University of Amsterdam
Faculty of Science
The Netherlands

Dutch Nao Team

Technical Report 2023

Students:
Gijs de Jong
Harold Ruiter
Derck W.E. Prinzhorn
Jakob Kaiser
Mark Honkoop
Joost Weerheim
Fyor Klein Gunnewiek
Ross Geurts
Dario Xavier Catarrinho
Stephan Visser
David Werkhoven
Rick van der Veen
Madelon Bernardy

Supervisor:
Arnoud Visser

December 31, 2023

Abstract

In this Technical Report, the Dutch Nao Team lists its progress and activities in the past academic year with the previous report [1] from 2022 as a starting point. Besides new developments, this report also lists older developments when relevant.

In the previous report, the decision to shift to a new framework was discussed. Initially, the Hulk's framework was used during the transition to the Rust programming language. However, this year, our focus has shifted to implementing a new framework called Yggdrasil. This framework emphasises continuity, extensibility, and developer experience. Our goal is to provide the Standard Platform League with a sleek, modern framework that has a lower entry barrier than existing ones. This report outlines the core features developed for this new framework.

This year the team has grown by a significant amount which enables the development of the new framework. This expansion also led to adjustments in the team structure, with five different groups formed to work on various projects. Two groups concentrated on artificial intelligence research, while three others focused on implementing features for the new framework.

Throughout the year, the Dutch Nao Team participated in various events. In addition to organising multiple workshops, the team took part in two competitions, including the GORE 2023 in April and the RoboCup 2023 in Bordeaux in July, where we secured a commendable fourth place in the challenge shield and first place in the data minimisation challenge.

Contents

1 Introduction

The Dutch Nao Team consists of 20 Artificial Intelligence (AI) students and 6 Computer Science (CS) students; 13 master students and 13 bachelor students, who are supported by a senior staff member, Arnoud Visser. The team was founded in 2010 and competes in the RoboCup Standard Platform League (SPL). This is a robot football league, in which all teams compete with identical NAO robots to play football autonomously (see Figure 1). The RoboCup is an international scientific initiative with the official goal that by the middle of the 21st century, a team of fully autonomous humanoid robot football players shall win a football game, complying with the official rules of FIFA, against the winner of the most recent World Cup [2].

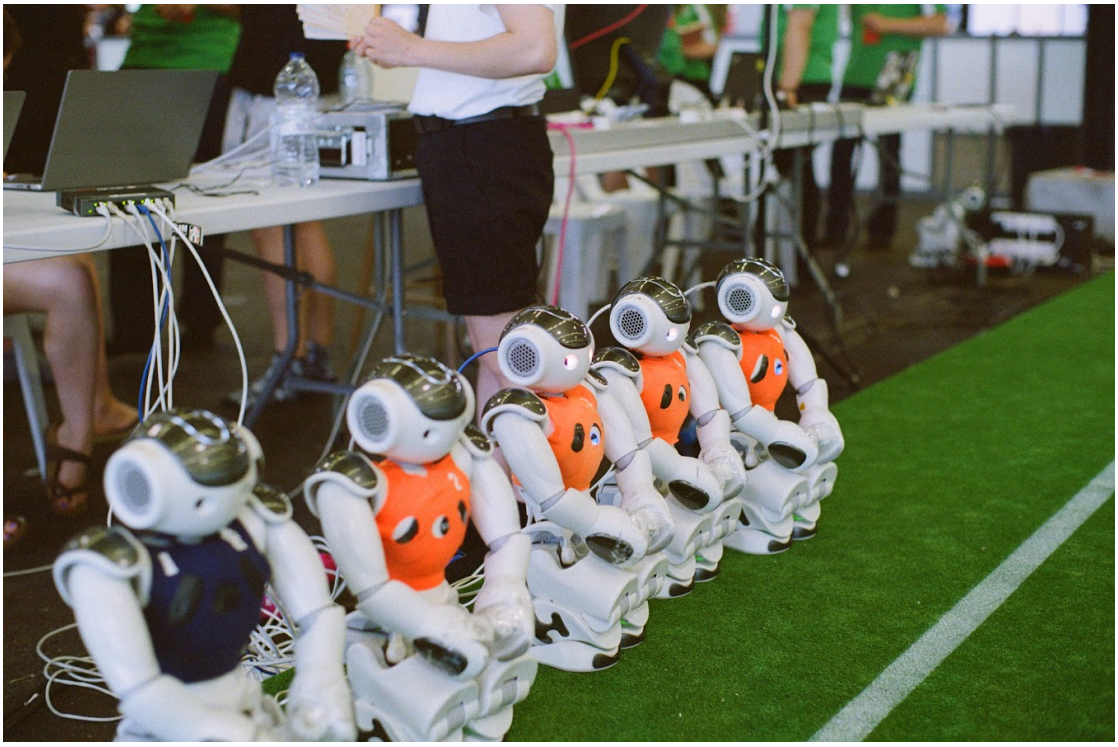

Figure 1: A team of NAO robots standing at the sideline of the football field

2 Hardware

2.1 The NAO Robot

The NAO robot, a programmable humanoid robot designed by Softbank Robotics (formerly Aldebaran Robotics), has undergone significant evolution over time. Before 2018, all versions from 4.x onwards shared the same computational hardware, differing only in their sensors or actuators. However, the introduction of the sixth version (V6) marked a significant upgrade in both hardware and proprietary software. Despite the V6's improved hardware, limitations in CPU resources continue to be a challenge, particularly for tasks requiring intensive computation, such as detailed pixel-wise image processing and deep neural network approaches. The NAO features two high-definition (HD) cameras and an inertial board, critical to use in the competition. The HD cameras are located in the head of the NAO; One camera is positioned for distant object detection, while the other is angled downwards for nearby objects. The inertial board is essential for detecting falls, a common occurrence in competitions. The robot also includes four sonars, an infrared emitter and receiver, pressure sensors, and tactile sensors. Among these, pressure sensors are vital for the walking engine, relying heavily on foot pressure data for stable movement. The other sensors are prone to breaking down and therefore are used less frequently due to their higher failure rate and less reliable measurements.

The NAO robot is designed with 25 degrees of freedom in its joints, enabling stable bipedal movement and various kicking techniques. The arms are primarily used for assisting the robot in standing up after a fall and for maintaining balance while walking. Although it is allowed for the goalie to hold the ball, this function is seldom utilised by teams. Each robot, despite being standardised, shows unique characteristics in performance, with older robots displaying less stable and fluent movements due to joint wear. To ensure consistent performance, individual joint calibration is necessary. Additionally, due to potential misalignment of the cameras within their enclosure, calibration is also required to adjust for any offsets relative to the robot's centre.

The V6 version operates on a 64-bit system and is equipped with a 1.91 GHz Intel Atom E3845 CPU, a quad-core processor with one thread per core. This version boasts improvements including 4 GB DDR3 RAM and 32 GB SSD storage, surpassing its predecessors. Enhancements in WiFi connectivity and more impact-resistant fingertips are also notable. With these hardware upgrades, new possibilities emerge for addressing challenges in localisation and ball detection.

2.2 Low Level Abstraction

Control of the robot is managed through Aldebaran's proprietary Low-Level Abstraction (LoLA) software. This software, which runs as a service on the robot, offers a UNIX socket interface that is used to effectively control the robot's motors and receive sensor data.

3 Yggdrasil: Framework Design

After moving our lab to a prominent new location within the university, there has been a large influx of new members. Combined with the departure of some existing members, we decided as a team to focus our efforts on creating an environment where knowledge can be preserved over multiple generations of the team. With this new focus and a larger workforce than ever before in the history of the team, we felt that our old framework left a lot to desire and it was the right time to move on to new developments.

This year, we have started development of our new framework called Yggdrasil. Built from the ground up with a focus on continuity, extensibility and developer experience, our goal is to provide the SPL with a lean, modern framework that has a lower barrier to entry than the existing frameworks. Yggdrasil is fully written in the Rust programming language, which compared to C++ provides more type safety, has friendlier compiler messages, fewer footguns and a batteries-included approach to dependencies with its cargo package manager. These, among other things, help empower both experienced and less experienced students to get productive in Yggdrasil faster and with more confidence. Furthermore, we put large efforts into minimising the use of code generation and implicit behaviours, making sure that public APIs and complicated logic are documented and motivated well, and keeping code quality high with regular code reviews and pair programming sessions.

3.1 Tyr: Dependency Injection and Parallel Execution

To keep our codebase manageable as the amount of features grows, it is important that we can write new features incrementally and in a modular fashion. At the same time, many features are dependent on one another and there are a lot of state which needs to be used by different features. To solve this problem, we developed Tyr, a dependency injection library which manages the execution of robot subsystems and the state between them.

```rust
use miette::Result;
use tyr::prelude::*;

#[derive(Default, Debug)]
struct Number(u32);

#[system]
fn print_number(number: &Number) -> Result<()> {
    println!("{number:?}");
    Ok(())
}

#[system]
fn update_number(number: &mut Number) -> Result<()> {
    number.0 += 1;
    Ok(())
}

fn main() -> Result<()> {
    App::new()
        .init_resource::<Number>()?
        .add_system(print_number)
        .add_system(update_number)
        .run()
}
```

Figure 2: A simple example that prints and updates a number using tyr.

In Tyr, everything is managed within the context of an *app*. The app allows you to define state through *resources* and the way it is used through *systems*. Resources allow you to store a single global instance of some data type, which can be used within an app. Finally, systems are simply functions which take some number of resources by (mutable) reference as arguments. Because of the strict lifetime rules in Rust, we mark systems with a macro annotation `#[system]`, that performs a simple type substitution from `&T/&mut T` to our internal `Res<T>/ResMut<T>` types. For the developer, the arguments behave just like using references normally.

In Yggdrasil, each system runs once every LoLA cycle. The order of systems is non-deterministic by default, but can be made deterministic by declaring it using the `.before()/.after()` methods.

```
let app = App::new().add_system(foo.before(bar).after(baz));
```

Figure 3: An example of system ordering. The system *foo* always runs before *bar* and after *baz*.

Using the information we get from adding systems to the app, such as parameter mutability and system orderings, we can create a directed acyclic graph for the execution schedule. In our initial implementation, systems are still executed sequentially on the main thread. However, with the schedule, we can run many systems in parallel in future updates to the scheduler.

Tyr is also designed to be very modular and extensible. To better structure an application, Tyr has a module system that allows the developer to put related resources and systems together in a *module*.

```
// src/nao.rs
pub struct NaoModule;

impl Module for NaoModule {
    fn initialize(self, app: App) -> Result<App> {
        Ok(app
            .add_startup_system(initialize_nao)?
            .init_resource::<LolaMessage>()?
            .add_system(write_to_lola))
    }
}

// src/main.rs
use nao::NaoModule;

let app = App::new()
    .add_module(NaoModule)?
    .run();
```

Figure 4: An example of a tyr module. We avoid cluttering our main.rs by having all NAO related logic in a separate module.

One extension we built using these primitives is the *task* module. Tasks allow us to run certain functions over multiple cycles by having them offloaded to a background thread. This is useful for applications such as compute-heavy ML, large IO operations, or networking, as we don't want these to block our LoLA communication.

4 Nidhogg: Abstraction Layer

To enable direct robot control in Yggdrasil, we developed Nidhogg, a layer built on top of LoLA (as detailed in Section 2.2). Building this as an abstraction layer for LoLA means the library has full control over the data structures used for robot data. Therefore the data structures in Nidhogg are designed to be intuitive to use and change, making it simpler for developers to create and refine robot behaviours without unnecessary hurdles.

Another benefit of building Nidhogg as an abstraction layer for LoLa is that it enables the creation of different back-ends for Nidhogg. This makes it possible for Yggdrasil to effortlessly work on different platforms, including simulations. The ability to run Yggdrasil in simulations is significant and makes applying various machine-learning techniques possible, while also ensuring tests can happen in a safe environment without risking real hardware.

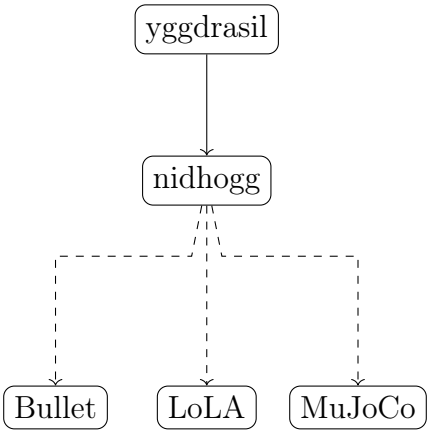

Figure 5: The nidhogg abstraction layer

```
fn main() -> Result<()> {
    let mut backend = LolaBackend::connect()?;
    let mut backend = MujocoBackend::connect()?;
}
```

Figure 6: Switching back-end is simple, just change the type and it works!

5 Heimdall: Camera Library

Heimdall, our in-progress camera library, is engineered to streamline interactions with the NAO robot's cameras, maintaining a balance between user-friendliness

and performance efficiency. It employs the Video4Linux (V4L) API for communication with the NAO's imaging hardware. The images captured by the NAO are in the YUYV format. Recent explorations have involved the potential conversion of these YUYV images to RGB format, aiming to broaden the applicability of various line- and object-detection algorithms. However, the current conversion algorithm on the NAO requires approximately 21 milliseconds per image, a duration deemed excessively slow for practical applications. Consequently, we have opted to continue utilising YUYV images, while actively investigating strategies to enhance the efficiency of the YUYV to RGB conversion process.

Future development plans for Heimdall include the integration of advanced features such as image segmentation, further expanding its utility in robotic vision applications.

6 Bifrost: Networking

6.1 Introduction

In the SPL, networking is important for the game controller, robot-to-robot communication and efficient debugging tools. At the same time, the new data minimisation challenge added the requirement to minimise the amount of data shared between robots. The data minimisation challenge made it important to have detailed control over the encoding process of packets.

Our solution to these challenges is Bifrost, a library designed to make it incredibly simple to create new packets while giving us consistent and precise control over the encoding process of these packets.

Using Rust derive macros developers can create a new packet without implementing any of the encoding or decoding logic themselves (Figure 7).

```rust
#[derive(Encode, Decode)]
struct Foo {
    bar: i32,
    baz: Vec<Foo>
}
```

Figure 7: Implementing a new packet using the `Encode` and `Decode` macros

6.2 Technical Details

Bifrost's encoding and decoding logic are based on Google's protocol buffers, where each packet needs to implement the `Encode` trait. This can be done automatically using the `Encode` derive macro, which requires each field of the packet to also implement `Encode`. This approach allows the macro to utilise the specific encoding implementation for each field, efficiently condensing the packet into a compact byte array. This allows us to reuse existing implementations for various data types automatically while leaving open the possibility of creating an optimised `Encode` implementation for a specific packet.

This design enables targeted optimisations for every possible data type within a packet. A prime example of such optimisation is the variable-length encoding for integers. This feature allows Bifrost to efficiently encode integers, using only the number of bits necessary for their representation.

7 Sindri: Build Tool

Sindri, the Yggdrasil Command Line Interface (CLI), is a development tool designed to enhance the efficiency of the development process. It is engineered to enable developers to perform critical tasks using just a single command. For instance, compiling the Yggdrasil binary can be achieved through the command `[./sindri build]` and subsequently deploy `[./sindri deploy]` it, along with any requisite assets, directly to the designated robot(s).

Moreover, Sindri is equipped with a network scanning capability `[./sindri scan]`, which effectively identifies online robots within a predetermined IPv4 address range. This feature significantly contributes to optimising the workflow. Looking ahead, we are planning to further improve Sindri with additional functionalities focused on efficient deployments and extended debugging capabilities.

8 Behaviour Engine

The behaviour engine in the Yggdrasil framework plays a crucial role in monitoring and managing the robot's actions. This system is what is referred to as the behaviour engine. This section outlines the design and implementation of the engine, which primarily ensures the timely execution of the correct behaviour in each cycle. The discussion begins with the engine's design and then moves to the practical aspects of behaviour execution.

To understand this system, some definitions are necessary. A *motion* is defined as

a series of joint angles that collectively produce a movement, such as a ball kick or standing up. A *behaviour* is a combination of motions, triggered by specific conditions. For example, the 'walk to ball' behaviour would involve locating the ball and walking towards it. The behaviour engine, therefore, tracks the state and information needed to execute and switch between these behaviours. Essential elements in this context include the *game phase* (like 'penalty shootout' or 'playing'), the robot's *primary state* (as per RoboCup rules, such as 'ready', 'playing', 'penalized'), and the *role* of the robot, which dictates a set of behaviours (e.g., 'keeper', 'striker').

8.1 Design

8.1.1 Execution versus Transition

The behaviour engine is designed as a state machine and separates the logic for transitioning between behaviours from their execution. When executing a behaviour, a corresponding function is called which is provided a context, containing all the necessary information for executing that behaviour. Each execution function then mutates a *control message* detailing joint angles and extra information like LED colors on the robot. An example of the implementation of a behaviour is given in Figure 8. Transitions between different behaviours are defined according to roles, offering a clear overview of the transitions specific to each role.

```
struct Dance;

impl Behavior for Dance {
    fn execute(
        &mut self,
        context: Context,
        control_message: &mut NaoControlMessage,
    ) {
        // Logic to perform the dance behavior
    }
}
```

Figure 8: Code snippet showing the implementation of a behaviour

8.1.2 States

The necessary state information is structured hierarchically. Different game phases may have various roles, such as 'keeper' and 'striker' during a penalty shootout. Each role's behaviour is influenced by the primary state and other contextual

data. For instance, a 'striker' might dribble in its half and shoot when near the opponent's goal. Behaviours can be shared over various roles as can be seen in Figure 9. Additionally, during execution, each behaviour can keep track of its own state that can be used to store any information needed for that behaviour.

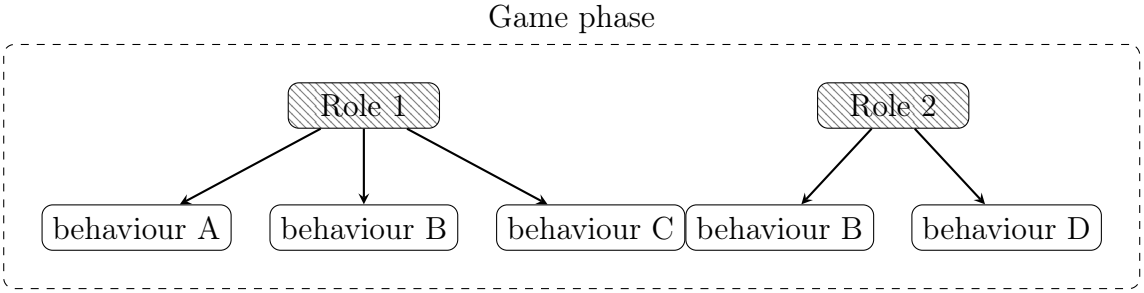

Figure 9: Schematic overview of states in the behaviour engine for a game game

8.2 The Flow of Execution

This section briefly describes the operational cycle of the behaviour engine.

First, a context is created that contains all the necessary information needed for transitions and the execution of behaviours. Then, the transitions between behaviours, if needed, are handled depending on the role of the robot. Finally, the engine calls the execution function for the current behaviour, which alters the control message according to its data and the given context. The final outcome of each behaviour is the control message, encompassing robot control data such as target joint angles.

8.3 Alternatives: Behaviours as Systems

Currently, execution and transition functions are given a broad range of data, which might be more than needed for specific behaviours or transitions, leading to unnecessary data dependencies. This means during execution, these functions will often be passed information that is needed by another function, but that is not necessary for this specific behaviour. An alternative would be treating behaviours as systems with tailored data inputs. However, this approach would require running all execution functions each cycle. The current method offers a more streamlined development experience and ensures only the relevant function is executed per cycle.

9 Mimir: Debug Panel

9.1 Introduction

Mimir is the debug panel we are building for testing and debugging new robotics features in a visually intuitive manner. It is built as a web panel using the dioxus framework[1], which allowed us to minimise code duplication.

9.2 Debug Channel

Mimir works by establishing a web socket connection directly to the target robot. This socket channel is then used to receive data from the robot and makes use of Bifrost (see Section 6) for efficiently encoding data.

9.3 Panels Overview

Mimir is composed of multiple specialised panels, each crafted for distinct functions. Users have the flexibility to rearrange these panels according to their needs, and the system supports saving specific configurations for reuse. This feature is particularly beneficial for typical workflows, such as calibration or data collection tasks.

Currently, Mimir is in its developmental phase, and the following panels are being actively developed:

9.3.1 Walking Debugger Panel

A key panel in Mimir, the Walking Debugger, is dedicated to graphically representing various critical properties of the walking engine (as outlined in Section 11). Its primary purpose is to visually plot the walking engine's components, offering a comprehensive insight into its operations. This visual aid is invaluable for developers, simplifying the process of debugging and fine-tuning the walking engine.

9.3.2 Vision Debugger Panel

The Vision Debugger panel is tailored to display feeds from the NAO robot's dual cameras. Enhancing the debugging experience, it overlays the output of different vision algorithms over the camera feeds, aiding in the efficient recognition and analysis of visual elements like lines. One of the panel's standout features is its

[1] `https://github.com/DioxusLabs/dioxus`

ability to run most vision algorithms directly in the browser using WebAssembly. This functionality is particularly useful as it allows developers to work with the vision panel even without access to a physical robot, making it a vital tool for vision-centric debugging and development.

10 Damage prevention

Damage prevention is the task of actively trying to decrease or prevent damage taken by the robots during matches. While our goal is to prevent the robot from falling altogether, falling cannot always be avoided. External elements like other robots pushing each other over combined with the overall difficulty of achieving a stable walk are elements that we simply cannot fully avoid. This is where damage prevention comes into play to minimise the damage that would occur when a robot falls.

Robots are not only expensive but also rather delicate. By reducing or preventing damage our robots can perform better and will be able to perform for longer. If a robot makes a fall, this can result in the robot turning off, or worse, the robot being reset during a match which would mean a considerable amount of a match has to be played with one robot down. Preventing damage is not only financially desirable, but this will also improve the overall performance of the robots as time goes by.

To prevent damage to the robots two mechanisms were developed; A fall catch and a getup motion.

10.1 Fall Catch

The fall catch is the act of catching yourself when you fall. This will reduce the impact on the ground and therefore reduce damage taken with falls. The Fall catch is achieved in a couple of steps.

The first step is detecting when and in what direction the robot is falling. This is done by monitoring the gyroscope, and accelerometer and setting certain thresholds that when crossed, indicate that the robot is falling.

The second step is to move the robot into a position that will minimise the impact of the fall on delicate parts of the robot. This is done by bending the knees of the robot to drop its centre of gravity, moving its head forwards or backwards depending on the direction of the fall and using its arms to dampen the blow. An additional feature of the catch is that we can preemptively move the robot in a position that will ease the getup motion.

10.2 Getup Motion

The getup motion does not necessarily actively prevent damage, but a faulty getup motion can cause a lot of unnecessary damage. Therefore making sure that the getup motion is a steady and safe motion will prevent a lot of damage. The procedure used for standing up, goes as follows:

First and foremost, the robot needs to know whether it is lying down or not. To do this we keep track of the gyroscope and store a set amount of previous accelerometer values in memory. Using this, we see whether the robot is in a somewhat horizontal position and check whether the standard deviation of the previous accelerometer values exceeds a certain threshold. If this threshold has been crossed, the robot will know that it is lying down.

Next we issue a command for the robot to stand up. This motion gets executed by moving the robot's joints to steady key positions that make sure that every intermediate motion has a safe stopping position.

Finally, the robot will validate whether the getup motion has succeeded. This is done simply by reading the pressure values of the soles of the robot. If the getup motion has ended and the robot is supporting its weight with its feet, we can confidently say that the getup motion has succeeded.

11 Walking Engine

Yggdrasil has been built from scratch, leading us to develop a new walking engine based on the 2014 rUNSWift gait generator [3], with several tweaks and improvements. One key update is in the inverse kinematics. Instead of the usual Jacobian method for estimating transformations, a closed-form implementation is adopted, that results in exact joint angles based on an implementation by the HULKs [4].

Additionally, the gait generator's architecture has also been improved. It now uses a Finite State Machine (FSM) to smoothly switch between different stages of the walking cycle. This modification facilitates a more straightforward process for modifying and enhancing the walking engine's behaviour, as the code is now structured in a succinct and deterministic manner.

Our initial walking engine is operational, demonstrating the ability to move in various directions. The ongoing development is directed towards improving the stability and speed of the walking mechanism. Simultaneously, efforts are underway to explore gait modulation within the gait generator, aiming to achieve a more stable and dependable walk.

12 Automatic Calibration

The concept of automatic calibration was introduced to us while dealing with camera calibration during GORE and the RoboCup of 2023. Due to the different hours of the day and different locations of the fields, there were a lot of issues with the sight of the robot. Calibrating the camera to deal with the different lighting circumstances solved these issues. But having to do this by hand before every single match that was played was a time-consuming job.

12.1 Approach

To automate the calibration of the camera some kind of ground truth is needed. This ground truth serves as a benchmark for calibrating the camera parameters. The idea is to understand the ideal appearance of a specific object under perfect conditions. By manipulating the camera parameters, one can then align the object's visual representation with the expected ideal appearance. The calibration is considered complete once the object reflects this ideal appearance.

The lines of the field are taken as this ground truth because every field has similar field lines. This enables the robots to automatically calibrate the camera on every field and every location on the field without the need for support from extra props or humans.

To implement this concept, a precise line detection system was developed to serve as an effective ground truth. The results from this algorithm are then compared with line detection that is run on the robot. The camera's parameters are adjusted iteratively to enhance the alignment of the robot's line detection results with the algorithmic ground truth. This is done until the difference to the ground truth algorithm is minimised.

12.2 Method

There are a few challenges that come with implementing such a line detection system. The main obstacle was the preprocessing of the images. This is a hard task because a lot of noise appears in the background of the camera during the setup of a match. To address this issue, we adopted the following approach for image preprocessing.

The first step in preprocessing is the removal of the background that is not part of the field. To do this, any green-coloured areas within the images of a certain area size are detected. All the areas of a certain size are then combined if they are of a certain distance from each other. Together they create the area of the field.

Everything outside of this area is turned black. This results in an image that only contains the field as can be seen in Figure 13.

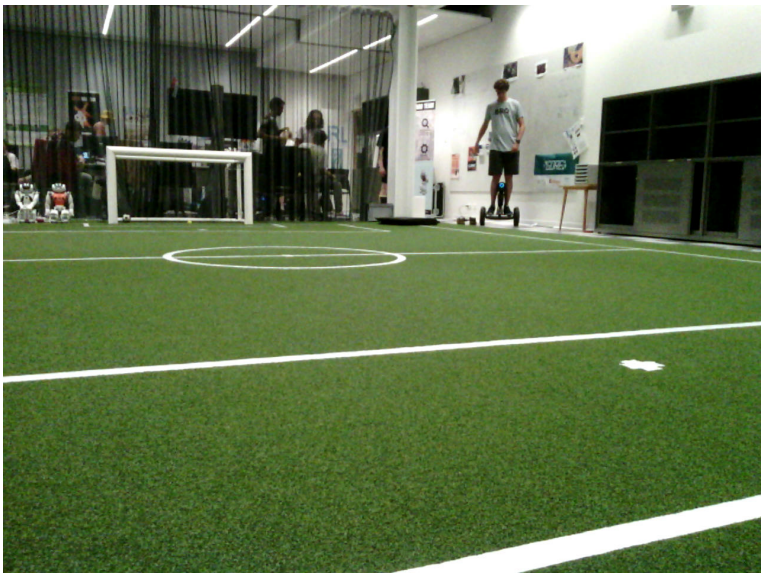

Figure 10: The original image

Figure 11: The detection of green areas within the image

Figure 12: The combined green areas

Figure 13: The image with no background

The next preprocessing step is to turn everything that was not a white line black as well. This then resulted in a fully black image showing the white lines of the field as can be seen in Figure 14. This final result can then be used in various ways to detect the lines within the image.

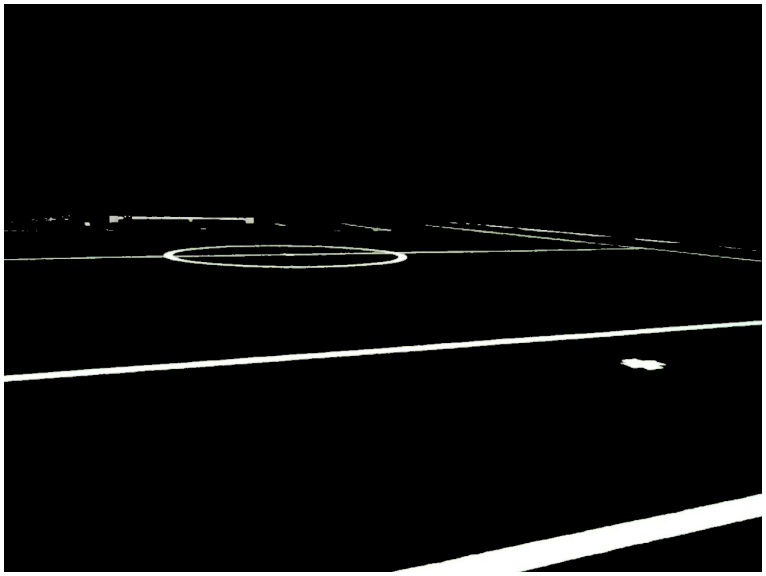

Figure 14: The black and white image

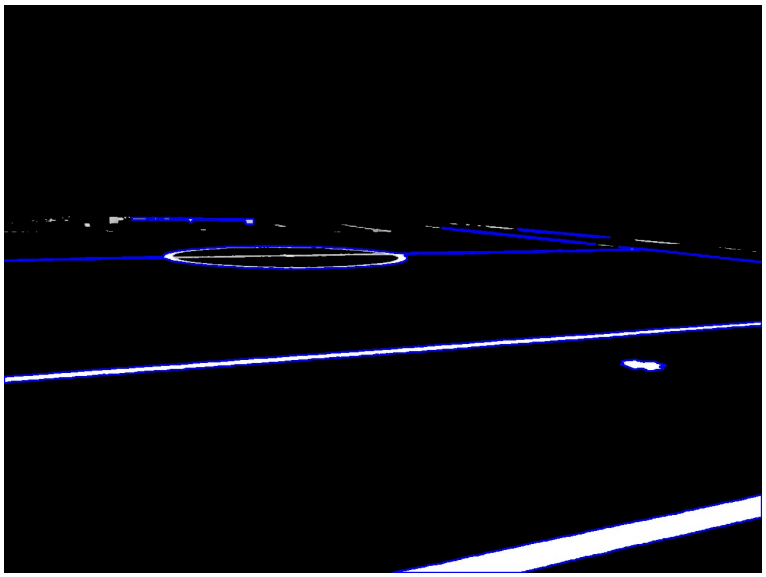

Figure 15: The image with detected lines

Automatic calibration is still in progress, but the next steps are to compare detected lines on the robot with the ground truth system. The line detection system used as a ground truth will be a slower model that is run on an external machine. The camera's parameters will then be dynamically altered until the best performance is achieved compared to the ground truth.

13 Results

13.1 GORE

The German Open Replacement Event took place from the 25th of April until the 30th of April and used a Swiss system tournament style. We attended with nine team members and results for the seeding rounds are shown in Table 1. After these initial rounds, we were ranked 5th and lost 0-3 versus the Nao Devils in the quarter-finals resulting in a final 6th place.

Opponent	Score
NomadZ	4-1
HULKs	0-5
R-ZWEI KICKERS	2-1
B-Human	0-10
HTWK Robots	0-7

Table 1: Results for the GORE.
Scores are notated as Dutch Nao Team - Opponent

13.2 RoboCup

This year the RoboCup took place in Bordeaux, France from the 4th of July to the 10th of July, and was attended by 18 member of the Dutch Nao Team. The competition in the Standard Platform League was split into two categories, the Champions Cup and the Challenge Shield. The Dutch Nao Team competed in the Challenge Shield and qualified for the RoboCup with a video [5] and qualification paper [6]. Results for the seeding rounds can be seen in Table 2.

Opponent	Score
UT Austin villa	7-0
R-ZWEI KICKERS	2-3
BadgerBots	1-2
RoboEireann	1-6
Naova	6-0
Rinobot-Jaguar	8-0
RedBackBots	2-1

Table 2: Results for the RoboCup seeding rounds.
Scores are notated as 'Dutch Nao Team - Opponent'

This qualified us for the semifinals against RoboEireann which we lost 10-0, leaving the team to play a final game for third place. This game was played against BadgerBots, which was lost 4-0, resulting in fourth place in the RoboCup.

There were also three technical challenges, the first involving ball handling, the second on recognizing hand signs from a referee, and the third challenge was to minimise the amount of bytes sent between robots during a match. The Dutch Nao Team only participated in the third challenge, which we won with an average data usage per robot second of 3.3461 bytes. This gained the team 25 points, leaving us to rank 7th of the challenges overall.

13.3 RoHOW

The Robotics Hamburg Open Workshop event took place at the Hamburg University of Technology from the 1st to the 3rd of December. The Dutch Nao Team saw 8 of its members attending this event, which aimed to stimulate knowledge transfer between the different SPL Teams through workshops and discussions. Here, the members attended these talks on various topics related to robot football, while also connecting with several teams over common ideas.

13.4 Public Events

During 2023, the Dutch Nao Team held and participated in several public events, such as workshops and demonstrations. Any funds secured by participating in these events went towards decreasing the price of participation in the RoboCup. The following is a list of these events:

1. 15-02-2023: The Dutch Nao Team gave a robot football demonstration for a Turkish class of students associated with Erasmus+.
2. 09-03-2023: The Dutch Nao Team participated in the Career Day Alkmaar, where the team gave a workshop about robot football to high-school students.
3. 17-03-2023: Open Campus Day during the Bachelor's Week of the UvA. The Dutch Nao Team gave demonstrations of football games with the robots, while being open to questions from attendees about their studies and work within the team.
4. 4-06-2023: The Dutch Nao Team hosted a workshop about programming in python to primary school students, as part of an incentive from the IMC Weekendschool.
5. 23-06-2023: The Dutch Nao Team gave several demo games during the ChatGPT Teacher Congress hosted by AlfaGammaPartners. The congress aimed

to educate (high-school) teachers about ChatGPT, while the Dutch Nao Team provided a refreshing intermezzo.

6. 24-8-2023: As part of the UvA Zomerweek, the Dutch Nao Team hosted a workshop about programming in Scratch to primary-school age students.
7. 7-10-2023: The Dutch Nao Team participated in the open day of the Science Park during the Weekend van de Wetenschap, where the team hosted robot football demonstrations throughout the day.
8. 19-10-2023: The Dutch Nao Team hosted an Open Lab to garner new team members from interested AI and CS students. The team gave a talk on various topics related to robot football, and hosted a small demonstration.
9. 10-11-2023: Open Campus Day during the Bachelor's Week of the UvA. The Dutch Nao Team gave demonstrations of football games with the robots, while being open to questions from attendees about their studies and work within the team.
10. 22-11-2023: The Dutch Nao Team appeared on the Super Data Science Podcast, where team member Dário Catarrinho talked all about AI and robotics with host Jon Krohn.
11. 27-11-2023: The Dutch Nao Team hosted a presentation, demonstration and interesting quiz about robot football for a group of French students visiting the Faculty of Science.
12. 11-12-2023: RoboCup Sponsor Event in Eindhoven. The Dutch Nao Team had a stand at this event, where they gave talks to interested investors for the RoboCup.
13. 15-12-2023: The Dutch Nao Team gave a small demonstration and talk about their current progress on robot football to a visiting delegation of the faculty of Informatics of the AGH University of Cracow.

14 Plans for 2024

A foundation has been laid for the new framework Yggdrasil with most of the essential modules being implemented. There are various projects like the debug panel, automatic calibration and damage prevention that will be continued next year. However, next year's goal is to play a RoboCup match with the new framework, so the focus will be on the features needed to do that. Additionally, two AI teams will be doing research focused on future applications.

14.1 Projects

Several key features are necessary for playing a match, some of these still need to be developed or are currently under development:

14.1.1 Walking Engine

The implemented walking engine needs to be integrated into Yggdrasil and further improved for stability and speed. This is one of the most important features since much of the other functionality depends on it.

14.1.2 Localisation

Localisation is needed such that robots know where they are. Currently, a simple odometry-based mechanism has been implemented to estimate the robot's position based on the walking engine. However, this approach is inaccurate and fails if the robot falls over. Therefore it is important to include visual information like field lines, middle circle, corners, etc. in the estimation of the robot's position. This will make localisation more robust and accurate.

14.1.3 Ball Detection

When playing a match it is crucial to know where the ball is. A bit of research has already been done on this topic and we will probably take a similar approach as other teams have done, with a selective search mechanism to find candidates and a small neural net to detect the soccer balls.

14.1.4 Behaviours

Yggdrasil now has a behaviour engine however to play a match, basic behaviours like 'walk to ball', 'dribble', etc. will need to be implemented for the different roles. Besides implementing these behaviours in closed-form one team will look into using reinforcement learning to learn complex behaviours.

15 Contributions

The following list credits the people who worked on the contributions mentioned in this report, in alphabetical order:

- **Dario Xavier Catarrinho**, worked on damage prevention.
- **David Werkhoven**, worked on automatic calibration.

- **Derck Prinzhorn**, acted as project manager and was responsible for team vision, project planning and management and improvement of the team as a vice chair. Additionally, he worked on various AI features such as pose classification, robot and ball detection, whistle detection and jersey detection. Finally, he led the process of writing the tech report.
- **Fyor Klein Gunnewiek**, worked on line detection and Bifrost.
- **Gijs de Jong**, acted as technical lead and was responsible for technical vision and building the foundation of the new Yggdrasil framework. He was very involved in the architecture and API design of all parts of the framework. Additionally, he worked on various features, such as Bifrost, Nidhogg, Mimir, Sindri and the walking engine.
- **Harold Ruiter**, responsible for the architecture and development of the core Tyr ecosystem, asynchronous/parallel task functionality. Furthermore, he worked closely with Gijs on the architecture and API design for the entire framework, whilst also working on the Nidhogg abstraction layer for the Yggdrasil framework and Localisation.
- **Jakob Kaiser**, acted as team lead together with Ross, providing support to various teams. He was also responsible for the implementation of the motion module and the behaviour engine and reviewed the tech report.
- **Joost Weerheim**, led a software team focused on various upcoming features for the new features. Additionally, he worked on foundational systems in the Yggdrasil framework related to sensors, Sindri and Mimir.
- **Madelon Bernardy**, contributed to the implementation of the behaviour engine.
- **Mark Honkoop** worked on Bifrost, Heimdall and the custom image for Yggdrasil.
- **Rick van der Veen**, worked on automatic calibration.
- **Stephan Visser**, worked on the implementation of damage prevention.
- **Ross Geurts**, acted as team lead together with Jakob, providing support to various teams and contributing to the foundations of Yggdrasil.

16 Conclusion

This year the Dutch Nao Team has experienced a significant expansion in its team size. After successfully transitioning to using Rust, the team worked on laying the

foundations for a new improved framework, Yggdrasil. Progress has been made on several essential modules like Bifrost for networking, Heimdall as a camera library, a new build tool Sindri, a behaviour engine and the Mimir debug panel. Development of the new framework will continue with the goal of playing a match at RoboCup 2024 using Yggdrasil.

References

[1] L. Bolt, F. K. Gunnewiek, H. L. gezegd Deprez, L. van Iterson, and D. Prinzhorn, "Dutch nao team - technical report," Tech. Rep., Dec. 26, 2022.

[2] [Online]. Available: `http://www.robocup.org/objective/` (visited on 08/02/2023).

[3] B. Hengst, "Runswift walk2014 report robocup standard platform league," *School of Computer Science and Eng., Univ. of New South Wales*, 2014.

[4] A. Essig, P. Gleske, K. Nölle, M. Schmidt, H. Sieck, and B. Wetters, "Team research report - 2021," Tech. Rep., 2021.

[5] D. N. Team, *Dutch nao team qualification video 2023*, 2023. [Online]. Available: `https://www.youtube.com/watch?v=3EJG7k0iBfY`.

[6] W. Duivenvoorden, G. de Jong, H. Ruiter, *et al.*, "Team qualification document for robocup 2023," Tech. Rep., Feb. 13, 2023.