Motion control of an omnidirectional mobile robot

T.A. Baede

DCT 2006.084

Traineeship report

Supervisors: Dr. M. H. Ang Jr. [1]
Prof. dr. ir. M. Steinbuch [2]

[1] National University of Singapore
Faculty of Engineering
Department of Mechanical Engineering
Control and Mechatronics Group

[2] Eindhoven University of Technology
Department of Mechanical Engineering
Dynamics and Control Technology Group

Eindhoven, September 18th, 2006

Table of contents

Introduction

In the field of robotics, the importance of mobile robots is steadily increasing. Due to their freedom of movement, mobile robots are more flexible and can perform more tasks than their conventional fixed counterparts.

Current applications [1] of mobile robots are broad and include domestic and public cleaning, transport of goods in hospitals, factories, ports and warehouses, exploration of inhospitable terrains such as space or oceans, mining, defusing explosives, entertainment and performing inspections and security patrols.

A special class of mobile robots are omnidirectional robots. These robots are designed for 2D planar motion and are capable of translation (x,y) and rotation around their center of gravity (θ) : three degrees of freedom (3 DOF). Unlike conventional vehicles, omnidirectional robots can control each of their DOFs independently.

In order to operate effectively, mobile robots should be able to keep track of their current position (localization), sense their surroundings (perception), be able to generate a path to their destination (path planning) and execute it (navigation) in an efficient manner. To a large extend, this is accomplished through sensing and smart algorithms.

To develop, implement and test new algorithms and sensing techniques, an omnidirectional mobile platform was built at the National University of Singapore (NUS). This test bed was designed to be very flexible, easy to program and to use commodity hardware. However, apart from the assembled hardware, the robot was not yet ready to be used because of a fixed power and programming interface and the lack of motion control.

In this project, the following goals have been set:

- Implement planar motion control algorithms and analyze performance in time domain.
- Adapt the robot for wireless control.

This report will start with an explanation of the concept and advantages of omnidirectional mobility. Then, a brief introduction to the omnidirectional mobile platform will be given. Thereafter, the robot kinematics are analyzed and a motion control design is developed. Experiments will be performed and results presented. Finally, a number of non-linear, performance-limiting effects will be discussed along with possible resolution strategies.

This is the final report of the traineeship project "Motion control of an omnidirectional mobile robot". The 12 weeks traineeship was conducted at the Control and Mechatronics Laboratory of the Mechanical Engineering department at the National University of Singapore. It was supervised by Dr. M. H. Ang Jr. of NUS and Prof. dr. ir. M. Steinbuch of TU/e.

Chapter 1: Omnidirectional mobility

In the introduction of this report, it was mentioned that the mobile robot, of which the movements are to be controlled, is capable of omnidirectional mobility. To gain a better understanding of this concept, it will now be discussed.

Robotic vehicles are often designed for planar motion. Some examples include floor cleaning or transport of goods in warehouses. In such a two-dimensional (2D) space, a body has three degrees-of-freedom (3 DOF). It can translate along the *x* and *y* axes and it can rotate around its center-of-gravity, the *θ* axis (see figure 1).

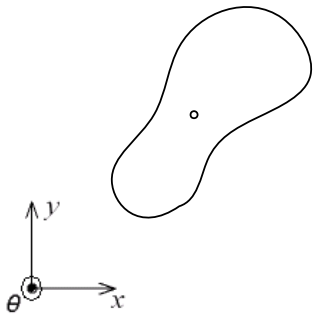

Figure 1: 3 DOF: *x*, *y*, *θ*

Most vehicles are not capable of controlling these three degrees-of-freedom independently, because of so called non-holonomic constraints. As example, consider a road where several cars are parked along the side of the road. If a driver wants to park his normal passenger car in an open space between two cars, he can't simply move sideways. The driver often has to drive forward and backward several times to make enough of an angle to insert his car into the free spot and to get a final orientation that is satisfactory (see figure 2). This it due to the inability of a car using skid-steering to move perpendicular to its drive direction: a non-holonomic constraint. While generally such a vehicle can reach every location and orientation in a 2D space, it may require complicated maneuvers and complex path planning to do so, regardless of whether it is a human or robot-controlled vehicle.

By contrast, a vehicle that is not hampered by these constraints is capable of *omnidirectional mobility*. It can travel in any direction under any orientation. In many cases where mobile robots are put into action, especially in confined or congested spaces, omnidirectional mobility is highly advantageous. It decreases control system complexity and enables faster and more accurate movement.

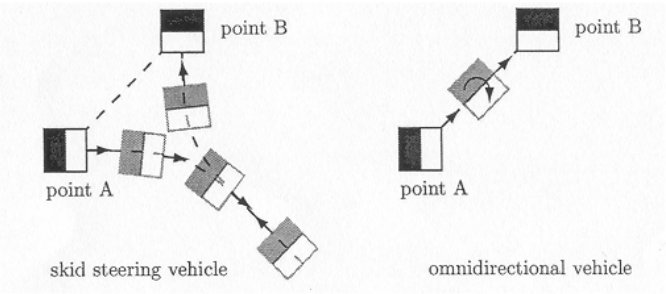

Figure 2: Moving from A to B: skid-steering (left) and omnidirectional movement (right)

Chapter 2: The omnidirectional mobile platform

In this chapter, the omnidirectional mobile platform or *robot* will be described in more detail.

2.1: Purpose and usage

As mentioned in the introduction of this report, the purpose of the robot is to function as a flexible test bed primarily for the development, implementation and testing of localization, trajectory planning and tracking algorithms.

These high-level algorithms are normally written in the C programming language.
The researcher would implement his or her algorithms on the robot's onboard computer, perform experiments and validate performance, preferably from a control station without having to physically touch the robot. A joystick at the control station could be used for basic guidance.

To this end, the robot has to be modified and a control station must be set up. The robot's original design and any modifications will be described in the next paragraph.

High-level algorithms are not enough to set the omnidirectional platform in motion. In order to function properly, the robot also needs a library of low-level algorithms that can send motion control signals to the motors, poll sensors and provide other important functionality. In the following chapters, these control algorithms will be addressed.

2.2: Robot subsystems

The robot was designed for human environments and this is reflected in the design, most notably its dimensions and dynamics. It consists of three major parts: a frame, drivetrain and electronics. A fourth part that plays a vital role is the control station and software. Below, all parts will be briefly discussed. For detailed information on the robot's design, see Rob van Haendel's report [2]. Photographs of the robot are shown in figure 3.

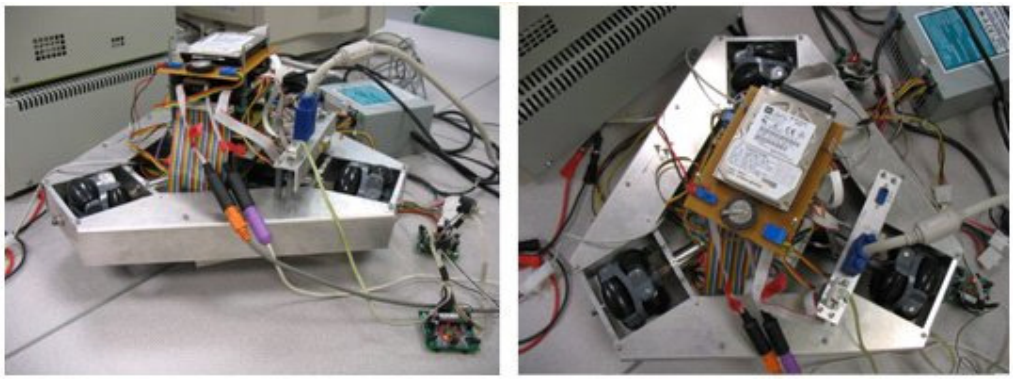

Figure 3: Photographs of the robot: side-view (left) and top-view (right)

2.2.1: Frame

In figure 4, a schematic overview of the robot is presented. The brown section in the overview is the lightweight aluminium frame. The purpose of the frame is to provide a stiff support for all the robot's components, to provide space for the research payload and to protect the delicate systems onboard, such as the electronics and wheels, from damage during possible collisions with objects.

The total mass of the frame is 1.4 kg whereas the mass of the fully equipped robot at present is 3.6 kg. It can carry a research payload of 5 kg on top of the base. The frame has the shape of a truncated equilateral triangle with long edges measuring 310 mm and the short ones 110 mm. The robot's footprint fits in a rectangle with length 420 mm and width 364 mm. The height depends on the payload, but normally does not exceed 350 mm to be able to move freely in a laboratory environment and under tables and workbenches.

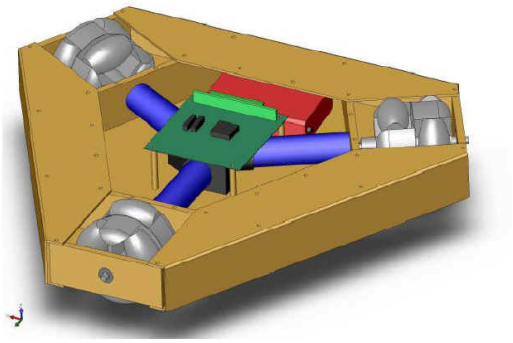

Figure 4: Schematic overview of robot

2.2.2: Drivetrain

The grey parts in the schematic overview are the omnidirectional wheels which provide omnidirectional mobility. In figure 5, a close-up of such a wheel is shown. The commercially available *Omniwheel* has six passive rollers on the periphery of the wheel, a trio on each side. The shafts of the rollers are perpendicular to the shaft of the wheel. An omniwheel is driven in a normal fashion, while the rollers allow for a free motion in the perpendicular direction (sideways).

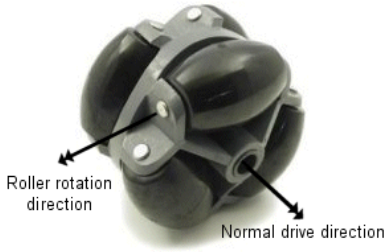

Figure 5: Omniwheel

In the frame, three 40 mm radius omniwheels are mounted at 120° intervals along the robot's center-of-gravity to create a statically determined structure. Due to this arrangement, the wheels are always in contact with the floor regardless of its roughness.

Each wheel is driven by a DC micromotor, Faulhaber series 2642 012 CR, depicted with a blue color in the schematic. The motor's maximum rotational velocity is 6000 rpm. The DC motors convert input currents to torques.

A Faulhaber 26/1 gearbox with 14:1 reduction ratio is mounted between wheel and motor shaft to increase the torque and decrease the maximum rotational velocity at the wheels to acceptable levels. The robot was designed for good mobility in a human environment: a top speed of 1 m/s, acceleration of 1 m/s^2, rotational velocity of 1 rad/s and rotational acceleration of 1 rad/s^2. Although these values have not been explicitly investigated, it was shown that the robot is able to easily surpass these values, depending on payload conditions.

In order to sense the rotary position of the wheels each electromotor is outfitted with a Faulhaber HEDM 5500B optical quadrature encoder with 1000 lines/revolution.

Each encoder sports two output channels which are phase-shifted 90° from each other. Hence, direction of motion can be detected.

The frequency of the encoder signal will be between 0 and 100 kHz:
Frequency encoder signal: 6000 rpm / 60 (s/min) x 1000 lines/revolution = 100 kHz.

2.2.3: Electronics

Electronics provide the means to control the motors, enable the use of sensors, process data and provide power to all electrical components of the robot. In figure 6, a schematic of the electronics is shown. All components will be detailed below.

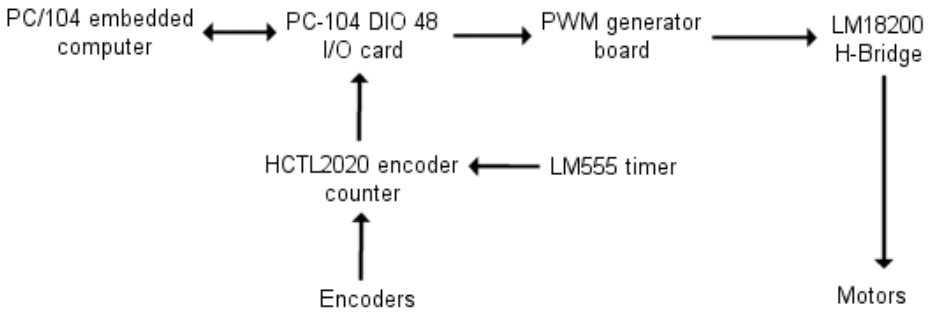

Figure 6: Schematic overview of the electronics. Arrows denote direction of signals.

PC/104 embedded computer bus:
The heart of the robot is a PC/104 embedded computer bus. In the field of robotics, the PC/104 is an industrial standard. The major advantage of this component is that it packs personal computer performance in a compact form that is flexible and easily extendible. Moreover, the device's power consumption is low and it is easily programmable. The PC/104 serves as the central processing unit of the mobile robot.

The installed PC/104 is a Kontron MOPSlcdGX1 module and measures 91 mm by 97 mm. It is outfitted with an AMD Geode GX1 processor running at 300 MHz, a 10/100 Mbit Ethernet adapter, onboard Compact Flash, two USB ports, two serial ports, VGA graphics engine, mouse and keyboard connections. At present, a 30 GB hard disk is connected to the PC/104.

I/O card:
In order to receive information from the encoders and to send signals to the motors, the PC/104's microprocessor needs an Input/Output (I/O) card. A PC104-DIO48 I/O card is installed on the robot, featuring 48 digital channels.

Encoder counter:
In order to read the signals from the encoders properly, a high sampling frequency is required. According to the Nyquist-Shannon sampling theorem, the sampling frequency must be greater than twice the bandwidth of the input signal in order to be able to reconstruct the signal correctly. In mathematical terms:

$$F_s > 2BW \qquad (2.1)$$

Here F_s is the sampling frequency and BW is the bandwidth of the input signal.
To follow this theorem, the six encoder channels (2 for each encoder) have to be sampled at 200 kHz each. In practice, one normally uses a sampling frequency that is ten times the bandwidth. In this case, that would mean 1 MHz.

Sampling and processing/decoding encoder signals at such a frequency would require a reasonable amount of computing power and slow down other processes in the central processing unit. Hence, a hardware solution is necessary in the form of a quadrature decoder circuit. Such a circuit takes encoder signal processing off the hands of the central processing unit.

Every encoder is outfitted with such a circuit consisting of a dedicated HCTL2020 encoder counter and a LM555 timer to generate clock pulses. This setup is expected to work at 1,4 MHz and thus fits the requirements. Another advantage of a quadrature decoder circuit is that the resolution of the encoder signals is multiplied by four when it is used for motion sensing. If we take this circuit and the gearbox into account, the nominal encoder resolution will then be equal to:

Nominal encoder resolution: 1000 lines/revolution x 4 x 14:1 = 56000 lines/revolution.

From experiments, it is known that this resolution does not exactly correspond with reality, therefore a real encoder resolution of 55184 lines/revolution is used. One line thus corresponds with $2\pi/55184 = 1.139 \cdot 10^{-4}$ rad. See appendix 1 for more information.

H-Bridge:
The robot's motors will be controlled by the PC/104 microprocessor. Or in other words, the motion control feedback loop will be closed in the central processing unit. This unit however is not able to send out high voltages and currents to deliver power to the three motors. Therefore, a flexible and low-level amplifier circuit is incorporated in the robot's electronics: a so-called H-Bridge (LM18200).

The H-Bridge consists of four electronic switches enabling forward and backward drive of the motor and braking. The bridge has three inputs: DIR for drive direction, BRAKE for braking and PWM for speed/duty cycle. This last input will be detailed in the next section.

PWM generator board:
The H-Bridge requires a PWM input signal to operate. This signal is a simple 0 to 5 V digital signal. The standard DIO I/O card by itself, however, is not able to produce such a signal. Therefore, a dedicated PWM generator board is included in the electronics.

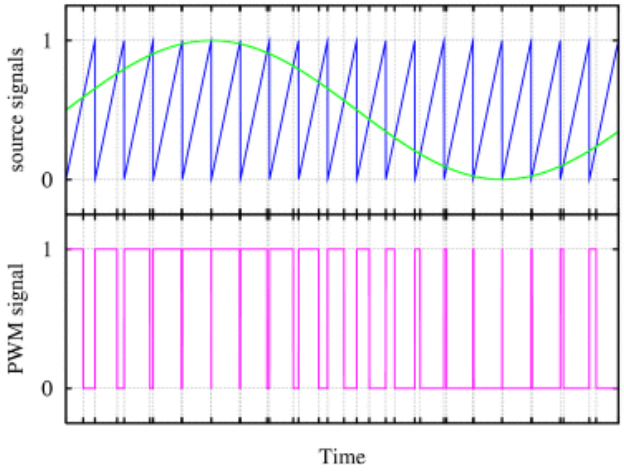

Figure 7: Pulse Width Modulation: continuous source signal (green), signal used to calculate PWM signal: sawtooth (blue), PWM signal (magenta).

PWM or sign/magnitude Pulse Width Modulation is a technique wherein the duty cycle (or number of pulses per unit time) of a signal is modulated to adjust the amount of power that is sent to the motors. The PWM signal is a square wave with just two voltage values that correspond with zero speed and maximum speed (see figure 7, obtained from [3]).

The technique consists of approximating a desired continuous signal by switching between the two voltage values (in essence switching the motors on and off), so that the resulting average value of the PWM signal matches the desired continuous value (speed).

Power supply:
At present, the robot is powered by two voltage sources; one providing 5 V for the electronics and the other providing 12 V for the motors. In the future, the robot will be driven entirely by batteries as this enables full independent movement. A set of batteries and electronics have been ordered. However, due to delivery delays, it was not possible to make the robot's power supply independent during the project described in this report.

The ordered batteries were a set of Polyquest PQ-B2600 HG 4S Lithium-polymer batteries (14.8 V). These four-cell batteries have a capacity of 2600 mAh and are able to discharge at a maximum continuous current of 20 A. Each battery pack only weighs 220 grams and has an exceptional energy density.

During modification experiments, it was discovered that the combination of PC/104 embedded computer and other electronics needs at least 5.3 V and 2.5 A to boot up and function properly, even without a pocket router connected (see next paragraph). This is not a standard voltage. Moreover, normal voltage regulators and diode circuits are not able to supply this voltage without excessive heat generation. A solution was found in the form of a ST Microelectronics GS-R400V switching regulator with programmable output voltage, which is able to provide the 5.3 V. An additional regulator can be used to reduce 14.8 V to 12 V for the motors.

2.2.4: Control station and software

One of the objectives of the project described in this report is to enable wireless control of the robot from a control station. This was achieved through installing dedicated hardware and software on the robot and setting up a control station. The control station (see figure 8) consists of a desktop computer running Windows XP, a Logitech Freedom 2.4 cordless joystick and a D-Link DWL-G122B1 Wireless USB Adaptor.

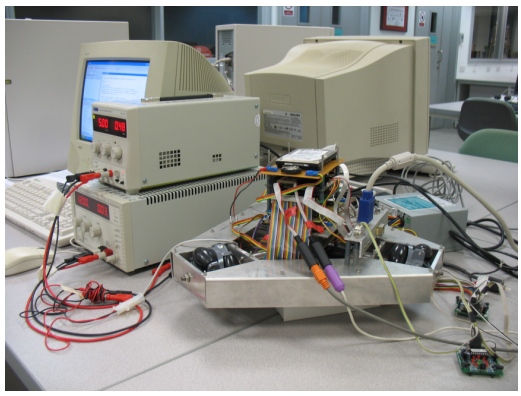

Figure 8: Control station and power supply (left) and robot (front right)

The robot currently runs Red Hat Linux 9 Kernel 2.4.20 with Real-Time Application Interface (RTAI) installed. The robot is outfitted with a D-Link DWL-G730AP Wireless Pocket Router/Access Point.

Communications use the SSH Secure Shell protocol and through a client program on the desktop computer commands can be executed on the robot. A schematic of this arrangement is shown in figure 9.

Figure 9: Schematic overview of wireless communication

All real-time control algorithms were implemented in the C programming language, which is ideal for this application due to its versatility, possibility of low-level bit and byte programming and high execution speed. The GCC compiler of the robot was used to generate stand-alone programs.

As mentioned before, a wireless joystick was added to the control station. At the time of writing, the robot's movements could not be controlled in real-time with the joystick. However, the library with control algorithms was specifically written to make a future implementation of steering with a joystick easier. It certainly is an end goal of the omnidirectional robot development program to be able to control the robot through a wireless joystick.

Chapter 3: Kinematics

Now that a basic understanding of the omnidirectional robot has been developed, we can investigate the vehicle's kinematics. If we want to prescribe the robot's movements in the environment, we need to know how these variables relate to the primary variables we can control: the angular positions and velocities of the wheel shafts. Therefore, a kinematical model of the robot has to be developed.

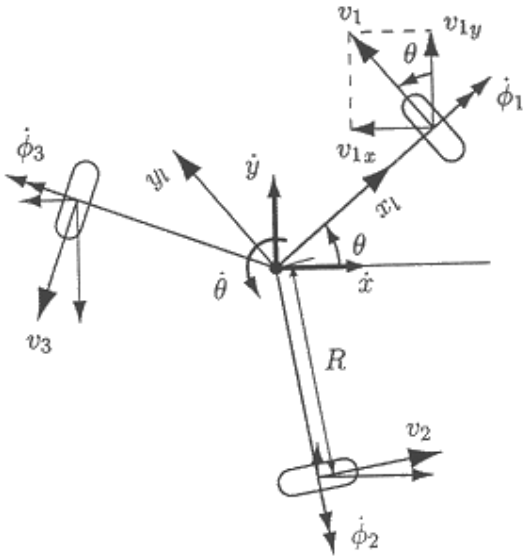

Figure 10: Kinematic diagram of the robot

In Rob van Haendel's report [2], the kinematic relations of the omnidirectional robot have been presented for the first time. This derivation will be presented and expanded upon below.

Let's start with defining a global frame [x, y] which represents the environment of the robot, see figure 10. The robot's location and orientation in this global frame can be represented as (x, y, θ). The global velocity of the robot can be written as $(\dot{x}, \dot{y}, \dot{\theta})$.

Now we can also define a local frame [x_L, y_L] that is attached to the robot itself. The center of this local frame coincides with the center of gravity of the robot. The three omniwheels are located at an angle α_i (i = 1, 2, 3) relative to the local frame. If we take the local axis x_L as starting point and count degrees in the clockwise direction as positive, we have $\alpha_1 = 0°$, $\alpha_2 = 120°$ and $\alpha_3 = 240°$.

From the figure, it is obvious that the elements that connect the environment with the robot are the wheel hubs and are thus a good starting point for a kinematic relation.

The translational velocities of the wheels v_i on the floor determine the global velocity of the robot in the environment $(\dot{x}, \dot{y}, \dot{\theta})$ and vice versa. The translational velocity of wheel hub v_i can be divided into a part due to pure translation of the robot and a part due to pure rotation of the robot:

$$v_i = v_{trans,i} + v_{rot} \qquad (3.1)$$

First, pure translation will be considered. In figure 11, a close-up of $v_{trans,1}$, the translational velocity at wheel hub 1, is shown. We can map the unit vector $v_{trans,1}$ onto the vectors $\dot{x}$ and $\dot{y}$ to obtain (3.2).

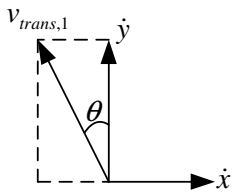

Figure 11: Close-up of translational velocity at wheel hub 1

$$v_{trans,1} = -\sin(\theta)\dot{x} + \cos(\theta)\dot{y} \tag{3.2}$$

We can generalize this vector mapping for all wheels when we take into consideration that the vectors v_i are positioned at an offset $\theta+\alpha_i$. Therefore, we can write:

$$v_{trans,i} = -\sin(\theta + a_i)\dot{x} + \cos(\theta + a_i)\dot{y} \tag{3.3}$$

When the platform executes a pure rotation, the hub speed v_i needs to satisfy the following equation:

$$v_{rot} = R\dot{\theta} \tag{3.4}$$

Here R is the distance from the center of gravity of the robot to the wheels along a radial path. The value of R is 0.171 m.

When we substitute both influences in (3.1), we end up with the following:

$$v_i = -\sin(\theta + \alpha_i)\dot{x} + \cos(\theta + \alpha_i)\dot{y} + R\dot{\theta} \tag{3.5}$$

We have now coupled the translational velocity of the wheels to the global velocities of the omnidirectional platform, but we can go a step further. The translational velocity of the hub is related to the angular velocity $\dot{\phi}_i$ of the wheels through:

$$v_i = r\dot{\phi}_i \tag{3.6}$$

where r is the radius of an omnidirectional wheel: 0.04m.

Thus we can substitute (3.6) in (3.5) and after rearranging this results in:

$$\dot{\phi}_i = \frac{1}{r}(-\sin(\theta + \alpha_i)\dot{x} + \cos(\theta + \alpha_i)\dot{y} + R\dot{\theta}) \tag{3.7}$$

We can transform (3.7) to matrix representation (3.8) and (3.9):

$$\underline{\dot{\phi}} = J_{inv}\,\underline{\dot{u}} \tag{3.8}$$

In this relation, J_{inv} is the inverse Jacobian for the omnidirectional robot that provides a direct relationship between the angular velocities of the wheels $\underline{\dot{\phi}}$ and the global velocity vector $\underline{\dot{u}}$.

$$\begin{bmatrix} \dot{\phi}_1 \\ \dot{\phi}_2 \\ \dot{\phi}_3 \end{bmatrix} = \frac{1}{r} \begin{bmatrix} -\sin(\theta) & \cos(\theta) & R \\ -\sin(\theta+\alpha_2) & \cos(\theta+\alpha_2) & R \\ -\sin(\theta+\alpha_3) & \cos(\theta+\alpha_3) & R \end{bmatrix} \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} \tag{3.9}$$

In most cases, it is not convenient for users to steer a robot in global coordinates however. It is far more natural to think and steer in local coordinates. Fortunately, we can now simply convert global to local coordinates with the following equation:

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} \cos(\theta) & 0 & 0 \\ 0 & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \dot{x}_L \\ \dot{y}_L \\ \dot{\theta} \end{bmatrix} \tag{3.10}$$

Substituting equation (3.10) in (3.9) leads to:

$$\begin{bmatrix} \dot{\phi}_1 \\ \dot{\phi}_2 \\ \dot{\phi}_3 \end{bmatrix} = \frac{1}{r} \begin{bmatrix} -\sin(\theta) & \cos(\theta) & R \\ -\sin(\theta+\alpha_2) & \cos(\theta+\alpha_2) & R \\ -\sin(\theta+\alpha_3) & \cos(\theta+\alpha_3) & R \end{bmatrix} \begin{bmatrix} \cos(\theta) & 0 & 0 \\ 0 & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \dot{x}_L \\ \dot{y}_L \\ \dot{\theta} \end{bmatrix} \tag{3.11}$$

This matrix relation in the local frame can also be expanded to three separate equations for easy implementation in programming applications:

$$\begin{aligned} \dot{\phi}_1 &= (-\sin(\theta)\cos(\theta)\dot{x}_L + \cos^2(\theta)\dot{y}_L + R\dot{\theta})/r \\ \dot{\phi}_2 &= (-\sin(\theta+\alpha_2)\cos(\theta)\dot{x}_L + \cos(\theta+\alpha_2)\cos(\theta)\dot{y}_L + R\dot{\theta})/r \\ \dot{\phi}_3 &= (-\sin(\theta+\alpha_3)\cos(\theta)\dot{x}_L + \cos(\theta+\alpha_3)\cos(\theta)\dot{y}_L + R\dot{\theta})/r \end{aligned} \tag{3.12}$$

Chapter 4: Control laws

When we start using the omnidirectional robot, we have an idea about the kind of movement we want the robot to perform in the environment: the desired or reference motion. With the kinematic relationships from the previous chapter, specifically equation (3.11), it is always possible to convert this reference motion in the environment to a reference motion for the robot's three wheel shafts: ($\dot{\phi}_{ref,1}, \dot{\phi}_{ref,2}, \dot{\phi}_{ref,3}$) and through integration: ($\phi_{ref,1}, \phi_{ref,2}, \phi_{ref,3}$). This topic will be further elaborated in the next chapter.

In general it is hard to perfectly follow such a reference path, because of the effects of friction and other disturbances on the system. To minimize these effects, a feedback control system will be designed.

For control, we have the angular positions (ϕ_1, ϕ_2, ϕ_3) and the velocities of the wheel shafts ($\dot{\phi}_1, \dot{\phi}_2, \dot{\phi}_3$) at our disposal by changing the voltage to be sent by the PC/104 to the PWM generator board. We can use the first set for a position controller and the second set for a velocity controller. In order to be able to compare performance, both types of controllers will be implemented.

4.1: Position and velocity control

In position control, the control action is computed based on the difference between the real angular positions and the reference positions. This can be expressed in a mathematical sense as:

$$e = \phi_{ref} - \phi \tag{4.1}$$

Here *e* is the tracking error [rad], ϕ_{ref} is the reference angular position [rad] and ϕ is the real angular position [rad].

For the case of the omnidirectional robot, equation (4.1) has to be expanded to take into account the fact that the robot has three wheels and consequently three errors:

$$e_i = \phi_{ref,i} - \phi_i \tag{4.2}$$

Here the subscript *i* (= 1,2,3) denotes which wheel is considered.

Another solution is to use velocity control. Here, the tracking error is defined based on the real angular velocities [rad/s] and reference velocities [rad/s]:

$$e = \dot{\phi}_{ref} - \dot{\phi} \tag{4.3}$$

Or for our multi-wheel case:

$$e_i = \dot{\phi}_{ref,i} - \dot{\phi}_i \tag{4.4}$$

A schematic diagram of a feedback control system is presented in figure 12.
Here, *r(t)* represents the group of reference signals, so either ($\phi_{ref,1}, \phi_{ref,2}, \phi_{ref,3}$) or ($\dot{\phi}_{ref,1}, \dot{\phi}_{ref,2}, \dot{\phi}_{ref,3}$) depending on the type of controller. The tracking errors are represented by *e(t)*. The feedback controller is *C(t)* and *u(t)* represents the control inputs/actions calculated.

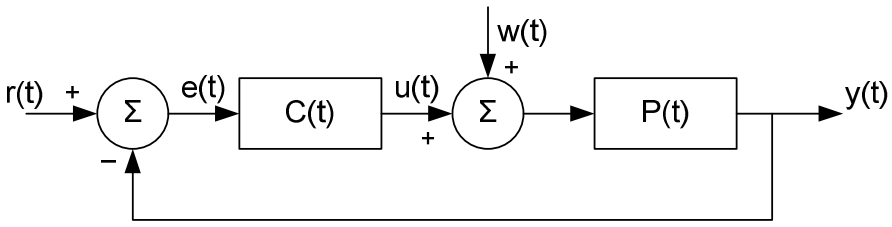

Figure 12: Feedback control system

The disturbances are collected in the variable *w(t)* and *P(t)* represents the system. The output signals are either (ϕ_1, ϕ_2, ϕ_3) for a position controller or ($\dot{\phi}_1, \dot{\phi}_2, \dot{\phi}_3$) for a velocity controller.

In the figure it is assumed that the optical encoders have an unity gain for their operating spectrum in this application which, according to their specifications, is a valid assumption.

Several basic controllers will be implemented based on proportional, integral and derivative feedback: P, PI, PID and PD.

4.2: Discrete implementation

The omnidirectional robot is a digital system. It samples signals from the optical encoders and uses a microprocessor to close the motion control loop. Unlike analogue electronics which operate in the continuous domain, a digital system operates in the discrete domain.
This implies that any implementation of a control law should also be made in discrete form.

Because of this, and the related fact that digital systems cannot perform pure integration and differentiation, the continuous motion control laws should be approximated with difference equations. In the following table, the continuous P, PI, PID and PD control laws are shown side by side with their discrete counterparts.

Table 1: Control law implementations

Type	*Continuous form*	*Discrete form*
P	$u(t) = K_p e(t)$	$u_k = K_p e_k$
PI	$u(t) = K_p e(t) + K_i \int_0^t e(t)dt$	$u_k = K_p e_k + K_i \Delta T \sum_{j=1}^{k} e_j$
PID	$u(t) = K_p e(t) + K_i \int_0^t e(t)dt + K_d \dot{e}(t)$	$u_k = K_p e_k + K_i \Delta T \sum_{j=1}^{k} e_j + \frac{K_d}{\Delta T}(e_k - e_{k-1})$
PD	$u(t) = K_p e(t) + K_d \dot{e}(t)$	$u_k = K_p e_k + \frac{K_d}{\Delta T}(e_k - e_{k-1})$

Here K_p is the proportional gain [-], K_i is the integral gain [-] and K_d is the derivative gain [-], *u(t)* is the control signal [-] and ΔT is the sampling period [s]. The index *k* denotes the current discrete time instant, *k-1* is the previous time instant.

In the discrete case, the derivative term is based on a first-order approximation of continuous differentiation, also called Euler's method [4]. The integral term is based on the right approximation of the rectangle method in integral calculus [5].

4.3: Performance indicators

In the introduction of this report, it was mentioned that the performance in time domain is important. Therefore, frequency domain performance will not be considered in the following chapters.

The main indicators of performance that will be considered are the steady state error, the overshoot, the settling time and rise time (see figure 13). The values of these indicators will be obtained from step response data.

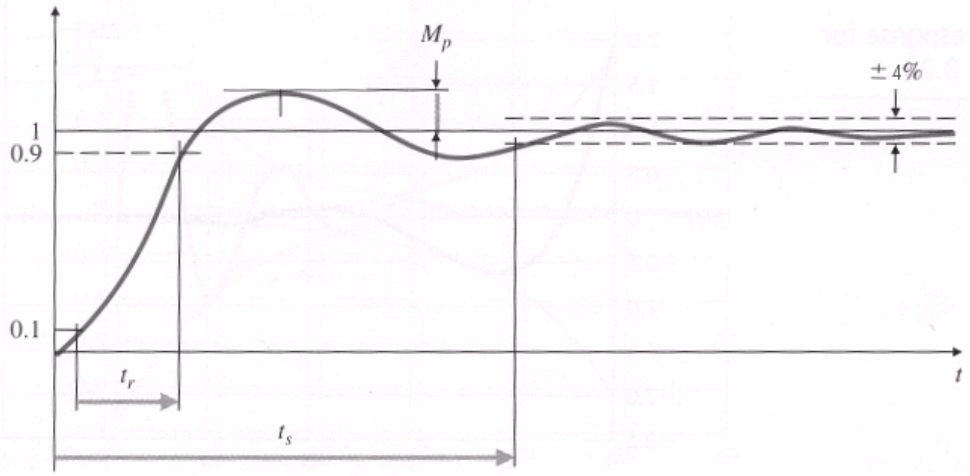

Figure 13: Time-domain performance indicators

The final value is defined as the value of the step response at the end of an experiment.

The **steady state error** is defined as difference between the final value and the desired value.

The **overshoot** M_p is the maximum amount the system overshoots its final value divided by its final value. The use of the final value is especially relevant for situations where the response exhibits an overshoot, but does not reach the setpoint.

The **settling time** t_s is the time it takes for the system transients to decay. Here, the 4% settling time is used. It is defined as the time interval from the moment when the step input is given until the moment when the step response falls into the 4% range surrounding the final value of the response.

Finally, the **rise time** t_r is the time the system needs to reach the vicinity of the final value. Here, it is defined as the time interval between the moments the response reaches 10% and 90% of the final value.

4.4: All-round and precise controller

In control, we know that some performance indicators can conflict. For example, when we want to reduce the steady state error, an increase of the integral gain K_i can be considered. However, this can lead to a higher overshoot, which might be undesirable. From this example it becomes obvious that often a trade-off is required between performance indicators.

To learn more about the capabilities of the robot, all controllers will be tuned twice. Once based on one set of requirements and once on another set (see table 2). The first set is called *all-round*. In this set, all four performance indicators are considered equally important.

The absolute value of the steady state error should be below 1° (≈ 0.017 rad) for position control or 1 rpm (≈ 0.105 rad/s) for velocity control within 0.25 s while the overshoot and rise time should remain reasonable.

The second set is called *precise* and focuses solely on a minimum absolute value of the steady state error. Other performance indicators are not considered important.

Table 2: Requirements for all-round and precise controller

Performance indicator	*All-round controller requirement*	*Precise controller requirement*
Absolute steady state error		
position:	< 1°	<< 1°
velocity:	< 1 rpm	<< 1 rpm
Overshoot	< 5%	Not important
Settling time	< 0.25 s	Not important
Rise time	< 0.1 s	Not important

These two different sets will give insight into the possibilities in regards to performance.

Chapter 5: Trajectory profiles

To test the control laws' performance, a way to specify desired movement must be found. Since it is not possible yet to do this with a joystick (see paragraph 2.2.4), an alternative method has to be developed.

This new method should satisfy a number of requirements:

- Formulating desired movement should be fast and easy.
- It should be possible to emulate the use of a joystick.
- Future adaptation of the C implementation to include use of a real joystick should be straightforward.

A method that meets these requirements was developed. The user can specify desired movement with a *trajectory profile*: a position or velocity reference as function of time.
The desired movement of the robot can be provided in one of the following ways:

- Position of the robot in local frame: x_L, y_L, θ. This profile is abbreviated as *posref*.
- Velocity of the robot in local frame: $\dot{x}_L, \dot{y}_L, \dot{\theta}, \theta$. This profile is abbreviated as *velref*. Note that $\dot{\theta}$ and θ are related to each other through integration.
- Angular velocities of the wheel shafts: $\dot{\phi}_{ref\,1}, \dot{\phi}_{ref\,2}, \dot{\phi}_{ref\,3}$. This profile is abbreviated as *wheelref*.

The profile is supplied to the robot in the form of a data file. In this file, the reference during one sample period is saved on one line. Thus, the file is essentially a side-by-side group of three or four arrays with each array representing the evolution of one coordinate. The length of the arrays is determined by the number of samples:

$$N_s = F_s \cdot L \tag{5.1}$$

Here N_s is the number of samples [-], F_s is the sampling frequency [Hz] and L is the length of the experiment [s].

For a quick example, consider a velocity profile (*velref*) that shows an acceleration in y_L with a sampling frequency of 200 Hz:

```
               XLdot - YLdot - Thetadot - Theta
t = 0.000 s:   0       0       0          0
t = 0.005 s:   0       0.1     0          0
t = 0.010 s:   0       0.2     0          0
...
```

Note that in the data file, no legend and time tags are included. These are simply added here for better readability.

At the start of an experiment, the robot's software will load the profile and calculate the required motion of the wheel shafts: the reference. This process is shown in figure 14.
The controller uses either angular reference positions (in case of a position control) or angular reference velocities (in case of velocity control) of the wheels. Apart from angular wheel velocities, profiles however can also be formulated in local frame positions and velocities. Thus, to calculate the reference, a number of conversions are build into the trajectory profile processing.

The trajectory profile implementation uses a combination of numerical differentiation/ integration and kinematic relations to convert profile data to $\phi_{ref,i}$ and $\dot{\phi}_{ref,i}$.

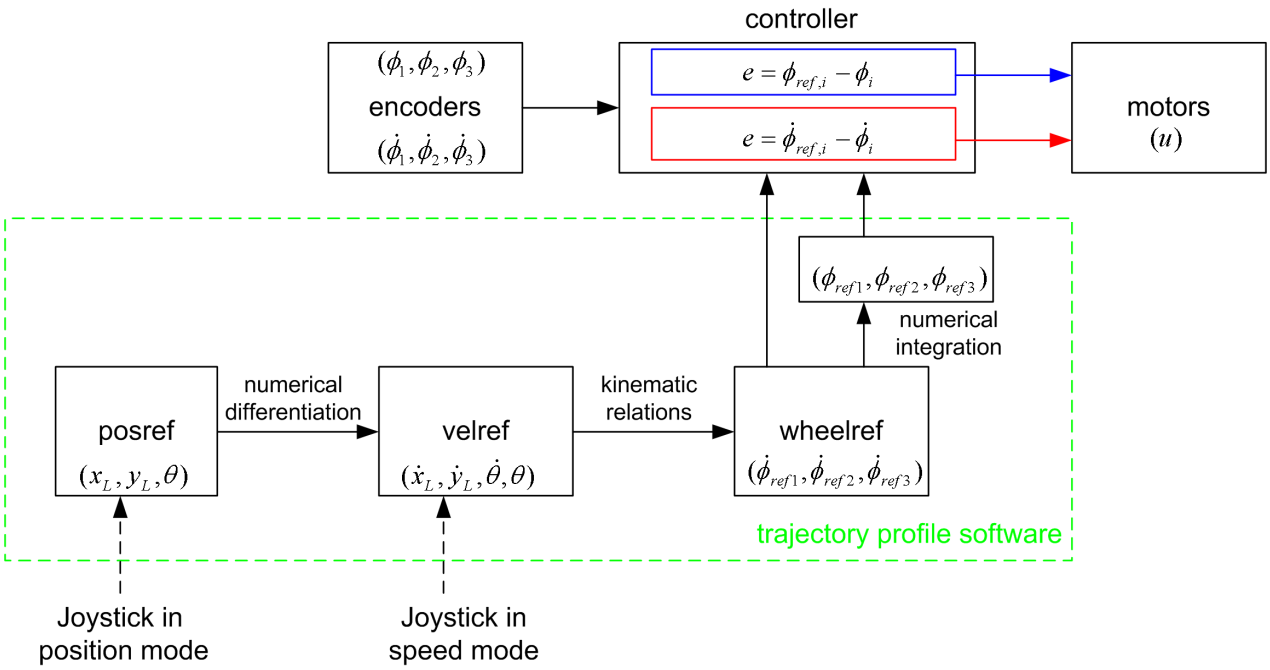

Figure 14: Schematic overview of the trajectory profile concept. The blue arrows are associated with position control, the red arrows with velocity control.

For differentiation, the following formula is used:

$$\dot{\phi}_{ref}[i] = \frac{\phi_{ref}[i] - \phi_{ref}[i-1]}{\Delta T} \tag{5.2}$$

For integration, a formula based on the trapezoidal rule, is used:

$$\phi_{ref}[i] = \phi_{ref}[i-1] + \frac{\Delta T}{2}(\dot{\phi}_{ref}[i] + \dot{\phi}_{ref}[i-1]) \tag{5.3}$$

In the equations above, the index [i] refers to the current sampling period, whereas [i-1] refers to the sampling period before the current one, with a difference of ΔT in between. The advantage of both equations is that they are fast to compute and reasonably accurate, particularly for profiles that are not smooth, such as step inputs.

Once the entire profile is converted to references for angular positions and velocities of the wheel shafts, these references are saved to memory. During the experiment, at every new sampling period, the controller has access to the reference at that specific time instant and can use it to calculate the tracking error and ultimately the control action.

Let's take a look at some examples of trajectory profiles. In figure 15, a simple *posref* profile is presented with a step in the x_L coordinate. Figure 16 shows a *velref* profile that consists of three parts. In the first part, the robot moves in the positive x_L direction with a constant velocity. Thereafter, a rotation around the center-of-gravity is performed with constant rotational velocity. In the last part, the robot is at rest. A *wheelref* profile is displayed in figure 17. Here, the first wheel accelerates to a speed of 40 rad/s (not far below the limit angular velocity) and then decelerates. When the first wheel comes to a stop, the second wheel mimics the movement. The third wheel does not move during the entire experiment.

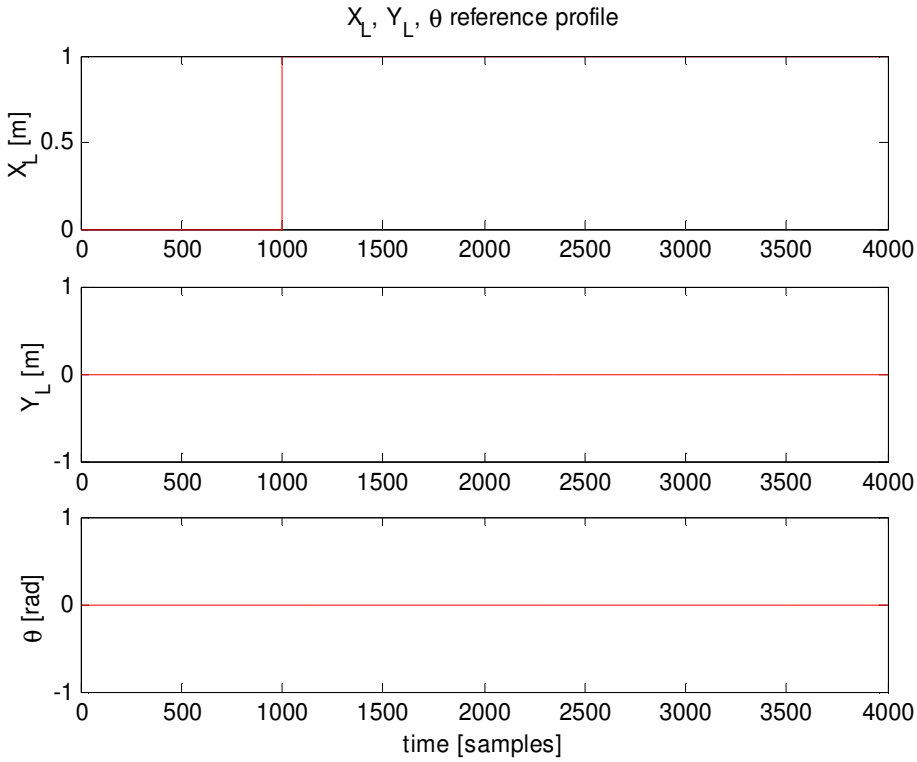

Figure 15: Example of a posref profile. Fs = 200 Hz.

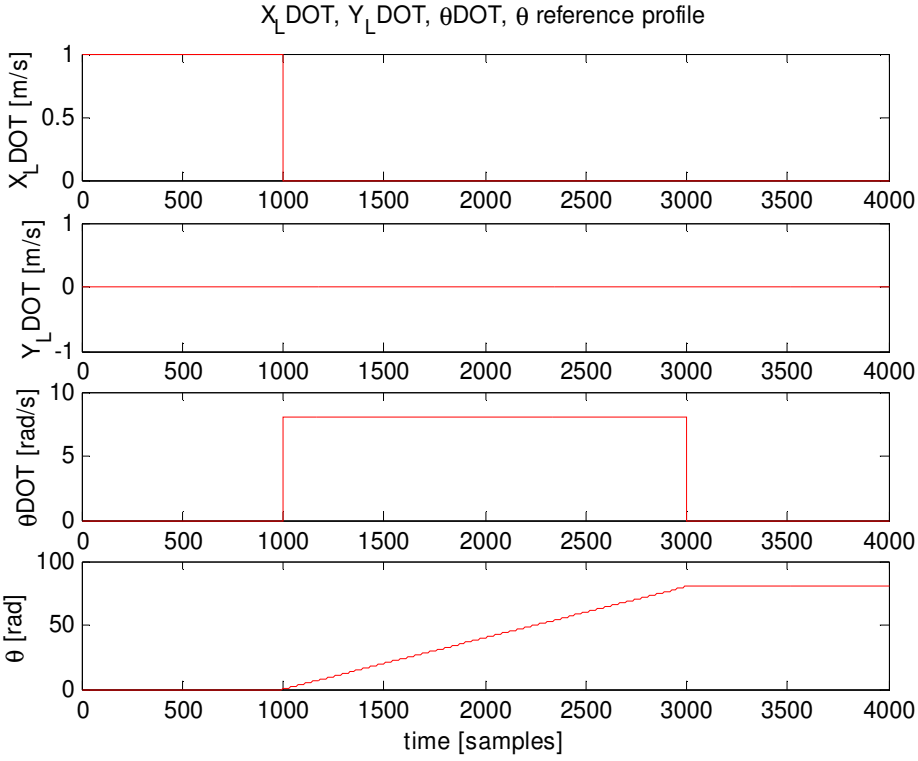

Figure 16: Example of a velref profile. F_s = 200 Hz.

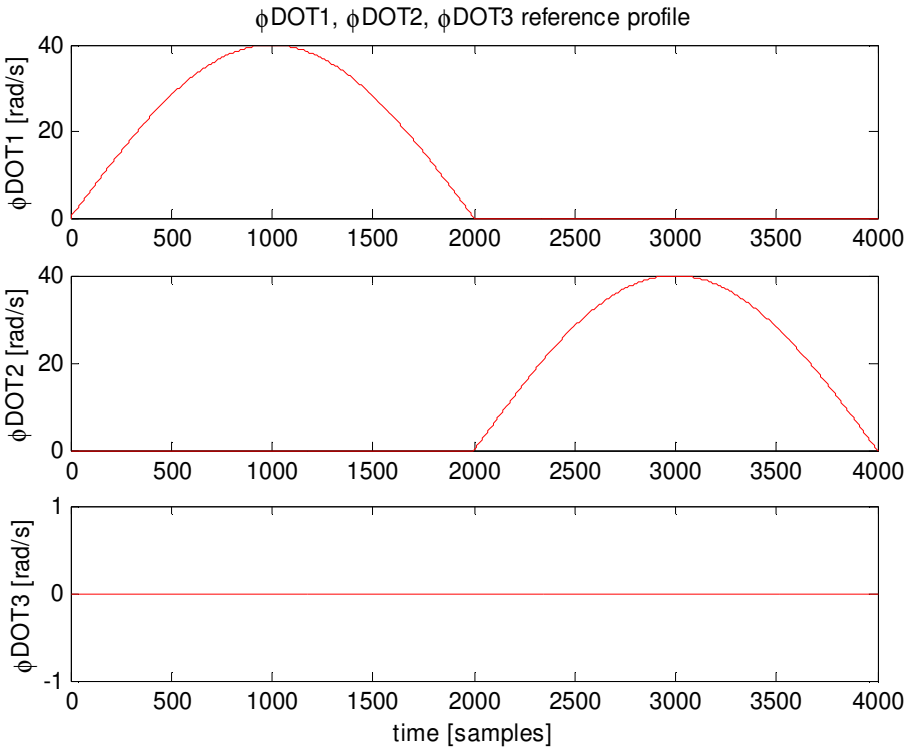

Figure 17: Example of a wheelref profile. F_s = 200 Hz.

Trajectory profiles provide maximum flexibility for the user and are easy to produce. In this project, Matlab scripts were used to generate them, but any mathematical package with the ability to write arrays to disk is usable.

When creating trajectory profiles, the designer should keep the system's capabilities in mind. The maximum angular velocity the motors and gearbox can provide is 45 rad/s or 430 rpm or 1.8 m/s. The angular acceleration is also limited. If the user's request exceeds the robot's capabilities, the best possible performance is given.

Earlier in this chapter, it was mentioned that it would be good to emulate the use of a joystick. The most intuitive way to control a vehicle with a joystick is to use it in speed mode. In this mode, the vehicle's speed is proportional to the amount of perturbation of the joystick from its center position. The vehicle is at rest when the stick is centered and unperturbed. From the above it should be obvious that a *velref* profile can be used to emulate a joystick in speed mode. When one would like to use a joystick in position mode, one could use a *posref* profile.

For use with a real joystick, the C implementation should be changed somewhat. The control station could send a single data packet every sampling period, for instance using a TCP/IP connection, with a desired movement (most likely a velocity reference in local frame) based on the joystick's perturbation. This desired movement should then be read and instantaneously converted into $\phi_{ref,i}$ and $\dot{\phi}_{ref,i}$ during the same sampling period, if possible. The exact implementation falls outside the scope of this report.

Chapter 6: Implementation in C

Aside from hardware, the omnidirectional robot needs software to function. This software was written in the programming language C and fulfils a dual purpose:

- It closes the motion control loop. The real-time controller is implemented as code.
- It provides low-level functionality in the form of a library of functions that can be used and built upon by high-level algorithms of other researchers. Some examples of available functions are: resetting and reading the encoders, sending commands to the motors and providing safety features.

In this chapter, the most important aspects of the software will be discussed. A version of the code called *omniPos* is used as guide. This version is meant for position control experiments. The other versions have essentially the same structure, but use different control strategies. It is assumed the reader has a basic understanding of C. A full listing of the code is available separate from this report.

The code consists of three main parts:

1. A main program (*omniPos.c*)
2. A function library (*omni.c*)
3. A header file containing definitions and declarations (*omni.h*)

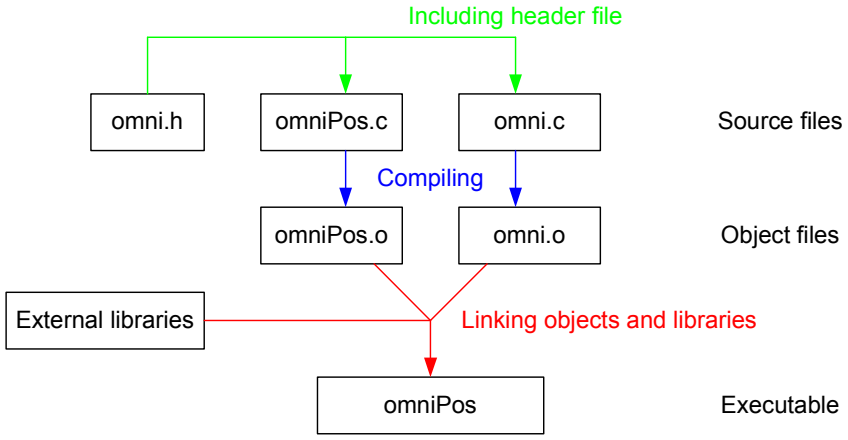

Figure 18: Software components

A graphical representation of the relationship between these parts is shown in figure 18. Below, the three main parts, the source files, will be described in detail. Afterwards, in paragraph 6.4, the other components in the figure, their function and usage in experiments will be addressed.

6.1: Main program (*omniPos.c*)

In this project, the main program is the most important part of the code. For future usage of the robot as test bed, the function library *omni.c* is more valuable however, as it can be considered a device driver for the omnidirectional robot. The main program simply makes use of this driver; it is specific and needs to be changed for other applications.

The heart of the main program is the function *main* (displayed in the next frame). This function prescribes exactly what actions should be taken and in what order.

```
// Main program:
 int main(void) {

   start_comedi();
   zero_encoders();
   read_pos_profile();

   start_rt_task();
   stop_rt_task();

   stop_comedi();
 }
```

The first action in the main function is a call to the *start_comedi* function. Comedi is a collection of open source drivers for data acquisition and I/O cards, such as the DIO48 I/O card. With Comedi we have an interface to interact with the I/O card from inside Linux (figure 19). The *start_comedi* function configures and initializes the I/O card's ports.

The next function call to *zero_encoders* makes sure all encoders are reset to zero position and velocity. Subsequently, the desired test trajectory profile is loaded into memory with the function *read_pos_profile*.

The next two function calls need some background information.

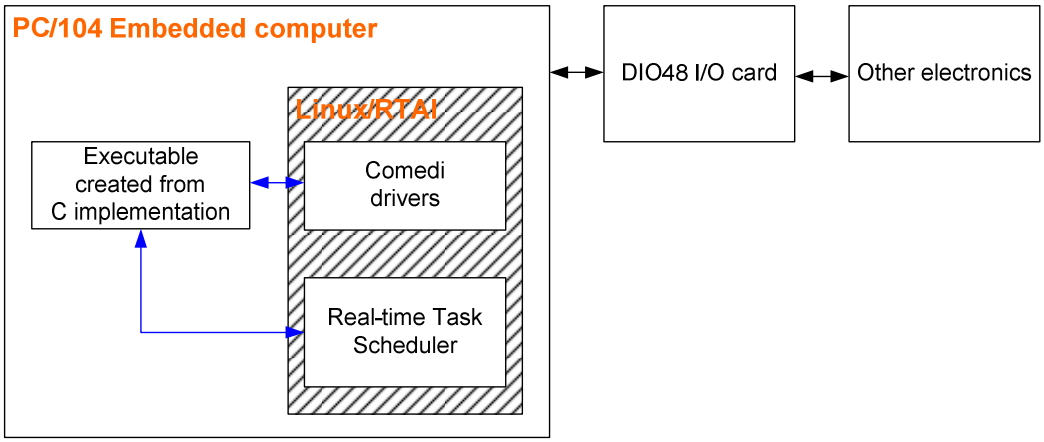

Figure 19: C implementation and interactions with the environment at run time. Blue arrows denote data transfer, black arrows denote voltage signals.

In applications such as motion control, it is very important that actions are executed at the exact right time instant to achieve good results, in other words "*executed in real time*". Examples would be polling a sensor, processing data and sending an actuator command. Real-time operating systems, such as RTAI Linux, provide this functionality. They schedule tasks to be executed at specific time instances. It is also possible to periodically execute the same instructions.

This is the kind of functionality that is desired for the implementation of a discrete motion controller. The function *start_rt_task* sets up a periodic real-time task in collaboration with the Linux/RTAI task scheduler. The function *thread_fun* specifies what the periodic real-time task is.

Every sampling period, a number of things need to be done (see figure 20). The sampling period starts with the input phase. In this phase, all sensors are read, in this case the encoders. In the subsequent calculation phase, all processing is done. Based on the encoder count values the current orientation and velocities of the wheels are calculated (calculation phase).

Subsequently, in *OmniPos*, the tracking error is calculated from the difference between the real angular positions and the desired ones. Using the control laws, suitable control actions are determined. In the next phase, the output phase, these are sent as commands to the motors.

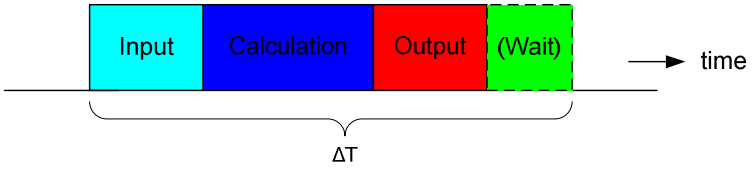

Figure 20: Composition of sampling period

In an ideal situation, the moment when the last command is sent in the output phase coincides with the end of the sampling period. The main program uses a hard real-time philosophy for non-ideal situations. In case there still is time available before the start of the next sampling period, the controller will perform no more actions and simply wait. Note that the motors are controlled by a voltage signal proportional to the desired duty cycle. In the wait phase, this voltage signal is held constant following the zero order hold principle. Thus in almost all cases, the motors will keep rotating in the wait phase. It is also possible that the sampling time expires before one or more of the input-calculation-output phases have been completed. In such cases, the remaining phases will be broken off and the new sampling period will start immediately.

A philosophy that is not used here is soft real-time. The behaviour of this philosophy is slightly different. In case the sampling time expires before the input-calculation-output phases end, these phases will be completed first and only then a new sampling period is started. Obviously, a soft real-time philosophy is less strict. For very high sampling frequencies, the controller might not have enough time for calculation and for sending commands. Both philosophies will then lead to loss of performance. Hard real-time might show moments where the motors should be actuated, but are not. Soft real-time will actuate the motors, but sample periods and thus experiments may take longer than specified. To summarize, one should be cautious when increasing the sampling frequency; it cannot be increased indefinitely.

At the end of the experiment, the function *stop_rt_task* will be called. This function instructs the Linux/RTAI task scheduler to remove the real-time task from memory. Additionally, performance data is written to disk.

The last function call of the function *main* is *stop_comedi.* This function closes the Comedi interface. With this, we have reached the end of the function *main*.

Finally, let's take a more in-depth look at the function *thread_fun* where the discrete controller is implemented. Note that some portions of the code (denoted with …) have been omitted for clarity:

```
void *thread_fun(void *arg) {

   // Declare RT task variables:
   ...
   double t1;
   int disp1 = 0, disp2 = 0, disp3 = 0;
   double phi1 = 0.0, phi2 = 0.0, phi3 = 0.0;
   double phidot1 = 0.0, phidot2 = 0.0, phidot3 = 0.0;
   double u1 = 0.0, u2 = 0.0, u3 = 0.0;
   double e_new1 = 0.0, e_new2 = 0.0, e_new3 = 0.0;
   double e_total1 = 0.0, e_total2 = 0.0, e_total3 = 0.0;
   double Kp = 90.0;
   double Ki = 18.0;
   double Kd = 0.0;
```

```
int FirstRun = 1;
...

while( RT_Count <= NT ) {

  t1 = rt_get_cpu_time_ns()/1000000 ;

  // -- Implement real-time tasks here: --
  Ct1[Count] = read_encoder(1);          // Read position motor 1
  Ct2[Count] = read_encoder(2);          // Read position motor 2
  Ct3[Count] = read_encoder(3);          // Read position motor 3

  if (FirstRun == 1) {
      // Calculate count displacement:
      disp1 = calc_disp(Ct1[Count], 0);
      disp2 = calc_disp(Ct2[Count], 0);
      disp3 = calc_disp(Ct3[Count], 0);
      FirstRun = 0;
  } else {
      // Calculate count displacement:
      disp1 = calc_disp(Ct1[Count], Ct1[Count-1]);
      disp2 = calc_disp(Ct2[Count], Ct2[Count-1]);
      disp3 = calc_disp(Ct3[Count], Ct3[Count-1]);
  }

  // Calculate angular position:
  phi1 = phi1 + disp1*(2.0*pi/55184.0);
  phi2 = phi2 + disp2*(2.0*pi/55184.0);
  phi3 = phi3 + disp3*(2.0*pi/55184.0);

  Pos1[Count] = phi1;
  Pos2[Count] = phi2;
  Pos3[Count] = phi3;

  // Calculate displacement:
  // Ang disp = counts * ((360 degrees/rev) / (55184 counts/rev))
  // Ang vel  = angular displacement / delta t
  phidot1 = (disp1*(2.0*pi/55184.0))/delta_t;
  phidot2 = (disp2*(2.0*pi/55184.0))/delta_t;
  phidot3 = (disp3*(2.0*pi/55184.0))/delta_t;

  Vel1[Count] = phidot1;
  Vel2[Count] = phidot2;
  Vel3[Count] = phidot3;

  // Calculate errors:
  e_new1 = phi_ref1[Count] - phi1;
  e_new2 = phi_ref2[Count] - phi2;
  e_new3 = phi_ref3[Count] - phi3;

  // PI control
  e_total1 = e_total1 + e_new1;
  e_total2 = e_total2 + e_new2;
  e_total3 = e_total3 + e_new3;
  u1 = Kp*e_new1 + Ki*delta_t*(e_total1);
  u2 = Kp*e_new2 + Ki*delta_t*(e_total2);
  u3 = Kp*e_new3 + Ki*delta_t*(e_total3);

  move_motor(1,limit(round(u1)));
  move_motor(2,limit(round(u2)));
  move_motor(3,limit(round(u3)));
```

```
    ...
    Count++;
    RT_Count++;
    rt_task_wait_period();
    ...
    // End of real-time task
  }
  ...
}
```

To start with, a number of variables are defined including controller gains, system status variables and more. These will be discussed below.

Then, the periodic real-time task is introduced in the form of a *while* loop. The loop will be repeated as long as *RT_Count* ≤ *NT*. The variable *NT* is the total number of samples that fit in the experiment length. The variable *RT_Count* is an index that stores the current sample number. Thus the *while* loop will be repeated until the desired experiment length has expired.

Subsequently, the encoders are read with the function *read_encoder* and their values are stored in the *Ct* arrays. The encoders each return an encoder count: an integer between 0 and 65535. When a motor shaft rotates in the positive direction, the associated encoder count will increase. Following the same logic, a motor that rotates in the negative direction will lead to a decrease of the associated encoder count. When the count exceeds 65535, the encoder will reset and count up from 0 because of C's memory allocation to integers data types. Similarly, if the count drops below 0, it will reset to 65535 and count down.

An encoder count value at a certain time instant on itself does not provide any information. However, since we know how many encoder counts make up a full rotation of a wheel shaft we can relate a change in encoder counts to a change in angular displacement. Using the function *calc_disp*, the displacements of the motor shafts are calculated and stored in the variables *disp*. This displacement is used to calculate the new angular positions of the wheel shafts *phi1-3*. The arrays *Pos1-3* are also used to store this information. From the angular displacements and time length of a sampling period, the angular velocities *phidot1-3* are calculated. The arrays *Vel1-3* are also used to store this information.

Now that the status of the robot is known, the tracking errors *e_new1-3* can be computed from the current angular positions *phi1-3* and the desired reference positions *phi_ref1-3*. These reference positions originate from the desired trajectory profile. In the case of a wheel profile, the function *read_wheel_profile* is used to fill the arrays *phi_ref1-3* directly. In the case of a position or velocity profile, the functions *read_pos_profile* and *read_vel_profile* are used to read the required movement in the environment. This movement is then translated to required movement of the wheel shafts (*phi1-3*) using the function *conv_speed* which implements the kinematic relations developed in chapter 3.

We have now arrived at the discrete implementation of the control laws. In this case, a PI controller is used. In chapter 3 the discrete PI control law was shown to be:

$$u_k = K_p e_k + K_i \Delta T \sum_{j=1}^{k} e_j \qquad (6.1)$$

In the implementation, the variables *e_total1-3* are intermediate variables and represent the sums from this equation. The variables *u1-3* represent the computed control signals. Comedi requires that these control signals are integers and thus the outputs of the control laws need to be rounded with the function *round*. Moreover, the integers should lie within the range between -255 and +255. The last value corresponds with the wheels rotating in the positive direction at maximum velocity, whereas the first value corresponds with the same in negative direction. A value of 0 corresponds with a motor (or wheel) that is at rest.

To ensure that the control signal lies within the specified range, the function *limit* is used which has the following effect:

$$\begin{cases} -255 \le u \le 255 & \rightarrow \quad u^* = u \\ u > 255 & \rightarrow \quad u^* = 255 \\ u < -255 & \rightarrow \quad u^* = -255 \end{cases} \tag{6.2}$$

Here u is the original control signal [-] and u^* is the limited control signal [-]. The limited control signal can now be sent through Comedi to the electromotors with the function *move_motor*. The robot's hardware converts the control signal to a voltage automatically.

Essentially, the while loop has now been executed once. The indices *Count* and *RTCount* are increased by one. The *rt_task_wait_period* function is called to put the system in the wait phase provided there is still time left in the sampling period, before the *while* loop starts from the beginning again.

With this, the most important aspects of the main program have been discussed.

6.2: Function library (*omni.c*)

The second part of the code, *omni.c*, can be regarded as the robot's function library. In this file, a number of functions are defined to provide low-level functionality. Most of these functions have been introduced in the previous paragraph:

```
void start_comedi()
void stop_comedi()

int round(double u)
int limit(int u)

void read_wheel_profile()
void read_vel_profile()
void read_pos_profile()
double conv_speed(int rot_id, double XLdot, double YLdot, double
                  THETAdot, double theta)

void zero_encoders()
int read_encoder(int enc)
int calc_disp(int c_new, int c_old)

void move_motor(int motor, int speed)
```

The entire code of all functions can be found in the source code, available separately.
A detailed description of these functions goes beyond the scope of this report.

The following two functions have been included both for functionality and safety. The function *stop_motor* does as its name suggests; it sends a stop command to the respective motor. Its counterpart *emergency_stop* does the same for all three motors.

```
void stop_motor(int motor)
void emergency_stop()
```

A group of five functions that provide IR sensor functionality are included in omni.c. These functions are in a preliminary stage and need testing and tweaking before use:

```
int open_port(void)
void close_port(int fd)
void config_port(int fd)
void write_to_port(int fd)
void read_from_port(int fd)
```

6.3: Header file (*omni.h*)

The header file consists of three parts. In the first part, important constants are defined. Most of these variables are experiment settings, such as the length of an experiment *exp_length*, the sampling period *delta_t* and the length of the data array that contains the reference profile *ref_size*.

In the second part, a set of shared variables is declared. Shared variables are variables that can be accessed by all three source files. The first group of these variables is related to the status of the robot, such as the current angular positions and velocities of the wheels. The second group contains the desired trajectory profile.

The third part of *omni.h* contains declarations of the functions, which are defined in *omni.c*. Or in C terminology, this part specifies the prototypes of those functions.

6.4: Use in experiments

In the previous paragraphs, the source code of the software was introduced. However, as already hinted at in figure 19, the source code cannot be used directly to control the robot during experiments. The PC/104's processor can only interpret machine code. Moreover, the Linux kernel can only communicate directly with a running executable. Therefore, the C code needs to be converted to an executable with the GCC compiler.

This process can be divided in three steps (see also figure 18). In the first step, the include statement of *omni.h* in *omniPos.c* and *omni.c* is replaced with a copy of the header file. The second step involves transforming the source files to object files (compiled executable code). In the third and last step, the object files are merged into one file and linked with external libraries leading to an executable file *omniPos* (an executable has no extension in Linux). This basically means that references to standard functions and driver libraries are processed and incorporated into the compiled code.

It is usually convenient to use a *makefile* during this process. In such a file, one can specify driver libraries, such as Comedi, that are to be linked to the C code during the compiling process. When the executable has been generated it can be started in a Linux Shell window. Note that if you are about to run a robot executable for the first time after booting, the *addMod* script should be executed first so that the Comedi modules are loaded and Linux associates the correct drivers with the I/O card.

As an example, suppose the *omniPos* program is to be executed. Then the following commands have to be given in a Linux Shell window:

```
>> make omniPos
>> ./addMod
>> ./omniPos
```

Note that if you use a SSH client on the control station, you can give the same commands at SSH's promptline. Of course, you first have to create a SSH connection with the robot.

Chapter 7: Basic controllers

In this chapter, an overview will be presented of all experiments that were performed with several basic controllers based on proportional, integral and derivative feedback. Furthermore, in paragraph 7.2, the results of these experiments will be analyzed and discussed in the same order.

7.1: Experiments

All experiments are performed with a sampling frequency of 200 Hz (ΔT = 5 ms) unless otherwise specified. Naturally, performance data files become larger for increasing sampling frequencies. The highest sampling frequency that gives reliable results is 500 Hz (ΔT = 2 ms). For sampling frequencies exceeding this value, real-time issues begin to occur and the input-calculation-output phases start taking longer than the allocated sampling period. The selected frequency of 200 Hz combines compact output files with reliable results.

7.1.1: Performance comparison of control laws

In the first set of experiments, four different control laws will be compared: P, PI, PID and PD. One candidate will be selected for subsequent experiments. The four control laws will be implemented for both position and velocity control as suitability might be different between these two setups. The experiments in this chapter are to be performed frictionless: the robot is lifted on a stand so that the wheels do not come into contact with the ground. This setup was chosen because of friction and practicalities. More on this in paragraph 8.3.3.

The trajectory profile during this experiment is a *posref* with a step of 0.1 m in the positive x_L direction. No movement in the y_L and θ direction is to occur. This profile is equivalent to the one displayed in figure 15, but with the difference that the step there is 1.0 m in the x_L direction. From equations (3.12), it becomes clear that a 0.1 m step in the x_L direction corresponds with a response of wheel 2 and 3 of roughly -2.165 rad and 2.165 rad respectively. Note that we can use this profile for both position and velocity control because the trajectory processing software automatically calculates setpoints for angular position and velocity.

The control laws will be tuned as follows:

Table 3: Tuning of control laws (position control)

Control law	*K_p [-]*	*K_i [-]*	*K_d [-]*
P	90	-	-
PI	90	18	-
PID	90	18	0.5
PD	90	-	0.5

Table 4: Tuning of control laws (velocity control)

Control law	*K_p [-]*	*K_i [-]*	*K_d [-]*
P	20	-	-
PI	20	15	-
PID	20	15	0.5
PD	20	-	0.5

In this experiment, we are specifically interested in incremental performance gains that occur when we switch to a new control law. The difference in tuning (K_p, K_i and K_d values) between position and velocity control is therefore irrelevant.

The results of this experiment can be found in paragraph 7.2.1.

7.1.2: Tuning for performance requirements

Now that the most suitable control law has been selected, the controllers can be tuned specifically for the two sets of requirements that were introduced in paragraph 4.4: all-round and precise. In this experiment, the best possible tuning for these sets will be determined and the performance will be compared with the requirements. The position controller is fed a *posref* profile with a step of 0.1 m in the x_L direction, just like the previous experiment. The velocity controller, however, will use a *velref* profile with a step of 0.2 m/s in $\dot{x}_L$ and no motion in the other directions. This *velref* profile is roughly equivalent to the *posref* profile used before.

7.2: Results

7.2.1: Performance comparison of control laws

The first results that will be discussed are those from the experiment concerning the comparison of four different control laws. Each control law was examined three times. As discussed in the previous paragraph, for this *posref* profile, a step in x_L will result in responses for wheel 2 and 3. The first row quoted for every control law is the response for wheel 2 and the second row is the response of wheel 3.

Table 5: Results for P, PI, PID and PD with position control

Control law	*Settling time [s]*	*Overshoot [%]*	*Rise time [s]*	*Absolute ss error [rad]*	*Absolute ss error [°]*
P (n=3)	0.145	0.1	0.105	0.0211	1.2109
	0.145	0.0	0.100	0.0244	1.3979
PI (n=3)	0.140	1.6	0.100	0.0067	0.3864
	0.135	2.7	0.095	0.0026	0.1514
PID (n=3)	0.155	0.2	0.110	0.0136	0.7811
	0.155	0.1	0.110	0.0137	0.7857
PD (n=3)	0.155	0.0	0.110	0.0208	1.1937
	0.155	0.0	0.105	0.0143	0.8186

Table 5 shows the results for the implementation of the four control laws as position controllers. Note that all performance indicators in this chapter and the next are averaged over the number of experiments. All settling times and rise times are rounded to the nearest multiple of the sampling period. The absolute steady state errors are computed by first taking the sum of absolute values of all steady state errors during subsequent experiments and then dividing by the number of experiments. The absolute values are used to prevent steady state errors with opposite signs from canceling each other, because that would falsely lead to 'better' steady state errors.

The table shows that there are clear differences in performance. By adding an integrating action, an increase can be seen in the overshoot percentage and a decrease is noticeable in the steady state errors. The settling time and rise time show a mixed result. The transition from P to PI shows a marked decrease of these two parameters, while going from PD to PID shows no change and a slight increase, respectively. When adding a differentiating action, one observes an increase of settling time and rise time and a decrease of the overshoot. The steady state error shows mixed results.

The control laws were also implemented for velocity control. In table 6, the results of these experiments can be found. The rounding was done in the same way as for position control.

Table 6: Results for P, PI PID and PD with velocity control

Control law	Settling time [s]	Overshoot [%]	Rise time [s]	Absolute ss error [rad]	Absolute ss error [°]
P (n=3)	0.095	91.4	0.005	2.1367	122.4
	0.030	36.9	0.005	2.1420	122.7
PI (n=3)	4.180	0.0	3.200	0.4600	26.4
	5.355	0.0	3.690	0.3716	21.3
PID	-	-	-	-	-
PD	-	-	-	-	-

During the experiments, it became clear that velocity control is not suitable for tracking a step profile. For the P control law, the result is a fast response that doesn't reach the amplitude prescribed by the trajectory. The PI control law shows a very slow response that succeeds better at attaining the correct amplitude, although the position controller's performance is far superior. Adding a differentiating action introduces a lot of noise into the system because it amplifies deviations between trajectory and real velocities. It leads to a very jittery unstable response, even for small values of K_d and larger sampling periods, which is harmful for the electromotors.

Based on the results of the two experiments above, the PI control law was selected for further use. The following reasons were compelling to make this choice:

1. PI is the optimal control law for position control. Although the overshoot is slightly higher than for the other laws, it is still well below 5%. For velocity control, PI gives a slow but more accurate response than P.
2. There is a preference for the same control law for both position and velocity control.
3. Derivative feedback did not yield improvements and often makes the actuators saturate (control inputs *u* reach their limits). More on saturation in the next chapter.

7.2.2: Tuning for performance requirements

In this paragraph, it will be checked whether the controllers satisfy the requirements presented in chapter 4.

The result of the tuning phase is shown in the table below. In position control, one notices that the K_p and K_i values for the precise controller are higher than those for the all-round controller. This is expected considering that an as-small-as-possible steady state error will require a lot of control effort. It was not possible to tune the velocity controller to the precise set of requirements, since the necessary K_p and K_i values would lead to a very noisy, unstable response. An all-round velocity controller was developed, though.

Table 7: Tuning for performance – selected values for K_P and K_I

Control type	K_p [-]	K_i [-]
Position, all-round	90	18
Position, precise	200	50
Velocity, all-round	20	15
Velocity, precise	-	-

Now let's take a look at the results. In figure 21, 22 and 23, the step response of the system with all-round position control is shown. Figure 21 shows that ϕ_1 stays zero, while ϕ_2 and ϕ_3 show a reasonably fast step response, as we expected. Figure 22 and 23 provide extra information as they also show the amplitude of the tracking error and the desired (not limited) control input. At the instant of the step in the reference, the tracking error is largest but decreases quickly soon afterwards. One also notices that control inputs *u* stay in the band between -255 and 255.

The step responses of the precise position controller look very similar. One interesting thing can be seen in figure 24. Due to the shock of the step, it sometimes happens that the position of wheel 1 also changes very slightly. In the upper subplot of the figure we can see a close-up of exactly that. The wheel moves $4.55 \cdot 10^{-4}$ rad, or exactly 4 encoder counts.

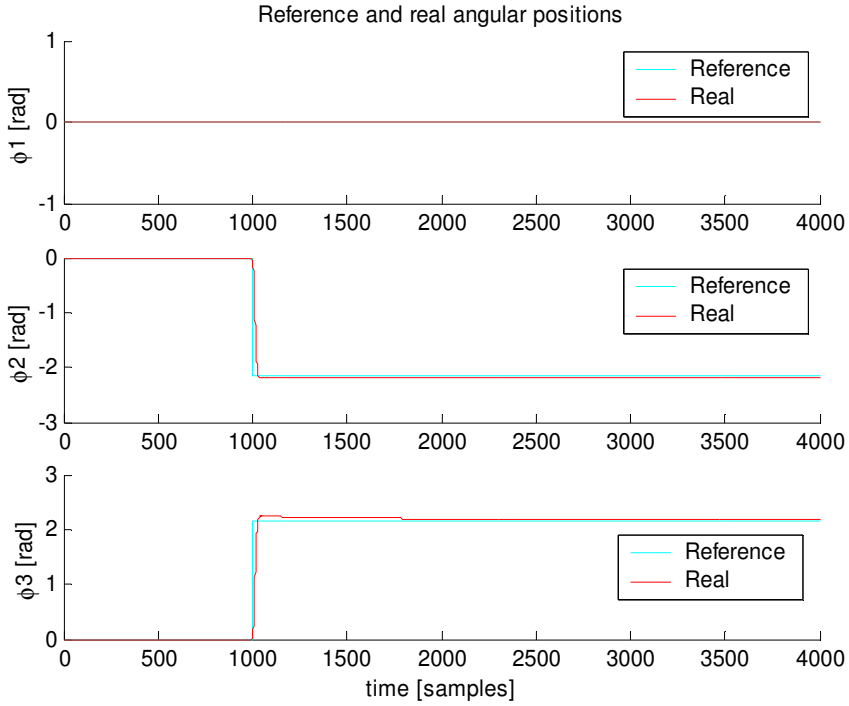

Figure 21: Reference and real positions (all-round position control)

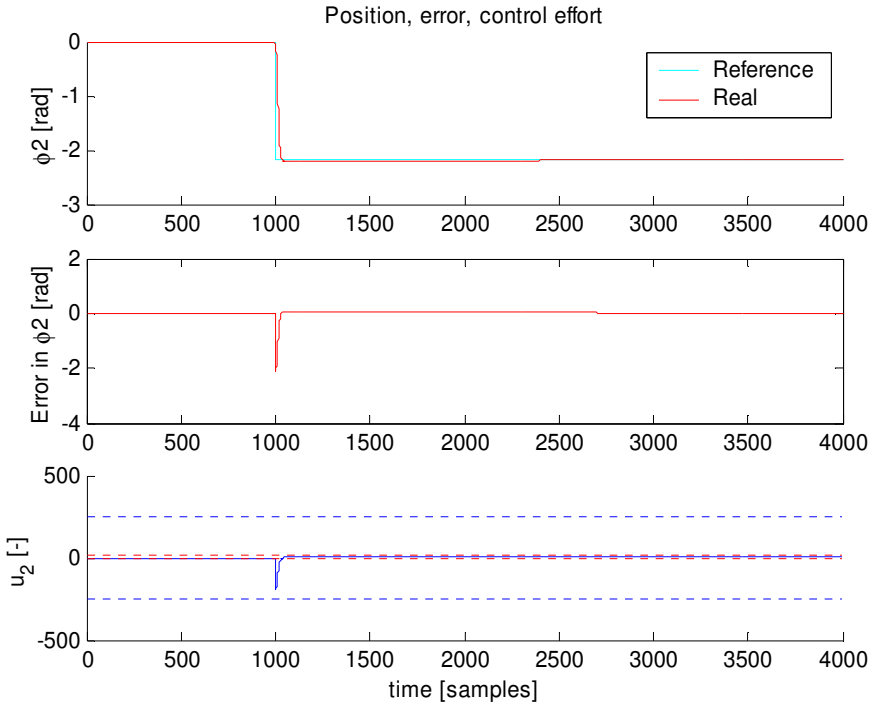

Figure 22: Position, tracking error and control input for wheel 2 (all-round position control). The blue dotted lines in the bottom graph represent -255 and +255.

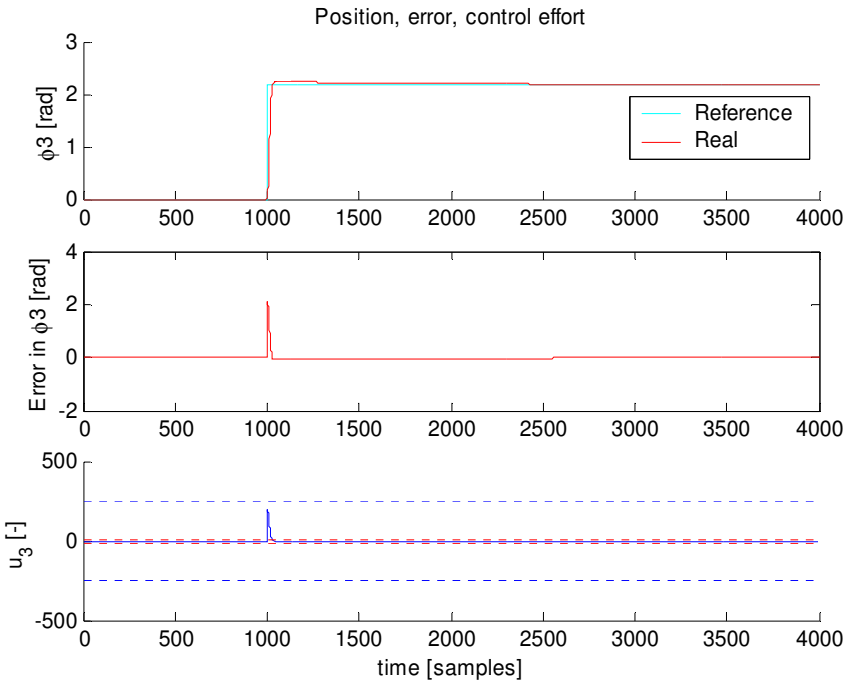

Figure 23: Position, tracking error and control input for wheel 3 (all-round position control). The blue dotted lines in the bottom graph represent -255 and +255.

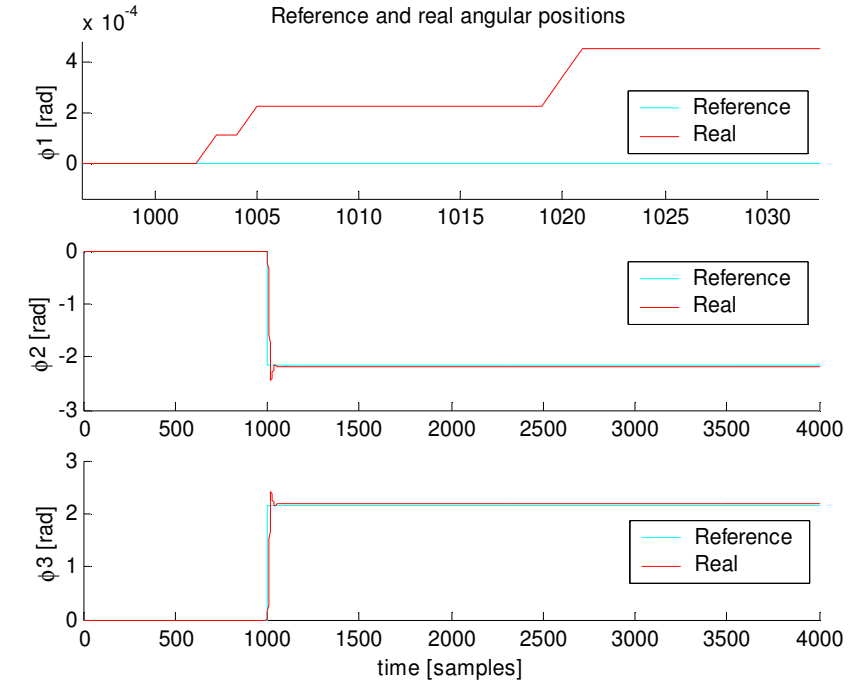

Figure 24: Reference and real positions (precise position control)

In figure 25 and 26, examples of step responses of the all-round velocity controller are presented. The results of this controller show a general step response with high-frequency components superpositioned on it. These components make it hard to determine the precise values of the performance indicators. Moreover, spikes can occur in the response and distort the values. Therefore, it is necessary to apply data filtering to the responses.

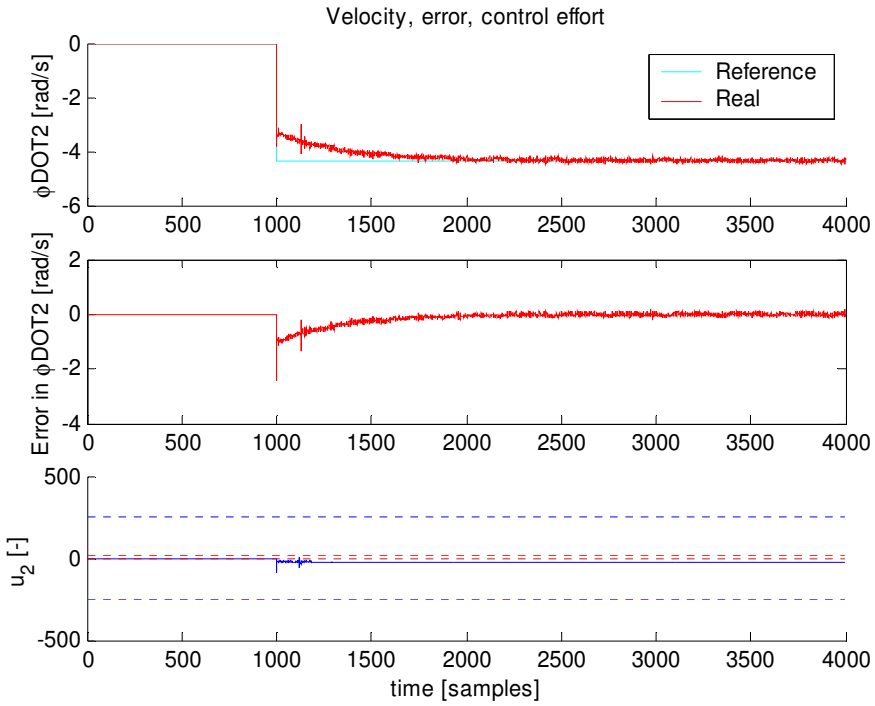

Figure 25: Velocity, tracking error and control input for wheel 2 (all-round velocity control). The blue dotted lines in the bottom graph represent -255 and +255.

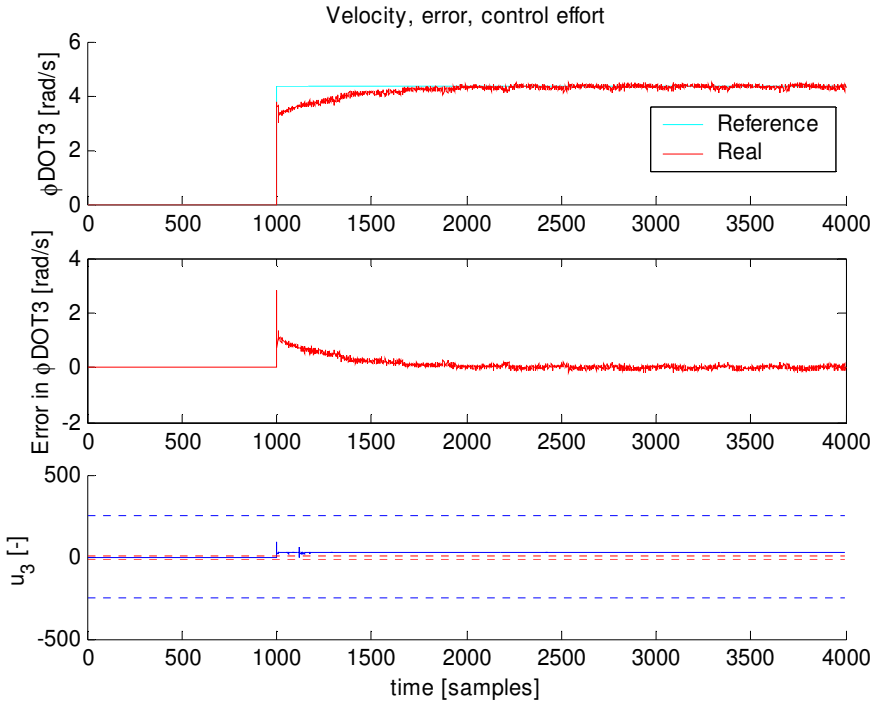

Figure 26: Velocity, tracking error and control input for wheel 3 (all-round velocity control). The blue dotted lines in the bottom graph represent -255 and +255.

In figure 27, which shows a step response of the velocity controller with spikes, the benefit of filtering is quite clearly visible.

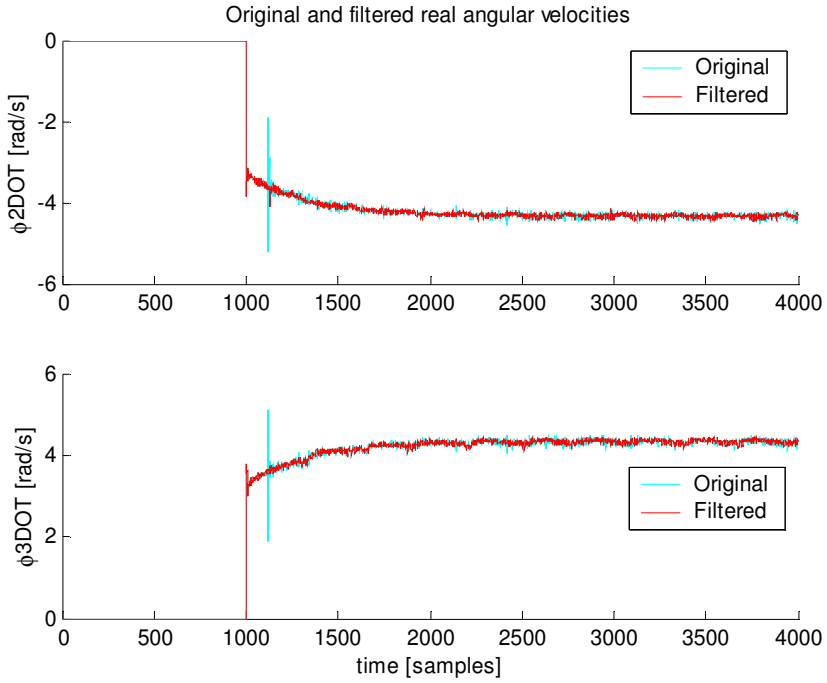

Figure 27: Original and filtered step response (all-round velocity control)

The filtering sequence is as follows:

1. Clipping of the signal at limits that do not affect the true signal. In this step, upper and lower bounds for the signal are defined. Spikes that exceed these bounds will be clipped: the part that exceeds the bound will be cut off and set equal to the violated bound.

2. The low-frequent trend of the signal is computed by applying a first order low-pass Butterworth filter to the signal using Matlab's filtfilt with the special property of zero phase distortion.

3. The detrended normalized response is computed by subtracting the trend from the clipped signal. This operation transforms the signal so that the trend becomes the zero value of the signal.

4. Despiking of the detrended signal by setting any signal value that exceeds the ± 2σ limits to zero, where σ is the standard deviation of the signal.

5. As last step the trend is restored by adding it to the filtered signal, yielding a filtered velocity response.

The overshoot is computed from the filtered response. The rise time is computed from the time instances where the response reaches 10% and 90% of the final value. This 10% value is based on the filtered response as the low-frequent trend does not represent the step in velocity well. The 90% value is based on the trend, though, as is the settling time.

For the steady state error we want to take the high frequent nature of the response into account because in the end this is what the user experiences when he uses the robot. Contrary to a step response of the position controller, no clear end values are visible for velocity control.

Therefore, a new error parameter is introduced for velocity control which resembles the idea behind the steady state error and will be used as equivalent performance indicator:

$$e_{500} = mean(abs(last500(error))) \quad (7.1)$$

e_{500} is the mean absolute error computed from the last 500 samples of the response [rad/s].

In tables 8 and 9, the results of the experiment are shown. In the first column labeled 'control type', the number of experiments that were performed are noted in round brackets. We can see that both the all-round and precise position controller satisfy their respective sets of requirements. Unfortunately, the all-round velocity controller does not satisfy the requirements. For the steady state error of the velocity control, the above-mentioned e_{500} is used. Although the e_{500} falls within specifications, the same cannot be said about the settling time, overshoot and rise time. The precise position control performs quite well with some instances where the absolute steady state error is only a few encoder counts. On average, the absolute steady state error is 40 to 50 counts.

Table 8: Results of 'tuning for performance' experiment (position control)

Control type	*Settling time [s]*	*Overshoot [%]*	*Rise time [s]*	*Absolute ss error [rad]*	*Absolute ss error [°]*
All-round					
Requirements	< 0.250	< 5.0	< 0.100 s	< 0.0175	< 1.0000
Position control (n=3)	0.175 0.135	2.6 3.2	0.095 0.095	0.0070 0.0091	0.4020 0.5212
Precise					
Requirements	Small	Small	Small	Small	<< 1.0000
Position control (n=3)	0.165 0.165	12.0 12.0	0.060 0.055	0.0053 0.0044	0,3037 0,2523

Table 9: Results of 'tuning for performance' experiment (velocity control)

Control type	*Settling time [s]*	*Overshoot [%]*	*Rise time [s]*	*e_{500} [rad/s]*	*e_{500} [rpm]*
All-round					
Requirements	< 0.250	< 5.0	< 0.100 s	< 0.1047	< 1.0000
Velocity control (n=4)	3.450 3.520	8.1 0.0	1.530 1.660	0.0448 0.0528	0.4280 0.5044

To summarize, one can say that position control satisfies the requirements whereas velocity control does not.

Chapter 8: Enhanced controllers

In the previous chapter, the performance of some basic controllers was investigated. By taking into account a number of non-linear effects, it might be possible to improve performance. First, several non-linear effects will be studied. Then, one of these effects will be selected and solutions for this phenomenon are implemented. Thereafter, experiments are performed, including one on the lab floor, and the results will be discussed.

8.1: Actuator saturation and other non-linearities

8.1.1 Actuator saturation

In control, one often has to deal with the fact that the allowable values of the control input *u(t)* are limited. This is also true for the omnidirectional robot. There is a constraint on the maximum angular velocity the electromotors, the actuators of the system, can provide. The motors are unable to go faster than 45 rad/s in any direction.

This limitation can lead to performance, which is worse than the situation with no limits, for two reasons:

1. The response-times of the system are usually longer, since the actual inputs are generally smaller.
2. The control strategy does not take the limitation into account and has poor behaviour at instances where the inputs are limited.

One can think of the actuator limitation as a saturating element between the controller *C(t)* and the system *P(t)*, which limits the control action, regardless of the controller's output. Or in other words, although the controller asks for a desired control input $u_{des}(t)$, the actuator can only deliver a constrained *u(t)*, see figure 28.

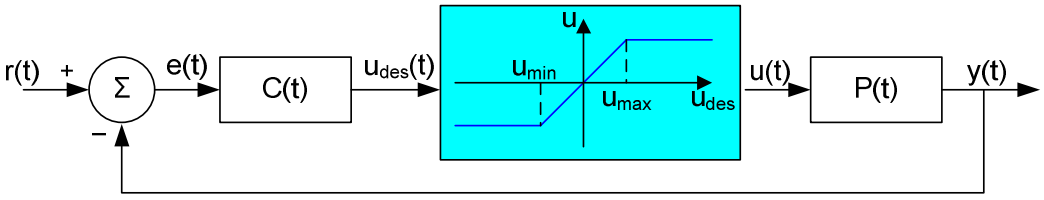

Figure 28: Feedback control system with saturating actuators

The behaviour of the saturating element can be written as:

$$\begin{cases} u_{des}(t) \geq u_{\max} & \rightarrow \quad u(t) = u_{\max} \\ u_{\min} < u_{des}(t) < u_{\max} & \rightarrow \quad u(t) = u_{des}(t) \\ u_{des}(t) \leq u_{\min} & \rightarrow \quad u(t) = u_{\min} \end{cases} \tag{8.1}$$

Here u_{max} and u_{min} are the voltages corresponding to the maximum angular velocities in positive (+255) and negative direction (-255), respectively.

From literature [4], it is known that when saturation of the actuators occurs, a normal PI controller will keep integrating and the $u_{des}(t)$ continues to increase even though this has no benefit. After all, the actuators are already providing what they can. Moreover, this charge caused by *winding up* of the integrator must be removed later, resulting in a substantial overshoot and waste of valuable time to get *u(t)* back to a useful level.

8.1.2 Deadband

Another non-linear effect that manifests itself on the robot is *deadband*. We use this term to describe the effect that for small control input values *u*, corresponding with small non-zero rotational velocities, no rotation of the wheels occurs because of friction. A quick investigation was undertaken to learn more about the deadband effect and it was learnt that the boundaries – the values *u* that were just on the boundary between stick/slip and rotation – varied among wheels, rotation directions and successive experiments. This makes deadband a complex problem to solve and addressing it falls outside the scope of this project.

8.2: Solutions for actuator saturation

From the two non-linear effects described in the previous paragraphs, actuator saturation was selected to be addressed. Below, two methods will be introduced to improve performance when actuator saturation occurs.

8.2.1: Integrator anti-windup

Resolving the winding up effect of the integrator is relatively straightforward. A strategy called *integrator anti-windup* will be implemented. The basis of this strategy is to stop integrating when we detect actuator saturation and thus improve the overshoot and response times.

This is implemented as follows (see page 23 for the implementation without anti-windup):

```
double umax = 255.0, umin = -255.0;
...
// PI control with anti-windup
if ((u1 < umax) && (u1 > umin)) e_total1 = e_total1 + e_new1;
if ((u2 < umax) && (u2 > umin)) e_total2 = e_total2 + e_new2;
if ((u3 < umax) && (u3 > umin)) e_total3 = e_total3 + e_new3;
u1 = Kp*e_new1 + Ki*delta_t*(e_total1);
u2 = Kp*e_new2 + Ki*delta_t*(e_total2);
u3 = Kp*e_new3 + Ki*delta_t*(e_total3);
```

The controller checks if the desired control inputs u_1, u_2, u_3 fall outside of the limits set by u_{max} and u_{min}. If this is so, no integration will be performed (e_{total} is not changed). If the control inputs fall inside the limits, integration is allowed as normal (e_{total} is changed).

Note that this is different from equation (6.2), because there the limits of the actuator are acknowledged, but the control logic itself is not adapted to overcome the negative implications of these limits.

Integrator anti-windup can be added to enhance both position control and velocity control. The expanded main programs are called *antiPos.c* and *antiVel.c*, respectively. In paragraph 8.3, a number of experiments with anti-windup will be discussed.

8.2.2: Saturation prevention

Another strategy to deal with actuator saturation is *saturation prevention*.

To derive an algorithm for saturation prevention, let us start with a slightly modified control law for PI position control:

$$u = K_p(x_d - x) - K_d\dot{x} \qquad (8.2)$$

Here x_d is the desired position [rad] and x the real position [rad]. Note that the derivative term uses the derivative of the real position, instead of the derivative of the error.

We can rewrite this equation as follows:

$$u = K_d\left(\left[\frac{K_p}{K_d}(x_d - x)\right] - \dot{x}\right) \tag{8.3}$$

The term in straight brackets will be called $\dot{x}_d$, the derivative of the desired position or *desired velocity* [rad/s], even though strictly speaking it is not the same as the real desired velocity.

A saturation parameter η [-] is introduced as a gain for $\dot{x}_d$ and (8.3) is rewritten as:

$$u = K_d\left(\eta\,\dot{x}_d - \dot{x}\right) \tag{8.4}$$

with:

$$\begin{cases} \dot{x}_d > \dot{x}_{\max} & \rightarrow \quad \eta = \dfrac{\dot{x}_{\max}}{\dot{x}_d} \\ \dot{x}_d \le \dot{x}_{\max} & \rightarrow \quad \eta = 1 \end{cases} \tag{8.5}$$

Here $\dot{x}_{\max}$ is the maximum attainable angular velocity of the wheels in either rotation direction [rad/s].

One could say that the position control law is rewritten in a special kind of velocity control law. Actuator saturation is defined as the moment when the desired velocity exceeds the maximum. Essentially, when saturation occurs, the desired angular velocity term in (8.4) is replaced with the maximum possible angular velocity. In all other cases the desired angular velocity is left in peace.

This can be implemented as follows (only shown for wheel 1):

```
double phidot_max = 45.0;
...
// PI control with saturation prevention
if (phidot_des1 > phidot_max) {
    eta1 = phidot_max / phidot_des1;
} else if (phidot_des1 < -phidot_max) {
    eta1 = -phidot_max / phidot_des1;
} else {
    eta1 = 1.0;
}
...
u1 = Kd*((eta1*phidot_des1) - phidot1);
...
```

A small number of experiments were performed with this strategy but the results did not conform to expectations. It is recommended that both the concept and implementation introduced above are checked thoroughly and possibly altered before use.

8.3: Experiments

8.3.1: Anti-windup: comparison with basic controllers

From the structure of the anti-windup implementation, it can be easily understood that a PI or PID controller using this strategy will perform as well as a normal PI or PID controller, in the absence of actuator saturation. This was also verified with a number of experiments, which will not be discussed in this report. Of course, when saturation occurs, we expect differences.

To determine whether anti-windup enhanced controllers really perform better than basic controllers when facing actuator saturation, an experiment has to be set up.

For position control, two different *posref* profiles are used with a step of 1.0 m and 2.0 m in x_L direction, respectively, and no movement in the other directions. These profiles require the robot to instantly move up to 2 meters. This will lead to saturation of the actuators because such an instantaneous move is physically impossible; it takes some time for the robot to perform such movement. This provides an ideal way to study actuator saturation. The two profiles will first be supplied to a basic PI position controller and then to a PI position controller with anti-windup. Both are tuned to the all-round set of requirements: $K_p = 90$, $K_i = 18$.

It is also interesting to see the results for velocity control. Therefore, the performance of a basic PI velocity controller and a PI velocity controller with anti-windup are also compared. The all-round set of requirements is used for both: $K_p = 20$, $K_i = 15$. These controllers are fed two *velref* profiles with a step of 2.0 m/s and 4.0 m/s in x_L direction.

Once again, the experiment is performed frictionless.

8.3.2: Anti-windup: using a 3rd order profile

Step responses provide a wealth of information on a system and this was the main reason to use them up until now. It is also interesting, however, to use other kinds of trajectory profiles. In this experiment, we will use a double-sided 3rd order profile and see how well our enhanced controllers can keep track of it.

In essence, a 3rd order profile is a very smooth position trajectory. This is accomplished by the absence of discontinuous steps in the acceleration profile from which it is generated. In fact, the acceleration profile only contains zero or linear slopes and is therefore of first order. The position profile, after two integrations, is thus of third order. The 3rd order profile contains a smooth acceleration phase, followed by a constant-velocity phase and finally a smooth deceleration phase. The double-sided 3rd order profile is created by fusing a normal and a mirrored 3rd order profile together. In this way, the wheel will return to its original starting position at the end of the experiment.

The profile is supplied as *posref* profile (see figure 29) and has the double-sided 3rd order function for x_L. No movement in y_L and θ is to occur. We expect a response for wheel 2 and 3.

As controllers, we will first use an *antiPos* and then an *antiVel* controller and the *posref* profile is supplied to both. The total length of the experiment is 40 seconds. It will be performed frictionless.

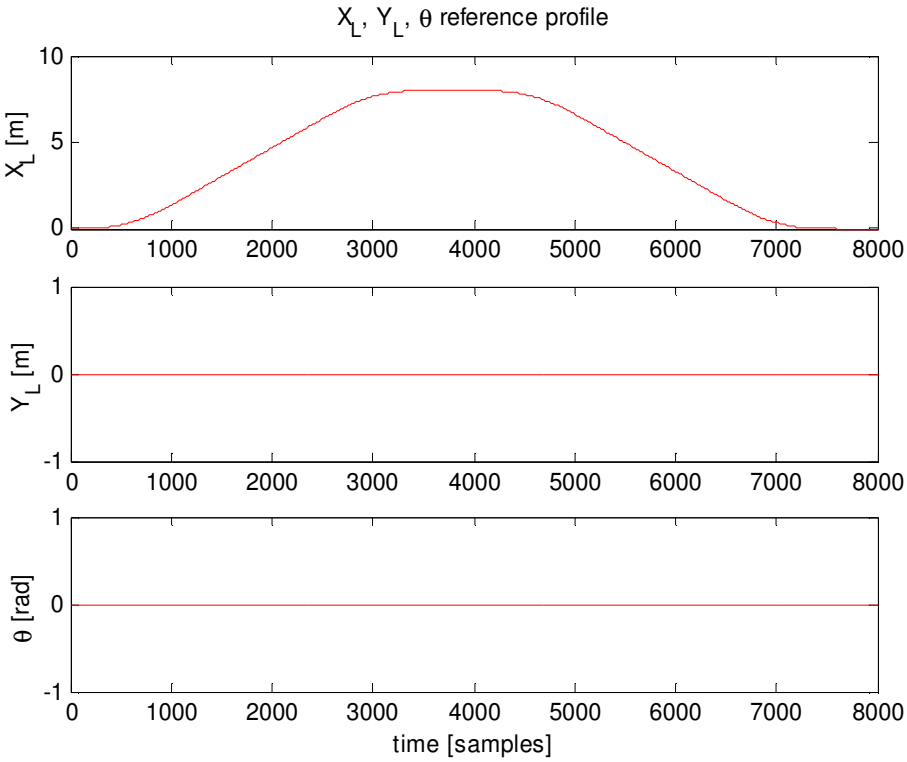

Figure 29: Double-sided 3rd order profile (posref)

8.3.3: Anti-windup: experiment on lab floor

Up until now, all experiments were performed frictionless. The motivation for this was threefold. First of all, performance comparison is more straightforward if rolling friction is kept out of the experiments. Secondly, the robot is currently supplied with power by stationary voltage sources through power cables instead of batteries. This set-up creates challenges when it comes to testing because the cables put limitations on the robot's movement. Thirdly, wireless communication with the robot, which makes ground experiments much more convenient, was established relatively late in the project. Therefore, the number of ground experiments are limited.

Nevertheless, the omnidirectional robot was designed as testing bed for new algorithms and its operating environment consists of the laboratory floor. Keeping this in mind, it is important to also test performance in that environment.

In this last experiment, a more elaborate maneuver will be performed than anything up until now. Figure 30 shows the letter L trajectory profile. Firstly, the robot has to move 2 meters in the positive x_L direction with a constant velocity of 0.2 m/s. This is then followed by a translation of 1 meter in the positive y_L direction with a constant velocity 0.2 m/s. Next, the robot will move back 1 meter in the negative y_L direction to arrive at the previous waypoint. Finally, the robot will remain stationary for 5 seconds. The total length of the experiment is 25 seconds.

This experiment will be performed solely using a precise PI position controller with anti-windup ($K_p = 200$, $K_i = 50$).

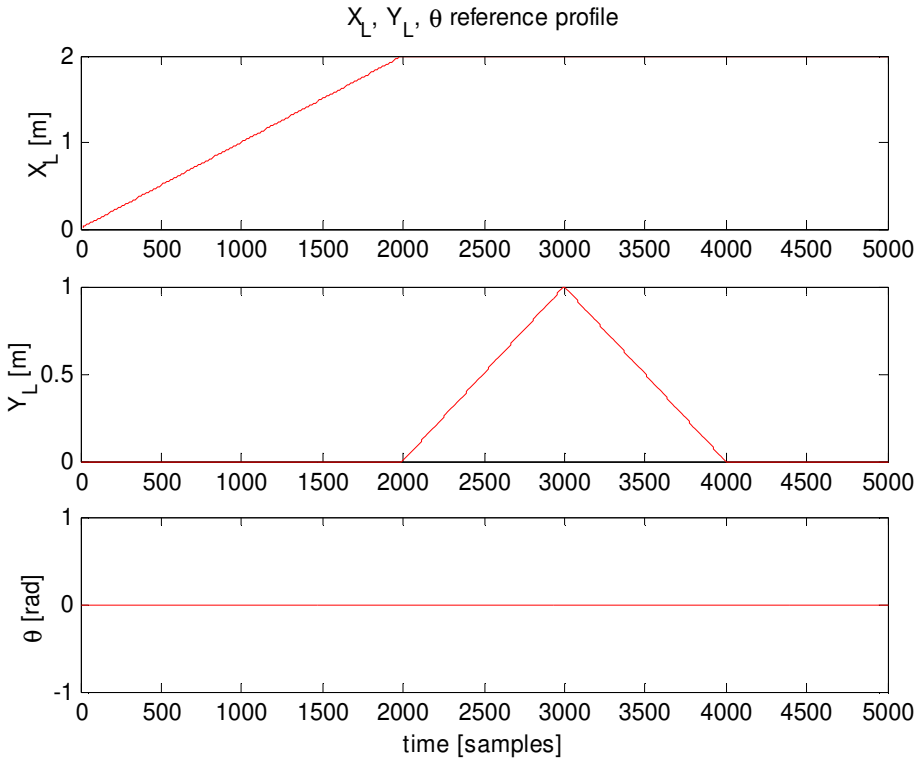

Figure 30: *Posref* trajectory profile for floor experiment

8.4: Results

8.4.1: Anti-windup: comparison with basic controllers

In this paragraph, the results of the comparison between basic controllers and those enhanced with anti-windup will be discussed. Two profiles with differing step amplitudes were supplied to the controllers to see if performance and the amount of saturation are related. First, let us discuss the results of this experiment when using position control (see table 10).

Table 10: Comparison of basic and enhanced controllers experiencing actuator saturation (position control)

Experiment type	*Settling time [s]*	*Overshoot [%]*	*Rise time [s]*	*Absolute ss error [rad]*	*Absolute ss error [°]*
Step 1.0 m					
Without a.w.	2.065	5.7	0.400	0.0740	4.2387
(n=5)	2.075	5.6	0.400	0.0659	3.7769
With a.w.	0.510	0.5	0.395	0.0052	0.3005
(n=5)	0.510	0.6	0.395	0.0112	0.6406
Step 2.0 m					
Without a.w.	5.175	9.9	0.795	0.2851	16.3339
(n=5)	5.160	9.9	0.795	0.2732	15.6521
With a.w.	0.975	0.4	0.790	0.0065	0.3707
(n=5)	0.975	0.3	0.790	0.0073	0.4167

In the first column labeled 'experiment type', we use the abbreviation *a.w.* for anti-windup. The number of experiments that were performed are noted in round brackets.

From the results, one can conclude that anti-windup substantially improves the performance of the controllers during actuator saturation. The settling times, overshoots and steady state

errors show a dramatic decrease. The decrease in rise time is much smaller, but still significant. From the results, we also learn that the performance improvement is not linear in the amount of saturation. The performance improvement of adding anti-windup when using a 2.0 m step is larger in a relative sense than when using a 1.0 m. This is especially striking when observing the overshoot.

In table 11, the results for velocity control are shown. As before, we use the e_{500} performance indicator instead of the steady state error. Unfortunately, for velocity control, we must conclude that there's no significant performance improvement when anti-windup is added. This does not invalidate the smarter controller usage of the anti-windup strategy, however.

Table 11: Comparison of basic and enhanced controllers experiencing actuator saturation (velocity control)

Experiment type	*Settling time [s]*	*Overshoot [%]*	*Rise time [s]*	*e_{500} [rad/s]*	*e_{500} [rpm]*
Step 2.0 m/s					
Without a.w.	2.935	0.0	1.305	0.2638	2.5191
(n=2)	3.070	0.0	1.345	0.2803	2.6767
With a.w.	3.120	0.0	1.435	0.3921	3.7438
(n=2)	2.960	0.0	1.370	0.4080	3.8956
Step 4.0 m/s					
Without a.w.	0.105	0.4	0.070	42.3082	404.0135
(n=2)	0.140	0.0	0.060	42.3611	404.5182
With a.w.	0.130	0.0	0.095	41.8388	399.5306
(n=2)	1.950	2.0	0.085	42.7128	407.8772

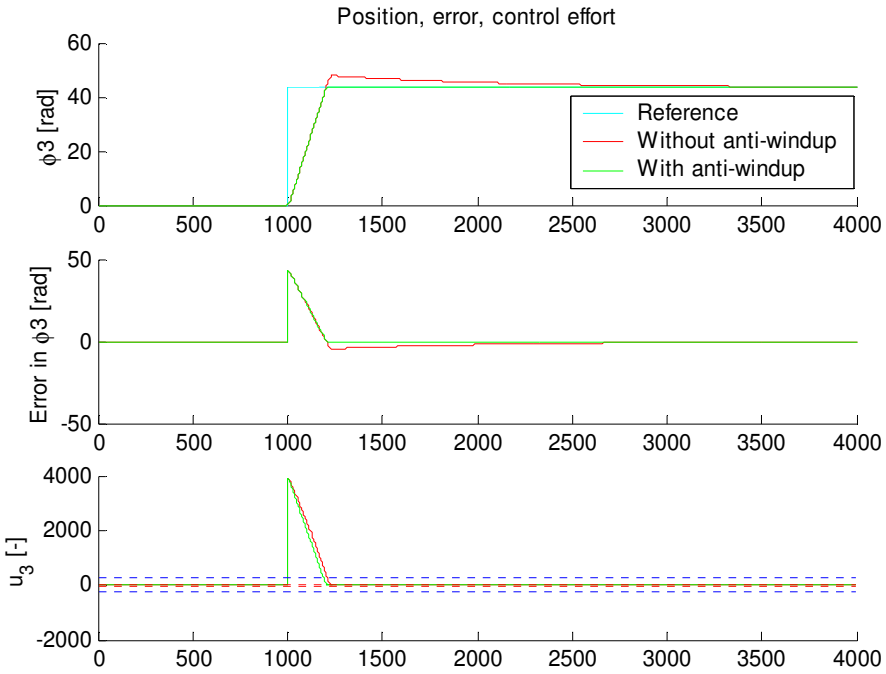

Figure 31: Comparison of position, tracking error and control input for wheel 3 without (red) and with anti-windup (green). The blue dotted lines in the bottom graph represent -255 and +255.

In figure 31, a representative graph is shown that depicts the performance of a basic and enhanced PI position controller during saturation. Note that the bottom subplot shows u_{des}, the desired control input, which exceeds the actuator limits as we expect.

8.4.2: Anti-windup: using a 3rd order profile

In this paragraph, the tracking of a 3rd order profile will be discussed. Since another setpoint trajectory is used, one can imagine that the original tuning for a step profile will not perform as well here. Therefore, new combinations of K_p and K_i values were examined. In the end, a new optimal pair was selected for the enhanced PI position controller: $K_p = 1000$, $K_i = 400$. This pair performed significantly better than the original tuning. For the enhanced PI velocity controller, no new tuning was selected because values exceeding $K_p = 20$, $K_i = 15$ lead to a bad response.

In tables 12 and 13, the results for position and velocity control are shown. The experiment has been performed five times for each controller. Note that for velocity control, numerical integration was used to arrive at results in "the position domain".

Since we are not dealing with a step input, the only performance indicator we can use is the error. We are not really interested in the final steady state error, but rather the tracking error during the maneuver.

For this experiment, two new ways of quantifying tracking error were introduced:

$$e_{mean} = mean(abs(error)) \tag{8.6}$$

$$e_{\max} = \max(abs(error)) \tag{8.7}$$

e_{mean} is the mean value of the absolute values of the tracking error during an experiment [rad]. e_{max} is the maximum value among the absolute values of the tracking error during an experiment [rad].

Table 12: Results of tracking a 3rd order profile (position control)

Wheel	Mean absolute error [rad]	Mean absolute error [°]	Max absolute error [rad]	Max absolute error [°]
2 (n=5)	0.0423	2.4259	0.0899	5.1497
3 (n=5)	0.0426	2.4397	0.0931	5.3365

Table 13: Results of tracking a 3rd order profile (velocity control)

Wheel	Mean absolute error [rad]	Mean absolute error [°]	Max absolute error [rad]	Max absolute error [°]
2 (n=5)	3.2985	188.9878	5.4305	311.1459
3 (n=5)	3.2909	188.5558	5.3987	309.3227

From the tables above, we can conclude that the mean tracking error for position control during the experiment was about 0.042 rad or roughly 2.4°. The maximum tracking error was about 0.093 rad or roughly 5.3°. When one considers that the profile requires relatively low velocities and thus experiences a relatively large influence from friction and deadband, this is a good result.

The results for velocity control are very different, even though we supplied the same input trajectory. The average tracking error for velocity control during the experiment was about 3.29 rad or roughly 189°. The maximum tracking error was about 5.43 rad or roughly 311°. Thus, on average, the velocity controller lags behind half a wheel rotation, with nearly a full wheel rotation in the worst case. We can conclude that position control is far superior to velocity control when it comes to controlling the robot.

In figures 32 and 33, representative responses for position and velocity control are shown.

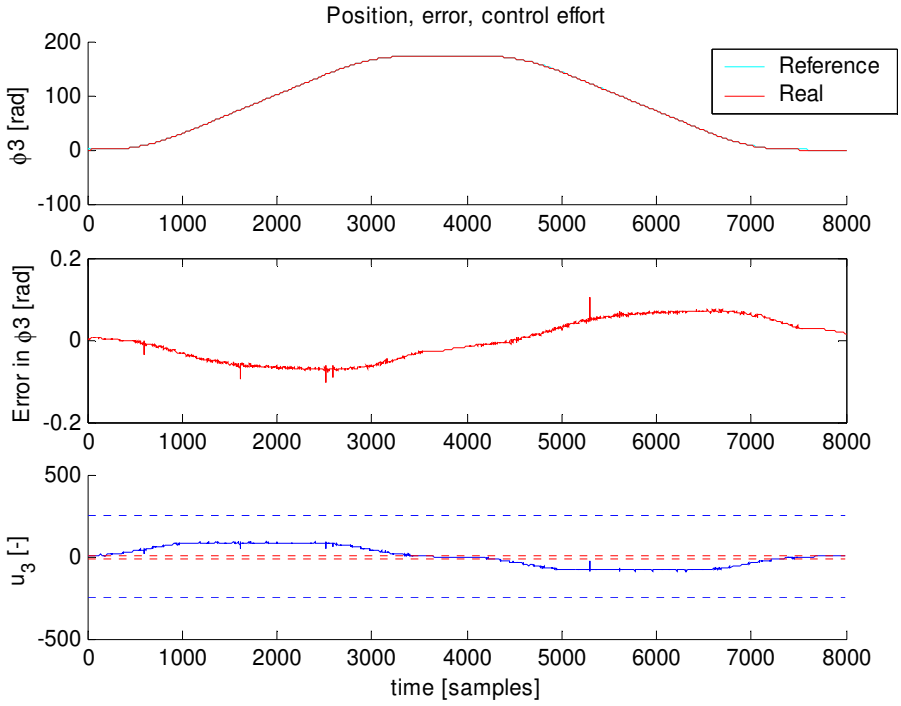

Figure 32: Position, tracking error and control input for position control (3rd order profile). The blue dotted lines in the bottom graph represent -255 and +255.

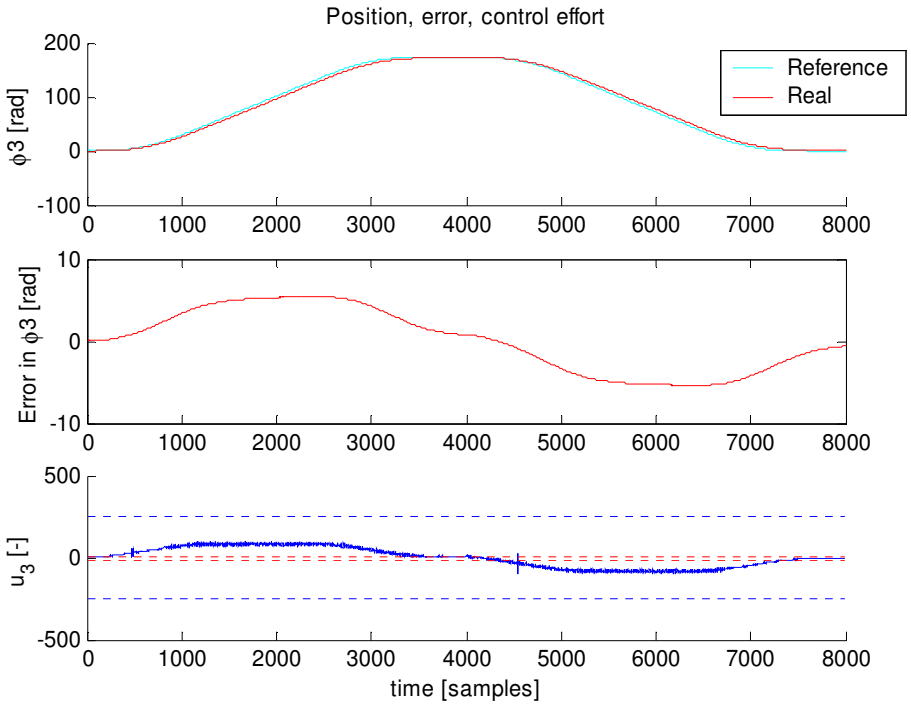

Figure 33: Position, tracking error and control input for velocity control (3rd order profile). The blue dotted lines in the bottom graph represent -255 and +255.

8.4.3: Anti-windup: experiment on lab floor

The ground experiment was performed four times to get significant results. The reference profile contains the x_L and y_L coordinates and therefore all three wheels are necessary to execute the task successfully.

We will once again look at the tracking error during the experiment: the mean absolute error and the maximum absolute error. In table 14, the results of this experiment are shown.

Table 14: Results of floor experiment

Wheel (n=4)	*Mean absolute error [rad]*	*Mean absolute error [°]*	*Max absolute error [rad]*	*Max absolute error [°]*
1	0.1588	9.1000	2.1566	123.6
2	0.1140	6.5332	0.6696	38.4
3	0.1537	8.8078	2.0499	117.5

One notices that the errors exceed the 1° mark, but please remember that this requirement was set for a response to a step input in a frictionless environment. All in all, a mean absolute error of up to roughly 9° is quite good. Moreover, the user now has an idea of the capabilities of the omnidirectional robot. With further research and smarter controllers, it is certainly possible to improve on these values.

In the following three figures, representative examples of the responses of the three wheels are shown.

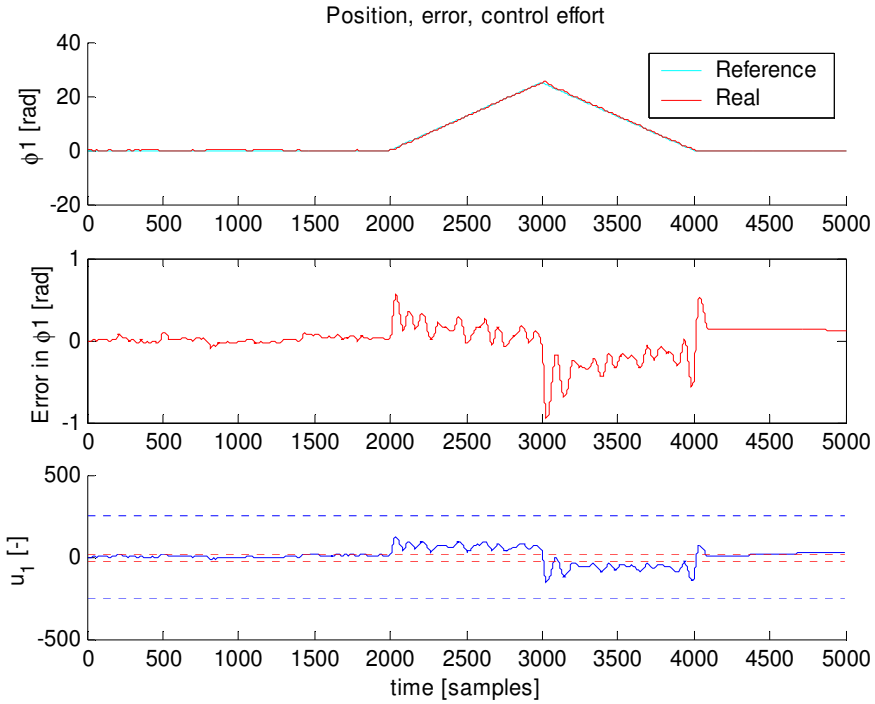

Figure 34: Position, tracking error and control input for wheel 1 (ground experiment). The blue dotted lines in the bottom graph represent -255 and +255.

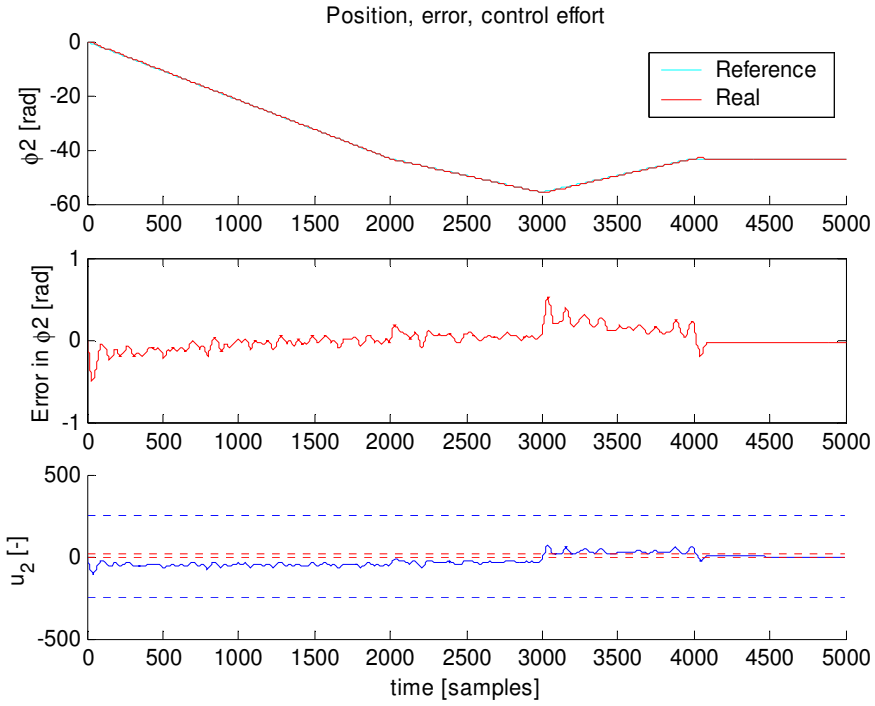

Figure 35: Position, tracking error and control input for wheel 2 (ground experiment). The blue dotted lines in the bottom graph represent -255 and +255.

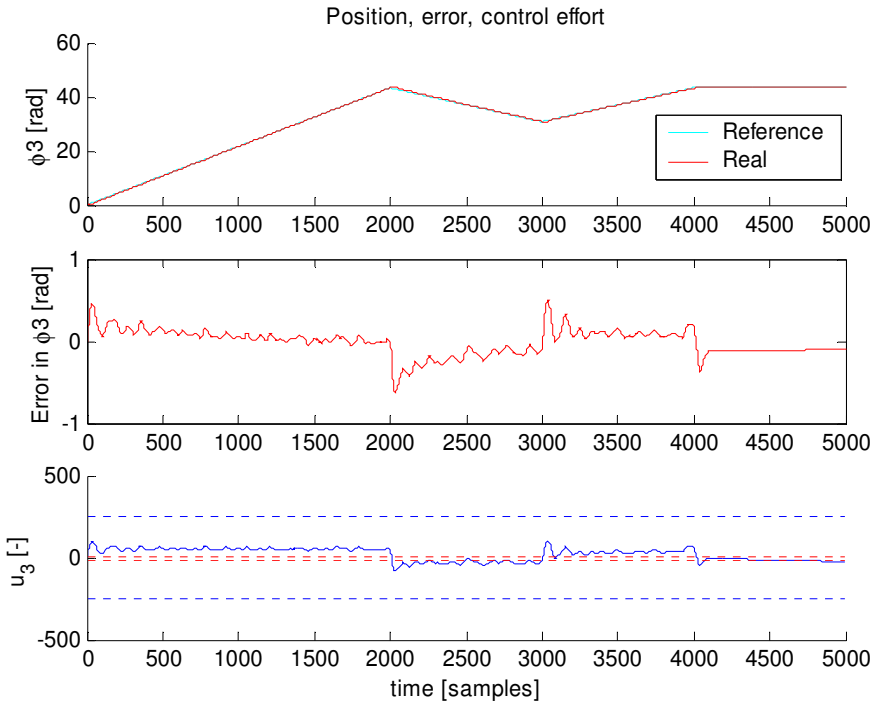

Figure 36: Position, tracking error and control input for wheel 3 (ground experiment). The blue dotted lines in the bottom graph represent -255 and +255.

Conclusions

In this report, the omnidirectional robot at NUS' Control and Mechatronics laboratory was introduced. Its systems and kinematics were described and the development and implementation of planar motion control algorithms were discussed. A new way to specify desired movement was created. Several experiments were performed to determine the time-domain performance of the motion controllers under a variety of conditions. Two non-linear performance-degrading effects were identified and strategies to mitigate them were proposed. Finally, the robot itself was modified during this project.

A number of conclusions can be drawn in regards to the above:

- The omnidirectional robot was adapted for wireless control via the SSH protocol.
- A library of low-level functions was created to simplify access to basic functionality of the robot. It can be easily built upon with high-level algorithms for localization, trajectory planning and tracking.
- Position control is perfectly suitable to control the robot. Velocity control is not recommended because of its significantly lower performance in time-domain.
- Of the four feedback control laws examined, proportional-integral (PI) control yielded the best results.
- Adding anti-windup to a controller with integrating action can significantly improve time-domain performance in light of actuator saturation.

References

[1] Wikipedia – “Robot” article: http://en.wikipedia.org/wiki/Robot

[2] van Haendel, R.P.A., 2005. Design of an omnidirectional universal mobile platform. Internal report DCT 2005.117. Eindhoven University of Technology, the Netherlands.

[3] Wikipedia – “Pulse-width modulation” article:
http://en.wikipedia.org/wiki/Pulse-width_modulation

[4] Franklin, G.F., Powell, J.D., Emami-Naeini, A., 1994. Feedback control of dynamic systems – third edition. Addison-Wesley Publishing Company, Reading, p. 196-200, 604.

[5] Wikipedia – “Rectangle method”: http://en.wikipedia.org/wiki/Rectangle_method

Appendix 1: Encoder count verification

During the development of control algorithms for the omnidirectional robot, it was observed that the real number of encoder counts per revolution of a wheel shaft did not exactly match the nominal encoder resolution that one would expect from the specifications: 56000 lines/revolution.

A visual marker was placed on the wheel shafts and a full number of rotations were executed. The end positions of the wheel shafts after rotation did not correspond with the marker, but differed significantly from the intended positions. This was a surprising find since these experiments were performed in a "frictionless" environment with no contact between wheels and floor. Furthermore, it was observed that when the number of executed rotations was increased, this also yielded a larger difference between intended and real end position. It was obvious that such a diverging error would degrade performance.

It was suspected that this situation had something to do with the encoder resolution. Subsequently, an experiment was devised to verify the nominal encoder resolution or quantify the correct number of encoder counts per revolution. Using the nominal encoder resolution (56000 lines/rev.), the wheels were rotated manually with extreme care until ten full rotations had been executed. This experiment was repeated to get a statistically relevant result.

Table 15: Encoder count verification

Experiment	*Measured displacement [rad]*
1	61.916393
2	61.917201

From mathematics, it is known that 10 rotations should give exactly 20π radians, which is clearly not the case here. Fortunately, with this experimental data, the real encoder resolution can be calculated:

$$\text{Real resolution} = \frac{measured\ displacement}{20\pi} \cdot \text{nominal resolution} \qquad \text{(A.1)}$$

Using equation (A.1) and substituting all values, it follows that the real encoder resolution is 55184 encoder counts/revolution.

This real encoder resolution was implemented and the experiment was repeated several times. No deviations between desired and real positions could be detected in any subsequent experiments.

It is challenging to find the reason behind this situation. Optical encoders are known for their accuracy and bitwise errors are unlikely. A mechanical explanation seems more probable. One of the possible reasons could be that the gearbox's reduction ratio might not be exactly 14:1 in reality, either due to non-linear effects or manufacturing imperfections. A lower reduction ratio would result in a reduction of the number of encoder counts/revolution. Finding the cause of the deviation falls out of the scope of this project, however.