Kinematics, analysis and control of a powered caster vehicle

TU/e technische universiteit eindhoven

Department Mechanical Engineering
Dynamics and Control Technology

DCT 2002.24

Kinematics, analysis and control of a powered caster vehicle

Students
D. van den Bercken
J.F. van den Eerenbeemt

Supervisors:
Dr. Marcelo H. Ang Jr.
Prof.dr.ir. M. Steinbuch

Singapore, 27-02-2002

Table of contents

Summary

Mobile manipulator systems can be used for many applications including assembly, transportation, and inspection. A mobile manipulator has to be integrated on a mobile base, which is driven by several wheels. For smooth motion, this mobile base must be able to move free in any direction, it must have three degrees of freedom of motion in the plane. A mobile robot with this property is called a holonomic mobile robot.

The ordinary mobile robots are all non-holonomic: they only have two degrees of freedom of motion in the plane. There are many different mechanisms that can be used to achieve holonomic motion. In this report, a mobile base with caster wheels (also called office chair wheels) will be discussed.

This mobile base has several wheel modules. Each wheel module has two DC motors: one for driving and one for steering. To translate the base velocity to wheel velocities, a kinematical model of the base is needed. This model is derived in chapter 2.

To achieve the desirable velocity, two controllers are needed, one for the steering velocity and one for the driving velocity. The design and implementation of the controller is discussed in chapter 4 and chapter 5.

1. Overview of problem

1.1 Design

The mobile base consists of two parts: a platform and several wheel modules. A wheel can be actuated (active) or passive. When a wheel is active, it is powered by two DC motors, one for steering and one for driving. In figure 1.1, a caster wheel is shown.

The shape of the platform and the number of active and passive wheels depend on the application the base is used for. The same goes for the offset (b in figure 1.1), the radius of the wheel and the position of the several wheels on the platform.

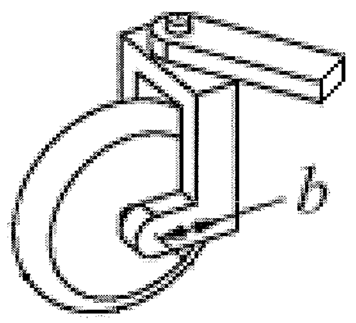


Figure 1.1: caster wheel

An active wheel is placed in a wheel module with two DC motors, two encoders and two amplifiers.

One wheel module was already designed, produced and available for tests.

1.2 Control problem

There are two levels in the control problem: task level and wheel level.
The figure below gives an overview of the control on task level.

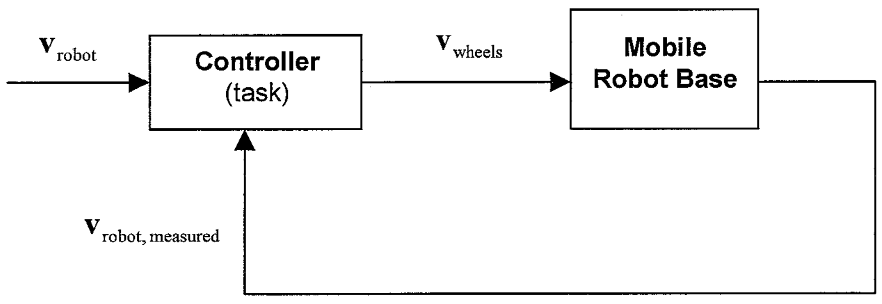


Figure 1.2: Control on task level

The mobile base must be able to follow prescribed trajectories, called tasks. A task is specified in terms of the velocity vector of the robot base as a function of time. This task must be translated into corresponding steering and driving velocities for the several wheels.

To realize the desired wheel velocities, a controller has to be designed for each DC motor. In figure 1.3 the basic control loop for the wheel velocities is shown schematically.

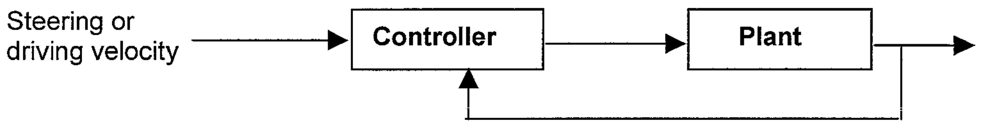


Figure 1.3: Control on wheel level

For each wheel velocity (steering and driving) a different controller is needed.

Because slip between the actuated wheels and the ground may occur, a second controller (figure 1.2) is necessary to guarantee the desired path of the base is followed. Therefore an independent measurement of the robot velocity is needed. This can be done with an optical sensor for example. Using the error between the actual and desired velocity of the robot base, the controller in figure 1.2 corrects the specified task at specific time intervals. This is done at a much lower frequency than the control on wheel level.

1.3 Main objective

The research described in this report consists of two parts:

- Deriving a kinematical model that describes the relationship between robot velocities and wheel velocities.
- Design of a controller for the steering and driving velocity of one wheel.

The kinematical modeling is described in chapter 2. Chapter 3 handles the used hardware and software and in chapter 4 a dynamical model of the system is derived. Finally, in chapter 5, the implementation and tuning of the controller is discussed.

2. Kinematical model

As mentioned in chapter 1, the task of the mobile base is given in terms of movement of a point on that base. To translate this task to wheel velocities, a kinematical model of the mobile base is needed. The derivation of this model is presented in this chapter.

2.1 Mobile base

In figure 2.1 an open-chain mechanism of the caster wheel is given. It has three degrees of freedom, called wheel variables: steer $\dot{\varphi}$, drive $\dot{\rho}$ and twist at the wheel contact $\dot{\sigma}$.

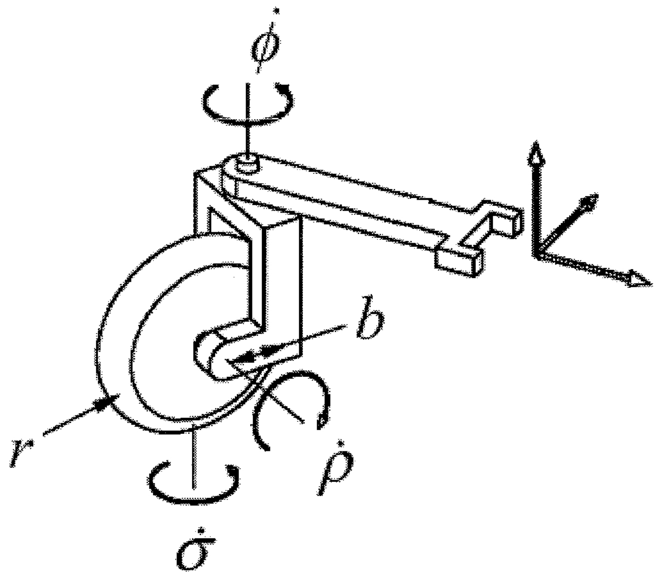


Figure 2.1: open-chain mechanism of caster wheel

Figure 2.2 shows a schematic view of the mobile platform with four wheels on it.

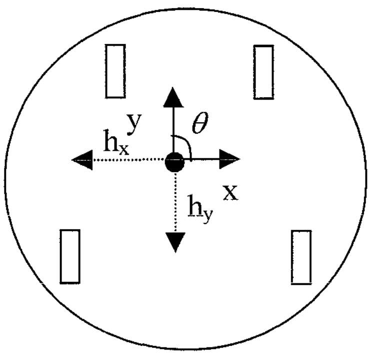


Figure 2.2: schematic view of the mobile base

The position of a wheel on the base is specified in terms of h_x and h_y. These are the distances to a fixed coordinate frame in space (figure 2.2). This frame can be fixed anywhere on the base.
The velocity of the robot is given in terms of forward velocity $\dot{y}$, sideward velocity $\dot{x}$ and angular velocity $\dot{\theta}$. The shape of the platform, the number of wheels, the wheel parameters b and r, and the positioning of the wheel on the platform (the parameters h_x and h_y), depend on the application the base is used for. Their influence is not considered in this report.

2.2 Assignment of coordinate frames

The robot is modeled as a multiple closed-link chain. Coordinate frames are assigned at both ends of each link. For the caster wheel the links are the floor, the robot body and the steering link. Three joints connect these links: the wheel, the steering axis, and the mid-point of the robot.

Consequently there are two coordinate frames at each joint. If a wheel is considered in isolation (figure 2.3), it has three links, three joints and six coordinate frames.

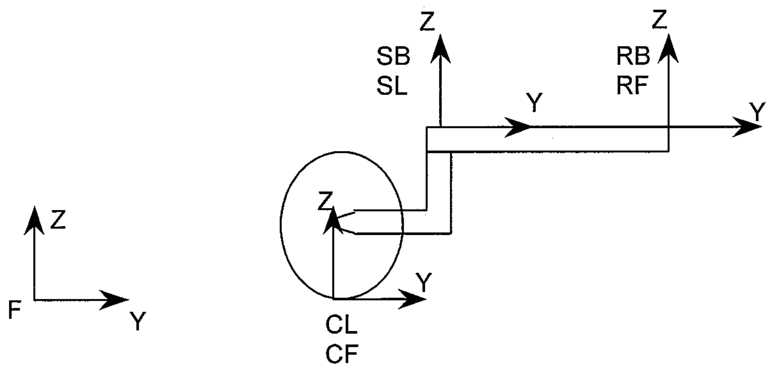


Figure 2.3: Coordinate frames isolated wheel (side view)

The instantaneous frame RF (robot-floor) is used to specify the velocities and accelerations of the robot relative to the floor, independently of the robot position at the instant of observation. The instantaneous contact frame CF (contact-floor) is used to calculate wheel velocities and accelerations relative to the floor. These frames are instantaneously fixed with respect to the floor and not to the robot. At the instant these frames are considered, they are coincident with the frames attached to the robot, frame RF coincides with frame RB (robot-body) and frame CF coincides with frame CL (contact-link).

The floor coordinate frame F is stationary, and serves as a reference frame for the motion of the robot. The robot frame RB is located at the robot, and serves to define the location of the robot with respect to the floor frame for external kinematics.

Two frames are attached to the steering joint. Frame SB (steering-body) is attached to the body and frame SL (steering-link) to the link. The angle between these two frames is the steering angle, φ.

In figure 2.4 a schematic overview of the coordinate frame of a single caster wheel is shown.

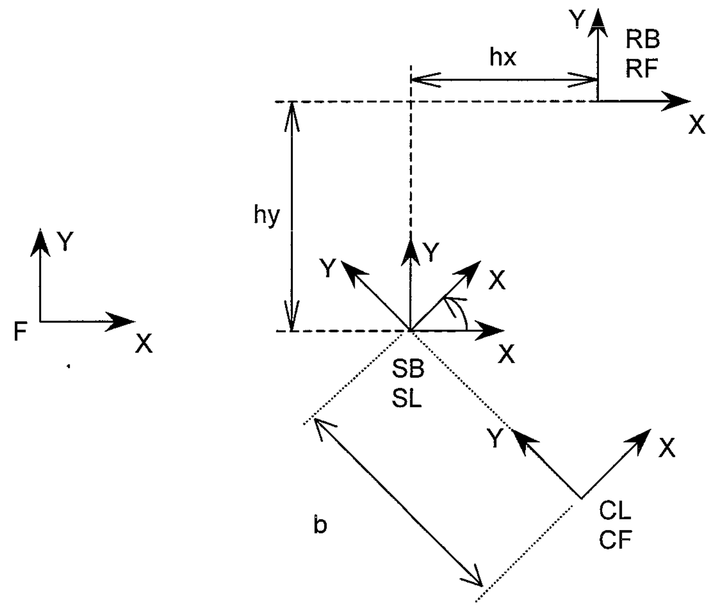


Figure 2.4: Coordinate frames isolated wheel (top view)

The parameters h_x and h_y describe the position of the wheel in reference with a point on the base. The offset of the caster wheel is given by b.

Now that all coordinate frames and variables are specified, the kinematical model of the mobile base can be derived.

2.3 Equations for kinematical modeling

2.3.1 One wheel

The next equations describe the relationship between the instantaneous velocity of the robot and the velocity vector for one wheel [1].

$$ {}^{RF}\mathbf{v}_{RB} = \mathbf{J}_{n\,\mathrm{pseudo}}\mathbf{v}_n = \begin{pmatrix}\dot{x} & \dot{y} & \dot{\theta}\end{pmatrix}^T \tag{2.1} $$

Equation 2.1 denotes the relative velocity of the robot base (RB) to the floor (RF).

In this equation $J_{n\,\text{pseudo}}$ is the pseudo Jacobian matrix for wheel *n*.

$$\mathbf{J}_{n\,\text{pseudo}} = \begin{bmatrix} \cos {}^{RB}\theta_{CL_n} & -\sin {}^{RB}\theta_{CL_n} & {}^{RB}p_{CL_{ny}} & -{}^{RB}p_{SB_{ny}} \\ \sin {}^{RB}\theta_{CL_n} & \cos {}^{RB}\theta_{CL_n} & -{}^{RB}p_{CL_{nx}} & {}^{RB}p_{SB_{nx}} \\ 0 & 0 & 1 & -1 \end{bmatrix} \tag{2.2}$$

Where

$${}^{RB}\theta_{CL} = \text{sum of the angles between the two frames} = \varphi$$
$${}^{RB}\mathbf{p}_{CL} = \text{vector from frame RB to frame CL} \tag{2.2a}$$

This vector is the sum of vector ${}^{RB}\mathbf{p}_{SL}$ and ${}^{SL}\mathbf{p}_{CL}$ (figure 2.5).

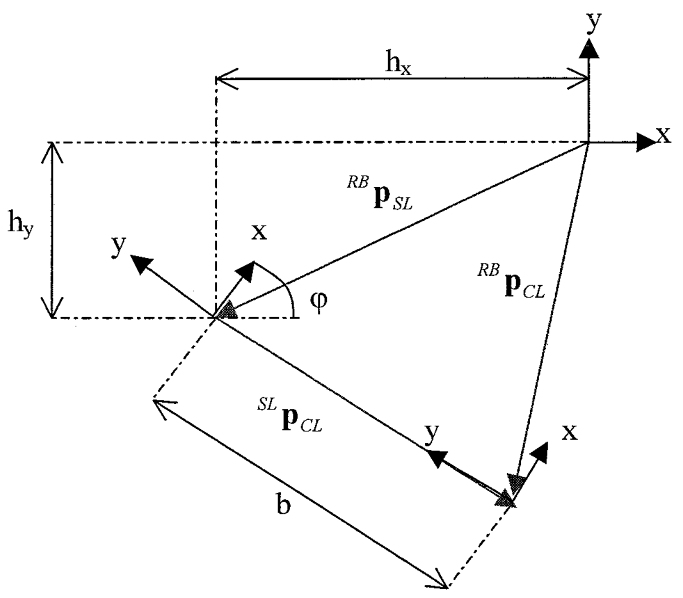


Figure 2.5: vector diagram for the two frames

$${}^{RB}\mathbf{p}_{CL} = \begin{bmatrix} -h_x + b\sin(\varphi) \\ -h_y - b\cos(\varphi) \end{bmatrix} \tag{2.3}$$

The pseudo-velocity vector of a wheel contains four components:

$$\mathbf{v}_{n\,\text{pseudo}} = \begin{bmatrix} {}^{CF}v_{CL_{nx}} \\ {}^{CF}v_{CL_{ny}} \\ {}^{CF}\omega_{CL_n} \\ {}^{SB}\omega_{SL_n} \end{bmatrix} \tag{2.4}$$

${}^{CF}v_{CL_n}$ is the instantaneous linear velocity of wheel n with respect to the floor

${}^{CF}\omega_{CL_n}$ is the instantaneous velocity of wheel n around the contact point

${}^{SB}\omega_{SL_n}$ is the angular velocity of the steering link around the hip joint

In this vector the first component is zero for the caster wheel, because there is no linear velocity in the x-direction (it is assumed there is no side slip). The second component equals $r\dot{\rho}$ and the third and fourth component are respectively $\dot{\sigma}$ and $\dot{\varphi}$.

The physical velocity vector contains all wheel variables. The linear velocity is calculated form the angular velocity of the wheel (${}^{CF}v_{CL} = r\dot{\rho}$). The pseudo velocity vector can then be related to the physical velocity vector by a wheel matrix W:

$$\mathbf{v}_{n\,\text{pseudo}} = \mathbf{W}_n \mathbf{v}_{n\,\text{physical}} \tag{2.5}$$

$$\mathbf{v}_{n\,\text{pseudo}} = \begin{bmatrix} 0 & 0 & 0 \\ r & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} {}^{CF}\omega_{wnx}(=\dot{\rho}) \\ {}^{CF}\omega_{wnz}(=\dot{\sigma}) \\ {}^{SB}\omega_{SLn}(=\dot{\varphi}) \end{bmatrix} \tag{2.5a}$$

The physical wheel Jacobian is derived in a similar way (equation 2.6).

$$\mathbf{J}_{n\,\text{physical}} = \mathbf{J}_{n\,\text{pseudo}} \mathbf{W}_n \tag{2.6}$$

Using equations (2.2), (2.2a), (2.3) and (2.5a) the physical Jacobian is

$$\mathbf{J}_n = \begin{bmatrix} -r\sin({}^{RB}\theta_{CL}) & {}^{RB}p_{CLy} & -{}^{RB}p_{SBy} \\ r\cos({}^{RB}\theta_{CL}) & -{}^{RB}p_{CLx} & {}^{RB}p_{SBx} \\ 0 & 1 & -1 \end{bmatrix} = \begin{bmatrix} -r\sin(\varphi) & -h_y - b\cos(\varphi) & h_y \\ r\cos(\varphi) & h_x - b\sin(\varphi) & -h_x \\ 0 & 1 & -1 \end{bmatrix} \tag{2.7}$$

Now a relationship is derived between robot speed and the velocities of one wheel:

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \mathbf{J} \begin{bmatrix} \dot{\rho} \\ \dot{\sigma} \\ \dot{\varphi} \end{bmatrix} \tag{2.8}$$

The determinant of the Jacobian is $-br$. These values are never zero, hence the Jacobian is always invertible. This means that it is always possible to calculate the wheel velocities given the base velocity vector.

2.3.2 Composite robot equation

Now the Jacobian for one wheel is known, the composite robot equation can be derived. Here two wheels are considered.

First the inverse solution will be derived and then the forward solution. The inverse solution is used to compute the wheel velocities, given a robot velocity. The forward solution is used to compute the robot velocity, given the separate wheel velocities.

2.3.2.1 Inverse solution

In section 2.1, a variable is introduced for twist at the roll contact ($\dot{\sigma}$). This must be done otherwise the wheel would be fixed to the floor. This variable cannot be actuated. It is therefore necessary to separate the actuated (steering and driving velocity) and the unactuated wheel variables.

$$\mathbf{v}_{\text{robot}\,n} = \mathbf{J}_{na}\mathbf{v}_{\text{wheel}\,a} + \mathbf{J}_{nu}\mathbf{v}_{\text{wheel u}} \tag{2.9}$$

Where

$\mathbf{J}_{na}$ = the Jacobian associated with the actuated wheel variables
$\mathbf{J}_{nu}$ = the Jacobian associated with the unactuated wheel variables
$\mathbf{v}_{\text{wheel}\,a}$ = vector with the actuated wheel variables ($\dot{\rho}$ and $\dot{\varphi}$)
$\mathbf{v}_{\text{wheel}\,u}$ = vector with the actuated wheel variables ($\dot{\sigma}$)

For one wheel this equation will be (derived form equation 2.7):

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} -r\sin(\varphi) & h_y \\ r\cos(\varphi) & -h_x \\ 0 & -1 \end{bmatrix} \begin{bmatrix} \dot{\rho} \\ \dot{\varphi} \end{bmatrix} + \begin{bmatrix} -h_y - b\cos(\varphi) \\ h_x - b\sin(\varphi) \\ 1 \end{bmatrix} \dot{\sigma} \tag{2.9a}$$

Now, the composite robot equation can be determined.

$$\begin{bmatrix}\mathbf{I}_1\\\mathbf{I}_2\end{bmatrix}\begin{bmatrix}\dot{x}\\\dot{y}\\\dot{\theta}\end{bmatrix}=\begin{bmatrix}\mathbf{J}_{1a} & 0 & \mathbf{J}_{1u} & 0\\0 & \mathbf{J}_{2a} & 0 & \mathbf{J}_{2u}\end{bmatrix}\begin{bmatrix}\dot{\mathbf{q}}_a\\\dot{\mathbf{q}}_u\end{bmatrix}=$$

$$\begin{bmatrix}-r\sin(\varphi_1) & h_{y1} & 0 & 0 & -h_{y1}-b\cos(\varphi_1) & 0\\ r\cos(\varphi_1) & -h_{x1} & 0 & 0 & h_{x1}-b\sin(\varphi_1) & 0\\ 0 & -1 & 0 & 0 & 1 & 0\\ 0 & 0 & -r\sin(\varphi_2) & h_{y2} & 0 & -h_{y2}-b\cos(\varphi_2)\\ 0 & 0 & r\cos(\varphi_2) & -h_{x2} & 0 & h_{x2}-b\sin(\varphi_2)\\ 0 & 0 & 0 & -1 & 0 & 1\end{bmatrix}\begin{bmatrix}\dot{\rho}_1\\\dot{\varphi}_1\\\dot{\rho}_2\\\dot{\varphi}_2\\\dot{\sigma}_1\\\dot{\sigma}_2\end{bmatrix} \quad (2.10)$$

Where I_1 and I_2 are 3*3 identity matrices.

This can be written as:

$$\mathbf{A}\mathbf{v}_{\text{robot}}=\mathbf{B}\dot{\mathbf{q}}_p \quad (2.11)$$

The composite inverse equation is

$$\dot{\mathbf{q}}_p=\mathbf{B}^{-1}\mathbf{A}\mathbf{v}_{\text{robot}}=\begin{bmatrix}\dot{\mathbf{q}}_a\\\dot{\mathbf{q}}_u\end{bmatrix} \quad (2.12)$$

B is always invertible (determinant B = r^2*b^2), so there is always a solution.

$$\begin{bmatrix}\dot{\rho}_1\\\dot{\varphi}_1\\\dot{\rho}_2\\\dot{\varphi}_2\\\dot{\sigma}_1\\\dot{\sigma}_2\end{bmatrix}=\begin{bmatrix}-\sin(\varphi_1)/r & \cos(\varphi_1)/r & -(h_{y1}\sin(\varphi_1)+h_{x1}\cos(\varphi_1))/r\\ -\cos(\varphi_1)/b & -\sin(\varphi_1)/b & (h_{x1}\sin(\varphi_1)-h_{y1}\cos(\varphi_1)-b)/b\\ -\sin(\varphi_2)/r & \cos(\varphi_2)/r & -(h_{y2}\sin(\varphi_2)+h_{x2}\cos(\varphi_2))/r\\ -\cos(\varphi_2)/b & -\sin(\varphi_2)/b & (h_{x2}\sin(\varphi_2)-h_{y2}\cos(\varphi_2)-b)/b\\ -\cos(\varphi_1)/b & -\sin(\varphi_1)/b & (h_{x1}\sin(\varphi_1)-h_{y1}\cos(\varphi_1))/b\\ -\cos(\varphi_2)/b & -\sin(\varphi_2)/b & (h_{x2}\sin(\varphi_2)-h_{y2}\cos(\varphi_2))/b\end{bmatrix}\begin{bmatrix}\dot{x}\\\dot{y}\\\dot{\theta}\end{bmatrix} \quad (2.13)$$

And

$$\dot{\mathbf{q}}_a=\begin{bmatrix}\dot{\rho}_1\\\dot{\varphi}_1\\\dot{\rho}_2\\\dot{\varphi}_2\end{bmatrix}=\begin{bmatrix}-\sin(\varphi_1)/r & \cos(\varphi_1)/r & -(h_{y1}\sin(\varphi_1)+h_{x1}\cos(\varphi_1))/r\\ -\cos(\varphi_1)/b & -\sin(\varphi_1)/b & (h_{x1}\sin(\varphi_1)-h_{y1}\cos(\varphi_1)-b)/b\\ -\sin(\varphi_2)/r & \cos(\varphi_2)/r & -(h_{y2}\sin(\varphi_2)+h_{x2}\cos(\varphi_2))/r\\ -\cos(\varphi_2)/b & -\sin(\varphi_2)/b & (h_{x2}\sin(\varphi_2)-h_{y2}\cos(\varphi_2)-b)/b\end{bmatrix}\begin{bmatrix}\dot{x}\\\dot{y}\\\dot{\theta}\end{bmatrix} \quad (2.13a)$$

Or

$$\mathbf{v}_{\text{wheel}}=\mathbf{J}^{-1}\mathbf{v}_{\text{robot}} \quad (2.14)$$

Now a relationship is derived between the robot velocities and the separate wheel velocities. Given the desired robot velocity, the according wheel velocities can be computed using the inverse Jacobian (equation 2.13a).

2.3.2.2 Forward solution

The forward solution can be computed using equation 2.11 and is a least square estimate, because A is a non square matrix.

$$\mathbf{v}_{\text{robot}} = (\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T\mathbf{B}\dot{\mathbf{q}}_p \tag{2.15}$$

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \frac{1}{2}\begin{bmatrix} -r\sin(\varphi_1) & h_{y1} & -r\sin(\varphi_2) & h_{y2} & -h_{y1}-b\cos(\varphi_1) & -h_{y2}-b\cos(\varphi_2) \\ r\cos(\varphi_1) & -h_{x1} & r\cos(\varphi_2) & -h_{x2} & h_{x1}-b\sin(\varphi_1) & h_{x2}-b\sin(\varphi_2) \\ 0 & -1 & 0 & -1 & 1 & 1 \end{bmatrix}\begin{bmatrix} \dot{\rho}_1 \\ \dot{\varphi}_1 \\ \dot{\rho}_2 \\ \dot{\varphi}_2 \\ \dot{\sigma}_1 \\ \dot{\sigma}_2 \end{bmatrix} \tag{2.16}$$

Or

$$\mathbf{v}_{\text{robot}} = \mathbf{J}\mathbf{v}_{\text{wheel}} \tag{2.17}$$

The last two components of $\mathbf{v}_{\text{wheel}}$, $\dot{\sigma}_1$ and $\dot{\sigma}_2$ are not sensed. Therefore it is more appropriate to write the sensed forward solution instead. This is done in a way similar to the actuated inverse solution (equation 2.18).

$$\begin{bmatrix} 1 & 0 & 0 & h_{y1}+b\cos(\varphi_1) & 0 \\ 0 & 1 & 0 & -h_{x1}+b\sin(\varphi_1) & 0 \\ 0 & 0 & 1 & -1 & 0 \\ 1 & 0 & 0 & 0 & h_{y2}+b\cos(\varphi_2) \\ 0 & 1 & 0 & 0 & -h_{x2}+b\sin(\varphi_2) \\ 0 & 0 & 1 & 0 & -1 \end{bmatrix}\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \\ \dot{\sigma}_1 \\ \dot{\sigma}_2 \end{bmatrix} = \begin{bmatrix} -r\sin(\varphi_1) & h_{y1} & 0 & 0 \\ r\cos(\varphi_1) & -h_{x1} & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -r\sin(\varphi_2) & h_{y2} \\ 0 & 0 & r\cos(\varphi_2) & -h_{x2} \\ 0 & 0 & 0 & -1 \end{bmatrix}\begin{bmatrix} \dot{\rho}_1 \\ \dot{\varphi}_1 \\ \dot{\rho}_2 \\ \dot{\varphi}_2 \end{bmatrix} \tag{2.18}$$

Or

$$\mathbf{A}_n\dot{\mathbf{p}}_n = \mathbf{B}_s\dot{\mathbf{q}}_s \tag{2.19}$$

Where

$\dot{\mathbf{p}}_n = \begin{bmatrix} \mathbf{v}_{\text{robot}} \\ \dot{\mathbf{q}}_n \end{bmatrix}$ and $\dot{\mathbf{q}}_n$ and $\dot{\mathbf{q}}_s$ are respectively the non-sensed and sensed wheel variables.

In appendix 1, the used Matlab script is given to compute equation 2.19.

Because A_n is not a square matrix, the inverse cannot be computed. Instead a least square estimate is used.

$$\dot{\mathbf{p}}_n = (\mathbf{A}_n^T \mathbf{A}_n)^{-1} \mathbf{A}_n^T \mathbf{B}_s \dot{\mathbf{q}}_s \qquad (2.20)$$

The relationship between the sensed wheel variables and the robot velocities is determined by using equation (2.20).

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \mathbf{J} \begin{bmatrix} \dot{\rho}_1 \\ \dot{\varphi}_1 \\ \dot{\rho}_2 \\ \dot{\varphi}_2 \end{bmatrix} \qquad (2.21)$$

With this equation, the robot velocities can be computed, given the wheel velocities. The composite Jacobian in this equation is very complex and therefore not given here.

2.3.3 More wheels

The kinematical model of the robot can be extended for more wheels. Then the inverse actuated model has the form of equation 2.13a.

$$\dot{\mathbf{q}}_a = \begin{bmatrix} \dot{\rho}_1 \\ \dot{\varphi}_1 \\ \dot{\rho}_2 \\ \dot{\varphi}_2 \\ \vdots \\ \dot{\rho}_n \\ \dot{\varphi}_n \end{bmatrix} = \begin{bmatrix} -\sin(\varphi_1)/r & \cos(\varphi_1)/r & -(h_{y1}\sin(\varphi_1)+h_{x1}\cos(\varphi_1))/r \\ -\cos(\varphi_1)/b & -\sin(\varphi_1)/b & (h_{x1}\sin(\varphi_1)-h_{y1}\cos(\varphi_1)-b)/b \\ -\sin(\varphi_2)/r & \cos(\varphi_2)/r & -(h_{y2}\sin(\varphi_2)+h_{x2}\cos(\varphi_2))/r \\ -\cos(\varphi_2)/b & -\sin(\varphi_2)/b & (h_{x2}\sin(\varphi_2)-h_{y2}\cos(\varphi_2)-b)/b \\ \vdots & \vdots & \vdots \\ -\sin(\varphi_n)/r & \cos(\varphi_n)/r & -(h_{yn}\sin(\varphi_n)+h_{xn}\cos(\varphi_n))/r \\ -\cos(\varphi_n)/b & -\sin(\varphi_n)/b & (h_{xn}\sin(\varphi_n)-h_{yn}\cos(\varphi_n)-b)/b \end{bmatrix} \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} \qquad (2.23)$$

Where *n* is the number of wheels.

Appendix 2 gives the used Matlab script to compute equation 2.23.

To fully describe the robot velocities only three wheel variables have to be actuated. This can be done by taking any three rows in the matrix of equation 2.23 and then inverting this 3 by 3 matrix. However, there can be configurations where this matrix is not invertible (determinant is zero). Then another wheel variable is necessary to get out of the singular point. To guarantee that there is always a solution, at least four wheel variables have to be actuated.

2.4 Model validation

In the previous section the inverse and forward solution are derived. To validate these equations, two tests are performed.

2.4.1 Mathematical validation

Using equation (2.14), the separate wheel velocities can be computed for any given value of the robot velocity vector. Then equation (2.21) can be used to check if the derived kinematical model is correct.

When the model is mathematically correct, the output of equation (2.21) must equal the input of equation (2.14):

$$\begin{bmatrix}\dot{x}\\ \dot{y}\\ \dot{\theta}\end{bmatrix} = \mathbf{J}\begin{bmatrix}\dot{\rho}_1\\ \dot{\varphi}_1\\ \dot{\rho}_2\\ \dot{\varphi}_2\end{bmatrix} = \mathbf{J}\mathbf{v}_{\text{wheel}} = \mathbf{J}\mathbf{J}^{-1}\mathbf{v}_{\text{robot}} = \mathbf{v}_{\text{robot}} = \begin{bmatrix}\dot{x}\\ \dot{y}\\ \dot{\theta}\end{bmatrix} \tag{2.22}$$

Various simulations have been done. In figure 2.6 one simulation is shown. The robot velocity is given and the wheel velocities are computed according equation 2.14. Then the derived Jacobian is used to compute the robot velocity according equation 2.21.
The error is plotted in figure 2.7. The error is due to the numerical integration.
The same applies for the other simulations.

The used Matlab script is given in Appendix 3.

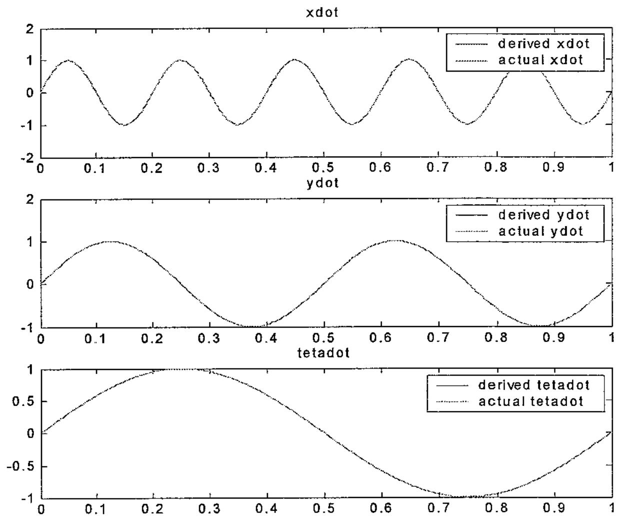


Figure 2.6: Mathematical validation

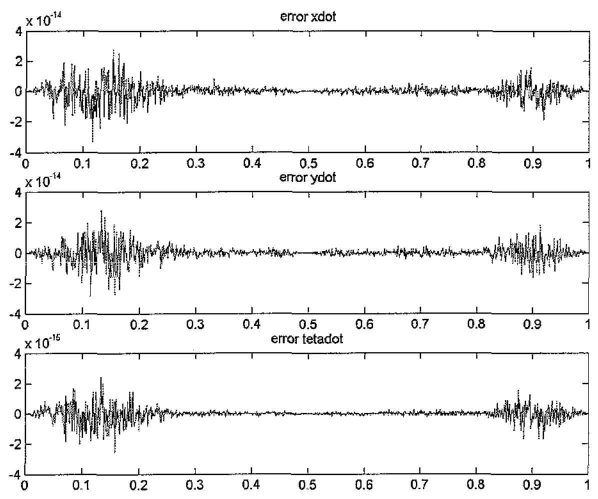


Figure 2.7: error

2.4.2 Physical validation

To test the physical correctness of the inverse model (2.14), it was simulated using Simulink. The orientation of the coordinate frames is shown in the figure below. There are two active wheels and one passive wheel. In the simulations the following dimensions were specified.

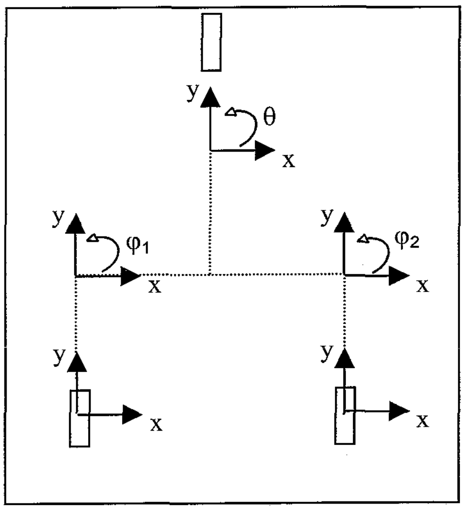
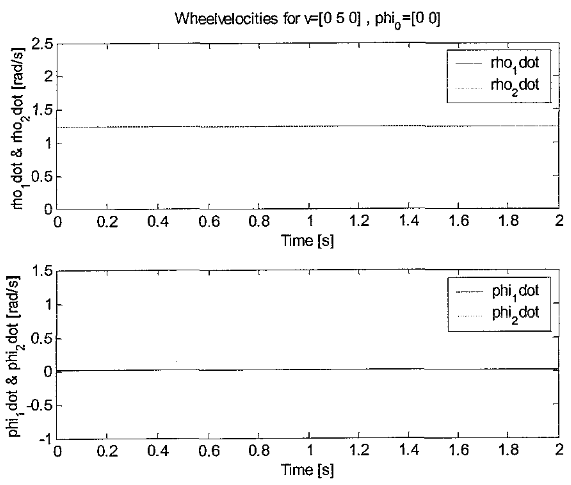


Figure 2.8: orientation coordinate frames

	Wheel 1	Wheel 2
h_x [cm]	-10	10
h_y [cm]	-10	-10
r [cm]	4	4
B [cm]	0.5	0.5

Table 2.1

The corresponding wheel velocities have been calculated given the desired robot velocity $v_{robot} = [\dot{x} \quad \dot{y} \quad \dot{\theta}]$ and the starting angles $\varphi_0 = [\varphi_{10} \quad \varphi_{20}]$ of the steering links.



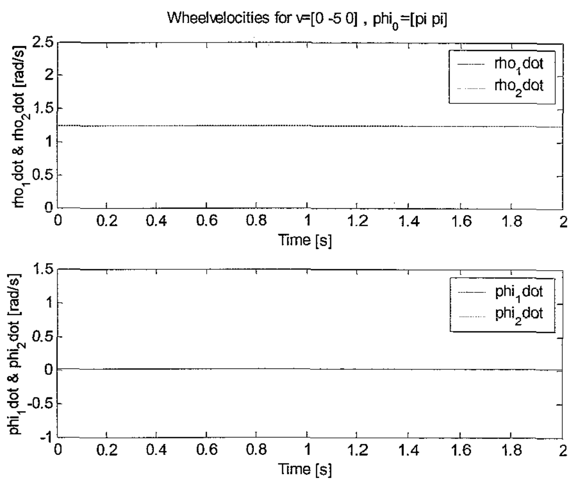
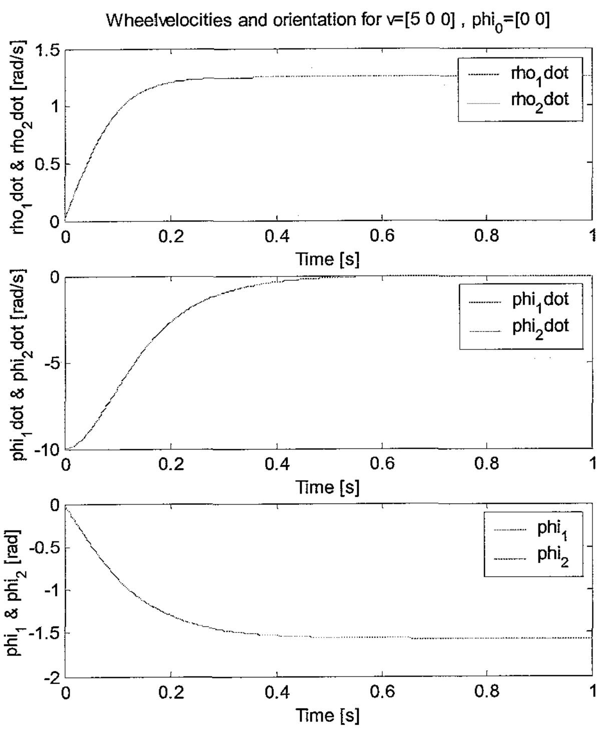


Figure 2.9a: positive velocity in y-direction

Figure 2.9b: negative velocity in y-direction

The results from figure 2.9 show that when a positive velocity in y-direction (5 cm per second) is specified at a zero starting angle (figure 2.9a), both driving velocities are equal and positive. The magnitude of the velocities equals 5/4 = 1.25 rad/s ($\omega = v/r$). The steering velocity is equal to zero, as expected.

When the starting angle equals π (figure 2.9b), a negative velocity in y-direction gives the same results, as expected.



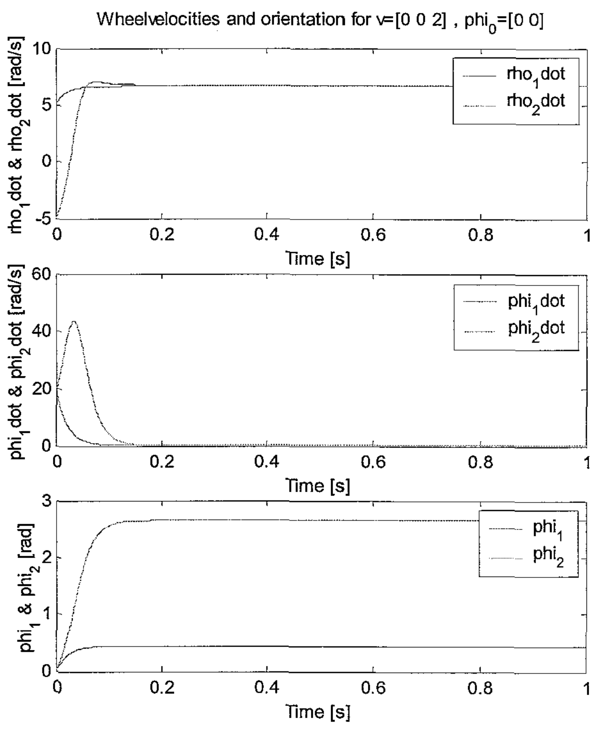


Figure 2.10a: positive velocity in x direction

Figure 2.10b: angular velocity

Figure 2.10a shows the simulation results of applying a positive velocity in the x-direction. To obtain the correct orientation, both wheels have to turn over a negative angle of $\pi/2$ radians. This results in a negative steering velocity, which finally reaches zero as the angle approaches $\pi/2$. In that case only a positive driving velocity exists.

When only an angular velocity is specified (figure 2.10b), both wheels start turning at a different speed until a constant angle is obtained. The final angle of wheel 1 is different from that of wheel 2. From that time on the steering velocity is equal to zero.

More tests have been performed and their results were satisfying the expectations.

3. Experimental setup

In this chapter the experimental setup will be discussed. Figure 3.1 gives an overview of the experimental setup.

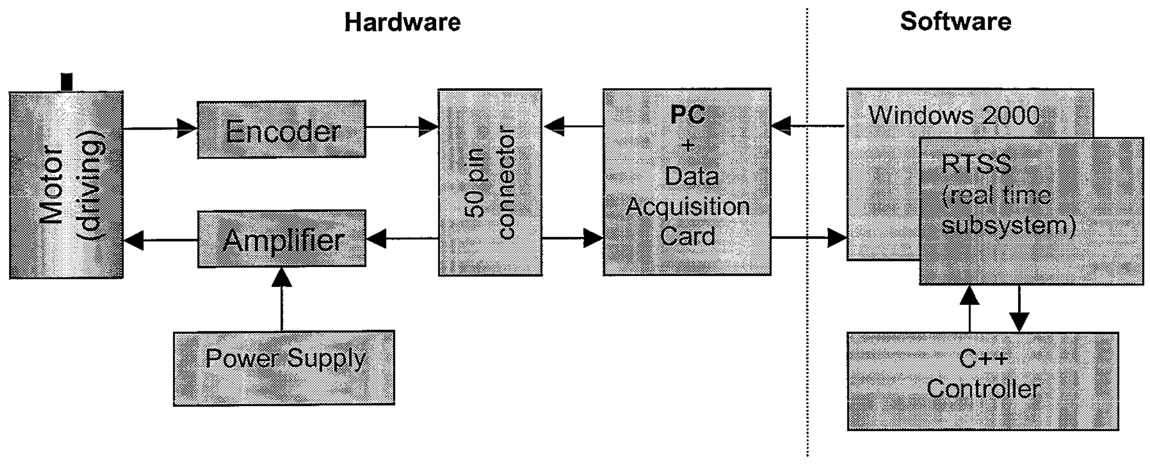


Figure 3.1: Overview hardware and software configuration

3.1 Hardware

The velocity is a result of the torque, generated by a dc motor (for specifications see [2]). The motor torque is proportional to the input current to the motor, which is supplied by a (pulse width modulated) amplifier (for specifications see [3]). This current is linearly dependent on the input voltage applied to the amplifier. The motors are powered by a regulated power supply.

Figure 3.2 shows the velocity control loop. The control output will be the voltage that is applied to the amplifiers: a change in voltage will lead to a change in the wheelvelocity.

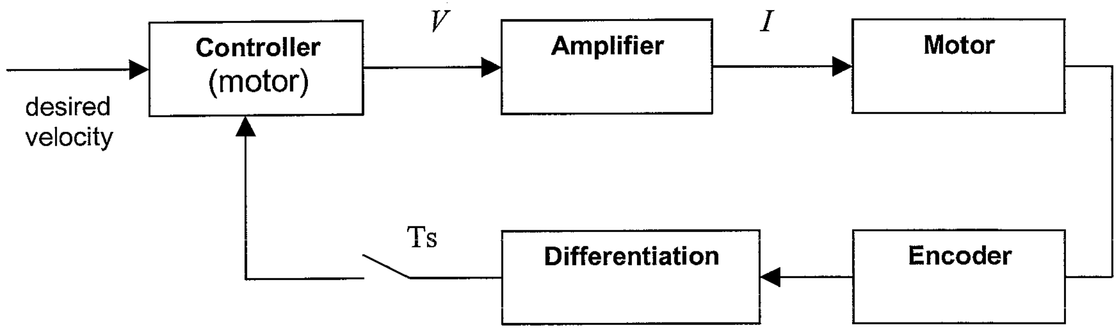


Figure 3.2: Velocity control loop

The input to the controller is the actual wheel velocity, which is obtained after differentiation of the encoder measurement (resolution: 1000 lines/rev).

For the communication between the hardware and software, a data acquisition card is used. It is connected to the real world by a 50 pins connector. The board is accessed through a set of registers. For detailed descriptions of the register and the connector-pin assignment the reader is referred to the online hardware documentation [4]. The output voltage of the DAC channels of the board ranges from –10 to +10 volts.

3.2 Software

The controller, together with the software for data acquisition is written in C++ language and is running on a PC in a Windows 2000 environment. Because Windows is not really suited for 'hard' real-time applications, a software program, called RTX, is used to run the controller software. RTX adds real-time capabilities to Windows by adding a real-time subsystem, known as RTSS, to Windows 2000 (figure 3.2). Instead of using the Windows 2000 scheduler, RTSS performs its own real-time thread scheduling. All RTSS thread scheduling occurs ahead of all Windows scheduling, including Windows-managed interrupts. In this case it enables application components or modules that require deterministic and high-speed response times along with other non-real-time applications to work together. RTX comes with a set of real-time C-functions. For a detailed description of these the reader is referred to the RTX reference and user guide.

In this section only the main working principle of the control software is discussed. The complete program code is in appendix 4 and 5.

Refer to figure 3.3. First the reference signal(s) are loaded. Here the setpoint for the velocity is specified. In the hardware initialization step, the program creates a piece of shared memory for fast read

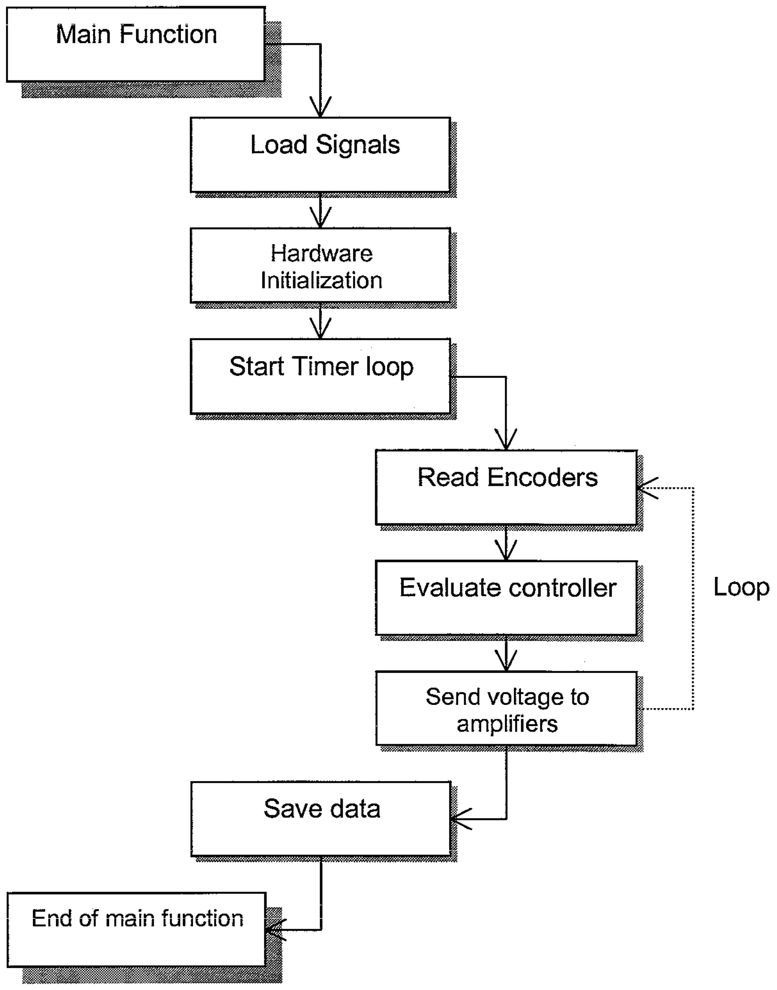


Figure 3.3: Working principle of the control program

and write access during operation. A periodic timer is created, the encoders are reset and access to the registers of the data acquisition card is enabled. After the initialization the timer loop is started. During the period time of the timer the encoders are read, the control law is evaluated and a voltage is send to the amplifiers at each sample time.

After completion of the timer loop, all data are saved to a binary file, and the program is terminated.

4 System identification

To design the velocity controller the relation between the process input and output has to be known. The input is the voltage sent to the amplifier and the output the angular velocity of the motor (figure 4.1).

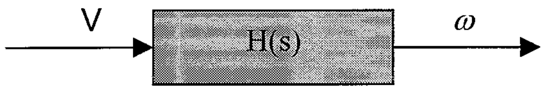


Figure 4.1: Transfer function

The objective of this chapter is to find a proper estimation of the process transfer function *H* and investigate the behavior of the system. In section 4.1 a simple model is derived, to get a first insight in the system. In order to estimate the parameters of the model, a system identification is performed, and validated.

4.1 Simple model

The input to the system is the voltage sent to the amplifier. The amplifier output current *I* is linearly dependent on this voltage *V:*

$$I = \frac{V}{K_E} \tag{4.1}$$

The torque *T* that the motor delivers is proportional to the current drawn from the power supply:

$$T = K_T I \tag{4.2}$$

Combining 4.1 and 4.2 we can write for the torque:

$$T = \frac{K_T}{K_E} V \tag{4.3}$$

A simplified model of a DC motor that relates the motor torque to the shaft velocity ω and acceleration $\dot{\omega}$ is given by equation 4.4:

$$T = J\dot{\omega} + B\omega \tag{4.4}$$

J is the motor inertia and B a damping constant. Combining equations 4.3 and 4.4 and taking the Laplace transform, gives the transfer function between V and ω:

$$H(s) = \frac{\Omega(s)}{V(s)} = \frac{k}{s+\tau} \tag{4.5}$$

Where $k = K_T / K_E$ and $\tau = B / J$. The relation between the input voltage and the output velocity can be described with this first order model.

4.2 Transfer function estimation

In this section a model of the process is derived, based on input-output data. First the structure of the model has to be chosen. The parameters in this model are then estimated by adjusting the parameters until the output of the model approximates the measured output. Here an ARX model (auto regression) is chosen. It has the following structure:

$$A(q)y(t) = B(q)u(t-nk) + e(t) \tag{4.6}$$

Where

$$\begin{aligned} A(q) &= 1 + a_1 q^{-1} + \ldots + a_{na} q^{-na} \\ B(q) &= b_1 + b_2 q^{-1} + \ldots + b_{nb} q^{-nb+1} \end{aligned} \tag{4.7}$$

Equation 4.6 is a differential equation that relates the current output y(t) to a finite number of past outputs y(t-k) and inputs u(t-k). A(q) is the output polynomial of order *na* where B(q) is the input polynomial which is of order *nb*. The delay associated with the input is specified by *nk*.

4.2.1 Identification data set

The input-output data set, used for the identification is shown in figure 4.2. The process is excited by a swept sine wave of constant amplitude (7 volts), which contains frequencies ranging from 0,1 to 100 rad/s. The corresponding output (angular velocity) is measured as a function of time.

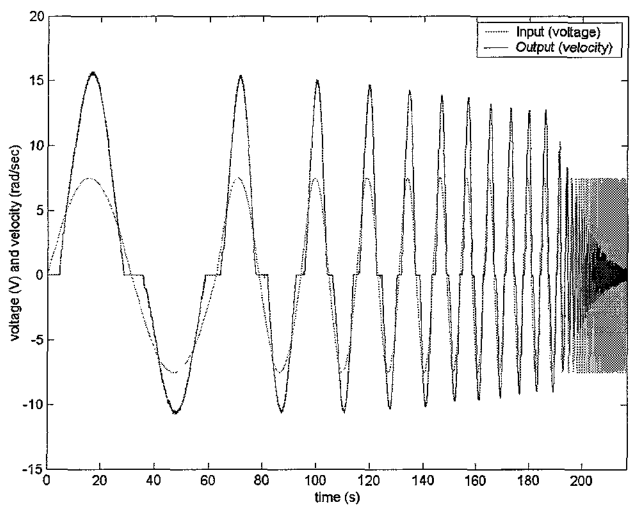


Figure 4.2: System identification data set

From this data the coefficients of A(q) and B(q) are estimated using the arx routine in Matlab.

The order of the input and output polynomials, *nb* and *na*, have to be specified, as well as the delay *nk*. From section 4.1 can be concluded that a first order model is the most appropriate. The delay is set to 1 sample, which is common for systems under sampled data control. This results in the following process transfer function estimate, which is plotted in the bode diagram in figure 4.3:

$$H(s) = \frac{\Omega(s)}{V(s)} = \frac{2.5}{s+1.6} \tag{4.8}$$

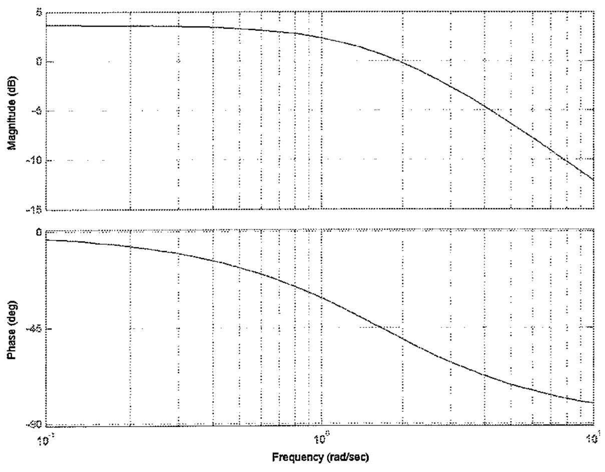


Figure 4.3: Bode diagram of ARX model

This system has a bandwidth of 2 radians per second.

Also higher order ARX identifications have been done. In those cases the obtained frequency response is almost the same as figure 4.3.

4.3 Model validation

Figure 4.2 shows that there is a difference between the positive and negative amplitudes of the velocity. The gain is not equal in both cases. This is due to the fact that the friction is different in both turning directions. The effect of friction is also visible in the 'dead zone' that occurs when the velocity changes from positive to negative sign. Apparently the process is not completely behaving like a linear system.

To validate the derived model, some experiments and simulations are performed. The output, predicted by the model (equation 4.8), is compared to the output of the physical system, using the same input. The results are shown in figures 4.4 to 4.6.

In figure 4.4, the input is a sine wave with amplitude of 10 volts at different frequencies (0.1, 1, 5, 10 and 50 radians per second).

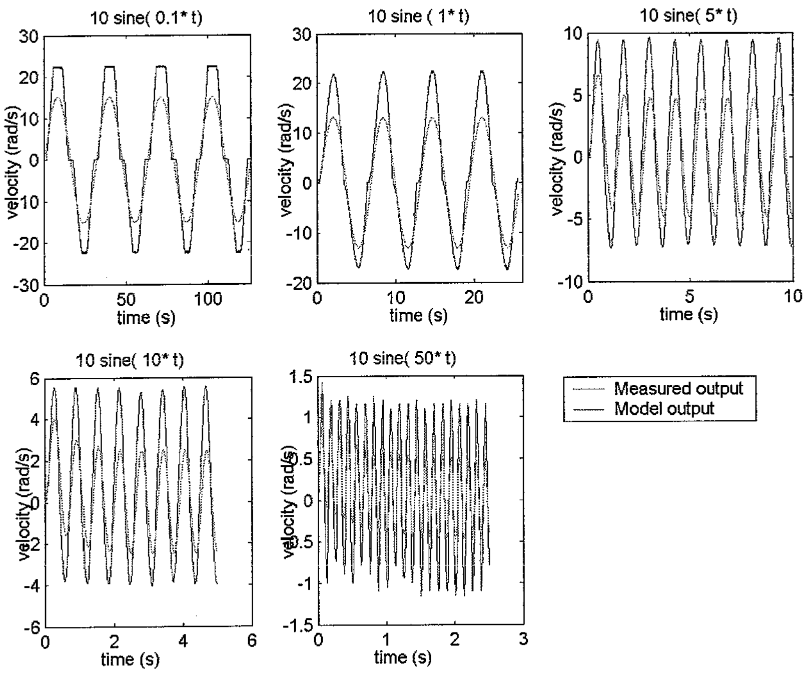


Figure 4.4: Model validation: sine wave input

In figure 4.5, the input is a sine wave with an amplitude of 5 volts at different frequencies (0.1, 1, 5, 10 and 50 radians per second).

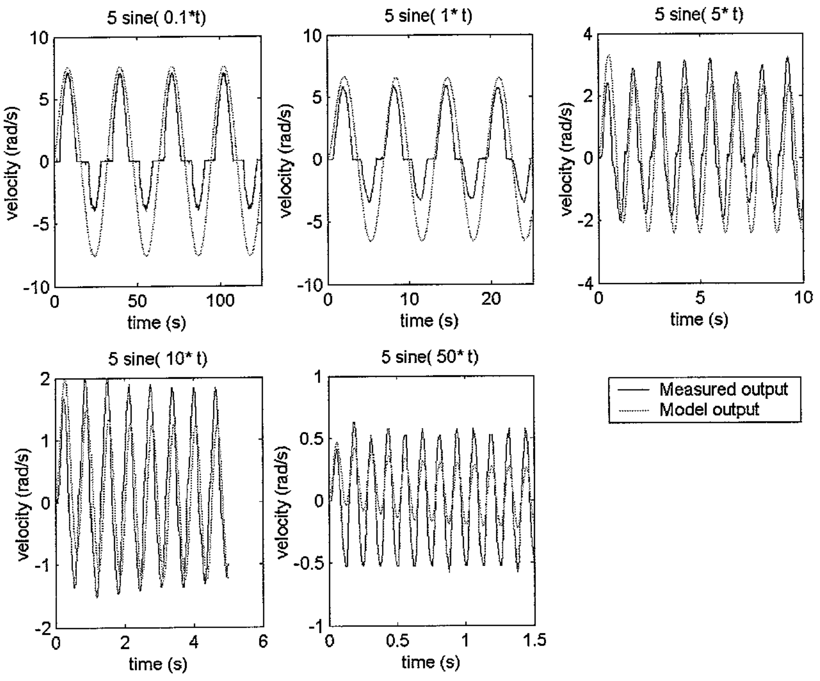


Figure 4.5: Model validation: sine wave input

For both inputs, the measured output and model output are not the same. The phase is good, but the gain is not predicted correctly. This is due to the effect of the friction, which is different in positive and negative turning direction.

Figure 4.6 shows the results for a positive and a negative step with an amplitude of 5 and 7 volts.

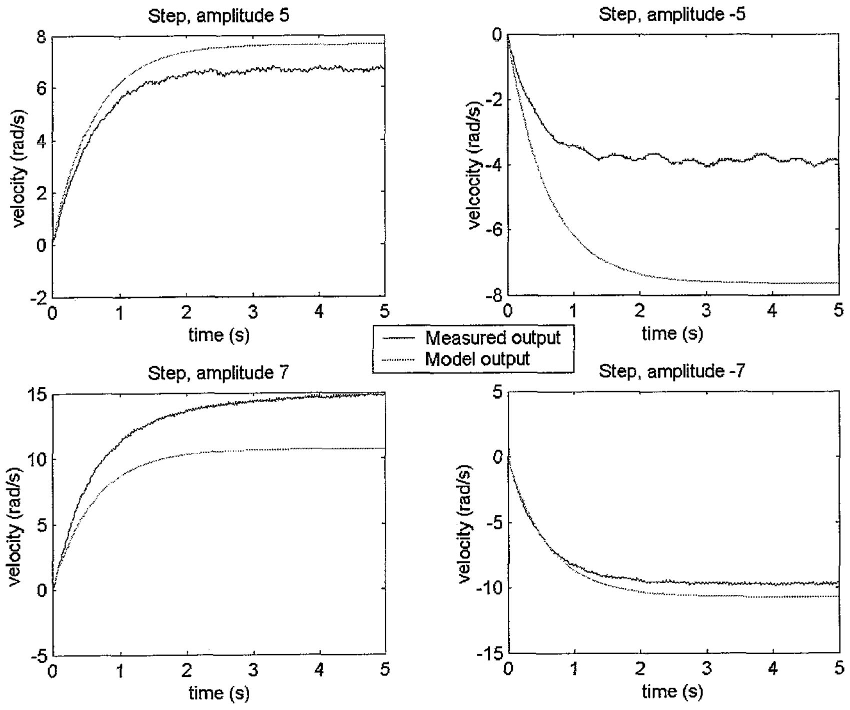


Figure 4.6: Model validation: step input

Again the gain is not the same for the experiment and the model.

4.4 Conclusion

The validation results show that the main difference between the outputs, predicted by the model, and the measured output of the physical system, is the output amplitude. As mentioned earlier, this is due to the effect of the friction, which is different in positive and negative turning direction.
The main shape and phase of the predicted output corresponds well to the measured output.

5 Controller implementation

To achieve the desired wheel velocity, a controller has to be designed. In the previous chapter, a first order model for the DC-motor is estimated. The model is used to design the controller, which will be discussed in this chapter.

5.1 Choice of controller

The system that has to be controlled is a first order system. To achieve desirable behavior of a first order system, usually a PI controller is used.

PI control has the transfer function $C(s) = K\left(1 + \frac{1}{T_I s}\right)$.

Where $T_i = 2\pi f$ and f in Hz.

This results in the following frequency-response for the controller (figure 5.1):

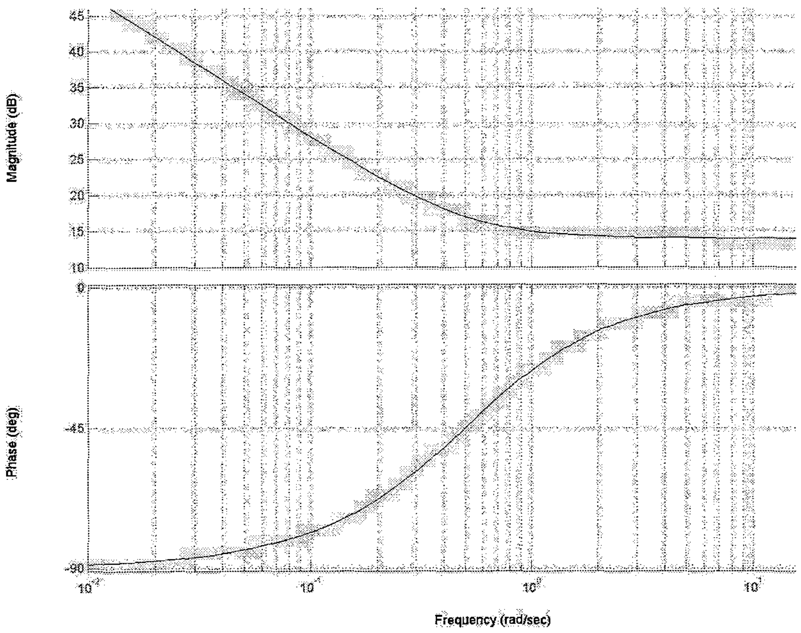


Figure 5.1: Frequency response of PI control for K = 5 and T_I = 2

5.2 First tests

To get some insight in the effect of the controller, some simple experiments were conducted. First a P controller (K=1) is implemented for a third order setpoint (for Matlab script see Appendix 6) in the velocity (upper left part of figure 5.2). The figure shows the measured position and velocity, the velocity error and the input to the process.

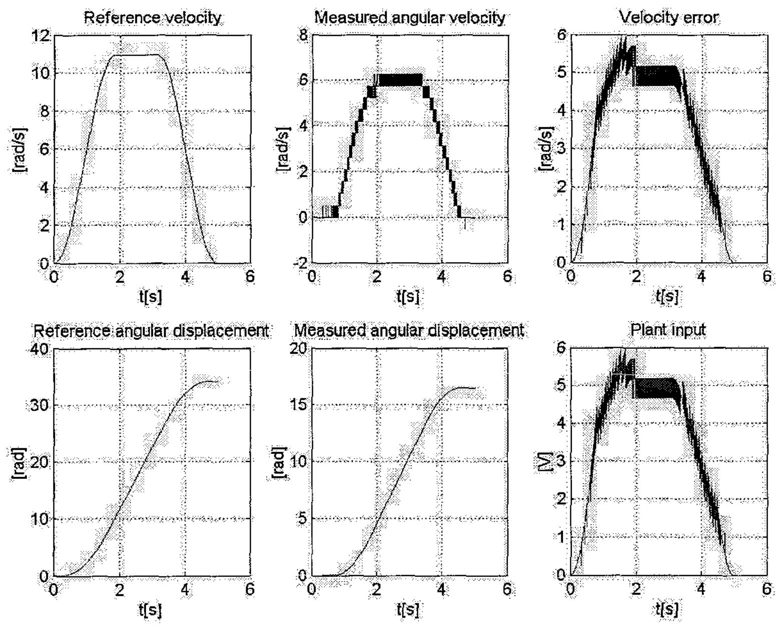


Figure 5.2: Setpoint and results for P controller with K = 1

The measured angular velocity shows strange steps. These are due to the differentiation that is necessary to compute the velocity from the angular displacement.
The plant input stays below 10 volt. That means that the shown input is the actual input to the process. When the controller computes an input higher than +10 volt or lower than –10 volt, the input is set to respectively +10 and –10 volt.

The simulation results of the ARX-model (chapter 4) are presented in figure 5.3.

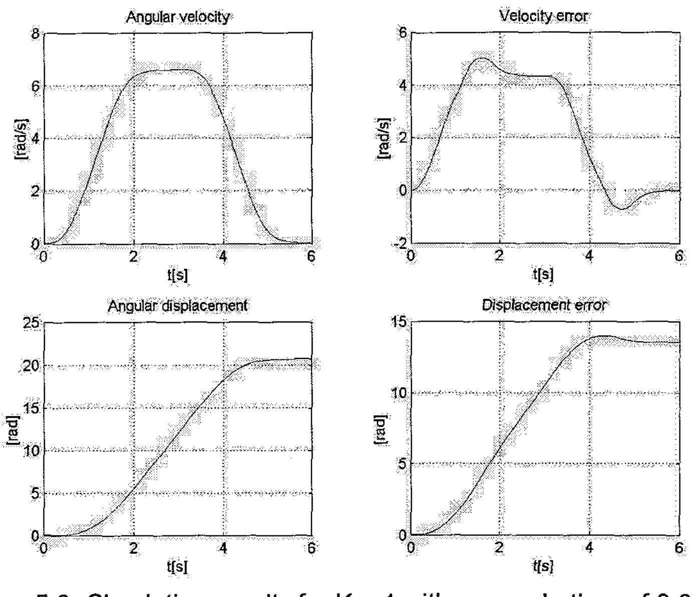


Figure 5.3: Simulation results for K = 1 with a sample time of 0.001 seconds

The simulation results and the experimental results show the same behavior, except for the steps in the actual velocity.

Because the error in the velocity and the displacement are quite big the K value can be further increased, to decrease these errors. The next figure shows the result for K = 40. Also the (theoretical) bode plot of the open loop (CH) is shown. The theoretical bandwidth equals 100 radians per second. The real bandwidth is expected to be much lower.

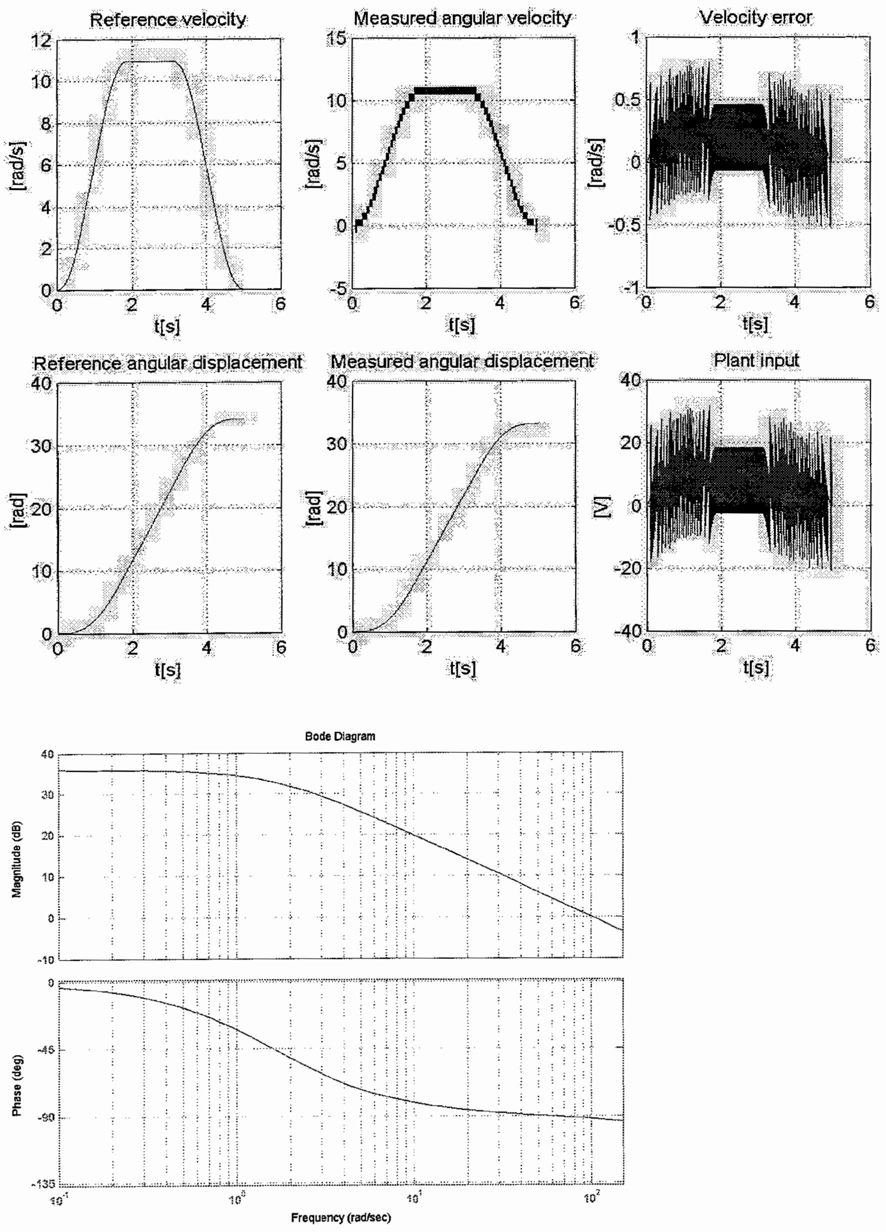


Figure 5.4: Setpoint and results for P controller with K = 40
Lower figure shows the open loop

There is not a visible trend in the velocity error. For the simulation the figures look the same as in figure 5.2, only with a different scaling. The main difference between simulation and experiment is that in the simulation the trends are clearly visible.

With these parameters, the process input exceeds the maximum absolute value, which is 10 volts. In that case, the sent voltage will be equal to plus or minus 10 volts.

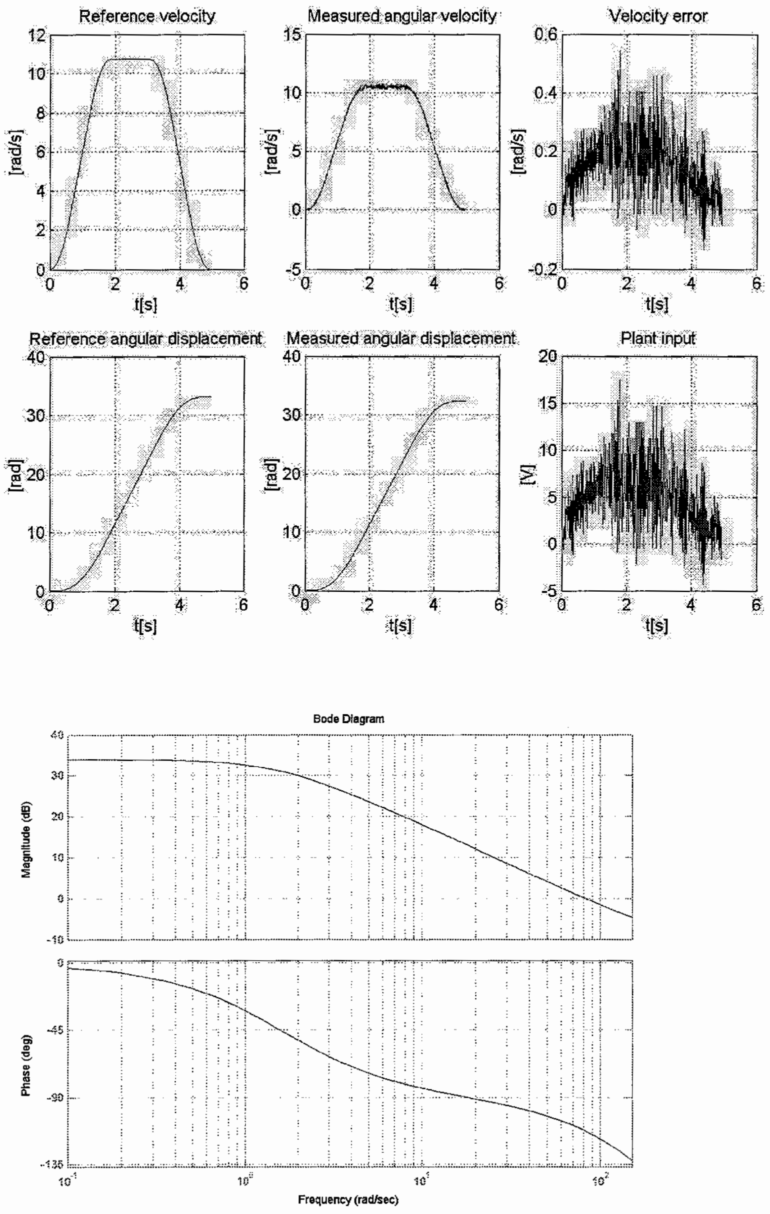


Figure 5.5: Results for K =32 with a sample time of 0.01 seconds
Lower figure shows the open loop

When the sample time is decreased to 0.01 seconds, the measured angular velocity is smoother and the trends are better visible (figure 5.5).

By increasing the sample time, the closed loop behavior changes. Instability will occur sooner for a higher sample time.

5.3 Results

A lot of experiments have been done for sample times of 0.001 and 0.01 seconds. In this chapter some results are presented and discussed.

5.3.1 High sample frequency

For the setpoint in figure 5.2, several values for K and f were implemented. Every experiment is preceded by a simulation. The next figure shows the results for K = 4, f = 32 and a sample time of 1 millisecond.

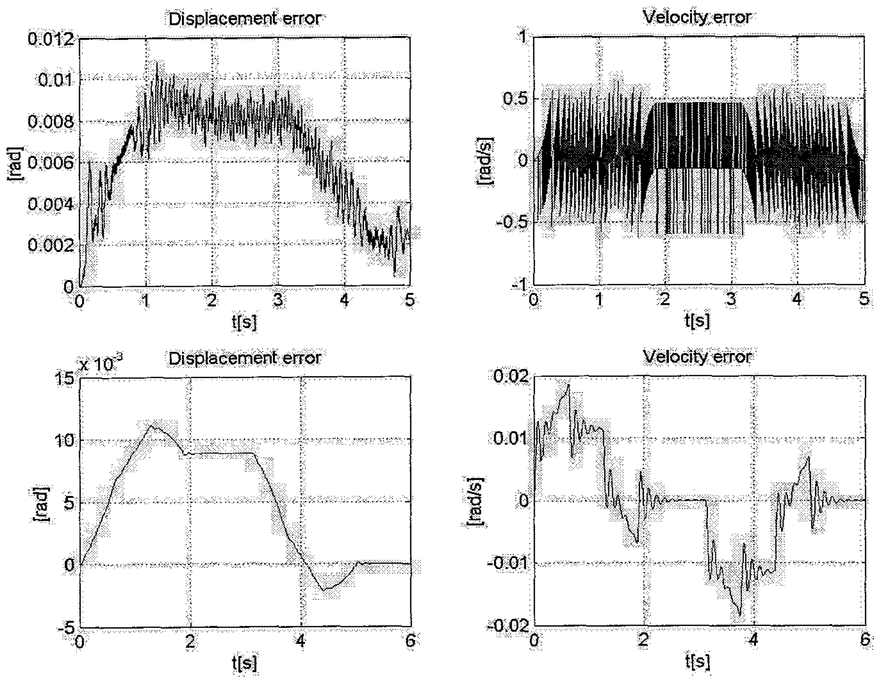


Figure 5.6: Errors for experiment and simulation for K = 4; f = 32; Ts = 0.001

The error in the displacement is almost the same for the simulation and the experiment. The measured velocity error is very noisy. For this sample time, the value and shape of the velocity error doesn't change for several settings of the control parameters (due to the noise introduced by the differentiation). The influence of these settings are better visible in the displacement error. Here, the error in the displacement has a maximum of 0.01 radians.

If the f value is further increased, the error in the displacement decreases. For K = 8 and f = 64, the results are shown in figure 5.7.

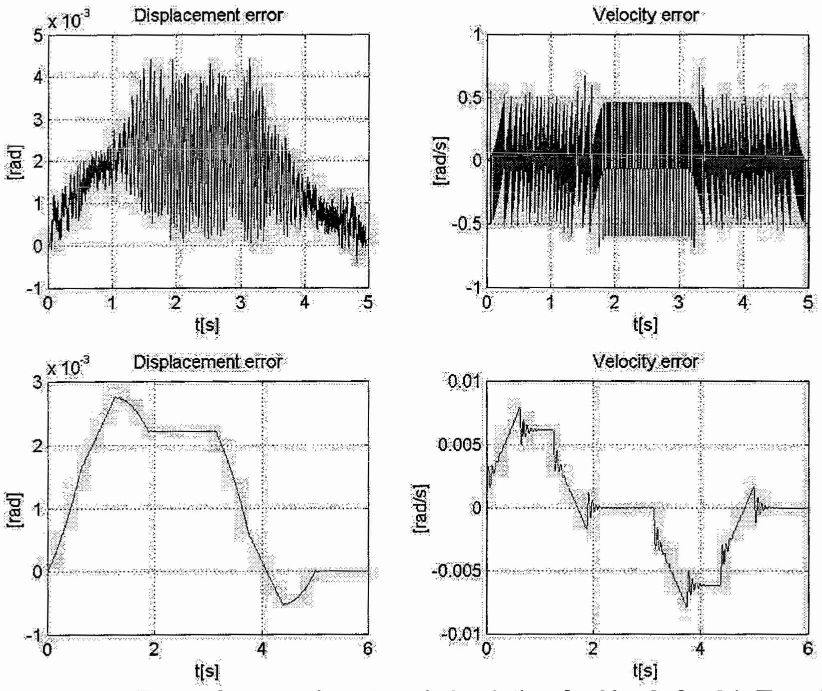


Figure 5.7: Errors for experiment and simulation for K = 8; f = 64; Ts = 0.001

The maximum error in the displacement for this experiment is 0.0044 radians.

The best results are obtained with K = 8 and f = 256 (figure 5.8). The maximum error in this case is 0.0021 radians.

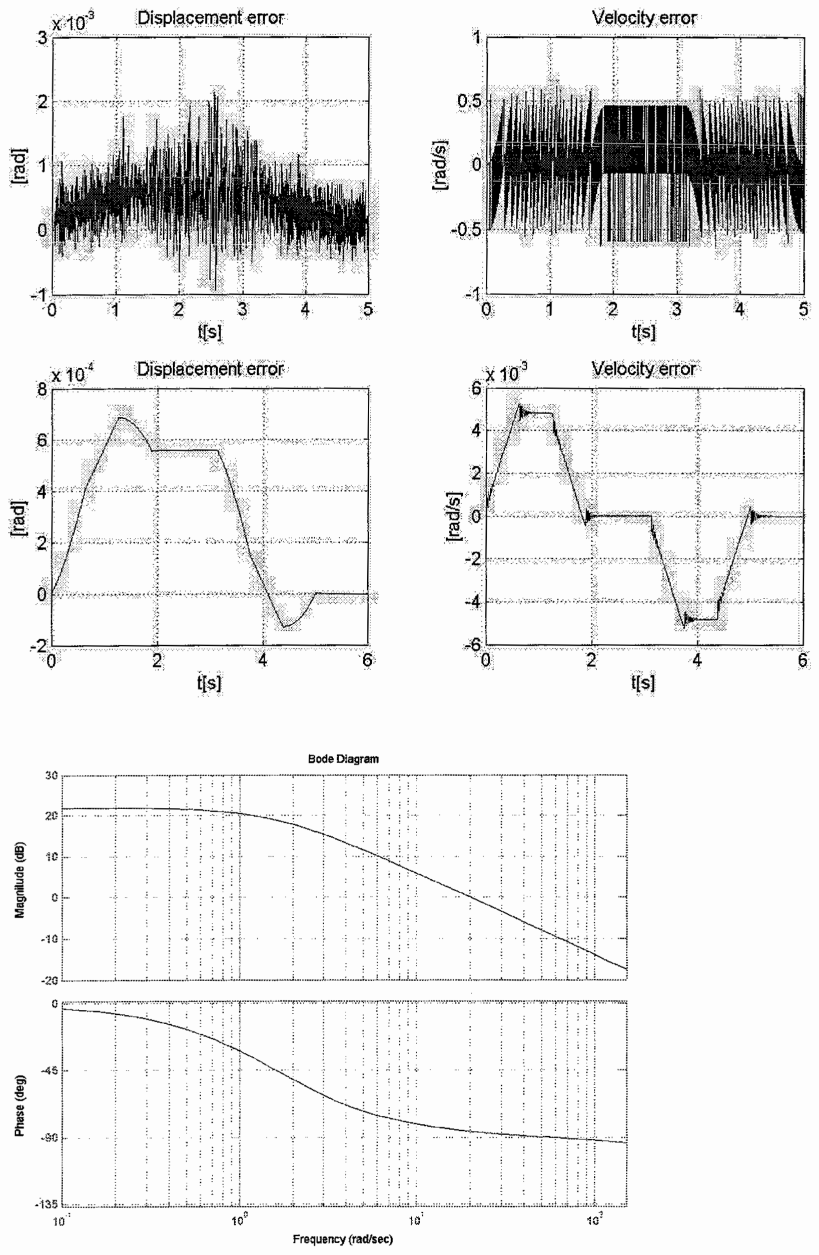


Figure 5.8: Errors for experiment and simulation for K = 8; f = 256; Ts=0.001 and bode plot of the open loop (lower figure)

Further increasing of K and/or f leads to unstable behavior. For these unstable settings, the discrete poles, which are computed in Matlab, are indeed larger then 1.

5.3.2 Lower sample frequency

To decrease the differentiation noise and to make the trends in the errors more visible, the sample time is increased to 0.01 seconds.

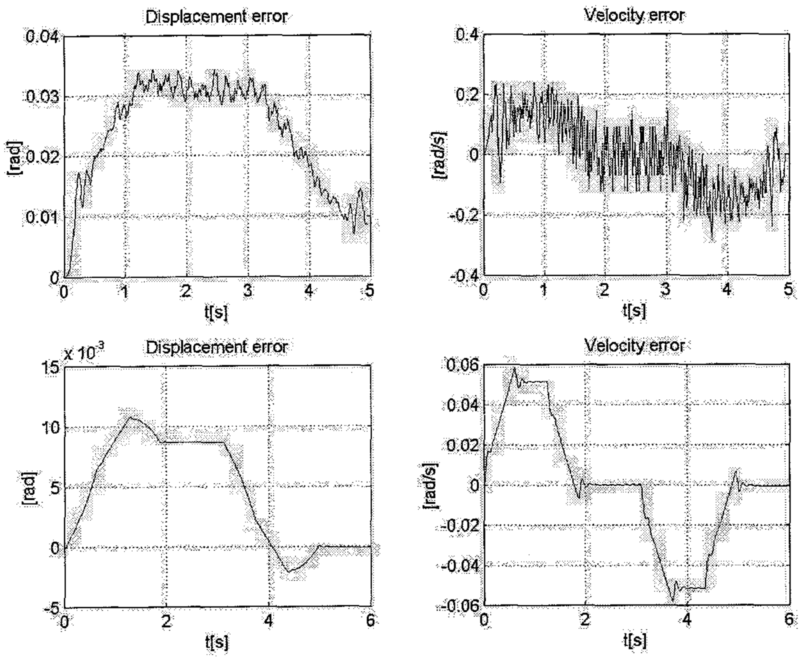


Figure 5.9: Errors for experiment and simulation for K = 4; f = 8; Ts=0.01

Now the trend in both the displacement and velocity error are the same for the experiment and the simulation. For these settings, the maximum velocity error is approximately 0.2 radians per second. Further increasing of K and f leads to instability.

One disadvantage of a low sample frequency is that the values for K and f cannot be significantly increased, due to the instability of the system.

According to the shape of the velocity error in figure 5.9 it can be expected that a feedforward would result in better performance.

5.4 Conclusion and recommendations

A lot of tests have been conducted and the important results are shown in this chapter. Every time the results of the experiment are compared with the results of the simulation. Due to the noise, caused by the differentiation, the trends, clearlily visible with the simulations, are not visible in the experiments with a high sample frequency. By reducing the sample frequency, the trends can be made clear. This sample frequency is too low to really reduce the error. Better results are achieved with the higher frequency, but then the noise intensity is too high. The displacement error could be minimised to 0.0021 radians.

The differentiation noise can be reduced by using a higher order approximation (for example quadratic or cubic) for the velocity obtained from the position measurement.

Due to a lack of time the implementation of a feedforward and development of the controller for the steering velocity could not be achieved.

Conclusion and recommendations

Kinematical modeling of a powered caster vehicle.

A kinematical model for a mobile base, driven by a number of caster wheels, has been derived and validated. Both the inverse and the forward solution were computed. The inverse solution is used to compute the wheel velocities, given a robot velocity. The forward solution is used to compute the robot velocity, given the separate wheel velocities.

A Matlab script is written, that computes a symbolic expression for the Jacobian and it's inverse. In this case the position of the wheels on the base and the number of active wheels are variable.

Velocity control of a separate wheel module

In order to design the velocity controller, an input output model was estimated for the process. This model has been used in simulations, to design a PI controller. This PI controller has been implemented on the real system. For high sample frequencies, the noise intensity, due to the differentiation, was too high to see what really happened to the velocity error. By decreasing the sample frequency, the trends became more visible, but the stability off the system descreased.

Due to lack of time, the steering velocity controller has not been designed.

Recommendations

The differention of the encoder value, to obtain the velocity, caused a lot of noise in the control loop. The influence of this was reduced by decreasing the sample time. A better solution would be to use a higher order approximation (quadratic or cubic) for the velocity obtained from the position measurement.

Better performance can also be achieved by adding a feedforward to the driving velocity controller.

Appendix 1 Forward solution

```
%-----------------------------------------------------------------------
%                    FORWARD SOLUTION FOR MULTIPLE WHEELS
%-----------------------------------------------------------------------
% Computation of forward solution using equation 2.11 :
% A * v_robot = B * qp_dot
% And working to solution of the form of equation 2.19:
% An * pn_dot = Bs * qs_dot
%-----------------------------------------------------------------------

% Clear memory, screen and figures:
clc; clear all; close all;

% Symbolic variables (extend for more wheels, phi2 phi3 hx3 hy3 etc.):
syms phi1 phi2 hx1 hx2 hy1 hy2 r b;

% Give the number of active wheels:
N = 2;

% Give the sensed wheelvariables: (1 is sensed, 0 is not)
% [rho1dot phi1dot sigma1dot ------ rhondot phindot sigmandot]
wv_s  = [1 1 0 1 1 0];       % Extend for more wheels

%Give the positions and dimensions for each wheel:
hx=[hx1,hx2];                % Extend for more wheels [hx1 ----- hx4]
hy=[hy1,hy2];                % Extend for more wheels [hy1 ----- hy4]

%Give the initial angles:
phi=[phi1,phi2];             % Extend for more wheels [phi1 --- phi4]

%Initialize An (called and Bs to be symbolic:
An = zeros(1,1);
Bs = zeros(1,1);
An = sym(An);
Bs = sym(Bs);

%Matrix An:
j = 1;
for i = 1 : N;
    An(j:j+2,1:3) = eye(3);
    j = j + 3;
end

%Matrix Bs:
rho_s = 1; phi_s = 2; sig_s = 3;q = 3;m = 1;w = 1;k = 1;p = 1;s = 4;
for i = 1 : 3*N;
   if wv_s(1,i) == 1
      if i == rho_s
          Bs(k:k+2,m) = [-r*sin(phi(1,w)); r*cos(phi(1,w)); 0];
          rho_s = rho_s + 3;
      elseif i == phi_s
          Bs(k:k+2,m) = [hy(1,w); -hx(1,w); -1];
          phi_s = phi_s + 3;
      elseif i == sig_s
          Bs(k:k+2,m) = [-hy(1,w) - b*cos(phi(1,w)); hx(1,w) -
                                                 b*sin(phi(1,w)); 1];
          sig_s = sig_s + 3;
      end
          m = m + 1;
    else
      if i == rho_s
          An(p:p+2,s) = [r*sin(phi(1,w)); -r*cos(phi(1,w)); 0];
          rho_s = rho_s + 3;
      elseif i == phi_s
          An(p:p+2,s) = [-hy(1,w); hx(1,w); 1];
          phi_s = phi_s + 3;
      elseif i == sig_s
          An(p:p+2,s) = [hy(1,w) + b*cos(phi(1,w)); -hx(1,w) +
                                                b*sin(phi(1,w)); -1];
          sig_s = sig_s + 3;
      end
          s = s + 1;
    end
    if i == q
        w = w + 1;
        q = q + 3;
        k = k + 3;
        p = p + 3;
    end
end
```

Appendix 2   Inverse solution

```
%-----------------------------------------------------------------------
%                    INVERSE SOLUTION FOR MULTIPLE WHEELS
%-----------------------------------------------------------------------
% Computation of inverse solution using equation 2.23 :
% qa_dot = Jinv * v_robot
%-----------------------------------------------------------------------

% Clear memory, screen and figures:
clear all; close all; clc;

% Symbolic variables (extend for more wheels, phi2 phi3 hx3 hy3 etc.):
syms phi1 phi2 hx1 hx2 hy1 hy2 r b;

%Give the number of active wheels:
N = 2;

%Give the positions and dimensions for each wheel:
hx=[hx1,hx2];        % Extend for more wheels [hx1 ----- hx4]
hy=[hy1,hy2];        % Extend for more wheels [hy1 ----- hy4]

%Give the initial angles:
phi=[phi1,phi2];     % Extend for more wheels [phi1 --- phi4]

%Compute Jinv:
j = 1;
for i = 1 : N;
    Jinv(j:j+1,:) = [-sin(phi(i))/r,  cos(phi(i))/r,
                          -(hy(i)*sin(phi(i))+hx(i)*cos(phi(i)))/r;
                     -cos(phi(i))/b, -sin(phi(i))/b,
                            (hx(i)*sin(phi(i))-hy(i)*cos(phi(i))-b)/b];

    j = j + 2;
end
```

Appendix 3 Validation

```
%------------------------------------------------------------------------
%                        MATHEMATICAL VALIDATION
%------------------------------------------------------------------------
% Using equation 2.14 and equation 2.21 to check the mathematical
% correctness.
% Here, done for two wheels. And as setpoint three sinewaves.
%------------------------------------------------------------------------

clear all; close all; clc;

% Specification of dimensions
hx1 = -10; hx2 =  10; hy1 = -10; hy2 = -10; r = 4; b = 0.5;

% Initial angles
phi1_0 = 0; phi2_0 = 0;

% End time and sample time
tfinal = 1;
tstep  = 0.001;
t      = 0 : tstep : tfinal;
a      = (tfinal/tstep) + 1;

% Robot task
xdot   = sin(2*pi*5.*t);
ydot   = sin(2*pi*2.*t);
tetadot= sin(2*pi*1.*t);

% Initialisation
phi1   = zeros(1,a);
phi2   = zeros(1,a);
phi1(1,1)  = phi1_0;
phi2(1,1)  = phi2_0;
A=[1 0 0; 0 1 0; 0 0 1; 1 0 0; 0 1 0; 0 0 1];

% Inverse solution:
for i=1:a

    Jinv = [-sin(phi1(i))/r    cos(phi1(i))/r
                       -(hy1*sin(phi1(i))+hx1*cos(phi1(i)))/r;
            -cos(phi1(i))/b    -sin(phi1(i))/b
                       -(-sin(phi1(i))*hx1+b+cos(phi1(i))*hy1)/b;
            -sin(phi2(i))/r    cos(phi2(i))/r
                       -(hy2*sin(phi2(i))+hx2*cos(phi2(i)))/r;
            -cos(phi2(i))/b    -sin(phi2(i))/b
                       -(-sin(phi2(i))*hx2+b+cos(phi2(i))*hy2)/b;
            -cos(phi1(i))/b    -sin(phi1(i))/b
                       -(-sin(phi1(i))*hx1+cos(phi1(i))*hy1)/b;
            -cos(phi2(i))/b    -sin(phi2(i))/b
                       -(-sin(phi2(i))*hx2+cos(phi2(i))*hy2)/b];

    vrobot1=[xdot(1,i);ydot(1,i);tetadot(1,i)];

    vwheel(:,i)=Jinv*vrobot1;                 % Equation 2.13 and 2.14

    % Computation of phi1 and phi2 out of wheel velocity
    phi1(1,1+i)=phi1(1,i)+vwheel(2,i)*tstep;
    phi2(1,1+i)=phi2(1,i)+vwheel(4,i)*tstep;
end
```

```
%Forward solution:
for i=1:a

    J = 0.5*[-r*sin(phi1(i))  hy1  -r*sin(phi2(i))  hy2
                    -hy1-b*cos(phi1(i))  -hy2-b*cos(phi2(i));
              r*cos(phi1(i)) -hx1   r*cos(phi2(i)) -hx2
                    hx1-b*sin(phi1(i))   hx2-b*sin(phi2(i));
              0             -1     0              -1
                    1                    1];

    vrobot(:,i)= J*vwheel(:,i);                % Equation 2.16 and 2.17
end

subplot(3,1,1),plot(t, vrobot(1,:), t, xdot)
title('xdot'); legend('derived xdot','actual xdot');
subplot(3,1,2),plot(t, vrobot(2,:), t, ydot)
title('ydot'); legend('derived ydot','actual ydot');
subplot(3,1,3),plot(t, vrobot(3,:), t, tetadot)
title('tetadot'); legend('derived tetadot','actual tetadot');
```

Appendix 4 CPP source file

```
//Filename        :    STG_IO_RTX.cpp
//Description     :    CPP source file for data acquisition and control

// Include header files

#include <windows.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <fstream.h>
#include <iostream.h>
#include "rtapi.h"
#include "sens1def.h"

// Define addresses (see [4]) and other constants

#define BASE_ADDRESS (PUCHAR)0x200
#define BASE_ADD ((PUCHAR) (0x200))
#define PORT_DA1  ((PUSHORT) (BASE_ADD+0x10))
#define pi 3.14159265358979

// Create handlers for timer and shared memory

HANDLE hThreadTimer;
HANDLE hShmData;
HANDLE hSharedMem;

// Type definitions

Data  data;
PData pdata;

// Define variables and constants

double ydot_d[5000];
double y_d[5000];
double w[5000];
double timerinterval;
double error;
double derror;
double ierror;
double u;
double K = 4;
double f = 10;
double Ti = 1/(2*pi*f);
double D = 0.05;
double enc = 0;
double time = 0;
int    ii = 0;
int    g = 1;
int    p = 0;
```

```
//Function definitions

/*****************************************************************************
/       To send a analog voltage out
/*****************************************************************************

void analog_out (int  channel, double voltage)
{
      double t_voltage=0.0;
      int dig;

      // Limit output voltage to 10 volt
      if (voltage >10.0) t_voltage = 10.0;
      else if (voltage < -10.0) t_voltage = -10.0;
      else t_voltage = voltage;

      t_voltage = -t_voltage;

      //    Convert voltage to digital 13 bit value
      dig = (int)(8191 * (t_voltage+10)/20.0);

      //    write to DAC Channel
      RtWritePortUshort( (PUSHORT) ((USHORT)PORT_DA1+(USHORT)(channel*2)),
                         (USHORT)dig);
}

/*****************************************************************************
/     Initialise encoders
/*****************************************************************************

void RltZeroCntr(int channel)
{
    PUCHAR addpuc, addpuc1;

      //    Counter command register address for channel 1 (CNT1.C) to access
            the command portion of the counter
      addpuc  = (PUCHAR)(BASE_ADDRESS + 0x03);

      //    Master control register:
            reset all control registers
      RtWritePortUchar(addpuc, (UCHAR)0x20);

      //    Input Control register:
            set counter operating mode
      RtWritePortUchar(addpuc, (UCHAR)0x68);

      //    Output control register:
            define output mode and the function of unused output pins
      RtWritePortUchar(addpuc, (UCHAR)0x80);

      //    Quadrature register:
            Enable times 1 (X1) quadrature mode
      RtWritePortUchar(addpuc, (UCHAR)0xc1);

      //    Master control register:
            Reset encoder counters to zero
      RtWritePortUchar(addpuc, (UCHAR)0x04);

      //    Timer command register address
      addpuc1=(PUCHAR)(BASE_ADDRESS + 0x400 + 0x0e);
      //    Timmer command register:
            Select timer 0
      RtWritePortUchar(addpucl, (UCHAR)0x3a);

      //    Timmer command register:
            Select timer 1
      RtWritePortUchar(addpucl, (UCHAR)0x7a);

      //    Timmer command register:
            Select timer 2
      RtWritePortUchar(addpucl, (UCHAR)0xba);
}

/*****************************************************************************
/     Read encoder counter i. Return 24 bit counter reading value
/*****************************************************************************

long RltReadCntr(int channel)
{
      PUCHAR addpuc, addpucl;
      ULONG cntl,cntm,cnth,cntr;

      //    Counter command register address for channel 1 (CNT1.C) to access
            the command portion of the counter
      addpuc = (PUCHAR)(BASE_ADDRESS + 0x03);

      //    Counter data register address for channel 1 (CNT1.D) to access the
      count portion of the counters
      addpucl = (PUCHAR)(BASE_ADDRESS + 0x01);

      //    Transfer 24 bit counter to latch
      RtWritePortUchar(addpuc,(UCHAR)0x02);

      //    Read counter latch low (8 bit)
      cntl = RtReadPortUchar(addpucl);

      //    Read counter latch medium (8 bit)
      cntm = RtReadPortUchar(addpucl);

      //    Read counter latch high (8 bit)
      cnth = RtReadPortUchar(addpucl);

      //    Compose 24 bit counter reading from the low, medium and high byte
      cntr = (((cntl << 8) + (cntm << 16) + (cnth << 24)) / 256);

      //    To enable counting negative values
      if(abs(cntr-enc)>5000){
            cntr = cntr - 16777216;
      }

      enc = cntr;

      return(cntr);
}
```

```
/*****************************************************************************
/     TimerHandler Function
/*****************************************************************************
void RTFCNDCL TimerHandler(PVOID unused)
{
      //    Set initial values
      if (p=0){

            pdata->i                 = 1;
            pdata->ydot_r[p]         = ydot_d[p];
            pdata->y_m[p]            = 0;
            pdata->e[p]              = 0;
            pdata->t[p]              = 0;
            pdata->ydot_m[p]         = 0;
      }

      p++;

      //    Save reference signal to structure
      pdata->ydot_r[pdata->i]= ydot_d[g];

      g++;

      //    Save angular displacement to structure
      pdata->y_m[pdata->i] = 2*pi/12000*RltReadCntr(1);

      //    Save angular velocity to structure
      pdata->ydot_m[pdata->i] = ((pdata->y_m[pdata->i])-
      (pdata->y_m[pdata->i-1]))/timerinterval;

      //    Compute error in angular displacement
      error = (pdata->ydot_r[pdata->i])-(pdata->ydot_m[pdata->i]);

      //    Compute integral of error in angular displacement
      ierror = (y_d[pdata->i])-(pdata->y_m[pdata->i]);

      //    Save error to structure
      pdata-> e[pdata->i] = error;

      //    Compute error in angular velocity
      derror = ((pdata->e[pdata->i])-(pdata->e[pdata->i-1]))/timerinterval;

      //    Save current time to structure
      pdata->t[pdata->i]= time;

      //    Evaluate PI controller
      u = K*(error + (1/Ti)*ierror);

      //    Save controller output to structure
      pdata->u[pdata->i]= u;

      //    Sent analog voltage
      analog_out(0,u);

      //    Increase counter
      pdata->i = (pdata->i + 1);

      //    Increase time one sampletime
      time = time + timerinterval;
```

```
//      This is here to make sure the output voltage of the cards is set to
        zero after the program is terminated
if (ii> 5000) {
        pdata->run_program = false;
        analog_out(0, 0);}

ii++ ;

//      Save data to binary file
if (ii == 5000) {
        FILE  *fpydot_m;

        if((fpydot_m=fopen("g:\\measured data\\ydot_m2","wb"))==NULL) {
                printf("Cannot open file.\n");
                return;
        }

        fwrite(pdata->ydot_m, sizeof pdata->ydot_m, 1, fpydot_m);
        fclose(fpydot_m);
}

if (ii == 5000) {
        FILE  *fpy_m;

        if((fpy_m=fopen("g:\\measured data\\y_m2","wb"))==NULL) {
                printf("Cannot open file.\n");
                return;
        }

        fwrite(pdata->y_m, sizeof pdata->y_m, 1, fpy_m);
        fclose(fpy_m);
}

if (ii == 5000) {
        FILE  *fpydot_r;

        if((fpydot_r=fopen("g:\\measured data\\ydot_r2","wb"))==NULL) {
                printf("Cannot open file.\n");
                return;
        }

        fwrite(pdata->ydot_r, sizeof pdata->ydot_r, 1, fpydot_r);
        fclose(fpydot_r);
}

if (ii == 5000) {
        FILE  *fpu;

        if((fpu=fopen("g:\\measured data\\u2","wb"))==NULL) {
                printf("Cannot open file.\n");
                return;
        }

        fwrite(pdata->u, sizeof pdata->u, 1, fpu);
        fclose(fpu);
}


if (ii == 5000) {
        FILE  *fperror;
        if((fperror=fopen("g:\\measured data\\error2","wb"))==NULL) {
              printf("Cannot open file.\n");
              return;
        }

        fwrite(pdata->e, sizeof pdata->e, 1, fperror);
        fclose(fperror);
  }

  if (ii == 5000) {
        FILE  *fpyr;

        if((fpyr=fopen("g:\\measured data\\yr2","wb"))==NULL) {
              printf("Cannot open file.\n");
              return;
        }

        fwrite(pdata->yd, sizeof pdata->yd, 1, fpyr);
        fclose(fpyr);
  }

}

/*********************************************************************************
/     Main function (THIS IS EXECUTED FIRST)
/*********************************************************************************

int main(){

      LARGE_INTEGER    Period;
      int   g = 0;
      FILE *fp;

      //    Set timer period (10000 = 1 millisecs)
      Period.QuadPart = 10000;
      timerinterval = 0.001;

      //    Load setpoint
      if((fp=fopen("g:\\Transferfunction\\velocity","rb"))==NULL) {
            printf("Cannot open file\n");
            return 1;
      }

      fread(ydot_d, sizeof ydot_d, 1, fp);

      fclose(fp);

      //    Load integrated setpoint
      if((fp=fopen("g:\\Transferfunction\\displacement","rb"))==NULL) {
            printf("Cannot open file\n");
            return 1;
      }

      fread(y_d, sizeof y_d, 1, fp);

      fclose(fp);

      //    Create shared memory and set the timer
      hShmData = RtCreateSharedMemory(PAGE_READWRITE, 0, sizeof(Data),
                                      MSGSTR_SHM_DATA, (LPVOID*) &pdata);
```

```
    //    Enable I/O address space for access
    if (! RtEnablePortIo(BASE_ADD,0x1F) ) {
          RtPrintf("RtEnablePortIo error = %d\n",GetLastError());
    }

    //    Always use fastest available clock
    if (!(hThreadTimer =
          RtCreateTimer(NULL,0,TimerHandler,NULL,RT_PRIORITY_MAX,
                            CLOCK_FASTEST))){
          printf("SHUTDOWN: ERROR: Could not create the timer.\n");
          ExitProcess(1);
    }


    if (!RtSetTimerRelative( hThreadTimer, &Period,&Period)){
          printf("SHUTDOWN: ERROR: Could not set and start the timer.\n");
          ExitProcess(1);
    }

    //    Reset encoders
    RltZeroCntr(1);

    //    Implement CNTRL0 register
    RtWritePortUchar(((PUCHAR)0x607), (UCHAR)0x8B);

    //    Start the timer (timer was created in suspend mode)
    ResumeThread(hThreadTimer);

    //    To keep program running indefinitely
    pdata->run_program = true;

    //    This while loop must be here or else the program will just stop
          The sleep function is nessecary, to allow the output of the dac's to
          become zero and to write the data in the structure to a file
    while (pdata->run_program == 1 ) {
          Sleep(500);
    }

    printf("End of program!!!\n");

    return 0;
}
```

Appendix 5 Header file

```
//Filename         :     sens1def.h
//Description      :     define structure to save data

#define MSGSTR_SHM_DATA "Message.ShmData"

struct Data {

long i;
int run_program;
double y_m[5000];        // measured angular displacement
double ydot_r[5000];     // reference speed
double ydot_m[5000];     // measured angular velocity
double e[5000];          // error array
double t[5000];          // time array
double u[5000];          // plant input
};

// Type definition of pointer to data structure
typedef Data *PData;
```

Appendix 6 Generation of setpoint and simulation

```
%---------------------------------------------------------------
%       SIMULATION VARIABLES FOR SIMULINK
%---------------------------------------------------------------
% Generate setpoint for simulink and RTX and compute
% simulation variables for Simulink file test.mdl

clear all; close all; clc;

Ts   = 0.001;  % Sample time

%Model according ARX [1 1 1]:
Kp   = 2.521;
tau_p = 1.648;

K    = 8;      % Gain P-action [-]
f    = 32;     % Zero from I-action [Hz]

Time  = 5;         % Time for setpoint
STime = Time + 1;  % Simulation time in Simulink
a     = 7;         % Amplitude setpoint (adjust it)

t1=0:Ts:(Time/8);
y1=a*t1.^2;
dy=y1(max(size(t1)))-y1(max(size(t1))-1);
dx=t1(3)-t1(2);
b=dy/dx;
y2=b*t1+a*t1(max(size(t1)))^2;
y3a=-a*t1.^2;
j=max(size(t1));
for i = 1 : max(size(t1));
   y3(i)=y3a(j)+2*a*t1(max(size(t1)))^2+b*t1(max(size(t1)));
   j = j - 1;
end
y4=y3(max(size(y3)))*ones(1,2*max(size(t1)));
y5=-a*t1.^2+2*a*t1(max(size(t1)))^2+b*t1(max(size(t1)));
y6=-b*t1+a*t1(max(size(t1)))^2+b*t1(max(size(t1)));
j=max(size(t1));
for i = 1 : max(size(t1));
   y7(i)=y1(j);
   j = j - 1;
end
ydot=[y1(1:(max(size(t1))-1)), y2(1:(max(size(t1))-1)),
     y3(1:(max(size(t1))-1)), y4(1:2*(max(size(t1))-1)),
     y5(1:(max(size(t1))-1)), y6(1:(max(size(t1))-1)),y7];

t=0:Ts:Ts*max(size(ydot))-Ts;

figure;plot(t,ydot);

% Save setpoint for velocity for simulink
setpoint = [t;ydot];
save setpoint.mat setpoint;


% Setpoint for angular displacement
% Integration of setpoint for velocity to get setpoint for angle
y(1)=0;
j=1;
```

```
for i = 2:max(size(ydot));
   y(i) = 0.5*(ydot(j+1) + ydot(j))*dx+y(i-1);
   j = j + 1;
end

figure;plot(t,y);

% Save setpoint for velocity and angular displacement for RTX:
[fid]=fopen('velocity','w');
COUNT = fwrite(fid,ydot,'double');
ST = fclose(fid);

[fid]=fopen('displacement','w');
COUNT = fwrite(fid,y,'double');
ST = fclose(fid);
```

References

[1] P.J. McKerrow, *Introduction to robotics*, Sydney: Addioson Wesley, 1991

[2] DC motor specifications, *Emoteq*, QB03400: www.emoteq.com

[3] PWM Amplifier specifications, *Advance Motion Control*, BE30A8: www.a-m-c.com

[4] Motion control board specifications, *Servo to Go*, model 2: www.servotogo.com

[5] RTX 5.0 Reference Guide For NT and Windows 2000, *Venturcom, Inc.*: www.vci.com

[6] G.F. Franklin, J.D. Powell, A.Emami-Naeini, *Feedback control of dynamic systems*, 3rd ed., USA: Addison Wesley, 1995

[7] H. Schildt, *C++ from the ground up*, USA: McGraw-Hill, 1994

[8] H. Schildt, *C: the complete reference*, 3rd ed., USA: McGraw-Hill, 1995

[9] S.S. Iyengar and A. Elfes, *Autonomous mobile robots*, USA: IEEE Computer Society press, 1991

[10] R. Holmberg and O. Khatib, *Development and control of a holonomic mobile robot for mobile manipulation tasks*, Stanford University, USA