University of Applied Sciences Koblenz

Department for Electrical and Information Engineering

Prof. Dipl.-Inf. H.J. Unkelbach

University of Western Australia

School of Electrical, Electronic and Computer Engineering

Prof. Dr. rer. nat. habil. T. Bräunl

Diplomarbeit

Balancing of a Biped Robot Using Force Feedback

Jochen Zimmermann

Student # 494881

ABSTRACT

This thesis presents an active balance control for a fully autonomous bipedal robot. The robot has 10 kinematic degrees of freedom and is driven by servos. The Denavit-Hartenberg-Notation was applied to describe the machine's kinematics. Force sensors in the feet are used to gather information on the masses' gravitational distribution. To stabilise the walker, a closed loop PID control was implemented in an interrupt using the force sensors' feedback. The software is designed in an object-orientated layered architecture to provide effective future-work integration. The robot is able to balance on a moving level surface.

KURZFASSUNG

In dieser Diplomarbeit wird eine aktive Balanceregelung für einen völlig autonomen zweibeinigen Roboter präsentiert. Das System besitzt 10 kinematische Freiheitsgrade und der Roboter wird mittels Servos bewegt. Die Denavit-Hartenberg-Notation wurde angewendet, um die kinematischen Zustände des Gerätes zu beschreiben. Kraftsensoren in den Füßen liefern Daten über die Schwerkraftverteilung der Massen. Um die Gehmaschine zu stabilisieren, wurde eine PID-Regelung implementiert, die in einem Interrupt läuft und rückgekoppelte Daten der Kraftsensoren verarbeitet. Die Software wurde objektorientiert und in Schichten aufgebaut, um zukünftige Erweiterungen zu vereinfachen. Der Roboter ist in der Lage, auf einem sich bewegenden Untergrund zu balancieren.

ACKNOWLEDGEMENTS

First, I would like to thank my supervisor Prof. Thomas Bräunl who gave me the opportunity to work on this project at the University of Western Australia.

I also would like to thank Prof. Heinz Unkelbach not only for supervising this project from my home university in Koblenz, but also for his support and advice throughout the whole project. Only his support made it possible to realise this project.

For his assistance during verifying the mathematical result, I would like to thank Prof. Armin Saam.

Many thanks to the UWA staff for their assistance and for helping me to orientate in the department.

Taking the decision about carrying out this project was easier with the advice of my family and my friends in Germany. Thanks!

Last, but not least, I want to say many thanks to Adrian, Antonio, Christoph, Jia and Sid for their support and for the good times we had together during the work on our projects.

TABLE OF CONTENTS

DECLARATION XXVII

ABBREVIATIONS

Abbreviations used in this thesis in order of appearance:

UWA	University of Western Australia
CIIPS	Centre for Intelligent Information Processing Systems
DOF	Degree of Freedom
COM	Centre of Mass
COF	Centre of Force
NPCM	Normal Projection of Centre of Mass
ZMP	Zero Moment Point
RoBIOS	Robot Basic Input Output System
HDT	Hardware Description Table
API	Application Programming Interface
RCS	Reference Coordinate System
BCS	Body Coordinate System
PWM	Pulse Width Modulation
L##	Joint on Left Leg
R##	Joint on Right Leg
#H#	Joint in Hip
#K	Joint in Knee
#A#	Joint in Ankle
##B	Bending Joint (Forward-Backward)
##S	Sideway Joint (Left-Right)
SOAR	Safe Operational Area

Joint Notations:
The first letter distinguishes the leg, the second one names the position of the joint, the third refers to the moving direction.

TABLE OF FIGURES

INDEX OF TABLES

1. INTRODUCTION

In 1921, Czech playwright Karel Čapek (1890 - 1938) published his best-known work "R.U.R." ("Rossum's Universal Robot"). This play featured machines, which had been created to simulate human beings. From this time on, the word robot, which Čapek derived from the Czech word robota (which stands for: work, slave) became more and more common as a synonym for electrically controlled mechanical devices. The idea of machines having the ability to work for humans came true. From the middle of the last century, different types of robots have been developed. During the 1980s, assembly robots became very popular in industry. These were the predecessors for the next generation of robots. Technological progress in computer technologies not only made both hardware and software affordable, but also increased computational power allowing more complex algorithms in combination with sensors that provided robots with autonomy. A few years later, Čapek's fiction of human-like robots appeared to now be a vision for researchers.
'Intelligent' and behaviour driven machines seem to be a challenge that many researchers like to accept. Out of the variety of behaviours that arise from imitating a human being, one of the most basic skills of man seems to be one of the most complex at the same time: walking. As many other robots, walking machines are designated to do tasks that could or should not be done by humans because of dangerous environments, reliability aspects or simply cost-effectiveness. Nevertheless, the more interesting intention in this focus is probably the stimulating task of walking as an end in itself. Starting with non-human-like structures, such as four legged walkers or large feet, researchers attempted to deal with the problem of instability. Although stable walking machines have been constructed in this way, this did not satisfy the desire to create a robot that could give the impression of being human-like. However, at high budgets nothing is impossible as proven by Honda's *ASIMO* [1], a highly sophisticated biped with the ability to walk. However, the use of a high number of sensors and actuators entails expensiveness. Realisations at low costs in this field are still not achieved.

The objective of this project is to research walking with a bipedal robot, called *Andy Droid*. It is a low cost bipedal robot with few sensors and actuators.

2. RELATED RESEARCH

It was intended to develop essential features that provide functional basics for further work to establish a dynamic walk. Some projects with different robots have been carried out in the past and still are in progress.

2.1. BIPED PROJECTS AT CIIPS

At the Centre for Intelligent Information Processing Systems (CIIPS) at School of Electrical, Electronic and Computer Engineering at the University of Western Australia (UWA) several projects have been carried out on bipedal robots. Most of them aimed at walking, as this is a basic skill of bipeds.

The first biped developed at CIIPS was *Johnny Walker*. It has 9 degrees of freedom: each leg is bendable in its ankle, knee, and hip and it can be rotated in the hip. An additional servo is situated in the torso allowing the robot to bend sideways. All actuators are servos and the upper body consists of an EyeBot-Controller and a digital camera. The torso servo is used to displace the centre of mass to the left or right.

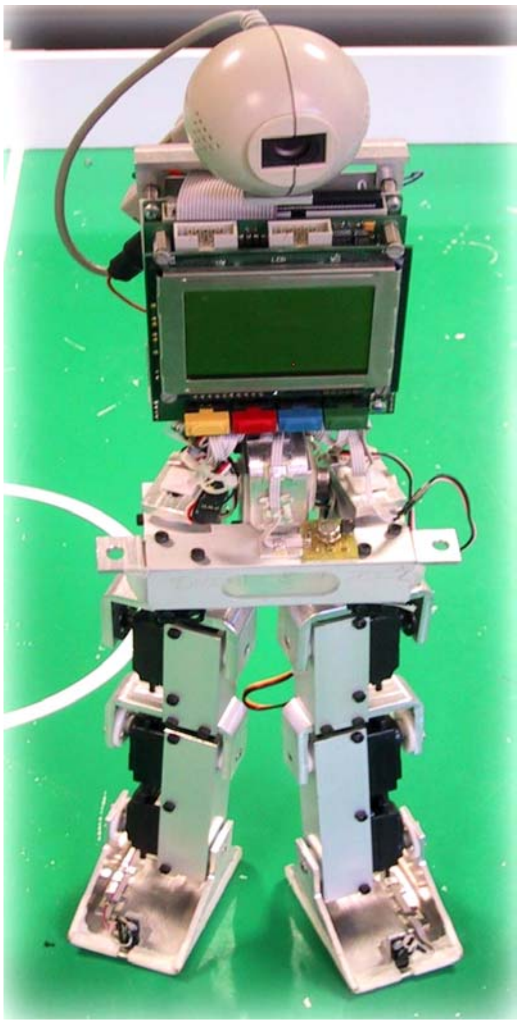


Figure 2.1: Johnny Walker

As an open loop controlled walking did not bring the desired success, a modification of the first biped was developed. The next model had arms, which could only be moved forwards and backward. This brings two more degrees of freedom and now the centre of mass could be shifted from the front to the back. The rest of the design of this new robot, named *Jack Daniels*, was similar to the former design. In addition, the camera had been removed to lower the centre of mass and stabilize the machine.

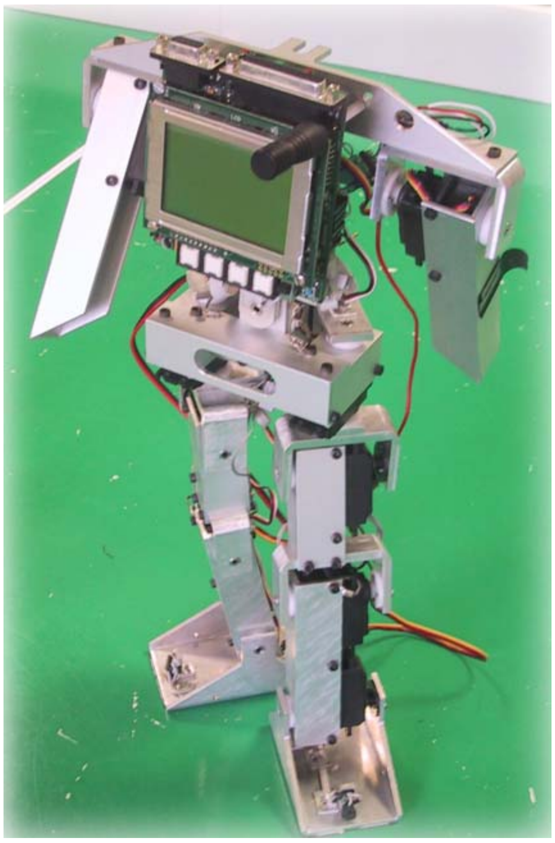


Figure 2.2: Jack Daniels

The next robot's design removed the torso's servo and the turning servos in the hip. The ankles obtained one more degree of freedom, allowing the robot to bend sidewards. In total, this robot, *Andy Droid*, was smaller than the first two designs and had 12 DOF. However, its arms had been attached to the robot for aesthetical reasons. During this project they were removed to save weight.

This thesis's research focused on the *Andy Droid* robot. Therefore, it is described in more detail in a later chapter.

An exceptionally different design is also currently being investigated. All robots until now were driven by servos. *Rock Steady* is driven by DC motors. Its mechanical structure is able to generate a gait in each leg driven by only one motor.

A third motor is mounted behind the controller display to carry a weight from left to right and thereby relocate the centre of mass. Thus, Rock Steady has only 3 degrees of freedom in total. The structure is made of light plastic, which makes this robot especially light.

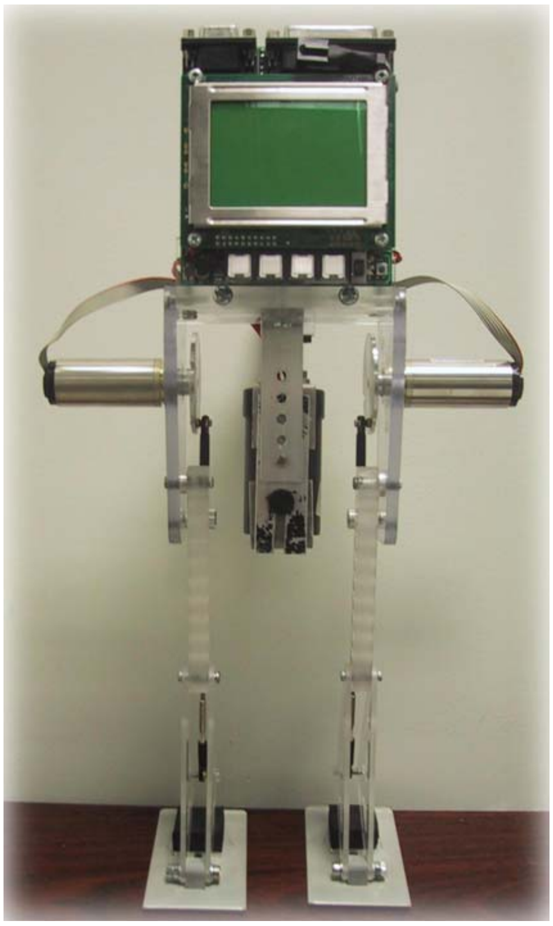


Figure 2.3: Rock Steady

2.2. Commercial Projects

As mentioned in the introduction, one of the most sophisticated walking bipeds is ASIMO, developed by Honda Motor Company. As the internet page [1] says, the 1.2m tall *ASIMO* has 24 DOF, 5 in one arm, 1 in each hand and 6 per leg. The 6 DOF of one leg are a combination of the following joints: ankle front-back, ankle left-right, knee front-back, hip front-back, hip left-right, and hip rotate. The joints are driven by servos. ASIMO weights 43 kg and is controlled by an onboard controlling unit. It carries an autonomous power supply in its backpack. Each foot has a six-axis foot-area sensor and the torso contains gyroscopes and acceleration sensors.

The robot is able to walk, it has a wide operating angle with his arms and hands and can imitate several human behaviours to interact with humans directly. Moreover, it also has a Japanese speech recognition system onboard.

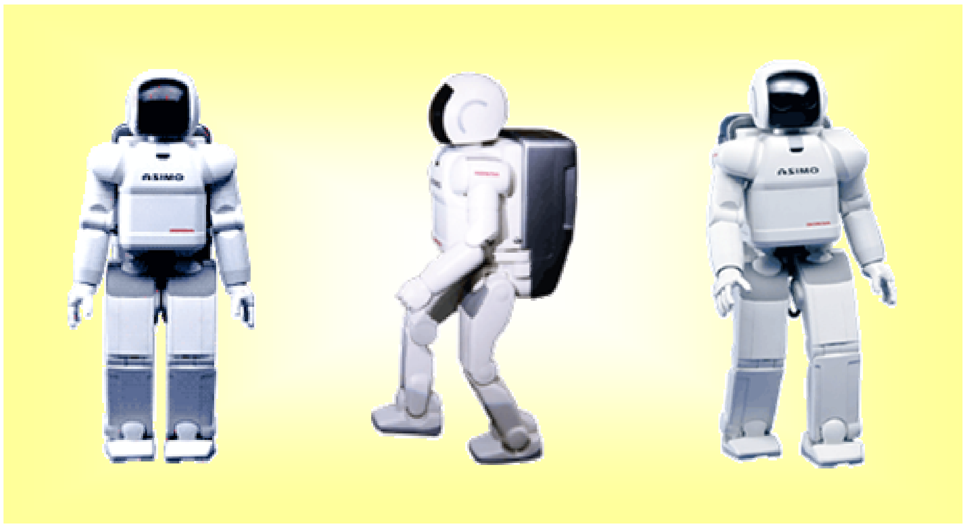


Figure 2.4: ASIMO

A different design is the *Shadow Robot*, developed by Shadow Robot Company Ltd. Its skeleton is made of wood to provide flexibility and its actuators are so-called air-muscles. In short, it is a flexible tube, which shortens when it is filled with air, similar to a balloon. The structure is closely orientated to the human skeleton, e.g. connecting the air-muscles to the joints by strings, which act as ligaments. The force of an air-muscle is detected by strain gauges at one end of the muscle. Under each foot, five pressure sensors detect the distribution of the robots centre of mass and a potentiometer determines the angle of each joint.

Additional sensory data is gathered from mercury tilt sensors to provide information on the balancing state of the walker.

This robot is only able to walk two steps.

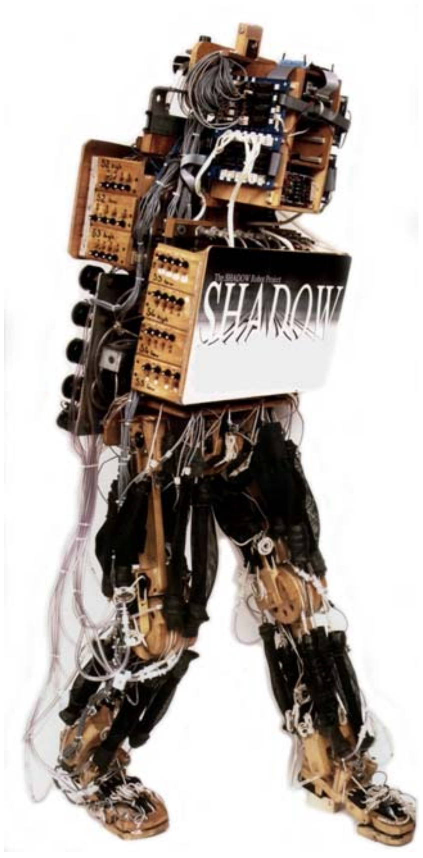


Figure 2.5: Shadow Robot

3. Balancing and Walking

3.1. Definitions

This chapter defines some fundamental terms and notations needed for the discussion of bipedal walking and balancing.

Orientation Angles:
The global orientation of a vehicle, aircraft, vessel, or walking machine can be expressed in three angles called pitch, roll, and yaw. These orientations are commonly used in aviation and navigation but their clearness makes them feasible for walkers also. Figure 3.1 shows their definition.

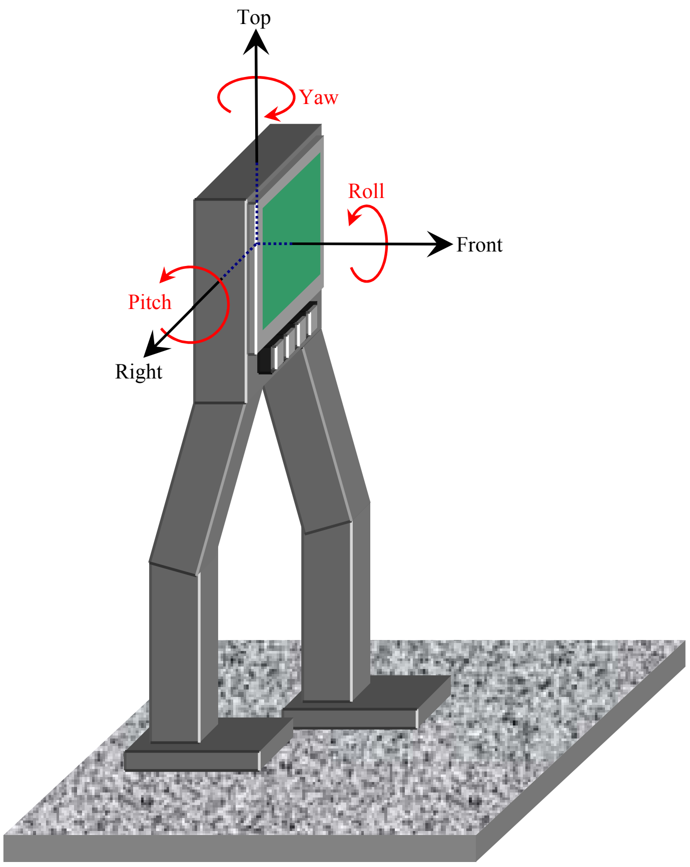


Figure 3.1: Pitch, Roll, and Yaw

Later in this thesis, pitch will mean a deflection of the centre of mass to the front or to the back. In this case, the body's bottom-up axis is no longer vertical but tilted. On the other hand, a roll-angle means that the body's axis is tilted either to the left or to the right. Yaw is not needed, as it describes the global moving direction, as does a compass. This angle is irrelevant for stability considerations; it is only needed for global navigation.

Supported Area:

The supported area or supported polygon is the area surrounded by the corners of the feet. This area is elementary for stability considerations. The following figure illustrates the definition.

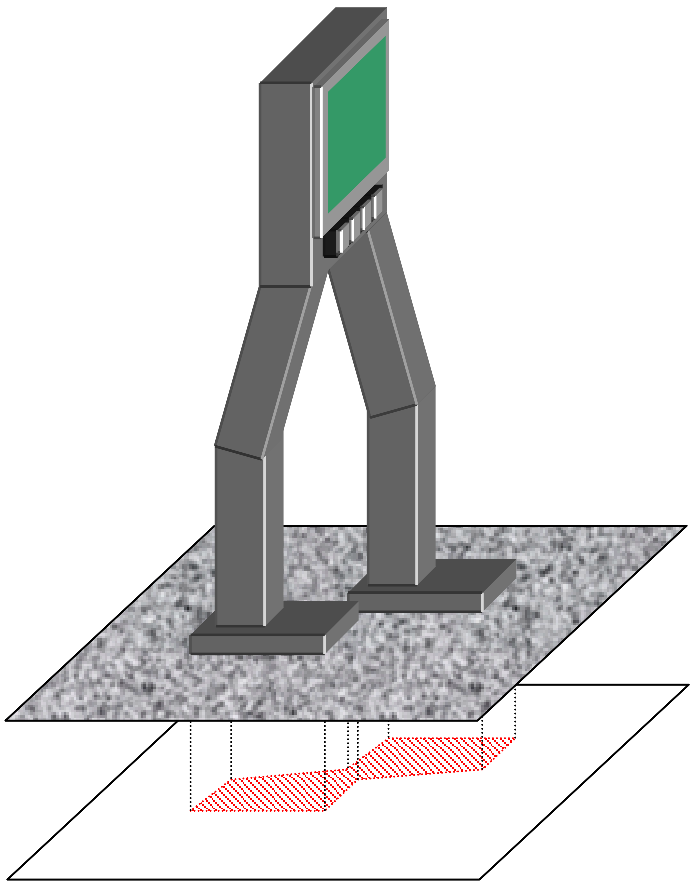


Figure 3.2: Supported Area

Centre of Mass (COM):

The Centre of Mass or Centre of Gravity is the defined by the gravity of a body. The total gravity is equal to sum of the gravities of its mass-elements. The total gravity results in the Centre of Mass.

Mathematically expressed, this means: $\vec{r}_{COM} = \frac{\sum_i \vec{r}_i \cdot m_i}{\sum_i m_i}$.

Centre of Force (COF):

Here, the COF is defined for the measured forces acting on one foot. The mean value of all single forces on a foot can be represented by one resulting force. This force is evoked by the gravity on the foot. The mathematical definition is similar to the centre of mass: $\vec{r}_{COF} = \frac{\sum_i \vec{r}_i \cdot F_i}{\sum_i F_i}$ but this point is always situated in the plane area between the foot and the subsurface.

Normal Projection of the Centre of Mass (NPCM):

The NPCM is the point on the subsurface where the COM is perpendicularly projected on.

Zero Moment Point (ZMP):

According to Ian J. Marshall's thesis [14], "the Zero Moment Point (ZMP) is the point on the ground surface about which the sum of all the moments of active forces is equal to zero." It is identical to the NPCM as long as the robot is standing still.

3.2. Balancing

The walking machines presented in the last chapter have many different features. Nevertheless, all of them have one thing in common. They have been developed to perform primarily one task: walking. All of them use feet that are in a proportion to their height similar to human beings. There have been many, relatively simple designed robots that were able to walk without any sensors and high-technology controlling units at all. However, their design is very different to that of humans. Some of them had more than two legs; others had feet supporting a very large area. Both features provided stability in standing and walking but these robots did not look like humans. Before starting to walk, a biped should be able to stand stable. This ability is called *balancing*. This word is related to balance, which is defined by the Encyclopaedia Britannica [13] as follows:

"Instrument for comparing the weights of two bodies, usually for scientific purposes, to determine the difference in mass. "

The task, balancing, which has to be fulfilled by a robot, is compensating the effects on its stability, from the forces acting on its body parts. The result would be a stable stand. For the detection of the static situation, pressure or tilt sensors under the feet are suitable as well as gyroscopes. For dynamics, acceleration sensors would be needed.

3.3. WALKING

Once having gained stability, the robot can take its first steps. First, it has to be decided which kind of walking should be performed as there are generally three different ways: static and dynamic walking, and running.

Static Walking:
As long as the normal projection of the centre of mass (NPCM) is inside the supported area, the robot is always stable. This means, that it could rest in this position at its actual joint angles, without toppling down. It is significant for this type of walking that the gaits look "mechanical", because the body has to be shifted from the left to right over relatively wide ranges to move the centre of mass above the supported area of the foot that is attached to the ground. To make sure, that the NPCM is always within the mentioned area, the dynamic influences must reduced. This slows down the forward velocity of the walker. Static walking is characterised by different phases. One half-cycle is sequenced in the "double support phase" when both feet are resting on the ground surface, the "swing phase" when one foot is lifted and swung to the front. This phase is followed by the next double support phase.

Dynamic Walking:
The elementary value to control dynamic walking is the Zero Moment Point. The NPCM is allowed to be outside the supported area as long as the statement above is fulfilled. Freezing a robot during this walk would bring the walker to an unstable condition. During dynamic walking, the ZMP must be inside the boundaries of the supported area. Otherwise, it is dynamically unstable. The walking phases are the same as in a static walk.

Running:

Running is a kind of dynamic walking at a high velocity. The underlying mathematical requirements are the same as those for dynamic walking. Yet, the phases are different: One cycle comprises a single support phase during which the opposite foot swings to the front. During the following phase both feet are lifted and the whole walker is in a ballistic movement which is absorbed by the next single support phase.

4. Provided Hard- and Software

4.1. EyeBot-Controller

The EyeBot is a controller designed for small, mobile, and autonomous robots. Implementations in wheeled, legged, and flying robots already exist. The intention for autonomous robots is obvious after a closer look at the technical features: A 32-bit microcontroller (Motorola 68332) runs at 25MHz to 33MHz. 2MB RAM and 512KB EEPROM provide memory for system and user programs. On board, there are two motor drivers, one parallel, and two serial ports. Each eight digital in- and outputs and eight analog inputs can be used for sensor or actuator control. Furthermore, 16 timing processor I/Os can be used, e.g. to drive servos. Everything is mounted on a double-layered printed circuit board, where a 128x64 pixel LCD and 4 buttons act as user interface. It has also has an interface for either a colour or a greyscale camera with onboard image processing. A speaker and a microphone provide audio features. In addition, an IR receiver can be connected and via a radio module, communication to other controllers or a PC can be established. Average power consumption of the controller is 235mA at 7.2V. The specifications presented in this chapter correspond with [2].

The picture on the right shows the controller board in front view. The LCD shows RoBIOS information after system start-up.

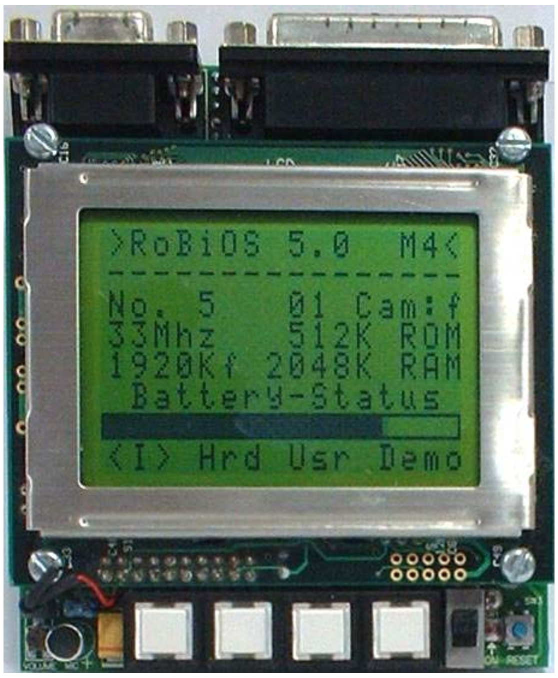


Figure 4.1: EyeBot Controller

4.2. ROBOT HARDWARE-SETUP

In addition to the controller itself, the robot in total consists of 10 servos that form two legs. The servos are connected by aluminium links. The feet are formed by three metal strips each, which are stellately attached to the lower central end of each leg. The servos are powered separately by an additional nickel-cadmium battery, which is carried between the controller's lithium-ion battery and the EyeBot controller on top of the robot, all borne on an aluminium construction, which is mounted on top of the servos, which make up the hip. Figure 2 shows the robot:

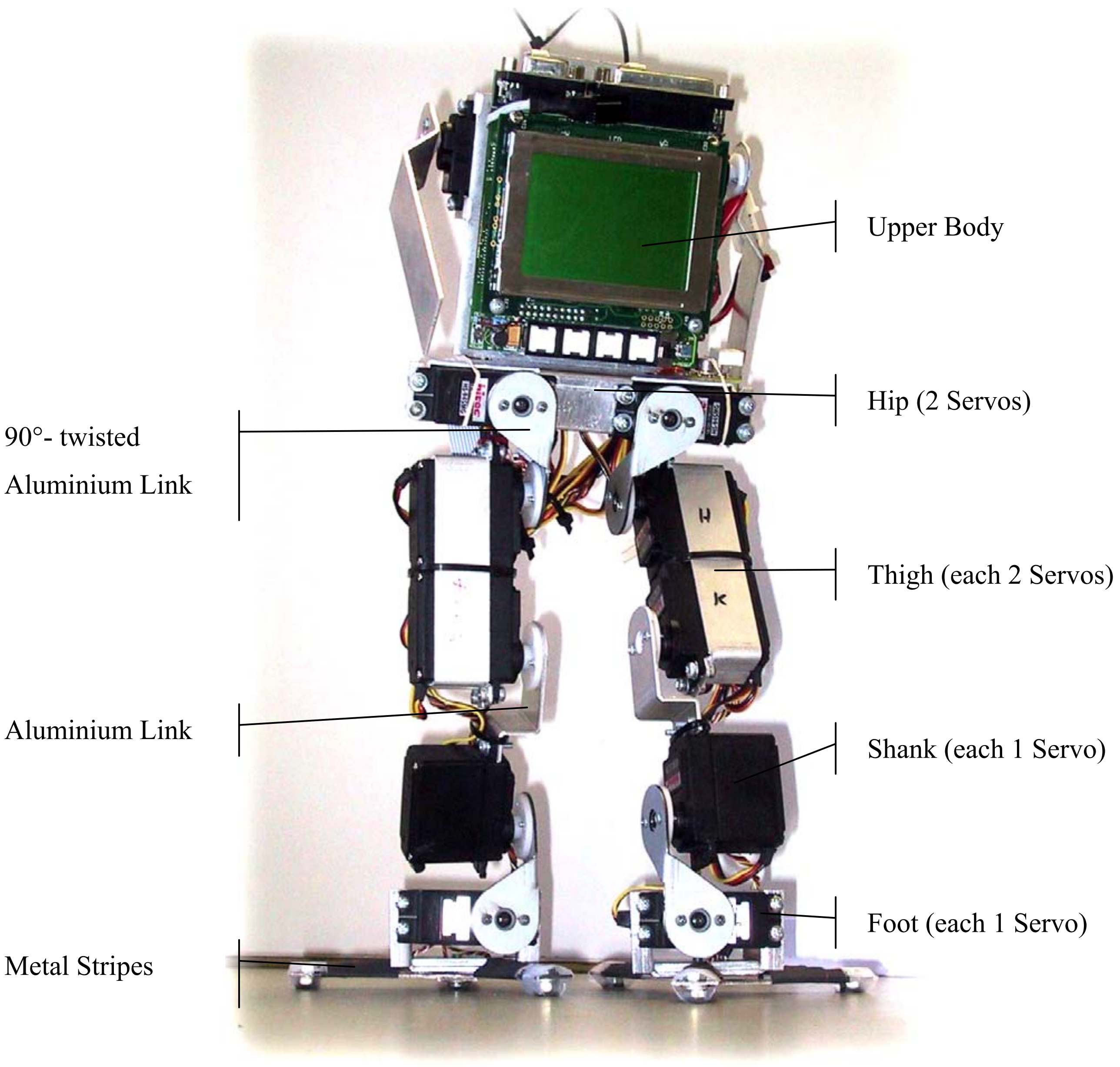


Figure 4.2: Andy Droid (front view)

The total height of the robot is approximately 350mm, when both legs are straightened. The width at the hip is 130mm while the outer tips of the feet spread over 210mm when the legs are stretched out. From front to back, it extends 85mm at the top and at 130mm at each foot. The total weight (including both battery packs) is 1440g.

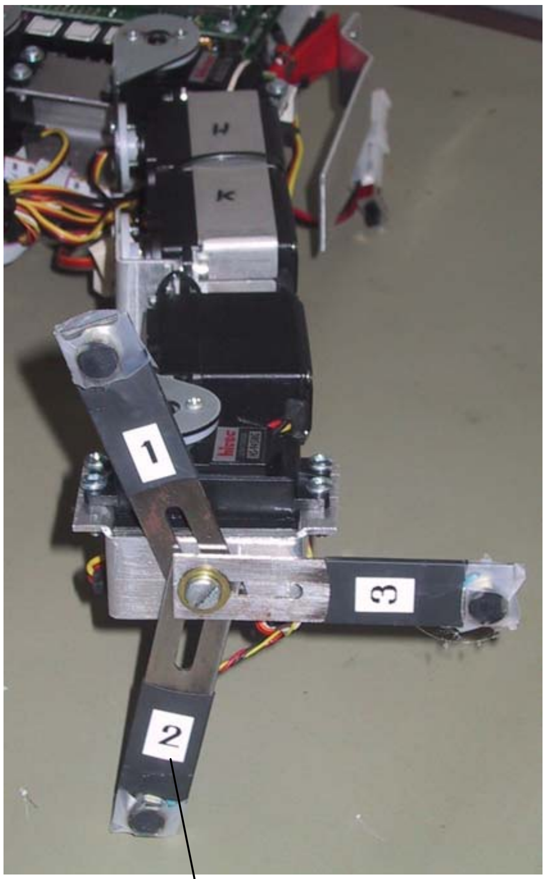


Figure 4.3: Andy Droid's left foot

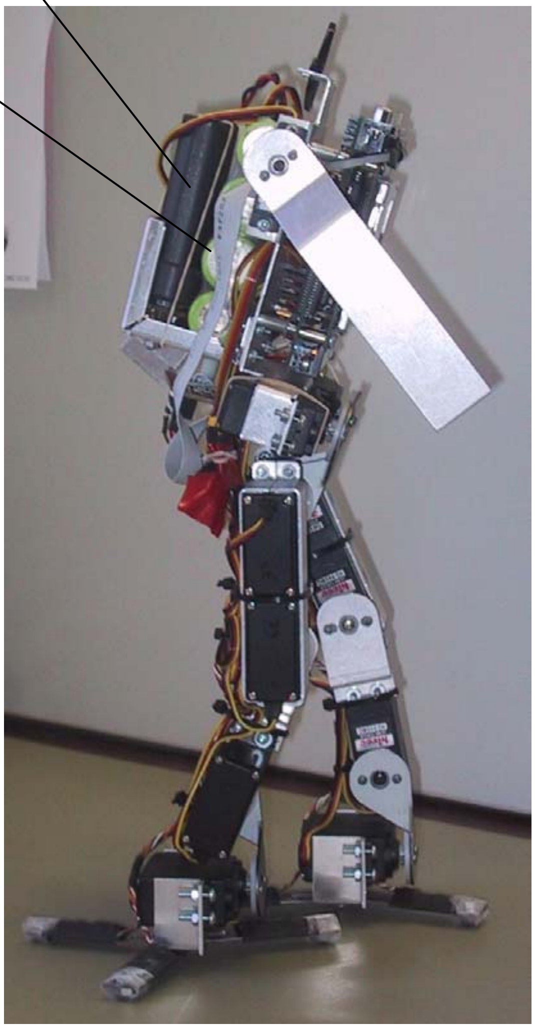


Figure 4.4: Andy Droid (side view)

Each of the three metal strips for each foot, henceforth referred to as toes, has a strain gauge on its top, connected in a bridge. Thereby, the deflection of a toe is expressed as a voltage, which is A/D-convertible by the analog inputs.

The ten servos used for the legs are all of the type *HS-945MG* from the American manufacturer *HITEC*. A servo is driven by a square wave pulse and according to its specifications [4]: the nominal range of the pulse is from 900µs to 2100µs. The refresh frequency is 50Hz. The length of the pulse defines the angle of the servo. The centre position is at a pulse length of 1500µs. The operating angle is given by 67.5° per 600µs. However, the range has been extended to ±72° that means a range from 860µs to 2140µs. The reasons will be explained in Chapter 4.5. The dead bandwidth is said to be 4µs, which results in a minimal angular resolution of 0.45°. The servos do not have a feedback of the actual position.

The strain gauges are connected in 6 Wheatstone quarter-bridges with temperature compensation as follows:

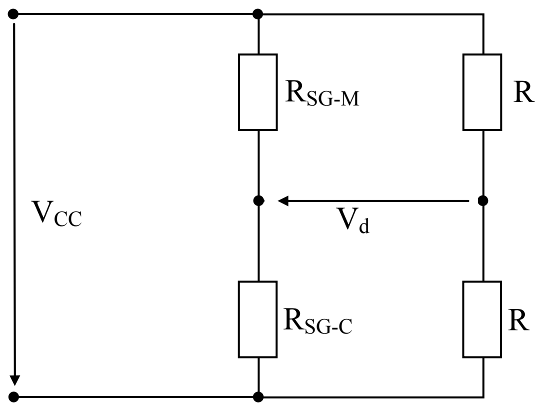


Figure 4.5: Gauging Circuit

R_{SG-M} here is the strain gauge that is stretched, when a spring metal stripe is bent. R_{SG-C} is not glued to and thus not influenced by the metal but just by the temperature. In that way, the total influence of temperature to the measured voltage V_d is eliminated. In addition to this, the bridge voltage is amplified with an AMP04 on a small PCB mounted directly on the toe. The amplification is $\frac{V_{out}}{V_d} \approx 200$. The circuit has a low pass filter with a cut-off frequency of 5.3 kHz. Finally, the voltage has a zero offset of about +1V and a range of approximately 200mV. The data is transformed with a 6-channel multiplexed A/D converter.

4.3. RoBIOS and Application Programming Interface

The EyeBot's basic input output system, called RoBIOS, provides an operating system that makes all hardware accessible and controls the EyeBot. It includes an Application Programming Interface (API) in the programming language C. This API comprises functions to access different features and hardware:

- Key Input
- IR Remote Control
- LCD Output
- Servos
- Analog-Digital Converter
- Timer
- Multi Tasking
- Semaphores
- System Functions
- Download and RS232

To complete the list of available function-groups, the following are mentioned but not used in the developed software, as the robot does not use the according hardware:

- Motors
- Image Processing
- Position Sensitive Detector
- Camera
- Audio
- V-Ω Driving Interface
- Bumper and Infrared Sensors
- Latches
- Parallel Port
- Radio Communication
- Compass

A complete list of all routines can be found on the EyeBot-Homepage.

To adapt a different hardware setup to the controller, the so-called Hardware Description Table (HDT) declares, which hardware is connected to the controller and in which way it has to be accessed. E.g., if a robot has a DC motor and a servo, these have to be named in the HDT. Furthermore, the output port for the PWM-driven motor has to be assigned as well as the input port of the encoder. A servo can here be adjusted by setting its pulse width and position. In modelling, where many servos are used, this is done by analog trimmers on the remote control.

Since there are no analog trimmers and due to the way the servos are driven, the entries in the HDT are as follows:

```
servo_type _NAME = {VER, CH, TIMER#, PERIOD, START, STOP};
```

The struct `servo_type` gets an instance *_NAME*. *VER* just names the drivers version, *CH* defines the TPU channel to be used for the output signal, *TIMER#* assigns a timer that repeatedly (*PERIOD*) generates the edges of the needed square impulse that lasts from *START* to *STOP*. The final entry of this servo is done with the code shown below. Here `HDT_entry_type` is again a struct with an array of instances named *HDT*. This assigns the previously defined `servo_type` *_NAME* to its `SemanticName` as a *SERVO*.

```
HDT_entry_type HDT[ ] =
{
…,
{SERVO,SemanticName,"TestNameString",(void *)&_NAME},
…,
}
```

4.4. Software Development

User programs can be written in C/C++ or Assembler. The source code then is compiled with a cross compiler, which is derived from the GNU GCC C/C++ compiler. After linking and compressing the produced so-called hex-files, these can be downloaded to the controller via serial connection.
It should be mentioned that the existing compiler did not work under Windows XP. Therefore, Antonio Pickel recompiled the cross compiler under Windows XP using the Linux emulator CygWin.
As the EyeBot is an embedded system, it does not provide the advantages of an integrated development environment. Although a debugging mode exists on the controller PCB, the background debugger software via parallel port did not work. Software development in this way is much more time consuming because error searching during runtime was reduced to status printings on the display.

4.5. CHANGES IN HDT

As explained in chapter 4.2., the servos are controlled with a square pulse signal. The range of this signal, from *START* to *STOP*, is divided in 256 equidistant steps by the RoBIOS' servo-driver. The minimum servo angle could be reached by calling the user-function `SERVOSet`[1)] with '0' as parameter. '255' drives the servo to its maximum angle. Before the changes as described in this section, the servo settings in the HDT were chosen to limit the physical angle of the servo with *START* and *STOP*. Thereby, the highest angular resolution was expected.

For example: a servo has a safe operating range of ±15°, the *START* and *STOP* should have been set to $1366.\overline{6}\mu s$ and $1633.\overline{3}\mu s$, which is, according to chapter 4.2, $START\,/\,STOP = 1500\mu s \mp \frac{15°}{67.5°} \cdot 600\mu s$. This resulted in uncertainties:

1. *START* and *STOP* are of integer type and thence could only be nearest value, here 1367µs and 1633µs. RoBIOS itself calculates the actual angle using the following algorithm:

$$range = STOP - START$$
$$angle = range \cdot desired\_angle$$
$$hightime = angle\,/\,256$$
$$hightime = hightime + START$$

2. Line 3 shows the linear interpolation. The *angle* is divided by 256. This again is an integer division. *Angle* is the product of *range* and *desired_angle*, where the latter is an integer value between 0 and 255. Therefore, just in the case if *range* is dividable by 256 without remainder, the resulting *hightime* has no error. This means in the example: $range = 1633 - 1367 = 266$ and accordingly $hightime = 266\,/\,256 = 1$ instead of approximately 1.04.

1) see App. A – Servos and Motors

3. Keeping *range* small should increase the resolution compared to bigger ranges. In the given example, the minimum step length in the servos signal is consequently $266\mu s / 256 \approx 1.04\mu s$. Nevertheless, the servo's specification mentions a dead bandwidth of 4µs, which means that the servo would not move until the signal length is not increased or decreased by this time.

4. Giving one servo a smaller range than another one also brings another problem. Once having routines to drive a servo at a desire speed, unequal ranges result in different angular velocity. This is because an increase of a servo's "software" position of for example 100 (out of 255) in a special interval, forces the one with the smaller range to drive a small angle, whereas the other one drives a greater angle in the same time. In addition, a "software" angle of e.g. is 50 results in a different physical angle for servos with different ranges.

Solution: The problem concerning the first issue is obvious: only use integer values. The next one is solved by using *START* and *STOP* values with a difference, which is dividable by 256 without remainder. The third solution is to make sure, that the previously stated range exceeds 4 times 256 so that a change in the value in the software results in a change of the servo's physical position. The solution to the last problem simply requires the same range setting for all servos.

The following term expresses the valid terms that will fulfil all four requirements: $\mathrm{range} = \mathrm{n} \cdot 256 \left| \mathrm{n} \in \mathbf{N}, \mathrm{n} \geq 4 \right\} \forall \, \mathrm{Servos}$.

The statements above were done by using n=5, to ensure the servos dead bandwidth of 4µs is exceeded. This results in a range of 1280µs, which means centre ± 640µs. Thus the range is ±72°. Uncertainties in the servos' centres positions can be adjusted by experimentally correcting these in the HDT without violating the requirements above.

The normal rotational direction of the servos is clockwise. RoBIOS allows the inversion of the direction by simply exchanging the *START* and *STOP* value. The directions have been chosen in the following way:

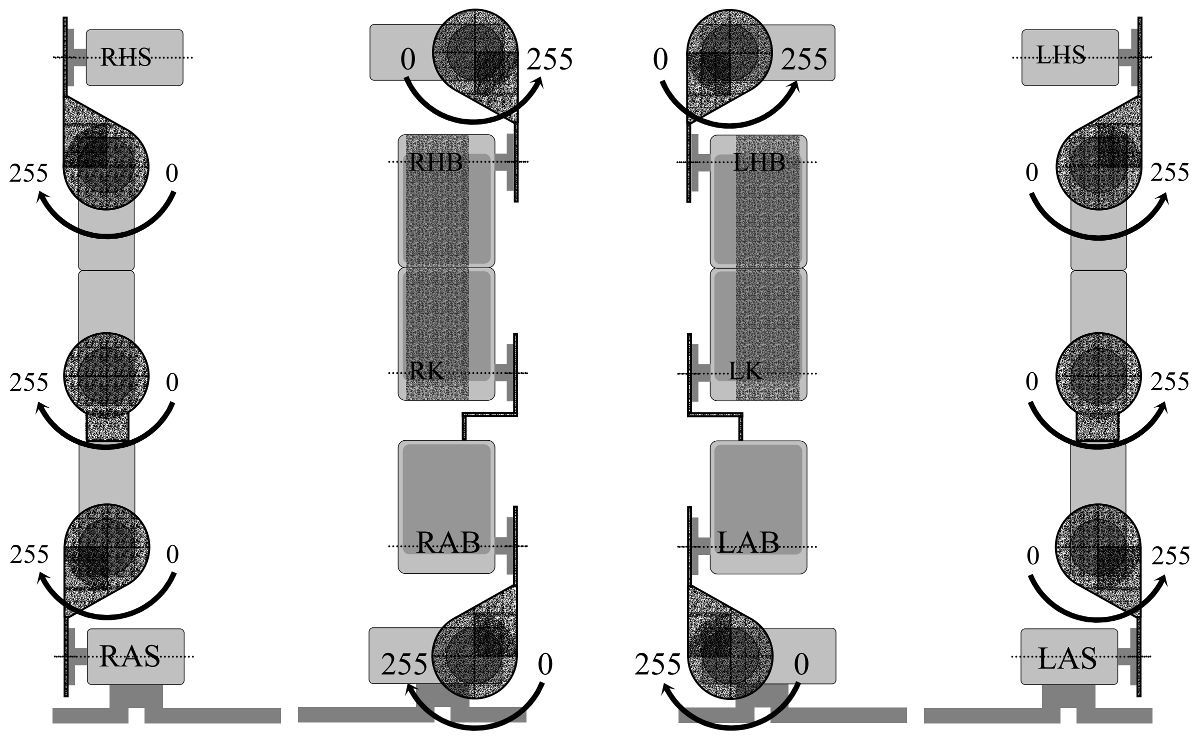


Figure 5.5: Rotational Directions

The letters refer to the joint's position. L and R stand for left and right leg, B stands for bend, whereas S stands for sideways. H means hip, K is knee, and A is ankle. Hip and ankle joint sideways turning directions are chosen in a way that a parallel movement around the same angle of all four R/LHS and R/LAS produces a parallel sideway swing. Bending joints are directed to bend a leg forward while increasing the angles. The directions have also been adapted to fit the demands in section 5.3.

The adaptation of the strain gauges to the controller does not need to be entered into the HDT, because the 8-channel A/D converter is mounted on the controller board directly and hence is always included in the RoBIOS. Filtering and adapting the sensory data has to be done in a user program.

5. Adapted Denavit Hartenberg Notation

5.1. Mathematical Basics and Conventions

For better understanding, it is necessary to give some short explanations on the notations and mathematical basics used in this chapter.

To characterise a body with non-exiguous dimensions in space mathematically, its position, extensions, and orientation are sufficient. Usually, Cartesian coordinate-systems are used in which a particular point is described with a vector:

$$\vec{p} = \begin{pmatrix} p_x \\ p_y \\ p_z \end{pmatrix} = \begin{pmatrix} p_x & p_y & p_z \end{pmatrix}^T = {}^{X}\vec{p}\,. \qquad (5.1)$$

Using multiple coordinate systems requires a method to describe a vector's affiliation. Therefore the ante-supra index names the coordinate system a vector is described in, as shown in (*5.1*), where *X* is the coordinate system. This coordinate system is further referred to as the *reference coordinate system* or *RCS*. To specify the orientation of a body, another coordinate system has to be fixedly assigned to the body. This one is called the *body coordinate system* or *BCS*. Having both of these, the position can be expressed as a translation of the origin of the body coordinate system in the reference coordinate system. The orientation is a rotation of the BCS compared to the RCS.

Rotations are calculated with matrices: ${}^{RCS}\vec{p}' = {}^{RCS}R \cdot {}^{RCS}\vec{p}_{0,BCS}$. In terms of coordinate transformation, a vector in a BCS can be expressed as a vector in the body's RCS as the result of the rotation in between the both systems: ${}^{RCS}\vec{p} = {}^{RCS}_{BCS}R \cdot {}^{BCS}\vec{p}$. As seen, the ante-sub index of the rotation matrix shows the source system to which the vector on the right must belong. The ante-supra index of *R* names the destination system in which the resulting vector is expressed. Translations are trivial. Again, in terms of transformations, a translation is the directed distance of the BCS' origin, seen from the RCS': ${}^{RCS}\vec{p} = {}^{RCS}_{BCS}\vec{t} + {}^{BCS}\vec{p}$. Here, the ante-sub index says that this translation transforms from the thereby referred system to the one named in the ante-supra index.

Now, a characterisation of a body is complete with a set of a rotation matrix and a translation vector, both relative to one reference coordinate system. This set is called *frame* $F = \begin{pmatrix} {}^X_U R & {}^X_U \vec{p}_0 \end{pmatrix}$ (*5.2*).

As seen before, a translation is mathematically an addition of two vectors, whereas a rotation is a multiplication of a vector and a matrix. To combine both and make the calculation more manageable, the frame can be extended by so-called homogeneous coordinates that simply extends the frame by the vector $\vec{h} = \{0 \quad 0 \quad 0 \quad 1\}$. The frame becomes a square matrix:

$$ {}^X_U F = \begin{pmatrix} {}^X_U R & {}^X_U \vec{p}_0 \\ \vec{0}^T & 1 \end{pmatrix} \qquad (5.4.)$$

The upper left 3x3 elements contain the rotation matrix, the upper three elements of the right column are the translation. In the case of no translation at all, theses three are zero: $T_{rot}(\alpha) = \begin{pmatrix} R(\alpha) & \vec{0} \\ \vec{0}^T & 1 \end{pmatrix}$ and if there is only translation, the rotational square becomes a 3x3 identity matrix: $T_{trn}(p) = \begin{pmatrix} I & \vec{p} \\ \vec{0}^T & 1 \end{pmatrix}$.

So, if ${}^X_U F$ means a transformation from *U* to *F*, then concatenated transformation is a sequential multiplication of system-to-system transformations:

$$ {}^A_D F = {}^A_B F \cdot {}^B_C F \cdot {}^C_D F \, . \qquad (5.5)$$

This tool makes the calculation of multi-body systems feasible.

5.2. Denavit Hartenberg Notation

Anthropomorphic robots generally are made up of links that are connected by each one joint to their preceding and subsequent links. The origin herein is a base coordinate system. Joints could be of rotational or translational character. Each joint has one degree of freedom (DOF). Therefore, a concatenation of n+1 links by n joints means in total n DOF. *Andy Droid* has 5 servos in each leg and as a servo represents a rotational joint, *Andy* in total has 10 DOF.

Denavit and Hartenberg first introduced the quadruple $\{a_i, \alpha_i, d_i, \theta_i\}$ to describe robot kinematics. As Dieter Kraft explains in [5] (Chapter 1.1.2.), these kinematic parameters are divided into two groups: link parameters and joint parameters. The first group comprises *link length* and *link twist*, which are determined by the mechanical construction and are thus invariant. The latter group defines the *joint distance*, which is the drooping of a translational joint and the *joint angle*, which is the deflection of a rotational joint.

Link Parameters:

Link i is the interconnection of the joints g_i and g_{i+1}. The shortest distance of both skew joint axes is the distance between both root points of their common normal. It is called *link length* a_i.

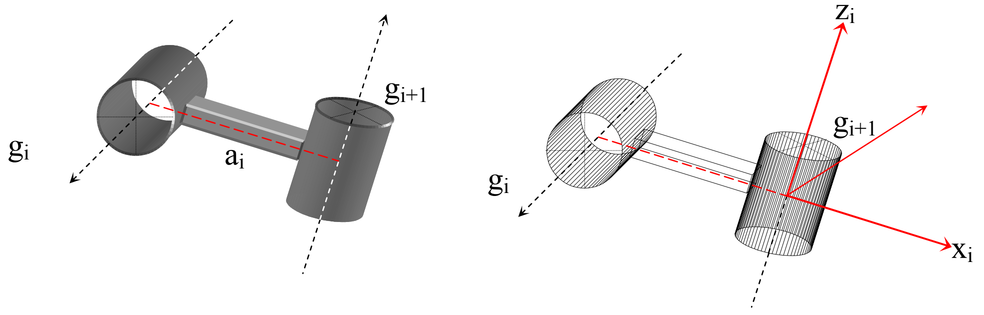


Figure 5.1: Link Length

The origin of link i's coordinate system $X_i = \{\, x_i \quad y_i \quad z_i \,\}$ is congruent with the root point of the common normal in the axis of g_{i+1}. Likewise, link i-1's coordinate origin $X_{i-1} = \{\, x_{i-1} \quad y_{i-1} \quad z_{i-1} \,\}$ lies in the foot point of the common normal at the g_i side.

The angle covered by the coordinate axes z_i and z_{i-1}, when shifting X_i into X_{i-1} along a_i, is called *link twist* α_i.

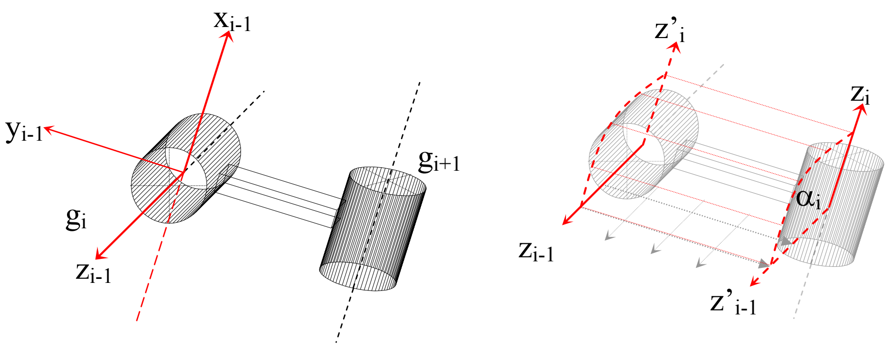


Figure 5.2: Link Twist

Joint Parameters:

These parameters refer to possible movements and are variable. The distance between the origin of the coordinate system X_{i-1} and the root point of the common normal on g_i is referred to by *joint distance* d_i. This is the degree of freedom of a translational joint.

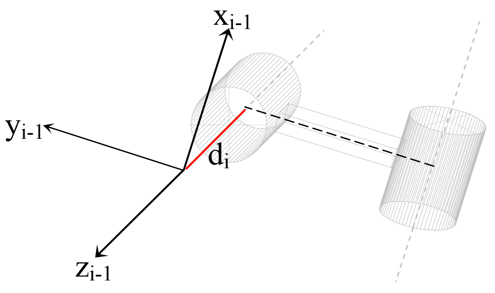


Figure 5.3: Joint Distance

Of more importance in the case of *Andy* is the so-called *joint angle* θ_i, which represents the variable of rotational joints. The rotation of link i in joint g_i rolls axis x_{i-1} into x_i by moving it by θ_i.

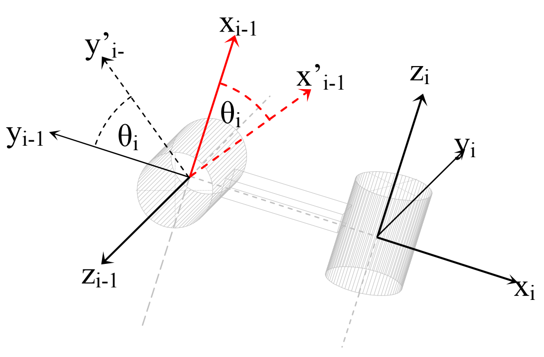


Figure 5.4. Joint Angle

According to Denavit Hartenberg, all links have to be labelled from zero to *n*, beginning in the base body. The concatenated link chain comprises *n*+1 links, connected by *n* joints and establishes a set of 4*n* parameters, out of which *n* parameters are variable, stating, that no joints are translational and rotational joints at the same time. Joint *i* connects link *i*-1 with link *i*. Dieter Kraft proposes the following algorithm:

Step	**Operation**
1. Labelling	Labelling the joints from 1 to *n*.
2. Base	Define base coordinate system $X_0 = \{x_0 \quad y_0 \quad z_0\}$ in the base body, so that moving axis 1 and coordinate axis z_0 are collinear.
3. Joint Coordinate Systems $\forall\ (1 \le i \le n\text{-}1)$	Align z_i-axis[2)] in moving direction of joint $i+1$.
	Choose origin of coordinate system X_i in - intersection of z_i- and z_{i-1}-axis or - intersection of common normal ($z_{i-1} \rightarrow z_i$) and z_i-axis.
	Determine x_i-axis either - ortho normal to both *z*-axes[2)] $x_i = \pm \frac{(z_{i-1} \times z_i)}{\lVert z_{i-1} \times z_i \rVert}$ or - along common normal, if both *z*-axes are parallel.
	Complete right-handed coordinate system with $\boldsymbol{y_i}$-axis $y_i = \frac{(z_i \times x_i)}{\lVert z_i \times x_i \rVert}$.
4. Link Parameter $\forall\ (1 \le i \le n)$	Link length a_i is the distance between the intersection of z_{i-1}-axis with x_i-axis and the origin of the coordinate system X_i, along x_i-axis.
	Link Twist α_i is the angle, around x_i-axis, that turns z_{i-1}-axis into z_i-axis.
5. Joint Parameter $\forall\ (1 \le i \le n)$	Joint distance d_i is the distance between the origin of X_{i-1} and the intersection of its z_{i-1}-axis and x_i-axis, along z_{i-1}-axis.
	Joint angle θ_i is the angle around z_{i-1}-axis that turns x_{i-1}-axis into x_i-axis.

Table 5.1: Denavit Hartenberg Algorithm

The following section shows the application of this algorithm to *Andy*.

[2] Determination is not one-to-one.

5.3. Coordinate Systems

Applying the Denavit Hartenberg Algorithm means, every joint has to be provided with a coordinate system. Figure 5.5 shows the left leg:

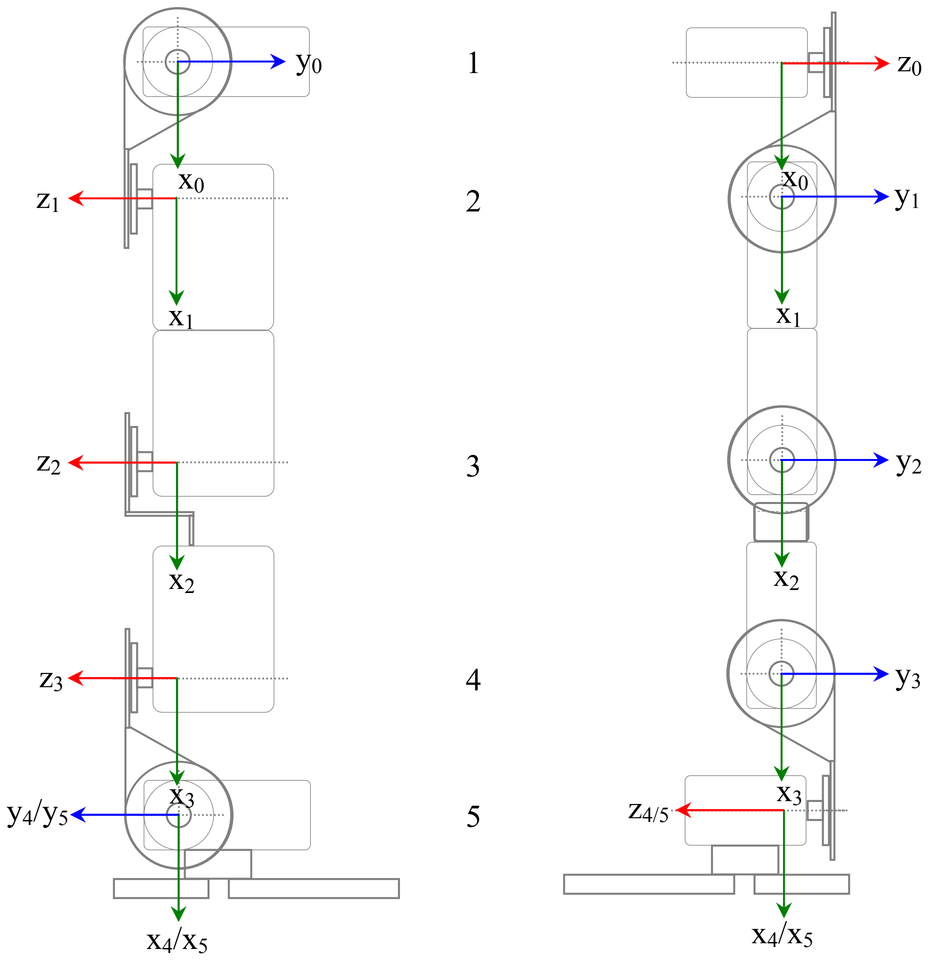


Figure 5.5: Coordinate Systems Left Leg (left: front view, right: side view)

i	Link	Joint	a_i/[mm]	α_i/[°]	d_i/[mm]	θ_i/[°]
0	Hip	none	0	0	0	0
1	90° Joint HB-HS	Hip-Side	35	90	0	HS
2	Thigh	Hip-Bend	60	0	0	HB
3	Shank	Knee	65	0	0	KN
4	90° Joint AB-AS	Ankle-Bend	35	90	0	AB
5	Foot	Ankle-Side	0	0	0	AS

Table 5.2: DV-Parameters Left Leg

All z-axes are pointed in the direction, so that they perform a right-screw in mathematical term. This provides the possibility to use the servos' angles as shown in Chapter 4 Section 5.

The right leg's coordinate systems and parameters:

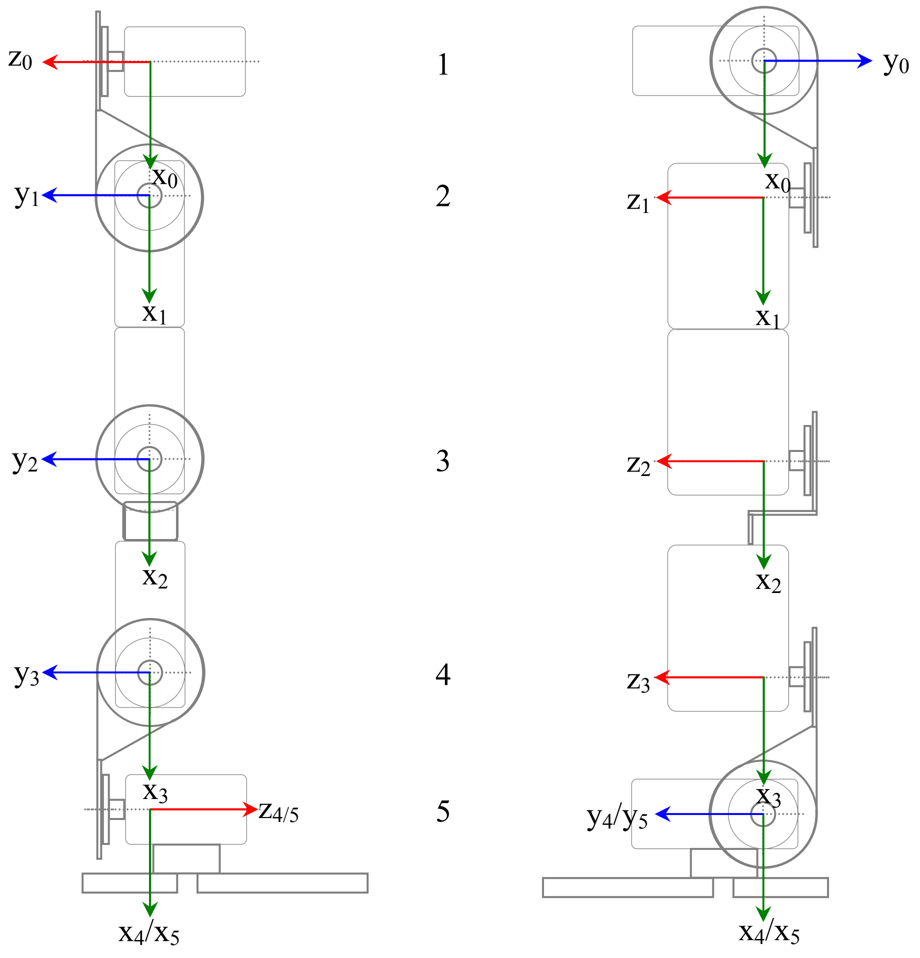


Figure 5.6: Coordinate Systems Right Leg (left: side view, right: front view)

i	**Link**	**Joint**	***a*** a_i/[mm]	α_i/[°]	d_i/[mm]	θ_i/[°]
0	Hip	none	0	0	0	0
1	90° Joint HB-HS	Hip-Side	35	90	0	HS
2	Thigh	Hip-Bend	60	0	0	HB
3	Shank	Knee	65	0	0	KN
4	90° Joint AB-AS	Ankle-Bend	35	90	0	AB
5	Foot	Ankle-Side	0	0	0	AS

Table 5.3: DV-Parameters Right Leg

The last two coordinate systems of both legs have the same origin, but X_5 is moved by joint 5, whereas X_4 is attached to joint 4. The variable parameters are the joint angles. The positions shown in Figure 5.5 and Figure 5.6 mean that all angles are 0°. Seemingly, the results for both legs are the same, but the angles are generally different as the two legs are usually not in the same position.

5.4. Transformation Matrices

To express coordinate system X_i in X_{i-1} the previously determined parameters can be used to calculate the transformation with each two translations and rotations. According to [5], the following matrix transforms from X_i to X_{i-1}:

$$ {}^{i-1}_{i}T = rot(\theta_i)\cdot trn(d_i)\cdot rot(\alpha_i)\cdot trn(a_i) = \begin{pmatrix} \cos\theta_i & -\cos\alpha_i\cdot\sin\theta_i & \sin\alpha_i\cdot\sin\theta_i & a_i\cdot\cos\theta_i \\ \sin\theta_i & \cos\alpha_i\cdot\cos\theta_i & -\sin\alpha_i\cdot\cos\theta_i & a_i\cdot\sin\theta_i \\ 0 & \sin\alpha_i & \cos\alpha_i & d_i \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{5.6} $$

Using the results from 5.3. the link-to-link transformations are identical for both legs:

$$ {}^{0}_{1}T = \begin{pmatrix} \cos\theta_{HS} & 0 & \sin\theta_{HS} & 35\cdot\cos\theta_{HS} \\ \sin\theta_{HS} & 0 & -\cos\theta_{HS} & 35\cdot\sin\theta_{HS} \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{5.7} $$

$$ {}^{1}_{2}T = \begin{pmatrix} \cos\theta_{HB} & -\sin\theta_{HB} & 0 & 60\cdot\cos\theta_{HB} \\ \sin\theta_{HB} & \cos\theta_{HB} & 0 & 60\cdot\sin\theta_{HB} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{5.8} $$

$$ {}^{2}_{3}T = \begin{pmatrix} \cos\theta_{KN} & -\sin\theta_{KN} & 0 & 65\cdot\cos\theta_{KN} \\ \sin\theta_{KN} & \cos\theta_{KN} & 0 & 65\cdot\sin\theta_{KN} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{5.9} $$

$$ {}^{3}_{4}T = \begin{pmatrix} \cos\theta_{AB} & 0 & \sin\theta_{AB} & 35\cdot\cos\theta_{AB} \\ \sin\theta_{AB} & 0 & -\cos\theta_{AB} & 35\cdot\sin\theta_{AB} \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{5.10} $$

$$ {}^{4}_{5}T = \begin{pmatrix} \cos\theta_{AS} & -\sin\theta_{AS} & 0 & 0 \\ \sin\theta_{AS} & \cos\theta_{AS} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{5.11} $$

The transformation matrix from foot to hip is the concatenated transformation from link-to-link: ${}^{0}_{5}T = \prod_{i=1}^{5} {}^{i-1}_{i}T$. This results in:

$$
{}^{0}_{5}T_R = \begin{pmatrix}
\begin{array}{l} cAScABcKNcHBcHS \\ -cASsABsKNcHBcHS \\ -cAScABsKNsHBcHS \\ -cASsABcKNsHBcHS \\ +sASsHS \end{array} &
\begin{array}{l} sASsABsKNcHBcHS \\ -sAScABcKNcHBcHS \\ +sAScABsKNsHBcHS \\ +sASsABcKNsHBcHS \\ +cASsHS \end{array} &
\begin{array}{l} sABcKNcHBcHS \\ +cABsKNcHBcHS \\ -sABsKNsHBcHS \\ +cABcKNsHBcHS \end{array} &
\begin{array}{l} 35cABcKNcHBcHS \\ -35sABsKNcHBcHS \\ -35cABsKNsHBcHS \\ -35sABcKNsHBcHS \\ +65cKNcHBcHS \\ -65sKNsHBcHS \\ +60cHBcHS \\ +35cHS \end{array} \\
\begin{array}{l} cAScABcKNcHBsHS \\ -cASsABsKNcHBsHS \\ -cAScABsKNsHBsHS \\ -cASsABcKNsHBsHS \\ -sAScHS \end{array} &
\begin{array}{l} sASsABsKNcHBsHS \\ -sAScABcKNcHBsHS \\ +sAScABsKNsHBsHS \\ +sASsABcKNsHBsHS \\ -cAScHS \end{array} &
\begin{array}{l} sABcKNcHBsHS \\ +cABsKNcHBsHS \\ -sABsKNsHBsHS \\ +cABcKNsHBsHS \end{array} &
\begin{array}{l} 35cABcKNcHBsHS \\ -35sABsKNcHBsHS \\ -35cABsKNsHBsHS \\ -35sABcKNsHBsHS \\ +65cKNcHBsHS \\ -65sKNsHBsHS \\ +60cHBsHS \\ +35sHS \end{array} \\
\begin{array}{l} cAScABcKNsHB \\ -cASsABsKNsHB \\ +cAScABsKNcHB \\ +cASsABcKNcHB \end{array} &
\begin{array}{l} sASsABsKNsHB \\ -sAScABcKNsHB \\ -sAScABsKNcHB \\ -sASsABcKNcHB \end{array} &
\begin{array}{l} sABcKNsHB \\ +cABsKNsHB \\ +sABsKNcHB \\ -cABcKNcHB \end{array} &
\begin{array}{l} -35sABsKNsHB \\ +35cABcKNsHB \\ +35cABsKNcHB \\ +35sABcKNcHB \\ +65cKNsHB \\ +65sKNcHB \\ +60sHB \end{array} \\
0 & 0 & 0 & 1
\end{pmatrix} \qquad (5.12)
$$

To save space, Cosine (cos) is abbreviated with *c* and Sine (sin) with *s*. Due to the symmetry, only one matrix had to be calculated for both legs. The matrix above, as well as Formula *5.7*, refers to a hip (base) coordinate system with an origin in the hip servo's axis. The elements of matrix *5.12* are obviously in a way symmetric. Applying the following trigonometrical theorems $\cos\delta \cdot \cos\gamma \mp \sin\delta \cdot \sin\gamma = \cos(\delta \pm \gamma)$ and $\sin\delta \cdot \cos\gamma \pm \cos\delta \cdot \sin\gamma = \sin(\delta \pm \gamma)$ repeatedly, reduces the 231 trigonometrical functions of *5.12* to only 39.

$$
{}^{0}_{5}T = \begin{pmatrix}
\begin{array}{c} cAScHSc(AB+KN+HB) \\ +sASsHS \end{array} &
\begin{array}{c} -sAScHSc(AB+KN+HB) \\ +cASsHS \end{array} &
cHSs(AB+KN+HB) &
cHS \cdot \begin{pmatrix} 35 \cdot (1 + c(AB+KN+HB)) \\ +65c(KN+HB) + 60cHB \end{pmatrix} \\
\begin{array}{c} cASsHSc(AB+KN+HB) \\ -sAScHS \end{array} &
\begin{array}{c} -sASsHSc(AB+KN+HB) \\ -cAScHS \end{array} &
sHSs(AB+KN+HB) &
sHS \cdot \begin{pmatrix} 35 \cdot (1 + c(AB+KN+HB)) \\ +65c(KN+HB) + 60cHB \end{pmatrix} \\
cASs(AB+KN+HB) & -sASs(AB+KN+HB) & -c(AB+KN+HB) &
\begin{array}{c} 35s(AB+KN+HB) \\ +65s(KN+HB) + 60sHB \end{array} \\
0 & 0 & 0 & 1
\end{pmatrix}
$$

$$(5.13)$$

Introducing a new base coordinate system, which is the same for both legs and lies in between the two former ones, only needs a few changes in the matrix. Element (4,1) must be extended by a constant value, which specifies a displacement along the former base coordinate system's y_i-axis to its new position in the middle of the hip. Not only are transformations from one foot to the hip needed, but also vice versa. Therefore, the matrix shown above must be inverted. In addition to this, matrices translating from shank to hip, and from thigh to hip are necessary. These matrices are also needed in both directions, bottom-up and top down.

This brings up a set of twelve matrices: Foot-to-Hip, Shank-to-Hip, and Thigh-to-Hip, all of them for the left and the right leg, and all inverted. Below, all matrices are shown, to point out the similarities:

Foot-to-Hip with common-base extension:

$$
{}^{Hip}_{Foot}T_{\substack{left\\right}} = \begin{pmatrix}
\begin{array}{c} cAScHSc(AB+KN+HB) \\ +sASsHS \end{array} & \begin{array}{c} -sAScHSc(AB+KN+HB) \\ +cASsHS \end{array} & cHSs(AB+KN+HB) & cHS\cdot\begin{pmatrix} 35\cdot(1+c(AB+KN+HB)) \\ +65c(KN+HB)+60cHB \end{pmatrix} \\
\begin{array}{c} cASsHSc(AB+KN+HB) \\ -sAScHS \end{array} & \begin{array}{c} -sASsHSc(AB+KN+HB) \\ -cAScHS \end{array} & sHSs(AB+KN+HB) & sHS\cdot\begin{pmatrix} 35\cdot(1+c(AB+KN+HB)) \\ +65c(KN+HB)+60cHB \end{pmatrix}\pm 28 \\
cASs(AB+KN+HB) & -sASs(AB+KN+HB) & -c(AB+KN+HB) & \begin{array}{c} 35s(AB+KN+HB) \\ +65s(KN+HB)+60sHB \end{array} \\
0 & 0 & 0 & 1
\end{pmatrix}
$$

(*5.14*)

Hip-to-Foot:

$$
{}^{Foot}_{Hip}T_{\substack{left\\right}} = \begin{pmatrix}
\begin{array}{c} cAScHSc(AB+KN+HB) \\ +sASsHS \end{array} & \begin{array}{c} cASsHSc(AB+KN+HB) \\ -sAScHS \end{array} & cASs(AB+KN+HB) & \begin{array}{c} -cAS\cdot\begin{pmatrix} 35\cdot(1+c(AB+KN+HB)) \\ +60c(AB+KN)+65cAB \end{pmatrix} \\ \mp 28\begin{pmatrix} cASsHSc(AB+KN+HB) \\ -sAScHS \end{pmatrix} \end{array} \\
\begin{array}{c} -sAScHSc(AB+KN+HB) \\ +cASsHS \end{array} & \begin{array}{c} -sASsHSc(AB+KN+HB) \\ -cAScHS \end{array} & -sASs(AB+KN+HB) & \begin{array}{c} sAS\cdot\begin{pmatrix} 35\cdot(1+c(AB+KN+HB)) \\ +60c(AB+KN)+65cAB \end{pmatrix} \\ \mp 28\begin{pmatrix} -sASsHSc(AB+KN+HB) \\ -cAScHS \end{pmatrix} \end{array} \\
cHSs(AB+KN+HB) & sHSs(AB+KN+HB) & -c(AB+KN+HB) & \begin{array}{c} -35s(AB+KN+HB) \\ -60s(AB+KN)-65sAB \\ \mp 28(s(AB+KN+HB)sHS) \end{array} \\
0 & 0 & 0 & 1
\end{pmatrix}
$$

(*5.15*)

Shank-to-Hip:

$$
{}^{Hip}_{Shank}T_{\substack{left\\right}} = \begin{pmatrix} \mathrm{c}(HS)\cdot\mathrm{c}(HB+KN) & -\mathrm{c}(HS)\cdot\mathrm{s}(HB+KN) & \mathrm{s}(HS) & \mathrm{c}(HS)\cdot\big(65\,\mathrm{c}(HB+KN)+60\,\mathrm{c}(HB)+35\big) \\ \mathrm{s}(HS)\cdot\mathrm{c}(HB+KN) & -\mathrm{s}(HS)\cdot\mathrm{s}(HB+KN) & -\mathrm{c}(HS) & \mathrm{s}(HS)\cdot\big(65\,\mathrm{c}(HB+KN)+60\,\mathrm{c}(HB)+35\big)\pm 28 \\ \mathrm{s}(HB+KN) & \mathrm{c}(HB+KN) & 0 & 65\,\mathrm{s}(HB+KN)+60\,\mathrm{s}(HB) \\ 0 & 0 & 0 & 1 \end{pmatrix}
$$

(*5.16*)

Hip-to-Shank:

$$
{}^{Shank}_{Hip}T_{\substack{left\\right}} = \begin{pmatrix} \mathrm{c}(HS)\cdot\mathrm{c}(HB+KN) & \mathrm{s}(HS)\cdot\mathrm{c}(HB+KN) & \mathrm{s}(HB+KN) & -\Big(\mathrm{c}(HB+KN)\cdot\big(\pm 28\,\mathrm{s}(HS)+35\big)+60\,\mathrm{c}(KN)+65\Big) \\ -\mathrm{c}(HS)\cdot\mathrm{s}(HB+KN) & -\mathrm{s}(HS)\cdot\mathrm{s}(HB+KN) & \mathrm{c}(HB+KN) & \mathrm{s}(HB+KN)\cdot\big(\pm 28\,\mathrm{s}(HS)+35\big)+60\,\mathrm{s}(KN) \\ \mathrm{s}(HS) & -\mathrm{c}(HS) & 0 & \pm 28\,\mathrm{c}(HS) \\ 0 & 0 & 0 & 1 \end{pmatrix}
$$

(*5.17*)

Thigh-to-Hip:

$$
{}^{Hip}_{Thigh}T_{\substack{left\\right}} = \begin{pmatrix} \mathrm{c}(HS)\cdot\mathrm{c}(HB) & -\mathrm{c}(HS)\cdot\mathrm{s}(HB) & \mathrm{s}(HS) & \mathrm{c}(HS)\cdot\big(60\,\mathrm{c}(HB)+35\big) \\ \mathrm{s}(HS)\cdot\mathrm{c}(HB) & -\mathrm{s}(HS)\cdot\mathrm{s}(HB) & -\mathrm{c}(HS) & \mathrm{s}(HS)\cdot\big(60\,\mathrm{c}(HB)+35\big)\pm 28 \\ \mathrm{s}(HB) & \mathrm{c}(HB) & 0 & 60\,\mathrm{s}(HB) \\ 0 & 0 & 0 & 1 \end{pmatrix} \qquad (5.18)
$$

Hip-to-Thigh:

$$
{}^{Thigh}_{Hip}T_{\substack{left\\right}} = \begin{pmatrix} \mathrm{c}(HS)\cdot\mathrm{c}(HB) & \mathrm{s}(HS)\cdot\mathrm{c}(HB) & \mathrm{s}(HB) & -\mathrm{c}(HB)\cdot\big(\pm 28\,\mathrm{s}(HS)+35\big)-60 \\ -\mathrm{c}(HS)\cdot\mathrm{s}(HB) & -\mathrm{s}(HS)\cdot\mathrm{s}(HB) & \mathrm{c}(HB) & \mathrm{s}(HB)\cdot\big(\pm 28\,\mathrm{s}(HS)+35\big) \\ \mathrm{s}(HS) & -\mathrm{c}(HS) & 0 & \pm 28\,\mathrm{c}(HS) \\ 0 & 0 & 0 & 1 \end{pmatrix} \qquad (5.19)
$$

All matrices show a small number of different trigonometrical functions, which are used in altered combinations. Substitutions can reduce the number of trigonometrical calculations by using pre-calculation and a look up table (see chapter 9).

The following table shows which functions are needed in which matrices.

Trig. Function	Sub	(5.14)	(5.15)	(5.16)	(5.17)	(5.18)	(5.19)
cos(HS)	a	x	x	x	x	x	x
sin(HS)	b	x	x	x	x	x	x
cos(HB)	e	x		x		x	x
sin(HB)	f	x		x		x	x
cos(KN)	e’				x		
sin(KN)	f’				x		
cos(AB)	e’’		x				
sin(AB)	f’’		x				
cos(AS)	i	x	x				
sin(AS)	h	x	x				
cos(KN+HB)	o	x		x	x		
sin(KN+HB)	p	x		x	x		
cos(AB+KN)	o’		x				
sin(AB+KN)	p’		x				
cos(AB+KN+HB)	u	x	x				
sin(AB+KN+HB)	v	x	x				

Table 5.4: Substitutions and Distribution of Trigonometrical Functions

This table illustrates that, although sixteen different substitutions exist, only a maximum of ten are needed at any given time. The red ellipses point out the altering dependencies between forward and backward calculation; blue rectangles indicate invariants. This table is later used to generate efficient source code.

The coordinate systems and matrices derived in this chapter provide a compact method to calculate the positions of all links for Andy Droid relative to each other and in the coordinate system which fits the current problem best. The results above do not provide the opportunity to derive angles from given positions, because this problem is not one-to-one according to the given hardware setup. Dynamics are also not considered. However, the robot’s masses and their distribution will be investigated in the next chapter.

6. Mass Distribution and Centre of Gravity

This chapter will discuss the way in which the masses of the single moving bodies of the robot affect the total centre of mass. There are two different ways how the normal projection of the centre of mass (NPCM) can be determined. One method is recording the NPCM from the physical robot, and the other is by calculating it without having sensory feedback.

6.1. Force Gauging and Processing

The foot force sensors not only provide the opportunity to measure the force or detect the ground contact, but they also can be used to determine the load distribution for each foot.

As illustrated in Figure 4.3 in Chapter 4, each foot of the robot is constructed from three spring metal strips to which amplified strain gauges are attached. The borders of these toes form a triangle. Now the force on each toe can be determined with the sensors and the ratio of the values to each other give the centre of force in this triangle.

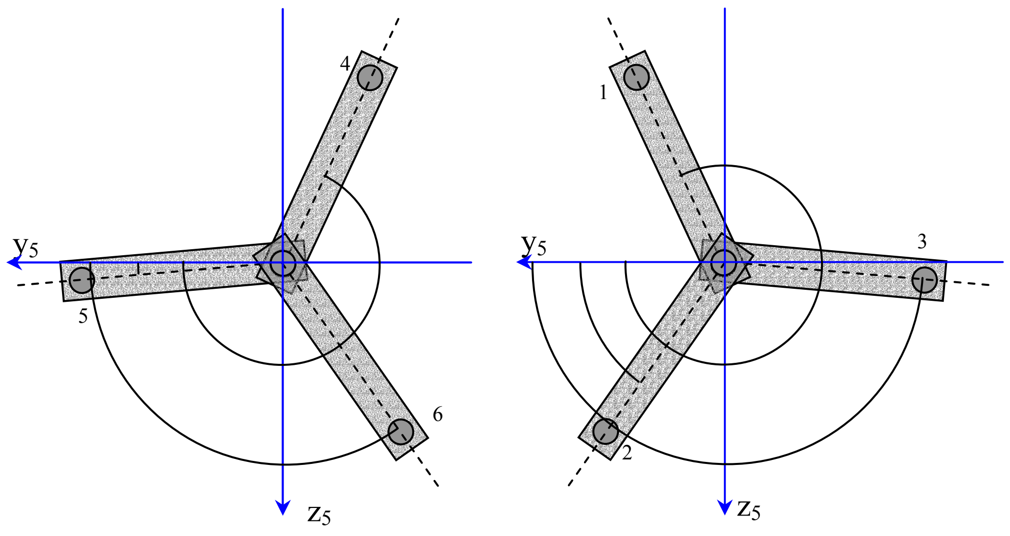


Figure 6.1: Toe-Setup

The illustration above shows both feet schematically, as seen from underneath the robot. This means, the left part in this picture represents the right foot and vice versa. The walking direction is against the direction of the *z*-axis. The toes are arranged in a star, so that they form a triangle of support. Thus, each foot creates a triangular safe operational area (SOAR). The polygon formed by the corners of both feet is a hexagon. While both feet are attached to the ground, the safe operational area extends to the area surrounded by this hexagon.

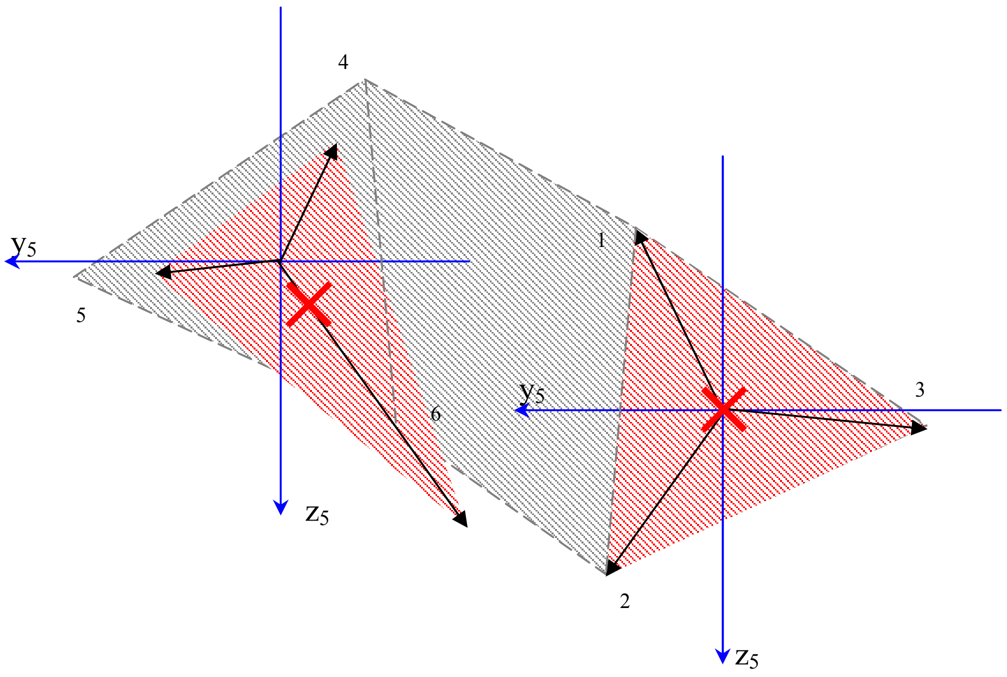


Figure 6.2: Centre of Force per Foot

<u>The main hypothesis for balancing states</u>: the robot is statically stable as long as the normal projection of the centre of mass is in the area surrounded by the polygon, which is formed by the feet’s corners.

This hypothesis was confirmed by A. L. Kun in his dissertation on “A Sensory-Based Adaptive Control Algorithm for Variable Speed Biped Robots” [6].
Figure 6.2 shows the two triangles formed by the toes and the hexagonal SOAR (grey-striped area). The analog-to-digital converted measurements are mathematically processed. Firstly, the offset is taken by reading the values from the A/D-Ports when the robot is not attached to the ground. Secondly, the average value of all toes is determined while the robot stands on his feet. At this time, no additional force other than the robot’s own gravitation should be exerted. Furthermore, only the difference between the actual value read and the offset is processed. This removes long term offset-drift influences. The values mentioned later all refer to this difference.

To calculate the centre of force of each foot, a value is measured, which represents the length of a vector in the same orientation as the associated physical toe. The three vectors again construct a triangle, but a virtual one. As the measured value equals the mean, the length of this vector is the same as the physical toe itself. The strain gauges are assumed linear, so that the length of the vector is linearly expanded with the value. The physical centre of force is represented by the result of this calculation, as it is the centre of gravity of the virtual triangle.

$$\vec{v}_{COF} = \tfrac{1}{3} \cdot \sum_{Toe=1}^{3} \tfrac{x_{act.,Toe}}{\bar{x}} \cdot \vec{v}_{Toe,phys.} \qquad (6.1)$$

The equation shown in (6.1) explains how the centre of force of each foot is calculated. Figure 6.2 illustrates the same. The striped red triangular area shows the virtual triangles, whose centres of gravity are marked with a red cross. The right side shows the state, when the average force is exerted on all toes. The left side demonstrates the state when the centre of mass on the right leg is displaced.

Remark: The relationship between the change of the resistance of the strain gauges and the measured voltage in a quarter-bridge is $\frac{V_d}{V_{CC}} = \frac{\Delta R}{4R + 2\Delta R} \approx \frac{1}{4} \cdot \frac{\Delta R}{R}$ as long as ΔR is small compare to R. When the bending radius itself is not too small, so that the relative change in length of the spring metal is nearly constant all over the measured length, then the strain of the gauges is constant. The ratio of relative resistance change to strain is called the *k*-factor. This factor depends on the gauge itself. This means: $\frac{V_d}{V_{Bridge}} \approx \frac{1}{4} \cdot \frac{\Delta R}{R} = \frac{1}{4} \cdot k \cdot \varepsilon$ is linear as long as strain ε is linear. For a detailed derivation, see [7]. The *k*-factor of the used strain gauges was not determinable and because of the lack of the appropriate measuring equipment, an exact calibration of the toes could not be performed. As mentioned above, only differences were processed, relative to the average values. The measured values were assumed linear.

The effect of both feet's COFs on the NPCM of the whole robot will be explained in section 3 of this chapter.

6.2. Link Masses and Positions

The last section described how the centre of force could be derived from the foot sensors, for each foot. This section illustrates how the masses are arranged in a way in which the centre of mass can be determined without requiring any sensors.

The robot comprises a large number of single moving masses. The first simplification assumed is that the wiring is not considered but its mass is distributed statistically to the rest. This simplification is justified, as the mass of the wiring is negligible compared to that of all other bodies. The masses of the controller and the servos have been taken from their technical specification. As far as it was possible to disassemble the robot, all other single parts have been weighed. Parts like the toes and mounts were calculated by their volume and the specific weight of the material. The following table shows the determined weights:

EyeBot Controller	190g
Batteries	320g
Servo	56g
Upper Body’s Mount and Wiring	131g
Thigh’s Mount and Wiring	35g
Shank’s Mount and Wiring	7g
Foot’s Mount, Wiring and Toes	52.5g
90°-Link (Hip-Thigh and Shank-Foot)	12.5g

Table 6.1: Single Weights

The complete robot showed a mass of 1440g on a scale. The accuracy of the scale used was 2g. The calculated weights have been adapted so as to reach the total mass measured.
As mentioned above the number of independently moving bodies was reduced. In addition, the 90°-links also had a small mass. Thus, their mass was also distributed to their neighbouring bodies.

This results in the following masses:

Upper Body	Batteries, 2 Servos, Controller, Mount	765.50g
Thigh	2 Servos, Mount	153.25g
Shank	1 Servo, Mount	69.25g
Foot	1 Servo, Mount, 3 Toes	114.75g

Table 6.2: Moving Masses

The masses shown in Table 6.2 are not to been seen as real, but of statistical character which represents the distribution over the whole robot. Therefore, it is justified to use the values in an unrealistic level of preciseness.

Once having determined the masses, the centre of each mass has to be assigned. This was done in the way that all positions in each coordinate direction had been added after multiplying it by its weight. Nevertheless, the centres of mass are more of an estimated character. The positions are related to the coordinate system that is fixed to the moving body. This results in the following vectors:

Foot: $${}^{5}\vec{V}_{\text{Foot}} = \begin{pmatrix} 12 \\ \pm 10 \\ 18 \end{pmatrix}_{\text{Leg}_{\text{right}}^{\text{left}}} \quad \Rightarrow \quad {}^{0}\vec{V}_{\text{Foot}} = {}^{0}_{5}T \cdot {}^{5}\vec{V}_{\text{Foot}}$$

Shank: $${}^{3}\vec{V}_{\text{Shank}} = \begin{pmatrix} -15 \\ 0 \\ \mp 17 \end{pmatrix}_{\text{Leg}_{\text{right}}^{\text{left}}} \quad \Rightarrow \quad {}^{0}\vec{V}_{\text{Shank}} = {}^{0}_{3}T \cdot {}^{3}\vec{V}_{\text{Shank}}$$

Thigh: $${}^{2}\vec{V}_{\text{Thigh}} = \begin{pmatrix} -30 \\ 0 \\ \mp 17 \end{pmatrix}_{\text{Leg}_{\text{right}}^{\text{left}}} \quad \Rightarrow \quad {}^{0}\vec{V}_{\text{Thigh}} = {}^{0}_{2}T \cdot {}^{2}\vec{V}_{\text{Thigh}}$$

Body: $${}^{0}\vec{V}_{\text{Body}} = \begin{pmatrix} -60 \\ 0 \\ -31 \end{pmatrix}_{\text{HipCentre}}$$

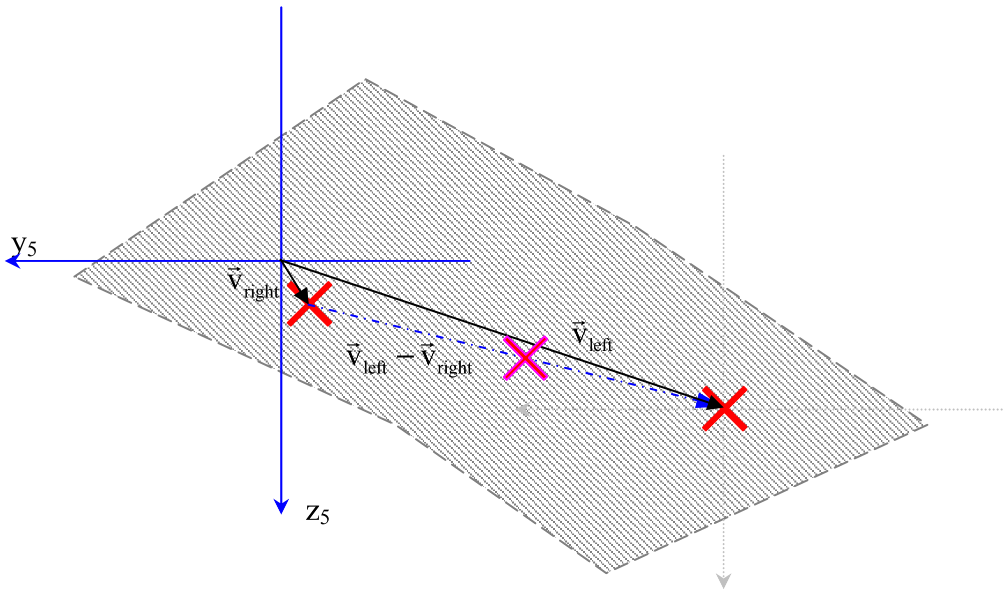
The total centre of mass is as follows: $${}^{0}\vec{V}_{\text{total}} = \frac{1}{m_{\text{total}}} \sum_{i} m_i \cdot {}^{0}\vec{V}_i \,. \qquad (6.2)$$

6.3. Calculation of the NPCM

Having gathered the centre of force of each foot, as described in Section 6.1., calculation of the total projection of the centre of mass (NPCM) is simplified. The plane on which the toes of one foot rest is situated 28mm in positive direction of the *x*-axis of a foot. If the foot rests on the subsurface, then this is coplanar to the *y*-*z*-plane. The bending of the spring metal stripes is neglected. The centre of force now is a distinct vector in each foot's coordinate system made up of the distance in *x*-direction as mentioned and its components in *y*- and *z*- direction as shown in Figure 6.2.

In the trivial case, when one foot is lifted, the NPCM is identical with the centre of force of the foot that rests on the ground. As long as both feet are attached to ground, the NPCM lies on the line that connects both centres of force. Therefore, both vectors are transformed into one coordinate system. The missing information is the distribution of the load between both feet. Therefore, the sum of all toe-sensor values of one foot must be in proportion to the total of all sensor values.

$$\vec{v}_{\mathrm{NPCM}} = \vec{v}_{\mathrm{right}} + \frac{s_{\mathrm{right}}}{s_{\mathrm{left}} + s_{\mathrm{right}}} \cdot \left(\vec{v}_{\mathrm{left}} - \vec{v}_{\mathrm{right}}\right) = \frac{1}{s_{\mathrm{left}} + s_{\mathrm{right}}} \cdot \left(s_{\mathrm{left}} \cdot \vec{v}_{\mathrm{right}} + s_{\mathrm{right}} \cdot \vec{v}_{\mathrm{left}}\right) \quad (6.3)$$



y5
z5
$\vec{v}_{\mathrm{right}}$
$\vec{v}_{\mathrm{left}}$
$\vec{v}_{\mathrm{left}} - \vec{v}_{\mathrm{right}}$

Figure 6.3: Measured NPCM

According to Chapter 6.2., the calculated centre of mass is ${}^{0}\vec{V}_{\text{total}} = \frac{1}{m_{\text{total}}}\sum_{i} m_i \cdot {}^{0}\vec{V}_i$.

Now the resulting NPCM is, as its name says, the normal projection of this vector to the ground plane. Without any further information on the absolute tilt or pitch angle of the robot this could not be done. As long as the ground plane is horizontal, it is sufficient to project the previously calculated centre of mass to the foot plane. This is done by transforming it into the foot's coordinate system and replacing the vector's *x*-value by 28, which means that it is projected into the plane x_5=28 (normal form of a plane). Obviously, it has to be given in the coordinate system of the foot, which is attached to the ground.

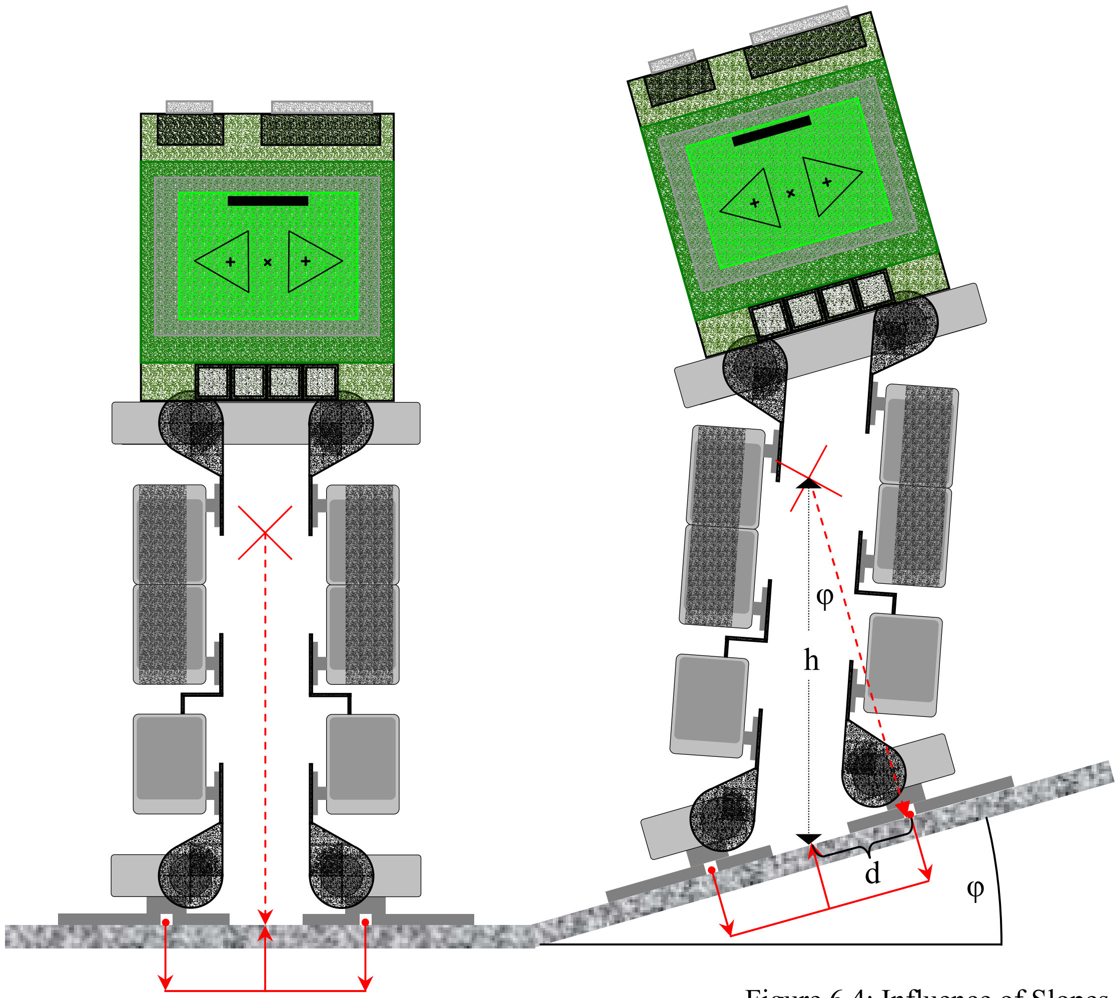


Figure 6.4: Influence of Slopes

As the figure above shows, the NPCM derived from the sensors and the calculated one are identical when the ground plane is horizontal. The example on the right shows a sloped surface and the robot in a stable position where the force on both feet is the same. In this case, only the sensored NPCM is realistic because the calculated version is not normally projected but orthogonal to the surface. This has a nice side-effect: Figure 6.4 names the length *d*, which simply is the *y*-component of the difference of both NPCMs. As marked in the figure, $\sin\varphi = \frac{d}{h}$, where *h* is the *x*-component of the centre of mass vector in a foot coordinate system. At the same time, φ is the angle between the slope and the horizontal in the rolling (sidewards) direction. The same could be calculated for the pitch angle by exchanging *y*- with *z*-components. This could make inclinometers obsolete.

As mentioned before, the servos do not have feedback and thus, their actual position must be trusted. They have an internal loop-back controller to hold the position but each gearbox has a backlash. These backlashes add up to tolerances that make both calculations incorrect. Flexibilities in the links caused problems as well and, thus, the links had to be exchanged with stiffer ones. Their weight did not increase because the former ones were made from steel, whereas the new ones are made from aluminium. But the uncertainties caused by the gearboxes still cannot be removed as the calculation of the total centre of mass is concerned. In the case of the measured NPCM, the system is relatively robust. The reason being that the supported area (hexagon) of the feet is large when compared to the robot's height and width. This allows tolerances in controlling the NPCM to a stable position.

7. Control System

With the results from the previous chapter there is a value well suited for a control loop. The measured NPCM represents global information on the robot's state as discussed before. The control algorithm presented in this chapter was designed according to "*Taschenbuch der Regelungstechnik*"[9], Chapter 11 „Digitale Regelungssysteme". All performance tests on API and user functions have been carried out by running the corresponding function in a loop repeatedly. The setup was an EyeBot MK4, running at 33MHz with RoBIOS 5.1 in 2MB RAM. Time stamps were taken from the internal timer and the overheads caused by the loops were taken into consideration as well.

7.1. Data Capture

As digital control systems act on analog control plants, some adaptations have to be made. Especially the data flow has to meet some requirements. This section discusses some problems concerning data validity.

As mentioned before, the data from the strain gauges are A/D converted. The 10-bit A/D converter works on an 8-channel multiplexer. The structure diagram is shown below, or see data sheet [10] for detailed information.

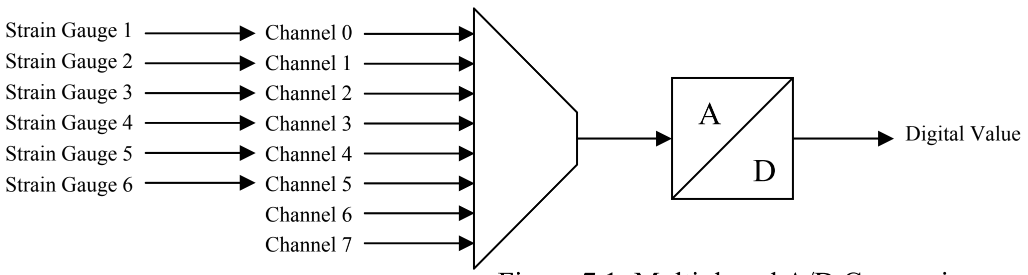


Figure 7.1: Multiplexed A/D Conversion

To get a value by calling the API-function `OSGetAD(channel_#)`, the single-chip module is forced to switch to the desired channel and retrieve the value. As the input circuit of the inbuilt sample circuit contains a capacitor, it takes some time to get this capacitor charged, before the hold-latch disconnects the outer circuit. This time is not only determined by the time constant of the input impedance of the A/D converter, but also of the output impedance of the measured circuit which is the strain gauge amplifier AMP04 (Analog Devices) itself. According to the data sheet of this instrumentation amplifier ([11], page 14, figure 31) the output impedance is below 20Ω and therefore negligible. The A/D converter needs some time to acquire the correct voltage: $t_{AZ} = 9 \cdot (R_{Source} + R_{IN}) \cdot 16pF = 9 \cdot (20\Omega + 5k\Omega) \cdot 16pF \approx 730ns$ (according to its data sheet [10] page 8, upper left paragraph "Track/Hold", in this setup). The call of one `OSGetAD(channel_#)` takes 20 times as long. Yet, tests have shown, that without any changes on the measured bridge, it takes at least 3 times as long as one reading to retrieve the correct value. During normal operation, it is assumed to wait at least ten times the duration of one call between switching to the channel and reading the value assumed to be valid.

This means a latency of at least 150µs. As there are 6 strain gauges to read and the channel must not be switched during one wait-state, it takes 900µs to read the strain gauges in total. Performance tests with the method that captures data and processes it, in order to make it available for the calling function, have shown that the time consumption is variable. In total, a pre-processed data capture of all sensors with the wait-states mentioned before, takes up to 6ms. This means that the maximum possible sampling frequency on one strain gauge is $f_S = \frac{1}{1000\mu s} = 1kHz$. Faster sampling requires a different A/D converter.

Actually, the low-pass filter, which is built in the amplification circuit has a filter-frequency of $f_{LP} = \frac{1}{2\pi \cdot 100k\Omega \cdot C_{EXT}} = 1.061kHz$ as $C_{EXT} = 1.5nF$. This means that the gauges are under-sampled and aliasing could occur, if the measured signal carries the respective frequencies. Thus, the low pass filter frequency has to be set to at most 500Hz.

7.2. Control Loop Frequency and Sample Period

As RoBIOS allows calling interrupt routines at a maximum frequency of 100Hz, the shortest period of the control loop could be 10ms. Nevertheless, as the computing time consumption of the control loop is between 10ms and 13ms, the interrupt should run at a maximum of 50Hz. This should provide nearly the same computation time for higher-level applications.

To establish a quasi-continuous digital control loop, it is necessary that the sampling time is short, compared to the dominating time-constant of the control plant. The ratio of both should be 10, which means minimal 10 times over-sampling. Therefore, the maximum frequency of the control plant must be reduced by shortening the low-pass bandwidth again. Regarding Shannon's theorem and the demand of 10 times over-sampling, $f_{CL} = 10 \cdot f_{Shannon} = 20 \cdot f_{Max,Plant} \overset{!}{=} 20 \cdot f_{Low-Pass}$ (*7.1*). This means, the highest frequencies required by the controller are below 2.5Hz. Improving the performance would only be possible by either reducing the time consumption of the controller software or increasing the computation power of the hosting controller.

The time consumption of the control loop, as mentioned above, is taken while the loop used the maximum servo speed available. If one intends to smooth the servos' movements, it should be noted that reducing the speed increases the time consumption of the whole control loop. The speed is reduced by inserting pauses in the driving commands that must return before other computations can be performed. Consequently, the available computation time for higher-level applications is reduced and easily suppressed at all, as interrupt-routines have higher priority.
The function, which calculates the subsurface slope angle should not be called within the control loop, as it is not necessary and consumes about 40ms computation time.

7.3. Control Algorithm

According to Taschenbuch der Regelungstechnik [9], Chapter 11.2.4.3, a control algorithm for PID-controllers can be derived from the analog standard PID controller.

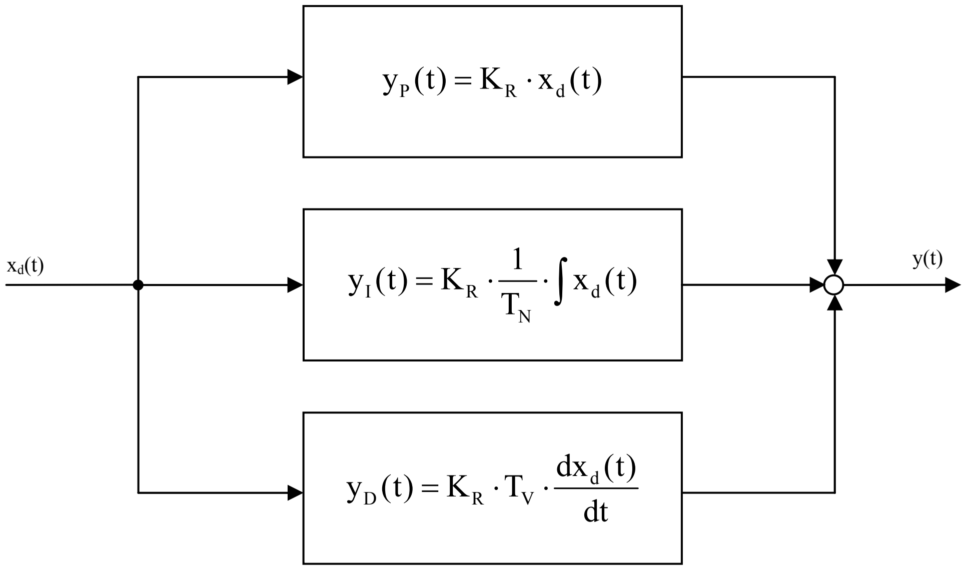

$y_P(t) = K_R \cdot x_d(t)$
$x_d(t)$
$y_I(t) = K_R \cdot \frac{1}{T_N} \cdot \int x_d(t)$
$y(t)$
$y_D(t) = K_R \cdot T_V \cdot \frac{dx_d(t)}{dt}$

Figure 7.2: PID Controller Structure

The differential quotients are replaced by discrete difference quotients. The input-output-relation $y_k = \sum_{i=0}^{N} b_i x_{d,k-1} + \sum_{i=1}^{M} a_i y_{k-1}$ uses the rectangular approximations $\dot{x} \approx \frac{x_k - x_{k-1}}{T}$ and $\ddot{x} \approx \frac{x_k - 2x_{k-1} + x_{k-2}}{T^2}$. This leads to the standard PID control algorithm:

$$y_k = y_{k-1} + K_R \cdot \left[\left(1 + \frac{T}{T_N} + \frac{T_V}{T}\right) \cdot x_{d,k} - \left(1 + 2 \cdot \frac{T_V}{T}\right) \cdot x_{d,k-1} + \frac{T_V}{T} \cdot x_{d,k-2} \right] \qquad (7.2).$$

The algorithm can be programmed according to the following flow chart:

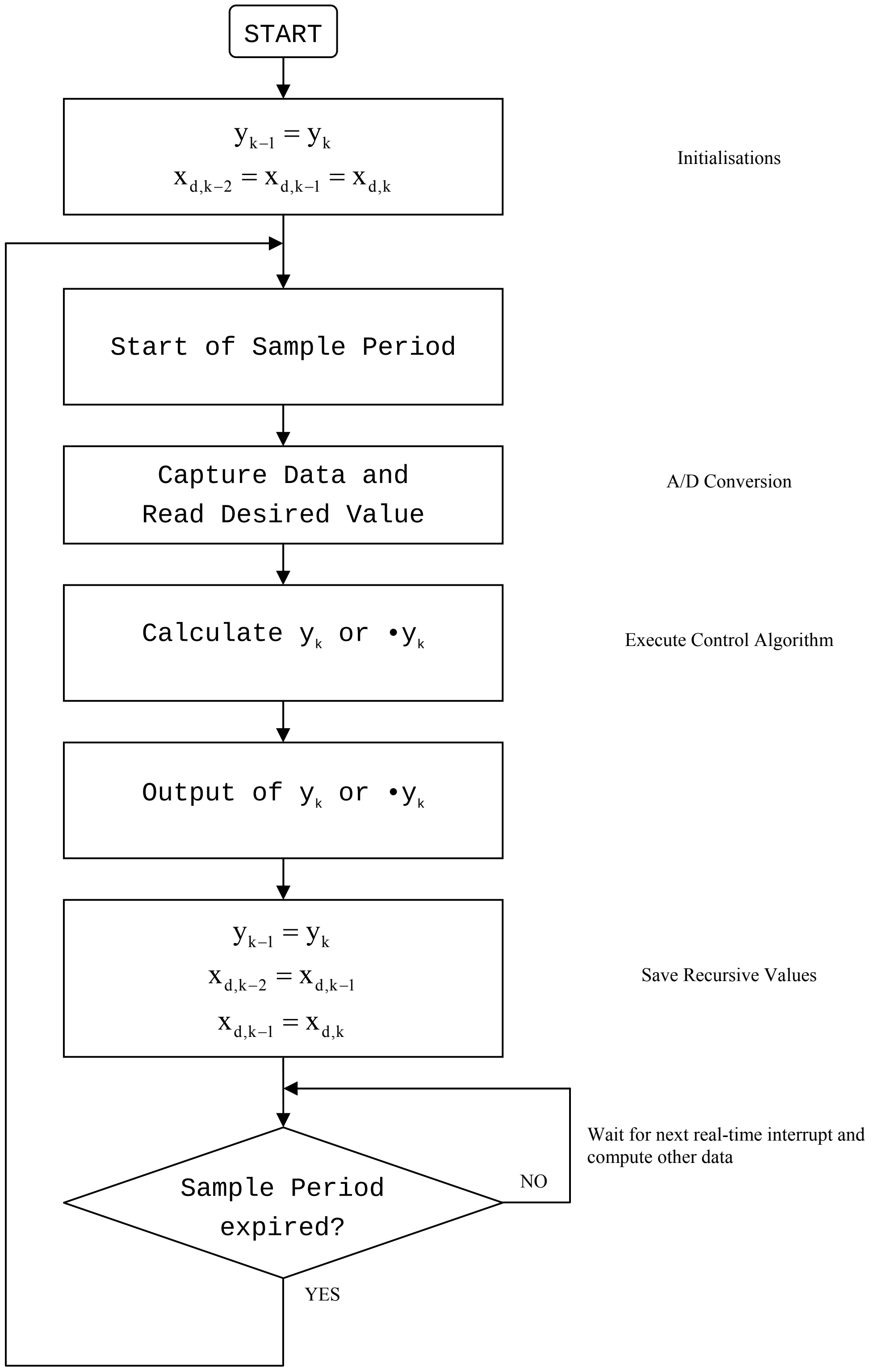

START
$y_{k-1} = y_k$
$x_{d,k-2} = x_{d,k-1} = x_{d,k}$
Initialisations
Start of Sample Period
Capture Data and Read Desired Value
A/D Conversion
Calculate y_k or $\bullet y_k$
Execute Control Algorithm
Output of y_k or $\bullet y_k$
$y_{k-1} = y_k$
$x_{d,k-2} = x_{d,k-1}$
$x_{d,k-1} = x_{d,k}$
Save Recursive Values
Wait for next real-time interrupt and compute other data
NO
Sample Period expired?
YES

Figure 7.3: Control Algorithm Flow Chart

7.4. Control Loop

The controlling algorithm has been presented in the last section. The adaptation to the given plant means that the signals have to be identified and pre-, respectively post-processed to meet the given environment. Furthermore, one has to determine which values are controllable to get the intended result: Balancing.

As discussed in Chapter 6, the normal projection of the centre of mass represents global information on the robots state. As it is a projection, its degrees of freedom in terms of coordinates are reduced from three to two. The NPCM is located in the surface-plane under the robot's feet. As shown in Chapter 6, it can be expressed with the mathematical utilities from Chapter 5. According to the statements in Section 5.3, it is directly expressible in a foot's *y*- and *z*-coordinate. This solves the first problem of the controller: there is a value that can be controlled. As these coordinates are mathematically independent they have to be controlled independently. This is done by two PID-controllers; each attached to one coordinate: *y*-component for sidewards deviation from the desired value, and the *z*-component for front-back-deviation.

This brings up the next question: what is the desired value and how is it expressed? The structure of it is obvious: the controlled variable is a two-dimensional vector, thus the desired value has to meet the same structure to create a reasonable control-error. The controller calculates the desired value by itself. The basis to calculate it is the actual or desired foot's positions in global coordinates and the distribution of the load between both feet. Thereby it is guaranteed to control to a safe state, according to the SOAR defined in Chapter 6. This means, the controller has to be given the positions of the feet relative to the base coordinate system in the hip-centre and a percentage of how much load one foot should carry.

The control algorithm is applied to each coordinate. The output is used to drive the appropriate servos, affecting the total centre of mass in the controlled direction. The closed control-loop is shown in the following figure.

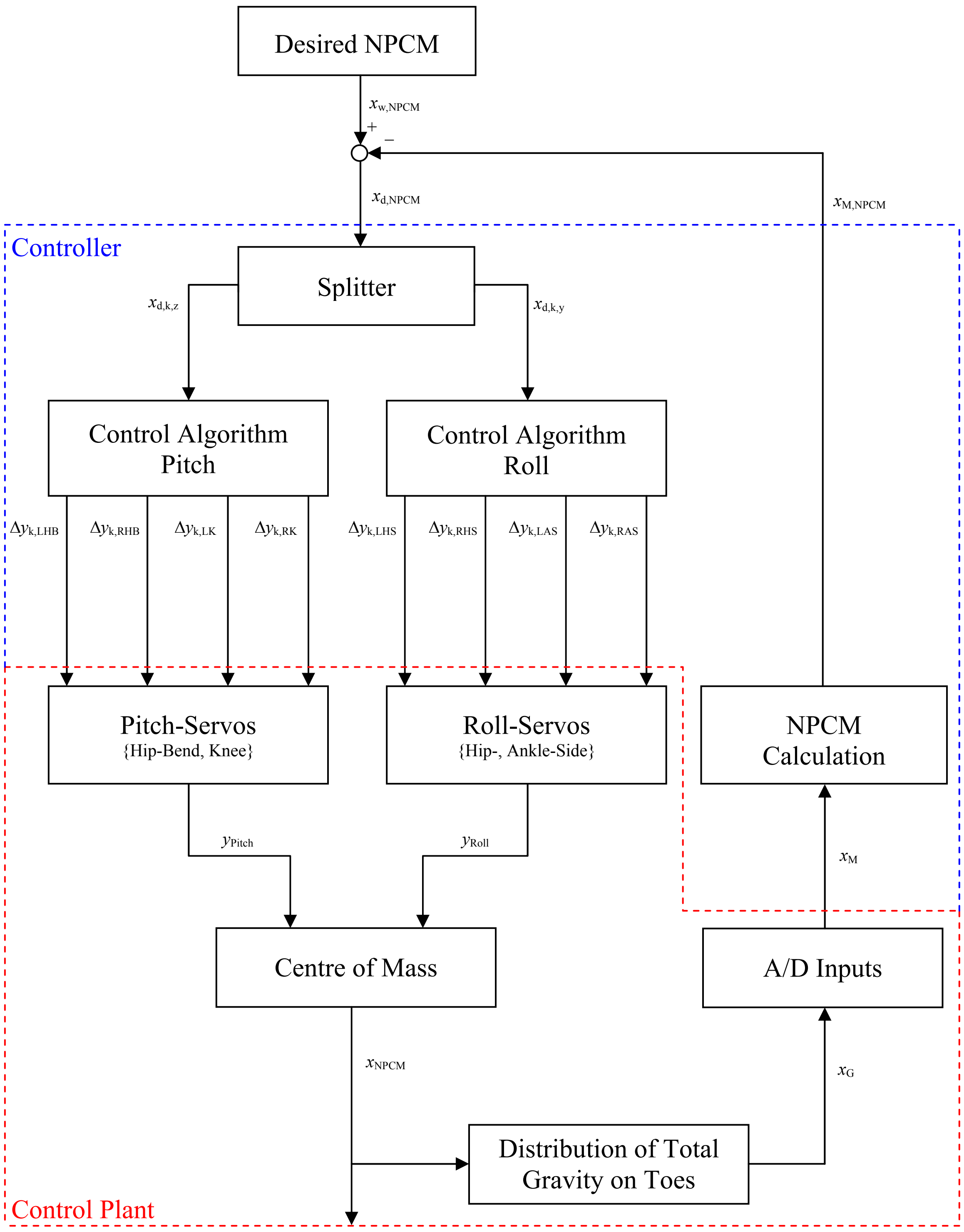


Figure 7.4: Control Loop

The lower red dashed-lined box indicates the control plant. It comprises all hardware components. This plant is obviously non-linear. The mathematical interrelationship is given in the previous chapters. However, the plant is too complex to model and describe in mathematical expressions. An experimental model could also not be taken because the only gauging instrument available was the A/D converter itself and it is too slow for an appropriate data capture. In addition to the non-linearity, the gaugeable value is not the NPCM directly, but can only be derived from it. The control algorithm explained in the last section is designed for linear systems only. For small changes in the servos' angles, the change of the NPCM is quasi-linear, as $\sin(\alpha) \approx \alpha \mid \alpha << \pi/4$. Therefore, it could be applied, although it is generally not suitable for the system. In lack of a model, even of a linearised model, the control-parameters, T_N, T_V, and K_R could only be adjusted according to the method of Ziegler and Nichols (see [9], Chapter *10.3.2 Einstellregeln von Ziegler und Nichols*).

Some tests showed that a differential part makes the plant unstable and, therefore, it was set to '0'. The proportional parameter was set to $K_R = 0.01$ and the integral parameter to $\frac{T}{T_N} = 1.5$. The algorithm still includes the differential terms to make it easily accessible if one wants to experiment with it.

8. SOFTWARE SYSTEM ARCHITECTURE

To provide concise software, it is necessary to arrange the code clearly. This is essential for efficient work during development and for a manageable starting-point for later modifications of this project.
Object oriented programming means that methods and data are contained by classes and instances. Therefore, real objects must be abstractedly described in software objects. Real objects could be either physical objects or problems, e.g. concerning mathematical, computational or data tasks. The software objects should have names, related to their real objects' names and they could interact among each other. They should provide abstraction by encapsulating several features in one object, modularise access by access-rights, and contain all data needed and directly related to the object's features.

8.1. SYSTEM STRUCTURE

In the case of an EyeBot and especially *Andy Droid*, abstraction is already provided by the API's C-functions. As the programming language C does not provide object oriented programming features, these functions had to be encapsulated in classes. The following figure shows the logic layers, the robot-model was divided into.

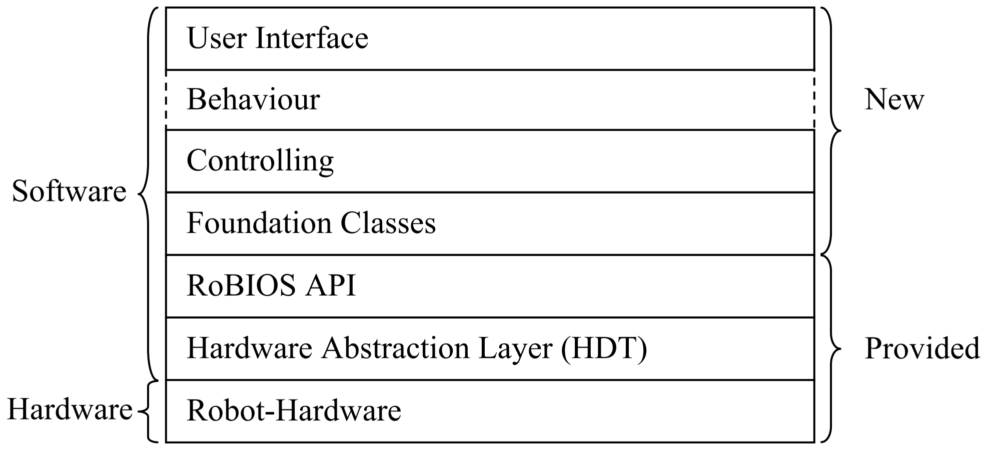


Figure 8.1: Layered System Structure

The lower three layers were already discussed in Chapter 4. The foundation classes provide encapsulations in C++-objects. This comprises mathematical features for linear algebra and Denavit Hartenberg calculations, as well as Singleton classes for hardware objects. The controlling layer implements the control algorithm explained in Chapter 7.

The layer on top of the control layer determines high-level behaviour, e.g. walking gait generation. At the top of the stack is the user interface. The behavioural implementations are rudimentary at this state of the project. This layer should be used for further implementations during future software work.

8.2. Singleton Classes

According to Chapter 5 of the script for the lecture "Softwaretechnologie II" by Prof. Hußmann at the Technical University of Dresden [8], singleton classes provide some advantages. They only allow one instance of themselves to be created, still obtaining the opportunity to be accessed from multiple points. The singleton design pattern is as the following code fragment shows:

```
class ClassName
{
    public:
     static ClassName* GetTheInstance();

    private:
     ClassName();

     static ClassName SingleInstance;
};

ClassName ClassName::SingleInstance;

ClassName* ClassName::GetTheInstance()
{
     return &SingleInstance;
}
```

The constructor of this class is declared private. This means, no functions, but member-functions of this class itself can create an instance. The only instance of this class is a static member of itself. The public function `GetTheInstance()` returns a pointer to this uniquely created instance. The way in which the Singleton design pattern is implemented and shown above differs from the method described by Hußmann. It was

proposed to have only a pointer to the single instance as a static member of the class. The method which returns this pointer, creates the instance if it does not exist. This means dynamic memory allocation, which requires a destructor to free the dynamic allocated memory. However, RoBIOS, or the compiler used does not support destructors. Therefore, Singletons were implemented in the way illustrated.
Access to the instance can be gained by calling the following line from anywhere in the user program:

```
ClassName *instance=ClasseName::GetTheInstance();
```

The scope operator enables the compiler to call the static member function without an instance and assign the function's start address correctly.

8.3. Class Structure

The idea behind Singletons is that a class is programmed as usual. However, by protecting the constructor and adding an instance of itself to its private member variables and one public function to retrieve a pointer to this instance, it is guaranteed that only one instance exists at the same time. In terms of data access, this could be achieved by declaring all member variables as static. In this case, constant member functions still could be accessed from different points of the software at the same time.
Different parts of the hardware must be accessed only one at a time by the software. The feature of Singleton Classes provides exactly the character of hardware components: they only exist once. To fulfil the requirement from the beginning of this chapter, this design pattern is appropriate for abstracting the real objects.

The following components have been designed as Singleton Classes:

- Legs, containing 10 servos	→ Servo-Class
- Feet, containing 6 sensors	→ Feet-Class
- Input, comprising keys and IR remote control	→ UserInput-Class
- Output, represented by the LCD	→ Display-Class

There are two more foundation classes, which are not designed as Singleton Classes because it is necessary to obtain multiple instances:

- Mathematical functions for matrix calculations	→ LinearAlgebra-Class
- Functions for Denavit Hartenberg calculations	→ DenavitHartenberg-Class

The latter inherits the mathematical features of previous one. For the next layer, controlling, a single class has been designed: Control-Class. This class is also a Singleton. However, if it becomes necessary during future work to have more than one controller, this class could easily be extended or redesigned. The behavioural layer exists in a testing class for balancing which includes the user interface.

The following structure shows the interdependencies of the foundation classes and the controlling layer:

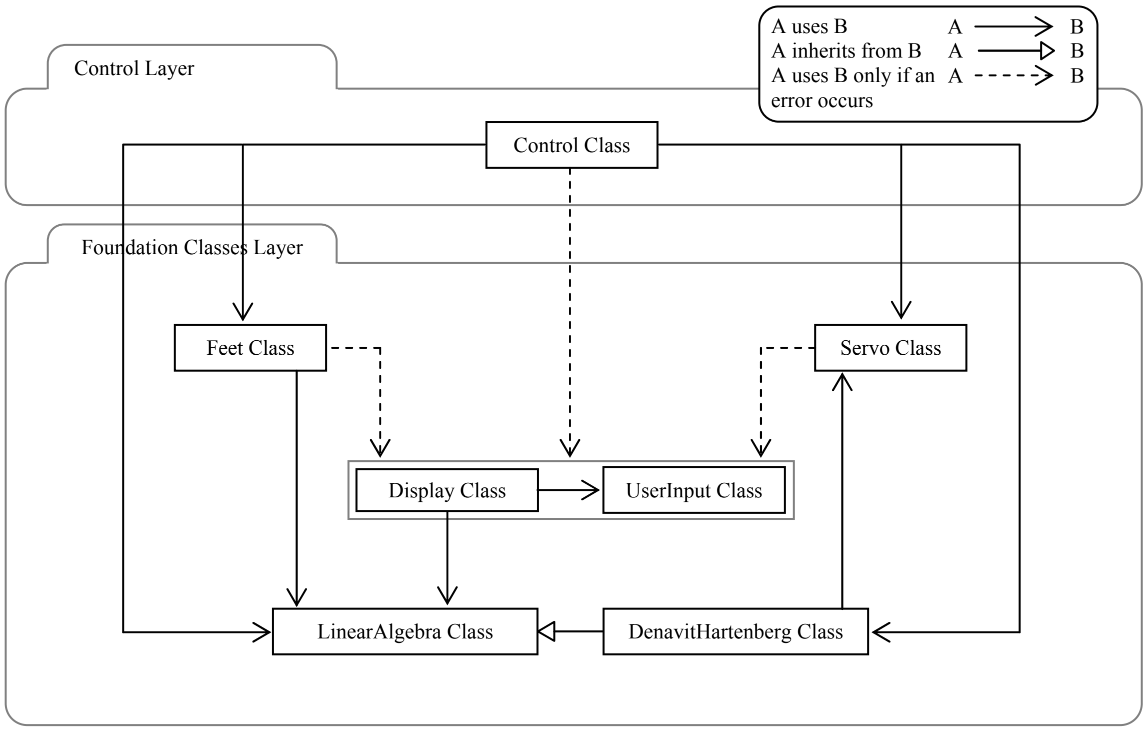


Figure 8.2: Class Structure

The main calculations take place in the DenavitHartenberg-Class, which has all the abilities inherited from its mother class LinearAlgebra. It accesses the Servo-Class only for retrieving the actual positions of the servos every time it calculates coordinate transformations. The Feet-class needs LinearAlgebra to use its structures and encapsulate sensored data in these. To synchronise some Outputs with the user, Display-class needs access to the UserInput-class. Due to standardised output-functions for LinearAlgebra-objects, it also uses this class. This output is mainly for debugging purposes. All dashed arrows are only accesses if an error occurs, whereas normally the highest layer accesses these two user-interfacing classes. The control layer mainly works with the servos and the feet. For some calculations, it also needs DenavitHartenberg and LinearAlgebra.

The layers above make use of these two straight top-down and, as the controlling layer exist of only one class, the structural diagram is obvious. For forward controlled behaviour and the user interface, the control layer can be passed by. Down bound layers are no more classes because the foundation class layer wraps the API's C-functions. These C-functions access the hardware via RoBIOS.

9. IMPLEMENTATION

Since the structure is now clear, this chapter examines some features and ways of implementing the previous results. It explains and clarifies the source code, but does not give a complete and detailed overview.

9.1. LINEARALGEBRA-CLASS

LinearAlgebra is a basic class for mathematical calculations concerning vector and matrix operations. It contains a 4 by 4 array of double variables and carries extra information on the dimensions of the actual data. As far as possible, it uses overloaded operator functions. It provides features to add vectors or matrices. Multiplications of vectors or matrices with scalar values are computed with the same functions as multiplications of vectors or matrices amongst each other or vectors with matrices. These functions decide by themselves which operation to choose and verify the needed dimensions. Furthermore, it can compute the absolute value of a vector, and if the current data is a matrix, the result will be the determinant of it.

9.2. FEET-CLASS

The purpose of this class is mainly to read the sensor values and reprocess them for further use. It reads all values at a time and calculates the NPCM of each foot. It also provides information on the distribution of the load on each foot and the condition of the foot if it is either attached to the ground or lifted. Reading procedures are adjusted to the time constants of the A/D converter to provide a reliable value. In addition, a public member named “`deadbandwidth`” provides the opportunity to mask displacements under a certain range. The amplification values of the strain gauges are normalised by mean values derived from series of measurements. It might be necessary to correct these values as the strain gauges grow older.

9.3. Servo-Class

The servos are driven by public methods of this class. It comprises data on the actual position, which can be changed only by this class itself, as the data is private. Furthermore, the driving functions verify the chosen positions with the limits of the servos to guarantee, that the links do not move across mechanical boundaries. These limiters prevent the servos from overheating or illegitimate angles. Addressing the servos is done via handles, which have to be initialised before driving. All servo-handles are stored in a class-local array and the index is addressed by an integer value between zero and nine. These values are overwritten by defined semantic names similar to those mentioned in section 5.3. The definitions are global, so that they can be used in the same context in other classes. To minimize latencies of procedure calls, the driving functions await an array with new angles for all servos at a time. Otherwise, the calling procedure must handle the movement and permanent switch between movements of different joints to provide a smooth movement.
The servos can be driven relatively to their actual position by calling a method with angle-deltas or absolutes. The movement can be performed with ten different angular velocities. The different velocities are generated by adding 0-10 wait-states. Speed 10 means the maximum speed producible by the servos. However, the maximum speed also depends on the torque on the joints. Therefore, it could be reasonable to drive the servos at a lower speed, which then should be guaranteed.

9.4. DenavitHartenberg-Class

Mainly, this class embodies the results of Chapter 5. It inherits all features of LinearAlgebra. The overloaded operator '=' had to be rewritten as it is not handed down by the mother class, according to ANSI C++. The only additional public member function calculates and returns a transformation matrix. Therefore, this function is addressed with a semantic servo name, defined in Servo-class. If this value is negative, the inverted matrix from hip-base to the chosen link is calculated. Which angles are needed for the chosen operation is decided before calculating the matrix itself. A lookup table is generated on the fly, according to Table 5.4. This means, that only at most ten trigonometric computations have to be performed for each matrix calculation, instead of 39. The matrix itself then is assembled in another private method.

In addition to the increase of calculation performance by the lookup table, the standard math-lib sine and cosine functions have been replaced. Performance test on the standard functions showed that the computation of one standard sine or cosine consumes approximately 1.9ms. A lookup table for sine-calculation has been generated, which takes about 24µs. The calculation time reduces to 1.2% of the normal consumption. The test setup therefore was an EyeBot MK4 running at 33MHz, operating RoBIOS V5.1 having access to 2MB RAM. The time consumption was taken over a partially unrolled loop (to minimize loop-decision influences) with 20000 cycles.
The lookup table maps a full cycle from 0° to 360°. The test was also done with a quarter-cycle sine-mapping using recursive calls taking the advantages of symmetry. This increased the computation time to 102.5µs. The advantage using the latter table would have been the reduced memory load. As the memory is 2MB in total and the lookup table is 640 float values, it consumes 2.5kB, which is 0.12%. It was decided to use the full cycle table.
The table was externally calculated with Microsoft Excel® at a precision of six decimals. The table comprises a resolution of 640 values for 360°. The reason for this was obvious: Chapter 4 Section 2 explains that the servos operate at a range of ±72°, driven by C-functions that accept therefore a value between 0 and 255. This means a full circle of 360° is represented by $x = 256 \cdot \frac{360°}{144°} = 640$ distinguishable values which are in the natural resolution of the servos. Thereby, additional uncertainties due to angle conversions are avoided. On the other hand, this means that changes of the servo resolution in the HDT entail adaptations of this lookup table. Cosine is processed by calling sine with 90°-shifted angle.
Using a lookup table for square root calculation was also tested but this brought no improvement. Besides, the square root function is not called very often and the time consumption is not serious.

9.5. Display- and UserInput-Class

These classes provide C++-encapsulated functions to access the LCD and process key or infrared inputs. Therefore, the UserInput-class initialises the infrared and combines both, keys and remote control. It provides blocking and non-blocking functions similar to those of RoBIOS itself, and operates via an internal buffer. It has also the ability to desensitise the input, which was necessary to eliminate unwished double readings of the remote control. The sensitivity is adjustable by inserting wait-states.
The Display-class imitates `printf`. It can receive a variable number of arguments by using the Ellipsis-Operator '…'. For the implementation, see the source code in the appendix. In this way, it can also display special data structures, such as vectors or matrices, and the cursor can be positioned in the same function-call. Furthermore, it provides methods to paint graphical elements like triangles and crosses. This was needed for the intended output. If more functions are needed, the inventory of functions can easily be extended.

9.6. Control-Class

This class establishes a closed-loop PID-controller. It stores the last and second last control errors and the last manipulated value. It is simply the translation of the results of Chapter 7 in code. A difficulty in this class was in implementing the interrupt routine. The EyeBot API's interrupt routine is called with two parameters: a timescale and the start-address of the routine to be executed in the interrupt. For this address, only C-functions are accepted, because a pointer to a member function could not be resolved at compilation time. Fortunately, static member functions are deemed C-functions, as they exist only once. Static member functions can only use other static methods and variables. The solution is, to pass a pointer to static member function to the interrupt-timer attaching function. The static function calls the routine to be executed in the interrupt. This is the controller itself. An additional function is needed to synchronise data passed to the controller. This function has to ensure that the interrupt reads the correct values. Therefore, it stops the running interrupt-routine and re-attaches it after synchronising the data. The function for switching on and off is handed over to the user by two simple functions. The following pseudo-code will illustrate the implementation.

```
class ClassName       //Singleton
{    public:
      void InterruptOn();
      void InterruptOff();
      void Synchronise(…, …);
    private:
      static void  InterruptKick();
      void ToRunInInterrupt();
      static ClassName SingleInstance;
      TimerHandle InterruptHandle;
      bool interrupt_active;                    };

void ClassName::InterruptKick()
{     SingleInstance.ToRunInInterrupt();        }

void ClassName::InterruptOn()
{     if(!interrupt_active)
       InterruptHandle=OSAttachTimer(2,InterruptKick);
      if(InterruptHandle) interrupt_active=true;      }

void ClassName::InterruptOff()
{     if(interrupt_active)
       OSDetachTimer(InterruptHandle);
      interrupt_active=false;                   }

void ClassName::Synchronise(…, …)
{     bool wason=false;
      if(interrupt_active)
      {    wason=true;
           InterruptOff();}
      //synchronise values
      …
      if(wason) InterruptOn();                  }

void ToRunInInterrupt(){ …; … ;                 }
```

As the controller’s class is a singleton, no additional static member variable has to be established which can be used by the kicking function to address the worker-routine. It uses the static member `SingleInstance`.

9.7. BEHAVIOUR-CLASS

This class implements some simple exercises for the robot. It runs in the highest level of the software architecture and interfaces the user as well as lower levels. Besides simple balancing, which means standing upright, even on sloped surfaces, the robot can perform squats with or without running controller-interrupt. It also can shift its weight from one foot to another using the desired value of the controller.

This class provides simple examples for the use of the controller. It should be considered to use this class for future work implementations.

10. TESTING

To test the robots behaviour, it was placed on a plate. This plate was lifted on one side so that the subsurface was sloped. The robot was able to stand upright. The integral part of the control algorithm made the process of erecting slightly slower but on the other hand, provided more stability to the control loop. During the first tests, a P-controller by itself was used. In this setup, the robot sometimes started to oscillate. The amplitude of this oscillation increased, depending on the amplification of the control error.
After the control loop used a PID-controller, the oscillations were suppressed and the robot was stabilised. As expected, the steady-state control accuracy did not appear.
The robot sometimes produced jerks and shakes. The jerks only appeared, while the control loop was active. These effects could be aliasing caused by the sensors, due to the missing low-pass filter.

The attached CD contains video files demonstrating the robot balancing.

11. FUTURE WORK

This chapter discusses some issues, which could be improved during future work. Most of the possible improvements are explained in detail in previous sections of this thesis.

Strain Gauge Amplification:

The digital resolution of the strain gauges is very low. The A/D converter has a linear 10-bit resolution over a voltage from 0V to 4.096V. The measured voltages range from approximately 1V to 1.2V. This results in a digital resolution of $50 = 1024 \cdot \frac{1.2\text{V}-1\text{V}}{4.096\text{V}}$ when one toe is exerted to one sixth of the total gravity. The worst case, but unrealistic, would be when one toe carries the total load. In this case, the linear strain gauge would have an amplified output of $1\text{V}_{\text{Offset}} + 6 \cdot 0.2\text{V} = 2.2\text{V}$. Doubling the amplification would still guarantee a voltage reserve of nearly 700mV. The digital resolution would double.

Low Pass Filtering:

As mentioned in Chapter 7, the strain gauge amplifier is equipped with a 1.061 kHz low pass filter. Using the fastest interrupt possible (and still providing enough computation time for other tasks) requires a low pass filter frequency of 5Hz. Maintaining the current control software, this frequency should be at 2.5Hz. It should be considered to implement this to guarantee realistic data and avoid aliasing effects.

Control Algorithm:

As mentioned in 7.4., the control parameters are estimated, as they are optimised according to the method of Ziegler and Nichols. Due to time restrictions, it was not possible to model the control plant and optimise the parameters with better methods. It could also be advantageous to exchange the linear controller with a Fuzzy controller.

Servo Driving:

The current classes for interfacing with servos only provide methods to drive the servos from one point to another with a constant angular velocity. It could be necessary to implement trapezoidal or sinusoidal driving profiles to minimize jerk (or jolt $j = \dot{a} = \ddot{v} = \dddot{s}$).

Desired Value:

Now, the control software calculates the desired value of the actual feet positions and the load distribution between both. In this way, the definition of stability is enclosed in the controller totally. Depending on future applications, it could be necessary to free these values and give the desired NPCM directly to the controller. In this case, the stability lies in the responsibility of the higher-level application.

User Interface and Behaviour Layer:

Maintaining the object-oriented structure of the code should be carried out through later programming. In this way, the advantages of the classes could be sustained, especially the hardware accessibility secured by the Singletons.

12. CONCLUSION

In this thesis, I presented an active control system for a bipedal robot using force feedback. The Denavit-Hartenberg-Notation was successfully adapted to the existing structure and provided the possibilities to calculate the kinematics. The sensory data was processed to provide useful information on the robots state. Together with the kinematics, the pre-processed data was used to establish a closed loop control on the stability. This control was used for balancing the robot.
The software was developed object orientated in a layered architecture, according to software engineering principles. The project can be extended to more complex tasks such as walking, using the developed control to stabilise it.

Finally, the robot was able to balance on a moving surface.

To the best of my knowledge, this is the first implementation of a balance control in a fully autonomous biped robot in this setup using only force sensors.

The control algorithm developed is only for kinematics. The controlling of dynamics would require more sensors and at least the same amount of computation time. As tests showed, the robot slows down its movements nearly to the half of its velocity when the controlling algorithm is switched on. In addition, the actual combination of the sensor-filters and the A/D-converter does not match the requirements, stated in Chapters 7.1 and 7.2.

REFERENCES

[1] **Honda Motor Co., Ltd.**: *Honda Robot Top Page*,
Website: http://world.honda.com/robot/,
(Avail. by 10.03.2003)

[2] **Bräunl, T.**: *Hardware Description*,
Website: http://robotics.ee.uwa.edu.au/eyebot/hardware.html
(Avail. by 10.03.2003)

[3] **Bräunl, T.; Schmitt, K.; Lampart, T.; Reinholdtsen, P.; Kapp, M.**: *Library of function build into RoBIOS*,
Website: http://robotics.ee.uwa.edu.au/eyebot/ftp/ROBIOS/docs/library.html
(Avail. by 10.03.2003)

[4] **HITEC RCD USA, Inc.**: *Announced Specification of HS-945MG Standard Coreless Motor High Torque Servo*,
Document: http://www.hitecrcd.com/Servos/hs945.pdf
(Avail. by 10.03.2003)

[5] **Kraft, D.**: *Modellierung, Simulation und Optimierung von Mehrkörpersystemen*,
Fachhochschule München, Fachbereich Maschinenbau,
Document: http://www.fm.fh-muenchen.de/~dkraft/ps_files/rob.ps
(Avail. by 10.03.2003)

[6] **Kun, A.L.**: *A Sensory-Based Adaptive Control Algorithm for Variable Speed Biped Robots*, Dissertation, University of New Hampshire, 1997, (available: at http://wwwlib.umi.com/dissertations/ under Ref.: AAT 9730831)

[7] **Schrüfer, E.**: *Elektrische Meßtechnik*, München: Hanser-Verlag, 1990

[8] **Hußmann, H.**: *Softwaretechnologie II*, Technische Universität Dresden, Wintersemester 2000/2001, Scriptum zur Vorlesung,
Website: http://www-st.inf.tu-dresden.de/Lehre/WS00-01/st2/vorl.html
(Avail. by 10.03.2003)

[9] **Lutz, H.; Wendt, W.**: *Taschenbuch der Regelungstechnik*, 4. korrigierte Auflage, Verlag Harri Deutsch, Frankfurt am Main 2002

[10] **Maxim Integrated Products, Inc.**: *Data Sheet MAX192*,
Document: http://pdfserv.maxim-ic.com/arpdf/MAX192.pdf
(Avail. by 26.03.2003)

[11] **Analog Devices, Inc.**: *Data Sheet AMP04*, Document:
http://www.analog.com/UploadedFiles/Datasheets/421758465AMP04_b.pdf
(Avail. by 26.03.2003)

[12] **Shadow Robot Company Ltd.**: *Shadow Robot Company Home Page*, Website:
http://www.shadow.org.uk/
(Avail. by 03.04.2003)

[13] **Encyclopaedia Britannica**: *Encyclopaedia Britannica*
Website: http://www.britannica.com/
(Avail. by 03.04.2003)

[14] **Marshall, I.J.**: *Active Balance Control for a Humanoid Robot*, School of Information Technology and Electrical Engineering, University of Queensland, Dorrington 2002, Queensland, Australia
Document: http://innovexpo.itee.uq.edu.au/2002/projects/s358239/thesis.pdf
(Avail. by 03.04.2003)

FIGURE SOURCES

Figure 2.1:	Johnny Walker Photo from Antonio Pickel, taken in the Mobile Robot Lab of CIIPS, Department of Electrical, Electronic and Computer Engineering, University of Western Australia, Perth 2003
Figure 2.2:	Jack Daniels (See Figure 2.1)
Figure 2.3:	Rock Steady (See Figure 2.1)
Figure 2.4:	ASIMO Source: Honda Motor Co., Ltd., Website: http://world.honda.com/robot/
Figure 2.5:	Shadow Robot Shadow Robot Company Ltd., Website: http://www.shadow.org.uk/
Figure 4.1:	EyeBot Controller Photo from Christoph Braunschädel, taken in the Mobile Robot Lab of CIIPS, Department of Electrical, Electronic and Computer Engineering, University of Western Australia, Perth 2003
Figure 4.2:	Andy Droid (front view) (See Figure 2.1)
Figure 4.3:	Andy Droid’s left foot (See Figure 2.1)
Figure 4.4:	Andy Droid (side view) (See Figure 2.1)

APPENDIX A – DOCUMENTATION OF DEVELOPED CODE

This Appendix is a code reference for all public variables and methods developed for balance control and related classes.

LINEARALGEBRA-CLASS

This class provides mathematic functions for matrix and vector operations. It has `class DenavitHartenberg` as a friend to allow this class access its private members.

```
LinearAlgebra();
```

Input:	none
Output:	none as it is a constructor
Semantics:	Standard constructor is automatically called for the creation of an instance of this class.

```
LinearAlgebra(double* data, int wrows=4, int wcolumns=4);
```

Input:	(`data`) array containing data the instance should be created with, matrix values must carry `{row1; row2;…}` (`wrows`) number of rows of matrix to be created with `data`, if omitted `wrows`=4, valid values 1-4 (`wcolumns`) number of columns of matrix to be created with `data`, if omitted `wcolumns` = 4, valid values 1-4
Output:	none as it is a constructor
Semantics:	Constructor is automatically called for the creation of an instance of this class when initialisation data is passed.

```
LinearAlgebra(const LinearAlgebra& tocopy);
```

Input:	(`tocopy`) instance, which has to be copied to created object
Output:	none as it is a copy-constructor
Semantics:	Copy-constructor is automatically called for the creation of an instance of this class when initialisation is done by operator =. All member data will be copied.

```
LinearAlgebra  operator * (double scalar);
```

Input:	(`scalar`) factor, which is used during operation
Output:	result of the operation
Semantics:	Overloaded operator * calculates $\vec{v} \cdot \text{scalar}$ if data in instance is a vector or $V \cdot \text{scalar}$ if data in instance is a matrix. The result of operation is returned in an object of this class. The calling instance stays unmodified.

```
LinearAlgebra  operator *=(double scalar);
```

Input:	(`scalar`) factor, which is used during operation
Output:	result of the operation
Semantics:	Overloaded operator * calculates $\vec{v} \cdot \text{scalar}$ if data in instance is a vector or $V \cdot \text{scalar}$ if data in instance is a matrix. The result of operation is returned in an object of this class and in the calling instance.

```
LinearAlgebra  operator * (const  LinearAlgebra& lag);
```

Input: (`lag`) second operand, which is used during operation
Output: result of the operation
Semantics: Overloaded operator * calculates $\vec{v}_{call} \cdot \vec{v}_{lag}$ (scalar product) if data in calling and passed instance are vectors (result in element (`0,0`)), $V_{call} \cdot V_{lag}$ if data in calling and passed instance is a matrix, or $V_{lag} \cdot \vec{v}_{call}$ if data in calling instance is a matrix and data in passed instance is a vector.. The result of operation is returned in an object of this class. Note: if calling instance is a vector and the data in passed instance a vector, or if the dimensions of the matrices do not match the multiplication requirements nothing is processed.

```
LinearAlgebra  operator *=(const  LinearAlgebra& lag);
```

Input: (`lag`) second operand, which is used during operation
Output: result of the operation
Semantics: Overloaded operator *= calculates $\vec{v}_{call} \cdot \vec{v}_{lag}$ (scalar product) if data in calling and passed instance are vectors (result in element (`0,0`)), $V_{call} \cdot V_{lag}$ if data in calling and passed instance is a matrix, or $V_{lag} \cdot \vec{v}_{call}$ if data in calling instance is a matrix and data in passed instance is a vector. The result of operation is stored in the calling instance and returned in an object of this class. Note: if calling instance is a vector and the data in passed instance a vector, or if the dimensions of the matrices do not match the multiplication requirements, nothing is processed.

```
LinearAlgebra  operator + (const  LinearAlgebra& lag);
```

Input: (`lag`) second operand, which is used during operation
Output: result of the operation
Semantics: Overloaded operator + calculates $\vec{v}_{call} + \vec{v}_{lag}$ if data in calling and passed instance are vectors, $V_{call} + V_{lag}$ if data in calling and passed instance is a matrix. The result of the operation is returned in an object of this class. Note: if the dimensions of the matrices/vectors do not match the addition requirements, nothing is processed.

```
LinearAlgebra  operator +=(const  LinearAlgebra& lag);
```

Input: (`lag`) second operand, which is used during operation
Output: result of the operation
Semantics: Overloaded operator += calculates $\vec{v}_{call} + \vec{v}_{lag}$ if data in calling and passed instance are vectors, $V_{call} + V_{lag}$ if data in calling and passed instance is a matrix. The result of the operation is stored in the calling instance and returned in an object of this class. Note: if the dimensions of the matrices/vectors do not match the addition requirements, nothing is processed.

```
LinearAlgebra  operator - (const  LinearAlgebra& lag);
```

Input: (`lag`) second operand, which is used during operation
Output: result of the operation
Semantics: Overloaded operator - calculates $\vec{v}_{call} - \vec{v}_{lag}$ if data in calling and passed instance are vectors, $V_{call} - V_{lag}$ if data in calling and passed instance is a matrix. The result of the operation is returned in an object of this class. Note: if the dimensions of the matrices/vectors do not match the subtraction requirements, nothing is processed.

```
LinearAlgebra  operator -=(const  LinearAlgebra& lag);
```

Input: (`lag`) second operand, which is used during operation
Output: result of the operation
Semantics: Overloaded operator -= calculates $\vec{v}_{call} - \vec{v}_{lag}$ if data in calling and passed instance are vectors, $V_{call} - V_{lag}$ if data in calling and passed instance is a matrix. The result of the operation is stored in the calling instance and is returned in an object of this class. Note: if the dimensions of the matrices/vectors do not match the subtraction requirements, nothing is processed.

```
LinearAlgebra Equals(double* data, int wrows=4, int wcolumns=4);
```

Input: (`data`) second operand as an array, which is used during operation
(`wrows`) number of rows in data, valid values 1-4
(`wcolumns`) number of columns in data, valid values 1-4
Output: result of the assignment
Semantics: Copies the array `data` in the calling instance's matrix using the assigned `wrows` and `wcolumns`. The result is returned. Note: if `wrows` and `wcolumns` are omitted, they are `4`.

```
LinearAlgebra Plus(double* data, int wrows=4, int wcolumns=4);
```

Input: (`data`) second operand as an array, which is used during operation
(`wrows`) number of rows in data, valid values 1-4
(`wcolumns`) number of columns in data, valid values 1-4
Output: result of the operation
Semantics: Calculates $\vec{v}_{call} + \vec{v}_{lag}$ if data in calling instance and passed data are vectors, $V_{call} + V_{lag}$ if data in calling instance and passed data is a matrix. The result of the operation is returned in an object of this class. Note: if the dimensions of the matrices/vectors do not match the addition requirements, nothing is processed and if `wrows` and `wcolumns` are omitted, they are `4`.

```
void GetData(double* data, int wrows=4,    int wcolumn=4,
                           int firstrow=1, int firstcolumn=1);
```

Input: (`data`) second buffer where desired data will be copied
(`wrows`) number of rows to copy to data, valid values 1-4
(`wcolumns`) number of columns to copy to data, valid values 1-4
(`firstrow`) row, where to start copying, valid values 1-3
(`firstcolumn`) column, where to start copying, valid values 1-3
Output: none
Semantics: Extracts the data contained in the calling instance to assigned buffer `data`. Note: if `wrows` and `wcolumns` are omitted, they are `4` and if `firstrow` and `firstcolumn` are omitted, they are 1.

```
double Absolute();
```

Input: none
Output: result of operation
Semantics: Calculates $\left|\vec{v}_{call}\right|$ (absolute) if calling instance is a vector or $\det\left(V_{call}\right)$ if calling instance is a square matrix. Otherwise, nothing is processed.

`double Det();`

Input: none
Output: result of operation
Semantics: Calculates $\det\left(V_{call}\right)$ if calling instance is a square matrix. Otherwise, nothing is processed.

`bool IsMatrix();`

Input: none
Output: `true` if calling instance is a matrix, otherwise `false`
Semantics: Determines, wether the calling instance is a matrix or not.

`bool IsVector();`

Input: none
Output: `true` if calling instance is a vector, otherwise `false`
Semantics: Determines, wether the calling instance is a vector or not.

`int GetRows();`

Input: none
Output: number of rows in calling instance
Semantics: Returns number of rows in calling instance.

`int GetColumns();`

Input: none
Output: number of columns in calling instance
Semantics: Returns number of columns in calling instance.

FEET-CLASS

This class provides functions to read and pre-process the sensory data of *Andy Droid*. This class is a Singleton-Class.

`static Feet* GetTheInstance();`

Input: none
Output: pointer to Singleton-Instance
Semantics: This function returns a pointer to the Singleton Instance of this class. As this member method is not callable with an existing instance, use the Scope-operator during creation of the pointer:
`Feet* singleton_pointer=Feet::GetTheInstance()`
and access other member functions by dereferencing it:
`singleton_pointer ->Function()`.

`bool Init();`

Input: none
Output: true if initialisation succeeded, false if no success
Semantics: This function initialises members and the sensor. Especially, it reads the sensor values in their unstressed position. Therefore, make sure that the robot is lifted while calling this function. This function must have been called before using this class.

```
LinearAlgebra GetCOG( LinearAlgebra& buf,
                      int left=1,bool refresh=false);
```

Input: (`buf`) buffer, where result is stored in
(`left`) value to determine the foot's centre of gravity to return, 1=left, 2=right
(`refresh`) determines wether the sensors should be read or last readings should be used

Output: returns the cog assigned by passed values, this object carries the same data as buf

Semantics: The function returns the calculated centre of force (gravity) on the assigned foot. If refresh is true, the sensors will be read again, otherwise the last value is returned. The position of the COG are given in the foot's coordinate system.

```
bool GetCOG(double* buf);
```

Input: (`buf`) buffer, where result is stored in

Output: true, if success, otherwise false

Semantics: The function returns the calculated centres of force (gravity) of both feet. `buf[0]=left_z`, `buf[1]=left_y`, `buf[2]=right_z`, `buf[3]=right_y`. The sensors will be read again and the values will be calculated new.

```
bool LoadDistribution(double *left,double *right,double *total,
                      bool refresh=true);
```

Input: (`*left`) pointer for left-foot's value
(`*right`) pointer for right-foot's value
(`*total`) pointer for total load's value
(`refresh`) determines wether the sensors should be read or last readings should be used

Output: true if success, otherwise false

Semantics: This function returns the sum of the normalised sensor values of each foot and in total in the assigned buffers. They represent the total load (force) on each foot.

```
int Lifted(bool refresh=false);
```

Input: (`refresh`) determines wether the sensors should be read or last readings should be used

Output: information which foot is lifted

Semantics: This function returns a number, which determines the lifted foot: 0 = no foot lifted, 1 = left foot lifted, 2 = right foot lifted, 3 = both feet lifted. The threshold above which a lift is recognised is given in the private member `liftsensitivity`.

```
int deadbandwidth;
```

Public member, which determines a threshold, below which no changes in read sensor values are considered.

```
int readings;
```

Public member, which determines a number of loop readings for each sensor. This value handles the validity time of the A/D-converter.

SERVO-CLASS

This class provides functions to initialise and drive the servos. This class is a Singleton-class. For driving, an array is used, to pass absolute or relative positions. The array has to be of 10 integer values. However, it can be of the size NUMBEROFSERVOS, as this is defined in the class' header. The array-elements can be addressed by the following defines, which are specified in the class' header file:

```
NUMBEROFSERVOS
```

```
RHIPS, LHIPS, RHIPB, LHIPB, RKNEE, LKNEE, RANKB, LANKB, RANKS, LANKS
```

For addressing links and joints in DenavitHartenberg-Class, the following definitions are made in the Servo class Header-file:

```
BODY, RTHIGH, LTHIGH, RSHANK, LSHANK, RFOOT, LFOOT
```

Public methods:

```
static Servo* GetTheInstance();
```

Input:	none
Output:	pointer to Singleton-Instance
Semantics:	This function returns a pointer to the Singleton Instance of this class. As this member method is not callable with an existing instance, use the Scope-operator during creation of the pointer: `Servo* singleton_pointer=Servo::GetTheInstance()` and access other member functions by dereferencing it: `singleton_pointer ->Function()`.

```
void Init();
```

Input:	none
Output:	none
Semantics:	This function initialises the servos and members. This function must be called before using other methods of this class.

```
void Release();
```

Input:	none
Output:	none
Semantics:	This function releases the servos. This function must be called after using them. After calling this function, the servos could not be used until the next call of `Init()`.

`int Set(int* position, int speed);`

Input: (`*position`) pointer to an array (10 integers) that contains the new absolute positions for the servos
(`speed`) speed for driving the servos to the new positions. Valid values 1(slow) to 10(fast)
Output: the function returns the time consumed by this function-call in µs.
Semantics: This function drives the servos to the absolute positions passed to the function in the array. The function controls the absolute limits to protect the hardware. If not all servos should be driven, the array should carry the actual position of this servo.

`void Get(int* position);`

Input: (`*position`) pointer to an buffer array (10 integers), in which the actual servo positions will be stored
Output: none
Semantics: This function returns the actual servo position in the assigned buffer.

`int GetLimit(int position);`

Input: (`position`) number of servo, which limit should be returned, negative servo-number returns lower limit, positive upper.
Output: absolute servo-limit
Semantics: This function returns the limit of the chosen servo as an absolute servo-angle. If the passed servo number is negative, the lower limit is returned, otherwise the upper limit.

`int Move(int* delta, int speed);`

Input: (`*delta`) pointer to an buffer array (10 integers), in which the desired servo delta-positions are passed
(`speed`) desired speed for movement
Output: the function returns the time consumed by this function-call in µs.
Semantics: This function drives the servos relative by from the actual positions. The deltas for all servos are passed in an array. The function controls the absolute limits to protect the hardware. If not all servos should be driven, the array entries should be zero for these servos.

DENAVITHARTENBERG-CLASS

This class provides functions for the calculation of coordinate transformation matrices for Andy Droid, according to Denavit Hartenberg. This class inherits from the `LinearAlgebra` class and knows all public member functions of its mother class.
For addressing links and joints in DenavitHartenberg-Class, the following definitions are made in the Servo class Header-file:

```
BODY, RTHIGH, LTHIGH, RSHANK, LSHANK, RFOOT, LFOOT
```

```
Public methods:
```

```
DenavitHartenberg();
```

Input:	none
Output:	none as it is a constructor
Semantics:	Standard constructor is automatically called for the creation of an instance of this class.

```
DenavitHartenberg(double *matrix, const int * const servopos,
                  bool docalc=true, bool left=true);
```

Input:	(*`matrix`) pointer to an array in which the data calculated by the recently created instance can be returned (*`servopos`) pointer to an array carrying servo positions, the class will normally request it from the servo class (`docalc`) determines, if the matrix should be calculated new or the last one calculated should be passed back in `matrix`, if omitted, `docalc` = `true` `(left)` latch, to determine the leg, valid values are `true` = left and `false` = right, if omitted `left` = true
Output:	none as it is a constructor
Semantics:	Constructor is automatically called for the creation of an instance of this class when initialisation data is passed.

```
DenavitHartenberg& operator = (const LinearAlgebra& tocopy);
```

Input:	(`tocopy`) second operand as a `LinearAlgebra` object, which is used
Output:	result of the assignment as an object of `DenavitHartenberg` class
Semantics:	Copies the `LinearAlgebra` data in the calling instance's matrix

`void GetDVMatrix(double *matrix, int whichservo=9);`

Input: (`*matrix`) pointer to an array, in which the matrix will be returned
(`whichservo`) determines which matrix should be calculated

Output: none

Semantics: This function returns a Denavit Hartenberg matrix. The matrix is a 4x4 frame in homogeneous coordinates. The variable `whichservo` determines the matrix. The table shows the matrices:

`whichservo`	Result	Transformation
`LFOOT`	${}^{0}_{5}T_{left}$	Left Foot to Hip Centre
`RFOOT`	${}^{0}_{5}T_{right}$	Right Foot to Hip Centre
`-LFOOT`	${}^{5}_{0}T_{left}$	Hip Centre to Left Foot to
`-RFOOT`	${}^{5}_{0}T_{right}$	Hip Centre to Right Foot
`LSHANK`	${}^{0}_{2}T_{left}$	Left Shank to Hip Centre
`RSHANK`	${}^{0}_{2}T_{right}$	Right Shank to Hip Centre
`-LSHANK`	${}^{2}_{0}T_{left}$	Hip Centre to Left Shank
`-RSHANK`	${}^{2}_{0}T_{right}$	Hip Centre to Right Shank
`LTHIGH`	${}^{0}_{3}T_{left}$	Left Thigh to Hip Centre
`RTHIGH`	${}^{0}_{3}T_{right}$	Right Thigh to Hip Centre
`-LTHIGH`	${}^{3}_{0}T_{left}$	Hip Centre to Left Thigh
`-RTHIGH`	${}^{3}_{0}T_{right}$	Hip Centre to Right Thigh

`DenavitHartenberg& Calc(int whichservo=9);`

Input: (`whichservo`) determines which matrix should be calculated

Output: `DenavitHartenberg` object carrying the result

Semantics: This function returns a Denavit Hartenberg object. The object carries a 4x4 frame in homogeneous coordinates. The variable `whichservo` determines the matrix to be calculated. The table above shows the matrices.

DISPLAY-CLASS

This class provides functions to access the display and to print and paint information for the user. This class is a Singleton-Class.

```
static Display* GetTheInstance();
```

Input: none
Output: pointer to Singleton-Instance
Semantics: This function returns a pointer to the Singleton Instance of this class. As this member method is not callable with an existing instance, use the Scope-operator during creation of the pointer:

```
Display* singleton_pointer
                  =Display::GetTheInstance()
```

and access other member functions by dereferencing it:

```
singleton_pointer ->Function().
```

```
bool Init();
```

Input: none
Output: true if initialisation succeeded, false if no success
Semantics: This function initialises members and the display. This function must have been called before using this class.

```
void Clear();
```

Input: none
Output: none
Semantics: This function initialises members and the sensor. Especially, it reads the sensor values in their unstressed position. Therefore, make sure that the robot is lifted while calling this function. This function must have been called before using this class.

```
int Print(const char format[], ...);
```

Input: (`format`) formatted string
(...) variable number of variables as done in `printf( )`
Output: 0 if failed, other when successful
Semantics: This function can be used as `printf( )`. The ellipsis operator deals with the variable number of variables.

```
int Print(int x,int y,const char format[], ...);
```

Input: (`x`) x-position for string
(`y`) y-position for string
(`format`) formatted string
(...) variable number of variables as done in `printf( )`
Output: 0 if failed, other when successful
Semantics: This function positions the cursor to assigned position before printing to display. Furthermore, this function can be used as `printf( )`. The ellipsis operator deals with the variable number of variables.

```
int Print(LinearAlgebra toprint);
```

Input: (`toprint`) LinearAlgebra object to print in display
Output: 0 if failed, other when successful
Semantics: This function display the matrix elements of a LinearAlgebra object in the display.

```
double PaintExtensions(int leftx =CENTREXL,int lefty=CENTREY,
                       int rightx=CENTREXR,int righty=CENTREY);
```

Input:	(`leftx`) x-position for left foot's boundaries (`lefty`) y-position for left foot's boundaries (`rightx`) x-position for right foot's boundaries (`righty`) y-position for right foot's boundaries
Output:	0 if failed, other when successful
Semantics:	This function paints two triangles representing the feet's boundaries. Both could be positioned by feet coordinate positions. The function zooms automatically to fit the triangles to the display.

```
double PaintExtensions(LinearAlgebra left,LinearAlgebra right);
```

Input:	(`left`) position for left foot's boundaries (`right`) position for right foot's boundaries
Output:	0 if failed, otherwise the resize-factor (zoom) is returned
Semantics:	This function paints two triangles representing the feet's boundaries. Both could be positioned by feet coordinate positions. The function zooms automatically to fit the triangles to the display.

```
void PaintCross (float *positionnew, int identity,
                 double resizefactor);
```

Input:	(`*positionnew`) pointer to buffer carrying position for cross (`z,y`) (`identity`) unique identity for cross, valid values are 1 to 10 (`resizefactor`) zoom factor
Output:	none
Semantics:	This function paints a cross on the assigned position. If identity is even, it is a '`+`', if identity is odd it will become a '`x`'. The `resisefactor` is needed if the currently displayed screen is already zoomed to recalculate the positions.

```
void PaintCross (float *positionnew, int identity);
```

Input:	(`*positionnew`) pointer to buffer carrying position for cross (`z,y`) (`identity`) unique identity for cross, valid values are 1 to 10
Output:	none
Semantics:	This function does the same as the method above, except `resizefactor`.

```
void PaintCross (LinearAlgebra vector,int identity,
                 double resizefactor);
```

Input:	(`vector`) position for cross (`z,y`) (`identity`) unique identity for cross, valid values are 1 to 10 (`resizefactor`) zoom factor
Output:	none
Semantics:	This function paints a cross on the assigned position. If identity is even, it is a '`+`', if identity is odd it will become a '`x`'. The `resisefactor` is needed if the currently displayed screen is already zoomed to recalculate the positions.

```
void PaintCross (LinearAlgebra vector,int identity);
```

Input:	(`vector`) position for cross (`z,y`) (`identity`) unique identity for cross, valid values are 1 to 10
Output:	none
Semantics:	This function does the same as the method above, except `resizefactor`.

```
void PaintLoadBalance(double left,double right,int total);
```

Input:	(`left`) percentage of load on left foot (`right`) percentage of load on right foot (`total`) sum of sensor values read
Output:	none
Semantics:	This function paints a horizontal bar on the top of the display, its width represents the total load and its position represents the distribution between both feet.

```
int Menu(  const char str1[],const char str2[],
           const char str3[],const char str4[]);
```

Input:	(`str1`) string for menu entry 1 (`str2`) string for menu entry 2 (`str3`) string for menu entry 3 (`str4`) string for menu entry 4
Output:	0 if no success, otherwise non-zero
Semantics:	This function prints the assigned strings on the bottom of the display above the keys.

USERINPUT-CLASS

This class provides functions to read the keys and the IR remote control. This class is a Singleton-Class.

```
static UserInput* GetTheInstance();
```

Input:	none
Output:	pointer to Singleton-Instance

Semantics:	This function returns a pointer to the Singleton Instance of this class. As this member method is not callable with an existing instance, use the Scope-operator during creation of the pointer:
	`UserInput*singleton_pointer`
	`= UserInput::GetTheInstance()`
	and access other member functions by dereferencing it:
	`singleton_pointer ->Function()`.

```
int Init();
```

Input:	none
Output:	not zero if initialisation succeeded, 0 if no success
Semantics:	This function initialises members and the display. This function must have been called before using this class.

```
int Read();
```

Input:	none
Output:	key number if key was pressed, otherwise 0
Semantics:	This function reads from the keys and from the IR remote control. This function is not waiting, if no key was pressed, it returns 0 immediately.

`int Wait();`

Input:	none
Output:	key number
Semantics:	This function waits until a key on the controller or on the IRRC is pressed. The key's number is returned.

`int Wait(int specified_key);`

Input:	(`specified_key`) number of key to wait for
Output:	key number
Semantics:	This function waits until the specified key on the controller or on the IRRC is pressed. The key's number is returned.

`int WaitTime(int time=-1);`

Input:	(`time`) wait-time-out = (`time`*10ms)
Output:	key number
Semantics:	This function waits until a key on the controller or on the IRRC is pressed or the time-out is elapsed. The key's number is returned if a key was pressed. If time elapses, the function returns 0. If time is omitted or `time`=-1 is passed to the function, it waits infinite for a key to be pressed.

`int Flush();`

Input:	none
Output:	none
Semantics:	This function empties the class' internal buffer for pressed keys.

BEHAVIOUR-CLASS

This class contains some examples of the use of the developed controlling algorithm.

`Behaviour();`

Input:	none
Output:	none as it is a constructor
Semantics:	This function is a constructor does nothing, as there exist no members.

`void Init();`

Input:	none
Output:	none
Semantics:	This function initialises all lower level classes. It must be called before using other member methods of this class.

`void Release();`

Input:	none
Output:	none
Semantics:	This function releases all lower level classes. It should be called before exiting `main()`.

`void Run();`

Input:	none
Output:	none
Semantics:	This function runs the main-application.

APPENDIX B – USED SOFTWARE

The software was developed on a PC with a Microsoft Windows XP® operating system. The code was edited in UltraEdit-32® and compiled with a modified GNU C/C++-cross-compiler. The matrix 5.12, presented in chapter 5, was verified with MathCAD Plus 6.0©. Microsoft Excel® was used to calculate the look-up tables for sine and square root. This document and all drafts and diagrams contained were created with Microsoft Word®. The videos were taken with the digital camera of the Mobile Robot Lab and composed and rendered with Movie Xone 4.0®.

DECLARATION

I hereby declare that this submission is my own work and that I only used the referenced aids.

Perth, 28.04.2003

_(signed)________________________

Jochen Zimmermann