FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Inverse kinematic control of a hexapod robot

Leonor de Castro Gothen

Mestrado em Engenharia Eletrotécnica e de Computadores

Supervisor: Paulo Costa (FEUP)

Supervisor: Poramate Manoonpong (SDU)

July 21, 2022

Resumo

Robôs hexápodes são muito adaptáveis ao ambiente em que se inserem e, portanto, são indicados para executar tarefas em vários ambientes diferentes. Este projeto utiliza o MORF, um robô com pernas, numa configuração hexápode para executar tarefas simples.

Devido à adaptabilidade dos robôs hexápodes já ter sido muito estudada e, como tal, já estar bastante desenvolvida, este projeto foca-se na utilidade que este tipo de robô pode ter ao executar tarefas. Para alcançar esse objetivo, foram implementados dois métodos de cinemática inversa: as equações da cinemática inversa e redes neuronais artificiais.

No método que utiliza as equações da cinemática inversa, analisou-se a geometria do MORF para deduzir as equações. Visto que são dependentes da geometria do robô, estas equações apenas podem ser aplicadas em robôs com pernas com a mesma configuração que as pernas do MORF. No entanto, os comprimentos das pernas não estão especificados nas equações, logo, estas podem ser aplicadas a robôs que tenham pernas de outros comprimentos.

Para o método que utiliza as redes neuronais artificiais, foi necessário criar dados para treinar as redes, tendo sido testados três métodos: geração de dados com as equações da cinemática inversa, geração de dados com a simulação e geração de dados com o robô real. Nestes dois últimos, o robô explora a sua área de trabalho, passando por todas as combinações de ângulos possíveis e recolhendo a posição resultante na simulação ou na realidade, respetivamente. As redes neuronais foram configuradas com as coordenadas da posição desejada como as 3 entradas e com os ângulos que resultam nessa posição como as 3 saídas.

Neste projeto, os resultados que permitiram fazer análises foram recolhidos em simulação e todos os métodos foram validados no robô real. As câmaras simulada e real são diferentes devido a limitações no simulador, portanto a identificação em simulação é feita com *ORB detection*, enquanto que no robô real é feita com *blob detection*. Como a câmara real é do tipo *fisheye*, a distorção das imagens tem de ser revertida, o que foi feito com bons resultados. O MORF foi bem sucedido a identificar um botão, andar na sua direção, e ao chegar perto do botão, parar, estabilizando a sua posição para mexer as pernas dianteiras.

Foi na fase da execução da tarefa que foram implementados os métodos de cinemática inversa, tendo isso sido bem sucedido com ambos. O método de geração de dados com o robô real ainda não está desenvolvido o suficiente para ter sucesso, mas o método de geração de dados com a simulação já o está, o que mostra que as equações da cinemática inversa podem ser eliminadas do controlo com cinemática inversa de um robô.

Abstract

Hexapod robots are very adaptable to their environment and, as such, they are suitable for tasks in many environments. This projects uses MORF, a legged robot, in a hexapod configuration for simple task execution.

As the adaptability of a hexapod robot's locomotion has been widely studied, the primary goal is to make a type of adaptable robot even more useful, by allowing it to perform simple tasks. Two different inverse kinematics methods are implemented to achieve this: the inverse kinematics equations and artificial neural networks.

The geometry of MORF's legs was analysed in order to derive the IK equations. These equations are specific to robots that have the same leg configuration as MORF, but they can be used on robots with different leg lengths, as those are unspecified in the equations.

As is the nature of neural networks, they need data to train, so three different methods of data generation were tested: data generation with the IK equations, data generation in simulation, and data generation with the real robot. The last two methods consist of having the robot explore its workspace in simulation and in reality, respectively. The neural networks were set up with 3 inputs - the coordinates of the desired position - and 3 outputs - the leg's angles that result in the desired position.

The project was implemented in the simulation for data collection and in the real robot for validation. MORF successfully identifies a button, walks towards it and, as MORF gets near the button, it stops and successfully adopts a position that is stable for moving the two front legs. Because the simulated camera is different from the real camera due to simulator constraints, the button detection in simulation is done with ORB detection and, in the real robot, it is done with blob detection. As the real camera is of the fisheye type, the images need to be undistorted, which was done with good results.

The task execution was also implemented successfully, both with the IK equations and with the neural networks. The data generation method with the real robot is not well-developed enough to be successful, but the data generation method in simulation is. Therefore, this project shows that the IK equations can be eliminated from the inverse kinematic control of a robot.

Acknowledgments

First of all, I would like to thank Mathias Thor for suggesting this project, for all his guidance and for his constant availability.

I thank Poramate Manoonpong for welcoming me to the ENS Lab and for his help and advice at every stage.

I would also like to thank Paulo Costa for all his help and for making it possible for me to carry out this project.

To my beloved sister, Margarida, my favourite person in the whole world, who has been with me through thick and thin and who has always been there for me, my thanks and love.

To my dear parents, Sofia and Peter, my sincere gratitude and love for raising me to be the person I am today and for encouraging me to always do my best and be whatever I wanted to be.

And last, but not least, I would like to thank my friends for making these 5 years so incredible and for all their support along the way.

Leonor Gothen

"The Ultimate Answer to Life, The Universe and Everything is... 42!

[...]

So once you do know what the question actually is, you'll know what the answer means."

Douglas Adams, The Hitchhiker's Guide to the Galaxy

Contents

List of Figures

List of Tables

Abbreviations and Symbols

ANN	Artificial Neural Network
API	Application Programming Interface
CPG	Central Pattern Generator
IK	Inverse Kinematics
MORF	MOdular Robot Framework
ROS	Robot Operating System
VGM	Vision Guided Manipulation

Chapter 1

Introduction

There are many types of mobile robots. It is widely accepted that wheeled robots are amongst the most popular for their overall simplicity (see, for example, Thor [1]). However, this type of robot is very limited in its adaptability, especially compared to legged robots. As Jianhua [2] writes, hexapod robots are suitable for unstructured terrain and they can move with many different gaits, which makes them very adaptable. This makes hexapod robots adept at tasks in challenging environments, such as search and rescue missions or exploration of hazardous environments (examples are mentioned by Bo *et al.* in [3]).

This project developed a controller for MORF in the hexapod configuration (see Figure 1.1). This controller generates cyclic movement, stabilises MORF with four legs, and uses inverse kinematics to perform a task - pressing a button - with a front leg. Different methods for the task execution are compared, using inverse kinematics equations and neural networks. The data generation for the neural networks is done with three different methods, with the inverse kinematics equations, with a simulation and with the real robot.

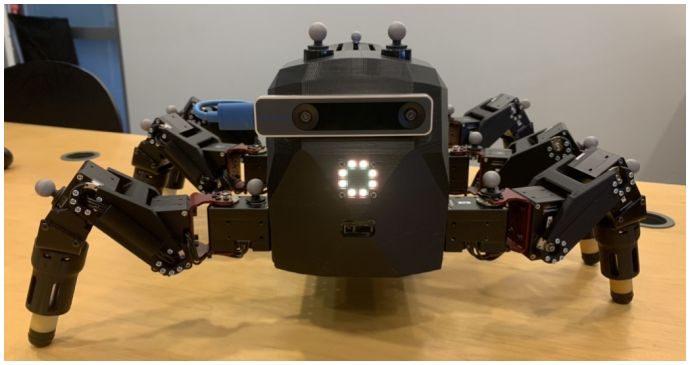


Figure 1.1: MORF in the hexapod configuration.

1.1 Context and motivation

MORF stands for MOdular Robot Framework and it was developed by Thor [1]. MORF is composed of a legged robot (also called MORF) and a software suite. The robot can be configured as a quadrapod or a hexapod, that is, as a mammal or an insect. MORF was created for research studies in a way that it is accessible and simple to use.

The strongest motivation behind this project is to allow MORF to be useful by performing simple tasks. The task used in this project is pressing a button, but the objective is for this solution to be easily adapted for other tasks.

The inverse kinematic control can be done with inverse kinematic equations. However, it is also possible to use neural networks instead. The main motivation for using neural networks is that the need for deriving the equations can, ultimately, be eliminated. This is done by generating data via exploration of the robot's workspace, instead of having to analyse the geometry of the robot.

1.2 Document structure

This document is organized in six chapters, including this introduction as Chapter 1.

Chapter 2 presents a study of the state of the art. This includes a study of other existing hexapod robots, kinematics, gait generation, detection with computer vision and robotic simulators. The sections on kinematics and gait generation also include theoretical concepts for a better understanding of the subjects.

Chapter 3 provides an explanation of all the elements in the implemented solution. These elements are the inverse kinematics calculations, the implementation of neural networks, the computer vision solution - which includes depth perception, detection and pose estimation -, and the movement and task execution.

Chapter 4 shows the results of the different methods implemented in this project.

Chapter 5 presents a discussion of the results, a discussion of the code adaptability and a comparison to other works.

Chapter 6 presents the conclusions drawn from this project.

Chapter 2

State of the art

The first section of this chapter gives context to the world of hexapod robots, by exploring other works with this type of robot.

There are several concepts which are important to this project. The understanding of kinematics, gaits and central pattern generators is fundamental in this area of work. Sections 2.2 and 2.3 define and explain these concepts, and analyse different solutions to the problems that are related to these concepts, specifically kinematics in hexapod robots and gait generation algorithms.

Computer vision is crucial to this project, and there are several detection algorithms available. Some of these algorithms are compared in Section 2.4.

Furthermore, there is a robotic simulators comparison, in order to choose the best simulating environment for this project.

2.1 Hexapod robots

There are various works with hexapod robot's that have different features and different methods of achieving them. Table 2.1 shows several works with hexapod robots, and presents the features developed and the methods used.

Table 2.1: Works with hexapod robots.

Title	Author(s)	Features	Methods
Design and Simulation of Special Hexapod Robot with Vertical Climbing Ability	Zhou and Zhu [4]	Capable of vertical wall climbing.	The robot has crescent-shaped legs and it is equipped with suction cups and spikes that allow it to climb vertical surfaces. The spikes are controlled by electromagnets in order to adapt to different roughnesses in the vertical surfaces.

Design and Configuration of a Hexapod Walking Robot	Bo *et al.* [3]	Walks in unstructured terrain.	The robot has a special foot design, force sensors and proximity sensors to facilitate walking in unstructured terrain.
Terrain adaptation gait algorithm in a hexapod walking robot	Isvara *et al.* [5]	Walks in unstructured terrain.	The robot is equipped with simple push buttons as tactile sensors and a simple gait algorithm is used. The joint angles necessary to generate the gait are calculated with inverse kinematics equations.
Enhancing adaptability with local reactive behaviors for hexapod walking robot via sensory feedback integrated central pattern generator	Yu *et al.* [6]	Enhanced adaptability to the environment.	The robot uses feedback from sensors in the central pattern generator.
Neural coupled central pattern generator based smooth gait transition of a biomimetic hexapod robot	Bal [7]	Smooth gait transition.	This solution uses a central pattern generator, a leg trajectory generator and an inverse kinematics module.
Object carrying of hexapod robots with integrated mechanism of leg and arm	Deng *et al.* [8]	Can carry objects.	The hexagonal robot converts one or two legs into arms. The gait is adjusted to maintain the centre of gravity's position and to compensate for the carried object's weight.

Of these works, the one by Deng *et al.* is the closest to this project, as it also uses some of the legs to interact with the environment. However, the physical structure of MORF and this robot are very different - one is rectangular and the other is hexagonal. Therefore, the type of task to be executed is different, as MORF interacts with the environment while stopped by pushing a button.

2.2 Kinematics

2.2.1 Theoretical concepts

This subsection is based on the chapter "Kinematics" in [9].

Siciliano and Khatib [9] define *kinematics* as pertaining "to the motion of bodies in a robotic mechanism without regard to the forces/torques that cause the motion". This means that we use kinematics in order to describe or set the robot's positions, without taking forces or torques into account.

It is important to understand the concept of joint, because this is what makes the robot's motion possible and, therefore, it is essential for kinematics. A *joint* is what connects rigid parts of the robot, in order to allow movement. A simple way of understanding this is by thinking of the human body: the elbow joint connects the forearm and the upper arm, which allows movement with these two rigid parts. There are many different types of joints, which allow for different motions. The simplest types of joints, and therefore very common, are the revolute joint and the prismatic joint (see Figure 2.1).

These two simple joints are explained next, because they can describe the other types of joints when combined in different ways. For example, a spherical joint can be described as three revolute joints.

The *revolute joint* allows angular movement, where one of the rigid parts rotates around an axis relative to the other rigid part. The relative position of the rotating part is given by "the angle between two lines normal to the joint axis, one fixed in each [part]", as written in [9].

The *prismatic joint* allows a translational movement, that is, one of the rigid parts slides along an axis relative to the other rigid part. The relative position of the sliding part is given by "the distance between two points on a line parallel to the direction of sliding, with one point fixed in each [part]", as written in [9].

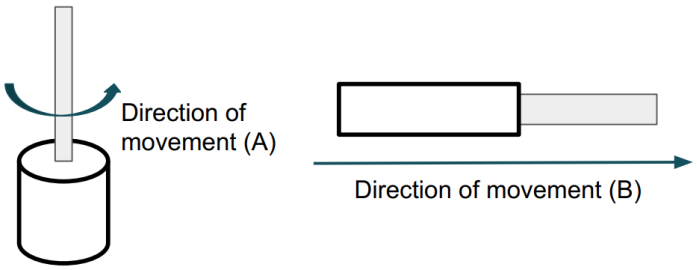


Figure 2.1: Revolute (A) and prismatic (B) joints.

In order to generate more complex positions, robots usually have several joints in series. This generates the need to either discover the position of the endpoint when in a known configuration, or to set the joints in the correct relative positions in order to have the endpoint in a certain position. This is where kinematics are used. Kinematics can be divided into forward kinematics and inverse kinematics. These two types of kinematics solve opposite problems. *Forward kinematics* define the position of the endpoint when in a known configuration, while *inverse kinematics* define the correct relative positions of the joints in order to have the endpoint in a certain position.

When using forward kinematics, there is only one solution for the endpoint position. All the angles and displacements are known, which provides certainty. However, there may be more than

one solution to the inverse kinematics problem. More than one configuration of the joints can lead to the same endpoint position, which leads to ambiguity.

When faced with more than one solution to an inverse kinematics problem, only one of those solutions can be chosen. This creates the need for decision criteria, of which there are many. For example, one can choose a solution that creates more overall stability for the robot, or a solution that requires a smaller movement. There may even be solutions that exist analytically, but are impossible because of the external environment in which the robot is situated.

2.2.2 Kinematics in hexapod robots

There are several solutions for defining the kinematics of a hexapod robot. This subsection compares four solutions that have been implemented in this area for hexapod robots with a similar design to MORF's. Hence, the analysis is restricted to hexapod robots with three revolute joints in a similar configuration to the one shown in Figure 2.2.

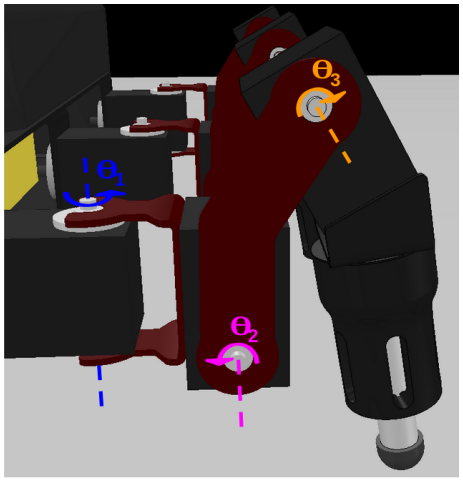


Figure 2.2: MORF's leg joints' rotation axes and angles.

The methods analysed fall into two categories: algebraic and geometric. The algebraic methods use matrices to derive the inverse kinematics parameters, while the geometric methods use the inherent geometry of the robot's leg.

In [10], Sun *et al.* use a method based on transformation matrices. The first step is to define the direct kinematics of the hexapod robot. Sun *et al.* [10] use the Denavit–Hartenberg parameters, which means that, for each link in the robot, the reference frame for that link is defined taking into account the displacement or rotation of the previous joint, as well as the displacements or rotations inherent to the robot build. This allows these authors to define the transformation matrix from the beginning of the leg to the endpoint (${}^{o_{m0}}T_{o_{m4}}$). These authors then proceed to use the transformation matrix ${}^{o_{m0}}T_{o_{m4}}$ to define the inverse kinematics parameters, namely the three joints' angles. Sun *et al.* [10] use the notation ${}^{o_{mx}}T_{o_{my}}$ to refer to the transformation matrix from the frame o_{mx} to the frame o_{my}. These frames are the coordinate frames for each joint - in the order presented in Figure 2.2 - and for the endpoint as frame o_{m4}.

The transformation matrix ${}^{o_{m0}}T_{o_{m4}}$ is obtained by multiplying the transformation matrices for each link (${}^{o_{mk-1}}T_{o_{mk}}$). These transformation matrices are, in turn, obtained from the Denavit–Hartenberg parameters. Sun *et al.* [10] use the equation

$$ {}^{o_{m0}}T_{o_{m1}}^{-1} \cdot {}^{o_{m0}}T_{o_{m4}} = {}^{o_{m1}}T_{o_{m2}} \cdot {}^{o_{m2}}T_{o_{m3}} \cdot {}^{o_{m3}}T_{o_{m4}} \tag{2.1} $$

to define the inverse kinematics parameters as the equation's solutions. The left-hand side of Equation 2.1 consists of multiplying the inverse of the first link's transformation matrix by the transformation matrix ${}^{o_{m0}}T_{o_{m4}}$. The right-hand side consists of the multiplication used to obtain ${}^{o_{m0}}T_{o_{m4}}$ also multiplied by the inverse of the first link's transformation matrix.

In [11], Hasnaa and Mohammed base their inverse kinematics analysis on the robot's coordinate frames and how these frames relate to each other. As such, their first step is to choose the world frame and the frames for the robot's body, each joint in the leg and the endpoint. These authors define the rotation matrices between these frames.

Hasnaa and Mohammed [11] then use the vectorial relation between the joints. The vector from the origin of the world frame to the endpoint of the leg is defined as the sum of the vector from the origin of the world frame to the beginning of the leg and the vectors from one joint to the next until the endpoint. After defining this relationship, the authors use the rotation matrices in order to project all of the vectors in the world frame. The final step in this approach is to use the vectorial relationship and unspecified geometric properties of the robot to obtain the inverse kinematics parameters.

In [12], Priandana *et al.* use a geometric approach to calculate the inverse kinematics parameters of their hexapod robot. The first joint angle is the simplest to calculate, because it has a different orientation from the other two joints, so the direction of the leg is defined with only this angle. As such, these authors project the leg onto the ground plane and define θ_1.

The two remaining angles are calculated based on a side view of the robot leg. By projecting the leg onto the vertical plane that is perpendicular to the robot's body, Priandana *et al.* [12] define two triangles: one where the vertices are the centre of the second joint, the endpoint and the point on the ground that forms an orthogonal triangle with the two other points, and another where the vertices are the centre of the second joint, the centre of the third joint and the endpoint. With these two triangles, the authors geometrically derive the expressions for θ_2 and θ_3.

Aju and Napolean [13] also use a geometrical approach to determine the inverse kinematics parameters of the hexapod robot. The first joint angle is determined in a similar way to the one used by Priandana *et al.* [12]. The only difference is that Aju and Napolean [13] consider the angular offset for the servo motor that controls this joint.

Aju and Napolean [13] define triangles with the same points as Priandana *et al.* [12], but Aju and Napolean [13] use a different position of the leg. Aju and Napolean [13] use a position where θ_2 is below the horizontal, while Priandana *et al.* [12] use a position where θ_2 is above the horizontal. As such, the geometrical relationships used to derive the expressions for θ_2 and θ_3 are slightly different.

After analysing these methods, it is possible to observe that a geometrical approach is the simplest one. The geometry of the robot's leg in this type of configuration can be defined by several triangles and the relationships between the angles in triangles are very well defined. The method used by Sun *et al.* [10] is relatively simple, but it is still more laborious than a geometric approach, as it requires the direct kinematics analysis and then algebraic manipulation of matrices. The method used by Hasnaa and Mohammed [11] is the most complicated, as it requires an algebraic and geometric analysis of the robot.

2.3 Gait generation

2.3.1 Gaits

A *gait* is defined by Haynes and Rizzi [14] as "a cyclic motion pattern that produces locomotion through a sequence of foot contacts with the ground". Many different types of gaits exist to produce different types of locomotion. There are two important terms associated with gaits: stance and flight (or swing), which are also defined in [14]. Stance is the period in which the leg is in contact with the ground, that is, the period in which the leg is generating the robot's movement. Swing is the period in which the leg is not in contact with the ground, that is, the period in which the leg is returning to the beginning of the stance period.

The remainder of this subsection is based on Sun *et al.* [10], which focuses on the tripod gait.

The *tripod gait* is a simple gait that groups the robots legs in two groups of three, so there are three legs in the stance phase and three legs in the swing phase at any given moment. In this type of gait, the front and back legs of each side are grouped with the middle leg of the other side. This gait is shown in Figure 2.3.

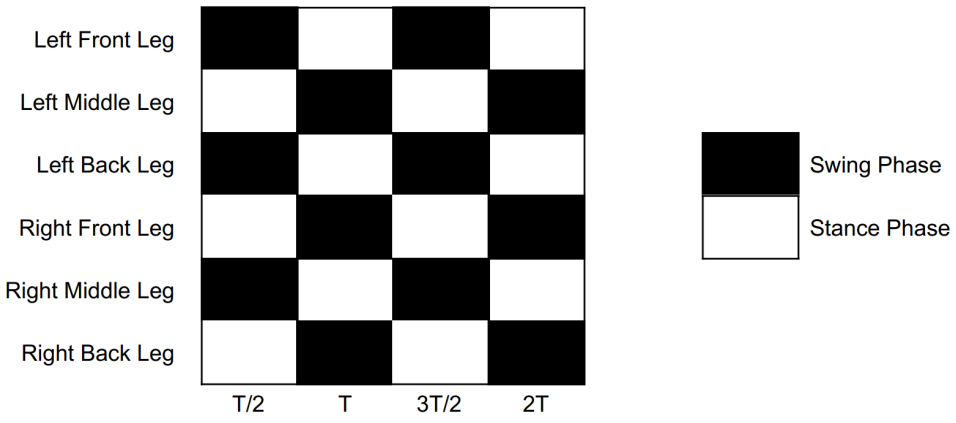
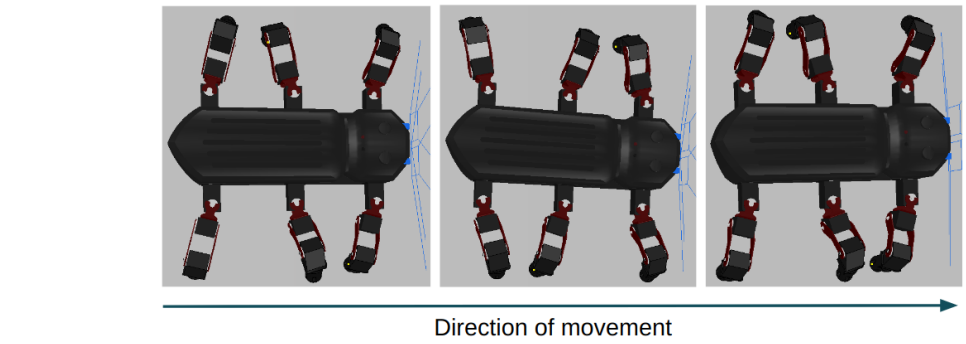
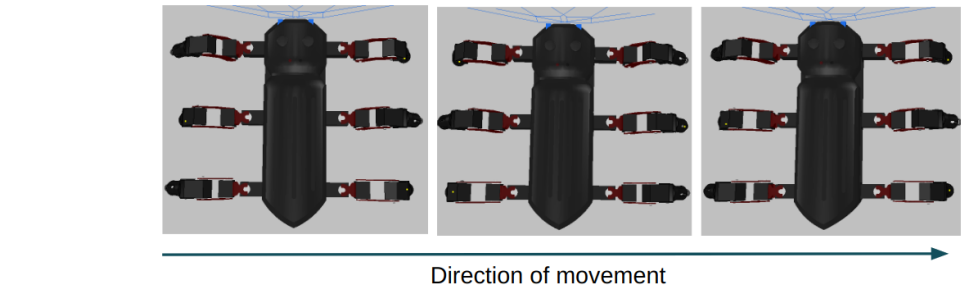


Figure 2.3: Tripod gait diagram (based on [10, Fig. 3.]).

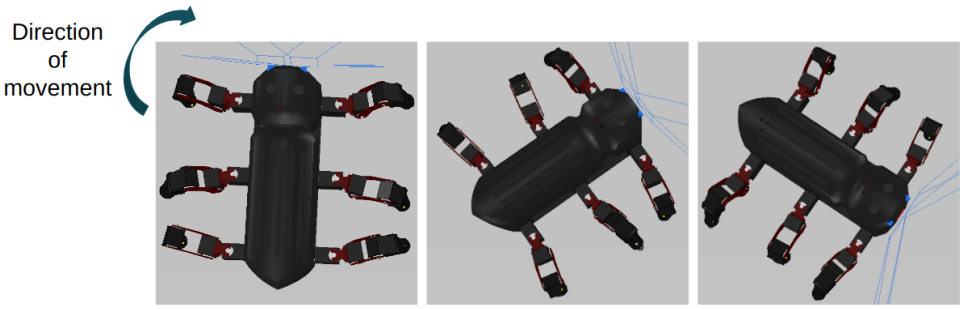
The tripod gait can be divided into three different gaits (see Figure 2.4): the straight gait, the transverse gait, and the swivel gait. These three gaits differ in the orientation of the movement, but they all have the leg groups of the tripod gait.



(a) Straight gait.



(b) Transverse gait.



(c) Swivel gait.

Figure 2.4: Resulting movement from straight gait, transverse gait and swivel gait.

The *straight gait* generates forward and backward movement. This means that the stance phase consists of moving the foot parallel to the robot's body in a straight line and the swing phase consists of a circular movement parallel to the robot's body.

The *transverse gait* generates sideways movement. The foot movement is very similar to the movement in the straight gait, except that it is perpendicular to the robot's body.

The *swivel gait* generates angular movement. This means that the legs can be arranged around the robot, instead of being parallel to each other. The movement of the feet is tangent to a circle that has the centre of the robot as origin. As in the previous gaits, the stance phase consists of moving the feet in a straight line and the swing phase consists of moving them in a circular arc.

2.3.2 Central Pattern Generators

This subsection is based on Ijspeert [15].

Ijspeert [15] defines *central pattern generators* (CPGs) as "neural circuits found in both invertebrate and vertebrate animals that can produce rhythmic patterns of neural activity without receiving rhythmic inputs". The natural CPGs found in nature serve as inspiration for robotics control.

It is important for a system that uses CPGs to receive sensory feedback, in order to coordinate the motion of the robot and the generated signal. It is possible to generate a pattern without this feedback, but then it cannot adapt the motion to the environment.

A CPG is a neural network, which means that there are several neurons that interact with each other. A very commonly used neuron model is the Hodgkin–Huxley model, which simulates the behaviour of a biological neuron. CPGs can be used to generate locomotion, particularly when there is cyclic movement. As written earlier in this report, gaits are cyclic, which makes CPGs a good way of generating them.

CPGs are well suited for generating robot movement, for of several reasons. Firstly, the CPG generally has parameters that allow for modification of the pattern, which makes for an easily adaptable system. Secondly, the sensory feedback results in an automatic integration of the locomotion with the environment. Thirdly, the CPG returns to the original pattern after a perturbation in a smooth way.

2.3.3 Gait generation algorithms

Gaits can be generated using several methods. This subsection compares four different approaches to gait generation: a kinematic solution and three solutions with CPGs.

In [10], Sun *et al.* use a kinematic approach to solve the problem of gait generation. These authors determine the inverse kinematics parameters as explained in Subsection 2.2.2. Sun *et al.* [10] calculate the endpoint's trajectory equations for the intended gait, in this case the tripod gait. These equations are then substituted in the inverse kinematics parameters. As such, the authors define the joint's angles for each moment in time that create the desired gait.

In [6], Yu *et al.* use a CPG network with three coupled neurons to generate movement in each leg. The approach taken by these authors involves two parts, a phase regulator and a linear converter. The three neurons that compose the phase regulator generate sinusoidal signals with different phases. The linear converter transfers these signals to the joints. In total, this method needs 6 CPG networks - one for each leg. These networks are grouped according to the tripod gait. The networks for each group are initialised with the same values and these values are symmetrical between groups, in order to create the different phases of the gait.

The CPG network Yu *et al.* [6] implement has a *leader-follower* relationship where the neuron that controls θ_1 (see Figure 2.2 for reference) is the leader and the other two neurons are the followers. This means that the leader neuron has an influence over the other two neurons and the followers do not influence the leader, but they do influence each other. The leader influences the followers in that it dictates the oscillating frequency of the followers as double its own oscillating frequency. The follower neurons influence each other to synchronise their oscillation.

Yu *et al.* [6] use inverse kinematics to determine the coefficients for the linear converter. This is necessary to determine the range of the different joints and, as such, prevent the controller from sending signals that are incompatible with the joints.

In [16], Rostro-Gonzalez *et al.* also use a CPG network to generate movement. These authors use two neurons for each leg and each neuron controls one joint. Rostro-Gonzalez *et al.* [16] connect these neurons in different ways to implement different gaits.

In [17], Ren *et al.* propose two different solutions to the gait generation problem. The first solution is to implement a single chaotic CPG, while the second solution is to implement one chaotic CPG for each leg. A chaotic CPG is composed of two neurons and a chaos controller. This chaos controller defines the input signals to the CPG and it makes it possible to define different oscillation periods.

For the first solution, all the legs share the same CPG, which means that, as long as all of the robot's legs are fully functional, it works. However, if one of the joints in one leg has a problem, the single CPG cannot adapt to that, as it has to control all of the other legs as well. That is the reason for the second solution proposed by Ren *et al.* [17]. With a CPG for each leg, the robot's control is much more adaptable, as each CPG can adapt to the situation in the corresponding leg through the CPG inputs.

Through the use of one phase switching network and two velocity regulating networks, Ren *et al.* [17] can control the three joints in each leg with the CPG that consists of two neurons. These networks are also neural networks that accept the CPG outputs as inputs.

After this analysis, it becomes clear that a CPG-based solution is the most desirable. The kinematic solution demands a constant processing of the kinematics parameters through the trajectory equations, as it has to be calculated for every moment in time. Furthermore, this approach is not adaptable, because it does not take into consideration possible changes in the environment, as the trajectory equations are fixed for a perfectly smooth surface.

In regard to the different CPG-based solutions, it is not as obvious which would be the best one. The solution presented by Rostro-Gonzalez *et al.* [16] only uses two joints, and it is, as such, the simplest one. Even though generating cyclic movement is a necessary part of this project, it is not the primary focus. Therefore, a similar simple solution to the one proposed by Rostro-Gonzalez *et al.* [16] is the most suited.

2.4 Detection with computer vision

Dawson-Howe [18] defines computer vision as "the automatic analysis of images and videos by computers in order to gain some understanding of the world". This definition highlights the fact that computer vision is, in a way, an imitation of the human vision. Just like humans use our eyes to gain visual information about our environment, artificial systems use computer vision.

An image needs to be processed by the computer to detect different aspects of it. For the computer, an image is simply a collection of pixels in a specific arrangement. As such, a detection algorithm has to be used to collect usable information from the image.

There are many detection algorithms available. SIFT (Scale Invariant Feature Transform), SURF (Speeded Up Robust Features), FAST (Features from Accelerated Segment Test) and ORB (Oriented fast and Rotated Brief) are feature detection algorithms that are available in OpenCV [19]. Another detection algorithm available in OpenCV is Blob Detection [20].

Lowe [21] presented SIFT, which finds the keypoints by cascade filtering the image with a difference-of-Gaussian function. Bay *et al.* [22] proposed SURF to speed up feature detection compared to SIFT by using a detector based on an approximation of the Hessian matrix. Rosten *et al.* [23] introduced FAST, which identifies keypoints by comparing the intensity of a pixel with the pixels surrounding it. Rublee *et al.* [24] developed ORB as an open source alternative to SIFT and SURF.

Blob detection is defined as a method "aimed at detecting regions in a digital image that differ in properties, such as brightness or color, compared to surrounding regions" [25]. This means that blob detectors identify regions with certain specified properties. The blob detector in OpenCV can filter the blobs by area, circularity, ratio of the minimum inertia to maximum inertia or convexity [20]. This type of detector seems to be the most indicated for this project, as the goal is to identify a regular circular button.

2.5 Robotic simulators

Simulation is defined in the Merriam-Webster dictionary [26] as "the imitative representation of the functioning of one system or process by means of the functioning of another". As such, simulation is a very useful tool in robotics, because it allows the study of a real system without the need for having it physically. This makes it possible to test our work in a safe environment, more economically [27].

There exist many simulators with different characteristics, so it is important to do some research, in order to choose the simulator which is best suited for our needs. This section analyses four simulators used in robotics: SimTwo [28], CoppeliaSim [29] (previously V-Rep), Gazebo [30] and Webots [31].

SimTwo is a free simulator that uses ODE as its physics engine. This simulator is compatible with Windows and does not have dependencies beyond what is included in the GitHub releases [28]. Control code can be implemented directly in a script in the simulator or remotely with communication via UDP or serial port [32]. SimTwo does not have a ROS integration, but it is possible to use ROS, by passing the information to the simulator via UDP, as proposed by Pinho *et al.* in [32].

CoppeliaSim is a cross-platform simulator that is compatible with Linux, Windows and MacOS. It is paid for commercial use, but free for educational purposes and there is also a free player version that allows interaction with CoppeliaSim simulations. CoppeliaSim supports 7 programming languages, using the regular and remote APIs. This simulator has 5 different programming approaches - embedded scripts, remote API clients, ROS nodes, plugins and add-ons - that are compatible with each other, which makes for a highly costumizable simulation. CoppeliaSim

supports 4 physics engines - Bullet Physics, ODE, Newton and Vortex Dynamics. All of this information is available on their website [29].

Gazebo is open-sourced and licensed under Apache 2.0 [33], so it is completely free to use. As is written on their website [30], Gazebo runs best on Ubuntu, but it can also be used on other Linux distributions, Windows and MacOS. The programming language used for the plugins and for the external API is C++ [27]. Gazebo supports 4 physics engines - ODE, Bullet, Simbody, and DART. According to [33], ROS uses Gazebo as its default simulator and, as such, there are plugins in the ROS repository that facilitate the integration of ROS and Gazebo.

Webots is also open-sourced and licensed under Apache 2.0, and it is therefore free to use. Webots is compatible with Windows, Linux and MacOS and it uses ODE as its physics engine. This simulator allows programming in C, C++, Python, Java, MATLAB and ROS. Their website [31] has this information available.

It is possible to observe that three of these simulators are available on Linux, Windows and MacOS. Webots has less versatility in terms of physics engines, but it is more versatile in terms of programming languages than Gazebo, although less than CoppeliaSim. All of them can be used with ROS, but Gazebo is easier to integrate than the others. Both Gazebo and Webots are completely free to use, while CoppeliaSim is only free for educational purposes.

A model of MORF was already implemented in CoppeliaSim, using the Vortex Dynamics physics engine. Given that this project has an educational purpose, CoppeliaSim can be used for free, so that ceases to be an impediment for using this simulator. Although ROS is more easily integrated with Gazebo, it is not hard to use ROS with CoppeliaSim, by running it in a separate terminal. Considering all of these factors, CoppeliaSim is the simulator used for this project.

Chapter 3

Methods

The overall method for executing the task of pressing a button is represented in Figure 3.1. The robot walks with the CPG until it is near the button, which is detected with computer vision. When this happens, the task is executed either with the inverse kinematic equations or with the neural networks, depending on the input from the user.

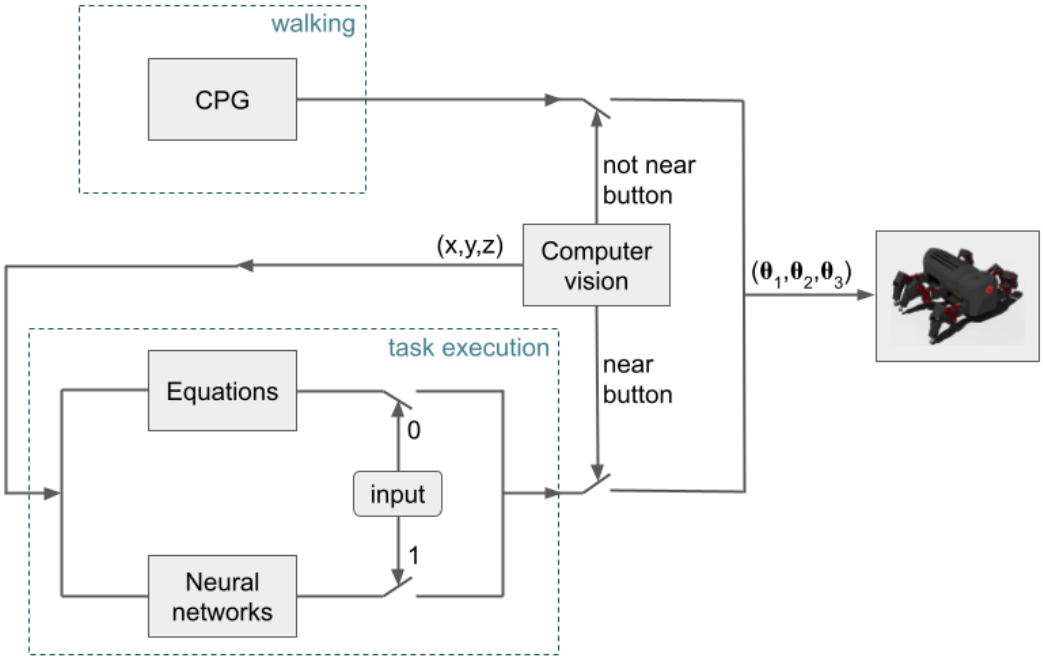


Figure 3.1: Overall view of the system

MORF is equipped with 3 motors in each leg - which gives a total of 18 motors - and a stereo fisheye camera. With a lithium battery of 22.2V with 6 cores, this robot has an autonomy of about 10 to 15 minutes.

3.1 Computer vision

The image_transport and cv_bridge libraries are used to receive the images from the robot via ROS. The images are received in the openCV Mat format.

3.1.1 Depth perception

MORF is equipped with a stereo camera. A stereo camera imitates the human eye setup, as it is composed of two cameras to allow for depth perception.

Sankowski *et al.* [34] define stereo vision as "an imaging technique that allows the reconstruction of point coordinates in three-dimensional space based on images acquired from two cameras". By having two images of the same scene with slightly different scopes, it is possible to calculate the depth of objects relative to the camera.

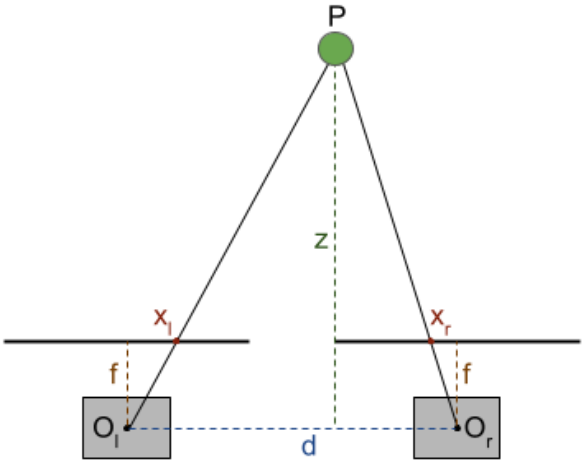


Figure 3.2: Triangulation geometry (based on [35, Figure 4]). z is the depth, f is the focal length, d is the distance between the cameras, x_l and x_r are the detected point in the image planes, and O_l and O_r are the cameras' centres.

The method used to determine the depth is called triangulation. Figure 3.2 represents the geometry of the system in stereo vision. As Kitayama *et al.* [35] write, this geometry yields the relation

$$z = \frac{f \cdot d}{x_l - x_r}. \tag{3.1}$$

In the simulation, the focal length was determined empirically by trial and error as it was impossible to determine otherwise, while the focal length of the real camera was determined by calibrating it.

3.1.2 Detection in the simulation

In the simulation, the button is detected using the ORB detection, an algorithm discussed in Section 2.4. This yields keypoints and their corresponding descriptors in each image that have to be

matched between them. This matching was achieved with OpenCV's Flann based feature matcher. The matches need to be filtered in order to only use the correct ones. This is achieved with Lowe's ratio test [21], a commonly used method which compares the distance of a keypoint in one image to it's two closest keypoints in the other image.

This algorithm detects all the features in the images, so the features on the button have to be selected. This was done by finding the matched keypoints corresponding to pixels with a green hue in both images. The pixel hue was obtained by converting the images from the RGB format to the HSV format.

After having discovered the button features, their average position was calculated to be used in the depth perception and the pose estimation.

3.1.3 Detection in the real robot

Unlike in the simulation, the real stereo camera has to be calibrated, especially as it is composed of two fisheye cameras that introduce considerable distortion to the images.

The first step in the calibration is obtaining several images of a chessboard pattern with the robot. In this project, six pairs of images were taken - one taken with the left camera and the other taken with the right camera at the same time. Then, the OpenCV function "findChessboard-Corners" is used to, as the name indicates, find the corners in the chessboard pattern. These points are used by the OpenCV function fisheye::stereoCalibrate, which finds the intrinsic camera matrices and the distortion coefficients.

In order to use the images for the button detection, they are undistorted using the matrices and distortion coefficients determined in the callibration. This is done by applying a map obtained with the OpenCV function fisheye::initUndistortRectifyMap to the images.

In the real robot, the button is detected using blob detection, another algorithm discussed in Section 2.4. The OpenCV library has the SimpleBlobDetector, which identifies the blobs and filters them as specified. The filters used in this project are area, circularity and inertia. These filters ensure that the blob has a minimum area, is mostly circular and can have a small distortion. Because the real camera captures pictures in grayscale, the filters are the only criteria available for recognising the button.

3.1.4 Pose estimation

After detecting the button in both images, Equation 3.1 is used to calculate the depth of the button (z coordinate in the camera frame). This depth information is used to calculate the two other coordinates of the button. Figure 3.3 illustrates the relationship between the vertical field of view (fov_y), the depth and the height at that depth, which yields the equation

$$height = 2z\tan(\frac{fov_y}{2}). \tag{3.2}$$

Equation 3.2 is also valid for the width by adjusting the field of view value to the horizontal field of view. It is then straightfoward to convert the position in pixels (x_{pix}, y_{pix}) to camera frame coordinates, by taking into account the number of vertical pixels (p_y), with the expression

$$y = -\frac{y_{pixL}}{p_y} \cdot height + \frac{height}{2}. \tag{3.3}$$

Again, Equation 3.3 is also valid for the x coordinate by using width instead of height and the x pixel coordinate instead of the y pixel coordinate. For the real robot, the camera pixel coordinates (cam_x, cam_y) are not necessarily in the middle of the image, so that has to be taken into account by modifying Equation 3.3 to

$$y = -\frac{y_{pixL}}{p_y} \cdot height + \frac{cam_y \cdot height}{p_y}.$$

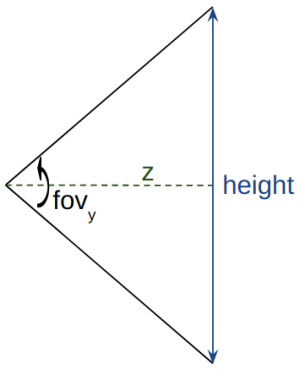


Figure 3.3: Relationship between the vertical field of view (fov_y), the depth (z) and the height at that depth.

In the simulation, the fields of view were known, as they were parameters in the simulated camera. However, the fields of view of the real camera were calculated with the intrinsic matrix, which contains the pixel position of the camera in the image. As such, the equation

$$fov_x = \arctan 2(cam_x, f) \tag{3.4}$$

defines the horizontal field of view. If the x camera pixel coordinate is substituted with the y camera pixel coordinate in Equation 3.4, the vertical field of view is obtained.

3.1.5 Algorithms

Algorithms 1 and 2 show the pseudocode for applying the detection and pose estimation in simulation and on the real robot, respectively.

Algorithm 1 Detection and pose estimation with computer vision in simulation.

function vision_sim(img) ▷ Images are the inputs
$hsv \leftarrow$ rgb2hsv(img) ▷ Convert images to HSV
$keypoints \leftarrow$ ORB.detect(img) ▷ Detect keypoints with ORB
$descriptors \leftarrow$ ORB.compute($img, keypoints$) ▷ Compute descriptors with ORB
$matches \leftarrow$ Flann($descriptors$) ▷ Find matches with Flann matcher
for $i \leftarrow 0$ to $matches.size$ **do** ▷ Go through all the matches
    if Lowe's ratio($matches[i]$) **then** ▷ Do Lowe's ratio test
        if green($hsv, keypoints[i]$) **then** ▷ Check if keypoints are green
            $green\_matches \leftarrow matches[i]$ ▷ Only save the good matches
        end if
    end if
end for
for $i \leftarrow 0$ to $green\_matches.size$ **do** ▷ Go through all the good matches
    $z \leftarrow \frac{f \cdot d}{green\_matches[i].L.x - green\_matches[i].R.x}$ ▷ Triangulation
end for
$height \leftarrow 2 \times \text{mean}(z) \times \tan(\frac{fov_y}{2})$ ▷ Compute the height of the images in real world units
$width \leftarrow \frac{848}{800} \times height$ ▷ Compute the width of the images in real world units
$x \leftarrow -\frac{\text{mean}(green\_matches.L.x)}{848} \times width + \frac{width}{2}$ ▷ Calculate the x coordinate
$y \leftarrow -\frac{\text{mean}(green\_matches.L.y)}{800} \times height + \frac{height}{2}$ ▷ Calculate the y coordinate
return (x, y, z) ▷ Coordinates are the outputs

Algorithm 2 Detection and pose estimation with computer vision in the real robot.

function vision_real(img) ▷ Images are the inputs
$img \leftarrow$ undistort(img) ▷ Remove fisheye distortion
$params.minArea \leftarrow 100$ ▷ Define blob parameter for the area filter
$params.minCircularity \leftarrow 0.9$ ▷ Define blob parameter for the circularity filter
$params.minInertiaRatio \leftarrow 0.6$ ▷ Define blob parameter for the inertia filter
$keypoints \leftarrow$ blob.detect($img, params$) ▷ Detect keypoints with the blob detector
$fov_x \leftarrow \arctan 2(cam_x, f)$ ▷ Calculate the horizontal field of view
$fov_y \leftarrow \arctan 2(cam_y, f)$ ▷ Calculate the vertical field of view
$z \leftarrow \frac{f \cdot d}{keypoints.L.x - keypoints.R.x}$ ▷ Triangulation
$height \leftarrow 2 \times \text{mean}(z) \times \tan(\frac{fov_y}{2})$ ▷ Compute the height of the images in real world units
$width \leftarrow 2 \times \text{mean}(z) \times \tan(\frac{fov_x}{2})$ ▷ Compute the width of the images in real world units
$x \leftarrow -\frac{keypoints.L.x}{848} \times width + \frac{cam_x \times width}{848}$ ▷ Calculate the x coordinate
$y \leftarrow -\frac{keypoints.L.y}{800} \times height + \frac{cam_y \times height}{800}$ ▷ Calculate the y coordinate
return (x, y, z) ▷ Coordinates are the outputs

3.2 Gait generation

The CPG controls two joints in each leg, as Subsection 2.3.3 shows this as the simplest approach. The CPG is composed of two neurons that use the hiperbolic tangent as their activation function. Figure 3.4 shows the neuron's configuration, which results in the output signals shown in Figure 3.5.

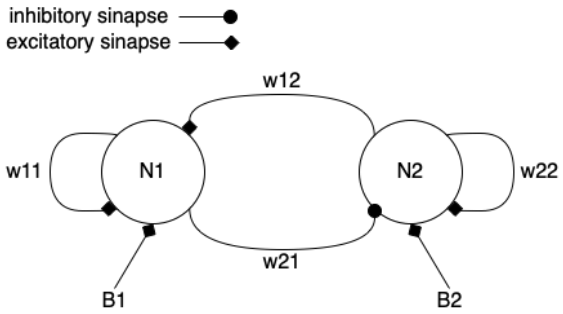


Figure 3.4: CPG neurons configuration (based on [17, Fig. 1.]).

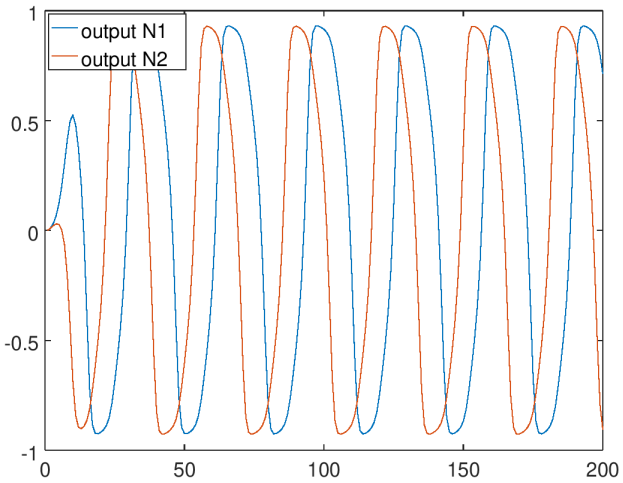


Figure 3.5: CPG outputs before scaling.

The output from the neurons is scaled to ensure that there are no collisions between the legs. The scaling constant is also used to slow down MORF's movement by being decreased.

The scaled outputs from the CPG (oH_1 and oH_2) are used to control MORF's joints as shown in Figure 3.6. The direction of the movement is influenced by the factor d, which is the horizontal displacement of the identified button relative to a predetermined position. This predetermined position is defined so that the front left leg is able to reach the button.

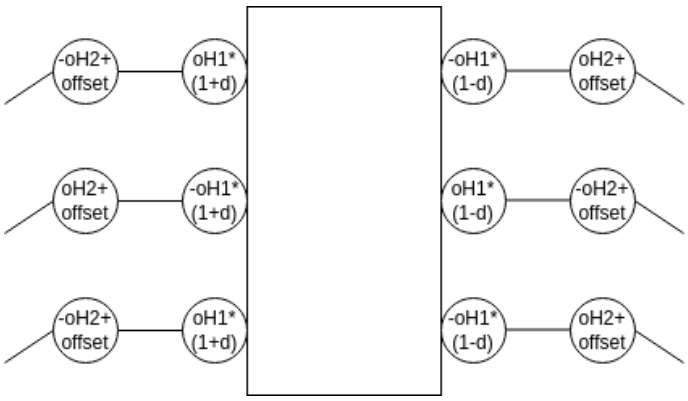


Figure 3.6: CPG outputs in MORF's joints

3.3 Control

The movement control is implemented with state machines. Figure 3.7 presents the state diagram for the general control and Figure 3.8 presents the state diagram for the stabilisation process.

The general control is composed of 6 states. When the program starts, MORF is in a default position and, when images from the stereo camera are received, MORF starts moving with the CPG. As MORF gets closer to the button, the movement is slowed until the button is reached. When MORF is at the button, it performs the stabilisation process. After MORF is stable, the endpoint of the front left leg goes in front of the button, but with a small distance to the button. Then, the distance to the button is slowly decreased until MORF feels the button with the force sensor.

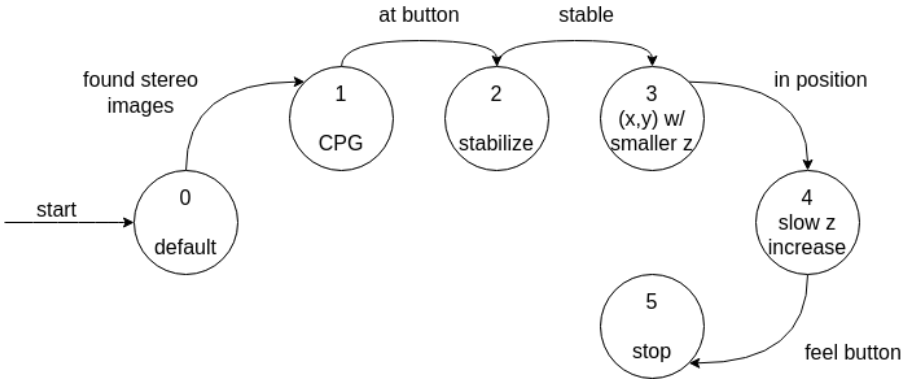


Figure 3.7: General control state diagram

The stabilisation process is composed of 9 states. This process is started when the general process reaches state 2. The legs are individually placed in their stable configuration in the order FL (front left) > BL (back left) > ML (middle left) > FR (front right) > BR (back right) > MR (middle right). The legs need an auxiliary state in which they are in the air to ensure that MORF does not move while rearranging it's legs.

3.4 Task execution

As previously mentioned, the task execution is where the inverse kinematic control is used. Two methods were explored: using inverse kinematics equations and using neural networks.

In both of these methods, the desired foot position is given as an input and the leg angles that result in that position are received as outputs. After having derived the neural networks and after having trained the neural networks, the logic is the same.

The difference in these methods is that the inverse kinematic equations are derived by analysing the geometry of the robot, while the neural networks use data - the pairs of input-output explained in the last paragraph - to train.

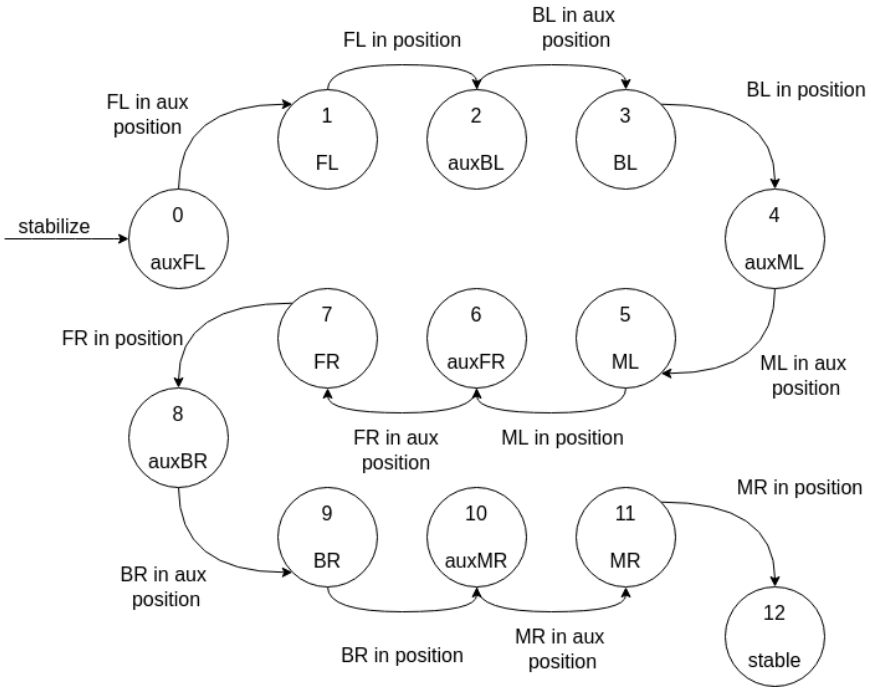


Figure 3.8: Stabilisation state diagram

3.4.1 Inverse kinematics with equations

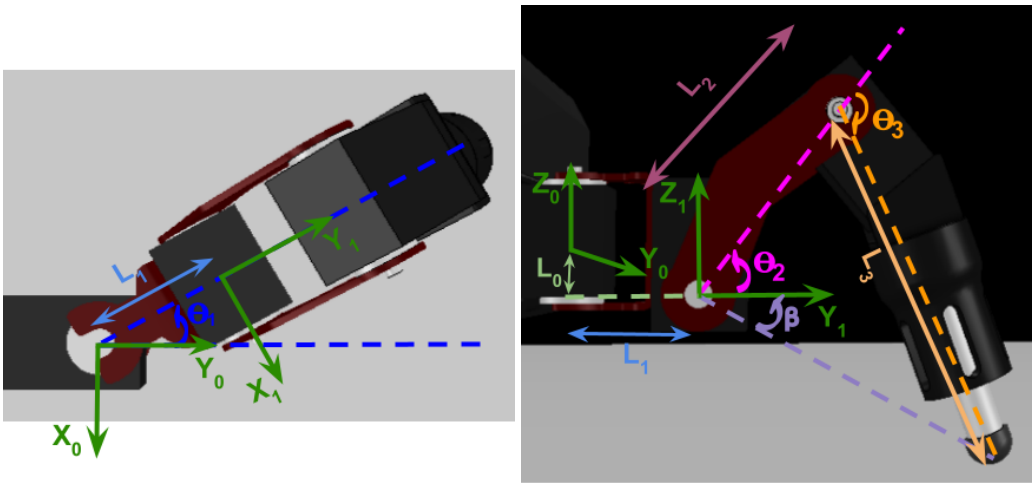
As discussed in Chapter 2, it is simplest to take a geometric approach to solve the inverse kinematics problem. In order to do this, it is necessary to define the angles, lengths and reference frames in the robot's leg. Figure 3.9 shows a graphical representation of how these parameters are defined: each angle used for the inverse kinematics (θ_1, θ_2, θ_3) is in a different colour and the lengths corresponding to that joint (L_1, L_2, L_3) are in a slightly different shade of that colour, the angle β is an auxiliary angle necessary for the calculations, the reference frames are in green and the vertical displacement between the frames (L_0) is in a slightly lighter green.

Let (x_{c0}, y_{c0}, z_{c0}) and (x_{c1}, y_{c1}, z_{c1}) be the endpoint coordinates of a leg in reference frame 0 and the endpoint coordinates of the same leg in reference frame 1, respectively. The coordinates in frame 1 can be derived from the coordinates in frame 0 by applying a rotation of θ_1 and a translation of L_1. This gives the relation

$$\begin{bmatrix} x_{c1} \\ y_{c1} \\ z_{c1} \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta_1 & \sin\theta_1 & 0 & 0 \\ -\sin\theta_1 & \cos\theta_1 & 0 & -L_1 \\ 0 & 0 & 1 & L_0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x_{c0} \\ y_{c0} \\ z_{c0} \\ 1 \end{bmatrix}. \tag{3.5}$$

The first joint's angle, θ_1 is the simplest to calculate. It can be done directly in frame 0, and it is defined by

$$\theta_1 = \arctan 2(-x_{c0}, y_{c0}).$$



(a) Top view. (b) Side view.

Figure 3.9: MORF's leg: top view and side view with angles, lengths and reference frames. The reference frames are in green, each angle has its own colour and the corresponding length has a different shade of the same colour. The angles θ_1, θ_2, θ_3 are the inverse kinematics parameters and β is an auxiliary angle.

The triangle formed by the centre of the second joint, the centre of the third joint and the endpoint of the leg allows us to find the equation for θ_3. This triangle has sides that measure L_2, L_3 and $\sqrt{y_{c1}^2 + z_{c1}^2}$.

Using the law of cosines with the triangle's angle that is $\theta_3 - \pi$ and knowing that $\cos(\theta_3 - \pi) = -\cos\theta_3$, it follows that

$$\theta_3 = \arccos\left(\frac{y_{c1}^2 + z_{c1}^2 - L_2^2 - L_3^2}{2L_2L_3}\right). \tag{3.6}$$

With the triangle that has vertices on the second joint, the endpoint and the point on the femur that creates a right angle, and with $\gamma = \theta_2 + \beta$, it is possible to write

$$\gamma = \arctan 2\left(L_3 \sin\theta_3, L_2 + L_3\cos\theta_3\right).$$

It is also straightforward to define $\beta = \arctan 2(-z_{c1}, y_{c1})$, which gives

$$\theta_2 = \arctan 2\left(L_3 \sin\theta_3, L_2 + L_3\cos\theta_3\right) - \arctan 2(-z_{c1}, y_{c1}). \tag{3.7}$$

Substituting the relation from Equation 3.5 in Equations 3.6 and 3.7, all of the kinematics parameters can be related to the endpoint coordinates in frame 0.

It is worth noting that the origin positions for θ_2 and θ_3 assumed for these calculations have an offset relative to the actual origin positions for the angles, so that is taken into consideration.

3.4.2 Inverse kinematics with neural networks

The neural networks have an input layer, an output layer and hidden layers in between. In this project, there are 3 inputs - the target position (x, y, z) - and 3 outputs - the leg angles $(\theta_1, \theta_2, \theta_3)$. Therefore, the network has 3 neurons in the input and output layers, as shown in Figure 3.10. The number of hidden layers and the number of neurons in those layers are not always the same and several combinations were tested.

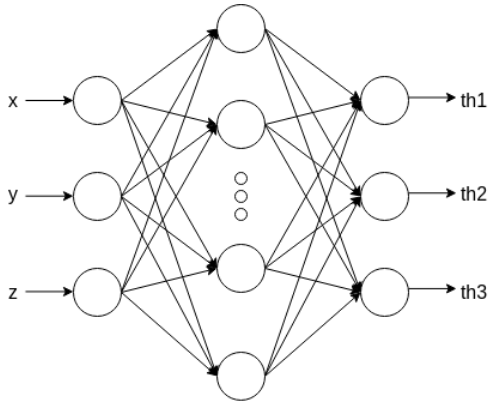


Figure 3.10: Graphical representation of a neural network.

The workspace of a leg is too big to have a single neural network - this was discovered empirically -, so the workspace is divided into several smaller blocks, each with their own neural network.

3.4.2.1 Data generation

The data used to train the neural networks was generated using three different methods: with IK equations, with the simulation and with the real robot.

The data generation with IK equations consists of randomly defining points inside a block and using the equations to obtain the angles. This is done for each smaller block, in order to have many pairs of inputs and outputs for each one. The workspace in this situation was defined as a parallelepiped in a leg's reference frame. However, not all the points inside this workspace can be reached and there are no angles that correspond to them. As such, those points are discarded.

In the data generation with the simulation, the workspace is explored by the robot in simulation. The angles are passed to the robot in simulation and the resulting foot position is returned by the simulation. The limits for the angles are defined so that there are no colisions with the rest of the robot. The angles' intervals are incrementally explored by going through θ_3's interval for each value of θ_2 and going through θ_2's interval for each value of θ_1. This method is more time consuming than using the IK equations, as the leg movement takes more time than the calculations.

The method of data generation with the real robot is very similar to the data generation with the simulation. The workspace is explored in the same way, but the foot position is obtained by using a tracking system, Motive [36]. As shown in Figures 3.11 and 3.12, the tracking system consists

of six cameras that are positioned around the laboratory and tracking spheres that are attached to the robot. The tracking spheres are identified by the cameras, and Motive has an interface for seeing their positions, as shown in Figure 3.13. For this project, the data was collected offline, that is, the tracking system and the robot recorded all the data for the positions and for the angles, respectively. The data was collected at the end of the workspace exploration from the two sources and it was then synchronised to generate input-output pairs.

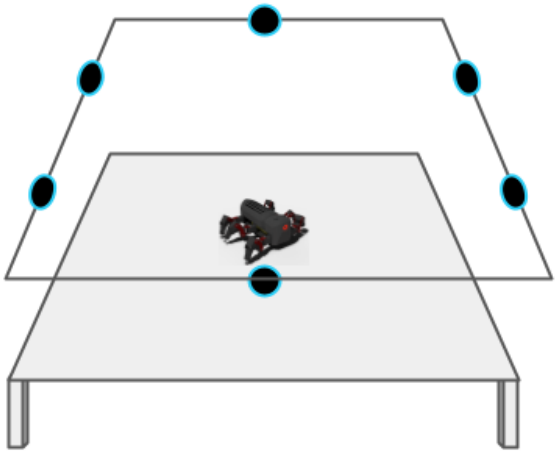


Figure 3.11: Representation of the tracking system.

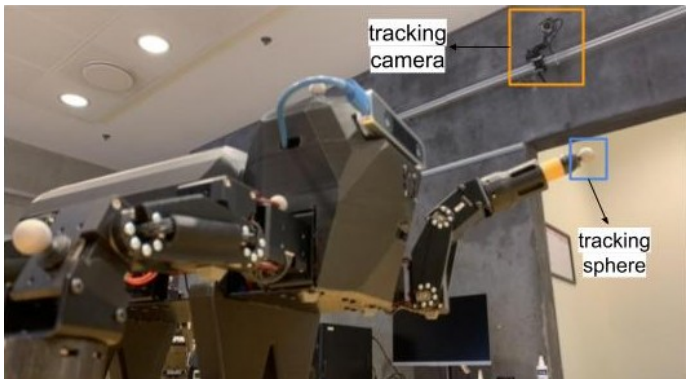
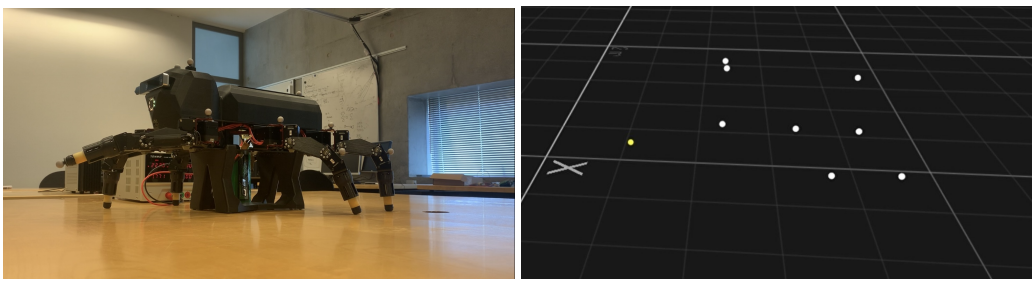


Figure 3.12: MORF with tracking sphere and a tracking camera.



(a) MORF in the real world. (b) MORF in the Motive interface.

Figure 3.13: Motive tracking. Real world robot vs. Motive interface.

3.4.2.2 Training and usage

The neural networks are implemented in C++ using the fann library.

The neural networks are trained using batch training with a learning rate of 0.1. Batch training consists of passing all of the training data through the network before updating the weights, instead of updating them for each input-output pair.

The activation function used for the neurons is the symmetric sigmoid function, also known as hyperbolic tangent. As Figure 3.14 shows, the relevant input values of the hyperbolic tangent are between -1 and 1, and this function only outputs values in that same range. As such, the inputs and outputs in the generated data have to be scaled. In order to give a small margin for the usage of the networks, the data was scaled to the interval $[-0.9, 0.9]$.

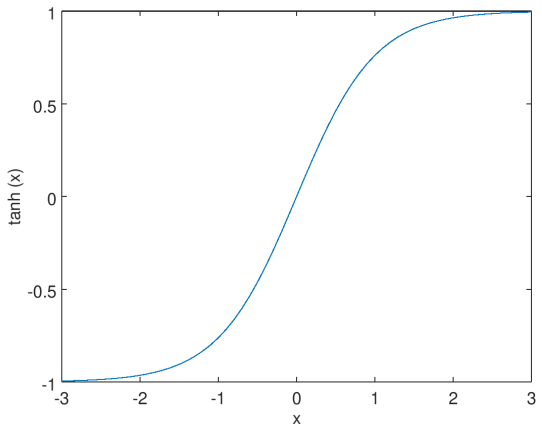


Figure 3.14: Hiperbolic tangent function.

The trained networks are saved in files that can be opened by the control program. When the control program starts running, if the neural networks are to be used, all of the neural networks are initialized in a 3 dimensional matrix of neural networks. This makes it so that the files only need to be accessed once instead of every time the joint angles are requested.

When the angles have to be determined, the program determines the block of the workspace to which the target position belongs. Then, the target position is scaled with parameters associated with the correct network. The scaled values are used as input to the neural network and it outputs the scaled angle values. The last step is to descale the output of the network to determine the correct angle values.

3.4.3 Algorithms

Algorithms 3 and 4 show the pseudocode for the calculation of the leg angles with the IK equations and with the neural networks, respectively.

Algorithm 3 Calculation of the leg angles with the IK equations.

function IK_equations($target$) ▷ Target coordinates are the inputs
$L_0 \leftarrow 0.017011$ ▷ Define the leg lengths
$L_1 \leftarrow 0.039805$
$L_2 \leftarrow 0.070075$
$L_3 \leftarrow 0.11542$
$offset2 \leftarrow 0.8$ ▷ Define the angle offsets
$offset3 \leftarrow -2.5$
$\theta_1 \leftarrow \arctan 2(-target.x, target.y)$ ▷ Calculate θ_1
$x_{c1} \leftarrow \cos(\theta_1) \times target.x + \sin(\theta_1) \times target.y$ ▷ Change reference frame
$y_{c1} \leftarrow -\sin(\theta_1) \times target.x + \cos(\theta_1) \times target.y - L_1$
$z_{c1} \leftarrow target.z + L_0$
$\theta_3 \leftarrow \arccos\left(\frac{y_{c1}^2 + z_{c1}^2 - L_2^2 - L_3^2}{2L_2L_3}\right)$ ▷ Calculate θ_3
$\theta_2 \leftarrow \arctan 2(z_{c1}, y_{c1}) + \arctan 2(L3\sin(\theta_3), L_2 + L_3\cos(\theta_3)) + offset_2$ ▷ Calculate θ_2
$\theta_3 \leftarrow \theta_3 + offset_3$ ▷ Add θ_3's offset
return $(\theta_1, \theta_2, \theta_3)$ ▷ Angles are the outputs

Algorithm 4 Calculation of the leg angles with the neural networks.

function IK_ann($target$) ▷ Target coordinates are the inputs
for $i \leftarrow 0$ to n_div **do** ▷ Go through the divisions in x to find the correct block
    if $target.x < x\_start$ and $target.x > x\_start + x\_step$ **then** ▷ Find the correct block for x
        $blockX \leftarrow i$ ▷ Save which block
        break ▷ Stop searching
    else
        $x\_start \leftarrow x\_start + x\_step$ ▷ Continue searching
    end if
end for
... ▷ Repeat lines 1 to 7 for y and z
$target\_ann \leftarrow ann[blockX][blockY][blockZ]$ ▷ Choose the correct ANN
$input \leftarrow \text{scale}(target\_ann, target)$ ▷ Scale the input for the ANN
$output \leftarrow \text{run}(target\_ann, input)$ ▷ Run the ANN
$angles \leftarrow \text{scale}(target\_ann, output)$ ▷ Descale the output from the ANN
return $angles$ ▷ Angles are the outputs

3.5 Code structure

The code developed for this project is available at https://github.com/lcgothen/IK_MORF. The code is organized in source files and libraries.

The source files are the main control code files (Subsection 3.3), the data generation code files (Subsection 3.4.2.1), the neural networks training code files (Subsection 3.4.2.2) and camera calibration code file (Subsection 3.1.3).

The library files are the coordinates files and the controller files. The coordinate files contain the coordinate frame transformations and the point class created for this project. The controller files contain classes for the angles with the different methods of calculating them, for the robot with the callbacks for receiving information and variables to store information, for the images

with the callbacks to receive them and the detection methods, and for the CPG with the cyclic behaviour and the walking method.

Chapter 4

Results

In order to test all of the different methods, 30 trials were performed in simulation for each method. The simulations were performed in the CoppeliaSim simulator, as previously discussed, with a circular green button with a 2 cm radius. The data collected from these tests are the duration of the calculations and, for the neural networks, the deviation from the expected values. The computer used to obtain the results runs Ubuntu 18.04.6 LTS, it has 16GB of RAM and its processor is the Intel® Core™ i5-9400F, which has 6 CPU cores and a processor base frequency of 2.90GHz.

The means of the duration and the deviation are compared between methods to determine which one has the best results. This assessment is corroborated by the p-values obtained with the Student's t-test. This test can be used to determine the statistical significance of the mean of one method being less than the mean of another method. The p-value limit used is 0.1, because, with a p-value of 0.1, there is only a 10% chance that one method does not produce a lower mean than the other method.

All of the methods are validated on the real robot also, but no data is collected from that. The real robot tests were performed in a well lit room and on a flat surface. The button was emulated by a paper with a printed green circle that contrasted with the background wall.

The videos of the real robot can be found here and the videos of the simulation can be found here, clearly named according to the method used. In these videos, MORF walks towards the button, assumes the stable position safely when it is near the button, and touches the button successfully. A video of the data generation with the real robot is available here.

4.1 Vision, movement and control

In simulation, the green button was detected successfully, as shown in Figure 4.1. The detected features are on the button and they are correctly matched between the two images. The estimated button position was also good enough, because, even though it was not always precisely in the buttons centre, it was always within the button.

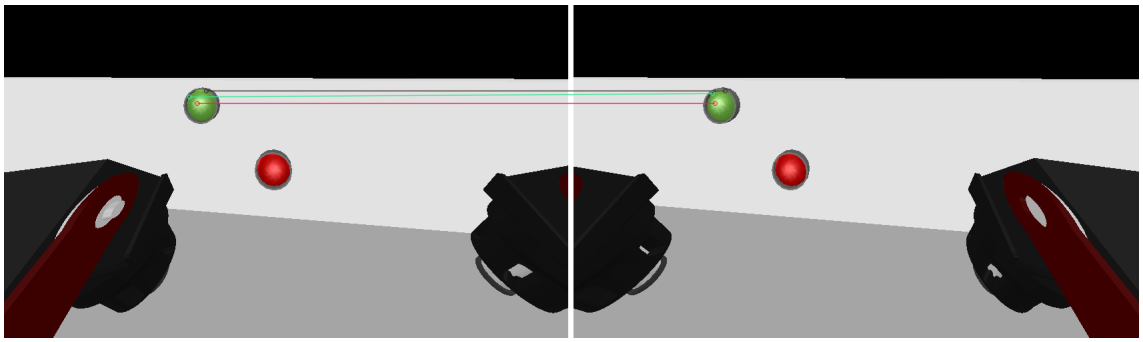
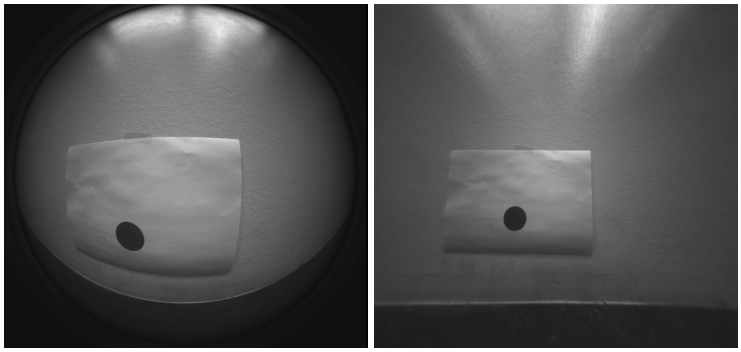


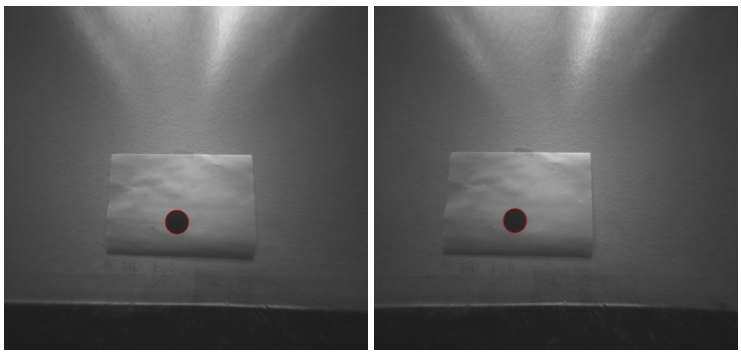
Figure 4.1: Button detection in simulation: the circles are the keypoints and they are connected by the lines to the matching keypoint in the image from the other camera.

In the real robot, the calibration was successful, as it was possible to undistort the images correctly - see Figure 4.2. The button was identified - see Figure 4.3 - and its position was estimated successfully.



(a) Before undistortion. (b) After undistortion.

Figure 4.2: Undistortion of an image from the fisheye camera.



(a) Image from the left camera. (b) Image from the right camera.

Figure 4.3: Button identified with the real camera, marked with a red circle.

The CPG functioned correctly and MORF walked towards the button. As the robot got close to the button, it stopped walking and achieved a stable position before attempting the task execution, as expected. This was validated in simulation and with the real robot.

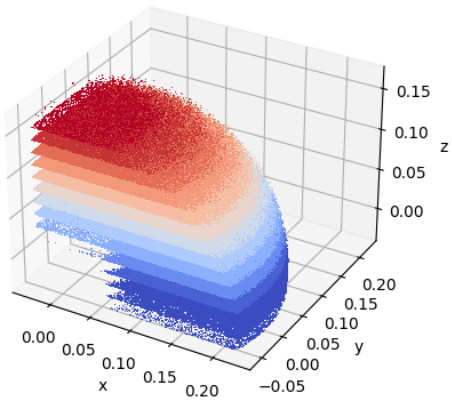
4.2 Task execution

The inverse kinematics equations method was successful in simulation and with the real robot, as shown in Figure 4.4. The videos mentioned at the beginning of this chapter can be consulted to view these successful results.

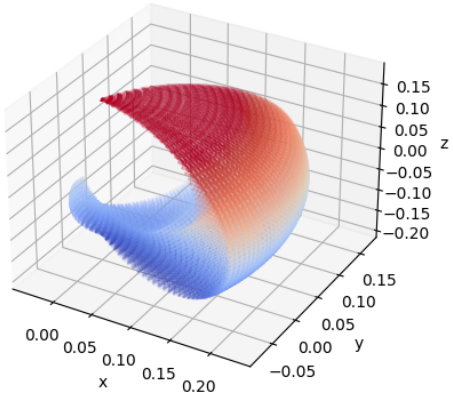
(a) Robot in simulation. (b) Real robot.

Figure 4.4: MORF touching the button with the IK equations.

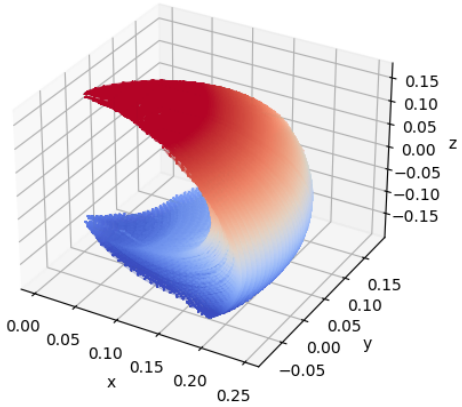
The three methods of data generation resulted in the workspaces shown in Figure 4.5. It is possible to observe that the workspace resulting from the inverse kinematic equations is much smaller than the others.



(a) With the IK equations.



(b) With the simulation.



(c) With the real robot.

Figure 4.5: Data generated for neural network training.

The workspace generated with the IK equations is smaller than the others, because the workspace limits for that method were defined with the expected possible positions for the button in mind, while the other methods go through almost all of the possible positions.

The positions obtained with the tracking software in the data generation method with the real robot have some noise, which introduces an error to the data and makes it impossible to perfectly synchronise the positions from the tracking and the angles from the robot. Table 4.1 shows this noisy data, in which the foot is unmoving.

Table 4.1: Coordinates obtained through the tracking software for the foot in an unmoving state.

Elapsed time (s)	**x**	**y**	**z**
0.000	0.192813	-0.0653421	-0.146529
0.005	0.192799	-0.0654019	-0.146548
0.010	0.192788	-0.0653494	-0.146541
0.015	0.192809	-0.0653202	-0.146515
0.020	0.192794	-0.0653459	-0.146503
0.025	0.192767	-0.065363	-0.146505
0.030	0.192792	-0.0653641	-0.146489
0.035	0.192778	-0.0653737	-0.146529
0.040	0.192811	-0.0653912	-0.146531
0.045	0.192796	-0.0653823	-0.146567
0.050	0.192825	-0.0653735	-0.146556

The tests with a single neural network for the whole workspace generated with the IK equations resulted in many failures to hit the button, as shown in Figure 4.6. The results presented in that figure were obtained with five configurations: 1 hidden layer with 5 neurons, 1 hidden layer with 20 neurons, 1 hidden layer with 40 neurons, 2 hidden layers with 20 neurons each and 2 hidden layers with 40 neurons each.

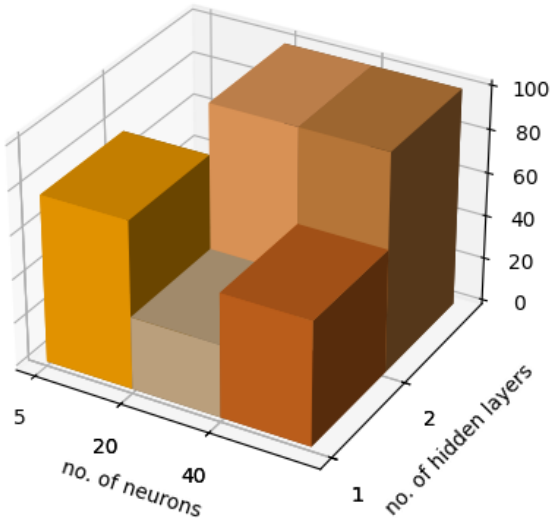


Figure 4.6: Percentage of failures with a single neural network.

The division of the workspace for the data generated with the IK equations is shown by Figure 4.7. Because the workspace generated with the simulation is nearly double in the z dimension, that workspace is divided into double the number of blocks in that dimension.

As shown in Figure 4.5c, the dimensions of the real workspace are very similar to the dimensions of the simulation workspace, so the division method is the same. To make the duration of the methods more correctly comparable, the lower blocks of the workspace generated in simulation - the part not present in the workspace generated with the IK equations - were not taken into account when searching for the desired neural network.

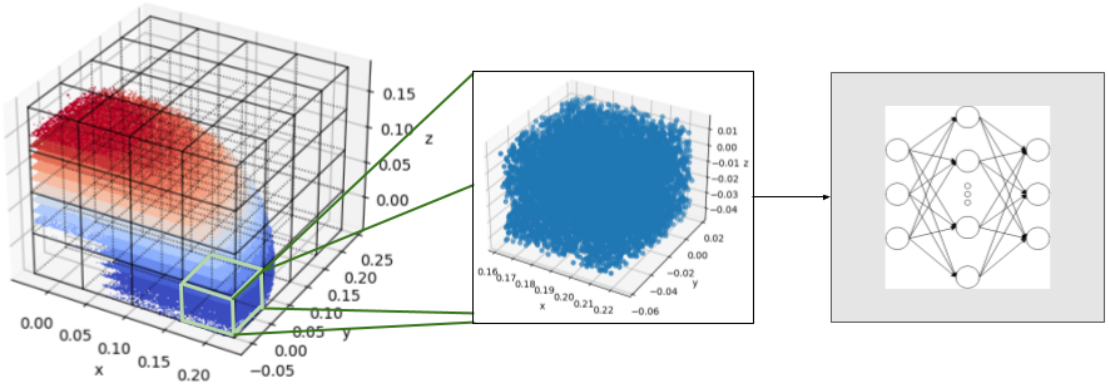


Figure 4.7: Division of the workspace generated with the IK equations.

With the division of the workspace, there were 3 configurations tested for the neural networks: 1 hidden layer with 5 neurons, 1 hidden layer with 10 neurons and 2 hidden layers with 10 neurons each.

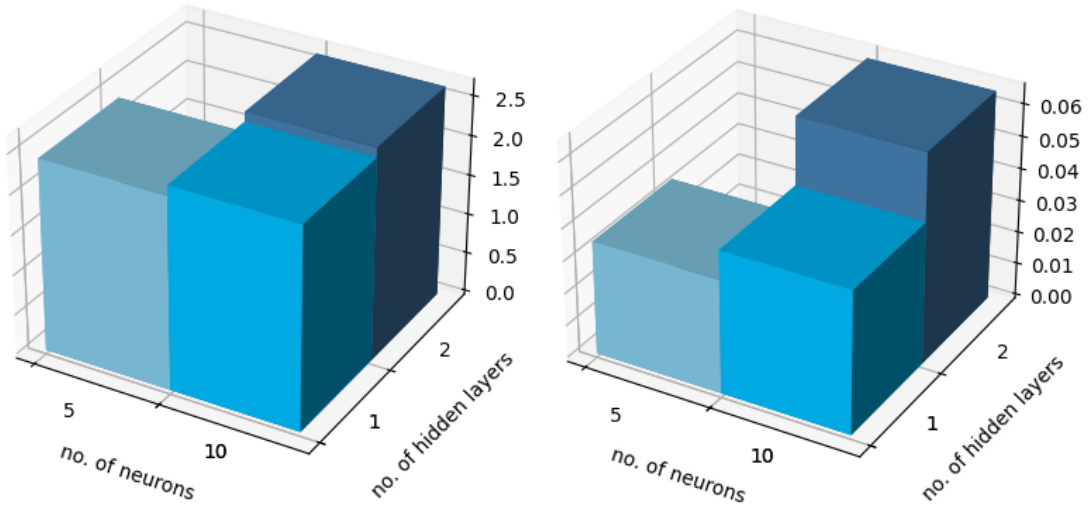
As Figure 4.8 shows, the best configuration for the case of 4 divisions of the workspace generated with the IK equations is the one with 1 hidden layer with 5 neurons, as it has the lowest calculation time and deviation from the IK equations.

Table 4.2 presents the p-values for determining the statistical significance of the mean values of the configuration of 1 hidden layer with 5 neurons being less than the mean values of the other two configurations. The only p-value that is over 0.1 is for the deviation comparing with the configuration of 2 layers with 10 neurons, which denotes that the mean duration of the chosen configuration is not significantly lower than that one. Because the other three p-values show that the other comparisons are relevant, the choice of best configuration is corroborated.

Table 4.2: P-values for corroborating the configuration chosen as the best for 4 divisions of the workspace generated with the IK equations.

	1 layer with 10 neurons	**2 layers with 10 neurons**
duration	0.00445	1.62×10^{-6}
deviation	0.0253	0.158

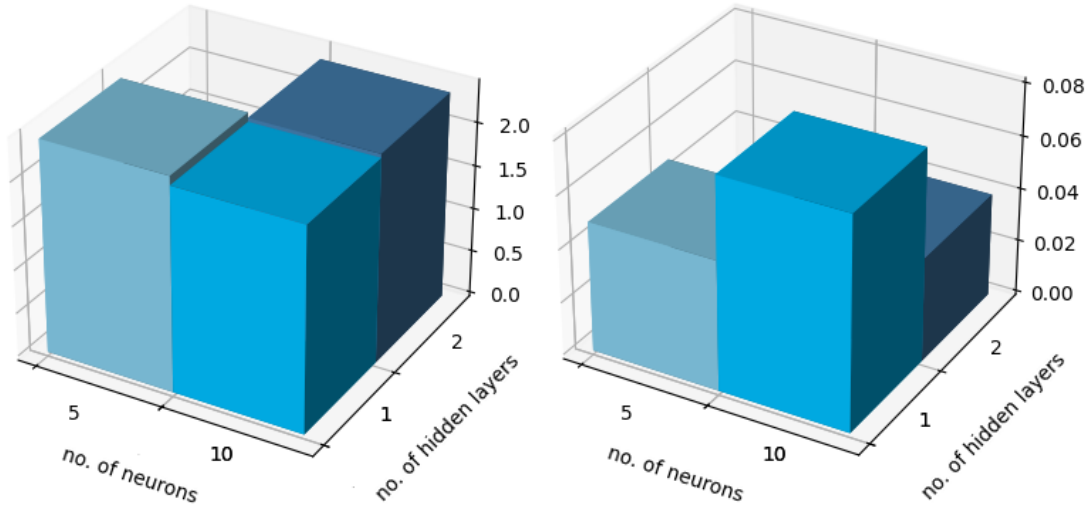
In the case of 5 divisions of the workspace generated with the IK equations, shown in Figure 4.9, it is not as straightforward to determine the best configuration. The lowest average deviation



(a) Average calculations duration (ns). (b) Average deviation from equations (rad).

Figure 4.8: Results for 4 divisions of the workspace generated with the IK equations.

from the IK equations is obtained with 2 hidden layers with 10 neurons each, but the lowest average duration of the calculations is obtained with 1 hidden layer with 10 neurons. However, the difference between the average durations of those two configurations is negligible, while the average deviation from the IK equations of the neural networks with 2 hidden layers with 10 neurons each is less than half of the average deviation of the neural networks with 1 hidden layer with 10 neurons. Therefore, the overall best performing configuration is the one with 2 hidden layers with 10 neurons each.



(a) Average calculations duration (ns). (b) Average deviation from equations (rad).

Figure 4.9: Results for 5 divisions of the workspace generated with the IK equations.

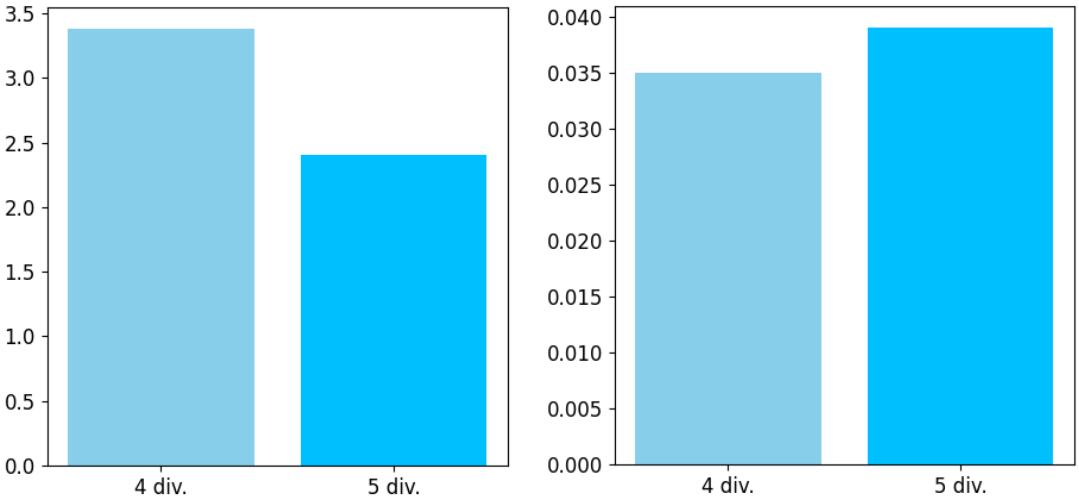
Table 4.3 presents the p-values for determining the statistical significance of the mean values of the configuration of 2 hidden layer with 10 neurons being less than the mean values of the

other two configurations. In this case, only one p-value is under 0.1, but two others are very near that value, so their means can also be counted as significantly lower than the mean of the chosen configuration. The one p-value that is very high makes sense, as the mean duration for 1 layer with 10 neurons is slightly lower than the mean duration for the chosen configuration.

Table 4.3: P-values for corroborating the configuration chosen as the best for 5 divisions of the workspace generated with the IK equations.

	1 layer with 5 neurons	**1 layer with 10 neurons**
duration	0.111	0.857
deviation	0.107	0.0979

Figure 4.10 compares the best result for the 4 divisions of this workspace - 1 hidden layer with 5 neurons - to the best result for the 5 divisions of this workspace - 2 hidden layers with 10 neurons. It is possible to see that the deviation from the neural networks is lower in the case of 4 divisions, but the duration of the calculations is lower in the case of 5 divisions. The p-value of the difference between the deviations is 0.222 and the p-value of the difference between durations is 0.298. This means that neither difference is very relevant, statistically, but the difference between deviations is slightly more so. A lower deviation from the equations means that the configuration is more robust and consistent. Therefore, the best configuration for the neural networks method of generating data with the equations is to have 4 divisions of the workspace and neural networks with 1 hidden layer with 5 neurons.

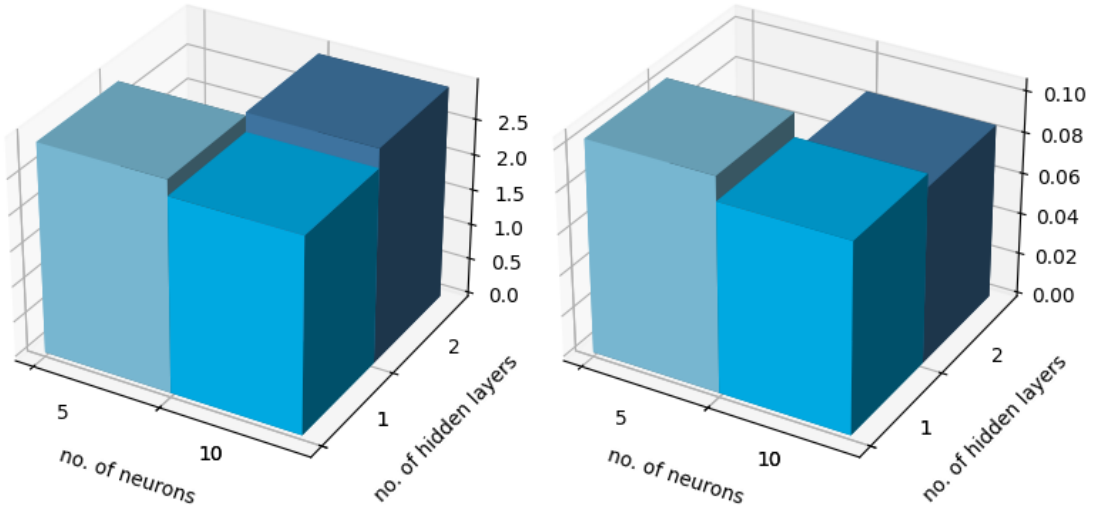


(a) Average calculations duration (ns). (b) Average deviation from equations (rad).

Figure 4.10: Best results for the workspace generated with the IK equations.

Figure 4.11 presents the results from the case of 4 divisions of the workspace generated with the simulation. This case has the same situation as the last one, as the configuration with the smallest average deviation from the IK equations is not the same as the configuration with the lowest average calculations duration. The difference in average calculations duration of these two

configurations is less than 0.5 ns, and the difference in average deviation from the IK equations is less than 0.02 rad. With these very small differences, the average deviation from the IK equations can be considered more important, which makes the best configuration the one with 2 hidden layers with 10 neurons each.



(a) Average calculations duration (ns). (b) Average deviation from equations (rad).

Figure 4.11: Results for 4 divisions of the workspace generated with the simulation.

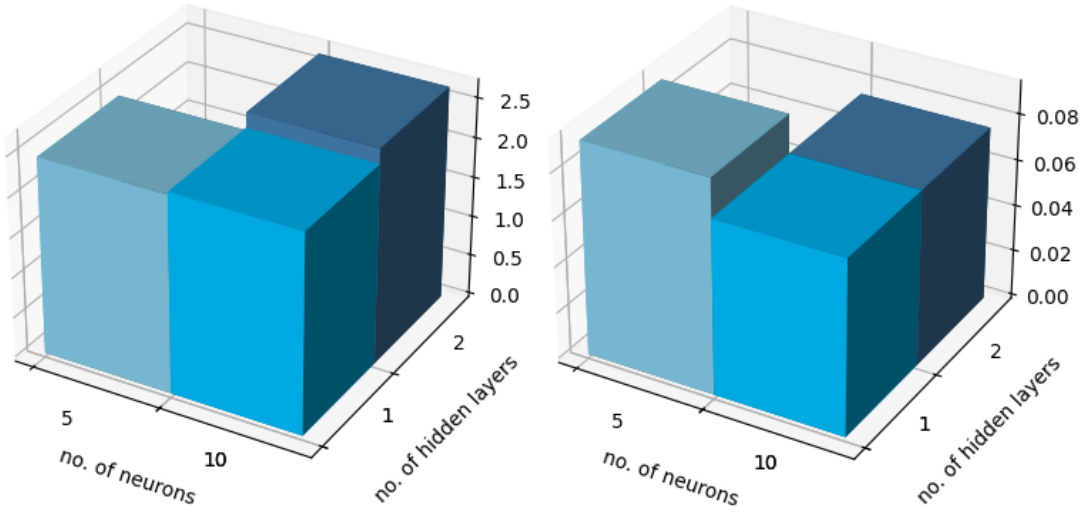
Table 4.4 presents the p-values for determining the statistical significance of the mean values of the configuration of 2 hidden layers with 10 neurons being less than the mean values of the other two configurations. The minimal differences mentioned previously justify the fact that only one of the p-values is under 0.1. However, it is clear that the deviation differences are more significant, as those p-values are considerably lower than the p-values for the duration differences.

Table 4.4: P-values for corroborating the configuration chosen as the best for 4 divisions of the workspace generated with the simulation.

	1 layer with 5 neurons	**1 layer with 10 neurons**
duration	0.769	1.00
deviation	0.0454	0.257

The case of 5 divisions of the workspace yielded the results shown in Figure 4.12. The average deviations from the IK equations of the two configurations with 10 neurons are the same and lower than of the configuration with 5 neurons. The difference between the two lowest average calculations durations is negligible and one of them is the with the configuration of 1 hidden layer with 10 neurons, which makes this the best one. However, it is worth noting that this division of the workspace caused the program to crash up to 9.01% of the time, because the desired foot position was in a block without data and, consequently, without a neural network.

Table 4.5 presents the p-values for determining the statistical significance of the mean values of the configuration of 1 hidden layer with 10 neurons being less than the mean values of the



(a) Average calculations duration (ns). (b) Average deviation from equations (rad).

Figure 4.12: Results for 5 divisions of the workspace generated with the simulation.

other two configurations. These p-values corroborate that the differences that were considered negligible are indeed not significant, as they resulted in high p-values. Meanwhile, the differences that were considered for making the decision have p-values of less than 0.1, so the means of the chosen configuration are significantly lower than the means of these other configurations.

Table 4.5: P-values for corroborating the configuration chosen as the best for 5 divisions of the workspace generated with the simulation.

	1 layer with 5 neurons	**1 layer with 10 neurons**
duration	0.802	0.000766
deviation	0.0204	0.503

With these results, it is clear that it is best to have 4 divisions of the workspace generated in simulations, as having 5 divisions can result in the program crashing. the best configuration for the neural networks method of generating data with the equations is to have 4 divisions of the workspace and neural networks with 2 hidden layers with 10 neurons.

With the data generation with the real robot, none of the cases tested were able to touch the button every time. Figure 4.13 shows the percentage of times the robot failed to touch the button with each network configuration for two quantities of workspace divisions. In this case, the number of workspace divisions used were 3 and 4, instead of the 4 and 5 used previously, because the case of 4 divisions suffered from the same problem as the case of 5 divisions of the workspace generated with the simulation - it caused the program to fail several times, because the desired foot position was in a block without data and, consequently, without a neural network. The choice of testing with 3 divisions of the workspace generated with the real robot solved this problem.

It is clear from Figure 4.13 that the neural networks configuration of 1 layer with 10 neurons is the one that yields the best results for both quantities of divisions of the workspace. Even though

the case of 4 divisions has a lower rate of failures than the case of 3 divisions, it is best that the program not crash, so the best solution in this situation is using a configuration of 1 layer with 10 neurons for 3 divisions of the workspace.

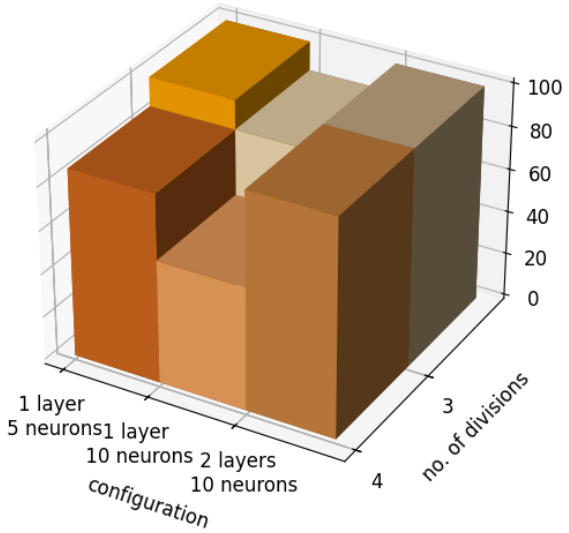


Figure 4.13: Percentage of failures with networks trained with data generated with the real robot.

Chapter 5

Discussion

5.1 Task execution

Recapitulating what was presented in Chapter 4, Table 5.1 shows the configuration that yielded the best results for each case of neural networks methods.

Table 5.1: Best configuration for each case of neural networks methods.

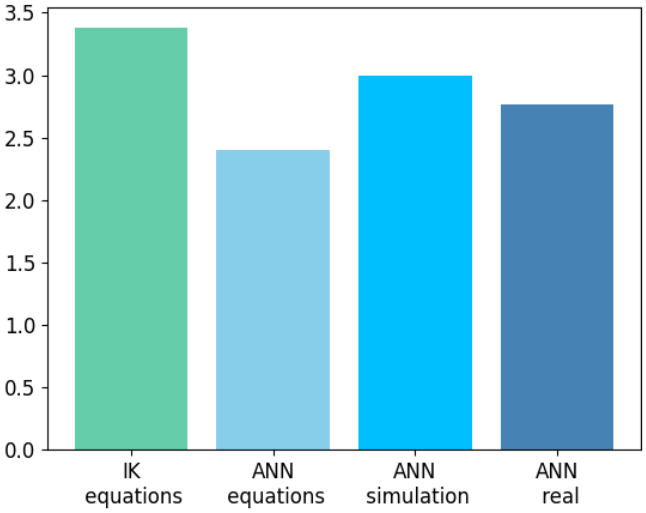
Data generation method	Number of divisions	Network configuration
IK equations	4	1 hidden layer 5 neurons
Simulation	4	2 hidden layers 10 neurons
Real robot	3	1 hidden layer 10 neurons

The average calculations duration of those cases and of the IK equations are presented in Figure 5.1a. This figure clearly shows that using neural networks is faster than using the equations. However, Figure 5.1b shows that the IK equations are clearly more accurate.

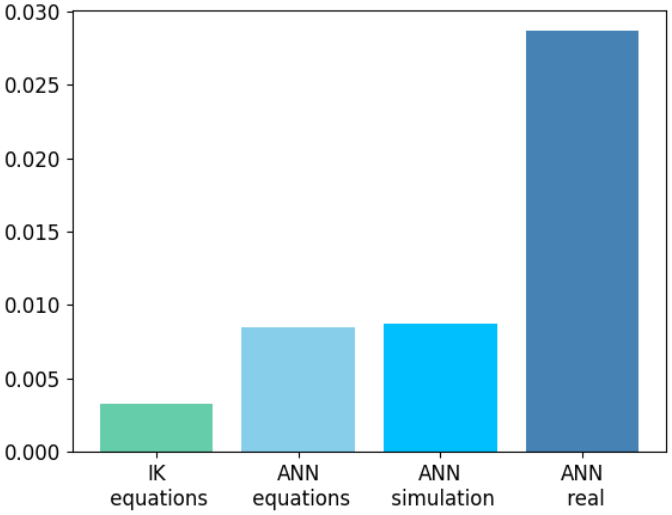
The reason for the distance observed in Figure 5.1b with the IK equations is the error in the computer vision. This error is also present when using the neural networks, as the computer vision part of the solution is completely independent of the task execution. Therefore, it is more pertinent to analyse the deviation of the neural networks from the IK equations, instead of the distance from the button centre.

The different methods have different strengths and weaknesses. The IK equations are specific to robots with the same leg configuration as MORF, even though they are adaptable to different lengths in the legs. This is also the most accurate method, although it isn't the fastest way of doing the calculations.

The neural networks are generally faster at the calculations than the IK equations, but they need more preparation time than the IK equations, as the data has to be generated and then the networks have to be trained.



(a) Average calculations duration (ns).



(b) Average distance from the button's centre (m).

Figure 5.1: Best result of each case.

The method of generating data with the IK equations ends up being more a proof of concept than a useful solution, as this method still needs the IK equations to be written. Therefore, as with the IK equations method, this method is specific to robots with the same leg configuration as MORF. Having this in mind, this method is most useful for situations where the IK equations are easily discovered and the duration of the calculations is a very important factor.

The method of generating data with the simulation is very useful, as it eliminates the need for the IK equations. Because the data is generated by exploration of the robot's workspace, this method is not specific to just one robot configuration, but it can be applied to any type with minor adaptations. The only adaptations that could be necessary are in the network configuration, as the number of outputs might not be the same if the number of joints in the leg is different. However, this method of data generation is time consuming for the quantity of data generated, especially

comparing to the data generated with the IK equations.

In theory, the method of generating data with the real robot has the same usefulness as the previous method, with the added advantage of the data not being influenced by possible inaccuracies in the simulation. However, the results clearly show that this method is not well-developed enough for actual use yet.

The main problem with the method of data generation with the real robot was the tracking system. The data had to be collected offline - the tracking system recorded the leg movements and the data was exported at the end -, so the foot position data from the tracking system had to, afterwards, be synchronised with the angles data from the robot. This is very difficult to do perfectly because of the noise present in the tracking data and because the sampling times on the robot and on the tracking system may not be exactly the same. This problem can be solved by developing an online data collection program from both the tracking system and the robot, as the simulataneous data collection would eliminate the need for synchronisation.

However, as proven by the successful results of the data generation in simulation, the need for calculating the IK equations has been eliminated with the neural networks. Even though the IK equations are generally more accurate, this method is fully functional and it makes this inverse kinematic control more adaptable to other robots.

5.2 Code adaptability

The structure of the code makes it easy to adapt to other types of tasks. Most of the files can be used without having to be modified, specifically the data generation code files, the neural networks training code files, the camera calibration code file and the coordinates files.

The controller files would only have to be adapted in the vision functions and possibly the walking functions. The vision functions are very specific to identifying a button and the walking functions adjust the speed of the robot to the distance from the target, so if the target is something different or if the robot should adjust the speed in another way, these functions must be modified. The main control code files are the ones that need more work to adapt, as that is where the specfic task execution is.

The code is, however, very specific to MORF's software. The ROS organisation - topics and data structure - is the one used by MORF, and the coordinate system is also specific to MORF.

For minor physical changes, the code can be adapted. The IK equations can be adapted to different leg lengths by changing the values of the corresponding variables. The neural networks would need no adaptation except in the data generation method that uses the IK equations, where the IK equations need to be adapted.

5.3 Comparison to other works

Table 5.2 presents a comparison of the features developed in this project and the features of other state of the art works with hexapod robots. It is worth noting that most works with hexapods focus

on locomotion and the adaptability of movement that a hexapod robot has, specifically walking in difficult terrain or vertical climbing. Only one of the presented projects has the feature of task execution and even that one does not use ANN (Artificial Neural Networks) or VGM (Vision Guided Manipulation).

Table 5.2: Comparison between this project's and other works' features.

Title	**Locomotion method**		**Locomotion type**			**Task execution**		
	CPG	IK	Horizontal	Vertical	Unstructured terrain	IK eqs.	ANN	VGM
Inverse kinematic control of a hexapod robot (this project)	✓	✗	✓	✗	✗	✓	✓	✓
Design and Simulation of Special Hexapod Robot with Vertical Climbing Ability [4]	-	-	-	✓	✓	✗	✗	✗
Design and Configuration of a Hexapod Walking Robot [3]	✗	✓	✓	✗	✓	✗	✗	✗
Terrain adaptation gait algorithm in a hexapod walking robot [5]	✗	✓	✓	✗	✓	✗	✗	✗
Enhancing adaptability with local reactive behaviors for hexapod walking robot via sensory feedback integrated central pattern generator [6]	✓	✗	✓	✗	✓	✗	✗	✗
Neural coupled central pattern generator based smooth gait transition of a biomimetic hexapod robot [7]	✓	✓	✓	✗	✗	✗	✗	✗
Object carrying of hexapod robots with integrated mechanism of leg and arm [8]	✗	✓	✓	✗	✗	✓	✗	✗

Chapter 6

Conclusion

The main objective of this project was to allow MORF to perform simple tasks with inverse kinematic control. This was achieved by implementing two methods of inverse kinematic control: inverse kinematics equations and artificial neural networks.

The inverse kinematic equations were obtained by analysing the geometry of MORF's legs. They are specific to the leg configuration that MORF has, but they can be applied to robots that have other leg lengths.

The neural networks need data to train, which was created with three different methods: data generation with the inverse kinematics equations, data generation in simulation and data generation with the real robot. The data generation with the inverse kinematics equations was done by randomly defining points inside the robot's workspace and applying the equations to obtain the corresponding angles. The data generation in simulation was done by defining the leg angles to explore the workspace, while the simulation returned the corresponding foot position. The data generation with the real robot was done in the same way as the data generation in simulation, except that the foot position was obtained with a tracking system.

The project was implemented in the simulation for data collection and in the real robot for validation. MORF was able to identify a button, walk towards it and, as it got near the button, assume a stable position for manipulation with the two front legs. The simulated camera was different from the real camera, because of simulator constraints. As such, ORB detection was used in simulation and blob detection was used on the real robot. The real camera is of the fisheye type, so the images had to be undistorted, which was done successfully.

MORF successfully performed the task of pressing a button, both with the inverse kinematics equations and with the artificial neural networks. The data generation method with the real robot needs further development, but the method of data generation in simulation had very good results. As such, the need for the inverse kinematics equations has been eliminated from the inverse kinematic control of the robot, making it easier to apply to different robots.

6.1 Future Work

There is still work that can be done as a continuation of this project.

The data generation method with the real robot is the most important thing that needs further work. Specifically, live data collection should be implemented. This means that the data from the tracking system and from the robot are collected simultaneously and don't need to be synchronised later.

Even though the neural networks used achieved good results, they can still be optimised to get even better results. Different activation functions can be implemented (e.g. ReLU) and other networks configurations (the number of hidden layers and the number of neurons in those layers) can be tested.

Also regarding the neural networks, the workspace should be divided into fewer blocks to find the limit that minimises the number of neural networks that are used, while still producing good results.

Lastly, the code is made to be adaptable to other tasks, so doing so can further expand MORF's usefulness.

Appendix A

Instructions for code usage

This project contains the tools to perform inverse kinematic control on MORF, either with equations or with neural networks. This includes:

- control code with inverse kinematic equations or with neural networks
  - for simulation
  - for the real robot
- data generation methods for the neural networks
  - with the inverse kinematics equation
  - with the simulation
  - with the real robot
- code for neural network training

Note: the executables are in /IK_MORF/catkin_ws/devel/lib/morf_ik

A.1 Dependencies

This code is dependent on:

- fann library (this page has instructions for installation)
- CoppeliaSim (download from here)
- ROS (this page has instructions for installation)
- Ansible (install with the command `sudo apt install ansible`)

Note: make sure that the CMakeLists.txt (/IK_MORF/catkin_ws/src/morf_ik/CMakeLists.txt) is according to your machine. This means that you should change the path in line 11 to your installation of the fann library and the path in line 12 to you installation of CoppeliaSim.

A.2 Control code in simulation

- Start a ROS core with the command `roscore`
- Open CoppeliaSim
- Open the simulation scene (File > Open scene... > MORF.ttt)
- execute `./main_sim <TYPE> <N_TRIALS>` in the terminal
  - TYPE: 0 is using the equations and 1 is using the neural networks
  - N_TRIALS: number of trials to be executed

Note: the neural networks used can be changed by changing the file **ann_config.yml**. For neural networks trained with data generated with the equations, use `reverse: 1`. For neural networks trained with data generated with the simulation or the real robot, use `reverse: 0`.

A.3 Control code for the real robot

- Connect to morf's network
- Transfer the code by executing `ansible-playbook -i inventory morf_transfer_controller.yml`
- If using the neural networks, execute also `ansible-playbook -i inventory morf_transfer_nn.yml`
- SSH into MORF and execute the command `cd /home/morf-one/workspace/gorobots-mthor/projects/morf/real/catkin_ws/src/morf_controller/bin`
- Execute `./morf_controller_real <TYPE>`
  - TYPE: 0 is using the equations and 1 is using the neural networks

Note: the neural networks used can be changed by changing the file **ann_config.yml**. For neural networks trained with data generated with the equations, use `reverse: 1`. For neural networks trained with data generated with the simulation or the real robot, use `reverse: 0`.

A.4 Data generation for neural networks

A.4.1 Inverse kinematics equations

- Execute `./gen_data_eqs`

A.4.2 Simulation

- Start a ROS core with the command `roscore`
- Open CoppeliaSim
- Open the simulation scene (File > Open scene... > MORF_gen_data.ttt)
- Execute `./gen_data_sim`
- When that is done, execute `./sort_data_sim`

Note: go to line 100 of the file **sort_data.cpp** (/IK_MORF/catkin_ws/src/morf_ik/src/sort_data.cpp) to change the number of divisions of the workspace (variable named **div**). Do `catkin_make` in the catkin_ws folder in the terminal to update the executable.

A.4.3 Real robot

- Connect to morf's network
- Transfer the code by executing `ansible-playbook -i inventory morf_transfer_genData.yml`
- If using the neural networks, execute also `ansible-playbook -i inventory morf_transfer_nn.yml`
- SSH into MORF and execute the command `cd /home/morf-one/workspace/gorobots-mthor/projects/morf/real/catkin_ws/src/morf_controller/bin`
- Open another terminal and repeat the last step
- Start recording in the tracking software
- Execute `./save_gen_data_real` in one terminal
- Execute `./gen_data_real` in the other terminal
- When that is done, execute `scp morf-one@192.168.0.1: /workspace/gorobots-mthor/projects/morf/real/catkin_ws/src/morf_controller/bin/real_data/output.data /<path>/IK_MORF/catkin_ws/devel/lib/morf_ik`
- Execute `./process_real_data`
- Execute `./sort_data_real`

A.5 Neural networks training

A.5.1 Data generated with inverse kinematics equations

- Execute `./nn_eqs`

Note: go to the file **nn.cpp** (/IK_MORF/catkin_ws/src/morf_ik/src/nn.cpp) to change the number of divisions of the workspace (variable named **div**) and the training parameters. Do `catkin_make` in the catkin_ws folder in the terminal to update the executable.

A.5.2 Data generated with simulation

- Execute `./nn_sim`

Note: go to the file **nn_sim.cpp** (/IK_MORF/catkin_ws/src/morf_ik/src/nn_sim.cpp) to change the number of divisions of the workspace (variable named **div**) and the training parameters. Do `catkin_make` in the catkin_ws folder in the terminal to update the executable.

A.5.3 Data generated with the real robot

- Execute `./nn_real`

Note: go to the file **nn_real.cpp** (/IK_MORF/catkin_ws/src/morf_ik/src/nn_sim.cpp) to change the number of divisions of the workspace (variable named **div**) and the training parameters. Do `catkin_make` in the catkin_ws folder in the terminal to update the executable.

References

[1] M. Thor. MORF - Modular Robot Framework. Master's thesis, SDU, 02 2019.

[2] G. Jianhua. Design and Kinematic Simulation for Six-DOF Leg Mechanism of Hexapod Robot. In *2006 IEEE International Conference on Robotics and Biomimetics*, pages 625–629, 2006. doi:10.1109/ROBIO.2006.340272.

[3] J. Bo, C. Cheng, L. Wei, and L. Xiangyun. Design and Configuration of a Hexapod Walking Robot. In *2011 Third International Conference on Measuring Technology and Mechatronics Automation*, volume 1, pages 863–866, 2011. doi:10.1109/ICMTMA.2011.216.

[4] Z. Zhou and X. Zhu. Design and simulation of special hexapod robot with vertical climbing ability. In *2020 IEEE 5th Information Technology and Mechatronics Engineering Conference (ITOEC)*, pages 1–1, 2020. doi:10.1109/ITOEC49072.2020.9141755.

[5] Y. Isvara, S. Rachmatullah, K. Mutijarsa, D. E. Prabakti, and W. Pragitatama. Terrain adaptation gait algorithm in a hexapod walking robot. In *2014 13th International Conference on Control Automation Robotics Vision (ICARCV)*, pages 1735–1739, 2014. doi:10.1109/ICARCV.2014.7064578.

[6] H. Yu, H. Gao, and Z. Deng. Enhancing adaptability with local reactive behaviors for hexapod walking robot via sensory feedback integrated central pattern generator. *Robotics and Autonomous Systems*, 124:103401, 2020. doi:https://doi.org/10.1016/j.robot.2019.103401.

[7] C. Bal. Neural coupled central pattern generator based smooth gait transition of a biomimetic hexapod robot. *Neurocomputing*, 420:210–226, 2021. URL: https://www.sciencedirect.com/science/article/pii/S0925231220313187, doi:https://doi.org/10.1016/j.neucom.2020.07.114.

[8] H. Deng, G. Xin, G. Zhong, and M. Mistry. Object carrying of hexapod robots with integrated mechanism of leg and arm. *Robotics and Computer-Integrated Manufacturing*, 54:145–155, 2018. URL: https://www.sciencedirect.com/science/article/pii/S0736584517301126, doi:https://doi.org/10.1016/j.rcim.2017.11.014.

[9] B. Siciliano and O. Khatib. *Springer Handbook of Robotics*. Springer, 2008.

[10] J. Sun, J. Ren, Y. Jin, B. Wang, and D. Chen. Hexapod robot kinematics modeling and tripod gait design based on the foot end trajectory. In *2017 IEEE International Conference on Robotics and Biomimetics (ROBIO)*, pages 2611–2616, 2017. doi:10.1109/ROBIO.2017.8324813.

[11] E. H. Hasnaa and B. Mohammed. Planning tripod gait of an hexapod robot. In *2017 14th International Multi-Conference on Systems, Signals Devices (SSD)*, pages 163–168, 2017. doi:10.1109/SSD.2017.8166964.

[12] K. Priandana, A. Buono, and Wulandari. Hexapod leg coordination using simple geometrical tripod-gait and inverse kinematics approach. In *2017 International Conference on Advanced Computer Science and Information Systems (ICACSIS)*, pages 35–40, 2017. doi:10.1109/ICACSIS.2017.8355009.

[13] M.T. Aju and A. Napolean. Modeling and Simulation of Hexapod Kinematics with Central Pattern Generator. *IAES International Journal of Robotics and Automation (IJRA)*, 5:72, 06 2016. doi:10.11591/ijra.v5i2.pp72-86.

[14] G.C. Haynes and A.A. Rizzi. Gaits and gait transitions for legged robots. In *Proceedings 2006 IEEE International Conference on Robotics and Automation, 2006. ICRA 2006.*, pages 1117–1122, 2006. doi:10.1109/ROBOT.2006.1641859.

[15] A.J. Ijspeert. Central pattern generators for locomotion control in animals and robots: A review. *Neural Networks*, 21(4):642–653, 2008. doi:10.1016/j.neunet.2008.03.014.

[16] H. Rostro-Gonzalez, P.A. Cerna-Garcia, G. Trejo-Caballero, C.H. Garcia-Capulin, M.A. Ibarra-Manzano, J.G. Avina-Cervantes, and C. Torres-Huitzil. A CPG system based on spiking neurons for hexapod robot locomotion. *Neurocomputing*, 170:47–54, 2015. Advances on Biological Rhythmic Pattern Generation: Experiments, Algorithms and Applications Selected Papers from the 2013 International Conference on Intelligence Science and Big Data Engineering (IScIDE 2013) Computational Energy Management in Smart Grids. URL: https://www.sciencedirect.com/science/article/pii/S0925231215008784, doi:https://doi.org/10.1016/j.neucom.2015.03.090.

[17] G. Ren, W. Chen, S. Dasgupta, C. Kolodziejski, F. Wörgötter, and P. Manoonpong. Multiple chaotic central pattern generators with learning for legged locomotion and malfunction compensation. *Information Sciences*, 294:666–682, 2015. Innovative Applications of Artificial Neural Networks in Engineering. URL: https://www.sciencedirect.com/science/article/pii/S0020025514005192, doi:https://doi.org/10.1016/j.ins.2014.05.001.

[18] K. Dawson-Howe. *A practical introduction to computer vision with opencv*. Wiley, 2014.

[19] OpenCV feature detection. [Online] https://docs.opencv.org/3.4/db/d27/tutorial_py_table_of_contents_feature2d.html. Accessed: march 2022.

[20] OpenCV blob detection. [Online] https://docs.opencv.org/3.4/d0/d7a/classcv_1_1SimpleBlobDetector.html. Accessed: march 2022.

[21] D. G. Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60:91–110, 2004. doi:https://doi.org/10.1023/B:VISI.0000029664.99615.94.

[22] H. Bay, T. Tuytelaars, and L. Van Gool. Surf: Speeded up robust features. In Aleš Leonardis, Horst Bischof, and Axel Pinz, editors, *Computer Vision – ECCV 2006*, pages 404–417, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[23] E. Rosten and T. Drummond. Machine learning for high-speed corner detection. volume 3951, 07 2006. doi:10.1007/11744023_34.

[24] E. Rublee, V. Rabaud, K. Konolige, and G. Bradski. Orb: An efficient alternative to sift or surf. In *2011 International Conference on Computer Vision*, pages 2564–2571, 2011. doi:10.1109/ICCV.2011.6126544.

[25] Blob detection. [Online] https://en.wikipedia.org/wiki/Blob_detection. Accessed: march 2022.

[26] Merriam-Webster.com Dictionary. "Simulation.". [Online] https://www.merriam-webster.com/dictionary/simulation. Accessed: january 2022.

[27] M. Santos Pessoa de Melo, J. Gomes da Silva Neto, P. Jorge Lima da Silva, J. M. X. Natario Teixeira, and V. Teichrieb. Analysis and Comparison of Robotics 3D Simulators. In *2019 21st Symposium on Virtual and Augmented Reality (SVR)*, pages 242–251, 2019. doi:10.1109/SVR.2019.00049.

[28] SimTwo. [Online] https://github.com/P33a/SimTwo. Accessed: june 2022.

[29] CoppeliaSim. [Online] https://www.coppeliarobotics.com/. Accessed: january 2022.

[30] Gazebo. [Online] http://gazebosim.org/. Accessed: january 2022.

[31] Webots. [Online] https://cyberbotics.com/. Accessed: january 2022.

[32] T. M. Pinho, A. Moreira, and J. Cunha. Framework Using ROS and SimTwo Simulator for Realistic Test of Mobile Robot Controllers. volume 321, 07 2014. doi:10.1007/978-3-319-10380-8_72.

[33] Lucas Nogueira. Comparative Analysis Between Gazebo and V-REP Robotic Simulators, 12 2014. doi:10.13140/RG.2.2.18282.36808.

[34] W. Sankowski, M. Włodarczyk, D. Kacperski, and K. Grabowski. Estimation of measurement uncertainty in stereo vision system. *Image and Vision Computing*, 61:70–81, 2017. doi:https://doi.org/10.1016/j.imavis.2017.02.005.

[35] D. Kitayama, Y. Touma, H. Hagiwara, K. Asami, and M. Komori. 3d map construction based on structure from motion using stereo vision. In *2015 International Conference on Informatics, Electronics Vision (ICIEV)*, pages 1–5, 2015. doi:10.1109/ICIEV.2015.7334018.

[36] Motive. [Online] https://optitrack.com/software/motive/. Accessed: june 2022.