

Universidade Federal do Maranhão - UFMA  
Centro de Ciências Exatas e Tecnologia - CCET  
Coordenadoria do Curso de Ciência da Computação – COCOM

JAMESSON AMARAL GOMES

**Criação de um Time de Futebol Robótico  
para o *SIMULATION LEAGUE* do *RoboCup***

São Luís

2017

JAMESSON AMARAL GOMES

**Criação de um Time de Futebol Robótico  
para o *SIMULATION LEAGUE* do *RoboCup***

Monografia apresentada ao curso de Ciência da Computação da Universidade Federal do Maranhão, como parte dos requisitos necessários para obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Prof. Dr. Paulo Rogério De Almeida Ribeiro

São Luís

2017

Ficha gerada por meio do SIGAA/Biblioteca com dados fornecidos pelo(a)  
autor(a).Núcleo Integrado de Bibliotecas/UFMA

Amaral Gomes, Jamesson

CRIAÇÃO DE UM TIME DE FUTEBOL ROBÓTICO PARA O  
SIMULATION LEAGUE DO ROBOCUP / Jamesson Amaral Gomes. -  
2017

56p.

Orientador: Prof. Dr. Paulo Rogério De Almeida Ribeiro.  
Monografia (Graduação) - Curso de Ciência da Computação,  
Universidade Federal do Maranhão, São Luís,  
2017.

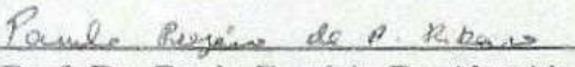
1. Futebol robótico. 2. Simulação 2D. 3. Sistemas multiagente.  
I. De Almeida Ribeiro, Paulo Rogério. II. Título.

JAMESSON AMARAL GOMES

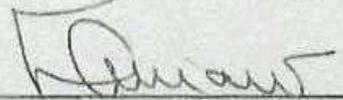
## Criação de um Time de Futebol Robótico para o Simulation League do Robocup

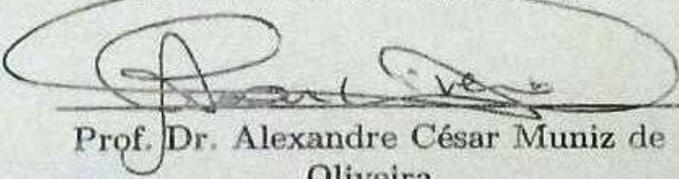
Monografia apresentada ao curso de Ciência da Computação da Universidade Federal do Maranhão, como parte dos requisitos necessários para obtenção do grau de Bacharel em Ciência da Computação.

Aprovada em São Luís, 27 de Janeiro de 2017.

  
Prof. Dr. Paulo Rogério De Almeida  
Ribeiro  
Orientador  
Universidade Federal do Maranhão

  
Prof. Dr. Geraldo Braz Junior  
Examinador 1  
Universidade Federal do Maranhão

  
Prof. Dr. Luciano Reis Coutinho  
Examinador 2  
Universidade Federal do Maranhão

  
Prof. Dr. Alexandre César Muniz de  
Oliveira  
Examinador 3  
Universidade Federal do Maranhão

São Luís  
2017

# Agradecimentos

Agradeço primeiramente a Deus, meus pais e familiares. Especialmente à minha mãe, Ivanilde, que como pedagoga sempre me incentivou a buscar a educação e erudição como prioridade e sempre propiciou as condições necessárias para o meu desenvolvimento pessoal.

Agradeço também a Juliana Solrac que como companheira de todas as horas nos momentos bons e ruins, incentivou-me a finalizar esta graduação.

Agradeço aos professores e amigos desta instituição, aos colegas de laboratório (LACMOR) e ao colega de curso Rodrigo Garcês, por todo o conhecimento adquirido e incentivo dado para concretização deste trabalho. Em especial ao orientador e amigo Paulo Rogério, o qual foi determinante para o desenvolvimento deste projeto.

*Ainda que eu ande pelo vale da sombra da morte,  
não temerei mal algum, pois tu estás comigo,  
a tua vara e o teu cajado me protegem...  
(Bíblia Sagrada, SALMOS 23, 4)*

# Resumo

O futebol robótico apresenta-se como um problema ideal para combinar Robótica e Inteligência Artificial, uma vez que apresenta um ambiente dinâmico, engloba um sistema multiagente e é um esporte coletivo. Adicionalmente, ressalta-se que o futebol de robôs pode integrar outras áreas de pesquisa, por exemplo pesquisa operacional, controle de processos, processamento de imagem, realidade virtual entre outras.

Este trabalho descreve a simulação em duas dimensões do futebol de robôs do *Robot World Cup Initiative* (RoboCup), detalhando a estrutura do ambiente de simulação e desenvolvimento do futebol robótico, objetivando incentivar a criação de times e principalmente estratégias inteligentes por estudantes e pesquisadores da área, uma vez que tem-se uma baixa difusão do tema na comunidade acadêmica maranhense.

A partir da escolha e estudo do time base UVA Trilearn - Holanda, foram desenvolvidas três estratégias básicas (Chutar ao Gol, Chutar de acordo com a coordenada do jogador e Drible) usando a linguagem de programação C++. A combinação desse time base e das estratégias possibilitou a criação do primeiro time de futebol de robôs simulado na UFMA.

Tais estratégias foram aplicadas em partidas contra o time base Agent2D -Japão e o próprio time criado neste trabalho. Os resultados obtidos em partidas contra o Agent2D demonstraram a necessidade de melhorias devido a superioridade técnica do Agent2D, enquanto que as partidas contra um time sem essa superioridade técnica mostraram-se mais equilibradas. Essas melhorias a serem realizadas demonstram o potencial do time recém-criado, assim como criam novos horizontes para a robótica e para outras áreas de pesquisa.

**Palavras-chave:** *sistemas multiagente, futebol robótico, simulação2D, UVA Trilearn.*

# Abstract

Robotic soccer is an ideal problem to combine Robotics and Artificial Intelligence, since it is a dynamic environment, encompasses a multi-agent system and is a collective sport. Moreover, robotic soccer can integrate other areas of research, for instance operational research, process control, image processing, virtual reality and so on.

This work describes the 2D Soccer Simulation League of the Robot World Cup Initiative (RoboCup), elucidating the simulation structure and development environment, aiming to encourage the creation of teams and mainly intelligent strategies by students and researchers in this research field, since this is not a well known subject in the Maranhão's academic community.

Based on the choice and study of the UVA Trilearn - Netherlands base team, three basic strategies (Kick to Goal, Kick according to player's coordinate, and Dribble) have been developed using the C++ programming language. The combination of this base team and the strategies enabled the creation of the first simulated robotic soccer team at UFMA.

These strategies were applied in matches against the team base Agent2D - Japan and the team created in this work. The results against Agent2D demonstrated the need for improvements due to the mastery of Agent2D, whereas matches against a team without this technical superiority proved to be more balanced. These improvements to be implemented demonstrate the potential of the newly created team as well as it breaks new ground for robotics and other research fields.

**Key-words:** *multiagent systems, robotic soccer, 2D simulation, UVA Trilearn*

# Lista de ilustrações

Figura 1 – Humanoid League . . . . .	16
Figura 2 – Middle Size League . . . . .	16
Figura 3 – Small Size League . . . . .	17
Figura 4 – Standard Platform League . . . . .	17
Figura 5 – Partida do RoboCup 2D Soccer Simulation League em execução . . . . .	18
Figura 6 – 3D Simulation League . . . . .	19
Figura 7 – Arquitetura do SoccerServer . . . . .	20
Figura 8 – Arquitetura do time UvA TriLearn . . . . .	26
Figura 9 – Diagrama UML da arquitetura do UVA Trilearn . . . . .	27
Figura 10 – Dependências da Classe PlayerTeams do UvA Trilearn 2003 . . . . .	31
Figura 11 – Eixos X e Y do campo. . . . .	33
Figura 12 – Aplicação da Estratégia I. . . . .	35
Figura 13 – Aplicação da Estratégia II. . . . .	37
Figura 14 – Aplicação da Estratégia III. . . . .	38

# Lista de tabelas

Tabela 1 – Protocolo de Conexão . . . . .	21
Tabela 2 – Protocolo de Percepção . . . . .	22
Tabela 3 – Protocolo de Ação . . . . .	23
Tabela 4 – Resultados da Estratégia I da partida UVA I x Agent2D. . . . .	39
Tabela 5 – Resultados da Estratégia II da partida UVA II x Agent2D. . . . .	40
Tabela 6 – Resultados da Estratégia III da partida UVA III x Agent2D. . . . .	40
Tabela 7 – Gol sofridos e marcados pelo UVA II na partida UVA II X UVA III . .	40

# Lista de abreviaturas e siglas

FIFA	<i>Fédération Internationale de Football Association</i>
FIRA	<i>Federation of International Robot-Soccer Association</i>
GPL	<i>General Public License</i>
PET	<i>Programa de Educação Tutorial</i>
RCSS	<i>Robocup Soccer Simulator</i>
TDP	<i>Team Description Paper</i>
UDP	<i>User Datagram Protocol</i>
UML	<i>Unified Modeling Language</i>
UVA I	<i>Time UVA Trilearn com estratégia I</i>
UVA II	<i>Time UVA Trilearn com estratégias I e II</i>
UVA III	<i>Time UVA Trilearn com estratégias I e III</i>

# Sumário

<b>1</b>	<b>INTRODUÇÃO</b>	<b>13</b>
<b>1.1</b>	<b>Objetivos</b>	<b>14</b>
1.1.1	Objetivos Específicos	14
<b>1.2</b>	<b>Justificativa</b>	<b>14</b>
<b>2</b>	<b>FUNDAMENTAÇÃO TEÓRICA</b>	<b>15</b>
<b>2.1</b>	<b>RoboCup</b>	<b>15</b>
2.1.1	Robocup Soccer – Humanoid	16
2.1.2	Robocup Soccer - Middle Size	16
2.1.3	Robocup Soccer - Small Size	17
2.1.4	Robocup Soccer - Standard Platform	17
2.1.5	RoboCup Soccer - Simulation League	18
2.1.5.1	RoboCup Soccer – 2D Simulation League	18
2.1.5.2	RoboCup Soccer – 3D Simulation League	19
<b>2.2</b>	<b>2D Simulation League – SoccerServer</b>	<b>19</b>
2.2.1	2D Simulation League – Comunicação e Percepção	20
2.2.2	Ações no Futebol Robótico	22
<b>2.3</b>	<b>Times Base</b>	<b>24</b>
2.3.1	WrightEagle Base	25
2.3.2	Helios Base (Agent2D)	25
2.3.3	UvA TriLearn Base	26
2.3.4	BAHIA2D	28
<b>3</b>	<b>ESTRATÉGIAS E JOGADAS PARA O UVA TRILEARN BASE</b>	<b>30</b>
<b>3.1</b>	<b>Estratégia I: Chutar ao gol</b>	<b>31</b>
<b>3.2</b>	<b>Estratégia II: Chutar de acordo com a coordenada do jogador</b>	<b>35</b>
3.2.1	Estratégia III: Drible	36
<b>4</b>	<b>RESULTADOS - ESTRATÉGIAS PROPOSTAS.</b>	<b>39</b>
<b>4.1</b>	<b>Estratégia I: Chutar ao Gol</b>	<b>39</b>
<b>4.2</b>	<b>Estratégia II: Chutar de Acordo com a Coordenada do Jogador</b>	<b>39</b>
<b>4.3</b>	<b>Estratégia III: Drible</b>	<b>40</b>
<b>4.4</b>	<b>Estratégia II e III Aplicadas em uma partida do UVA II x UVA III</b>	<b>40</b>
<b>5</b>	<b>CONCLUSÃO</b>	<b>42</b>

<b>REFERÊNCIAS</b> . . . . .	<b>43</b>
<b>APÊNDICES</b>	<b>45</b>
<b>APÊNDICE A – INSTALAÇÃO E CONFIGURAÇÃO DE ARQUI- VOS</b> . . . . .	<b>46</b>
<b>APÊNDICE B – CÓDIGO COM NOVAS ESTRATÉGIAS PARA O PLAYERTEAMS.CPP</b> . . . . .	<b>51</b>

# 1 Introdução

O futebol é um dos esportes mais praticados no mundo e possui como principal característica a cooperação entre os jogadores de um mesmo time para alcançar a vitória. Por ser um ambiente imprevisível, não determinístico e dinâmico, tem sido considerado um problema padrão para a Inteligência Artificial e gerado vários tópicos de pesquisa em diversas áreas, tais como: sistemas com múltiplos agentes, algoritmos de cooperação, inteligência distribuída, aprendizagem, reconhecimento de padrões e robótica inteligente (SILVA; SIMÕES; ARAGÃO, 2007).

Criada em 1996 (KITANO et al., 1997), a RoboCup fornece um problema padrão, o Futebol de Robôs, onde variadas linhas de pesquisa podem ser integradas e examinadas, tais como: desenvolvimento de agentes autônomos, colaboração entre agentes, raciocínio em tempo real e robótica. A comunidade internacional RoboCup promove o desenvolvimento de robôs inteligentes definindo e executando competições que são usadas por cientistas e estudantes de todo o mundo para testar e demonstrar seus robôs em cenários mais próximos da realidade possível.

A primeira competição oficial organizada pela Robocup foi realizada no ano de 1997. Desde então a RoboCup tornou-se uma iniciativa internacional que visa encorajar pesquisas na área de Inteligência Artificial, com o objetivo de capacitar robôs para a realização autônoma de atividades e formação de equipes na realização de tarefas cooperativas.

Atualmente existem duas iniciativas que abordam o tema: a RoboCup (Robot World Cup) Federation (KITANO et al., 1997) e a FIRA (Federation of International Robot-Soccer Association). As duas se diferenciam na abordagem do problema. Enquanto a FIRA apenas aborda a solução de problemas onde os robôs possuem baixo grau de autonomia, a RoboCup fornece problemas mais complexos envolvendo robôs completamente autônomos. Essas ligas ganharam notoriedade internacional e continuam realizando campeonatos anualmente. As duas entidades possuem diversas categorias, que envolvem desde robôs simulados até robôs bípedes. Além do futebol, essas organizações ainda realizam outros tipos de competições envolvendo robôs (resgate de vítimas em escombros, robôs domésticos e categorias infantis) (SILVA; SIMÕES; ARAGÃO, 2007).

As competições de futebol robótico simulado são consideradas um laboratório de pesquisa para sistemas multiagentes, onde a categoria de simulação 2D, que utiliza um emulador virtual, provê um ambiente similar à realidade, com variáveis como atrito, inércia, ruído, ação do vento, etc (TEIXEIRA, 2011). Tais características permitiram o surgimento de times base, que podem ser usados por equipes iniciantes que desejam desenvolver suas próprias estratégias, a fim de participar das competições nacionais e internacionais.

## 1.1 Objetivos

Pretende-se avaliar entre os times bases disponíveis na plataforma RoboCup, qual se adequa melhor ao desenvolvimento de algumas estratégias básicas e iniciais, para assim criar o 1º time de futebol robótico simulado do Maranhão. Esse time virá a subsidiar pesquisadores e estudantes que desejam ingressar nesta plataforma e conciliar suas linhas de pesquisa com este time base a ser desenvolvido.

### 1.1.1 Objetivos Específicos

Entre os objetivos específicos estão:

- Elucidar conceitos e características únicas da simulação do futebol de robôs;
- Criação de estratégias e jogadas básicas, avaliando o desempenho do time em partidas simuladas;
- Estudo dos códigos fonte, processo de comunicação com ambiente de simulação e detalhes da plataforma;
- Fornecer um time base que poderá ser utilizado por pesquisadores de diversas áreas como: robótica, inteligência artificial, controle de processos, etc.

## 1.2 Justificativa

O Brasil, diferentemente do futebol real, não tem tradição no futebol robótico. A *RoboCup Soccer - Simulation League* é uma das ligas que possui mais equipes brasileiras, no entanto nenhuma maranhense. Visando estimular a inserção do Maranhão em competições de robótica nacionais e internacionais, para um maior desenvolvimento de tecnologias e pesquisas na área de robótica, iniciar-se-á o desenvolvimento de um time de futebol robótico.

Portanto, a realização deste projeto adicionará o primeiro time maranhense nas competições de robótica e conseqüentemente mais um time brasileiro ao RoboCup, além de providenciar uma plataforma para testes de diversos algoritmos e estratégias desenvolvidos por alunos e pesquisadores da UFMA, assim como capacitará melhor alunos na área de pesquisa.

## 2 Fundamentação Teórica

Esta seção visa descrever as principais ligas e sub ligas do RoboCup Soccer, assim como aprofundar estudos sobre a plataforma utilizada pela subliga de simulação 2d, descrevendo detalhes de seu funcionamento e da arquitetura do simulador de partidas do futebol robótico. Realizar também o detalhamento de algum times base para assim fundamentar a escolha de um deles e iniciar-se o desenvolvimento de estratégias básicas para o time a ser criado neste trabalho.

### 2.1 RoboCup

A RoboCup propõe um problema padrão a ser resolvido por diferentes pesquisadores ao redor do mundo, tendo como vantagem o fato de que diferentes soluções podem ser encontradas para o problema. Essas soluções são devidamente compartilhadas pela comunidade científica através de um documento, chamado *Team Description Paper* (TDP), que deve ser submetido anualmente por cada equipe que deseja participar do evento.

Nas primeiras edições do RoboCup existia apenas a liga *RoboCup Soccer*, no entanto, posteriormente novas ligas foram criadas, como: a *RoboCup Rescue*, *RoboCup@Home*, *RoboCup Industrial* e *RoboCupJunior*. Todas as ligas do RoboCup adotam robôs completamente autônomos, ou seja, não existe nenhum comando externo para o controle dos robôs.

A principal competição da RoboCup é o futebol, onde fomenta-se as pesquisas em ambientes cooperativos multi-robôs e sistemas multi-agentes nas adversidades de um ambiente dinâmico, no qual todos os robôs são totalmente autônomos, conforme descrito em (FRACCAROLI, 2011).

A liga RoboCup Soccer divide-se em: *standard platform league* (anteriormente denominada *four legged league*); *mall size league*; *middle size league*; e *simulation league*. As subligas *standard platform league*, *small size league* e *middle size league* têm robôs reais, enquanto a *simulation league* usa simuladores.

Entretanto, a *simulation league* tem como vantagem em relação as outras subligas: ser uma plataforma de testes para algoritmos sem custos de *hardware* com os robôs; ser um ambiente para implementação em alto nível sem detalhes de *hardware*; e oferece uma investigação replicável e robusta de sistemas robóticos complexos (BUDDEN et al., 2015). A descrição de cada categoria da liga RoboCup Soccer é feita a seguir:

### 2.1.1 Robocup Soccer – Humanoid

A categoria Humanoid (Figura 1) possui robôs bípedes para cada time, e visa a pesquisa em áreas como: visão computacional, auto localização, comunicação, entre outras. Esses robôs tem o corpo semelhante ao corpo humano tanto nas características físicas como sensoriais (SILVA, 2015).

Figura 1 – Humanoid League



Fonte: (ROBOCUPGALLERY, 2015)

### 2.1.2 Robocup Soccer - Middle Size

Na categoria middle size (Figura 2) cada time pode ter 4 robôs com até 75 cm de altura e 50 cm de diâmetro, onde todos os sensores são do tipo *on-board* e visa a pesquisa em áreas como: auto localização, visão computacional, integração de sensores, percepção distribuída e controle de movimentos do robô.

Figura 2 – Middle Size League



Fonte: (ROBOCUPGALLERY, 2009)

### 2.1.3 Robocup Soccer - Small Size

A categoria *small size* (F180) (Figura 3) contém times com até 5 jogadores e que pode ter até 15 cm de altura e 18 cm de diâmetro. Suas áreas de pesquisa são as mesmas presentes na categoria *middle size*, diferenciando apenas no tamanho dos robôs. O foco é dado no problema de inteligência cooperativa multi-robôs/agentes e controle em um ambiente altamente dinâmico.

Figura 3 – Small Size League



Fonte: (ROBOCUPGALLERY, 2009)

### 2.1.4 Robocup Soccer - Standard Platform

A categoria *standard platform* (Figura 4) contém robôs idênticos em cada equipe diferindo apenas no software desenvolvido por cada uma, sendo os robôs totalmente autônomos e não havendo controladores externos, nem por interferência humana ou computacional. A plataforma usada atualmente são os humanoides *NAO*, desenvolvidos pela empresa francesa *Aldebran Robotics* (ALDEBRAN, 2016).

Figura 4 – Standard Platform League



Fonte: (ROBOCUPGALLERY, 2009)

### 2.1.5 RoboCup Soccer - Simulation League

A subliga *simulation league* do RoboCup Soccer, diferentemente das outras subligas – *standard platform league*, *small size league* e *middle size league* que usam robôs reais, usa um ambiente virtual para a realização dos jogos. A *simulation league* é dividida em *RoboCup 2D Soccer Simulation League* e *RoboCup 3D Soccer Simulation League*, sendo a primeira com um ambiente de simulação em duas dimensões enquanto a segunda usa um ambiente com três dimensões. A Figura 5 mostra um jogo da *RoboCup 2D Soccer Simulation League*.

Figura 5 – Partida do RoboCup 2D Soccer Simulation League em execução



#### 2.1.5.1 RoboCup Soccer – 2D Simulation League

A liga de simulação do RoboCup é baseada no simulador conhecido como *Soccer-Server* (REIS, 2003) que simula um jogo virtual de futebol disputado entre duas equipes compostas por 11 jogadores autônomos cada. O simulador foi construído como um ambiente de simulação multiagente, incerto e com funcionamento em tempo-real, capaz de permitir a competição entre equipes de jogadores virtuais, cada qual controlada separadamente por um agente autônomo.

A simulação em duas dimensões é disputada por dois times com 11 agentes (jogadores) de linha e um agente técnico. Cada agente representa um jogador de futebol que recebe várias informações contendo diversas situações de jogo, com alguns ruídos propositalmente introduzidos pelo servidor do simulador do de partidas. Com isso, cada jogador deve ser capaz de inferir sob a melhor estratégia no momento, seja ela individual ou coletiva, assim como no futebol convencional.

Disputada no Brasil desde 2005, as regras de jogo são muito semelhantes às de uma partida real, mantidas pela *Fédération Internationale de Football Association* (FIFA). Também são utilizadas algumas regras específicas da RoboCup como em um caso que haja um time sobrecarregando o servidor com mensagens a fim de atrasar o jogo, este será punido.

### 2.1.5.2 RoboCup Soccer – 3D Simulation League

A liga simulada em três dimensões possibilita a competição entre robôs humanoides em uma simulação realista tanto nas regras quanto na parte física de uma partida de futebol. A plataforma reproduz os desafios de programação de software enfrentados ao construir robôs físicos reais para o futebol (SIMULATION, 2012), conforme a Figura 6.

Figura 6 – 3D Simulation League



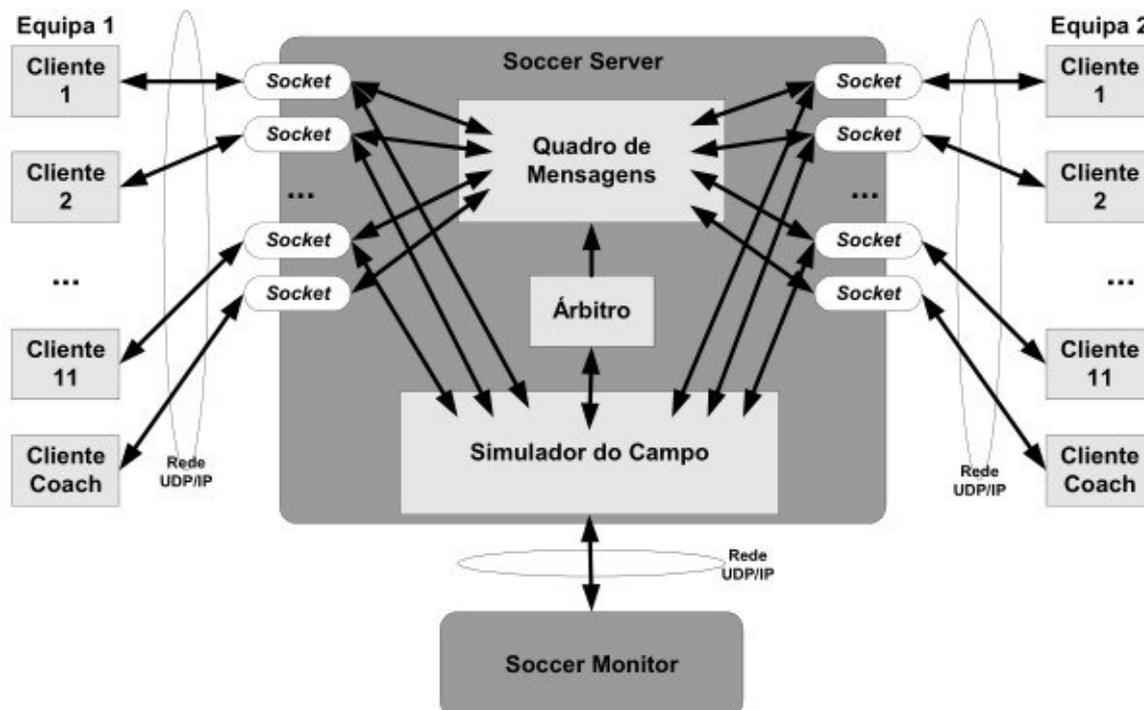
Fonte: (Wikipedia, 2010)

## 2.2 2D Simulation League – SoccerServer

O ambiente de simulação é representado por um servidor central chamado *Soccer Server* (ROBOCUP 2D SOCCER SIMULATION LEAGUE). Os jogadores e o coach conectam-se ao servidor para enviar e receber informações. Essa comunicação é realizada usando a arquitetura cliente-servidor através de sockets *User Datagram Protocol* (UDP). A Figura 7 mostra a arquitetura do SoccerServer, onde verifica-se que cada cliente é um processo separado e controla um único jogador.

Uma partida é realizada seguindo o padrão cliente/servidor, cada cliente controla os movimentos de um jogador. A comunicação entre eles é realizada através de soquetes UDP/IP. Depois de estabelecida a conexão, o cliente envia sinais de controle para um jogador e recebe de volta informações dos sensores áudio-visuais do jogador. Estes sinais

Figura 7 – Arquitetura do SoccerServer



Fonte: (REIS, 2003)

sufrem propositalmente a intervenção do servidor que acopla ruídos às informações, para dificultar mais ainda o ambiente, testando mais profundamente o agente.

Além disso a Figura 7 mostra os elementos presentes durante a simulação do jogo: jogadores e treinador que são representados por clientes, quadro de mensagens, árbitro, simulador e campo, e o *soccer monitor* que é usado para visualização dos jogos conforme visto na Figura 5. Como demonstrado na arquitetura, não é permitida uma comunicação direta entre os jogadores, uma mensagem de um determinado jogador deve ser enviada para o servidor que a envia para o outro jogador.

### 2.2.1 2D Simulation League – Comunicação e Percepção

A comunicação entre clientes e o servidor é regida por um conjunto de três protocolos: conexão, ação e percepção. Os clientes devem possuir capacidades que lhes permitam seguir um determinado conjunto de protocolos de comunicação com o servidor. O protocolo de conexão permite aos clientes ligarem-se, desligarem-se e reconectar-se ao servidor conforme descrito na Tabela 1.

Como o *Soccer Server* é um sistema em constante evolução, permite a conexão de clientes que utilizam versões do protocolo distintas, estabelecendo desta forma que clientes desenvolvidos para versões anteriores do *Soccer Server* possam-se conectar ao sistema. No caso do cliente se conectar com um protocolo superior ou igual a versão 7.0, então receberá

Tabela 1 – Protocolo de Conexão

<i>Conexão (Cliente para o Servidor)</i>	<i>Conexão (Resposta do Servidor)</i>
<p><b>(init</b>&lt;NomeEquipe&gt;[(version&lt;NumVer&gt;)][(goalie)]            &lt;NomeEquipe&gt; ::= (- _ a-z A-Z 0-9)            &lt;NumVer&gt; ::= versão do protocolo de comunicação a utilizar (p.e. 7.0)</p>	<p><b>(init</b>&lt;Lado&gt;&lt;Unum&gt;&lt;ModoJogo&gt;            &lt;Lado&gt; ::= l r            &lt;Unum&gt; ::= 1-11            &lt;ModoJogo&gt; ::= um dos modos de jogo válido (<b>error</b> no_more_team_or_player) (<b>error</b> reconnect)</p>
<i>Reconexão (Cliente para o Servidor)</i>	<i>Resposta do Servidor</i>
<p><b>(reconnect</b>&lt;NomeEquipe&gt;&lt;Unum&gt;            &lt;NomeEquipe&gt; ::= (- _ a-z A-Z 0-9)</p>	<p><b>(init</b>&lt;Lado&gt;&lt;Unum&gt;&lt;ModoJogo&gt;            &lt;Lado&gt; ::= l r            &lt;Unum&gt; ::= 1-11            &lt;ModoJogo&gt; ::= um dos modos de jogo válido (<b>error</b> no_more_team_or_player) (<b>error</b> reconnect)</p>
<i>Desconexão (Cliente para o Servidor)</i>	<i>Desconexão (Resposta do Servidor)</i>
<p><b>(bye)</b></p>	

Fonte: (REIS, 2003)

adicionalmente a informação relativa aos parâmetros do servidor e aos parâmetros dos jogadores heterogêneos disponíveis (REIS, 2003).

De forma similar, ocorre o processo de comunicação entre o treinador e os jogadores, evitando assim uma comunicação direta entre o único elemento da equipe que recebe informação livre de ruído, neste caso o treinador, e os que recebem informação com ruído, jogadores.

Os jogadores enviam comandos que são executados pelo servidor durante um ciclo e recebem informações atuais do ciclo, por exemplo: velocidade e posição dos jogadores e da bola. Adicionalmente, as condições físicas dos jogadores são enviadas pelo servidor. Essa simulação da condição física tem como objetivo evitar que os agentes corram continuamente no campo à máxima velocidade, pois os jogadores têm energia limitada (REIS, 2003) e essa energia diminui durante as jogadas, levando o jogador a ficar cansado. A percepção dos clientes está dividida em três partes distintas: percepção auditiva (*hear*), visual (*see*) e sensorial (*sense\_body*), conforme a Tabela 2:

Assim as mensagens recebidas pela classe *Body* possuem um formato específico do simulador, e devem ser tratadas pelo mesmo para que as informações contidas na mensagem sejam extraídas e armazenadas. As mensagens enviadas pelo servidor são:

- **(server\_param (parâmetro valor) (parâmetro valor)\_(parâmetro valor))**: esta mensagem contém os parâmetros utilizados na simulação, como *kickable\_margin* que define a distância mínima entre o jogador e a bola para que um chute possa ser executado.
- **(see tempo (nomeDoObjeto distancia direção mudançaDeDistancia mudançaDeDireção direçãoCorpo direçãoCabeça))**: nesta mensagem o agente recebe informações sensoriais sobre seu campo de visão e os objetos.

Tabela 2 – Protocolo de Percepção

<p><b>Percepção (Servidor para o Cliente)</b></p> <p>(<b>hear</b> &lt;Tempo&gt;&lt;Emissor&gt;&lt;Mensagem&gt;)          &lt;Tempo&gt; ::= ciclo de simulação do simulador          &lt;Emissor&gt; ::= <i>online_coach_left</i>   <i>online_coach_right</i>   <i>referee</i>   <i>self</i>   &lt;Direção&gt;          &lt;Direção&gt; ::= -180..180          &lt;Mensagem&gt; ::= [string]</p>
<p>(<b>see</b> &lt;Tempo&gt;&lt;InfoObj&gt;)          &lt;Tempo&gt; ::= ciclo de simulação do simulador          &lt;InfoObj&gt; ::= (&lt;NomeObj&gt; &lt;Distância&gt; &lt;Direção&gt; &lt;DistVar&gt; &lt;DirVar&gt; &lt;DirCorpo&gt;          &lt;DirCabeça&gt;)   (&lt;NomeObj&gt; &lt;Distância&gt; &lt;Direção&gt; &lt;DistVar&gt; &lt;DirVar&gt;)   (&lt;NomeObj&gt;          &lt;Distância&gt; &lt;Direção&gt;)   (&lt;NomeObj&gt; &lt;Direção&gt;)          &lt;NomeObj&gt; ::= (p &lt;NomeEquipe&gt; [&lt;Unum&gt; [goalie]])   (b)   g [l r]   (l [l r t b])   (f c)   (f [l c r] [t b])   (f p [l r]          [t c b])   (f g [l r] [t b])   (f [l r t b] 0)   (f [t b] [l r] [10 20 30 40 50])   (f [l r] [t b] [10 20 30])          &lt;Distância&gt; ::= Real positivo          &lt;Direção&gt; ::= [-180.0 .. 180.0]          &lt;DistVar&gt; ::= Real          &lt;DirVar&gt; ::= Real          &lt;DirCorpo&gt; ::= [-180.0, 180.0]          &lt;DirCabeça&gt; ::= [-180.0, 180.0]          &lt;NomeEquipe&gt; ::= [string]          &lt;Unum&gt; ::= [1..11]</p>
<p>(<b>sense_body</b> &lt;Tempo&gt;(view_mode high/low narrow/normal/wide) (stamina &lt;Energia&gt; &lt;Esforço&gt;))          (speed &lt;ValorVel&gt; &lt;DirVel&gt;)(head_angle &lt;DirCabeça&gt;)(kick &lt;ContagemKicks&gt;)(dash          &lt;ContagemDashes&gt;)(turn &lt;ContagemTurns&gt;)(say &lt;ContagemSays&gt;)(turn_neck          &lt;ContagemTurnNecks&gt;)(catch &lt;ContagemCatches&gt;)(move &lt;ContagemMoves&gt;)(change_view          &lt;ContagemChangeViews&gt;)) &lt;Tempo&gt; ::= ciclo de simulação do simulador &lt;Energia&gt; ::= [0..4000]          &lt;Esforço&gt; ::= [0..1.0]          &lt;ValorVel&gt; ::= real positivo          &lt;DirVel&gt; ::= [-180.0 .. 180.0]          &lt;DirVel&gt; ::= [-180.0 .. 180.0]          &lt;DirCabeça&gt; ::= [-180.0 .. 180.0] &lt;Contagem*&gt; ::= inteiro positivo (para todas as contagens)</p>

Fonte: (REIS, 2003)

- (**hear tempo origem “Mensagem”**): esta mensagem possui um propósito geral de comunicação na simulação, o campo “origem” define quem enviou a mensagem e é utilizada principalmente para mensagens de alteração de modo de jogo, nesse caso a mensagem é enviada pelo *referee*, através do Monitor.
- (**sense\_body ((atributo valor) (atributo valor)\_\_(atributo valor))**): neste ponto o agente recebe informações sobre seus próprios atributos como: força, velocidade, etc.

### 2.2.2 Ações no Futebol Robótico

Para os agentes decidirem quando devem efetuar um chute ao gol, tocar a bola ao companheiro ou tomar a bola do adversário, primeiro devem conhecer o ambiente à sua volta e quais as possibilidades de movimentos existentes. Conforme descrito no protocolo de ação da Tabela 3, há três tipos de sensores que permitem aos jogadores realizarem essas ações com eficiência conforme o tipo do jogador.

O sensor visual (**see**) detecta as distâncias e as direções dos objetos e jogadores.

Tabela 3 – Protocolo de Ação

<b>Ação (Cliente para o Servidor)</b>
( <i>dash</i> <Potência>) <Potência> ::= [-100, 100]
( <i>turn</i> <Momento>) <Momento> ::= [-180, 180]
( <i>move</i> <PosX><PosY>) <PosX> ::= [-52.5, 52.5] <PosY> ::= [-34.0, 34.0]
(( <i>kick</i> <Potência><Direção>) <Potência> ::= [-100, 100] <Direção> ::= [-180, 180]
( <i>tackle</i> <Potência>) <Potência> ::= [0, 100]
( <i>catch</i> <Direção>) <Direção> ::= [-180, 180]
( <i>turn_neck</i> <Direção>) <Direção> ::= [-180, 180]
( <i>change_view</i> <Abertura> <Qualidade>) <Abertura> ::= narrow   normal   wide <Qualidade> ::= high   low
( <i>attentionto</i> <Equipe> <Unum>)   ( <i>attentionto</i> off) <Equipe> ::= opp   our   l   r   left   right   <Nome_Equipe>
( <i>say</i> <Mensagem>) <Mensagem> ::= “Texto”
( <i>pointto</i> <Distância> <Direção>)   ( <i>pointto</i> off) <Distância> ::= Inteiro <Direção> ::= [-180, 180]
<b>Ação (Resposta do Servidor)</b>
( <i>error unknown_command</i> ) ( <i>error ilegal_command_form</i> ) ( <i>score</i> <Tempo> <GolosEquipe> <GolosOponente>) para comandos ( <i>score</i> ) ( <i>sense_body</i> <Tempo> (view_mode high low narrow normal wide) (stamina <Energia> <Esforço>) (speed <ValorVel> <DirVel>) (head_angle <DirCabeça>) (kick <ContagemKicks>) (dash <ContagemDashes>) (turn <ContagemTurns>) (say <ContagemSays>) (turn_neck <ContagemTurnNecks>) (catch <ContagemCatches>) (move <ContagemMoves>) (change_view <ContagemChangeViews>) ) para comandos ( <i>sense_body</i> )

Fonte: (REIS, 2003)

O sensor auditivo (*hear*) detecta as mensagens enviadas pelo árbitro, treinador, colegas de time e adversários. A audição dos jogadores, nas competições, está limitada a uma distância máxima de 50 metros, exceto para os treinadores e árbitro que não têm limites de distância para serem ouvidos. O sensor físico (*sense\_body*) detecta o estado do jogador, incluindo a sua energia, velocidade e ângulo do pescoço em relação ao corpo.

Para os jogadores executarem ações, o simulador tem comandos parametrizáveis à disposição dos agentes durante o período de jogo de futebol simulado. Existem quatro tipos de ações principais: movimento (*dash*, *turn* e *move*); interação com a bola (*kick*, *tackle* e *catch*); controle de percepção (*turn\_neck*, *attentionto*, *change\_view*) e comunicação (*say*, *pointto*). Os agentes com a bola, só podem enviar um comando do tipo movimento ou interação por ciclo.

Esta restrição se faz necessária, pois por exemplo: quando um jogador chuta a

bola não pode estar se movimentando ao mesmo tempo. Caso sejam enviados comandos que violem as restrições, o servidor escolhe um dos jogadores aleatoriamente, devido a informação sensorial disponível aos jogadores ser muito limitada, para simular os jogadores do futebol real usa cone de visão de 90° e comunicação limitada a 10 *bytes* por ciclo de simulação

Para serem bem sucedidos nas suas ações é conveniente que os agentes de uma equipe, efetuem previsões de quais as ações que os adversários (e companheiros de equipe) irão realizar, desta forma é possível, por exemplo, chutar ao gol, de costas para este, sem ver o goleiro, prevendo onde ele estará numa determinada posição, ou passar a bola a um colega de equipe, sem o ver, prevendo qual será a sua posição no campo na situação atual. O *SoccerServer* é o mesmo para todas as equipes e uma vez configurado passa-se a fase de implementação dos algoritmos para passar a bola, chute, estratégia, etc. Essa programação concentra-se em receber as informações do servidor, tomar as decisões e enviá-las de volta ao mesmo para que sejam executadas

Algumas ações executadas pelo agente são específicas do futebol real e são definidas como a seguir:

- **Tentar desarmar o adversário:** para esta ação o jogador deve se aproximar da bola em posse de um adversário o suficiente para chutá-la, e então chutar a bola para uma posição mais favorável, onde exista um companheiro próximo.
- **Driblar:** neste caso o jogador define a direção do adversário marcando e chuta a bola na diagonal deste adversário, podendo então correr novamente em direção a bola, ficando em uma posição mais favorável e, se possível, livre da marcação do adversário.
- **Posicionar-se para atacar:** esta ação é executada quando a bola esta em posse de um companheiro ou um companheiro é o jogador mais provável para alcançar a bola, o jogador neste caso deve se posicionar em um setor do campo determinado pela função definida para este jogador.
- **Posicionar-se para defender:** analogamente a ação de posicionar-se para atacar, esta ação é executada quando um adversário esta em posse da bola ou é o jogador mais provável para alcançá-la. O jogador deve buscar uma posição no campo seguindo uma definição de posição defensiva de acordo com sua função no campo.

## 2.3 Times Base

O time-base é uma plataforma arquitetada para abstrair códigos de baixo-nível e auxiliar outras equipes, mas o seu uso é opcional, visto que cada equipe pode desenvolver sua

própria estrutura. Esta seção objetiva apresentar alguns dos times-base mais utilizados no cenário mundial e nacional apontando suas principais características, com um detalhamento aprofundado para o Uva Trilearn, o qual é utilizado neste trabalho.

### 2.3.1 WrightEagle Base

O time WrightEagle (LI; CHEN; CHEN, 2015) da China, disputa a RoboCup Simulation desde o ano 1999 e nos últimos 3 anos ganhou duas vezes o campeonato mundial, disponibiliza parte de seus códigos para a comunidade contribuindo para que possam se preocupar menos com a implementação de baixo-nível e focar na parte estratégica e inteligente para a disputa de uma partida. Este *framework* possui um comportamento básico munido com redes neurais artificiais para a tomada de decisão e também com muitas implementações baixo-nível. É bem modularizado sendo seu código separado por comportamento (*Behavior*) e por ações (*Action*). Sendo utilizada a linguagem de programação C++.

### 2.3.2 Helios Base (Agent2D)

O time Helios do Japão, disputa a RoboCup Simulation desde o ano 2000, disponibiliza parte de seus códigos (chamado de plataforma de desenvolvimento Agent2D) sob licença GNU GPL com a intenção de contribuir com outras equipes e com a sociedade acadêmica.

O Agent2D foi desenvolvido em C++ por Hidehisa Akiyama (AKIYAMA; SHIMORA; NODA, 2008). Esse time base é atualizado com certa frequência e possui mais ampla cobertura com relação às habilidades básicas se comparado ao UvaTrilearn, o que permite ao desenvolvedor centrar seu trabalho na construção de habilidades mais complexas. Ao contrário do *framework* Uva Trilearn, a documentação do Agent2D é escassa e se restringe a algumas poucas em japonês (incluindo um manual) e inglês, o que dificulta o entendimento da estrutura e de seu funcionamento.

Os integrantes do time são responsáveis também pelo RoboCup Soccer Simulator (RCSS), o que facilita a comunicação do time com o servidor. O framework Agent2D já tem um comportamento básico para os agentes. Roda sob a biblioteca *librcsc2* e seu código é separado por comportamento (*Behavior*) e ações (*Action*). A modelagem deste time é feita seguindo a estrutura conhecida por *Action Chain*, que significa cadeia de ações. Esta cadeia funciona de forma similar a um grafo, onde cada nó é constituído por um par Ação/Estado. A ação é algo em que o agente pode realizar como um passe, um chute, um drible. Já o estado é a previsão do modelo do mundo após a ação ser consumada.

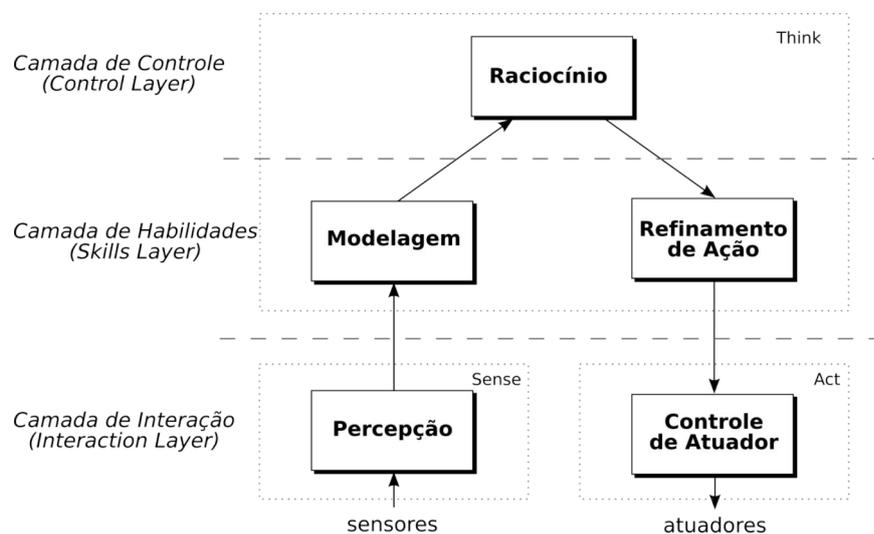
A modelagem *Action Chain* objetiva uma sequência de previsões que possam consolidar a obtenção de um resultado alvo, que se dá por meio de ações individualistas

e/ou cooperativas. Por exemplo, se o agente prevê que uma sequência de 3 jogadas irá levá-lo ao gol, ele começa a executar estas jogadas.

### 2.3.3 UvA TriLearn Base

O desenvolvimento do UvA TriLearn (BOER; KOK, 2002) baseou-se na divisão arquitetural em três camadas, baseando-se que tarefas complexas podem sempre decompor-se hierarquicamente em várias sub-tarefas simples. Esta arquitetura é mostrada na Figura 8 de forma hierárquica no sentido da mesma conter três camadas em diferentes níveis de abstração e implementadas na linguagem C++.

Figura 8 – Arquitetura do time UvA TriLearn



Fonte: (BOER; KOK, 2002)

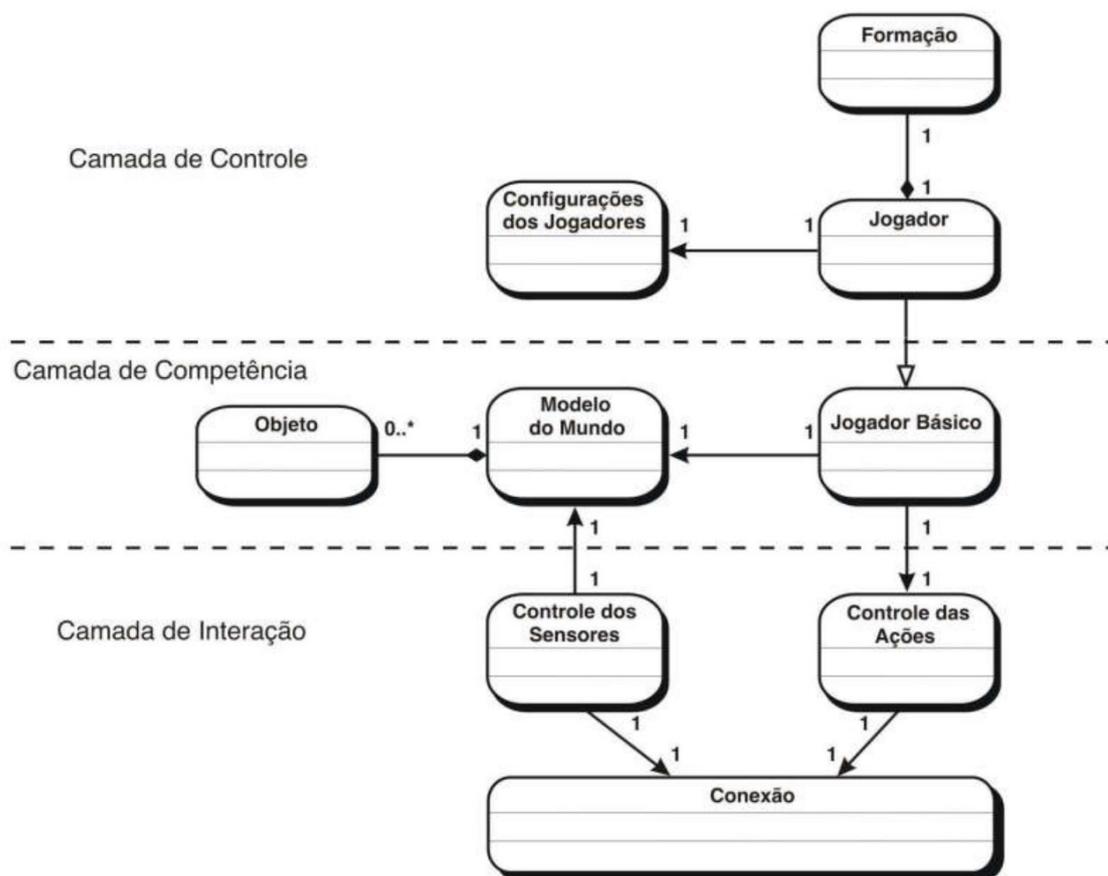
A camada mais baixa (Camada de Interação) é a parte responsável pelas interações com o ambiente de simulação do servidor, com a entrada no sistema através de sensores e a saída (ação) nos atuadores (agentes). Ela é capaz de abstrair o máximo de detalhes do servidor possível para as outras camadas superiores. A camada intermediária (Camada de Habilidades) usa a funcionalidade oferecida pela camada de interação a fim de construir um modelo abstrato do mundo e implementar várias habilidades para cada agente.

A camada do topo (Camada de Controle) contém os componentes de raciocínio do sistema, na qual é feita a escolha da melhor estratégia para o momento corrente. O time-base UvA TriLearn foi fornecido por pesquisadores da Universidade de Amsterdã - Holanda que após a conquista mundial da Robocup 2003, disponibilizou parte de seus códigos, retirando apenas a camada de controle.

No diagrama de classes da Figura 9, são apresentadas as relações de associação, composição e generalização entre as três camadas da arquitetura.

Cada classe é melhor detalhada abaixo:

Figura 9 – Diagrama UML da arquitetura do UVA Trilearn



Fonte: (BOER; KOK, 2002)

- **Conexão:** componente responsável por criar uma conexão *socket UDP* com o servidor, contendo métodos para enviar e receber mensagens por meio dessa conexão;
- **Controle dos sensores:** esse componente processa as mensagens recebidas do servidor, particiona e envia as informações extraídas para o componente Modelo do Mundo;
- **Modelo do Mundo:** responsável por criar a representação atual do jogo, do ponto de vista observado pelo agente, determinando-se a posição do agente, da bola e de suas respectivas velocidades;
- **Objeto:** esse componente contém informações sobre todos os objetos na simulação, como demarcações do campo, jogadores, bola e posição do gol;
- **Formações:** esse componente contém informações sobre as possíveis formações do time e do método para determinar a posição estratégica;
- **Configuração do Jogador:** esse componente contém os valores de muitos parâmetros do jogador, tal como a influência para resolver algum processo do agente e

de definir métodos para receber e atualizar esses valores.

- **Jogador:** contém métodos para resolver sobre a possibilidade da melhor ação em uma determinada situação;
- **Jogador Básico:** esse componente contém todas as informações necessárias para um perfil de jogador, como interceptar a bola ou chutar a bola para uma determinada posição do campo;
- **Controle da ação:** responsável por executar os comandos que o jogador deve realizar, assim que foi recebido do componente jogador Básico.

O time UvA Trilearn apresenta como maior contribuição uma arquitetura de três camadas capaz de realizar o processamento paralelo dos dados, um esquema flexível de sincronização do ambiente, métodos eficazes para localização de objetos e estimação de velocidades usando filtros de partículas e uma camada hierárquica de competências. O time UvA Trilearn possui 3 formações táticas,  $(3-4-3)$ ,  $(4-4-2)$  e  $(4-3-3)$ . O UvA Trilearn utiliza a formação tática  $(4-3-3)$ , que corresponde em separar os jogadores ao longo do campo em áreas de atuação. No caso, os quatro primeiros jogadores atuarão como zagueiros, os três outros jogadores atuarão como meio-campo e os últimos três jogadores atuarão como atacantes.

### 2.3.4 BAHIA2D

A arquitetura adotada pelo BAHIA 2D é dividida em três camadas inter-relacionado, o Nível Reativo é o que executa as ações. Ele possui os métodos importados do UVA Trilearn e alguns do próprio Bahia2D desenvolvidos em linguagem C++. Esses métodos são fundamentais, pois eles recebem os dados brutos do simulador, tratando-os e transformando-os em informação. O desenvolvedor tem essa informação e passa um comando para o simulador através deste dado já tratado, não sendo mais necessário manipular diretamente coordenadas.

Um time com apenas essa primeira camada não possui uma inteligência propriamente dita, pois ele apenas reage às informações, ele não toma decisões ainda. Para isso foi desenvolvido uma segunda camada, o Nível Instintivo. Neste nível, o time passa a analisar as informações recebidas e tomar uma decisão com base nestes resultados. Assim, ele age conforme essas decisões e não indiscriminadamente como fazia antes, adquirindo um grau de inteligência mais significativo.

Como uma terceira camada, ainda em fase de implementação no Bahia 2D, temos o Nível Cognitivo. Este é o mais avançado, pois é ele que traça os planos e metas do time com base nas informações da partida. Nesse contexto, pode-se fazer uma comparação das camadas da arquitetura do time com os níveis de classificação de um Sistema de

Informação. O Nível Reativo seria um sistema do Nível Operacional, que cuida das tarefas do dia-a-dia, repetitivas; o Nível Instintivo seria do Nível Tático, que toma decisões para o grupo a curto prazo, com repercussões imediatas; e o Nível Cognitivo seria do Nível Estratégico, que, como o nome sugere, traça a estratégia do grupo, planos e metas a longo prazo.

## 3 Estratégias e jogadas para o UvA TriLearn Base

Como o simulador das partidas *SoccerServer* <sup>1</sup> é o mesmo para todas as equipes e uma vez configurado, passa-se a fase de implementação dos algoritmos para a realização de ações como: passe da bola, chute ao gol, interceptação da bola, etc. Essa programação concentra-se em receber as informações do servidor, tomar as decisões e enviá-las para o servidor no intuito de serem executadas. Para isto, são realizados diversos testes e análises através das plataformas oferecidas pela RoboCup.

Alguns algoritmos podem envolver estratégias simples, tais como correr e chutar, assim como desenvolver estratégias de jogo mais complexas e elaboradas que podem unir diferentes áreas da computação, por exemplo: controle de processos, inteligência artificial, otimização, processamento de sinal, sistemas multiagentes etc.

Exemplos de alguns desses algoritmos podem ser vistos na: equipe ITANDROIDS (Instituto Tecnológico de Aeronáutica - Brasil) aplicou teoria de probabilidade para modelos de predições de posicionamento e velocidade dos jogadores e da bola (XAVIER; BARBOSA; MATSUURA, 2006); equipe WrightEagle (*University of Science and Technology of China* - China) implementou um sistema de inferência fuzzy para prever os comportamentos dos oponentes (LI; CHEN; CHEN, 2015); equipe HFutEngine (*Hefei University of Technology* - China) usou uma rede neural para regular a orientação do corpo dos jogadores (ZHIWEI-LE; WANG; HAO-WANG, 2015); equipe Ri-one (*Ritsumeikan University* - Japão) aplicou um algoritmo genético para analisar uma função que decide o movimento a ser realizado quando o jogador tem a bola.

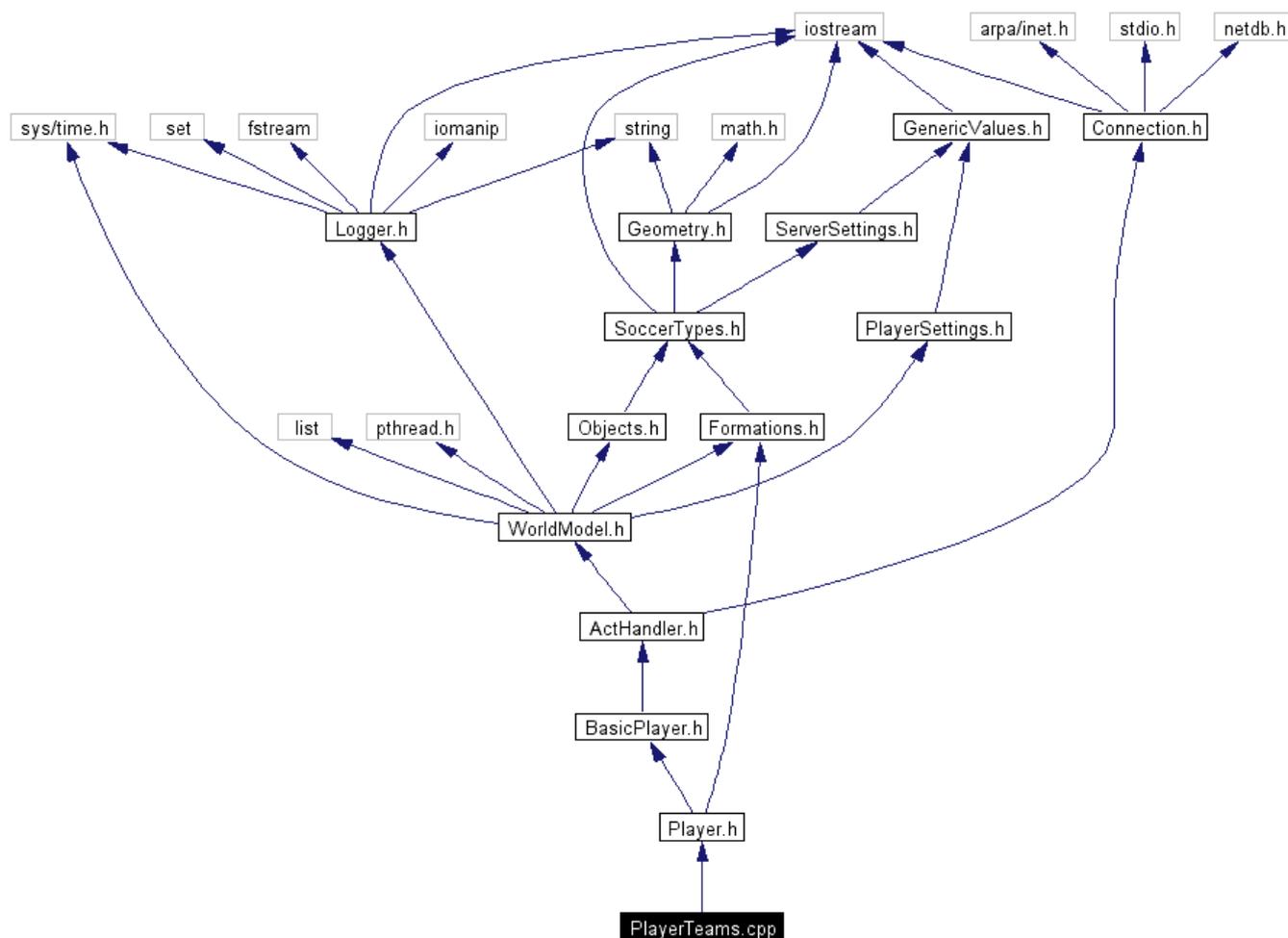
Escolheu-se como time base, o UVA Trilearn <sup>2</sup>, disponibilizado pela Universidade de Amsterdã - Holanda, para testes e implementação de algoritmos básicos. O algoritmo principal desse time base utiliza agentes que foram programados para ficar em uma determinada formação e o que estivesse mais próximo da bola interceptaria a mesma e chutaria em direção ao gol adversário. Será utilizado a classe *PlayerTeams* que herda métodos e objetos das classes *Player* e *BasicPlayer* para implementação de algoritmos, como ilustrado na Figura 10.

O time base possui várias funções já prontas, por exemplo: carregar a bola, marcação, passe, esconder a bola, parar a bola, etc. Essas funções, e muitas outras

<sup>1</sup> A liga de simulação do RoboCup é baseada no simulador conhecido como SoccerServer (NODA et al., 1998) que simula um jogo virtual de futebol disputado entre duas equipes compostas por 11 agentes autônomos cada.

<sup>2</sup> Todas as configurações iniciais para utilização da plataforma RCSS e UVA Trilearn podem ser seguidas através do tutorial em anexo no Apêndice-A

Figura 10 – Dependências da Classe PlayerTeams do UvA Trilearn 2003



Fonte: Base Code Documentation 2003

encontram-se na classe *BasicPlayer*. As duas classes responsáveis pela atuação do jogador são *WorldModel* e *Player*. O *WorldModel* é responsável por manter um modelo do mundo (campo, bola, jogadores) e usá-lo para interação com a classe *Player*, sendo essa responsável por tomar as decisões e enviar os comandos para o simulador.

As estratégias propostas neste trabalho são apresentadas nas sub-seções a seguir. Foram realizadas implementações iniciais, definidas a partir de 3 estratégias de jogadas: Chutar ao gol, Chutar de acordo com a coordenada do jogador e Drible, com base em modificações de estratégias propostas no manual para times base (PET-UFES, 2016).

### 3.1 Estratégia I: Chutar ao gol

Esta estratégia visa chutar em direção ao gol de qualquer posição do campo. A estratégia do time base é chutar para o gol sempre que a bola estiver na área de chute do jogador. A bola está na área de chute quando a distância dela até o jogador é menor ou igual ao `kickable_margin`, que é em torno de 0.7 metros, dependendo do jogador.

Opta-se em chutar em um dos cantos do gol adversário, de acordo com o ciclo corrente da partida. O trecho de código a seguir ilustra a implementação dessa estratégia.

```

1 /* UFMA2D Estrategia I: Chutar ao gol @JG */
2 int sideGoal = (-1 + 2*(WM->getCurrentCycle()%2));
3
4 if( WM->isBallKickable())// if kickable
5 {
6     VecPosition posGoal ( PITCH_LENGTH/2.0, sideGoal * 0.4 * SS->
7     getGoalWidth() );
8     soc = kickTo( posGoal, SS->getBallSpeedMax() ); // kick maximal
9
10 ACT->putCommandInQueue( soc );
11 ACT->putCommandInQueue( turnNeckToObject( OBJECT_BALL, soc ) );
12 }

```

A primeira decisão do agente é verificar se a bola está na sua área de chute. Isso é feito através da função `WM->isBallKickable()`, que calcula a distância do jogador até a bola e verifica se esta é menor que a `kickable_margin` do jogador.

Logo em seguida é definido o objeto chamado `posGoal`. Ele é um objeto do tipo `VecPosition`, que contém as coordenadas `x` e `y` de determinado ponto no campo. A origem deste sistema de coordenadas é o centro do campo, sendo que a sua orientação é relativa ao campo de ataque do time. Se o time estiver atacando para a direita, então a coordenada `x` positiva estará apontada para a direita e a coordenada `y` positiva estará apontada para baixo. Por exemplo, o ponto `x = 10` e `y = 0`, sempre estará no lado do campo adversário. Uma forma de inicializar uma variável do tipo `VecPosition` segue o formato:

```

1 VecPosition variavel (x, y);

```

A Figura 11 ilustra o sistema de Coordenadas de um time que está na esquerda e atacando para a direita.

No objeto `posGoal`, a coordenada `x` é definida como: `PITCH_LENGTH/2.0`. O `PITCH_LENGTH` é o tamanho do campo em comprimento, cerca de 102,5 metros. Como a origem do sistema de coordenadas é o centro do campo, o `PITCHLENGTH / 2.0` simboliza a extremidade direita do campo, ou seja, a linha do gol adversário. Já a coordenada `y` do `posGoal` é definida como:

```

1 VecPosition posGoal (PITCH_LENGTH/2.0, (sideGoal) *0.4 * SS ->getGoalWidth
2 ( ) );

```

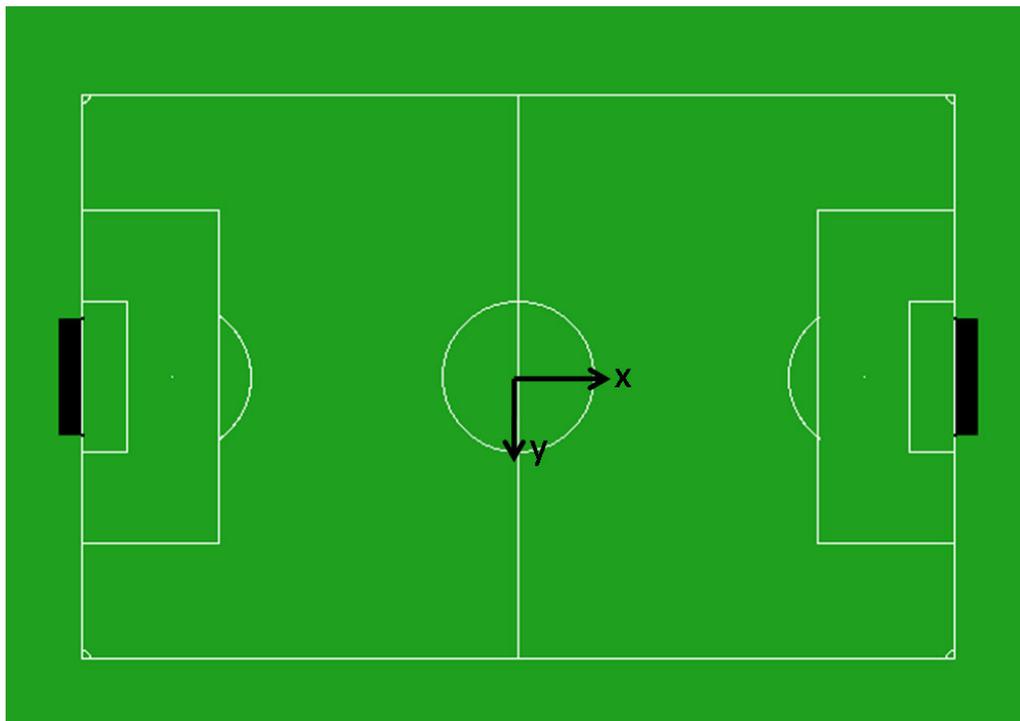
A variável `sideGoal` terá como resultado 1 ou -1, calculado da seguinte maneira por ciclo:

```

1 int sideGoal =( -1 + 2*(WM->getCurrentCycle()%2)

```

Figura 11 – Eixos X e Y do campo.



Onde a função `WM->getCurrentCycle()` retorna o ciclo da partida, sendo que no primeiro tempo este ciclo varia de 1 a 3000 enquanto que no segundo tempo este valor varia de 3001 a 6000. Portanto, o trecho de código `WM->getCurrentCycle()%2` será o resto da divisão desse valor por 2 que retornará 0 se o ciclo for par e 1 caso esse seja ímpar. Multiplicando esse valor por 2 e somando a -1, obteremos 1 ou -1, caso o ciclo seja ímpar ou par, respectivamente.

O valor da coordenada do gol será determinada positiva ou negativa dependendo do ciclo da partida (conforme explicitado anteriormente). Quanto ao trecho de código a seguir tem-se:

```
1 0.4 * SS->getGoalWidth()
```

a função `SS->getGoalWidth()` retorna o tamanho do gol (14,02m). Multiplicando esse valor por 0,4 obteremos a coordenada `y` de valor 5,608 da trave positiva do gol.

Assim, o trecho seguinte determinará o lado do gol no eixo `Y` a ser chutado em cada ciclo:

```
1 VecPosition posGoal (PITCH_LENGTH/2.0, (sideGoal) *0.4 * SS->getGoalWidth()
);
```

Nota-se que se o ciclo da partida for ímpar, o resultado desse trecho de código será:  $-1 * 5,608 = -5,608$ , que simboliza a coordenada `y` da trave de cima. Resumindo, a coordenada armazenada pela variável `posGoal` será um ponto próximo da trave (5,608 ou -5,608) de cima nos ciclos pares e um ponto próximo da trave de baixo nos ciclos ímpares.

Uma alternativa de determinar as coordenadas possíveis para a posição do gol seria:

```
1 posGoal ( PITCHLENGTH/2.0 , 0.5*SS->getGoalWidth() ) ,
```

obtendo-se os seguintes valores de coordenadas  $x = 51.25$  e  $y = 7.01$ . Essa coordenada é exatamente a coordenada da trave positiva do gol adversário, isso implica em chutar em direção ao mesmo canto do gol sempre.

No entanto, uma vez que o simulador insere ruídos nas informações de posições que o jogador recebe, assim como na força e direção do seu chute, logo, ao se multiplicar o tamanho do gol por 0,5, o jogador pode chutar para fora do gol. Por isso, multiplica-se o tamanho do gol por 0,4 , obtendo assim uma margem de segurança.

Na próxima linha, tem-se a instrução:

```
1
2 soc = kickTo( posGoal , SS->getBallSpeedMax() );
```

Nela, a variável *soc* recebe o retorno da função *kickto*, ela recebe como parâmetro o ponto do campo para o qual se deseja chutar a bola e a velocidade que a bola deve ter quando chegar a esse ponto. Com isso, a função calcula qual é a força e ângulo necessário para fazer o chute, sendo o *posGoal* um ponto do gol adversário. A linha seguinte, *SS->getBallSpeedMax()*, retorna a velocidade máxima da bola. Portanto, ao final o jogador chutará para o gol com força máxima.

Para o envio de comandos ao servidor, utiliza-se a função:

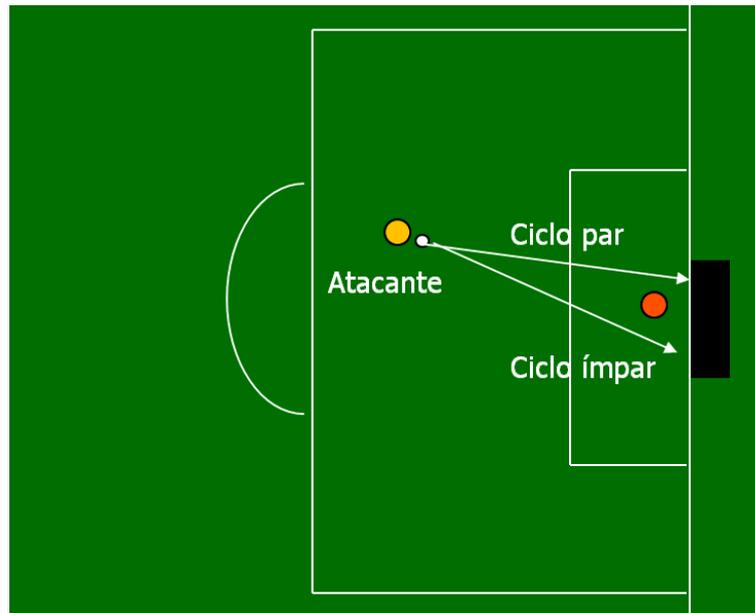
```
1
2 ACT -> putCommandInQueue( soc );
3 ACT -> putCommandInQueue( turnNeckToObject( OBJECT_BALL, soc ) );
```

que insere o comando *soc* na fila de comandos que serão enviados para o servidor. A variável *soc* contém o comando de chute em direção ao gol. Nota-se ainda que logo após é colocado na fila de comandos a função:

```
1
2 turnNeckToObject ( OBJECT_BALL, soc );
```

que retorna o comando de girar o pescoço com um ângulo que fará com que a cabeça do jogador fique direcionada para a bola. Os comandos de chute, virar o corpo e correr são mutuamente exclusivos, ou seja, não é possível executar dois deles no mesmo ciclo. A Figura 12 ilustra as ações possíveis com a Estratégia I.

Figura 12 – Aplicação da Estratégia I.



### 3.2 Estratégia II: Chutar de acordo com a coordenada do jogador

Assim como na estratégia I o time continuará a chutar ao gol mas com um refinamento dessa ação, pois na estratégia II, faz-se necessário saber a posição do jogador antes de tocá-la ou chutá-la. Assim o jogador verifica se a bola está na sua área de chute e testa sua posição no campo (ataque ou defesa) através do objeto *posAgent*, como pode ser visualizado no trecho de código a seguir:

```

1  /* UFMA2D Estrategia II: Chutar de acordo com a coordenada do jogador @JG
   */
2  VecPosition posAgent = WM->getAgentGlobalPosition();
3  int sideGoal = (-1 + 2*(WM->getCurrentCycle()%2));
4  int posXField = 40;
5  if( WM->isBallKickable()) // if kickable
6  {
7      if (posAgent.getX() < 0)/* side deffense */
8      {
9          soc =kickTo(VecPosition(posAgent.getX()+10,posAgent.getY()), 1);
10     }
11     else if (posAgent.getX() < 35)
12     {
13         soc = kickTo ( VecPosition (posXField,-15), 0.3);
14     }
15     else
16     {
17         VecPosition posGoal( PITCH_LENGTH/2.0, sideGoal * 0.4*SS->
getGoalWidth() ); //kick maximal
18         soc = kickTo( posGoal, SS->getBallSpeedMax() );
19     }

```

```

20     }
21     ACT ->putCommandInQueue( soc );
22     ACT ->putCommandInQueue( turnNeckToObject( OBJECT_BALL, soc ) );
23 }

```

No trecho:

```

1 VecPosition posAgent = WM->getAgentGlobalPosition();

```

o objeto *posAgent* guardará a posição do jogador. As coordenadas x e y do jogador são obtidas através das funções *getX()* e *getY()* do *VecPosition*, respectivamente.

Adiante na função:

```

1 kickTo( VecPosition( posAgent.getX() + 10, posAgent.getY() ), 1 );

```

é passado como parâmetro um ponto com a mesma coordenada y onde encontra-se o jogador e a coordenada x do jogador acrescida de 10, fazendo com que o jogador chute a bola mais a frente com apenas 1 m/ciclo, uma vez que o segundo parâmetro da função *kickTo()* refere-se a velocidade do chute.

E no trecho restante:

```

1 else if ( posAgent.getX() < 35 ) {
2     soc = kickTo ( VecPosition ( posXField, -15 ), 0.3 );
3     } else {
4     VecPosition posGoal( PITCH_LENGTH/2.0, sideGoal * 0.4 * SS->getGoalWidth
5     ( ) ); //kick maximal
6     soc = kickTo( posGoal, SS->getBallSpeedMax() );

```

Se o jogador estiver no campo de ataque, na entrada da grande área e  $x < 35$ , então, este fará um lançamento para a coordenada (posXField, -15), sendo que posXField = 40 e este valor fora selecionado por corresponder a entrada da pequena área acima. Uma vez que a coordenada y = -15 é negativa o jogador realizará um passe para a posição superior do campo com velocidade de chute de 3m/ciclo, isto é parte de cima, do campo de ataque objetivando que o jogador mais próximo desse ponto receba a bola e chute ao gol.

Assim como na estratégia I tem-se as instruções:

```

1 ACT ->putCommandInQueue( soc );
2 ACT ->putCommandInQueue( turnNeckToObject( OBJECT_BALL, soc ) );

```

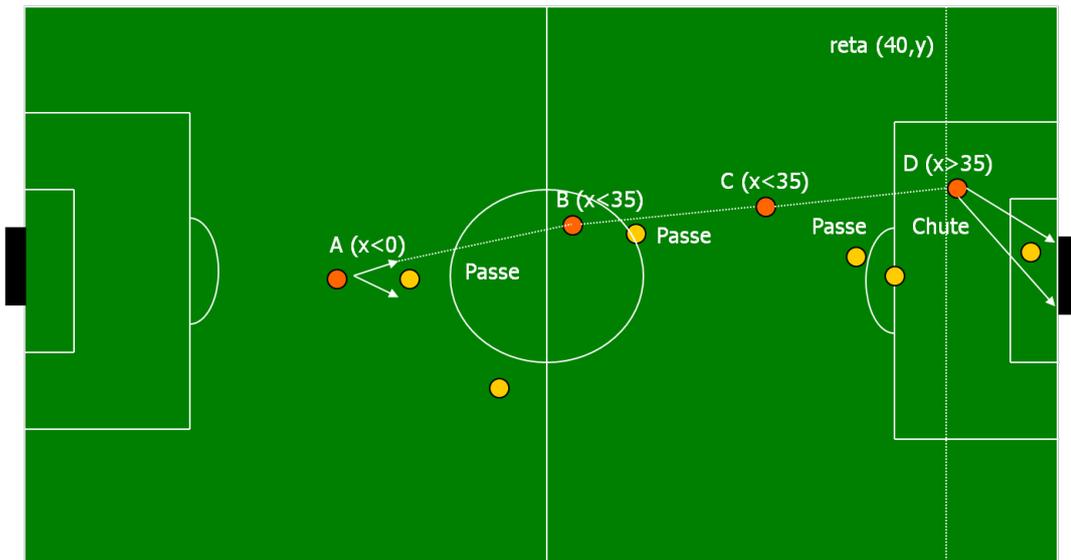
que realizam a mesma função de envio de comandos ao servidor.

A Figura 13 ilustra as ações possíveis com a Estratégia II.

### 3.2.1 Estratégia III: Drible

Na estratégia III um dos jogadores deve carregar a bola driblando até a entrada da grande área, região onde a coordenada x é maior que 35. E chutar em seguida em direção

Figura 13 – Aplicação da Estratégia II.



ao gol.

Ao exemplificar o uso da função *dribble()*, que é uma função usada para carregar a bola, percebe-se que a mesma faz o jogador chutar a bola com menos força para um ângulo previamente determinado. Assim, o jogador estará na verdade carregando a bola. O seu exemplo de uso pode ser encontrado no código abaixo:

```

1 /* UFMA2D Estrategia III: Drible @JG */
2 int sideGoal =(-1 + 2 * (WM->getCurrentCycle()%2));
3 if( WM ->isBallKickable() // if kickable
4 {
5     if (posAgent.getX() < 35) //middle field of attack
6     {
7         soc = dribble (0,DRIBBLE_SLOW);
8     }
9     else
10    {
11        VecPosition posGoal(PITCH_LENGTH/2.0,sideGoal*0.4*SS->getGoalWidth());
12        soc = kickTo( posGoal, SS->getBallSpeedMax() ); // kick maximal
13    }
14 ACT ->putCommandInQueue(soc);
15 ACT ->putCommandInQueue(turnNeckToObject(OBJECT_BALL,soc));
16 }

```

No trecho de código:

```

1 if (posAgent.getX() < 35){ //middle field of attack
2     soc = dribble (0,DRIBBLE_SLOW);
3 }

```

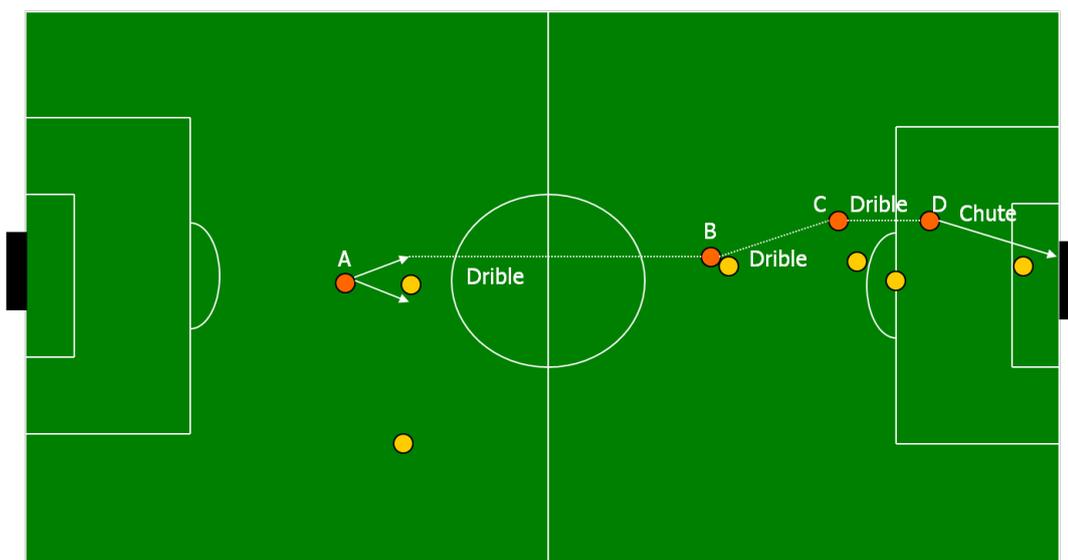
se o jogador estiver antes da posição  $x = 35$ , irá carregar a bola em um ângulo =

0°, ou seja, para frente e driblando de forma a somente conduzir a bola. Quando passar da posição  $x = 35$  chutará a bola para o gol assim como na Estratégia I.

Na definição da função *dribble()* do time base do UVA TriLearn existem outros dois tipos de drible que são passados no segundo parâmetro da função *dribble(0,\_\_\_)*: DRIBBLE\_WITHBALL e DRIBBLE\_FAST. Eles diferenciam-se apenas na força do chute.

A Figura 14 ilustra as possíveis ações com a Estratégia III.

Figura 14 – Aplicação da Estratégia III.



## 4 Resultados - Estratégias propostas.

Nesta seção são apresentados resultados de cada uma das 3 estratégias aplicadas gradualmente ao time UVA Trilearn, em 3 partidas realizadas contra o time do Agente2D, visando analisar as quantidades de gols sofridos, marcados, finalizações e defesas realizados pelo time, considerando os pontos positivos e negativos obtidos ao acréscimo de cada estratégia. Adicionalmente, foram realizadas partidas entre o próprio UVA Trilearn com versões de estratégias diferentes, ou seja, confrontando-se as estratégias II e III. As partidas duraram 2 tempos de 5 minutos cada, totalizando 10 minutos por partida.

### 4.1 Estratégia I: Chutar ao Gol

A Tabela 4 demonstra resultados obtidos com a estratégia I de chutar em direção ao gol com potência máxima (*kick maximal*), como abordado na seção anterior.

Tabela 4 – Resultados da Estratégia I da partida UVA I x Agent2D.

UVA I x Agent2D	Gols Marcados	Gols Sofridos	Finalizações	Defesas
1º Tempo	0	9	0	0
2º Tempo	0	14	0	0
Total	0	23	0	0

Durante o 1º tempo foram marcados 0 gols e sofridos 9. No 2º tempo com a diminuição de stamina por parte de alguns jogadores do UVA Trilearn, devido ao chute com potência máxima, notou-se uma dificuldade para esse jogadores cansados interceptarem a bola, visando bloquear os adversários, assim mais 14 gols foram sofridos pelo time proposto, totalizando 23 gols, sendo que nenhum gol foi marcado utilizando-se a estratégia I, evidenciando-se a necessidade do desenvolvimento de estratégias mais refinadas para um placar mais equilibrado.

### 4.2 Estratégia II: Chutar de Acordo com a Coordenada do Jogador

A Tabela 5 demonstra os resultados obtidos com a estratégia II, que além chutar ao gol utiliza a coordenada do jogador que vai realizar o passe ou o chute. Durante o 1º tempo foram marcados 0 gols e sofridos 12 pelo time proposto, apresentando um aumento de gols sofridos no 1º ciclo – quando comparado a estratégia I – o que pode ser explicado devido a maior ofensividade obtida com a estratégia II. No 2º tempo sofreu-se 11 gols totalizando 23 gols, sendo marcados 0 gols nos 2 tempos com a estratégia II, mostrando-se apenas uma melhora no 2º tempo em relação a Tabela 4 em que a estratégia I é aplicada.

Tabela 5 – Resultados da Estratégia II da partida UVA II x Agent2D.

<b>UVA II x Agent2D</b>	<b>Gols Marcados</b>	<b>Gols Sofridos</b>	<b>Finalizações</b>	<b>Defesas</b>
1º Tempo	0	12	0	0
2º Tempo	0	11	0	0
Total	0	23	0	0

### 4.3 Estratégia III: Drible

A Tabela 6 ilustra os resultados obtidos com a estratégia III. Na estratégia III aplica-se a tentativa de drible, onde o jogador visa desviar do jogador adversário.

Tabela 6 – Resultados da Estratégia III da partida UVA III x Agent2D.

<b>UVA III x Agent2D</b>	<b>Gols Marcados</b>	<b>Gols Sofridos</b>	<b>Finalizações</b>	<b>Defesas</b>
1º Tempo	0	13	1	0
2º Tempo	0	9	0	1
Total	0	22	1	1

A estratégia III aplicada na partida diminuiu a quantidade de gols sofridos no segundo tempo, assim como o placar final - quando comparada com as estratégias I e II - passando de 23 para 22 gols sofridos, assim como passou a evidenciar finalizações e defesas.

### 4.4 Estratégia II e III Aplicadas em uma partida do UVA II x UVA III

Com as três estratégias propostas objetivou-se dá maior ofensividade, entretanto, percebeu-se que as partidas contra times de elevada capacidade técnica - nomeadamente Agent2D – não oferecem métricas satisfatórias para efeitos comparativos. Portanto, optou-se em realizar novas partidas utilizando-se somente o time proposto, ou seja, um time com a estratégia II e o outro time com a estratégia III, sendo seus resultados apresentados na Tabela 7:

Tabela 7 – Gol sofridos e marcados pelo UVA II na partida UVA II X UVA III

<b>UVA IIxUVA III</b>	<b>Gols Marcados</b>	<b>Gols Sofridos</b>	<b>Finalizações</b>	<b>Defesas</b>
1º Tempo	2	0	17	9
2º Tempo	1	1	10	14
Total	3	1	27	23

A Tabela 7 apresenta os resultados da partida do UVA Trilearn com as estratégias II e III confrontadas em uma partida de 2 tempos. Fez-se a opção em testar somente as

estratégias II e III já que ambas usam a estratégia I. Os resultados demonstram a melhor capacidade ofensiva dos times neste cenário, tanto em alcançar-se o campo adversário como no número de defesas. Os resultados da Tabela 7 demonstram a vitória do time UVA II com a estratégia II por 3 x 1 em relação ao UVA III que utiliza a estratégia III. Sendo realizadas 17 finalizações e 9 defesas no 1º tempo e 10 finalizações e 14 defesas no 2º tempo, totalizando 27 finalizações e 23 defesas nos 2 tempos pelo time UVA II.

## 5 Conclusão

Este trabalho objetivou o desenvolvimento do primeiro time de futebol robótico simulado na UFMA, visando estimular a inserção do Maranhão e do Brasil nas competições de robótica internacionais. Apesar dos resultados obtidos contra o Agent2D, um time notadamente superior, os objetivos do trabalho foram alcançados, uma vez que o objetivo principal era a criação de um time inicial na UFMA.

Nas partidas comparando as estratégias II e III, houve uma maior compreensão e aproveitamento das ações propostas em cada uma das estratégias, pois foram avaliados times de mesmo nível de habilidade. A fim de aprofundar os testes das estratégias, visa-se futuramente realizar-se mais testes da estratégia II e III contra si mesmas.

Ressalta-se o potencial do time proposto, assim como suas lacunas para o uso de novas estratégias. O goleiro do time sugerido neste trabalho precisa ser aperfeiçoado por ainda possui sua codificação original, isto é nenhuma estratégia específica foi implementada para esse jogador.

Ainda como trabalhos futuros, visa-se o estudo mais aprofundado dos códigos fonte dos principais times base, implementação dos principais algoritmos usados na Liga *RoboCup Soccer - Simulation League 2D*, criação de estratégias mais avançadas, por exemplo introduzindo mais conceitos de sistemas multiagentes, processamento de sinal, otimização, entre outros.

## Referências

- AKIYAMA, H.; SHIMORA, H.; NODA, I. Helios2008 team description. In: *12th RoboCup International Symposium. (July 2008)(CD Supplement)*. Japan: Information Technology Research Institute, 2008. Citado na página 25.
- ALDEBRAN. *Aldebran Robotics Corporate Site, 2016*. (acessado em 11 de Setembro de 2016). 2016. Disponível em: <[http://doc.aldebaran.com/2-1/home\\_ao.html](http://doc.aldebaran.com/2-1/home_ao.html)>. Citado na página 17.
- BOER, R. de; KOK, J. *The incremental development of a synthetic multi-agent system: The uva trilearn 2001 robotic soccer simulation team*. Tese (Doutorado) — Master's thesis, University of Amsterdam, The Netherlands, 2002. Citado 2 vezes nas páginas 26 e 27.
- BUDDEN, D. M. et al. Robocup simulation leagues: Enabling replicable and robust investigation of complex robotic systems. *IEEE Robotics & Automation Magazine*, IEEE, v. 22, n. 3, p. 140–146, 2015. Citado na página 15.
- FRACCAROLI, S. E. *Análise de desempenho de algoritmos evolutivos no domínio do futebol de robôs*. *Biblioteca Digital de Teses e Dissertações da USP, 2011*. Tese (Doutorado) — Universidade de São Paulo, 2011. Citado na página 15.
- KITANO, H. et al. Robocup: The robot world cup initiative. In: *ACM. Proceedings of the first international conference on Autonomous agents*. California, USA, 1997. p. 340–347. Citado na página 13.
- LI, X.; CHEN, R.; CHEN, X. Wrighteagle 2d soccer simulation team description 2015. 2015. Citado 2 vezes nas páginas 25 e 30.
- PET-UFES. *Programa de Educação Tutorial de Engenharia de Computação - Universidade Federal do Espírito Santo (acessado em: 21 de Setembro de 2016)*. 2016. Disponível em: <<http://www.inf.ufes.br/~pet%3E>>. Citado na página 31.
- REIS, L. Coordenação em sistemas multi-agente: Aplicações na gestão universitária e futebol robótico. 2003. Citado 5 vezes nas páginas 18, 20, 21, 22 e 23.
- ROBOCUPGALLERY. *Robocup Graz-Austria, (acessado em 11 de Outubro de 2016)*. Austria, 2009. Disponível em: <<http://www.robocup2009.org/>>. Citado 2 vezes nas páginas 16 e 17.
- SILVA, H.; SIMÕES, M.; ARAGÃO, H. Desenvolvimento de controladores fuzzy para robôs de futebol simulado do tipo atacante. In: *Workshop de Trabalhos de Iniciação Científica e Graduação Bahia-Alagoas-Sergipe*. Porto Alegre: SBC, 2007. Citado na página 13.
- SILVA, P. H. G. Ajustes no time ufla2d no domínio de futebol de robôs. 2015. Citado na página 16.
- SIMULATION, R. S. *RoboCup 3D Soccer Simulation League*. (acessado em: 02 de Agosto de 2016). 2012. Disponível em: <<https://wiki.robocup.org/wiki/SoccerSimulationLeague>>. Citado na página 19.

---

TEIXEIRA, C. P. Mecateam 2011-abordagem multiagente em um sistema de controle distribuído para sistemas multirobos. 2011. Citado na página [13](#).

XAVIER, R. O.; BARBOSA, R.; MATSUURA, J. P. O time de futebol simulado itandroids-2d. 2006. Citado na página [30](#).

ZHIWEI-LE, K.-L.; WANG, G.; HAO-WANG, B.-F. Hfutengine2015 simulation 2d team description paper. 2015. Citado na página [30](#).

# Apêndices

# APÊNDICE A – Instalação e configuração de arquivos

Este manual foi baseado no tutorial disponível em <<https://github.com/herodrigues/robocup2d/wiki/Instalando-o-simulador>>. Desenvolvido em conjunto com o aluno Rodrigo Garcês de Ciência da Computação (UFMA).

## Dependências

Este comando instala todas as dependências necessárias para o correto funcionamento do futebol robótico. Todos os passos devem ser seguidos na sequência especificada, não podendo ser omitida qualquer parte deste tutorial.

O comando apenas instala e configura as bibliotecas que ainda não estiverem instaladas no sistema. Caso a biblioteca já esteja instalada, o sistema acusa que a mesma está instalada e prossegue para o próximo comando.

```
1 $ sudo apt-get install -y g++ build-essential libboost-all-dev qt4-dev-  
   tools libaudio-dev libgtk-3-dev libxt-dev doxygen tsh
```

Adicionar o repositório para posterior instalação do simulador

```
1 $ sudo apt-add-repository -y ppa:gnurubuntu/rubuntu  
2 $ sudo apt-get update
```

## Instalação Rcssserver (servidor)

```
1 $ sudo apt-get install -y rcssserver
```

## Monitor

```
1 $ sudo apt-get install -y rcssmonitor
```

## Soccersim (servidor e monitor juntos)

```
1 $ sudo apt-get install -y rcsoccersim
```

## Librcsc

```
1 $ wget http://c3sl.dl.osdn.jp/rctools/51941/librcsc-4.1.0.tar.gz
2 $ tar -zxpf librcsc-4.1.0.tar.gz
3 $ cd librcsc-4.1.0
4 $ sudo ./configure
5 $ sudo make
6 $ sudo make install
```

## Agent2d

```
1 $ wget http://c3sl.dl.osdn.jp/rctools/55186/agent2d-3.1.1.tar.gz
2 $ tar -zxpf agent2d-3.1.1.tar.gz
3 $ cd agent2d-3.1.1
4 $ sudo ./configure
5 $ sudo make
```

## UVA Trilearn (atualizado e adaptado pelo PET de engenharia da computação da UFES)

```
1 $ cd ~
2 $ echo "Instalando o UVA Trilearn"
3 $ wget http://inf.ufes.br/~pet/projetos/Simulacao_2D/Sim2D/
   trilearn_base_sources-3.3-v13-by_PET.tar.gz
4 $ tar -zxpf trilearn_base_sources-3.3-v13-by_PET.tar.gz
5 $ cd trilearn_base_sources-3.3-v13-by_PET
6 $ sudo ./configure
7 $ sudo make
```

## Execução

### Correção do problema do separador decimal

Antes de iniciar com o simulador, corrija o problema do separador decimal. Existem dois métodos. O primeiro não exige nenhum conhecimento técnico, mas exige que seja executado a cada vez que iniciar a máquina e desejar utilizar o simulador. O segundo método exige um mínimo de conhecimento sobre linux, mas precisa ser executado uma única vez, para configuração. Depois não é necessário executar novamente, como ao reiniciar a máquina.

### Método 1

Execute no terminal o comando:

```
1 $ export
2 LC_NUMERIC="C"
```

Este comando garante que o simulador não terá problema em trabalhar com o separador decimal sendo a vírgula. Este comando deve ser executado a cada sessão do usuário (em que pretenda usar o simulador), caso contrário, o simulador irá executar, mas a simulação não irá rodar de maneira satisfatória. Você só precisa executar o comando no primeiro terminal.

## Método 2

Abra o arquivo que contém os comandos do terminal com o comando

```
1 $ sudo nano ~/.bashrc
```

Caso sua distribuição não possua o arquivo `.bashrc`, crie-o!

Adicione a seguinte linha ao final do arquivo

```
1 $ export LC_NUMERIC="C"
```

Salve o arquivo e reinicie a máquina

## Monitor/Servidor

```
1 $ rcsoccersim
```

Caso apareça o erro "could not read or create config directory '/home/<usuário>/rcsserver/'", feche e abra novamente o `rcsoccersim`. Caso o erro persista, rode o programa como `root`. Este erro pode acontecer somente ao abrir o `rcsoccersim` pela primeira vez. Agora, você tem dois times base, o `agent2`, da equipe `HELIOS`, e o `UVA Trilearn`, da equipe de mesmo nome. Ambos os times são implementações prontas das funções de baixo nível das respectivas equipes, e são distribuídos gratuitamente para que os iniciantes gastem o tempo fazendo o que realmente conta, a estratégia. Ou seja, todas (ou pelo menos a maioria) das funções básicas já vem implementadas, de novo que o iniciante precise apenas implementar funções específicas e a estratégia de sua equipe, reduzindo de maneira muito significativa a quantidade de tempo e esforço gastos na fase inicial de implementação de uma nova equipe.

## Agent2D

### Primeira Equipe

```
1 $ cd <pasta do agent2d>/src
2 $ ./start.sh
```

### Segunda Equipe

```
1 $ cd <pasta do agent2d>/src
2 $ ./start.sh -t <nome da equipe>
```

## UVA Trilearn

### Primeira Equipe

```
1 $ cd <pasta do UVA Trilearn>/src
2 $ ./start.sh
```

### Segunda Equipe

```
1 $ cd <pasta do UVA Trilearn>/src
2 $ ./start.sh localhost <nome da equipe>
```

Abra um terminal, copie e cole o bloco de texto inteiro.

```
1 $ echo "O processo inteiro de instalacao pode demorar entre 15~30 min
   dependendo da velocidade do seu computador e da sua internet"
2
3 $ echo "Instalando as dependencias necessarias para compilar corretamente"
4 $ sudo apt-get install -y g++ build-essential libboost-all-dev qt4-dev-
   tools libaudio-dev libgtk-3-dev libxt-dev doxygen tclsh
5
6
7 $ echo "Adicionando o repositorio do simulador"
8 $ sudo apt-add-repository -y ppa:gnurubuntu/rubuntu
9 $ sudo apt-get update
10
11
12 $ echo "Instalando o rcserver"
13 $ sudo apt-get install -y rcserver
14
15
16 $ echo "Instalando o rcservermonitor"
17 $ sudo apt-get install -y rcservermonitor
18
19
20 $ echo "Instalando o rcsoccersim"
21 $ sudo apt-get install -y rcsoccersim
22
23
24 $ echo "Instalando a librcsc"
25 $ wget http://c3sl.dl.osdn.jp/rctools/51941/librcsc-4.1.0.tar.gz
26
27 $ tar -zxpf librcsc-4.1.0.tar.gz
28 $ cd librcsc-4.1.0
29 $ sudo ./configure
```

```
30 $ sudo make
31 $ sudo make install
32
33
34 $ cd ~
35 $ echo "Instalando o agent2d"
36 $ wget http://c3sl.dl.osdn.jp/rctools/55186/agent2d-3.1.1.tar.gz
37 $ tar -zxpf agent2d-3.1.1.tar.gz
38 $ cd agent2d-3.1.1
39 $ sudo ./configure
40 $ sudo make
41
42 $ cd ~
43 $ echo "Instalando o UVA Trilearn"
44 $ wget http://inf.ufes.br/~pet/projetos/Simulacao_2D/Sim2D/
    trilearn_base_sources-3.3-v13-by_PET.tar.gz
45 $ tar -zxpf trilearn_base_sources-3.3-v13-by_PET.tar.gz
46 $ cd trilearn_base_sources-3.3-v13-by_PET
47 $ sudo ./configure
48 $ sudo make
49
50 $ echo "Corrigindo problema do separador decimal"
51 $ sudo echo 'LC_NUMERIC="C"' >> ~/.bashrc
52
53
54 $ echo "Instalacao concluida!"
55 $ echo "Faca logoff e logon novamente ou reinicie o computador para aplicar
    todas as mudancas!"
```

## APÊNDICE B – Código com novas estratégias para o playerteams.cpp

```

1  /*
2  Copyright (c) 2000–2003, Jelle Kok, University of Amsterdam
3  All rights reserved.
4
5  Redistribution and use in source and binary forms, with or without
   modification, are permitted provided that the following conditions are
   met:
6
7  1. Redistributions of source code must retain the above copyright notice,
   this list of conditions and the following disclaimer.
8
9  2. Redistributions in binary form must reproduce the above copyright notice
   , this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.
10
11 3. Neither the name of the University of Amsterdam nor the names of its
   contributors may be used to endorse or promote products derived from
   this software without specific prior written permission.
12
13 THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
   AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO,
   THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
   PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
   CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
   EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
   PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
   PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
   LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
   NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
   SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
14 */
15
16 /*! \file PlayerTeams.cpp
17 <pre>
18 <b>File:</b>                PlayerTest.cpp
19 <b>Project:</b>            Robocup Soccer Simulation Team: UvA Trilearn
20 <b>Authors:</b>           Jelle Kok
21 <b>Created:</b>           10/12/2000
22 <b>Last Revision:</b>    $ID$
23 <b>Contents:</b>         This file contains the class definitions for the

```

```

24         Player that are used to test the teams' high level
25         strategy.
26 <hr size=2>
27 <h2><b>Changes</b></h2>
28 <b>Date</b>           <b>Author</b>           <b>Comment</b>
29 10/12/2000         Jelle Kok           Initial version created
30 </pre>
31 */
32
33 #include "Player.h"
34
35
36 /*!This method is the first complete simple team and defines the actions
37    taken by all the players on the field (excluding the goalie). It is
38    based on the high-level actions taken by the simple team FC Portugal
39    that it released in 2000. The players do the following:
37    - if ball is kickable
38        kick ball to goal (random corner of goal)
39    - else if i am fastest player to ball
40        intercept the ball
41    - else
42        move to strategic position based on your home position and pos ball
43 */
44
45 SoccerCommand Player::deMeer5( )
46 {
47
48     SoccerCommand soc(CMD_ILLEGAL);
49     VecPosition    posAgent = WM->getAgentGlobalPosition();
50     VecPosition    posBall  = WM->getBallPos();
51     int            iTmp;
52     int sideGoal =(-1 + 2 * (WM->getCurrentCycle()%2));
53     int posXField = 40;
54
55     /*****Estrategia I: Antes de inicio da partida*****/
56
57     if( WM->isBeforeKickOff( ) )
58     {
59         if( WM->isKickOffUs( ) && WM->getPlayerNumber() == 9) // 9 takes kick
60         {
61             if( WM->isBallKickable( ) )
62             {
63                 VecPosition posGoal( PITCH_LENGTH/2.0,
64                                     sideGoal*
65                                     0.4 * SS->getGoalWidth( ) );
66                 soc = kickTo( posGoal, SS->getBallSpeedMax( ) ); // kick maximal

```

```

67     Log.log( 100, "take kick off" );
68     }
69     else
70     {
71         soc = intercept( false );
72         Log.log( 100, "move to ball to take kick-off" );
73     }
74     ACT->putCommandInQueue( soc );
75     ACT->putCommandInQueue( turnNeckToObject( OBJECT_BALL, soc ) );
76     return soc;
77     }
78     if( formations->getFormation() != FT_INITIAL || // not in kickoff
formation
79         posAgent.getDistanceTo( WM->getStrategicPosition() ) > 2.0 )
80     {
81         formations->setFormation( FT_INITIAL );          // go to kick_off
formation
82         ACT->putCommandInQueue( soc=teleportToPos( WM->getStrategicPosition()
));
83     }
84     else // else turn to center
85     {
86         ACT->putCommandInQueue( soc=turnBodyToPoint( VecPosition( 0, 0 ), 0 )
);
87         ACT->putCommandInQueue( alignNeckWithBody( ) );
88     }
89     }
90
91 /*****Durante a partida*****/
92
93     else
94     {
95         formations->setFormation( FT_433_OFFENSIVE );
96         soc.commandType = CMD_ILLEGAL;
97
98         if( WM->getConfidence( OBJECT_BALL ) < PS->getBallConfThr() )
99         {
100             ACT->putCommandInQueue( soc = searchBall() ); // if ball pos
unknown
101             ACT->putCommandInQueue( alignNeckWithBody( ) ); // search for it
102         }
103         else if( WM->isBallKickable() // if kickable
104         {
105
106             VecPosition posGoal( PITCH_LENGTH/2.0, sideGoal * 0.4 * SS->
getGoalWidth() );
107

```

```

108
109     soc = kickTo( posGoal, SS->getBallSpeedMax() ); // kick maxima
110     ACT->putCommandInQueue( soc );
111     ACT->putCommandInQueue( turnNeckToObject( OBJECT_BALL, soc ) );
112     Log.log( 100, "kick ball" );
113 }
114
115
116 /*****Se mais rapido que a bola*****/
117
118 else if( WM->getFastestInSetTo( OBJECT_SET_TEAMMATES,
119 OBJECT_BALL, &iTmp )
120         == WM->getAgentObjectType() && !WM->isDeadBallThem() )
121     {
122         // if fastest to ball
123         Log.log( 100, "I am fastest to ball; can get there in %d cycles",
124 iTmp );
125         soc = intercept( false ); // intercept the ball
126
127         if( soc.commandType == CMD_DASH && // if stamina low
128             WM->getAgentStamina().getStamina() <
129             SS->getRecoverDecThr()*SS->getStaminaMax()+200 )
130         {
131             soc.dPower = 30.0 * WM->getAgentStamina().getRecovery(); // dash
132             slow
133             ACT->putCommandInQueue( soc );
134             ACT->putCommandInQueue( turnNeckToObject( OBJECT_BALL, soc ) );
135         }
136         else // if stamina high
137         {
138             ACT ->putCommandInQueue( soc ); // dash as intended
139             ACT ->putCommandInQueue( turnNeckToObject( OBJECT_BALL, soc ) );
140         }
141     }
142     else if( posAgent.getDistanceTo(WM->getStrategicPosition()) >
143             1.5 + fabs(posAgent.getX()-posBall.getX())/10.0
144             // if not near strategic
145     pos
146     {
147         if( WM->getAgentStamina().getStamina() > // if stamina high
148             SS->getRecoverDecThr()*SS->getStaminaMax()+800
149         )
150         {
151             soc = moveToPos(WM->getStrategicPosition(),
152                             PS->getPlayerWhenToTurnAngle());
153             ACT ->putCommandInQueue( soc ); // move to strategic
154             pos
155             ACT ->putCommandInQueue( turnNeckToObject( OBJECT_BALL, soc ) );

```

```

150     }
151     else // else watch ball
152     {
153         ACT->putCommandInQueue( soc = turnBodyToObject( OBJECT_BALL ) );
154         ACT->putCommandInQueue( turnNeckToObject( OBJECT_BALL, soc ) );
155     }
156 }
157 else if( fabs( WM->getRelativeAngle( OBJECT_BALL ) ) > 1.0 ) // watch
ball
158 {
159     ACT->putCommandInQueue( soc = turnBodyToObject( OBJECT_BALL ) );
160     ACT->putCommandInQueue( turnNeckToObject( OBJECT_BALL, soc ) );
161 }
162 else // nothing to do
163     ACT->putCommandInQueue( SoccerCommand(CMD_TURNNECK,0.0) );
164 }
165 return soc;
166
167
168 /**Estrategia 2: Chutar de acordo com a coordenada do jogador**/
169
170
171 else if( WM->isBallKickable() // if kickable
172     {
173         if (posAgent.getX() < 0)/*esta no nosso campo*/
174         {
175             soc = kickTo(VecPosition(posAgent.getX()+10, posAgent.getY()),
176                 1);
177         }
178         else
179         {
180             if (posAgent.getX() < 35)
181             {
182                 soc = kickTo ( VecPosition (posXField,-15), 3); //
183                 velociddde de 3m/s
184             }
185             else
186             {
187                 VecPosition posGoal( PITCH_LENGTH/2.0, sideGoal * 0.4 * SS
188                 ->getGoalWidth() );
189                 soc = kickTo( posGoal, SS->getBallSpeedMax() ); // kick
190                 maximal
191             }
192         }
193     }
194     ACT->putCommandInQueue( soc );
195     ACT->putCommandInQueue( turnNeckToObject( OBJECT_BALL, soc ) );
196 }

```

```
192
193
194 /*****Estrategia 3: DRIBBLE*****/
195
196     else if( WM->isBallKickable() ) // if kickable
197     {
198         if (posAgent.getX() < 35) //middle field of attack
199         {
200             soc = dribble (0,DRIBBLE_SLOW);
201         }
202         else
203         {
204             VecPosition posGoal(PITCH_LENGTH/2.0,sideGoal*0.4*SS->getGoalWidth
205             ());
206             soc = kickTo( posGoal, SS->getBallSpeedMax() ); // kick maximal
207             ACT ->putCommandInQueue( soc );
208             ACT ->putCommandInQueue( turnNeckToObject( OBJECT_BALL, soc ) );
209         }
210
211         else // nothing to do
212             ACT->putCommandInQueue( SoccerCommand(CMD_TURNNECK,0.0) );
213     }
214     return soc;
215 }
```