Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Bachelor's Thesis

Learning the Odometry on a Humanoid Robot

Fachbereich Informatik

MIN Fakultät

Universität Hamburg

Autorin:	Valerie Bartel
Studiengang:	B. Sc. Informatik
Matrikel-Nr.:	7406472
Erstgutachter:	Prof. Dr. Jianwei Zhang
Zweitgutachter:	Niklas Fiedler
Betreuer:	Jasper Güldenstein
Abgabedatum:	2. Mai 2024

Abstract

In this thesis we present a lightweight method of calculating the odometry of a bipedal humanoid robot which is independent of environmental conditions like lighting. Our approach consists of training a neural network to predict the walking step length based on proprioceptive sensor data. Through the integration of these predictions, the robot's current pose relative to a designated origin is determined. This approach is developed and evaluated in the RoboCup Humanoid Soccer League domain.

We collect two distinct datasets: One in simulation using the Webots simulator and one in the real world using a motion capture system. Different neural network architectures are trained and the best-performing model is integrated into the existing software stack of the Hamburg Bit-Bots RoboCup team. Subsequently, it is compared against two established odometry approaches that have been used to win competitions. Our new approach decreased the average drift over a five-second period by 80 percent in simulation and by 14 on real-world data in comparison to the baseline methods.

Zusammenfassung

In dieser Abschlussarbeit wird ein umgebungsunabhängiger und leichtgewichtiger Ansatz zur Bestimmung der Odometrie eines zweibeinigen humanoiden Roboters präsentiert. Ein neuronales Netz wird auf propriozeptiven Daten trainiert, um die Schrittlänge des Roboters vorherzusagen. Indem man diese integriert, erhält man die aktuelle Position des Roboters im Verhältnis zu einem Ursprung. Die Umgebung, in welcher dieser Ansatz entwickelt und ausgewertet wird, ist die RoboCup Humanoid Soccer League.

Es werden zwei separate Datensätze aufgenommen: Einer in Simulation unter Nutzung des Webots Simulators und einer in der realen Welt mit einem Motion-Capture-System. Dann werden verschiedene Modellarchitekturen trainiert und das performanteste Modell in die bestehende Software des Hamburg Bit-Bots RoboCup-Teams integriert. Wir vergleichen unseren Ansatz mit zwei bestehenden Ansätzen für Odometrie, welche genutzt wurden, um Wettbewerbe zu gewinnen. Unser neuer Ansatz verringerte den durchschnittlichen Drift nach fünf Sekunden um 80 Prozent in der Simulation und um 14 Prozent in der echten Welt im Vergleich zu bestehenden Ansätzen.

Contents

List of Figures

List of Tables

1 Introduction

Autonomous mobile robots can explore and interact with their environment independently. To navigate and take further action, they need to localize themselves, which is the process of determining where the robot is positioned on a map of the environment [HD99]. This is particularly important in the RoboCup Humanoid Soccer League, where bipedal humanoid robots compete autonomously in soccer matches, as seen in Figure 1.

The robots need to localize themselves on a map of the playing field for tactical play and high-level behavior, such as effective passing. They also require knowledge of how their current position relates to where they started, as the fields are symmetric. Otherwise, they can confuse the enemy's side of the field with theirs, leading to their own goal. Due to the limitation of computational resources and the need to make decisions in real-time, the approach should be computationally inexpensive and fast. Additionally, the Humanoid League imposes sensor restrictions by only allowing those that can measure what humans can perceive.

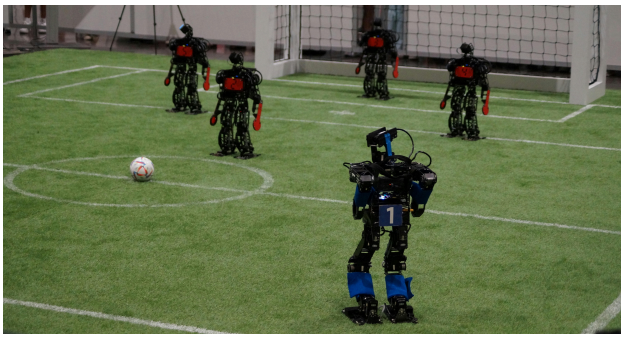


Figure 1: Robots of two different teams during the RoboCup 2023 in Bordeaux. They need to enter the playing field autonomously and position themselves for the kick-off. Source: [Bit24]

Odometry is a solution to these challenges. This method consists of integrating motion estimations [SK16] [p.737] to obtain a pose in relation to a starting point. For that, it is often only relying on information gained from proprioceptive sensors like joint encoders and inertial measurement units (IMUs), making it inexpensive to compute and conforming to the Humanoid League's rules. Odometry can either be used as a lightweight localization approach if the starting position in the world is known or be integrated with other measurements in more complex localization algorithms like the Monte-Carlo-Localization [Del+99, Thr+01].

For wheeled robots, it is relatively simple to infer knowledge of their current position from past motion sensors as they always have a specified link with ground contact called support link and a simple wheel model [SK16] [p.737].
For legged robots, on the other hand, it is more complicated. Their current posi-

tion can be computed by counting the walking step sizes and integrating them, assuming stable foot-to-ground contact during step-taking.

Regarding the Humanoid Soccer League, this approach often suffers from unbounded drift. The step lengths are computed incorrectly if the robot's geometric model is inaccurate, which is probable as they are mostly low-cost. Cheap sensors are usually less accurate and noisy. Inexpensive actuators tend to suffer from more backlash. If the robot has 3D-printed parts, which are light and cheap to produce, they are also prone to being unstiff. Additionally, the stable foot-to-ground-contact assumption is often violated when slippages happen. These frequently occur as walking with only two legs can be unstable, and the playing field consisting of artificial grass is slippery.

To address these issues, we propose a method to improve legged odometry by using machine learning. The systematic error originating from irregularities in the geometric model can be learned from the discrepancy between the step sizes the robot computes based on joint angles and the actual distance between the feet. The dynamic error caused by slippages or deformation differences can be estimated by using measurements of an inertial measurement unit, which detects linear acceleration and angular velocity.
We present a lightweight solution to the problem of unbounded drift of simple odometry for low-cost and resource-constrained bipedal robots. We achieve this by training a neural network on data generated in simulation or the real world to predict the step size based on proprioceptive sensor data and then integrating these predictions to determine the robot's current pose relative to a designated origin. This can be used as a localization or to enhance other localization algorithms. In the RoboCup Humanoid Soccer League this is a fundamental requirement for robots to participate in the game usefully. Additionally, we investigate the improvement of legged odometry through the machine learning model by comparing it with two existing odometry approaches, which have been used successfully by the Hamburg Bit-Bots RoboCup team in competitions. Furthermore, we assess the transferability of results obtained in simulation to real-world scenarios, as data collection is performed in both environments.

First, the basic knowledge to understand this thesis is introduced in Chapter 2, where we present the environment and robot platform used. We also explain odometry, supervised learning, and other software used.
Second, other approaches that also implement odometry are presented in Chapter 3. These can be divided into traditional and machine-learning approaches, shown in Section 3.1 and Section 3.2.
Then, we present our approach in Chapter 4. In Section 4.1 and Section 4.2, we show how we collect and process data in simulation and the real world. Next, the neural network architectures and the training procedure are described in Sec-

tion 4.3 and their integration into the existing software of the Hamburg Bit-Bots RoboCup-team is shown in Section 4.4. The code used in this Chapter is made publicly available on GitHub.[1]
In Chapter 5, we discuss the best-performing neural networks and evaluate our odometry against two already existing odometry approaches used by the Hamburg Bit-Bots.
Finally, a conclusion is given in Chapter 6, and future work is briefly outlined.

[1]https://github.com/val-ba/model-odometry

2 Fundamentals

This Chapter provides the basic knowledge needed to understand this thesis. We offer a brief overview of the environment the odometry proposed in this thesis is set and evaluated in by introducing the RoboCup in Section 2.1 and the robotic hardware platform in Section 2.2. Afterwards, as we want to calculate the odometry of a humanoid robot, we define the robot's motion mathematically and present an approach for odometry on legged robots in Section 2.3. Enhancements by the addition of different sensors are also mentioned. Then, we explain the concept of supervised learning in Section 2.4, with an emphasis on neural networks and their peculiarities in Section 2.4.1 and Section 2.4.2, as they can approximate the errors present in conventional odometry. Following that is an introduction to ROS 2 in Section 2.5, which is the underlying robotics middleware worked with, and a presentation of the advantages and disadvantages of simulation in Section 2.6 and Webots Section 2.6.1 for data collection.

2.1 RoboCup

The RoboCup is an annual robotics competition with the objective to win against the reigning soccer world champions adhering to official FIFA rules by 2050. It first took place in 1997 to foster the public's interest in robotics and artificial intelligence [Fed24]. Initially focused on soccer due to it being a popular sport with many challenges like navigation, object detection, self-localization, and high-level behavior, the competition now includes leagues to advance robotics usage in domestic tasks, crisis response, or logistics.

We are interested in the Humanoid Soccer Leagues, where humanoid robots must play entirely autonomously, as depicted in Figure 2. In the following, we focus on the Humanoid Soccer Kid-Size League, which is mostly similar to the Adult-Size League but differs in the height of the robots, the size of the playing field, and some peculiarities regarding robot handling.
The robots compete in teams of four under modified FIFA rules [Fed23]. Their heights can range from 40 cm to 100 cm, and they can have custom hardware, which has to be humanoid. This means the model has to have a structure similar to a human body, and the sensors are restricted to the ones that can measure features humans can sense. The playing field has an area of 6 times 9 meters and consists of artificial grass, making it non-trivial to walk on. The field lines are modeled after real soccer field lines but scaled down.

Figure 2: RoboCup Humanoid League match in Bordeaux 2023. Source: [Bit24]

Besides the main league, there occasionally is the Humanoid League Virtual Season taking place in simulation using Webots which is described in Section 2.6.1. Here, matches on a modeled field seen in Figure 11 and digital models of robots and their behavior are simulated.

A good odometry can either reduce the complexity of or function as a naive localization during soccer matches. This is important because localization on the field is one of the key problems that must be solved before intelligent play can occur. Because of the playing field's symmetry, even good localization needs some sort of knowledge of which side the robot stands on.
The odometry needs to be computationally efficient and should not rely on non-human sensor modalities due to the computational restrictions of mobile robots and the limitation of sensors imposed by the Humanoid League rules.
Additionally, it needs to handle inaccuracies in the robot model and slippages on the artificial grass. That is why the RoboCup provides an environment in which odometry can be developed and tested under non-trivial conditions.

2.2 Wolfgang OP

Wolfgang-OP [Bes+21] is the robot platform used to participate in the RoboCup Humanoid League by the Hamburg Bit-Bots RoboCup team. It is a low-cost humanoid robot platform that is derived from the Nimbro-OP. It is 80 centimeters tall, weighs 7.5 kilograms, and has 20 degrees of freedom. It is made of aluminum, carbon, and PLA, which are light materials, and TPU, which is flexible. An ASUS Mini PC PN51 is mounted on the robot and provides the computational resources. The actuators used are Dynamixel MX-64, MX-106. and XH540, which have joint angle encoders [MX-24]. The MX-64 motors are located in the arms and head, and the XH540 in the knees. The rest of the actuators are MX-106. A Basler acA2040-35gc on the robot provides images. Two MPU6500 internal measurement

units are placed on the head and in the body to obtain information on the robot's acceleration, which is useful for fall detection. There are also four strain gauge-based foot pressure sensors on each foot to detect ground contact.

As this robot is mainly used to participate in the RoboCup Humanoid League, some custom changes have been made to improve performance for this use case. Because robots frequently fall during matches, elastic elements in the shoulder actuators and neck are added, as well as bumpers to the front and back of the torso. The robot usually stands or walks, which means there is much weight on the knee's actuator. A torsion spring is integrated to reduce this torque.

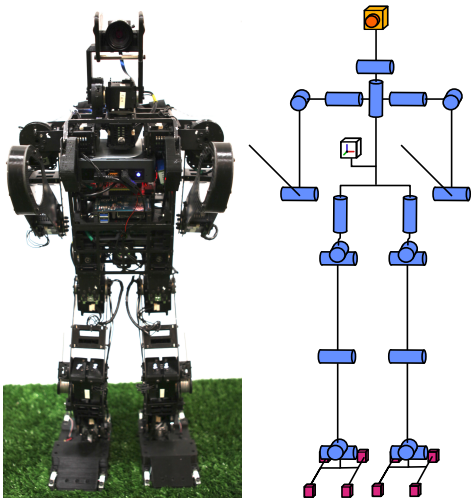


Figure 3: Wolfgang and a schematic representation of its kinematic chain. Source: [Bes+21]

Figure 4: The Webots model of Wolfgang.

A Webots model of Wolfgang exists, which was refined when the Hamburg Bit-Bots participated in the 2021 RoboCup worldwide and the Humanoid League Virtual Season.

In this thesis, we work with forward kinematics and mention a few specified links of the robot, which are the base link in the body and the left and right feet. They are depicted in Figure 5.

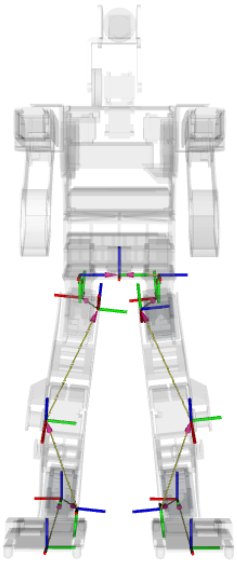


Figure 5: The sole frames are located at the bottom of the feet, and the base link is located in the robot's body. For forward kinematics, the chain of links in between needs to be traversed for that, so these are also visualized.

Due to the platform's inexpensive hardware and use of 3D-printed parts, inaccuracies in the geometric model are introduced, which should be accounted for. By using the robot's proprioceptive sensors, a neural network-based odometry can be developed to address these imprecisions.

2.3 Odometry

As mentioned in Chapter 1, odometry in robotics integrates motion estimations to compute the current pose in relation to an origin. It can be used for naive self-localization or as a starting reference for more computationally expensive localization algorithms [BM18]. Usually, information from proprioceptive sensors, like joint encoders and IMUs, are used to measure the distance traversed in comparison to a starting point. However, there are also approaches in which data from exteroceptive sensors is taken into consideration.

2.3.1 Rigid Body Motion

We want to describe how the 3-dimensional movement of a rigid body, which is a non-deformable object, can be expressed mathematically because this is how we can represent the movement of a robot and, thus, odometry on a computer. For this, we refer to Chapter 4 of Modern Robotics by Lynch et al. [LP17]. Then, we

show how this can be applied to a robot that moves in relation to a fixed reference frame, which is the origin of the odometry.
We focus on translation on the x and y-axis and a rotation around the z-axis as most mobile robots move on a 2-dimensional plane, which is true for the soccer field in the RoboCup. However, some moving objects like drones can translate and rotate along all three spatial axes.

A rotation by θ around the z-axis can be represented by the following matrix:

$$R_z = \begin{pmatrix} \cos(\theta) & -\sin(\theta) & 0 & 0 \\ \sin(\theta) & \cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{1}$$

Translation by n on the x-axis and m on the y-axis is represented by a 4x4 matrix:

$$T = \begin{pmatrix} 1 & 0 & 0 & n \\ 0 & 1 & 0 & m \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{2}$$

The extra dimension is added so that the transformation distinguishes between points and vectors. Points can be scaled, but vectors cannot.

If a robot moves n units along its x-axis and m units along its y-axis and turns by θ, then the transformation can be expressed by a single matrix:

$$\begin{aligned} T \cdot R_z &= \begin{pmatrix} 1 & 0 & 0 & n \\ 0 & 1 & 0 & m \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} \cos(\theta) & -\sin(\theta) & 0 & 0 \\ \sin(\theta) & \cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \\ &= \begin{pmatrix} \cos(\theta) & -\sin(\theta) & 0 & n \\ \sin(\theta) & \cos(\theta) & 0 & m \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \end{aligned} \tag{3}$$

We want to express the movement of a robot in comparison to a fixed reference frame, so we define two frames: the fixed world frame $\{w\}$ and the robot's base link frame $\{b\}$. By knowing where the robot initially stands, we know how the

base link and world frame relate: T_{WB}. Thus, we can express any point p specified in the base link frame in the world frame:

$$p_w = T_{\mathrm{WB}} \cdot p \tag{4}$$

It is easier to detect one's movement than to relate it to a global coordinate system. That is why when the robot moves, the odometry estimates the change of position $T_{\mathrm{BB'}}$ based on the base link and not the world frame. For the odometry, we want to know the robot's new pose in the world frame $T_{\mathrm{WB}}{}'$, which can be calculated by:

$$T_{\mathrm{WB}'} = T_{\mathrm{WB}} \cdot T_B \cdot T_{\mathrm{BB}'} \tag{5}$$

T_{WB} is updated to the values of $T_{\mathrm{WB}'}$, and the process is repeated for the next steps to always mirror the robot's pose in the world.

2.3.2 Proprioceptive Odometry

In this thesis, we are working with a low-cost bipedal robot with limited computational resources in the RoboCup Humanoid League, in which many sensors are prohibited, as explained in Section 2.1. That is why we present a method of calculating proprioceptive odometry (PO) for legged robots and briefly describe frequently used sensors.
PO only relies on information gained from the robot itself and not the environment to estimate the positional displacement of the robot. This data is usually consistently available and published in high frequency. It is also computationally efficient compared to algorithms that rely on visual information as the data is low-dimensional, which is important for systems with limited resources.

For wheeled robots, estimating this movement can be straightforward, provided the wheel model is accurate. Through the usage of wheel encoders, which measure how much the wheel's actuator is spinning, and the knowledge of wheel circumference, the distance laid back can be calculated by multiplying the number of wheel rotations by the circumference [BM18]. In the following, the focus is on legged robots.

Measuring the actuators' velocities would not be enough to estimate the robot's position as the geometric model is more complex.
A simple approach is to define a link in each of the robot's feet and a base link in its torso. We know how the feet and the base link relate to each other through forward kinematics [SK16] [p. 28]. A kinematic chain between the links needs to be defined, then we can calculate the transformation between its frames by matrix

multiplication.
One foot is always on the ground, and we assume that it has zero velocity, which means that it is not slipping. This foot is called a support foot. When another foot comes into ground contact, meaning that a step is finished, it becomes the new support foot. Using forward kinematics, we can calculate the transformation from the previous to the current support foot.
When calculating the odometry, we define a frame $\{s\}$ and the reference frame $\{w\}$ to align in the beginning at step 0, meaning ${T_{\mathrm{WS}}}^{0} = \mathrm{I}$. After the ith step is taken, we use the transformation from the previous to the current support foot $T^{i}_{\mathrm{SS'}}$ and calculate the following:

$$T^{i}_{\mathrm{WS}} = T^{i-1}_{\mathrm{WS}} \cdot T^{i}_{\mathrm{SS'}} \tag{6}$$

Assuming the robot has taken k steps so far, to obtain the current position of the robot, we use the transformation T_{SB} from the support foot to the base and compute:

$$T_{\mathrm{WB}} = T^{k}_{\mathrm{WS}} \cdot T_{\mathrm{SB}} \tag{7}$$

This approach is prone to errors due to inaccurate modeling of the links and incorrect angle measurements. Additionally, walking in general is less stable, increasing the risk of stumbling and slipping.

In the following, we now explain frequently used proprioceptive sensors [Ble+18, Blo+15, Cam+20, Har+18, Yan+23].
Motor encoders, usually integrated within an actuator, provide information about the joint positions and angles. By adding knowledge about the distances between actuators, they can be used to calculate the transformation between adjacent joint frames through forward kinematics [SK16] [p. 28]. For humanoid robots, the links of interest are the left and right foot and the base frame, which is usually located in the robot's body. Their transformations can be computed by matrix multiplication because they can be viewed as a serial chain. If the robot's model is inaccurate, the transformation is off and less useful.
Inertial measurement units (IMUs) can sense linear acceleration and angular velocity [SK16] [p.743]. The IMU's angular velocity can be integrated to obtain the orientation. The linear acceleration needs to be integrated twice to obtain the translation. This leads to large errors, especially on cheap and noisy IMUs.
Foot pressure sensors can be used to detect ground contact.

2.3.3 Exteroceptive Odometry

PO can be augmented by using a variety of other sensors. This leads to more accurate results, but also more complex algorithms with more computational and hardware expenses. We only briefly mention them here for completeness, as this thesis focuses on proprioceptive odometry.
An example of using external information is tracking key features in the camera's image to detect positional changes and using this information to deduct to one's position [Aqe+16].
If only visual information is used, the approach is called Visual Odometry (VO). If it is an addition to the IMU, it is called Visual-Inertial-odometry (VIO) [Blo+15].
Other feasible sensors for extending the odometry are lasers, sonar scanners, or GPS [SK16] [p.744].
Sometimes, many different sensors are fused to create a robust model [WCF22].

2.4 Supervised Learning

As legged odometry usually suffers from unbounded drift, as mentioned in Section 2.3.2, we want to use supervised learning to mitigate this error by learning a corrected step size.
Supervised learning aims to infer a target value from observable information. A mapping from input to output can be approximated through the usage of annotated data, which means that the target values of given input features are known. The following explanation is based on the notation used in Chapter 6 of Data Science: An Introduction to Statistics and Machine Learning by Matthias Plaue [Pla23].
An n-dimensional input can be represented as a vector:

$$X = (x_1, x_2, ..., x_n) \tag{8}$$

If all input features are continuous, then $X \in \mathbb{R}^n$. The set of all feature vectors can be called $\mathbf{X}$.
Similarly, the target value can be defined as :

$$Y = (y) \tag{9}$$

If y is continuous, namely $y \in \mathbb{R}$, it is called a regression problem. The set of all output vectors is $\boldsymbol{\Upsilon}$. $\boldsymbol{\Upsilon}$ can have multi-dimensional elements as well, but they are treated as one-dimensional from now on for simplicity.

As mentioned above, we have annotated data which can be represented as pairs:

$$\{(X_i, Y_i) |\ X_i \in \mathbf{X} \wedge Y_i \in \boldsymbol{\Upsilon}\} \subset \mathbf{X} \times \boldsymbol{\Upsilon} \tag{10}$$

The set of all possible mappings from $\mathbf{X}$ to $\boldsymbol{\Upsilon}$ is defined as:

$$\Phi := \{f : \mathbf{X} \to \boldsymbol{\Upsilon}\} \tag{11}$$

We want to find the mapping $f \in \Phi$, which captures the relationship between input and output vectors by fulfilling $f(X_i) = Y_i$ for all pairs in the dataset.
For that, we choose a model which can represent several mappings of a subset of Φ:

$$\varphi := \{f(\cdot, \Theta) : \mathbf{X} \to \boldsymbol{\Upsilon} \mid \Theta \in P\} \subset \Phi \tag{12}$$

The model can take on different configurations of parameters P.
It is unlikely that we can find the exact mapping, so the goal is to find a configuration $\widehat{\Theta} \in P$ which makes $f\left(\cdot, \widehat{\Theta}\right)$ a good approximation. If the mapping is good, then for any input features X^* and the belonging output Y^* and a prediction Y':

$$f\left(X^*, \widehat{\Theta}\right) = Y' \sim Y^* \tag{13}$$

As we can only sample the input and output space most of the time, we create a dataset by doing so. In a dataset D of size N, we have N tuples $(X_i, Y_i), i \in \{1, ..., N\}$. Assuming that the dataset reflects the actual relation of input to output features reasonably well, the model makes good predictions on unseen data. If it does not, the prediction will be biased, which happens if datasets are imbalanced or too small.
How good a mapping approximates the relationship of input to output, can be measured by calculating the distance between the prediction on input X_i and the label Y_i with $(X_i, Y_i) \in D$. This is called a loss function:

$$\lambda : \boldsymbol{\Upsilon} \times \boldsymbol{\Upsilon} \to [0, \infty[\ , \quad \left(Y, \widehat{Y}\right) \mapsto \lambda\left(Y, \widehat{Y}\right) \tag{14}$$

We want the model to predict well on the whole dataset. Thus, we minimize the following function with respect to Θ:

$$\Lambda(\Theta) = \frac{1}{N} \sum_{n=1}^{N} \lambda(Y_n, f(X_n, \Theta)) \tag{15}$$

The loss depends on the distance metric chosen in Equation 14. For regression problems, feasible loss functions include the mean-squared and the mean absolute errors, which are calculated by taking the mean of the (squared) difference from label to function output.

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(Y' - Y^*)^2 \tag{16}$$

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}|(Y' - Y^*)| \tag{17}$$

There are several approaches to finding a local or global minimum for Λ by adapting Θ. This method depends on the model chosen. Approaches for neural networks are discussed in Section 2.4.1.

2.4.1 Neural Networks

One method to implement supervised learning is the training of artificial neural networks. Conceptually, they emulate the human brain's neurons [Agg18]. The most basic neural network is called perceptron and consists of n input nodes, which correspond to each dimension in the input vector, and one output node. This is depicted in Figure 6. The input nodes are connected to the output node by n distinct edges with weights $w_1, ..., w_n$. The output node computes $\left(\sum_{i=1}^{n} = w_i x_i\right) + b$, with b being an optional bias term, and then passes it through an activation function, which is the following in the case of a perceptron:

$$\text{sign}(x) := \begin{cases} 1 \text{ if } x \geq 0 \\ -1 \text{ else} \end{cases} \tag{18}$$

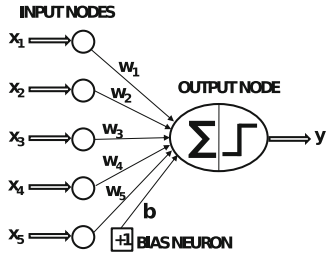


Figure 6: The five input features denoted as the x_i, $i \in \{1, ..., 5\}$ are all put into distinct input nodes. Then, the input features saved in the input nodes are multiplied by the weights w_i, $i \in \{1, ..., 5\}$ and then summed up with the bias term b. Lastly, the sum is passed through the activation function, which produces the final prediction of y. Source: [Agg18] P.5.

Multiple intermediate layers, which are called hidden layers, can be put between the in- and output layers. The number of hidden layers and their size can be varied, but the in- and output layers have to stay the same size as the in- and output features respectively. The nodes of each layer are fully connected to the nodes of

the previous and following layers. All these connections have their own weights. This is called a multi-layer perceptron and can be seen in Figure 7.

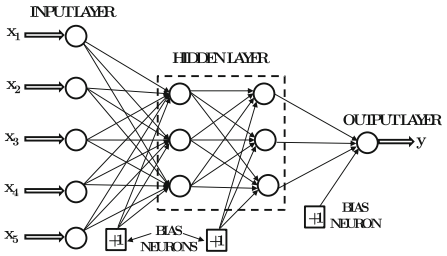


Figure 7: We see two hidden layers with size three, which are fully connected to the two adjacent layers. As mentioned, the in- and output dimensions are not changed. Source: [Agg18] P.18.

The weights and bias terms operate as parameters of the model, like Θ in Equation 12. In contrast, the set P is decided upon by the model architecture, meaning the size and connection of its layers. We want to find a configuration $\hat{\Theta} \in P$, which minimizes a chosen loss function like in Equation 15. Λ is calculated by passing the samples of the given dataset through the network and comparing the output with the label in Equation 14. Then, the gradient of the function Λ given the dataset is calculated, pointing in the direction of the steepest ascent. The weights are updated in the opposite direction of the gradient by a fraction of the actual value, which is defined by a scalar value called the learning rate. The learning rate must be sufficiently big because otherwise, the weights will be updated too slowly. If it is too big, on the other hand, the local or global minima may not be reached as it keeps overshooting.
There are variations of this approach: Stochastic gradient descent (SGD) updates the weights based on many randomly chosen samples, instead of deriving Λ. If a batch of samples is used instead of the whole dataset, it is called mini-batch gradient descent. There are also other more sophisticated approaches where the size of the derivation step is varied dynamically, such as the Adam optimizer [KB14], which uses momentum.
We still need to determine which node contributed to which part of the error to update the weights accordingly. This can be done by using an algorithm called backpropagation, which, after passing the input data through the network and computing the loss, recursively computes the gradient with respect to the parameters, i.e., weights and biases, starting at the output layer. This can then be used to update the weights according to the gradient.

We want the activation function to map values into narrow ranges like -1 and 1, as this way, big values do not have proportionally smaller weights. Due to the backpropagation approach, it should also be differentiable. Additionally, the function should be non-linear. Otherwise, the model can only learn linear relations. The tanh and sigmoid functions are frequently used and suffice these criteria.

$$\begin{aligned} \mathrm{sig}(x) &= \frac{1}{1+e^{-x}} \\ \mathrm{sig}'(x) &= \frac{1}{1+e^{-x}} \cdot \left(1 - \frac{1}{1+e^{-x}}\right) \end{aligned} \tag{19}$$

$$\begin{aligned} \tanh(x) &= \frac{e^{2x}-1}{e^{2x}+1} \\ \tanh'(x) &= 1 - \left(\frac{e^{2x}-1}{e^{2x}+1}\right)^2 \end{aligned} \tag{20}$$

The non-differentiable ReLU function can also be used by setting its derivative to zero at its undifferentiable point.

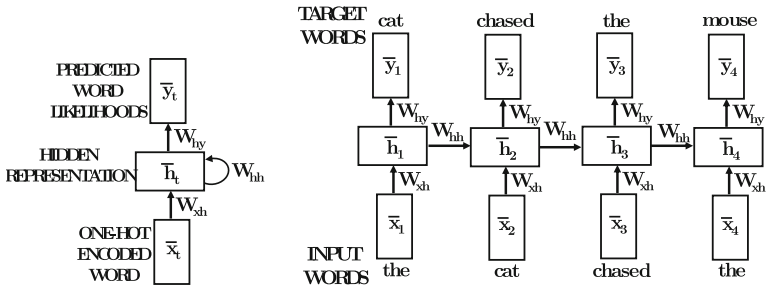
$$\begin{aligned} \mathrm{ReLU}(x) &= \max(x, 0) \\ \mathrm{ReLU}'(x) &= \begin{cases} 1 \text{ if } x > 0 \\ 0 \text{ else} \end{cases} \end{aligned} \tag{21}$$

Now, we present approaches to ensure the model performs well on unseen data.
To force the model to rely not only on specific neurons encoding some features, they are randomly set to 0 with a specific probability. This approach is called dropout.
Overfitting occurs when complex models are trained on small data sets, as presented in Section 2.4.2. The model remembers the dataset's samples instead of learning the proper relationship between input and output. Unseen inputs passed through the mapping are poorly predicted. This can be counteracted by using a simpler model or stopping the training early enough.
To determine the latter, the dataset is split into two parts. The first part is used to optimize the parameters. The samples in the second part are not used for training and thus show how well the model generalizes to unseen data. This part is also used to decide on the model architecture, the optimizer, the loss function, and other hyperparameters.

2.4.2 Recurrent Neural Networks

When a bipedal robot takes a step, the previous steps might be influencing the current one. Thus, information on preceding steps should be considered when training a neural network to predict the step size. If the order of input samples gives useful context, e.g. for time series, recurrent neural networks can be used.

They have hidden layers that have cycled connections between nodes. That means the values of previous input sent through the model affect the current prediction as seen in Figure 8.



(a) RNN (b) Time-layered representation of (a)

Figure 8: A neural network with a recurrent layer and its unrolled representation. In (a), we see that some edge weights represented by the edge w_{hh} are fed back into the hidden layers. This way, the network can remember information from previous inputs. In (b), we see that it does not have any cycles. Source: [Agg18] P.275

When optimizing the parameters, the network needs to be unrolled over time to have the same structure as a normal multi-layer perceptron, as seen in (b) of Figure 8. This unrolling can get very long for sequences with many input steps.
This way, the vanishing, and exploding gradient problems might occur when optimizing through backpropagation as the update rates for the weights between layers get very small or very big. The gradient usually vanishes for the weights between the first layers, almost not altering them at all and thus making learning impossible. On the other hand, the exploding gradient has huge update steps, which means the loss function probably does not converge.

To counter this, there is long-short-term memory [HS97], which can also remember the context of previous inputs but with fewer weights. It contains two different states. The cell state functions as a long-term memory as it can retain information from the whole input sequence. A hidden state, which is the output of the LSTM at the previous time step, realizes the short-term memory. The two states are managed using three different gates, which can be seen in Figure 9. The forget gate decides which information should be removed from the cell state, whereas the input gate checks which information should be forwarded to it. The output gate selects which information from the short- and long-term memory should be passed to the next layer and output.

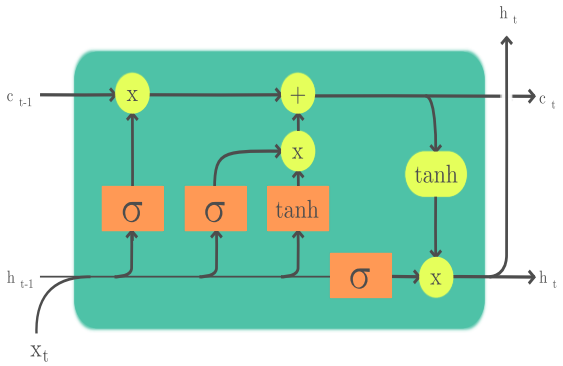


Figure 9: Depiction of an LSTM. The input vector at time t is denoted as X_t and the hidden state of the previous time step as h_{t-1}. They are concatenated and first passed through the forget gate, which consists of the first sigmoid and the combination of its output with the previous cell state c_{t-1}. The input gate comprises the second sigmoid and the tanh function, which are combined and added to the cell state. The output gate combines the previous hidden state passing through the third sigmoid and the cell state c_t passing through the tanh. The output is h_t, which also functions as the hidden state for input x_{t+1}. Source: [Lon24]

2.4.3 PyTorch

To implement the neural network for the odometry, we use PyTorch [Pas+19], a machine-learning library for Python. It has commonly used optimizers, loss functions, and components of neural networks like fully connected or recurrent layers already implemented and can be used modularly.
The underlying data structure used is called a tensor, which is the multi-dimensional extension of one-dimensional vectors and two-dimensional matrices. These tensors store values including edge weights between nodes and in- and output features. Tensor operations can be calculated on accelerating hardware such as GPUs, which can be used through the Nvidia CUDA framework.
When training a neural network, we usually want the data to come in batches and be shuffled. By inheriting from PyTorch's Dataset class, a dataset can be implemented that can be used by a PyTorch Dataloader, which automates tasks such as batching, shuffling, and multi-threaded loading, thereby streamlining the training process and improving efficiency.

2.5 ROS 2

The software run on Wolfang presented in Section 2.2 when participating in the RoboCup is mostly written using ROS 2 [Mac+22]. Therefore, it is used to implement neural network-based odometry.
ROS, standing for robot operating system, is an open-source middleware for usage in robotics and enables communication of otherwise independent components. There are two distinct frameworks named ROS 1, whose latest distribution reaches the end of life in May 2025 [Dis24], and ROS 2, with new distributions still released yearly [ROS24]. The underlying concepts and implementations differ, and in the following, the focus is on ROS 2.
The fundamental units of the software are called nodes. They have different functionalities and their own context and thus need to share data explicitly. The framework provides means of n-to-m communication via so-called topics. On these topics, standardized messages can be passed asynchronously through a publisher-subscriber mechanism seen in Figure 10. Custom messages can be defined by specifying their format following a given semantic.
Other means of communication are services that provide a request-response communication when the assurance of receiving and completion of a task is wanted, and actions that are similar to services but can provide continuous feedback are non-blocking and cancellable.

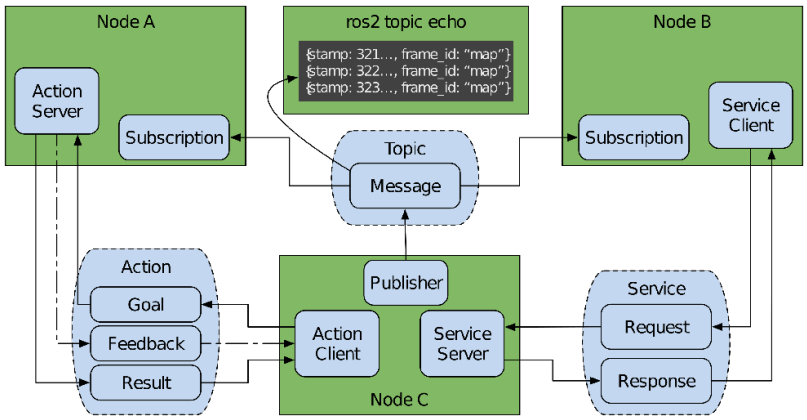


Figure 10: Node A and Node B subscribe to a topic on which Node C publishes messages. Node A provides an action server. Node C has an action client which can send requests to the server. The service follows a similar principle, with Node C providing it and Node B requesting it. Source: [Mac+22]

ROS 2 is implemented to provide a more secure and time-constraint reliable communication compared to ROS 1 by using DDS, a middleware for data exchange. It uses a decentralized peer-to-peer approach for every node to detect the others instead of a centralized node that handles the registration and opens the communication channels between other nodes.

ROS 2 is written in C but supports different languages, foremost C++ and Python, that implement the ROS client library (rcl). As ros nodes are structured into packages, reuse of existing implementations from other users is possible. Several tools enable visualization of data like plotjuggler [Fac24] for time series, Rviz [RVi] for general visualization, and rqt [ROS24] running graphical user interfaces as plugins. There is also a tool called rosbag, which enables recording, saving, and playing back messages published on given topics. They are stored as .mcap, which is a container format for time-stamped data. This way, data can be serialized and stored efficiently. Later, it can be deserialized again, and all the information about topics, messages, and their times of reception is kept. Another useful package is tf2, which provides geometric transformations between frames defined in a tree structure to given time stamps.

2.6 Simulation

In this thesis, we want to generate data for neural network-based odometry in simulation, as it is easier to obtain and does not pose a risk to the robot compared to the real world. Moreover, this simulation-based approach serves as a proof-of-concept for sim-to-real transfer, showing whether training neural networks in simulation and deploying them on physical robots is viable.
A system or objective can be investigated by building a model that behaves similarly but is easier to observe. Robots and their environment can be modeled digitally on a computer. There usually is a need for an accurate physical representation of the real world and thus mathematical models have to be implemented to compute the behavior of objects. Even if this is done well, there will be inaccuracies as a physics engine can only approximate real-world properties and does not consider all of them.

This means there is always a gap between soft- and hardware working in simulation and transferring it to the real world. The models are flawed as sensors underly different noise due to production errors. It is also difficult to model the materials used and inertia correctly. Additionally, the simulator does not run in continuous but discrete time steps and thus can only approximate continuous systems.
To minimize this difference, the model has to be either very accurate to the real world, which is difficult to obtain, or depend very little on the implementation details of the simulation software by focusing on modeling only the most important characteristics of the system. An example of that is the simulation of a robot's sensor used to detect an event and the difficulty of modeling the sensor readings and noise. To obtain good results, we either need to take special care during the modeling of the sensor to make it as realistic as possible or we only focus on whether the sensor detects the event or not and work with this information, dis-

regarding the concrete readings.
Nevertheless, simulation can be very useful for developing and testing robotic systems, especially when acquiring real-world data is challenging or impractical. The simulation keeps track of detailed information about objects in the world and their properties to properly simulate. This means obtaining this information does not pose an additional cost. In addition, it is usually cheaper, and there is no risk of damaging hardware or causing other harm during experiments. Simulation can also accelerate the development process as changes to the model can be made quicker than those to the hardware.

2.6.1 Webots

The simulation software Webots [Web24] has been used for the RoboCup Humanoid League Virtual Season and thus is an established environment for simulating humanoid robots. As mentioned in Section 2.1 and Section 2.2, there exist digital models of the robots and the field, which we use.
Webots is a simulation software for the physical simulation of mobile robots developed by Cyberbotics Ltd and has been open-source since 2019. Simulation of various sensors and actuators is supported, enabling users to model different hardware platforms.

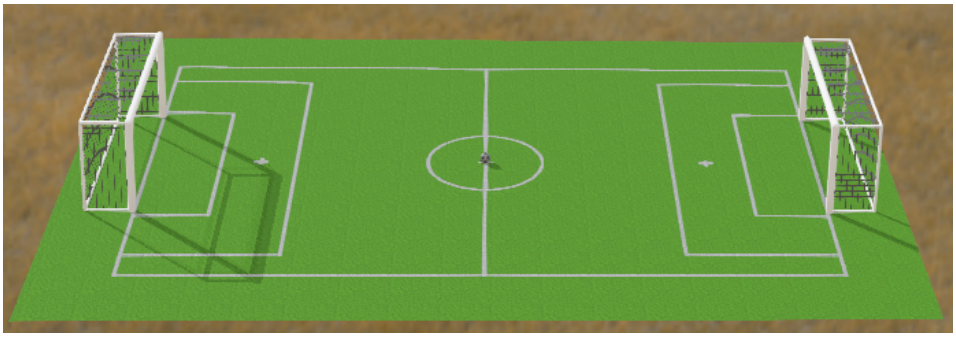


Figure 11: The model of a six times nine meters soccer field. The grass is an unstiff plane.

It is also possible to create a so-called Supervisor, which can edit the scene and also obtain information about every other object in the simulation, except internal measurements and computations made by the robot [Sup24]. Webots also provides an interface for implementation with ROS nodes, making running software written in it, using visualization tools, and rosbags for data collection possible.

There are some disadvantages of simulating robots in Webots due to the underlying physics engine, the Open Dynamics Engine (ODE) library developed by Russel Smith, which is used to simulate rigid bodies composed of many interconnected parts. Most legged robots have many joints, and the joint-dampening approach appears to not model the real-world counterpart well [Dru+10]. The friction mod-

eling has artifacts, meaning that movements differ from the real robot and joint constraints may be overstepped if a solver cannot find another way.

In this thesis, we want to generate data in simulation to leverage the advantages mentioned above. As the robot software uses ROS 2 and a model of the field and robot exists, Webots is a feasible simulator.

3 Related Work

Key problems in legged odometry are the inaccuracies of sensor measurements and the detection and modeling of slippages. There are several existing approaches that tackle these problems which can be roughly divided into two categories. The first consists of traditional approaches, which mostly use different models of Kalman Filters (KF) described in Section 3.1, and the second one uses machine learning on data from different sensors shown in Section 3.2.

3.1 Traditional Approaches

Because of noisy sensor information, many of the following approaches use a Kalman Filter, which can estimate a robot's position by making an initial guess about its starting position and velocity. Then, it can use its sensors to obtain a measured position. Afterwards, the predicted position is calculated by applying a weighted difference between the initial guess and the measured velocity and adding it to the initial guess [WBo95]. The same is done for the velocity. The next state estimation, which will replace the initial guess from then on, is calculated by adding the predicted velocity to the predicted position. The advantages are that it is comparatively easy to compute and returns the optimal linear solution. Often, we need to estimate non-linear relationships, which the Kalman Filter cannot do. For that, an Extended Kalman Filter (EKF) can be used, which linearizes all the non-linear components [JU97]. Drawbacks to this approach are that sometimes unstable filters are created and that the calculation the linearization is based on can be difficult to implement. These problems can be solved by using an Unscented Kalman Filter (UKF) that estimates the current state from a random variable whose mean and covariance are set based on deterministically chosen samples passed through the motion model.

Bloesch et al. model the odometry for legged robots using IMU measurements and kinematics in an Extended Kalman Filter [Blo+13]. They try to keep the model simple and independent of the robot platform. Their state vector is composed of the position of the center main body, its velocity and a rotation in relation to the internal coordinate frame to the body. Adding to that are the absolute position of its feet, an accelerometer bias, and a gyroscope bias. Their measurement model consists of the transformation from world to body frame, a negative velocity of the body in the body frame, and the rotation and bias terms for the IMU. Using a motion capture setup, their approach is evaluated on a quadruped robot on uneven and slippery ground. They conclude that they can estimate the robot's roll, pitch, and velocity well but not its general position or yaw. This is because when one foot is in ground contact, these values can be fully observed

through IMU measurements. To obtain the absolute position though, the linear acceleration measurements need to be integrated twice leading to accumulation of large errors. Computing the yaw is difficult as gravity has no vertical translation reference.
Following the previous approach, Rotella et al. make changes to it to fit a humanoid robot [Rot+14]. As humanoid robots usually have flat feet instead of a quadruped's point feet, there is less possibility for tilting, making the geometric model and thus computations simpler. This approach is again independent of the exact humanoid robot platform used. They tested their implementation in simulation and compared it to the filter implemented by Bloesch et al. They find that the filters behave mostly similarly but that the velocity estimation for flat-foot robots diverges less.

Bledt et al. [Ble+18] implement odometry on the quadruped MIT Cheetah 3, which can use a normal Kalman Filter instead of the extended version. They split the pose estimation into first computing the robot's body orientation and second using a KF to predict the linear position.

Camurri et al. provide a legged robot state estimation framework with velocity correction, which at its core consists of an Extended Kalman Filter that fuses IMU and leg odometry [Cam+20]. They can also fuse additional information from exteroceptive sensors like stereo image cameras and lidar scanners. They evaluated their approach on several robotic platforms, which are both biped and quadruped, and it showed a deviation of less than 35 centimeters on ten meters traveled in difficult environments. The robot's walking speed ranged up to half a meter per second.

Yang et al. [Yan+23] explain that normal odometry usually assumes zero velocity when there is feet-to-ground contact, which is not true in some terrains. They hope to detect slippages by placing one IMU on each of the robot's calves and using their measurements, joint encoders, and a body IMU. Because they only use proprioceptive sensors, their computations are lightweight because of the low dimension of their data. Their evaluation showed that their approach outperformed standard PO.

Approaches, that use other mathematical models, are presented in the following. Wisth et al. use stereo vision and IMU measurements, joint encoders and torque sensors to incorporate a velocity bias term into a factor graph, which models slippages in the leg odometry of a four-legged ANYmal robot [WCF20]. They assume that there is an almost constant velocity bias in their Inertial state estimator when walking. They do not want to further model the dynamics of non-rigid terrains as this can be robot-specific and does not abstract well. The Bias term consists of the IMU's gyro and accelerometer biases and the new angular and linear velocity

biases proposed by the researchers. Additionally to the robot state, the position of all visual landmarks is estimated. They model their factor graph by preintegrating their IMU measurements between two consecutive time steps. Their approach showed improvements to their previous work and to other proprioceptive odometry approaches.
Wisth et al. use factor graphs to fuse four different sensors [WCF22]. As visual information was used, they extensively tested their approach in an environment that was dark and dusty. Their evaluation showed an improvement to other state-of-the-art approaches.

3.2 Neural Network-based Odometry

Wang et al. train a deep recurrent neural network that directly infers position from raw image data which can represent the sequential dependencies between images, which are difficult to model explicitly [Wan+17]. Their architecture consists of a convolutional neural network to extract the features followed by a multi-layered LSTM to encapsulate relations of the CNN's outputs. The final output is the pose. Their approach showed stable outputs, although they had some trouble evaluating them as overfitting was hard for them to detect.

Chen et al. trained a recurrrent neural network on sequences of raw IMU data to output the location transformations in 2D coordinates, learning the positional displacement directly from IMU data [Che+18].
Liu et al. trained a regression model on IMU data, which outputs a 3D displacement [Liu+20]. Then, it was integrated into an EKF, which takes the net's output and predicts the transform, the velocity, and an IMU bias.

Buchanan et al. train a 1D ResNet18 architecture, which is a convolutional network, to predict the positional increment from gravity-aligned IMU measurements generated on a quadruped robot [Buc+22]. The network also outputs its uncertainty. The model is evaluated by integrating it into a Kalman Filter and a factor graph.
Following that, the same authors train an LSTM and a transformer, which are both recurrent neural networks, to predict a device-specific IMU bias, which is later used in a factor graph [Buc+22]. The data is not necessarily collected on the robot. They take a slice of their IMU measures and previous bias estimates as input for their model to output the current bias term. As the model is device-specific, it can be used not only on legged robots but also on drones or any vehicle that uses these IMUs. They show that this improves the performance of odometry, especially in environments where visual information cannot be obtained.

Rouxel et al. want to compute the transformation of the head pose of a humanoid after a walk cycle, which consists of two consecutive steps and trains two locally weighted projection regression models [Rou+16]. The first takes the last two steps, which are calculated based on motor goals and a variable defining what foot is swapped. This approach can also be used for planning purposes, as no sensory data has to be measured. The second model predicts based on the latest two steps calculated by using sensor information from the IMU, foot pressure sensors, and motor positions. It also takes the step duration of the steps as input in addition to the binary variable because these steps are not periodic. Both models output the foot displacement for the current step. They only used a translation on the x and y axis and a rotation θ around the z-axis for input displacements and the output. This shows improvements on carpet and artificial grass, on which the humanoid is prone to slippage.

Looking at state-of-the-art legged odometry, we see many approaches for odometry regarding quadruped high-end robots that can infer their position from IMU data and their kinematics model, often aided through supervised learning and recurrent models. This is mostly used to estimate a velocity factor, which is then accompanied by the usage of additional sensors to determine the final robot pose. This causes computational constraints as the velocity factor has to be assessed at a relatively high frequency, and additional sensors pose high-dimensional data. Additionally, there seems to be a gap in using data gained in simulation, which poses fewer constraints on human and financial resources, and transferring it to the real world.

4 Approach

As described in Chapter 3, many state-of-the-art approaches use data from IMUs or joint angles as input to train a neural network, which is used to calculate the odometry of a robot. Similarly, our approach involves training a neural network on input derived from such sensors. However, to save computational resources, we predict a transformation from one foot to the other when a step is taken instead of estimating a velocity directly, which has to be done at a high frequency.
Furthermore, we mainly collect data in simulation, enabling us to generate a large dataset quickly and without concerns regarding damaged hardware. Nonetheless, we also conduct real-world data collection for model training and subsequent evaluation purposes.

The specifics of our data generation setup are explained in Section 4.1. Following that, Section 4.2 outlines our data processing pipeline. Afterwards, the train and test procedure to determine the best-performing neural network is shown in Section 4.3. Lastly, in Section 4.4, the integration into the existing software stack of the Hamburg Bit-Bots is depicted.

4.1 Data Collection

Our data collection process aims to gather information on the robot's movement and sensory input to train a neural network. We conduct controlled walks at various velocities to achieve this, recording sensor data and ground truth measurements. Specifically, we track the positions of the right and left soles, as well as the base link, in a global coordinate system.
The robot can receive walk commands at time t, which are comprised of three velocity components: Forward/backward movement (represented by the x-component $\dot{x_t}$), left/right movement (represented by the y-component $\dot{y_t}$), and rotation around the z-axis (represented by the z-component $\dot{w_t}$). To decide on the velocities to use, different sampling strategies are presented.
Our strategies for data collection differ between simulated and real-world environments. Thus, we provide separate explanations for each of them.

To cover the velocity space, we want to sample from the space of all feasible velocities:

$$V = \{(\dot{x_i}, \dot{y_i}, \dot{w_i}) |\ i \in \{1, ..., n\}\} \tag{22}$$

We also want the dataset to contain transitions of walking velocities and define a delta:

$$\Delta = \left\{ \left(\delta_{\dot{x}_i}, \delta_{\dot{y}_i}, \delta_{\dot{w}_i} \right) |\ i \in \{1, ..., n\} \right\} \tag{23}$$

Our sampling space is the image of the function:

$$\begin{gathered} s : V \times \Delta \to \mathbb{R}^3 \times \mathbb{R}^3, \\ \left((\dot{x}_i, \dot{y}_i, \dot{w}_i), \left(\delta_{\dot{x}_i}, \delta_{\dot{y}_i}, \delta_{\dot{w}_i} \right) \right) \mapsto \left((\dot{x}_i, \dot{y}_i, \dot{w}_i), \left(\dot{x}_i + \delta_{\dot{x}_i}, \dot{y}_i + \delta_{\dot{y}_i}, \dot{w}_i + \delta_{\dot{w}_i} \right) \right) \end{gathered} \tag{24}$$

4.1.1 Simulation

To generate data in simulation, we use the Webots simulator as described in Section 2.6.1 and collect rosbags of the robot's movement and sensor data.

First, we need to decide on a sampling strategy for simulation. It is easier to obtain a well-balanced dataset in simulation than in the real world as we do not have to worry about hardware breaking and are less limited by human resources.
The following sampling intervals are found empirically by testing the boundaries of walkable combinations in Webots:

$$\dot{x}_i \in [-0.4, 0.3], \dot{y}_i \in [-0.2, 0.2], \dot{w}_i \in [-0.8, 0.8] \tag{25}$$

The first strategy is sampling the space in a grid. This means choosing n points that are equally spaced to one another in the range of the intervals in Equation 25. Then, we combine each of them to obtain all possible combinations.
Thus, we get

$$V_{\text{grid}} = \{(x_1, y_1, w_1), (x_2, y_1, w_1), ..., (x_n, y_n, w_{n-1}), (x_n, y_n, w_n)\} \tag{26}$$

We can sample a total of m different transitions for grid sampling.

$$\Delta_{\text{grid}} = \left\{ \left(\delta_{\dot{x}_i}, \delta_{\dot{y}_i}, \delta_{\dot{w}_i} \right) |\ i \in \{1, ..., m\} \wedge \delta_{\dot{x}_i} \oplus \delta_{\dot{y}_i} \oplus \delta_{\dot{w}_i} \neq 0 \right\} \tag{27}$$

We put Δ_{grid} and V_{grid} into Equation 24 to obtain the sampling space.

The other sampling strategy is randomly sampling all feasible walking velocities n times. Then, we combine them to obtain:

$$V_{\text{rand}} = \{(x_1, y_1, w_1), (x_2, y_2, w_2), ..., (x_n, y_n, w_n)\} \tag{28}$$

Then, each sample gets its own randomly generated transition, which is within 25 percent of the size of the interval of feasible velocities.
Although the first strategy ensures that the whole space is covered at least punctually, we chose the second one because it can create a more diverse sampling space. As we can collect enough samples in simulation of different walking velocities, more advanced sampling strategies like Brownian motion are not used.

We simulated 10.000 runs, which take about seven days on a computer with an Nvidia GeForce RTX 2080 Ti with 24 CPU cores. The robot's model is the Wolfgang OP specified in a .proto file [Bes+21], which is generated from CAD files that do not contain manufacturing tolerances or deformations and, thus, are potentially inaccurate to some extent. The environment in which the robot walks is the digital soccer field seen in Figure 11.
The simulator runs in discrete time steps of which its step size is a tradeoff between computational expense and accuracy. It is set to eight milliseconds.
We use the Webots Supervisor [Sup24] to obtain the position of the robot's feet and base link. This is used to compute the label for the samples of the dataset in Section 4.2.
To collect the actual data, we record 14-second rosbags. To ensure obstacles like the goalposts do not influence the robot, it is always reset to the side of the playing field after the recording is finished.

4.1.2 Real world

We record data in the real world for training a neural network because, as explained in Section 2.6, simulating the environment has disadvantages like overfitting on the simulator. Additionally, we also use it to evaluate how well the odometry using a neural network trained on data from simulation can perform in the real world.

A motion capture system is used to record the robot's position. Our setup consists of Qualisys' Miqus M3 cameras [Qua20] placed at different points in a room, as seen in Figure 12. The cameras have LED rings around them, which emit infrared light that is reflected well by small spheres. Using a program called Qualisys Tracking Manager (QTM), we can calculate the positions of the spheres seen by the cameras after they have been calibrated.

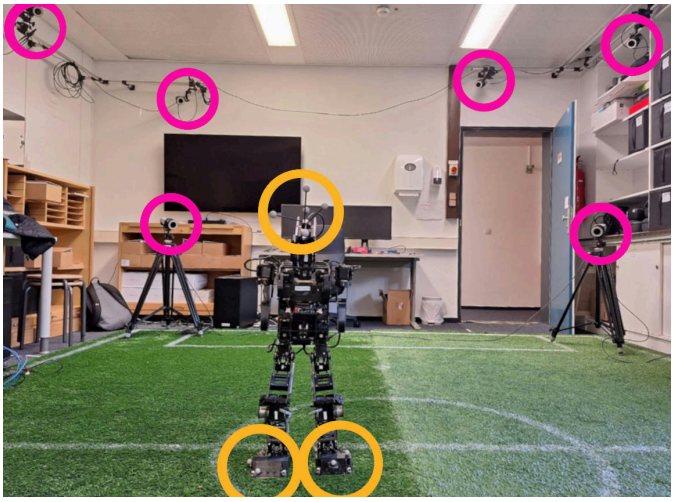


Figure 12: Motion capture set up with the robot and the artificial grass soccer field. The cameras are marked in magenta and the reflective spheres on the robot in orange. The cameras are placed in the same way for both sides of the field.

To capture the robots' position, we record the location of the robots' feet and head by placing reflective spheres in unique constellations on them, which are defined as rigid bodies in QTM. We record the head instead of the base link because the markers are more visible, and it is important that enough spheres can be detected. If there is too much occlusion, the rigid bodies cannot be located, rendering the recording useless.

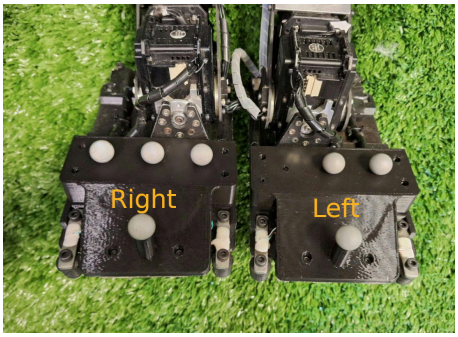


Figure 13: Reflective spheres on the robot's feet. The sphere constellations for the left and right foot differ so that their rigid body definitions can be distinguished.

As data collection in the real world is more difficult with robot breakages and limited time for recording, we do not have the option to record a plethora of data, and thus, we need to be more selective of the velocity space we sample.
Looking at the navigation used by the Hamburg Bit-Bots [Bit24], which sends the command velocities, we obtain different sampling intervals:

$$\dot{x_i} \in [-0.05, 0.1], \dot{y_i} \in [-0.04, 0.04], \dot{w_i} \in [-0.3, 0.3] \tag{29}$$

We also see that the maximum linear velocity is 0.15 m/s in total, which we do not want to overstep. We add an extra condition that caps the summed velocity of $\dot{x_i}$ and $\dot{y_i}$.
We first sample the x velocity from the defined space, which means for a randomly chosen $\dot{x_i}$ and $\dot{w_i}$ from the interval from Equation 29, we choose $\dot{y_i}$ the following way:

$$\begin{aligned} \dot{y_i} \in [-0.05 + \dot{x_i}, 0.1] \text{ if } x \geq 0 \\ \dot{y_i} \in [-0.05, 0.1 + \dot{x_i}] \text{ else} \end{aligned} \tag{30}$$

The angular velocity is handled separately as we found it to be less destabilizing and thus has no such constraint.
Similar to the simulated dataset, we want to include velocity changes. The path-planning transitions smoothly between velocities, and we want to emulate this by keeping δ relatively small. Additionally, walking on the artificial grass is less stable than in simulation and we do not want the robot to fall and break. Thus we choose a sample $\delta = (\delta_x, \delta_y, \delta_w)$ randomly from the following spaces:

$$\delta_x \in [-0.025, 0.05], \delta_y \in [0.02, 0.02], \delta_w \in [-0.1, 0.1] \tag{31}$$

Over 40 minutes of walking are recorded.
To perform these measurements we let the robot presented in Section 2.2 walk on a patch of artificial grass and record the information published by the robot's sensors in rosbags and the global pose by the motion capture system. After the recording, the global pose of the defined rigid bodies described by a rotation matrix and a position vector can be calculated and exported as a .tsv by QTM.

We briefly describe the workflow of recording the data in the real world:
1. A QTM recording and the sampling script are started.
2. The robot is placed on the field where it is visible to the motion capture system to avoid occlusions.
3. After pressing a button on the robot's shoulder, the robot walks with two different velocities. After it finishes, we continue with 2.

Some peculiarities of recording the data in the real world are mentioned in the following: First, we needed to make sure the time stamps of the ros messages and the recorded poses align and thus synced the clocks of the robot's and QTM's computer. To the best of our knowledge, the clocks had very similar times. Second, the robot never fell, as one person always followed the robot to stabilize it when it stumbled. These runs were noted down, but are still used for learning, as the dataset is small. Third, we also encountered hardware issues. The robot's actuators heated up during the recording, and the 3D-printed parts started to

deform, which led to worse walking performance. This was counteracted by taking breaks in between the recordings and letting them cool down. At the end of the recording session, the robot had problems with the hip actuators, and thus the quality of the data decreased.

Overall, we still think that the real-world data is valuable, as it can show how well the simulation-based odometry generalizes to the real world.

4.2 Data Processing

Before training the neural network, we preprocess the data. This involves reading information from the .mcap binary file, removing invalid data, selecting useful input features, and aligning the data with step times to predict step size. Finally, we implement the data in a format that can be used to train a neural network.

To remove invalid data, we check each for a fall detection. If a fall is detected, we discard data after the fall as it contains no useful information regarding the step size. In simulation, we detected 270 falls in 10,000 runs. Forty-five rosbags were rendered useless due to the robot falling at the start.
We select input features based on sensors mentioned in Section 2.3 and Chapter 3. We extract IMU data and forward kinematics data of tf2 transformations from rosbags. The transformation between the robot's soles best represents the actual step size as they are closest to the ground.
For simulated data, we extract ground truth pose from the simulation interface. For real-world data, we use poses calculated by the motion capture system, adjusting the head frame to approximate the base link.

We organized the data in pandas data frames[Mo10], which are a tabular data structure and enable fast data processing in Python. The Python subprocess library [Pyt24] was used to speed up the process of deserialization of data. In the end, the data frames of all processes are combined and saved as .feather files, which enable fast reading and writing [Gut].
Reading out 10,000 rosbags from simulation takes approximately 58 minutes on 24 CPU cores, while real-world data processing time is negligible.

With the data formatted, we narrowed the dataset to include only step times, which are defined as the switch of the current support foot, which is assumed to be the one closest to the ground, as shown in Figure 14.

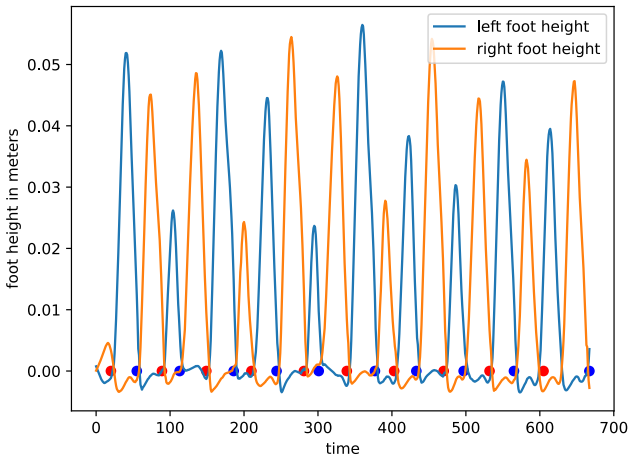


Figure 14: The height of the soles taken from ground truth is plotted while the robot walked. A right-to-left switch (Phase of -1) is shown as a blue dot, and a left-to-right switch (Phase of 1) is shown as a red dot. When the support foot switches, indicating both feet briefly made ground contact, a step has been taken.

We merge tables and resample data based on the step changes, addressing invalid data points through forward-filling interpolation. This ensures that the data at each time stamp contains the latest information, which was available at that point in time.
We define a phase for each step, which is saved based on whether the step was one from left to right (1) or right to left (-1).

For each step, we calculate the transformation from the previous support foot to the current one. Given the ground truth data containing transformations from the world frame to each foot frame (T_{WL} and T_{WR}), we derive the transformations from right to left foot (T_{LR}) and left to right foot (T_{RL}) as follows:

$$T_{\mathrm{LR}} = {T_{\mathrm{WL}}}^{-1} \cdot T_{\mathrm{WR}} \quad \text{and } T_{\mathrm{RL}} = {T_{\mathrm{WR}}}^{-1} \cdot T_{\mathrm{WL}} \tag{32}$$

Based on the computed phase, we retain either the left-to-right or right-to-left transformation and discard the other.

To simplify learning for the neural network, we convert rotation from quaternion to Euler angles, focusing solely on the yaw component, representing rotation around the z-axis for the transformation between soles for tf and the ground truth [Zho+19].

The resulting data frame comprises input features and corresponding labels for each row, saved to a feather file for use in the dataset outlined in Section 4.2.2. We can see that the data correlates as shown in Figure 15.

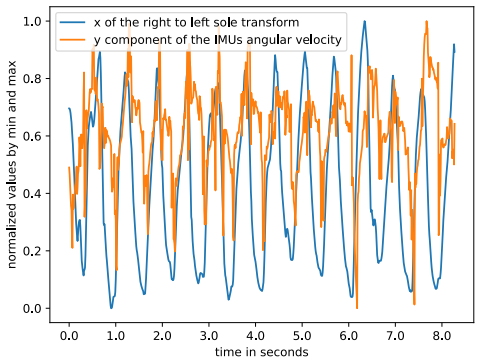


Figure 15: The x part of the ground-truth transform's translation from right to left sole and the angular velocity on the y axis measured by the IMU insimulationare depicted. They are normalized to make their correlation more visible. The data is taken from simulation.

The preprocessing parts also need to be done only once. For simulation, we cannot process all of the 10.000 rosbags at once, as the data is too large to fit into the RAM. From now on, we only use a quarter of the values for the simulation train dataset and another for the test dataset.

4.2.1 Normalization

Before implementing a dataset and training a neural network, we want to normalize the data to improve training speed [Agg18]. We use z-score normalization, which means for each input feature and label entry x in the dataset, we calculate the mean μ and the standard deviation σ and then use the normalized value z to pass into the neural network instead:

$$z = \frac{x - \mu}{\sigma} \tag{33}$$

We precompute the μ and σ for each feature and label and save them for the normalization of the train- and test dataset. The same values are also used in Section 4.4 to normalize the data when calculating the odometry on the robot.

4.2.2 PyTorch Dataset

For easier integration with PyTorch, a dataset is implemented as this can be used by a data loader, which batches and shuffles the data as explained in Section 2.4. We normalize the data using previously computed σ and μ from Section 4.2.1. To accommodate recurrent neural networks, we introduce the option to consider N subsequent features, grouping them into sequences using a sliding window ap-

proach. Each sample from the original data can be part of up to N sequences. To maintain context integrity, we avoid grouping data from different rosbags, ensuring sequences remain within the same walking context. The labels and input features are presented in Table 1 and Table 2.

Label	**Description**
x	The translation in x-direction of the transformation between the feet.
y	The translation in y-direction of the transformation between the feet.
yaw	The rotation around the z-axis of the transformation between the feet.

Table 1: The labels contain the x, y, and yaw of a transformation between the robot's feet from either left to right or right to left.

Feature	**Description**
IMU's angular velocity x	Angular velocity around the x-axis measured by the IMU.
IMU's angular velocity y	Same as above for y-axis, an example can be seen in Figure 15.
IMU's angular velocity z	Same as above for z-axis.
IMU's linear acceleration x	Linear acceleration in x-direction measured by the IMU.
IMU's linear acceleration y	Same as above for y-axis.
IMU's linear acceleration z	Same as above for z-axis.
IMU's orientation x	The x component of a quaternion of the orientation calculated from IMU data. A quaternion is a mathematical concept used to represent rotations in three-dimensional space.
IMU's orientation y	Same as above for y.
IMU's orientation z	Same as above for z.
IMU's orientation w	Same as above for w.
tf2transformof left to right sole x	Translation inx-directionof the tf2 transform between the left and right sole.
tf2 transfrom of left to right sole y	Translation in y-direction of the tf2 transform between the left and right sole.
tf2 transfrom of left to right sole yaw	Rotation around the z-axis of the tf2 transform between the left and right sole.
phase	The phase of the step, which is either -1 or 1.

Table 2: The 14 different input features for training a neural network with short descriptions. It is low dimensional compared to other problems like image recognition where neural networks are also commonly used

4.3 Neural Networks

We aim to predict the odometry of a humanoid robot by integrating neural network predictions of its step sizes. Our approach involves exploring different architectures, searching their corresponding hyperparameter space, and presenting the results of the best-performing network for this purpose.

4.3.1 Neural Network Architectures

We train various neural network architectures to determine their suitability for the problem domain. We consider standard multi-layer perceptrons (MLP), recurrent neural networks (RNN), and long short-term memory (LSTM) networks. MLPs are expected to learn static errors present in the transformation data and potentially obtain insights from IMU inputs. Recurrent architectures, stacked behind MLP layers, should capture temporal dependencies between steps as well.
The variation of the other parameters might also have an unforeseen positive effect on how well the trained model can predict, and thus, we try different options presented in Table 3 to further investigate which configuration of parameters from this subset learns the labels best.

Parameter	**Values**
Architecture	MLP, RNN, LSTM
Optimizer	Adam, SGD
Loss Function	MSE, MAE
Acitvaton Function	ReLU, tanh, sigmoid
Learning Rate	[0.0001, 0.1]
Dropout	[0.01, 0.15]
Epochs	[10, 50]
Number of Hidden Layers	[1, 5]
Layer Size	[4, 128]
Recurrent Size (LSTM/RNN only)	[4, 64]
Recurrent Depth (LSTM/RNN only)	[1, 16]
Batch Size	[128, 512]

Table 3: The hyperparameters used for the neural network architectures of which many are described in Section 2.4. The values presented in [·] are intervals.

PyTorch provides already implemented modules like recurrent layers to build a neural network, but the activation functions and optimizers are also shown in Table 3.

In the training process, after creating a neural network and using the desired hyperparameters, we proceed as follows:

1. We split our data into training and test sets using an 80/20 random split. For recurrent architectures, each sample consists of a sequence of five steps. The number of sequences varies based on the dataset. Insimulationfor recurrent architectures, we use 116,821 5-step sequences for the train dataset and 29,205 for the test dataset. For MLPs, we use 120,301 and 41.325 samples, respectively. For real-world data, we use 6994 5-step sequences for the train dataset and 1748 for the test dataset. For MLPs, we use 7174 and 1794 steps in each dataset.
2. We train the model to make predictions on input features and compute loss compared to the label. We employ a learning rate reduction strategy to adjust learning rates during training.
3. The model is then evaluated on the unseen test dataset to prevent overfitting.
4. Model weights and biases are periodically saved to a .pth file, and hyperparameters are stored in a yaml file for future reference.

The training and evaluation process is visualized using wandb [Bie20].

To find the best hyperparameter set, we use Optuna [Aki+19], which is a parameter optimization framework which can minimize an objective function. For that, a so-called study is created, which consists of running several trials in which we sample from the parameter set, which in our case is shown in Table 3. Optuna does not have to sample randomly but can apply more effective strategies like the Tree-structured Parzen Estimator [Oza+22], which we used. Our results are summarized in Table 4 in Section 5.2.

4.4 Integration

For the neural network that predicts the step sizes to be used in the RoboCup competition, it has to be integrated into the existing software stack of the Hamburg Bit-Bots. It can be used to compute a naive localization, which can function as the base for the particle filter in the RoboCup competition.
For that, we orient ourselves on the approach presented in Section 2.3 and call it *Model Odometry*. We implement a ROS node that takes the IMU data, the tf2 transformations from the robot's joints, and a walk support state as input and publishes a transformation from the origin to the base link.
A key challenge is accurate step detection, which is detailed in Section 4.4.1.

Upon starting the node, the specified model and its parameters containing architecture, as well as the normalization factors for input and output, are loaded.
A priority queue-like data structure is defined, which can return the closest mes-

sage to a time stamp given as a parameter and delete all older messages. We create subscribers for all the topics that publish the data we use as input to the model, as seen in Figure 16. This way, we can keep track of the current support foot and the IMU data at the time of the step. We also create a tf-listener to get the transformations between the left and right foot on a given time stamp.

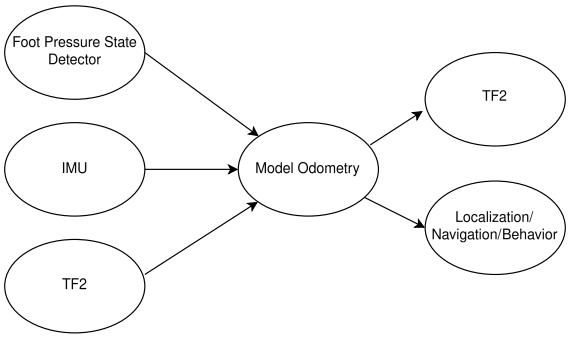


Figure 16: The model odometry takes input from the foot pressure state detector, IMU and the kinematics from TF2. The first two have callback functions in which we append the messages to the priority queue. It broadcasts a transformation from origin to base link on TF2 and publishes onto a topic that might be subscribed by components like localization.

When a step is detected, for which two approaches are presented in Section 4.4.1, the model predicts the step size and orientation. This is then used to update the transformation from the origin to the current support foot similar to Equation 6. The odometry should publish the transformation from the origin to the base link as in Equation 7. We approximate the transformation from the current support foot to the base link by using the tf2. For the locations of the links, refer to Figure 5.

4.4.1 Foot Pressure Step Detection

The model odometry depends on detecting steps accurately. The current node publishing the walk support state is not reliable due to its lack of sensory data integration, as evident in Figure 17.

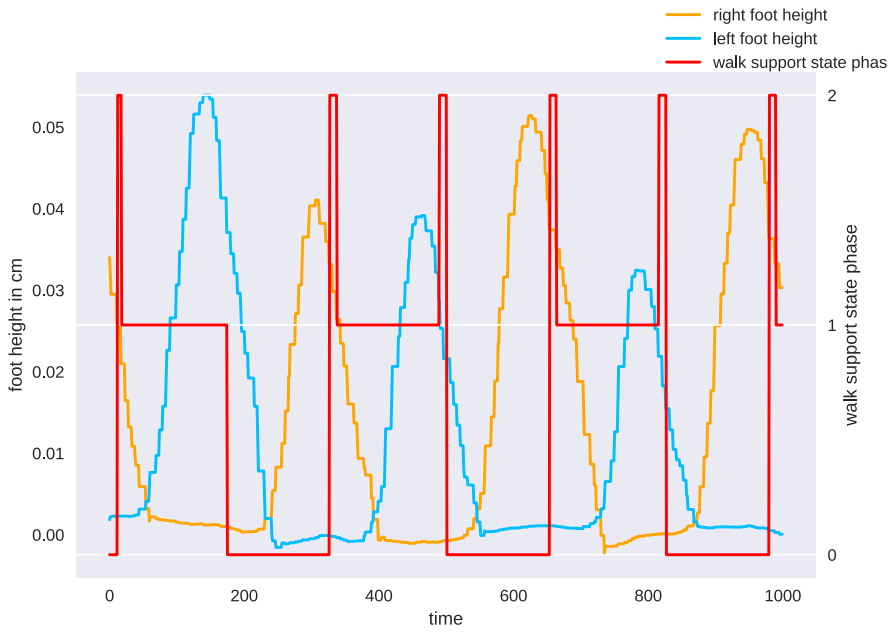


Figure 17: The walk support state and the height of the soles from Webots are visualized. We see the support state depicted in red alternating between the different states: right (0), left (1), and double support (2). The height of the left and right soles are depicted in blue and orange, respectively. It is plausible that the double support phase only happens for a short amount of time, but we see that it does not correlate with when both feet are on the ground (having a low height).

To address this, step detection based on foot pressure measurements is proposed. Each of the four cleats per foot publishes how much load they detect.
In Figure 18, we see that the summed pressure detected by the cleats of one foot is very high when the foot rises.

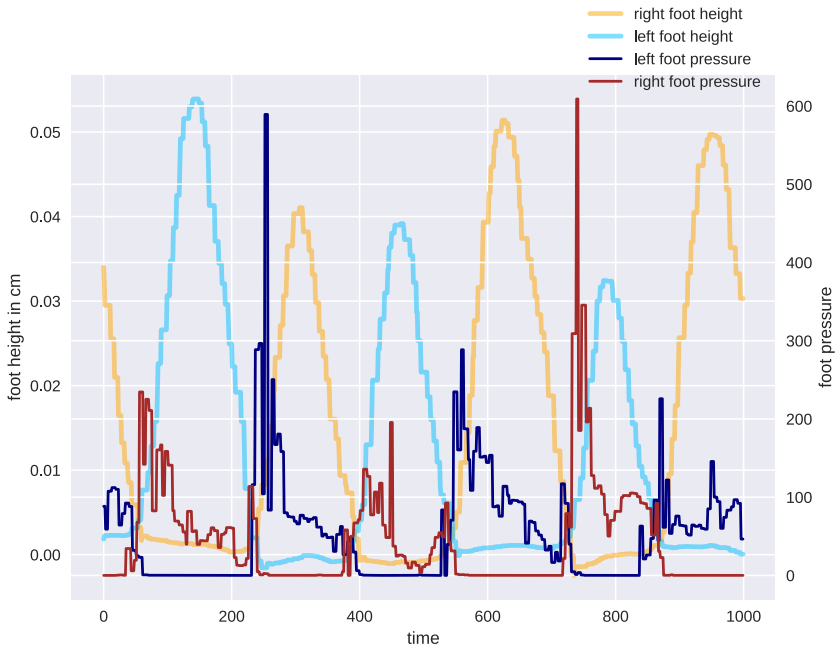


Figure 18: The cleats of the right foot are all depicted in brown and the cleats of the left foot in a dark blue. The height of the left foot is shown in blue, and the right foot in orange. For the time stamp 400, we see the right cleats detect high values as the left foot starts rising. On the other hand, when the cleats detect a very low load, we know that the other foot is currently in ground contact. If we can detect these maxima of pressure, we can reliably detect the double support phase and the switches. Data is taken from simulation.

Unfortunately, raw values from cleats differ in the real world, requiring calibration to ensure consistency. This calibration measures load absence and predefined load conditions to establish offsets and factors for filtering foot pressure.
A simple approach to detect the maxima is to sum the outputs of the cleats of each foot and threshold them. Before doing so, a low-pass filter, which keeps low-frequency components like the overall pressure detected when the foot is in ground contact but attenuates high frequencies like measurement noise, is applied.
However, determining an appropriate threshold for step detection proves challenging due to pressure variation with robot movement, especially in real-world scenarios.

An alternative approach is to use the information that the local minima of the pressure of one foot coincides with the ground contact of the other foot. We calculate the derivatives and identify the roots of the low-pass filtered sum of the cleats, which we assume to be a support foot switch. Though susceptible to noise, this approach offers improved accuracy compared to the original method, as demonstrated in Figure 19. We do not detect a double support phase anymore but implicitly assume them to be in between foot swap times.

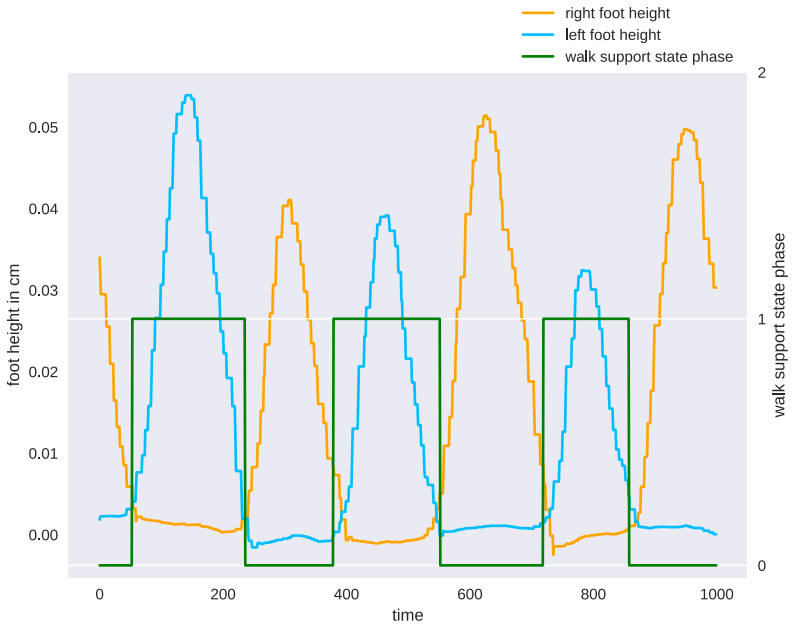


Figure 19: The walk support state is depicted in green. The right foot height in orange and the left foot height in blue. We see that the switch of phase happens when both feet are on the ground as at time 200. Sometimes it is a bit too early or late, as around time 400.

The foot pressure walk support state detector is implemented in C++ and launched with the bitbots motion.
We did not use this approach for label generation but to fully integrate the model odometry into the existing software stack.

5 Evaluation

In this Chapter, we want to evaluate the approach presented in Chapter 4. Firstly, we introduce two existing implementations of odometry utilized by the Hamburg Bit-Bots in the RoboCup domain, detailing their underlying models. Then, we describe the experiments conducted to assess the performance of our approach compared to these baselines. Finally, we present and discuss the results obtained from these experiments.

5.1 Baseline

We compare our odometry with two existing approaches used by the Hamburg Bit-Bots in the RoboCup Humanoid Soccer League. The two methods of odometry follow the same principle as laid out in Section 2.3 with Equation 6 and Equation 7.

The *Walk Engine Odometry* is calculated based on splines generated by the walk engine [BZ22]. To do so, it assumes the commanded end-effector poses for the feet to be reached at the time the step transitions are calculated in its internal model. The odometry itself is computed by the walk node, which handles the components needed for walking. It obtains the position of the support foot from the walk engine, which keeps track of the transformation from an origin for both feet. After every step, a calculated offset is added to the foot that took the step which represents Equation 6.
The robot's trunk functions as the base link. The support-foot-to-trunk transformation is also specified by the walk node's internal model. Then, odometry to base is computed as in Equation 7.
This odometry method does not use external measurements but only relies on the calculated splines for walking. It does not consider different step lengths or wiggling caused by the environment. The ground contact time is also not measured but computed and thus less accurate, but it fits this odometry from the timing.

The *Motion Odometry* is based on joint encoder measurements and IMU data at step transitions calculated by the walk engine. It uses transformations from sole to sole and from sole to base link obtained from the tf2 buffer. An odometry fuser node adjusts for tilting by calculating transformations from IMU measurements, affecting step lengths. At the node's initialization, the transformation from the origin to the support foot is set to the identity matrix and iteratively updated after each step, like in Equation 6. It continuously updates and publishes the transformation from origin to base link, as in Equation 7.
Despite similarities in step detection mechanism, simulation reveals issues with

this approach, as evident in Figure 20. However, both methods prove usable in real-world scenarios.

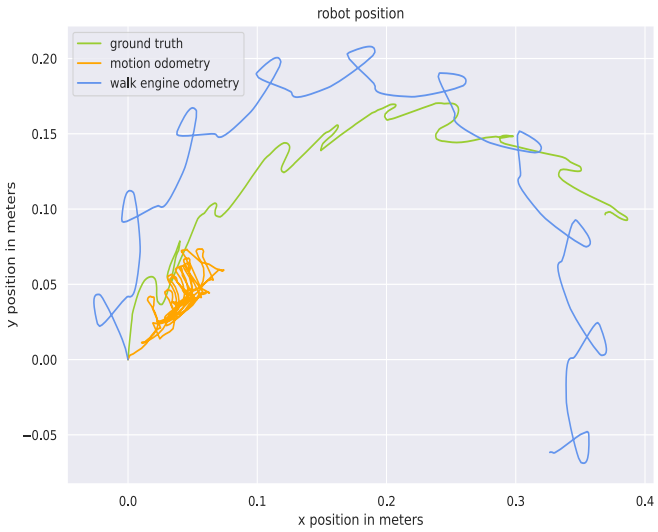


Figure 20: The walk engine and motion odometry plotted against the ground truth from simulation data. The base link of the walk engine odometry and the ground truth show sinusoidal movement due to the robot's walking pattern. The motion odometry performs very badly as the double support phases are detected without regard to the actual step taken, as described in Section 4.4.1. However, it works for model-based walk engine odometry.

5.2 Hyperparameters

In Section 4.3, we optimized various architectures and hyperparameters to develop a neural network capable of accurately predicting step sizes. Now, we present the best-performing architecture and hyperparameters for the neural network trained on simulation data with the one trained on real-world data and compare the two.

Parameter	SimNet	RealWorldNet
Architecture	LSTM	LSTM
Optimizer	Adam	Adam
Loss Function	MAE	MSE
Acitvaton Function	relu	tanh
Learning Rate	0.007	0.0261
Dropout	0.0688	0.1224
Number of Hidden Layers	1	1
Layer Size	87	125
Recurrent Size (LSTM/RNN only)	52	63
Recurrent Depth (LSTM/RNN only)	3	2
Epochs	81	85
Batch Size	512	497

Table 4: The table displays the best-performing sets of hyperparameters after 1000 trials. SimNet is trained on simulation data, and RealWorldNet is trained on real-world data. These parameters will be used for experiments in Section 5.3

As shown in Table 4, both the best-performing networks are LSTMs, indicating the importance of considering information from previous steps in predicting step sizes. Interestingly, they have similar numbers of epochs and batch sizes, possibly due to the LSTM's requirement for sufficient samples to adjust weights and biases effectively. On the other hand, this might not be the case as the real-world data was trained on a much smaller dataset and has a higher learning rate, which we will further discuss in the following.
The real-world network has a learning rate that is almost four times as high as the one of the simulation net. This might be due to the real-world dataset containing fewer samples and thus needing a higher learning rate to update its weights and biases. The dropout is also twice as high for the real-world net, which might be due to the real-world data being more noisy, and thus, the network can rely less on single input features.

Both neural networks feature a single fully connected linear layer preceding the LSTM layers, possibly indicating a nearly linear relationship between step sizes and input features. This alignment is supported by the similarity between tf transformations of soles and ground truth, as depicted in Figure 21.

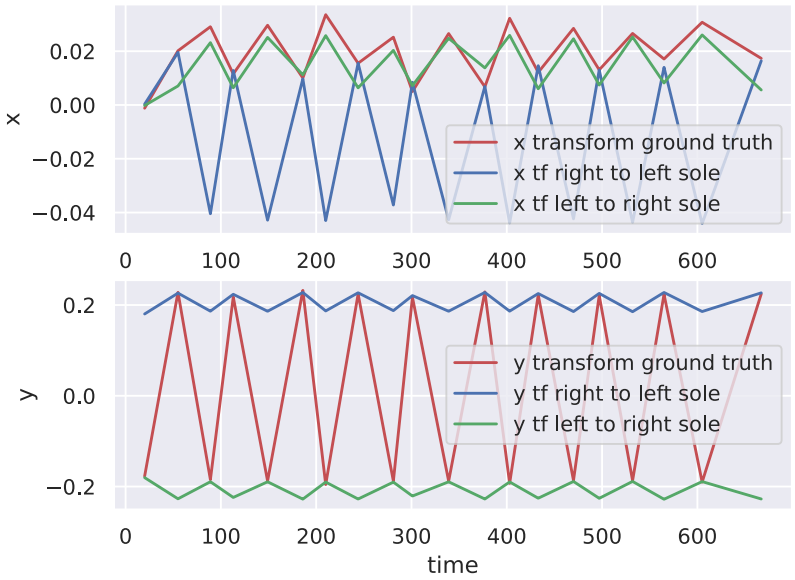


Figure 21: The plot illustrates values of the transform between soles according to tf and computed from simulation data. The red line represents the ground truth transform, ideally alternating between the right-to-left and left-to-right transform. While some deviation exists, particularly visible in the x-component, the overall similarity is evident.

The similarity in recurrent size and depth, as well as batch size and number of epochs, suggests comparable complexities in the models. However, differences in loss and activation functions may stem from variations in the datasets, but we cannot be sure of this.
Interestingly, the higher layer size and recurrent size in the real-world network may reflect the noisier nature of the real-world data, necessitating a greater number of neurons to effectively learn labels.

As LSTM models show the most promising results, the importance of previous steps in both simulation and real-world scenarios is shown. Despite the model trained on real-world data showing signs to account for more noisy input data, there is still a similarity in the architecture of the best-performing networks especially regarding the linear layers. This shows potential for a successful simulation-to-real transfer

5.3 Experiments

We want to assess the performance of the model odometry and the approaches presented in Section 5.1 relative to the ground truth over a specific timeframe. We

measure odometry drift over 5 seconds of walking, focusing on this short duration due to the disproportionate impact of incorrect rotations on overall estimation accuracy.
To evaluate our approach, referred to as *Model Odometry*, we select the best-performing neural networks based on Section 4.3.1 and test it on walk trajectories, which were not used in the Optuna optimization process.
We test *RealWorldNet* shown in Table 4 on real-world data and *SimNet* on both simulation and real-world data, as it is interesting to assess generalization to real-world scenarios.

We input the unseen prerecorded data into the neural network for predictions to obtain the model odometry estimate of the robot's pose after each step. We iteratively apply the transformation in Equation 6, resetting it at the start of each new run. To ensure a continuous transformation from an origin to the base link, we use the tf2 transformations from support sole to base to interpolate, similar to Section 4.4 and Equation 7.
Afterward, we load the Motion and Walk Engine Odometry and the ground truth, which all have fitting time stamps and bag numbers after the preprocessing in Section 4.2 and merge them.
We apply a sliding window with a 5-second size over the data, excluding sequences that involve a bag change. To ensure comparability, we align the pose estimations to start at the same position and angle. We then calculate the mean absolute deviation from x, y, yaw, and the Euclidean distance between each odometry estimate and the ground truth of the last pose of the sequence. This approach mitigates the influence of base link oscillations during the walk sequence on the results.

5.4 Results

We present the experiment results by assessing the total deviation relative to ground truth after 5 seconds of walking. Additionally, we showcase example trajectories and illustrate the distribution of deviation. Further analysis includes examining the median, spread of deviation, and identification of outliers.

5.4.1 SimNet on Simulation Data

First, we assess the Model Odometry using the SimNet on simulation data. A total of 15643 5-second sequences were utilized for evaluation purposes. The results of this evaluation are presented in the Table 5. There, we see that the model odometry outperforms both existing approaches. Specifically, the total deviation

in comparison to the walk engine odometry is reduced by over 80%, and compared to the motion odometry, it's decreased by almost 90%. Our approach exhibits only a 5-centimeter drift after 5 seconds, whereas the walk engine odometry displays a drift of over 27 centimeters. However, the motion odometry performs poorly in simulation due to discrepancies in the walk support phase and the actual step taken, as mentioned in Section 5.1 and Section 4.4.1, and is thus not a good comparison.

SimNet on sim data	Motion Odometry	Walk Engine Odometry	Model Odometry
total deviation in m after 5sec	0.4887	0.2761	**0.0506**
x deviation in m after 5sec	0.3428	0.1794	**0.0308**
y deviation in m after 5sec	0.2789	0.1716	**0.0334**
yaw deviation in rad after 5sec	1.1851	1.3609	**0.125**

Table 5: Performance of the model odometry using the SimNet on simulated data in comparison to the motion and walk engine odometry on unseen walk trajectories. The error is rounded to four decimal places. The best values are marked bold.

Example trajectories are illustrated in Figure 22 and Figure 24. The walk engine odometry tends to overestimate, possibly due to not considering current sensor measurements and external factors like friction. In contrast, our model odometry accurately predicts step lengths, resulting in closer proximity to the ground truth.

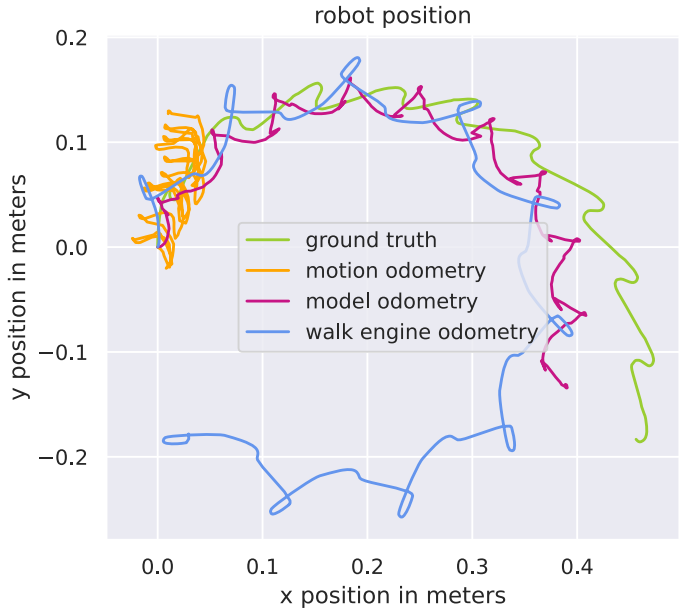


Figure 22: The different odometry approaches plotted against the ground truth. One box is 0.1 time 0.1 meters. The network used is SimNet, and it is evaluated on simulation data.

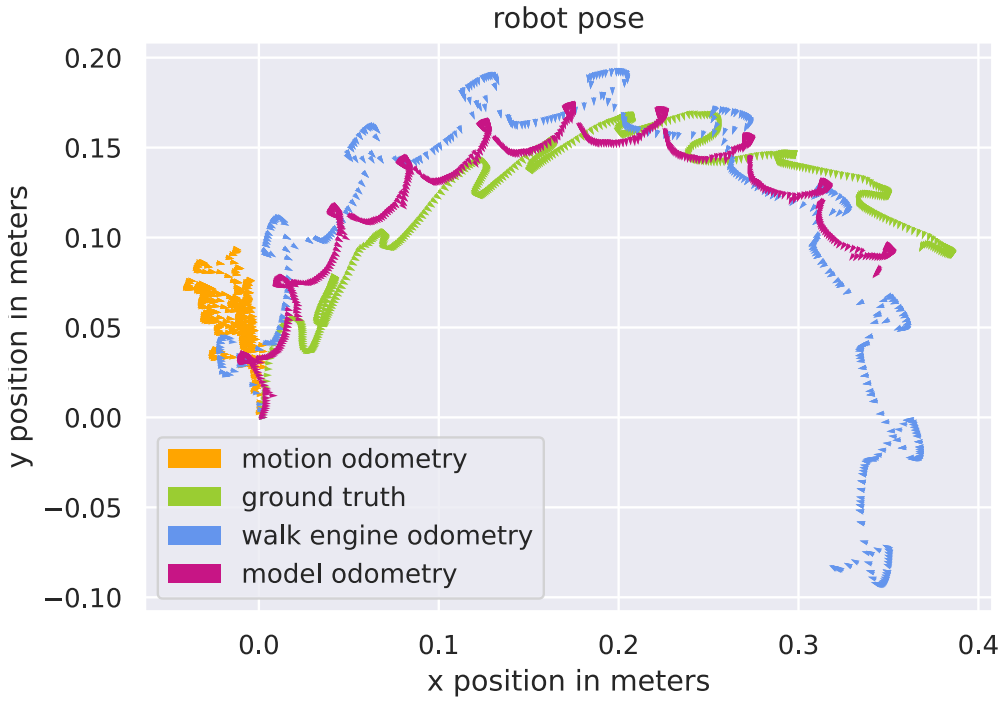


Figure 23: Quiver plot of the odometries compared with the ground truth. The model used is the SimNet. Here, we can see that the walk engine odometry aligns quite well with the ground truth. For the last pose, they happen to be orientated in different directions.

Very noticeable in Table 5 is the big difference in the rotational yaw component. The motion and walk engine odometry differ over one radiant from the ground truth, which is almost 60 degrees. Despite this, the walk engine odometry tends to be oriented correctly during walk sequences, as observed in Figure 22 and Figure 24. This discrepancy in the yaw component may stem from variations in base link definitions or step transition timings.
Figure 23 highlights discrepancies in the last pose orientation, indicating that yaw might not be the most reliable metric for assessing odometry performance in simulation. Therefore, we focus on the Euclidean distance to the ground truth in subsequent analyses.

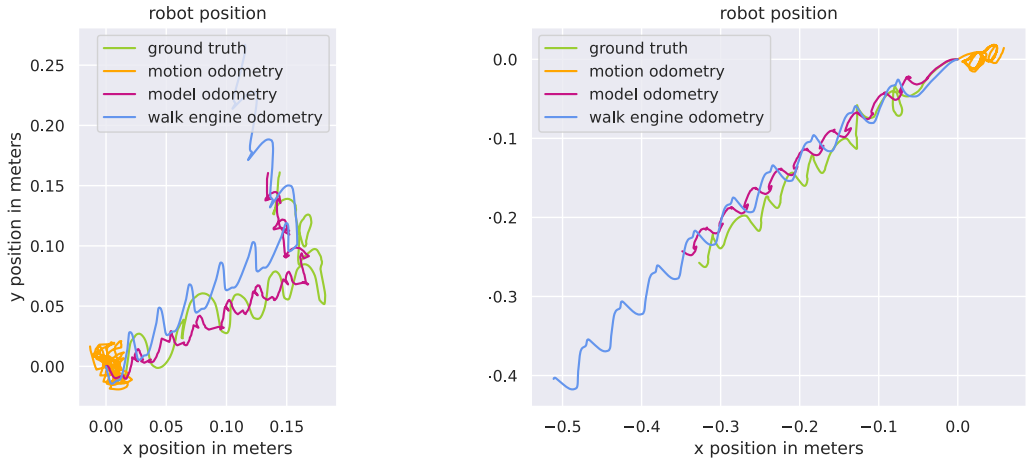


Figure 24: Walk trajectories of the different odometry approaches on simulation data using the SimNet.

Figure 25 presents the distribution of the distance to the ground truth after 5 seconds of walking. The model odometry exhibits a deviation of less than 10 centimeters in nearly all cases, with a median of around 5 centimeters. In contrast, the walk engine odometry has a median deviation of around 28 centimeters and shows more variability. The motion odometry tends to estimate the robot to stay close to the origin, which is often inaccurate.

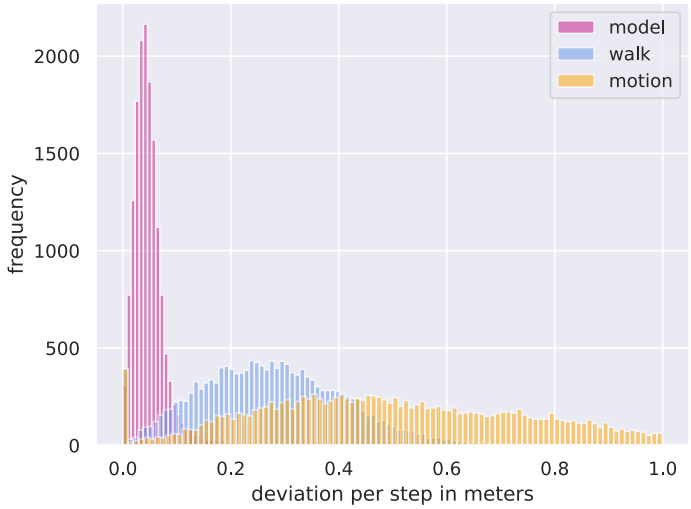


Figure 25: Histogram of the frequency of Euclidean distance to the ground truth from the different odometry approaches. The model used is the SimNet, and the data tested on is from simulation.

In Figure 26, the same data is depicted as a boxplot. For the model odometry, we see that there are indeed many outliers, but almost all values still lie in the normal estimation range of the walk engine odometry. The model odometry has a narrow interquartile range, and the median is close to the ground truth. This is a desirable property as it shows that the model odometry is reliable in its estimation.

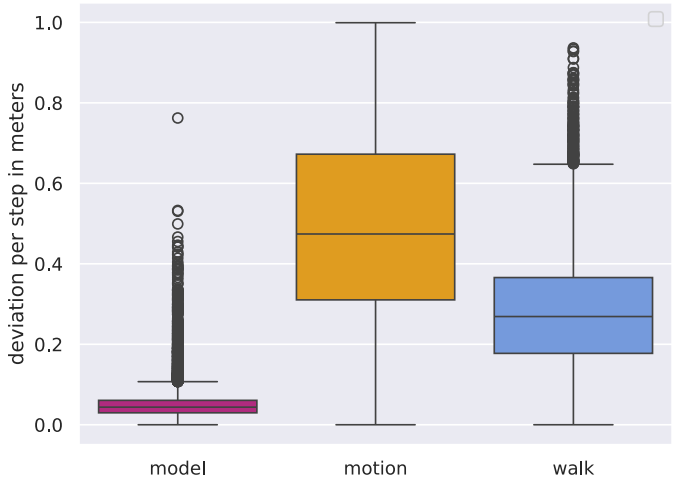


Figure 26: Boxplots of the different odometry approaches. The model used is the SimNet and the data evaluated on is from simulation.

In conclusion, employing a neural network to compute odometry based on predicted step sizes appears to be effective for simulation scenarios. The model odometry shows better performance than the existing approaches, which means more reliability and accuracy in estimation. In the context of the RoboCup Humanoid League Virtual Season, using the model odometry might improve the overall playing of the robots.

5.4.2 RealWorldNet on Real-World Data

Now, the performance of Real WorldNet on real-world data is shown. We again present the deviations of five-second sequences in Table 6, for which the RealWorldNet was tested on 775 5-second sequences of unseen data from the real world. The relatively small size of this unseen data must be considered, potentially limiting the representativeness of the results. While the total duration of these sequences exceeds 64 minutes, it's important to note that they were created using a sliding window approach, which means one second might appear in up to five different sequences. The unique sequences amount to roughly 13 minutes.

The model odometry still outperforms the other two approaches in terms of total deviation, although not as significantly as in simulation. It shows an increase in accuracy on the total deviation of almost 21% and 14.8% compared to the motion odometry and the walk engine odometry, respectively.

RealNet on real data	Motion Odometry	Walk Engine Odometry	Model Odometry
total deviation in m after 5sec	0.2269	0.2107	**0.1795**
x deviation in m after 5sec	0.16	0.1328	**0.097**
y deviation in m after 5sec	0.1258	0.1321	**0.1285**
yaw deviation in rad after 5sec	0.29	0.4961	**0.287**

Table 6: Performance of the model odometry using the RealWorldNet on real-world data in comparison to the motion and walk engine odometry on unseen walk trajectories. The error is rounded to four decimal places. The best values are marked in bold.

Both baseline approaches perform better in the real world compared to simulation, which could be attributed to the software initially being developed for the actual robot rather than for simulation purposes. Particularly, the motion odometry performs notably better than in simulation, likely because the estimation of the walk support phase aligns more closely with actual steps in the real world.

Figure 27 shows an example trajectory in which the model odometry ends close to the ground truth, although the path the base link walks along is less fitting than, for example, the motion odometry. This is probably due to the way we approximate transformation from sole to base link as described in Section 5.3. The current position is still based on the summation of the steps. We approximated the actual position of the base link in the ground truth data by the position of the head and then transformed it downwards, as mentioned Section 4.2. This might also be why the head is oscillating more than the actual base link would and might lead to the big oscillations.

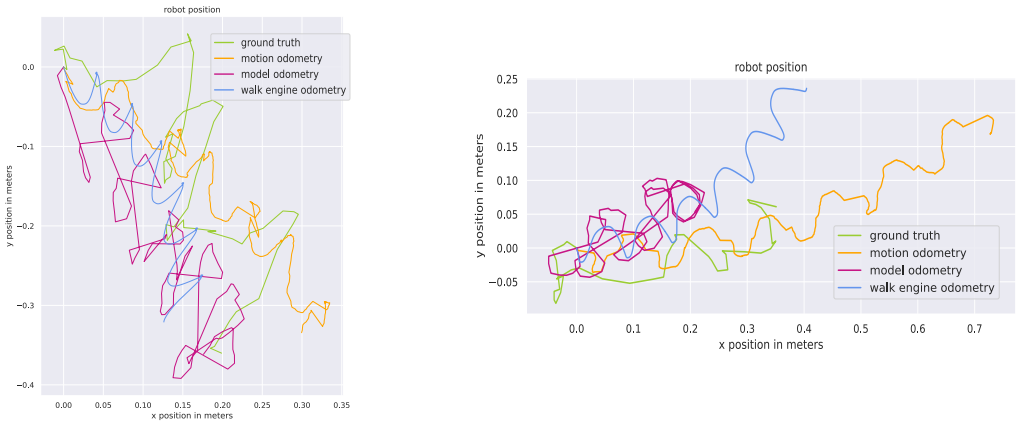


Figure 27: Walk trajectories using the RealWorldNet on real-world data in comparison to the motion and walk engine odometry on unseen data. We see that all odometry approaches can perform well on the left. Maybe due to slippages on the grass, the baselines overestimate, as seen on the right.

Again, we visualize the distribution of deviation in Figure 28. The model odometry still performs best but is less accurate than in simulation, as seen in Figure 25. This is probably at least partially due to the small dataset. The motion and walk engine odometry perform similarly to each other. They have very small deviations from the ground truth than the model odometry, but they also have more inaccurate estimations.

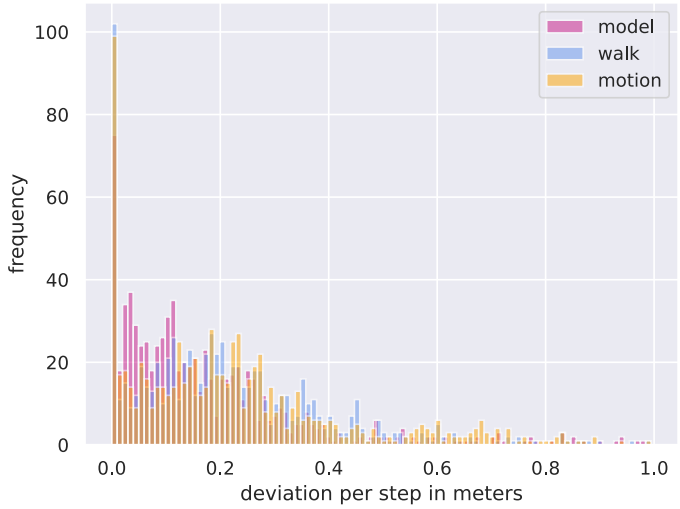


Figure 28: Histogram of the frequency of Euclidean distance to the ground from the odometry approaches in the real world. The model used is the RealWorldNet.

Looking at Figure 29, the model odometry's median is closer to the ground truth than the other two, with a slightly narrower interquartile range, indicating slightly better reliability. However, all odometry approaches exhibit some outliers.

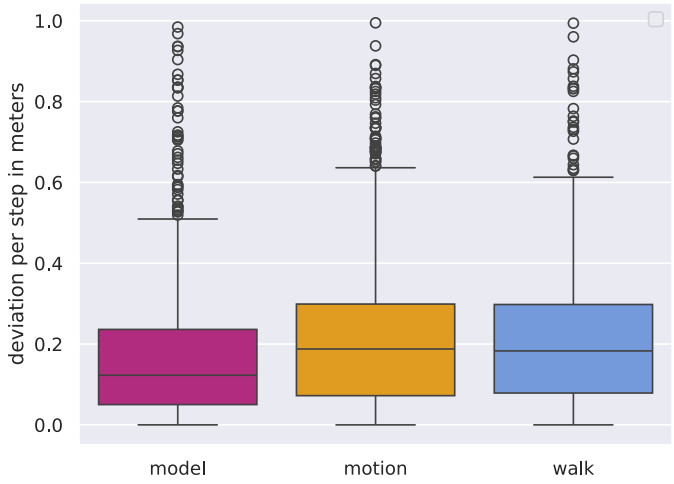


Figure 29: Boxplots of the different odometry approaches in the real world. The model used is the RealWorldNet.

In summary, despite being trained on a small dataset, the model odometry remains the best-performing approach in the real world, offering improved accuracy and reliability compared to the other two approaches, even though the motion odometry performs better in real-world conditions than in simulation.

5.4.3 SimNet on Real-World Data

In the following, we assess the performance of the model odometry using the SimNet on real-world data. This analysis is interesting as it provides insights into the model's transfer from simulation to the real world.
As all the data generated in the real world is unseen to this model, we have more samples we can evaluate and thus use 3235 5-second sequences in this comparison. Observations from Table 7 mirror those from Table 6, highlighting that despite the real-world setting, the model odometry performs best in total deviation compared to both motion and walk engine odometry. Notably, the model odometry shows an improvement of at least 12% compared to the motion odometry and 3% compared to the walk engine odometry.

SimNet on real data	Motion Odometry	Walk Engine Odometry	Model Odometry
total deviation in m after 5sec	0.2041	0.1849	**0.1794**
x deviation in m after 5sec	0.1433	0.1214	**0.1276**
y deviation in m after 5sec	0.1144	0.1142	**0.0989**
yaw deviation in rad after 5sec	**0.3067**	0.4921	0.3427

Table 7: Performance of the model odometry using the SimNet in comparison to the motion and walk engine odometry on unseen walk trajectories in the real world. The error is rounded to four decimal places. The best values are marked in bold.

Additionally, Table 8 showcases the comparison between the dataset used in Section 5.4.2 and the network trained on real-world data. The model odometry employing the RealWorldNet outperforms its counterpart trained on simulation data. This difference likely stems from the RealWorldNet being trained on a more representative dataset. The sensor measurements collected in simulation might not align with the actual sensors. Additionally, as noted in Section 4.1, potential hardware malfunctions in the robot toward the end of data collection could further amplify the disparities between simulation and real-world performance.

SimNet and RealWorldNet on real data	Motion Odometry	Walk Engine Odometry	[SimNet] Odometry	[RealWorld-Net] Odometry
total deviation in m after 5sec	0.2269	0.2107	0.2062	**0.1795**
x deviation in m after 5sec	0.16	0.1328	0.1369	**0.097**
y deviation in m after 5sec	0.1258	0.1321	**0.1221**	0.1285
yaw deviation in rad after 5sec	0.29	0.4961	0.4524	**0.287**

Table 8: Performance of the model odometry using the SimNet and RealWorldNet on real-world data. The error is rounded to four decimal places. The best values are marked in bold.

The histogram in Figure 30 and the boxplot in Figure 31 look similar to Figure 28 and Figure 29 respectively with regards to the relation of estimation differences between the different odometry approaches. We do notice the trends from Table 7 in less overall accuracy for the model odometry compared to simulation, probably due to sensor measurement values not aligning.

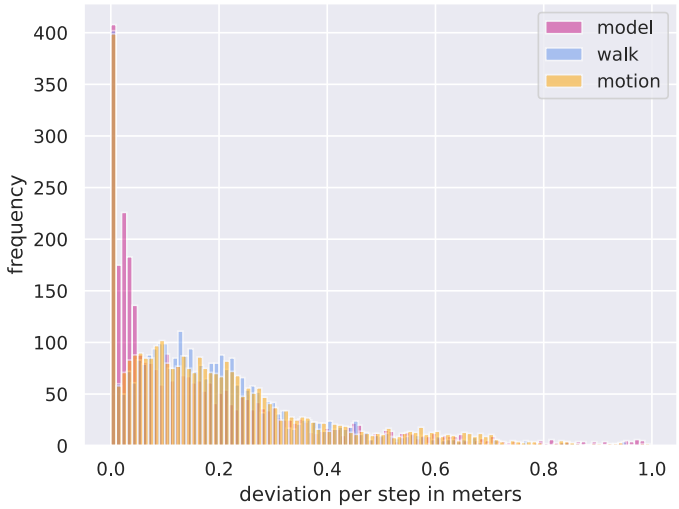


Figure 30: Histogram of the frequency of Euclidean distance to the ground from the odometry approaches in the real world. The model used is the SimNet.

Figure 31 illustrates that while the distribution remains similar, the model odometry's interquartile range is larger, and the median is further from the ground truth, indicating reduced accuracy and reliability compared to simulation and the RealWorldNet.

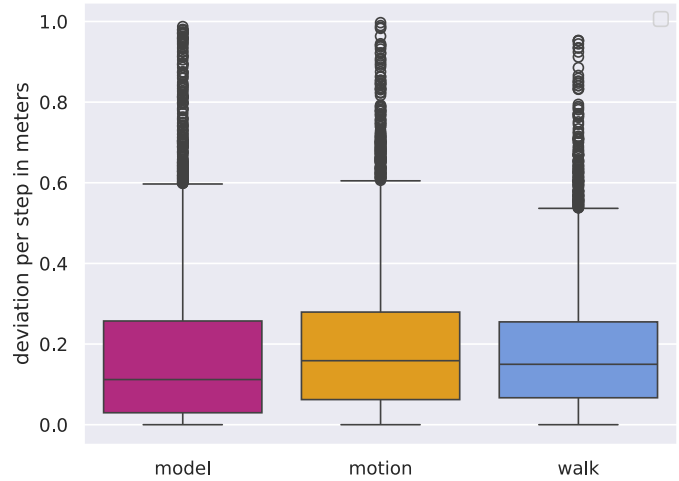


Figure 31: Boxplots of the different odometry approaches tested on real-world data. The model used is the SimNet.

Figure 32 illustrates instances where the model odometry estimates reasonably well, while other approaches, the walk engine odometry in this case, perform worse.

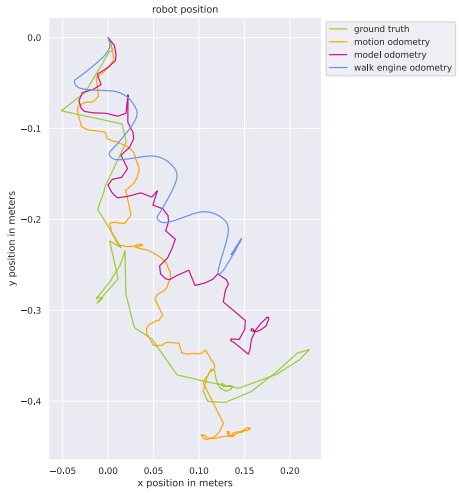


Figure 32: Trajectory of the three odometry approaches compared to the ground truth on real-world data. The SimNet is used.

In the real world, the model odometry using the SimNet is still the best-performing odometry approach, but only marginally. This indicates the feasibility of transferring insights from simulation-trained models to real-world scenarios. However, further research has to be conducted to fully leverage the possibilities depicted in Section 5.4.1, where the model odometry reaches very accurate results.

6 Conclusion

In this thesis, we proposed a lightweight odometry based on a neural network called model odometry, which uses proprioceptive sensor data to predict step sizes and orientations, which are then integrated to obtain the robot's pose.
For that, we first generated simulation and the real world data and processed them fitting our needs. Afterwards, we performed optimization to find the best configuration of hyperparameters for a neural network based on our problem. We integrated the neural network into the existing software stack of the Hamburg Bit-Bots and proposed a new step-detection mechanism based on foot pressure measurements to make the odometry reliable.
Finally, we evaluated our approach on 5-second walking sequences with two approaches currently used by the Hamburg Bit-Bots. We demonstrated that the model odometry can be trained on simulation data, transferred to the real world without adjustment, and still outperform existing methods. For a RoboCup Humanoid Soccer match, the model odometry could be used to improve the robots' playing as it enables better positioning.
Notably, the model odometry demonstrates high accuracy when utilized in simulation scenarios, making it feasible to use in the RoboCup Humanoid Soccer League Virtual Season.

There are several directions for future work.
First, many approaches, as seen in Chapter 3, calculate velocity factors to estimate directional velocity but are typically designed for quadruped robots. Adapting such an approach to a biped robot and evaluating the computational expense and performance could be a viable next step.
Second, we chose the input features for the network based on what seemed reasonable to us. The next step involves conducting an ablation study to identify the most influential features for network performance.
Third, as we only used IMU data from one point in time in the input vector for our neural network, we should try to incorporate data over larger time frames to estimate slippages better.
Furthermore, as the real-world performance of the walk engine and motion odometry was good, we could use their estimate as input for the neural network to improve the model odometry. Even for simulation, the walk engine odometry overscaled the data but usually estimated decently.
Moreover, we also showed that the network trained on data in simulation also makes good estimations in the real world but lacks in comparison to its performance in simulation. It could be tried to pre-train a model on data from simulation and then slightly adjust the weights by training on data from the real world with a low learning rate. Another possibility would be to adjust the model odometry's

parameters online.
Lastly, to get a stable odometry on the real robot, the model odometry, together with the footstep detection, should be tested thoroughly as it highly depends on how well steps are detected.

Bibliography

[HD99] S. Huang and G. Dissanayake, "Robot localization: An introduction," *Wiley Encyclopedia of Electrical and Electronics Engineering*, pp. 1–10, 1999.

[Bit24] "Bit-Bots Website." Available: https://bit-bots.de/en/. [Accessed: Apr. 25, 2024]

[SK16] B. Siciliano and O. Khatib, *Robotics and the Handbook*. Springer, 2016.

[Del+99] F. Dellaert, D. Fox, W. Burgard, and S. Thrun, "Monte carlo localization for mobile robots," in *Proceedings 1999 IEEE international conference on robotics and automation (Cat. No. 99CH36288C)*, 1999, pp. 1322–1328.

[Thr+01] S. Thrun, D. Fox, W. Burgard, and F. Dellaert, "Robust Monte Carlo localization for mobile robots," *Artificial intelligence*, vol. 128, no. 1–2, pp. 99–141, 2001.

[Fed24] R. Federation, "A Brief History of RoboCup," 2024. Available: https://www.robocup.org/a_brief_history_of_robocup. [Accessed: Jan. 18, 2024]

[Fed23] R. Federation, "RoboCup Humanoid League 2023 Rules," 2023. Available: http://humanoid.robocup.org/wp-content/uploads/RC-HL-2023-Rules.pdf. [Accessed: Jan. 18, 2024]

[Bes+21] M. Bestmann, J. Güldenstein, F. Vahl, and J. Zhang, "Wolfgang-op: a robust humanoid robot platform for research and competitions," in *2020 IEEE-RAS 20th International Conference on Humanoid Robots (Humanoids)*, 2021, pp. 90–97.

[MX-24] "MX-64 DYNAMIXEL." Available: https://emanual.robotis.com/docs/en/dxl/mx/mx-64/. [Accessed: Feb. 24, 2024]

[BM18] M. Ben-Ari and F. Mondada, "Robotic motion and odometry," Springer, 2018, pp. 63–93.

[LP17] K. M. Lynch and F. C. Park, *Modern robotics*. Cambridge University Press, 2017.

[Ble+18] G. Bledt, M. J. Powell, B. Katz, J. Di Carlo, P. M. Wensing, and S. Kim, "Mit cheetah 3: Design and control of a robust, dynamic quadruped robot," in *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2018, pp. 2245–2252.

[Blo+15] M. Bloesch, S. Omari, M. Hutter, and R. Siegwart, "Robust visual inertial odometry using a direct EKF-based approach," in *2015 IEEE/RSJ international conference on intelligent robots and systems (IROS)*, 2015, pp. 298–304.

[Cam+20] M. Camurri, M. Ramezani, S. Nobili, and M. Fallon, "Pronto: A multi-sensor state estimator for legged robots in real-world scenarios," *Frontiers in Robotics and AI*, vol. 7, p. 68–69, 2020.

[Har+18] R. Hartley, M. G. Jadidi, J. W. Grizzle, and R. M. Eustice, "Contact-aided invariant extended Kalman filtering for legged robot state estimation," *arXiv preprint arXiv:1805.10410*, 2018.

[Yan+23] S. Yang, Z. Zhang, B. Bokser, and Z. Manchester, "Multi-IMU Proprioceptive Odometry for Legged Robots," in *2023 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2023, pp. 774–779.

[Aqe+16] M. O. Aqel, M. H. Marhaban, M. I. Saripan, and N. B. Ismail, "Review of visual odometry: types, approaches, challenges, and applications," *SpringerPlus*, vol. 5, pp. 1–26, 2016.

[WCF22] D. Wisth, M. Camurri, and M. Fallon, "VILENS: Visual, inertial, lidar, and leg odometry for all-terrain legged robots," *IEEE Transactions on Robotics*, vol. 39, no. 1, pp. 309–326, 2022.

[Pla23] M. Plaue, *Data Science: An Introduction to Statistics and Machine Learning*. Springer Nature, 2023.

[Agg18] C. C. Aggarwal, "Neural networks and deep learning." Springer, 2018.

[KB14] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[HS97] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[Lon24] "Long short-term memory." Available: https://en.wikipedia.org/wiki/Long_short-term_memory. [Accessed: May 01, 2024]

[Pas+19] A. Paszke *et al.*, "Pytorch: An imperative style, high-performance deep learning library," vol. 32. 2019.

[Mac+22] S. Macenski, T. Foote, B. Gerkey, C. Lalancette, and W. Woodall, "Robot Operating System 2: Design, architecture, and uses in the wild. Science Robotics 7, 66 (2022), eabm6074," *Jin Seob Kim (Member, IEEE) received ysis*, 2022.

[Dis24] "Distributions ROS." Available: http://wiki.ros.org/Distributions,. [Accessed: Jan. 18, 2024]

[ROS24] "ROS Rolling Release Documentation." Available: https://docs.ros.org/en/rolling/Releases.html. [Accessed: Jan. 18, 2024]

[Fac24] D. Faconti, "PlotJuggler," 2024. Available: https://github.com/facontidavide/PlotJuggler.git. [Accessed: Feb. 04, 2024]

[RVi] "RViz: 3D Visualization Tool for ROS 2." Available: https://github.com/ros2/rviz

[ROS24] "ROS 2 Foxy Fitzroy - RQt Overview." Available: https://docs.ros.org/en/foxy/Concepts/About-RQt.html#overview. [Accessed: Jan. 18, 2024]

[Web24] Webots, "Open-source Mobile Robot Simulation Software." Available: http://www.cyberbotics.com/. [Accessed: Jan. 15, 2024]

[Sup24] "Supervisor Node Webots." Available: https://cyberbotics.com/doc/reference/supervisor#wb_supervisor_node_get_root. [Accessed: Feb. 23, 2024]

[Dru+10] E. Drumwright, J. Hsu, N. Koenig, and D. Shell, "Extending open dynamics engine for robotics simulation," in *Simulation, Modeling, and Programming for Autonomous Robots: Second International Conference, SIMPAR 2010, Darmstadt, Germany, November 15-18, 2010. Proceedings 2*, 2010, pp. 38–50.

[WBo95] G. Welch, G. Bishop, and others, "An introduction to the Kalman filter," 1995.

[JU97] S. J. Julier and J. K. Uhlmann, "New extension of the Kalman filter to nonlinear systems," in *Signal processing, sensor fusion, and target recognition VI*, 1997, pp. 182–193.

[Blo+13] M. Bloesch *et al.*, "State estimation for legged robots-consistent fusion of leg kinematics and IMU," *Robotics*, vol. 17, pp. 17–24, 2013.

[Rot+14] N. Rotella, M. Bloesch, L. Righetti, and S. Schaal, "State estimation for a humanoid robot," in *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2014, pp. 952–958.

[WCF20] D. Wisth, M. Camurri, and M. Fallon, "Preintegrated velocity bias estimation to overcome contact nonlinearities in legged robot odometry," in *2020 IEEE International Conference on Robotics and Automation (ICRA)*, 2020, pp. 392–398.

[Wan+17] S. Wang, R. Clark, H. Wen, and N. Trigoni, "Deepvo: Towards end-to-end visual odometry with deep recurrent convolutional neural networks," in *2017 IEEE international conference on robotics and automation (ICRA)*, 2017, pp. 2043–2050.

[Che+18] C. Chen, X. Lu, A. Markham, and N. Trigoni, "Ionet: Learning to cure the curse of drift in inertial odometry," in *Proceedings of the AAAI Conference on Artificial Intelligence*, 2018.

[Liu+20] W. Liu *et al.*, "Tlio: Tight learned inertial odometry," *IEEE Robotics and Automation Letters*, vol. 5, no. 4, pp. 5653–5660, 2020.

[Buc+22] R. Buchanan, M. Camurri, F. Dellaert, and M. Fallon, "Learning inertial odometry for dynamic legged robot state estimation," in *Conference on robot learning*, 2022, pp. 1575–1584.

[Buc+22] R. Buchanan, V. Agrawal, M. Camurri, F. Dellaert, and M. Fallon, "Deep imu bias inference for robust visual-inertial odometry with factor graphs," *IEEE Robotics and Automation Letters*, vol. 8, no. 1, pp. 41–48, 2022.

[Rou+16] Q. Rouxel, G. Passault, L. Hofer, S. N'Guyen, and O. Ly, "Learning the odometry on a small humanoid robot," in *2016 IEEE International Conference on Robotics and Automation (ICRA)*, 2016, pp. 1810–1816.

[Qua20] "Qualisys Miqus cameras," 2020. Available: https://cdn-content.qualisys.com/2020/01/PI_Miqus.pdf. [Accessed: Mar. 16, 2024]

[Bit24] Bit-Bots, "Path Planning Configuration Bit-Bots." Available: https://github.com/bit-bots/bitbots_main/blob/main/bitbots_navigation/bitbots_path_planning/config/path_planning.yaml. [Accessed: Mar. 15, 2024]

[Mo10] W. McKinney and others, "Data structures for statistical computing in Python.," in *SciPy*, 2010, pp. 51–56.

[Pyt24] "asyncio subprocess - Subprocess support for asyncio." Available: https://docs.python.org/3/library/asyncio-subprocess.html. [Accessed: Jan. 19, 2024]

[Gut] J. Gutsche, "Quantifying Player Performance in Simulated Humanoid Robot Soccer Games."

[Zho+19] Y. Zhou, C. Barnes, J. Lu, J. Yang, and H. Li, "On the continuity of rotation representations in neural networks," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2019, pp. 5745–5753.

[Bie20] L. Biewald, “Experiment Tracking with Weights and Biases,” 2020. Available: https://www.wandb.com/. [Accessed: Feb. 04, 2024]

[Aki+19] T. Akiba, S. Sano, T. Yanase, T. Ohta, and M. Koyama, “Optuna: A next-generation hyperparameter optimization framework,” in *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*, 2019, pp. 2623–2631.

[Oza+22] Y. Ozaki, Y. Tanigaki, S. Watanabe, M. Nomura, and M. Onishi, “Multiobjective tree-structured Parzen estimator,” *Journal of Artificial Intelligence Research*, vol. 73, pp. 1209–1250, 2022.

[BZ22] M. Bestmann and J. Zhang, “Bipedal walking on humanoid robots through parameter optimization,” *Robot World Cup*. Springer, pp. 164–176, 2022.

Eidesstattliche Erklärung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit im Bachelorstudiengang Informatik selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel – insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen – benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe.

Halstenbek, den 1. Mai 2024 (Valerie Bartel)

Veröffentlichung

Ich stimme der Einstellung der Arbeit in die Bibliothek des Fachbereichs Informatik zu.

Halstenbek, den 1. Mai 2024 (Valerie Bartel)