SAPIENZA
UNIVERSITÀ DI ROMA

Design and development of a PID controller to optimise the approach to the ball in robot kicker shooting

Facoltà di Ingegneria dell'informazione, informatica e statistica

Corso di Laurea in Ingegneria Informatica ed Automatica

Candidate

Samuele Civale

ID number 1938135

Thesis Advisor

Prof. Thomas Alessandro Ciarfuglia

Academic Year 2023/2024

Thesis not yet defended

Design and development of a PID controller to optimise the approach to the ball in robot kicker shooting

Bachelor's thesis. Sapienza – University of Rome

This thesis has been typeset by LaTeX and the Sapthesis class.

Author's email: samuele.civale@gmail.com

Abstract

The design of a controller represents a crucial element in the optimisation process of a system and its interaction with the surrounding environment. The primary objective of this research is to address the instability encountered in the behaviour of the kicking robot when approaching the ball, by implementing a Proportional-Integral-Derivative (henceforth PID) controller. The design phase was conducted using the SimRobot software, which allows the detailed simulation of the robot's dynamics on the pitch. This research is not limited to solving the specific problem of instability, but also aims to contribute to the broader understanding of control dynamics in robotic systems operating in dynamic and competitive environments, such as that characteristic of the RoboCup.

The thesis is organised into five chapters, each devoted to a specific aspect of the research. In the first chapter, the problem under analysis is outlined and a contextual framework is provided to understand its importance. The second chapter focuses on the tools used, offering an in-depth description of the simulator and robot involved in the work. In the third chapter, the research methodology is examined in detail, outlining the procedures adopted for the design of the controller and their practical application, and a comparison between the old and the new implementation. In the fourth chapter, the results obtained during the research are presented and analysed. Finally, in the fifth chapter, conclusions are drawn, highlighting the implications of the results and proposing possible future developments.

Contents

Chapter 1

Introduction

1.1 Contextualisation of the RoboCup

The RoboCup, also known as the Robot World Cup, was conceived in 1997 with the audacious goal of developing a team of humanoid robots capable of competing with the world champion football team by 2050.

In the 1990s, the idea of having robots play football first emerged thanks to Professor Alan Mackworth of the University of British Columbia [4]. In 1992, a group of Japanese researchers discussed the great challenges in Artificial Intelligence and proposed the use of football as a vehicle for disseminating the current state of science and promoting technological innovation. After feasibility studies, in June 1993, a group of researchers, including Minoru Asada, Yasuo Kuniyoshi and Hiroaki Kitano, launched the robot competition initially called Robot J-League. The term 'J-League' refers to Japan's top football league. Later, the competition was renamed 'RoboCup' in response to positive reactions from the international research community. Independently at the same time, Professor Minoru Asada's laboratory at Osaka University and Professor Manuela Veloso together with her student Peter Stone at Carnegie Mellon University were working on robots capable of playing football. Without the participation of these pioneers in the field, RoboCup would never have seen the light of day. In September 1993, specific regulations were established and organisational and technical problems were addressed at various conferences and workshops.

At the same time, Noda's ETL team announced Soccer Server version 0, an open system simulator for multi-agent football, followed by version 1.0 distributed via the web. The first public demonstration took place at IJCAI-95. During this conference, it was announced that the first Robot Football World Cup would be organised in conjunction with IJCAI-97.

It was also decided to organise the Pre-RoboCup-96 in conjunction with the International Conference on Robotics and Intelligence Systems (IROS-96) in Osaka. This competition comprised eight teams engaged in a simulation championship and demonstrations of real robots. Despite its limited size, this event contributed significantly to promoting research and education in the field.

The first official RoboCup competition and conference was held in 1997, with the participation of over 40 teams, both real and virtual. Since then, RoboCup has

continued to grow, becoming a major international event involving teams of robots in the context of football, stimulating development and research in this field.

Over the years, the RoboCup has evolved, not remaining anchored only to football simulations, but also embracing new types of challenges. Currently, the competition is divided into five main leagues, each with a distinctive focus:

- **RoboCup Soccer:** This league focuses on football competition between teams of robots and features several categories, including Small Size, Middle Size, Humanoid and 2D Simulation
- **RoboCup Rescue:** Dealing with simulations of rescue operations in disaster scenarios, this league aims to develop robots capable of operating in hazardous environments and assisting in rescue operations.
- **RoboCup@Home:** Aimed at home and assistance robotics, this league challenges participants to develop robots capable of performing useful tasks in home environments or assisting people with special needs.
- **RoboCup Industrial:** With a focus on the application of advanced robotic technologies in industrial settings, this league aims to improve automation and efficiency in manufacturing environments.
- **RoboCupJunior:** Aimed at younger students, this league provides a platform to engage future talent in robotics through disciplines such as robotic football, dance and other creative challenges.

The SPQR teams, representing the Antonio Ruberti Department of Computer, Control and Management Engineering at Sapienza University of Rome, have been actively participating in RoboCup competitions since 1998. My involvement focused on the distribution of the software used in the NAO robots, which compete in the Standard Platform League[2].

1.2 Motivation, Problem Relevance, Aim and Objectives of the Thesis

The design of a PID controller aimed at managing the approach to the ball in the RoboCup originates from the need to address one of the crucial challenges in football robotics. The precise control of how the robot approaches the ball plays a key role in ensuring the success of robot football teams, directly impacting the accuracy and speed of the game's actions. The lack of optimal control can compromise the robot's overall performance during competitions.

This thesis aims to contribute significantly to the understanding and optimisation of ball approach control, addressing a key challenge in the field of robotics applied to football. The RoboCup environment provides an excellent proving ground for the development and testing of innovative solutions that can be applied outside the football context. The specific application concerning the regulation of the approach speed to the ball can be generalised to any goal involving a robot moving in three-dimensional space. The optimal solutions developed for the RoboCup can thus be

Figure 1.1. RoboCup 2023

successfully adapted to self-driving vehicles, nanorobots used in the biomedical field and other applications.

The central objective of this thesis is the design, implementation and evaluation of a Proportional-Integral-Derivative (PID) controller specifically conceived to manage the approach to the ball by a soccer robot. The challenge is to optimise the regulation of the robot's approach speed to the ball, thereby improving its performance during matches.

In order to complete the central objective, a number of specific aims of the thesis need to be addressed. These include an in-depth analysis of the RoboCup context and the requirements associated with the approach to the ball. In addition, a detailed study of PID controllers is planned, followed by the design of a controller specially adapted to the specific requirements of the competition. This will be followed by the implementation and integration of this controller into the robotic system. A subsequent phase will include experimentation and performance evaluation through targeted tests, with the aim of analysing the results obtained and making any optimisations to the controller.

Chapter 2

Tools Used

2.1 Simulator description

The software used for analysing robot behaviour and designing the controller was SimRobot, included in the B-Human package. This package makes use of the SimRobot12 physical robot simulator as the main interface for software development. The simulator is not only used to work with simulated robots, but also serves as a graphical user interface to play log files and connect to physical robots via Ethernet or Wi-Fi.

Within SimRobot, it is possible to work on a number of ready-made scenarios. Personally, I focused my work on `bh.ros2`, making changes to it to better focus on the aspects of interest to me (specific changes will be discussed in the chapter). I mainly exploited SimRobot's Scene view, Plot views and Data views: the former allows us to observe the simulation through a dynamic representation of the playing field, the robots and the ball.

The Plot views allow the visualisation of data sent by the robot's control programme through the use of the PLOT macro. Data views, on the other hand, provide a structured visualisation of the data, facilitating developers to easily monitor and modify complex variables or fields over time. This approach simplifies the process of analysing data and making changes, improving efficiency in the development and debugging of the robot software. [1].

Let us now analyse the main advantages and disadvantages of using the simulator:

Advantages:

1. **reduced costs:** The use of simulators can significantly reduce the costs associated with developing and testing new algorithms and strategies. By eliminating the need for physical hardware, teams can concentrate on the software part without worrying about purchasing expensive components.

2. **Controlled environment:** Simulators provide a controlled virtual environment, allowing participants to play out specific scenarios, adjust variables and test the performance of their robots under controlled and repeatable conditions.

3. **Developer speed:** Simulation allows developers to quickly iterate and test

different strategies without having to wait for the physical construction of the robot or participation in live events. This speeds up the development process.

4. **Access to shared resources:** Sharing standard simulation environments can facilitate collaboration between teams and allow them to confront each other on neutral ground.

Limitations:

1. **Lack of realism:** Although the simulator attempts to replicate the real-world environment as closely as possible, there may be some lack of even minimal aspects of realism. This can lead to robot behaviour and performance that differs from that which would occur in a physical environment.

2. **Model complexity:** Creating an accurate and complex model of the environment and robots can be time and resource consuming. In addition, the complexity of the model may increase the demand for computing power.

3. **Overfitting:** Algorithms and strategies developed in a simulator may be too adapted to the specific conditions of the simulation and may not generalise well in the real world.

The scene view of the simulator consists of a shell from which it is possible to issue the commands to start and end the simulation, a window in which it is possible to display the trend of the parameters of interest by means of a function that shows the variation of the latter, and finally a section in which it is possible to see other data and general information about the robot, the latter however not being of interest to us.

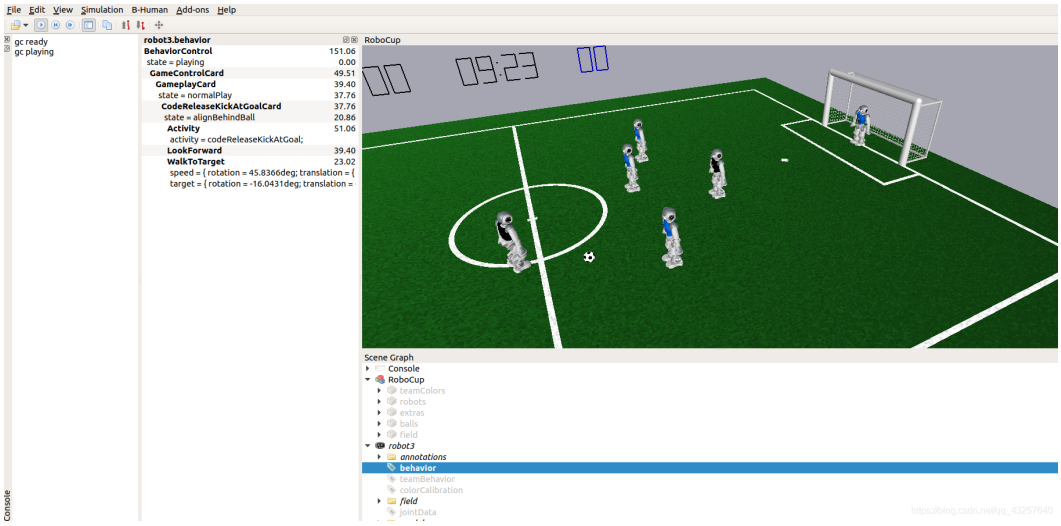


Figure 2.1. SimRobot. Scene View

2.2 Features of the Robot

The NAO humanoid robot is characterised by its mobility and innovative sensor design. With 25 degrees of freedom, it offers remarkable flexibility thanks to Maxon's coreless brush DC motors. Its patented pelvis kinematics and unique actuation system contribute to its agility.

In the context of the RoboCup, NAO reveals its sensory power. The 30 FPS CMOS camera, gyroscopes, accelerometers and underfoot force sensors provide real-time data essential for the challenges of the competition. Magnetic rotary encoders and front bumpers on the feet provide collision detection, while capacitive sensors on the forehead enable tactile input.

The robot comes with integrated software, such as Choregraphe, which supports functions such as speech synthesis and coloured shape recognition. Communication is via Ethernet or 802.11g wireless. NAO is programmable in several languages, including C, C++, Python and Urbi, offering versatility for developers.

Thanks to large-scale production and cost reduction, NAO is now affordable for academic institutions, solving financial challenges and opening up new research possibilities within the RoboCup. [6]

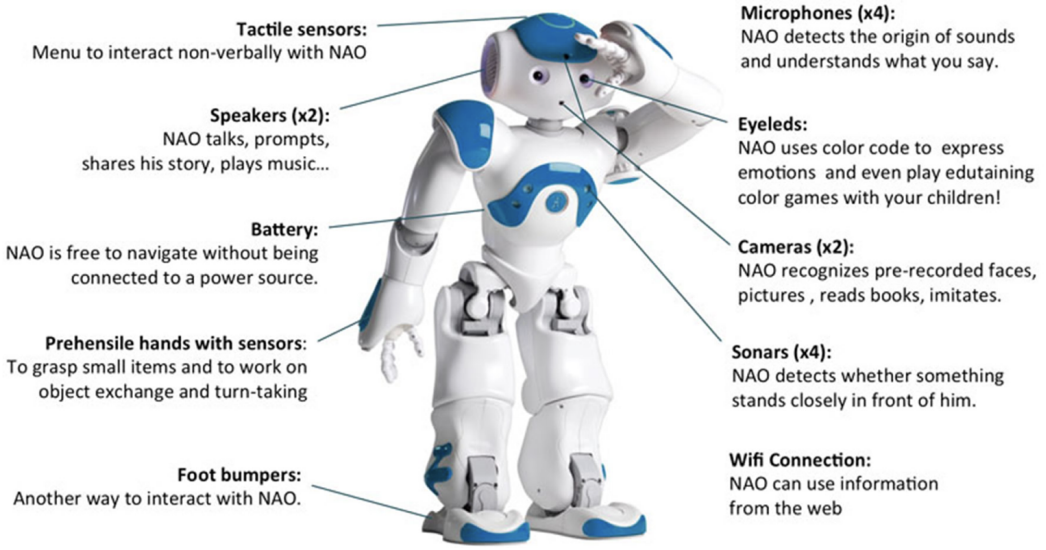


Figure 2.2. Nao Robot.

Chapter 3

Methodology

3.1 PID controller architecture

The PID controller is a control device widely used in engineering and automation to regulate dynamic systems. Its name is derived from its three fundamental components: Proportional (P), Integrative (I), and Derivative (D). This type of controller is designed to keep the controlled variable (output) as close as possible to the desired reference value.

The weighted sum of these three terms constitutes the overall output of the PID controller. In algebraic terms:

$$u(t) = k_p e(t) + k_i \int_0^t e(\tau)d\tau + k_d \frac{de(t)}{dt}$$

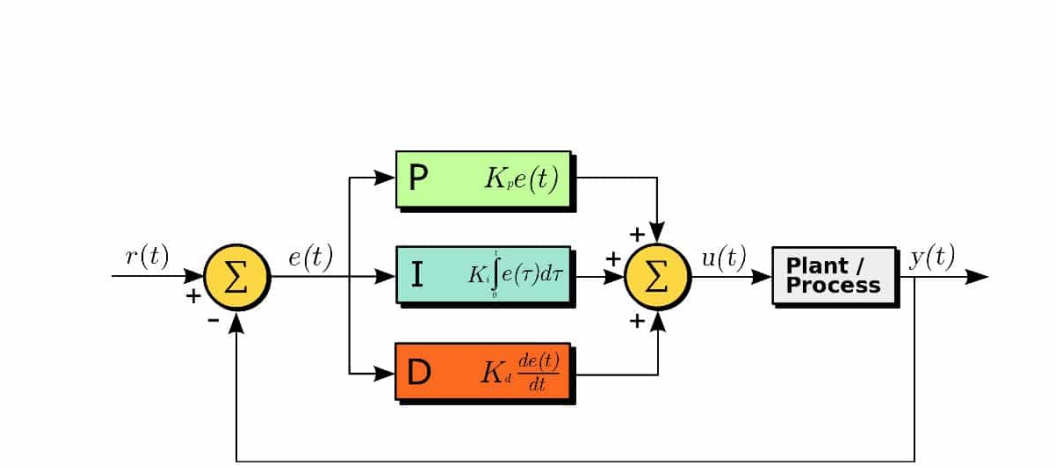


Figure 3.1. Diagram of a PID controller

Let us look at the three components in detail:

- **Proportional (P):**

  the proportional component responds proportionally to the actual error, which is the difference between the desired value and the actual value of the controlled variable. This means that the greater the error, the greater the output of the proportional term. Its main function is to reduce the current error and bring the controlled variable closer to the reference value.

- **Integrative (I):**

  The integrative component is proportional to the integral of the error over time. It therefore measures the cumulative sum of past errors. The integrative term helps to eliminate errors over time by reducing any persistent discrepancies between the controlled variable and its desired value. It helps to compensate for the accumulated error over time and stabilise the system.

- **Derivative (D):** The derivative component responds to the rate of change of the error over time. The derivative term anticipates the future behaviour of the system, reducing the risk of oscillations and improving stability. It counteracts rapid changes in the error, helping to prevent overfitting of the system.

advantages of PID controllers:

1. **Simplicity and Ease of Implementation:** PID controllers are simple to implement, requiring only error measurement and coefficient tuning, which can be done manually or automatically.

2. **Versatility:** PID controllers can handle a wide range of system dynamics and disturbances, adapting to changes in system parameters or the environment.

   Fast and Stable Responses: PID controllers can provide fast and stable responses by reducing error, eliminating steady-state error and minimising overshoot and oscillations.

PID controller advantages:

to Noise and Measurement Errors: PID controllers can be sensitive to noise and measurement errors, amplifying any fluctuations in the input signal and causing instability or oscillations. with Non-Linear Systems: PID controllers can encounter difficulties with non-linear systems, manifesting performance degradation or instability when system dynamics change significantly or unpredictably.**Integral Windup:** PID controllers can suffer from integral windup, where the integral term accumulates a large error over time, causing a noticeable delayed response. This typically occurs during system saturation or in the presence of significant disturbances.

We will begin our analysis by considering the context in which the function regulating the approach speed to the ball is called. Next, we will examine the previous implementation and the proposed new one.

3.2 Function call context

The function we are working on is "getApproachSpeed," which is called within the "LibStriker.h" library. The context in which it is called is the implementation of a basic behavior for a striker, referred to as the "card."

Card

The card is based on a behavior framework and declares the requirements and calls for various functionalities used within the behavior, such as the ball's position, opponents' positions, field dimensions, robot's position, etc.

Card's parameters

Some configurable parameters of the card are defined, such as walking speed (`walkSpeed`), initial waiting time (`initialWaitTime`), kick interval threshold (`kickIntervalThreshold`), and some PID tuning parameters (`kp`, `kd`, `minSpeed`, `ki`), etc.

Card Implementation

The `BasicStrikerCard` class inherits from the `BasicStrikerCardBase` class and implements the `preconditions` and `postconditions` methods. Preconditions and postconditions are conditions that must be verified before and after the execution of each state in the state machine.

State Machine

The card implements a state machine that manages the striker's behavior. The states include "start," "goToBallAndDribble," "goToBallAndKick," and "searchForBall." Each state has associated actions and transitions based on conditions.

1. **Start**:
   - *Transition Conditions:* After an initial waiting period (defined by `initialWaitTime`), the transition occurs to the "goToBallAndDribble" state.
   - *Actions:*
     1. Activates the skill to look forward (`theLookForwardSkill()`).
     2. Activates the skill to stand (`theStandSkill()`).

- **GoToBallAndDribble**:
  - *Transition Conditions:* It moves to "searchForBall" if the ball is not seen within a certain timeout; otherwise, it checks if a kick can be executed and transitions to "goToBallAndKick" if so.
  - *Actions:*
    1. Activates the skill to move towards the ball with dribbling (`theGoToBallAndDribbleSkill()`). </enumerate
  - **GoToBallAndKick**:

    - *Transition Conditions:* It moves to "searchForBall" if the ball is not seen within a certain timeout; otherwise, it returns to "goToBallAndDribble" if a kick cannot be executed.
    - *Actions:*
        1. Determines whether to kick immediately or not (`doItAsap`).
        2. Calculates kick information using the striker's library (`theLibStriker`).
        3. Activates the skill to move towards the ball and aim for the shot (`theGoToBallAndKickSkill()`).
- **SearchForBall**:
    - *Transition Conditions:* It moves to "goToBallAndDribble" if the ball is seen.
    - *Actions:*
        1. Activates the skill to look forward (`theLookForwardSkill()`).
        2. Activates the skill to walk at a relative speed (`theWalkAtRelativeSpeedSkill()`).

`theGoToBallAndKickSkill()` Function

This function is called in the "goToBallAndKick" state and is used to direct the robot towards the ball with the intention of kicking it. Some parameters passed to this function include the direction angle towards the ball (`angle`), the type of kick (`typeKick`), whether it should be executed as soon as possible (`!doItAsap`), the maximum kick length (`std::numeric_limits<float>::max()`), some additional options like precise alignment and turning during the kick, and finally the approach speed (`theLibStriker.getApproachSpeed(range, kp, kd, minSpeed)`) and direction accuracy (`Rangea(0_deg,0_deg)`).

In sostanza, la funzione `theGoToBallAndKickSkill()` gestisce il movimento del robot verso il pallone e la pianificazione del calcio, utilizzando una serie di parametri per controllare il comportamento del robot durante l'approccio. La velocità di approccio è determinata dalla chiamata a `theLibStriker.getApproachSpeed()`, che utilizza un controllore PID con i parametri forniti (`range, kp, kd, minSpeed`).

Andremo quindi a concentrarci sulla funzione "getApproachSpeed" e di riflesso ci concentriamo sullo stato "goToBallAndKick".

3.3 Previous Implementation of the PID Controller

```
1 Pose2f LibStrikerProvider::getApproachSpeed(Rangef range, float
    kp, float kd, float minSpeed) const{
2   if(theFieldBall.recentBallPositionRelative().norm() > range.max
    )
3     return Pose2f(1,1,1);
4
```

```
5   if(kd>kp)
6     OUTPUT_WARNING("Derivative gain is greater than proportional
      gain, this may lead to an extremely slow convergence");
7
8   if(kd == 0)
9     OUTPUT_WARNING("Derivative gain is set to 0, this may lead to
       lack of convergence due to rippling");
10
11  Vector2f target = goalTarget(true, true);
12
13  float e_angle = abs((theRobotPose.inversePose * target).angle()
      ),
14        e_p = mapToRange(theFieldBall.recentBallPositionRelative
      ().norm()+e_angle  , range.min, range.max, minSpeed, 1.f),
15        e_d = theMotionInfo.speed.translation.norm()/1e3,
16        out = clip(kp*e_p - kd*e_d, minSpeed, 1.f);
17
18  return Pose2f(out,out,out);
19 }
```

The `getApproachSpeed` function takes as input `range`, which is the range of distances within which to call the function, `kp`, which is the proportional gain, `kd`, which is the derivative gain, and `minSpeed`, which is the minimum allowed speed. The function returns a `Pose2f`, where `Pose2f` is a data structure representing a position in two-dimensional space with coordinates (x, y) and an orientation angle θ.

If the norm of the ball's relative position with respect to the field is greater than the maximum value specified by `range.max`, the method returns a `Pose2f` with all values set to 1.

If the derivative gain is greater than the proportional gain or if the derivative gain is zero, it prints a warning indicating a possible difficulty in achieving convergence.

A target vector is calculated using a function called `goalTarget`.

The angular error `(e_angle)` between the robot's current position and the target, the proportional error `(e_p)` based on the norm of the ball's relative position and the angular error, and the derivative error `(e_d)` based on the robot's current speed are all calculated. The final output `(out)` is determined using proportional and derivative control and is clipped between the minimum and maximum allowed speed values.

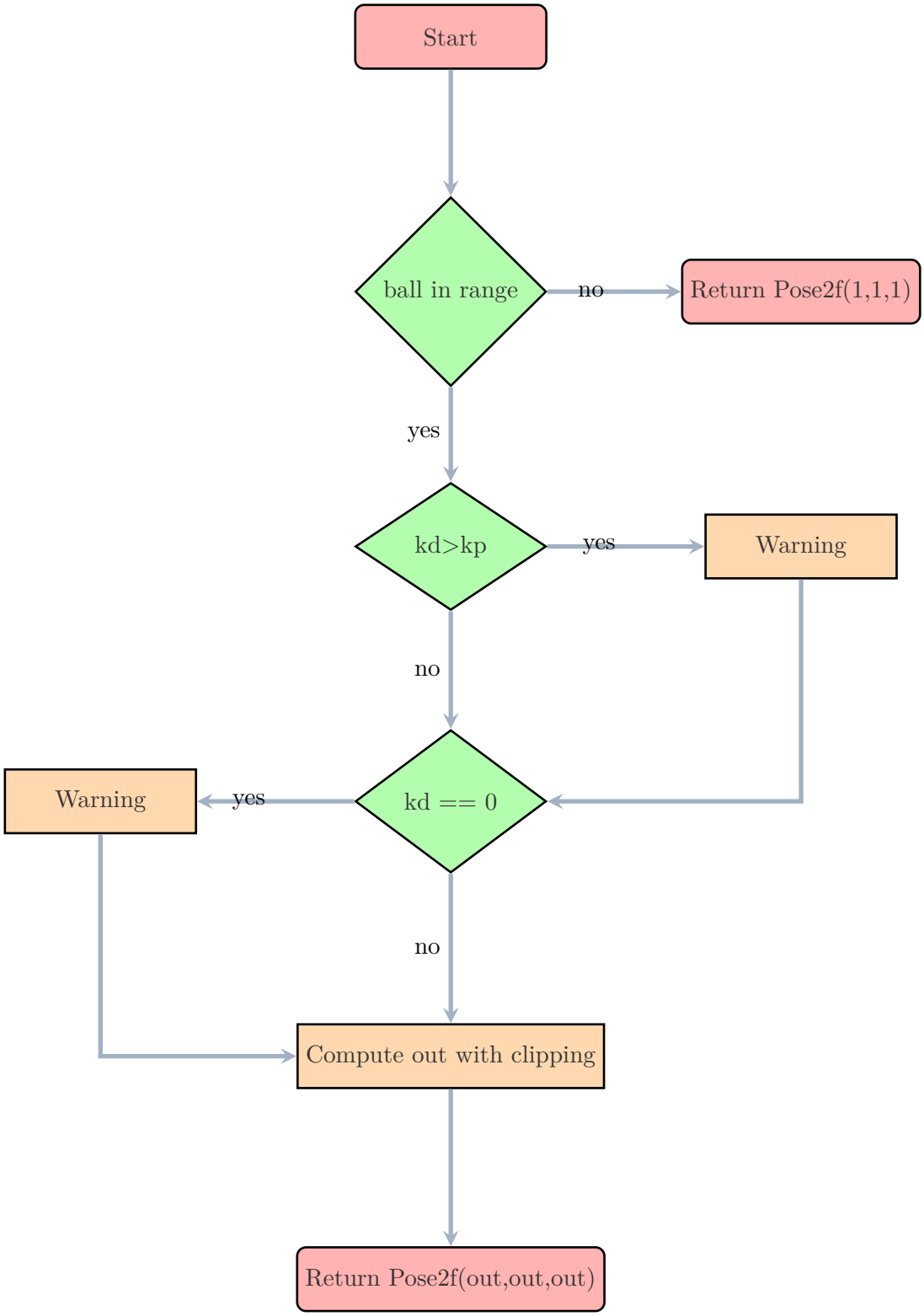
The `mapToRange` function is designed to map a value `val` from a given input range `[minInput, maxInput]` to an output range `[minOutput, maxOutput]`. The function uses the "clipping" operator `clip` to ensure that the input value `val` is within the specified input range.

The main formula for mapping is as follows:

$$result = minOutput + \frac{(v - minInput) \cdot (maxOutput - minOutput)}{sizeOfInputRange}$$

Where:

- v: The input value "clipped" within the range $[minInput, maxInput]$.
- `sizeOfInputRange`: The length of the input range, calculated as (`maxInput`−`minInput`).
- `sizeOfOutputRange`: The length of the output range, calculated as (`maxOutput` − `minOutput`).
- `result`: The value mapped to the output range.


Start
ball in range
no
Return Pose2f(1,1,1)
yes
kd>kp
yes
Warning
no
Warning
yes
kd == 0
no
Compute out with clipping
Return Pose2f(out,out,out)

3.4 Nuova implementazione del Controllore PID

```
1     /**
2  * PID controller that returns the approaching speed.
3  * @param range The distance range taken to pass from speed 1 to
     the specified minimum speed
4  * @param kp The controller proportional gain
5  * @param kd The controller derivative gain
6  * @param ki The controller integral gain
7  * @param minSpeed The minimum speed allowed
8  * @return speed in Pose2f in a range from minSpeed  to 1
9  */
10 static float integral = 0.0f;
11 float previousError = 0.0f;
12 const float MAX_INTEGRAL = 50.0f;
13 std::chrono::high_resolution_clock::time_point lastCallTime = std
     ::chrono::high_resolution_clock::time_point::min();
14 Pose2f LibStrikerProvider::myGetApproachSpeed(Rangef range, float
      kp, float kd, float ki, float minSpeed) const{
15   Vector2f position = theFieldBall.recentBallPositionRelative();
16
17   Angle rot = (theRobotPose.inversePose * position).angle();
18
19   std::cout<< "l'angolo    di: " << rot.toDegrees() << "\n";
20   if(position.norm() > range.max){
21     std::cout<< "position.norm > range.max:" << position.norm()
      << " > " << range.max << "\n";
22     return Pose2f(1,1,1);
23   }
24   if(kd>kp){
25     std:: cout << "kd > kp\n";
26     OUTPUT_WARNING("Derivative gain is greater than proportional
      gain, this may lead to an extremely slow convergence");
27   }
28   if(kd == 0){
29     std:: cout << "kd == 0\n";
30     OUTPUT_WARNING("Derivative gain is set to 0, this may lead to
       lack of convergence due to rippling");
31   }
32
33   Vector2f target = goalTarget(true, true);
34
35   auto currentTime = std::chrono::high_resolution_clock::now();
36   float dt = 0.0f;
37
38   if (lastCallTime.time_since_epoch().count() != 0){
39     std::chrono::duration<float> duration = currentTime -
      lastCallTime;
40     dt = duration.count();
41     std::cout << "dt: " << dt << "\n";
42   }
43
44   lastCallTime = currentTime;
45
46   float e_angle = abs((theRobotPose.inversePose * target).angle()
      );
47   std::cout << "e_angle calcolato: "<< e_angle << "\n";
```

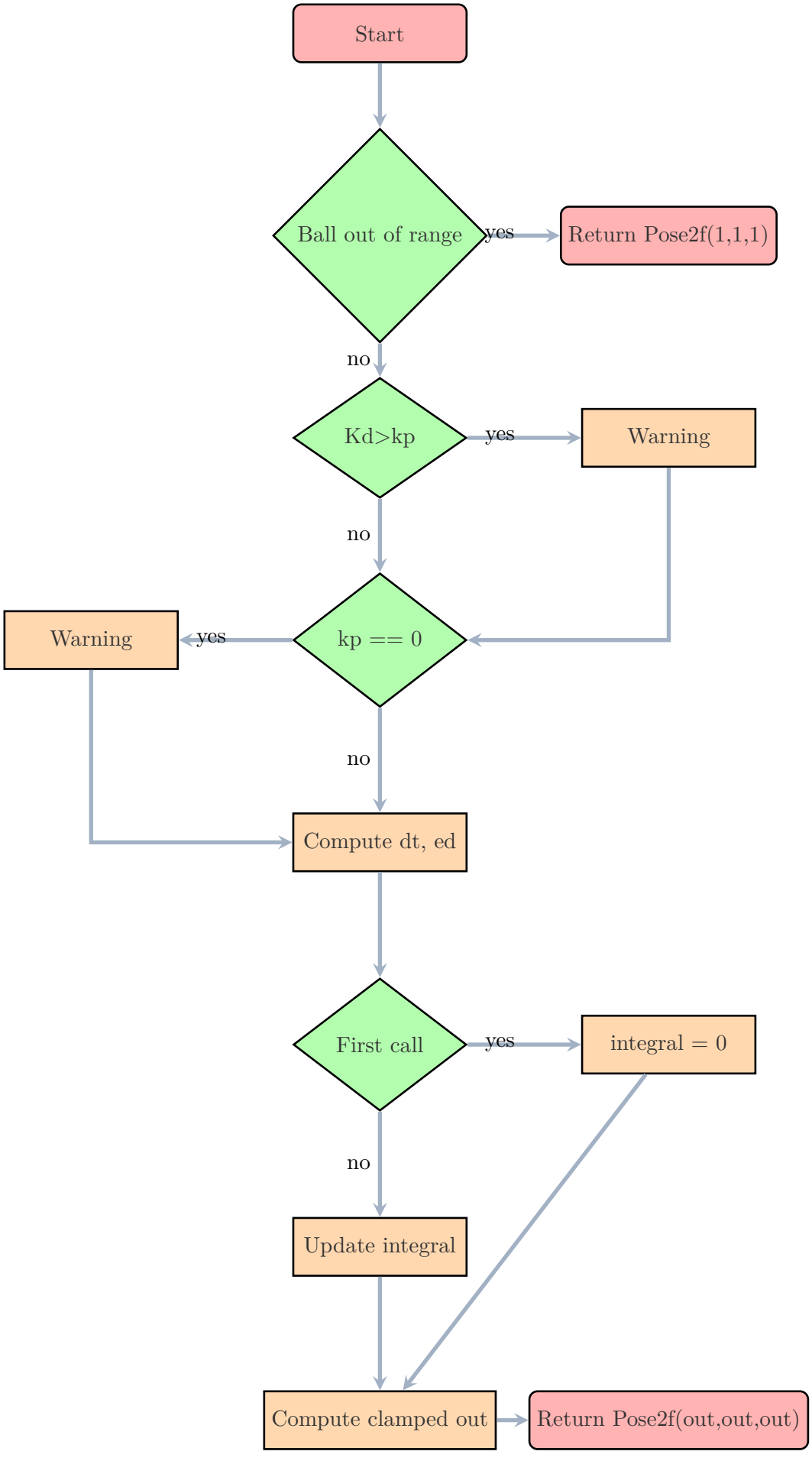
```
48
49
50    float e_p = mapToRange(theFieldBall.recentBallPositionRelative
        ().norm()+e_angle   , range.min, range.max, minSpeed, 1.f);
51    std::cout << "e_p calcolato: "<< e_p << "\n";
52
53    float e_d = calcDerivative(e_p, dt);
54    std::cout << "e_d calcolato: "<< e_d << "\n";
55
56    if (dt > 0){
57      integral += e_p*dt;
58      if (integral>MAX_INTEGRAL){
59        integral = MAX_INTEGRAL;
60      }
61      if (integral < -MAX_INTEGRAL){
62        integral = -MAX_INTEGRAL;
63      }
64    }
65    else{
66      integral = 0;
67      std::cout << "integral pre dovrebbe 0: "<< integral << "\n";
68    }
69
70    std::cout << "integral "<< integral << "\n";
71
72    float output = kp*e_p + ki*integral + kd*e_d;
73    std::cout << "output: "<< output << "\n";
74
75    float approachSpeed = customClamp(output, minSpeed, 1.0f);
76    std::cout<< "approachSpeed: " << approachSpeed << "\n";
77
78    return Pose2f(approachSpeed,approachSpeed,approachSpeed);
79  }
80  /**
81     * @param value the value to be clamped
82     * @param low the inferior limit
83     * @param high the superior limit
84     * @return the value clamped in range (low, high)
85    * **/
86  float LibStrikerProvider::customClamp(float value, float low,
      float high) const{
87    return low + std::max(0.0f,std::min(value, high-low));
88  }
89  float LibStrikerProvider::calcDerivative(float currentError,
      float dt) const{
90    if (dt == 0){
91      return 0.0f;
92    }
93    float derivativeError = (currentError-previousError)/dt;
94    previousError = currentError;
95    return derivativeError;
96  }
```

Let's analyze the code:

– 10-14: We define the variables that will be used during the function execution. `integral` is defined as static and represents the accumulation of er-

rors because we want its value to persist between calls, `previousError` is a variable used for the derivative calculation in the controller, `MAX_INTEGRAL` is a constant representing the maximum value that `integral` can assume to prevent windup, and `lastCallTime` is a variable initialized to the minimum possible value for `std::chrono::high_resolution_clock::time_point`, indicating that no previous call has been recorded.

- 15-33: The variables `position` and `rot` are defined, representing the ball's position relative to the field and the angle calculated from the robot's inverse position and the ball's position, respectively. Some debug and warning prints are made, and it is defined how to act if the ball is at a distance greater than the maximum defined in `range`. Finally, the ball's position in Cartesian coordinates is saved in the `target` variable.
- 35-44: The delta time between calls is calculated.
- 46-55: Calculations related to the PID controller are performed: `e_angle` represents the absolute angle between the robot's inverse position and the desired target, `e_p` is the proportional term and accounts for the ball's distance from the robot's current position and the angle between the robot and the target. The `mapToRange` function normalizes this value within the specified range. `e_d` represents the derivative term, calculated using the `calcDerivative` function. The implementation of this function can be seen from line 89 to line 96.
- 56-69: This step calculates and updates the integral term of the PID controller, considering the time variation between successive function calls. The integral term is constrained to prevent undesirable phenomena such as integral windup.
- 70-79: In `output`, the computation is carried out according to the PID controller architecture rules. This value is then "clamped" between the minimum allowed speed and 1. Additional debug prints are made, and the approach speed is returned with a `Pose2f` structure.
- 80-96: Auxiliary functions `calcDerivative` and `customClamp` are implemented.


Start
Ball out of range
yes
Return Pose2f(1,1,1)
no
Kd>kp
yes
Warning
no
Warning
yes
kp == 0
no
Compute dt, ed
First call
yes
integral = 0
no
Update integral
Compute clamped out
Return Pose2f(out,out,out)

3.5 Comparison Between the Two Implementations

Examining the two codes, it is evident that the new PID controller implementation includes an integral term `(ki*integral)` in the output calculation formula. This makes the PID controller more complex compared to the old implementation, which only included proportional and derivative terms. It is important to note that the absence of the integral term in the first implementation identifies it as a PD (proportional-derivative) controller.

The advantages of a PD controller over a PID controller include its simplicity of implementation and tuning. However, it has a significant disadvantage: it is less effective at handling steady-state errors. When the system approaches the desired value, the integral term in a PID controller helps accumulate and gradually correct the residual error, stabilizing the system and making it more precise.

Moreover, the absence of the integral term in the PD controller can lead to increased sensitivity to disturbances. Since there is no compensation for cumulative errors over time, the PD controller might be more susceptible to long-term disturbances. In summary, while the PD controller is simpler to implement and tune, the PID controller offers greater precision and stability in handling steady-state situations.

In the new implementation, there is a different approach to handling the time interval (`dt`): `std::chrono`[3] is used to calculate the duration between function calls and consequently modify the derivative term calculation. In the first implementation, the calculation of `e_d` is done using the robot's speed norm `(theMotionInfo.speed.translation.norm())` and dividing the result by 1000 (1e3). This provides an approximation of the derivative of the error with respect to time `(e_d)`, and `dt` is then manually set. In the second implementation, however, the elapsed time between function calls is measured using `std::chrono::duration`; `dt` is then calculated dynamically based on system time. Subsequently, `dt` is used in the derivative error calculation `e_d` in the `calcDerivative` function.

In summary, the new implementation offers the addition of the integral term, more accurate time interval handling, and a more precise derivative calculation compared to the old implementation. These advantages can translate into greater precision and stability of the control system, especially in situations where capturing more detailed temporal variations is crucial. The balanced approach among the PID components contributes to more predictable and responsive behavior.

Despite the new features leading to various benefits, the new implementation also presents some potential challenges and disadvantages that need to be considered:

- **Additional Complexity:** The addition of the integral term and more detailed time interval management increases the complexity of the control logic. Greater complexity can make the code harder to understand, maintain, and debug.
- **Sensitivity to PID Parameters:** A PID controller is sensitive to the tuning of its parameters (`kp`, `ki`, `kd`). The presence of the integral term can lead to instability if the `ki` parameter is too high, while excessively high `kd` can cause unwanted oscillations. The need for precise parameter tuning may require time and effort.
- **Possibility of Noise Integration:** The addition of the integral term can introduce the possibility of integrating noise into the system. If noise is present in the error, the integral term can accumulate it over time, leading to an unstable response. Excessive noise integration can make the system sensitive to external disturbances.
- **Overfitting to Test Data:** Accurate tuning of PID parameters might be based on specific test scenarios, risking poor generalization to real-world conditions. This could pose problems if the system is used in operational environments different from the test conditions.
- **Additional Computational Resource Usage:** The addition of new features and more detailed time interval management may require more computational power, especially in embedded systems or with limited resources.

In conclusion, optimizing the PID controller offers significant advantages in terms of precision and stability but requires a balanced approach considering the additional complexity and tuning challenges.

Chapter 4

Results and Analysis

4.1 Experimental Procedure

Since the code is non-deterministic, as the function `getApproachSpeed` is called based on other computations outside the development of the PID controller, which places the robot into the "goToBall&kick" state under various positions and circumstances, I decided to proceed as follows to evaluate the effectiveness of my implementation: I provided both functions with the same values for activation range, proportional gain, and derivative gain.

The choice of these parameters was made through a "grid search" exploration on a set of values. This process involves defining a set of candidate values for each model parameter and running the model with all possible combinations of these values. I selected a set of three values for each parameter and evaluated the best combination for both models. After choosing the three values that provided the best results for both models, I ran 25 simulations for each implementation.

To better assess the approach to the ball, I modified the scenario `bh.ros2`, removing the opposing robots (which were stationary during the simulation) to analyze the approach times more accurately. In the initial setup, as shown in Figure 4.1, there is a clock starting from 10:00 (mm:ss) and an opposing robot acting as a goalkeeper, remaining stationary. The goalkeeper was kept because, during the shooting phase, its stationary position serves as a crucial reference point. This setup allows us to observe whether the robot can kick the ball while avoiding the goalkeeper obstacle and accurately targeting the goal. It tests the robot's ability to calculate the trajectory and force required to overcome the obstacle and achieve the desired goal.

The starting conditions for each simulation are therefore identical to best evaluate the characteristics of both implementations. I chose to perform 25 simulations to ensure statistical validity of the results obtained. We will now define the evaluation metrics used.

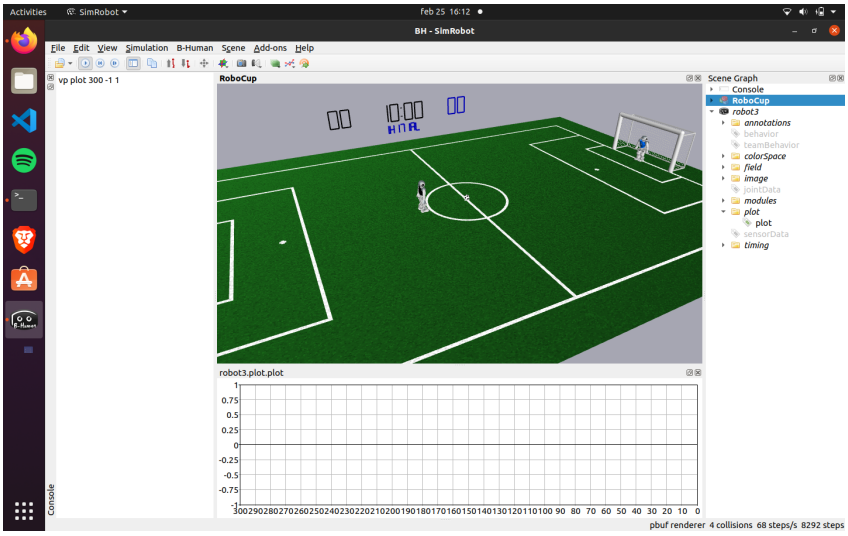


Figure 4.1. initial setup

4.2 Evaluation Metrics

The evaluation metrics used in our analysis are three: overshooting, approach time, and shooting conversion rate.

Overshooting is a measure that indicates how much a system goes beyond the desired value before stabilizing or reaching the set point (the target position that the system aims to reach and maintain) and is a critical metric for various reasons. Firstly, it directly affects the accuracy of the robot's positioning relative to the ball. Excessive overshooting could mean that the robot approaches the ball with too high a speed, exceeding the desired position and requiring additional time to correct the error. Moreover, significant overshooting can lead to unwanted oscillations or instability in the system, compromising the effectiveness of the ball approach. An overly aggressive control could cause oscillatory movements or erratic behaviors, negatively impacting the accuracy of the shot and the robot's ability to maintain a controlled trajectory; if it is too excessive, it might indicate the need to optimize the PID parameters to ensure more precise and stable behavior during the shooting phase. Carefully monitoring this metric can provide crucial insights for refining implementations and improving the overall effectiveness of the robot's control system. Therefore, in this context, a lower overshooting is preferable to a higher one.

Approach time is a metric that indicates the time from the start of the simulation to the moment of the robot's shot. It reflects the timeliness, efficiency, and precision of the robot during the shooting phase. An optimized approach time significantly contributes to the success of the robot's actions on the field and its ability to handle dynamic situations effectively. Intuitively, a shorter approach time is preferable to a longer one.

Finally, the last metric we consider is the shooting conversion rate. A higher conversion rate suggests greater precision and success in offensive actions, while a lower rate might indicate the need to optimize the shooting approach or the accuracy of ball control.

For a more detailed understanding of the data collection procedure, refer to Figure 4.2. The simulation was progressively advanced step by step, considering an image rendering frequency of 10 FPS (frames per second). In the top center of the figure, it is observed that the countdown clock shows 9:37. This implies that the approach time is 23 seconds ($10:00 - 9:37 = 00:23$ mm:ss).

In the lower part of the figure, the graph of the approach speed is presented. To obtain precise values for the maximum speed and the steady-state speed, these were printed on a terminal during the simulation execution.

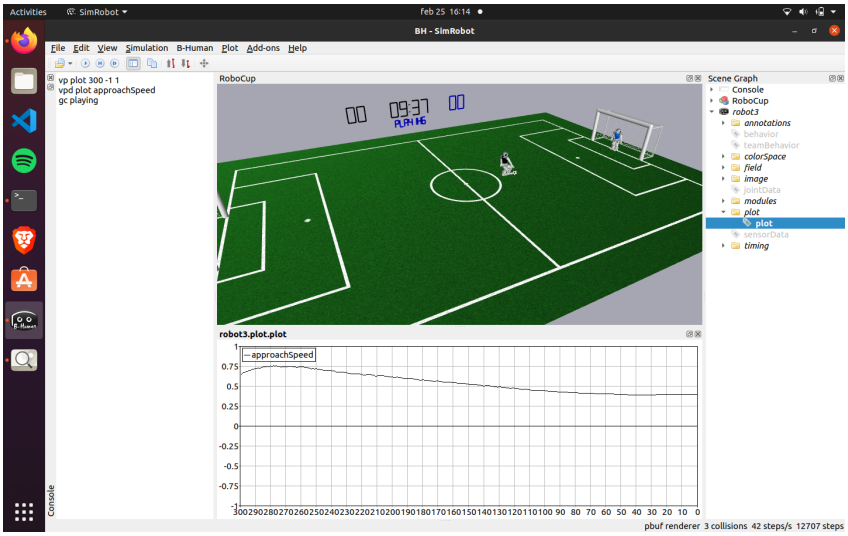


Figure 4.2. shooting moment of a simulation

4.3 Numerical Results

In the context of presenting the numerical results, I chose to use tables to provide a clear and organized representation of the measurements obtained. This choice aims to ensure an efficient and accessible presentation of the data collected during the experiment, allowing an easy evaluation of the performance of different implementations and highlighting the main results obtained.

Below are the main structural components of the tables:

- Columns:
  - `Y_Max`: Represents the maximum value reached by the robot's approach speed.
  - `Y_out`: Represents the steady-state output value reached by the robot's approach speed.

  - `Overshooting`: Represents the overshooting and is calculated with the following formula:
    $$\frac{Y\_Max - Y\_out}{Y\_out}$$
  - `Countdown (mm:ss)`: The remaining time in mm:ss format at the moment of the shot.
  - `Time (ss)`: The total time elapsed between the start and the moment of the shot.
  - `Goal`: Indicates whether the simulation resulted in a goal (1) or not (0).
- Data Rows: Each row represents a single simulation.
- Additional Rows: The rows labeled "Min", "Max", "Mean", "Median", and "Standard Deviation" provide summary statistics of the obtained results, offering an overall view of the system's performance in terms of accuracy, timeliness, and success in scoring goals.

New	Y_Max	Y_out	Overshoot	Countdown(mm:ss)	Time(ss)	Goal
1	0,49	0,44	0,1136	9,45	0,15	1
2	0,49	0,44	0,1136	9,44	0,16	1
3	0,45	0,44	0,0227	9,44	0,16	1
4	0,90	0,41	1,1951	9,44	0,16	1
5	0,60	0,44	0,3636	9,44	0,16	1
6	0,59	0,44	0,3409	9,43	0,17	1
7	0,48	0,45	0,0667	9,43	0,17	1
8	0,51	0,44	0,1591	9,42	0,18	1
9	0,67	0,44	0,5227	9,42	0,18	1
10	0,57	0,44	0,2955	9,41	0,19	1
11	0,48	0,44	0,0909	9,41	0,19	0
12	0,62	0,45	0,3778	9,41	0,19	1
13	0,65	0,45	0,4444	9,41	0,19	0
14	0,65	0,45	0,4444	9,41	0,19	1
15	0,63	0,44	0,4318	9,41	0,19	0
16	0,72	0,42	0,7143	9,40	0,20	1
17	0,85	0,45	0,8889	9,40	0,20	0
18	0,70	0,45	0,5556	9,40	0,20	1
19	0,60	0,44	0,3636	9,40	0,20	1
20	0,46	0,44	0,0455	9,40	0,20	1
21	0,76	0,44	0,7273	9,39	0,21	1
22	0,70	0,45	0,5556	9,38	0,22	1
23	0,82	0,40	1,0500	9,37	0,23	1
24	0,78	0,38	1,0526	9,36	0,24	1
25	1	0,40	1,5000	9,30	0,30	1
Min	0,45	0,38	0,0227	9,30	0,15	0
Max	1	0,45	1,50	9,45	0,30	1
Mean	0,6468	0,4352	0,4975	9,4068	0,1932	0,84
Median	0,63	0,44	0,4318	9,4100	0,19	1
Standard Deviation	0,1328	0,0103	0,3453	0,0096	0,0096	0,0000

Table 4.1. New implementation data table

OLD	Y_Max	Y_out	Overshoot	Countdown(mm:ss)	Time(ss)	Goal
1	0,29	0,27	0,0741	9,45	0,15	0
2	0,30	0,29	0,0345	9,45	0,15	1
3	0,37	0,29	0,2759	9,43	0,17	0
4	0,30	0,29	0,0345	9,43	0,17	1
5	0,35	0,29	0,2069	9,42	0,18	1
6	0,35	0,29	0,2069	9,42	0,18	0
7	0,41	0,29	0,4138	9,41	0,19	1
8	0,41	0,29	0,4138	9,41	0,19	1
9	0,53	0,31	0,7097	9,41	0,19	1
10	0,50	0,31	0,6129	9,40	0,20	0
11	0,35	0,29	0,2069	9,40	0,20	1
12	0,47	0,29	0,6207	9,39	0,21	0
13	0,55	0,29	0,8966	9,38	0,22	1
14	0,54	0,29	0,8621	9,37	0,23	1
15	0,51	0,29	0,7586	9,37	0,23	1
16	0,44	0,30	0,4667	9,37	0,23	1
17	0,55	0,29	0,8966	9,36	0,24	1
18	0,42	0,29	0,4483	9,36	0,24	0
19	0,35	0,29	0,2069	9,35	0,25	1
20	0,61	0,30	1,0333	9,35	0,25	1
21	0,70	0,27	1,5926	9,34	0,26	1
22	0,78	0,26	2,0000	9,32	0,28	1
23	0,97	0,26	2,7308	9,32	0,28	1
24	1,00	0,26	2,8462	9,26	0,34	1
25	1,00	0,25	3,0000	9,25	0,35	1
Min	0,29	0,25	0,0345	9,25	0,15	0
Max	1	0,31	3	9,45	0,35	1
Mean	0,5220	0,2856	0,8620	9,3768	0,2232	0,76
Median	0,47	0,29	0,6129	9,38	0,22	1
Standard Deviation	0,0784	0,0108	0,2240	0,0162	0,0162	0,4899

Table 4.2. Old implementation data table

4.4 Analysis of Results

The analysis of results will proceed as follows: we will begin by examining the data for one implementation and then for the other, column by column. Finally, a comparison will be made between the results obtained from the two implementations.

Let's start by analyzing the data for the old implementation shown in Table 4.2.

– **Y_max:** The distribution indicates significant variability in the maxi-

mum speed reached during the simulations, with a standard deviation of 7.84%. The mean and median are approximately midway between the minimum and maximum values, suggesting a balanced distribution.

- **Y_out:** The distribution shows less variability compared to Y_Max with a relatively low standard deviation of 1.08%. Most simulations seem to converge to a similar output value.
- **Overshooting:** The overshooting exhibits a variety of behaviors during the simulations. The presence of a very high maximum value and the difference between the mean and median suggest that some simulations may be less stable and significantly influence the overall variability.
- **Approach Time:** The total time elapsed ranges from 0.15 to 0.35 seconds, with the mean and median being about midway and matching, indicating a balanced distribution.
- **Goal:** The success rate is 76% on average.

Regarding the data for the new implementation shown in Table 4.3:

- **Y_Max:** The data indicate considerable variability, with values ranging from 0.45 to 1. The standard deviation suggests significant dispersion around the mean values, with some simulations reaching very high speeds.
- **Y_out:** The variability is more contained compared to Y_Max, indicating greater stability in the steady-state output values and a more precise convergence value.
- **Overshooting:** The overshooting shows considerable variability, with some extremely high values. The high standard deviation indicates a broader distribution, with some simulations exhibiting significantly higher overshooting than the average.
- **Approach Time:** The total time elapsed ranges from 0.15 to 0.30 seconds, indicating good consistency in approach times.
- **Goal:** The success rate shows that, on average, 84% of the simulations resulted in a goal.

For both implementations, the high standard deviation is due to the fact that the function call does not always occur under the same conditions. This primarily depends on how the ball is kicked during the `GoToBall&Dribble` state and consequently on the ball's position and speed when transitioning to the `GoToBall&Kick` state.

We observe that both implementations converge to a well-defined steady-state speed, both with a very low standard deviation on the order of 1%. This implies good stability in the system's performance. The limited variation in the steady-state speed suggests consistency and reliability in the system's responses. In terms of direct comparison, we observe that the average convergence value in the new implementation (0.4352) is higher than that of the old implementation (0.2856), which, as we will see, is reflected in the overshooting

statistics and consequently in the approach times to the ball.

It is in the comparison of the overshooting where we can best appreciate the differences and improvements made by my solution. The maximum overshooting achieved by the new solution (1.5) is half of the maximum reached by the previous one (3), and the minimum value (0.0227) is also less than the minimum value of the old one (0.0345). Both measures suffer from a large standard deviation due to the contribution of Y_max, which introduces significant deviation for the reasons discussed above. What most highlights the superiority of my implementation is the analysis of the average overshooting: my solution is indeed slightly more than half (specifically 56%) of the average value of the old implementation. This indicates a better capability of the system to control and respond more precisely and quickly to variations in operational conditions. More specifically:

- A lower overshooting indicates greater precision in controlling the system variables. The system reaches and maintains the target with less "overshoot" compared to the previous configuration.
- A lower overshooting suggests a faster response to disturbances. The system adapts more quickly and stabilizes its output without excessive oscillations.
- Reducing overshooting contributes to system stability. More precise control reduces the risk of instability and undesirable behaviors, improving the overall robustness of the system.
- The system reaches the desired target more directly with less resource waste due to potential trajectory corrections.
- The time required to reach stability after a disturbance can be reduced with lower overshooting, contributing to shorter stabilization times and improving system reactivity.

It is also important to highlight a certain consistency and a small gap between the mean (0.4975) and median (0.4318) of my implementation compared to the difference between the mean (0.8620) and median (0.6129) of the old implementation.

Reflecting the significant improvement in overshooting, we find similar improvements in approach times. Although the minimum value for both implementations is the same (0.15), the maximum value in the new implementation is lower by 5 seconds. The average and median values of each implementation are similar, with a difference of about three seconds in favor of my implementation.

The following are two graphs **??** and **??** where the x-axis represents the approach time intervals, ranging from 0.15 to 0.40 seconds. The bars on the graph will show the number of simulations falling within each approach time interval, with varying heights depending on the distribution of simulations in those intervals.

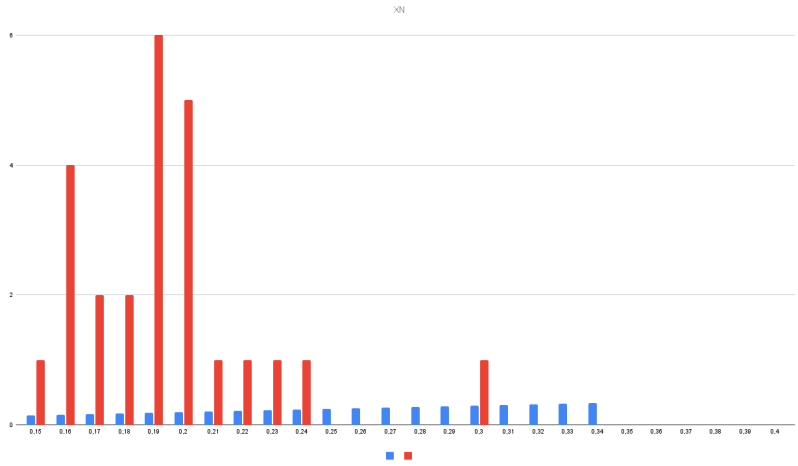


Figure 4.3. Distribution of the new implementation

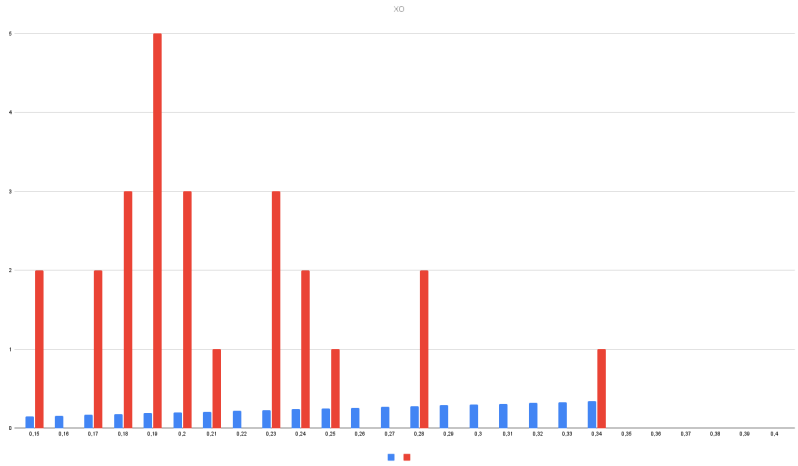


Figure 4.4. distribution of the old implementation

4.5 Analysis of Graphs

Now let's analyze two particularly significant samples of the approach speed graphs, one from the old implementation and one from the new implementation.

As highlighted in Figures 4.5 and 4.6, both functions predominantly exhibit a decreasing trend, with a "sawtooth" pattern more pronounced in Figure 4.5 compared to Figure 4.6. This characteristic indicates the influence of the proportional and derivative components of the controller. In the graph of the new implementation, the "sawtooth" effect is more subdued due to the contribution of the integral component, which helps to reduce oscillations and improve the overall stability of the system.

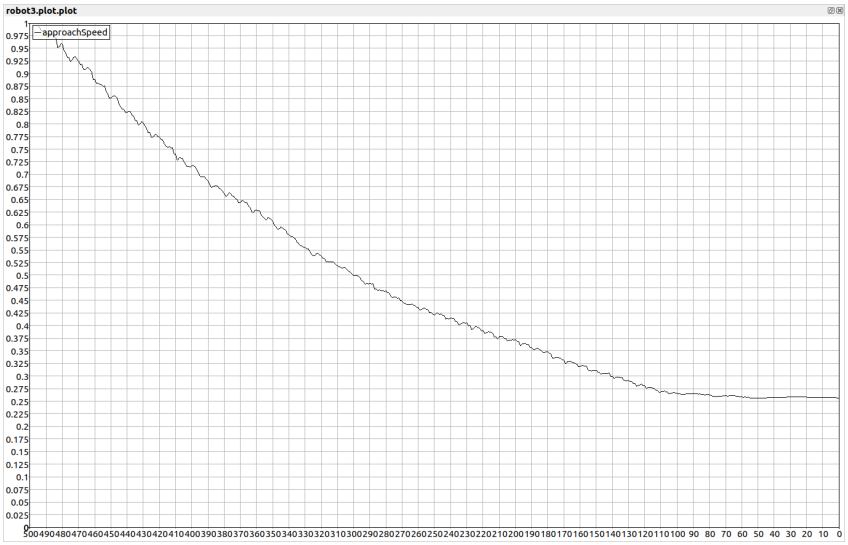


Figure 4.5. ApproachSpeed graph of the old implementation

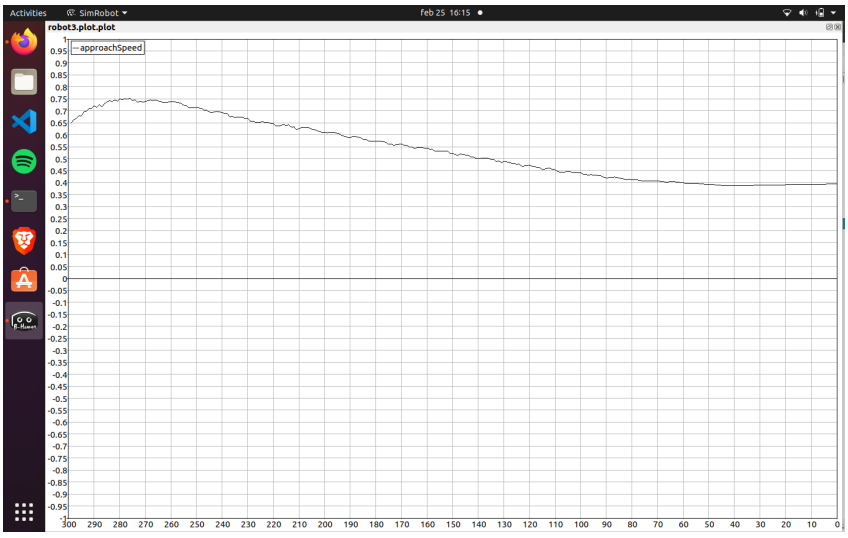


Figure 4.6. ApproachSpeed graph of the new implementation

Chapter 5

Conclusions

In conclusion, the RoboCup field and the development of robotic controllers represent a continuously evolving area of research. Despite current advancements, we are still in search of new opportunities and innovations to further enhance the performance and efficiency of these systems.

From the data analysis, it is clear that the new implementation generally outperforms the previous one. The average overshoot is lower, the steady-state output is higher on average, and it is achieved in less time. However, it is important to consider some additional aspects.

My solution is more complex: the controller in the old implementation lacks an integral component, effectively transforming the controller from PID to PD. Furthermore, my implementation features dynamic delta time calculation, unlike the old implementation which uses a static delta time. Both of these elements could pose a problem in an embedded system like the NAO robot. Embedded systems often operate with limited hardware resources such as memory and computational power. These limitations can affect the performance and complexity of the operations managed by the system. Therefore, it is essential to carefully evaluate the impact of these considerations on the practicality and effectiveness of the implementation.

However, to ensure a more comprehensive evaluation of the performance of my implementation, further simulations both in the simulator and on the robot are necessary to gain additional confirmations regarding the findings from the data analysis.

Possible further developments of the work could include the use of a more extensive grid search to explore the parameters and make the tuning more accurate. Different optimization techniques could also be tested, such as the Ziegler-Nichols method [5], a classic technique for obtaining an initial estimate of PID controller parameters. This method is based on observing the step response of a dynamic system and involves determining three key parameters: the proportional band, integral period, and derivative time.

An additional step could involve comparing the current controllers with those using different architectures, such as Model Predictive Controllers (MPC) or

controllers based on Fuzzy Logic. Fuzzy logic controllers are based on fuzzy systems theory, an approach that manages uncertainty and vagueness in input data. This type of controller uses fuzzy sets, linguistic rules, and fuzzy inference to translate imprecise inputs into defined control actions. Fuzzy sets represent concepts like "small," "medium," and "large" through membership functions, while fuzzy rules link input sets to output sets. Fuzzy inference combines these rules, producing a fuzzy output that is then defuzzified to obtain the control value.

On the other hand, the Model Predictive Control (MPC) controller adopts a more advanced approach based on predicting the future behavior of the system. With a mathematical model of the system expressed through differential or difference equations, MPC operates in a predictive manner. Initially, the model is discretized, and a prediction horizon is defined. The controller then calculates a sequence of optimal control actions, minimizing a cost function that considers constraints on state and control variables. Only the first control action is implemented, and this iterative process repeats at each step, allowing the controller to adapt to the system dynamics over time.

Bibliography

[1] Simrobot - b-human, 2022. https://wiki.b-human.de/master/simrobot/.

[2] Spqr team | official website of spqr team spl, 2022. https://spqr.diag.uniroma1.it/.

[3] cppreference.com, 2023. https://en.cppreference.com/w/cpp/chrono/time_point.

[4] A. K. Mackworth. On seeing robots. In *Computer Vision: Systems, Theory and Applications*, pages 1–13. World Scientific, 1993.

[5] P. Shahverdi, M. J. Ansari, and M. T. Masouleh. Balance strategy for human imitation by a nao humanoid robot. In *2017 5th RSI International Conference on Robotics and Mechatronics (ICRoM)*, pages 138–143. IEEE, 2017.

[6] S. Shamsuddin, L. I. Ismail, H. Yussof, N. Ismarrubie Zahari, S. Bahari, H. Hashim, and A. Jaffar. Humanoid robot nao: Review of control and motion exploration. In *2011 IEEE International Conference on Control System, Computing and Engineering*, pages 511–516, 2011.