to create a simulated soccer team which ran using the soccer simulator that had been created by previous students. Our task was to make a team that would successfully defeat all of the other teams which had been created previously. The project was both difficult and rewarding. We learned about many other strategies that had been implemented, and in the end, made one of our own using the Matlab environment. The laboratory atmosphere was productive and warm, and everyone was friendly and willing to help out wherever they could.

The city of Budapest was a beautiful place, especially in the spring time. Margaret Island, which has one of the most gorgeous parks in Budapest, had a running track around it which was 5.3km long and provided a constant view of the city skyline. The sights to see were also captivating. Fisherman's Bastion was a castle with a beautiful church on top of a hill. It provided a fantastic view of the parliament building, which is the second largest in Europe. The food was also incredible. Gulyás soup and Lángos were definite highlights, as well as all of the pastries which could be purchased at any metro station around town. At the end of spring was a festival which included horse racing in a track that was set up around Heros' Square. Many traditional shops were set up around the square selling musical instruments, jewelry, and all kinds of delicious food, my favourite of which is Kenyér Lángos: deep-fried dough with meat and cheese on it. A close second place winner for my favourite Hungarian food would be Kürtőskalács, which is a spiral cylinder of baked sugar dough.





The university did an excellent job of integrating me with the Erasmus Student Network (ESN), which organized events for the foreign students. Because of this, I was able to meet students from all over Europe and the world. It was fantastic to be able to interact with people who all came from a different place, and had upbringings and ideas. Through the program I went spelunking in the caves of Budapest, and among other things, got to travel around Europe!

During the program, through ESN and by other means, on weekends I got to go to Kraków, Poland; Bratislava, Slovakia; Eger, Hungary; Szentendre, Hungary; and Munich, Germany. Each of these places provided a different perspective of how to live, bringing with it their own sights, sounds, and tastes.

Being in Europe also allowed me to travel to places after the

project was over that I would not have had the opportunity to visit otherwise. These included Germany, Austria, Croatia, and Czech Republic.

I value having had the opportunity to see all of these different cultures and groups of people. It's difficult to see objectively when there's nothing to compare them to. I feel that through this trip I have gained a greater understanding of both where I've been, and where I've come from.

Overall, this trip was an excellent experience. It put me in a place where I could learn many things, both technical and otherwise, and I have grown much because of this opportunity. I have also gained and grown valuable friendships through this program, and it has provided a growing experience which I would not have otherwise been able to have.

Thank you to Istvan Harmati, Jean Paul Burak, Nariman Sepehri, Wiltold Kinsner, and Cyrus Shafai for making this voyage possible and for coordinating the program.