Control of a Model Sized Hovercraft

H.J.L.M. Consten

Report nr.: 2003.32

Research report

Coach: Prof. V. Solo

University of New South Wales
School of Electrical Engineering & Telecommunications
Systems & Control Research Group

Supervisor: Prof.dr.ir. M. Steinbuch

Eindhoven University of Technology
Department of Mechanical Engineering
Control Systems Technology

Eindhoven, April 2003
Foreword

I would like to thank the school of Electrical Engineering & Telecommunications of the University of New South Wales for giving me the opportunity to do my external internship at the Systems and Control Research Group in Sydney. I would especially like to acknowledge the guidance and encouragement of my coach at the UNSW, Prof. V. Solo. I am grateful for the regular assistance of Chris Lu with various topics. Together with Dr. D. Clements, Dr. R. Eaton and Jeff Lee they contributed to a pleasant and learning working environment. I very much appreciate that I could use the facilities of the Systems and Control Research Group which made it possible for me to do the necessary research and experiments for the hovercraft project that I did together with Ruud Sanders from September until December 2002.

I would also like to thank J. Katupitiya and J. Sanderson of the school of Mechanical Engineering for letting us use a test-setup in their laboratory and the support with the experiments.

I look back on an inspiring and successful period in Sydney.

Bart Consten.
Summary

During a project at the University of New South Wales, a radio controlled, model sized hovercraft is connected to a computer. The background of the hovercraft is studied to understand its principles and to see how the one used for this project works. The hardware parts are documented and the process of adapting them for computer control is discussed. Software is made to control the hovercraft and the device drivers are discussed. The Data Acquisition board is no problem but for the camera no driver is available.

The finished hardware and software are used for experiments. Velocities and trajectories are measured and represented in figures. They are compared to the simulation data to verify the dynamical model.

The dynamical model is derived via the Newton-Euler method and the model parameters are determined both experimentally and mathematically.

At the end conclusions about the project are made and some recommendations for future development are given since the Systems & Control Research Group would like to continue and improve control engineering with radio controlled hovercrafts.
Summary (Dutch)

Tijdens een project aan de University of New South Wales is een radiografisch bestuurbare hovercraft gekoppeld aan een computer. De achtergronden van een hovercraft zijn bestudeerd om het werkingssprincipe te kunnen begrijpen en om te zien hoe degene die voor dit project gebruikt is, werkt. De hardware componenten zijn gedocumenteerd en het aanpassingsproces, om ze geschikt te maken voor computer control, is besproken. Software is gemaakt om de hovercraft te besturen waarbij ook de device drivers aan de orde komen. De driver voor het Data Acquisition board is geen probleem maar voor de camera is er geen beschikbaar.

Als de hardware en software klaar zijn, worden ze gebruikt voor experimenten. Snelheden en trajectories zijn gemeten en weergegeven in figuren. Ze zijn vergeleken met resultaten van de simulatie om het dynamische model te verifiëren.

Het dynamische model is afgeleid via de Newton-Euler methode en de model parameters zijn zowel experimenteel als mathematisch bepaald.

Aan het einde worden conclusies gegeven over het project en worden aanbevelingen gedaan voor toekomstige ontwikkeling, omdat de Systems & Control Research Group regeltechniek met radiografisch bestuurbare hovercrafts wil voortzetten en verbeteren.
Contents

Foreword ................................. 2

Summary .................................. 4

Summary (Dutch) ......................... 5

Introduction ............................. 9

1 A Hovercraft .......................... 11
  1.1 The Used Hovercraft .............. 12
  1.2 Theory of the Hovercraft ........ 13

2 Hardware .............................. 16
  2.1 Hovercraft .......................... 16
  2.2 Radio Control System .............. 16
      2.2.1 The System .................... 17
      2.2.2 Working Principle .......... 17
      2.2.3 Adjustment for Computer Control 18
      2.2.4 Practical Notes ............ 18
  2.3 The Data Acquisition Board ...... 20
  2.4 The Camera .......................... 21
3 Software ................................................................. 23
   3.1 Device Drivers ................................................. 23
   3.2 RTLinux .......................................................... 24
   3.3 Manual Control Program ................................. 24
     3.3.1 Structure of HOVERCRAFT ....................... 25
     3.3.2 Real-time Control Module .................... 27
   3.4 HOVERCRAFT ................................................... 30

4 Experiments with the Hovercraft ............................................. 31

5 Modelling of the Hovercraft ................................................... 33
   5.1 Kinematic Model Development .......................... 33
   5.2 Rigid Body Dynamics ....................................... 35
   5.3 Translational Motion ....................................... 37
   5.4 Rotational Motion ........................................... 38
   5.5 Adjustment for Model Sized Hovercraft ............ 39
   5.6 Body-fixed to Earth-fixed Transformation .......... 40
   5.7 Hovercraft Parameter Estimation and Identification .... 42
     5.7.1 Mass .......................................................... 42
     5.7.2 Moment of Inertia ....................................... 42
     5.7.3 Forces of the Motors .................................... 49
     5.7.4 Damping ..................................................... 50
     5.7.5 Friction ...................................................... 50

6 Simulations with the Hovercraft Model ..................................... 51

Conclusion and Recommendations ............................................. 54

Bibliography .................................................................. 56

A c-program: mbuff.h ..................................................... 57

B c-program: cdsm.c ...................................................... 63
CONTENTS

C  c-program: cdsm.h  64
D  c-program: cloop.c  65
E  c-program: cloop.h  74
F  c-program: rtmodule.c  78
G  c-program: thread_code.c  82
H  c-program: hovercraft.c  87
I  c-program: hovercraft.h  96
J  c-program: Makefile  97
K  Linux Installation Manual  98
L  RTLinux Installation Manual  105
M  J-experiment Data  110
N  Specifications of the Board for Inertia Test  112
O  m-files: DataFit  113
P  Dimensions of the Hovercraft  115
Q  m-files: Simulation  117
Introduction

This report is about controlling a model sized hovercraft. The school of Electrical Engineering and Telecommunications of the University of New South Wales has a model sized hovercraft that can be controlled by a standard radio control device. The goal for this project is to make the hovercraft controllable of a computer in stead of manually. Once the hovercraft is controllable of a PC, all kind of control strategies can be implemented to test their performance on a hovercraft.

A hovercraft is especially interesting for controller design for numerous reasons. For example, a hovercraft is not connected to a fixed point and therefore it is not easy to stabilize the hovercraft in a desired point or to let it follow a given trajectory. Once the lift force is maximum, the hover takes away almost all of the friction which will give difficulty when steering or braking actions want to be performed.

The way to achieve this goal can be divided into two parts. The first task is to set up the hardware. The hardware part consists of modifying the transmitter, installing a DAC board and connecting the X10 camera to the hovercraft. Secondly, a software program needs to be written. This needs to be some sort of GUI to control the two channels as was usually done by moving the sticks on the transmitter. In this program it must be taken into account that the goal of the project is not open-loop but closed-loop control.

The next step is to operate the hovercraft completely by computer. The computer will then autonomously guide the hovercraft to a desired position or along a predefined trajectory. The computer will need information about the position of the hovercraft. since the hovercraft is remotely controlled and not attached to a reference frame, it is not possible to use an incremental encoder as is the case in most control applications. A possibility to get information about the position of the hovercraft is by using a camera. Therefore a wireless camera can be installed on the hovercraft but there are more cameras available so it is also possible to monitor the hovercraft from above. Via image processing the images from the cameras can be processed to get information about the position and orientation of the hovercraft. A stabilizing controller needs to be designed for which a dynamical model of the hovercraft is necessary.

A model for the hovercraft will be derived. The parameters of the model like mass, moment of inertia, friction and damping coefficients must be determined. The model can be used to simulate the behavior of the hovercraft. The results of the simulations
will be used to validate the model by comparing them to the results of experiments with the hovercraft.

Finally some conclusions are drawn from the obtained results. The performed tasks will be evaluated and the subjects for further investigation will be mentioned.
Chapter 1

A Hovercraft

A hovercraft is a relatively new means of transportation. Its principle was invented around 1955 by a British inventor named Christopher Cockerell. It is different from traditional vehicles because it has no surface contact when it is in motion. It floats on an air cushion and is therefore also called an Air-Cushion-Vehicle or ACV. The hovercraft itself generates the air cushion. The fact that a hovercraft has no direct ground contact has a few advantages to traditional vehicles. One advantage is that it can move over land as well as over water or ice and has no problem with small obstacles.

The main advantage however is the reduction of frictional resistance. Cockerell’s idea was to develop a vehicle that used air as lubrication since this medium produces the least friction. He did experiments to determine the force that an air jet could produce. Encouraged by the results he designed a simple hovercraft, which was basically a plate with a hole in the middle. A fan supplied the air flow through the hole lifting the plate off the ground. This design was far from optimal. The pressure of the fan was not used efficiently and obstacles in the surface were a problem. After this first design the hovercraft has undergone a lot of development. The British government provided funding for the development of hovercrafts because they saw possibilities to use it for military purposes.

There are many ways to establish an air cushion under a vehicle. Mostly used is a design were part of the propulsion airflow is used to fill a skirt underneath the vehicle. The skirt retains the air underneath the vehicle. The skirt makes it possible for the vehicle to rise much higher of the ground without the need for more power. The higher ground clearance and the flexibility of the skirt improve the hovercrafts capability to overcome small obstacles.

In 1959 the first hovercraft prototype crossed the English Channel and in 1962 a passenger service by hovercraft began. The largest passenger hovercraft in the world is used for the Dover to Calais crossing. It can carry 380 passengers and 40 cars. It can reach speeds of 70 mph, which makes it also one of the fastest ferries in the world.
Nowadays hovercrafts have a lot of different application fields. Large hovercrafts are used for passenger and cargo transport. Although hovercrafts are suited for almost every underground, they usually operate on water. The infrastructure on land is not designed for large hovercrafts, but small hovercraft are often used at remote locations with rough terrain.

1.1 The Used Hovercraft

The hovercraft used for this project is a design of an American company specialized in designing model sized hovercrafts. For more information see their internet site: www.hovercraftmodels.com. In figure 1.1 some pictures of the model hovercraft are depicted. The hovercraft is delivered as a kit containing all material, parts and a construction manual. It has two separate motors. One for lift thrust and another for propulsion thrust. At the front there is a space for the electrical components like speed controller, receiver and batteries. The hovercraft can be operated by a standard 2 channel radio control. See section 2.1 for more information on the used equipment.
The design is based on that of the electrocruiser, which is adjusted to make it more suited for the control tasks it is going to be used for. Three adjustments are:

- The two motors have their own battery. One battery is connected directly to a motor and the other is connected to a motor via a speed controller. The user can choose which motor should get a constant and which a variable power. The transmitter has only two channels. One is used for the rudder so one channel is available to control a motor. Advantage of the fact that the motor with constant power can be made variable and vice versa is more influence in the behavior of the hovercraft. Normally the lift is constant and back thrust variable but than the hovercraft keeps moving when the back thrust is turned off. Because there is little friction and the back thrust motor has no reverse. If the lift motor is variable the hovercraft can be stopped more easily because without lift the friction is too big for the thrust motor. A greater influence in the behavior of the hovercraft is good for control purposes especially when the hovercraft is operated in a small space. The two batteries double the operation time of the hovercraft. Now a run of about 15 minutes can be achieved.

- A duct is formed around the lift motor. The advantage of this is that the efficiency is higher. It reduces the pressure loss through the sides.

- A camera is attached to the hovercraft to be able to perform Vision and Control.

The two batteries in the front cause a shift of the center of gravity to the front of the hovercraft. The position of the camera compensates this a little. To determine if the adjustments have effect on the performance of the hovercraft and if the airflow design needs to be changed, the designer at HOVERCRAFTmodels.com was contacted. The current design and the force of the two motors on the hovercraft proved to be sufficient.

1.2 Theory of the Hovercraft

In the basic hovercraft design with skirts, part of the airflow generated by the ducted fan is supplied to an air cavity under the vehicle. The cavity is also called a plenum chamber. The chamber is enlarged by a flexible skirt surrounding it, which makes it possible for the hovercraft to hover higher above the ground and overcome obstacles. The lift is a result of the air pressure in the cavity. When the airflow is increased, the pressure rises, giving the vehicle more lift. At a certain point the air pressure is big enough to carry the weight of the vehicle and air starts to leak from the plenum chamber around the edges. If the airflow is increased even further the vehicle rises higher off the ground and more airflow is needed to maintain the high pressure and compensate the increased leakage. In figure 1.2 the airflow caused by the lift fan and the back-thrust fan is displayed. Part of the airflow generated by the lift fans goes to the skirt. The airflow from the lift fan has two effects. In the figure it can be seen that part of the
airflow firstly creates a lift force to the surface underneath the hovercraft. Secondly the airflow passes underneath the bottom plate of the hovercraft where the pressure causes a lift force to the bottom of the hovercraft. These two effect together make it possible for the vehicle to hover above the ground.

**Skirt**

Hovercrafts can have a bag or a fingered skirt or a combination of both. For small sized hovercrafts like the model used for this project, usually a bag skirt is used. It is difficult to shrink a fingered skirt to model size. Also a fingered skirt tends to scoop water at the back and an inflated bag skirt is more stiff in roll and pitch.

A bag skirt is an inflated loop consisting of a tube of material which is inflated at a slightly higher pressure than the air cushion beneath the vehicle. The size of the air holes in the bottom plate of the hovercraft determine the magnitude of the pressure difference between skirt and plenum chamber. The skirt is one of the most design sensitive parts of a hovercraft. If not right the skirt can be damaged very easily since it encounters the obstacles. The skirts material needs to be light, flexible and durable.

Stability is an other important issue for a hovercraft. From early experiments it was clear that a hovercraft without a skirt is not stable and would bounce up and down. This phenomenon can be explained by the fact that the weight distribution of a hovercraft is never uniform. The hovercraft performs the following cycle:
1. The pressure underneath the hovercraft rises as a result of the airflow created by the lift fan. The hovercraft is lifted off the ground by the air cushion.

2. The air escapes from the side of the hovercraft that has the least weight.

3. The pressure in the air cushion drops and the hovercraft comes down when the pressure rises again the circle starts again at 1. To overcome the instability Cockerell developed the momentum curtain. The airflow produced by the lift fan is inserted in the air chamber at the outer edge of the chamber. The airflow is directed from the sides of the hovercraft to the middle instead of from the middle to the sides. The air cushion in the center is more stable with bigger hover height and less power. The centered air chamber also works as a buoyancy tank. As a result the hovercraft can float on water when it is in rest. The bag skirt helps to increase the stability because it is stiff for pitch and roll and gives a uniform pressure distribution around the hovercraft. The air cushion can't leak on one side and thus the described "bouncing" effect is prevented. More information about the background and design of a hovercraft can be found at Yun [5].
Chapter 2

Hardware

In this chapter the hardware components used to be able to control the hovercraft are discussed. First the hovercraft is described followed by the Radio Control system, the data acquisition board and the camera.

2.1 Hovercraft

The hovercraft is already shortly described in the previous chapter and now the hardware is discussed more thoroughly. The material of which the hovercraft is made is very light but still reasonably rigid. Once the body is finished the electrical parts can be installed. In figure 2.1 the diagram of the electrical components is displayed. The two rechargeable batteries are one of 7.2 V and 2200 mAh (for the lift motor) and one of 7.2 V 1500 mAh (for the back thrust motor). The motors are two Graupner speed 400 high quality motors. The back thrust motor has got a transmission of 1:3 and the lift thrust motor is without a transmission. A speed controller is used to control the lift motor. The receiver and servo to control the rudder angle, are included in the Tower Hobbies radio control set.

2.2 Radio Control System

The radio control system that is used is a standard one by Tower Hobbies. It is a two channel AM radio control system which is enough to control the hovercraft. The hovercraft is designed to control the speed of one motor and the rudder. The transmitter needs a power supply of 8 AA alkaline batteries. These don’t need to be rechargeable because the transmitter doesn’t use a lot of energy. The system has a receiving range of about 350 yards.
2.2.1 The System

The total system consists of a transmitter, a receiver and a servo. The servo is, together with the speed controller, connected to the receiver. The servo is connected to channel 1 and is used to operate the rudder. It is a small motor that turns a certain amount of degrees in proportion with the movement of the stick on the transmitter. The servo has an arm to push a rod that is connected to the rudder of the hovercraft. The speed controller, connected to channel 2, controls the power input of the lift motor in proportion to the movement of the stick on the transmitter. The speed controller also supplies the power for the receiver from the 7.2 V battery. The receiver can be powered by a separate battery but since the hovercraft is powered by electro-motors it is easier to use one of those batteries. With the throttle trim next to the throttle stick on the transmitter, the motor speed at the neutral position of the stick can be set. The same can be done with the steering trim for the rudder angle. There is also a reverse switch for both channels. If the rudder turns right when the stick is pushed left this can be reversed. Again the same can be done for the throttle.

2.2.2 Working Principle

The transmitter sends a signal to the receiver which translates the signal into the appropriate action. The speed controller changes the power output or the servo changes the rudder angle by a certain amount. The amount of change is in relation with the amount in which the matching stick is pushed in a certain direction. In order to attach the transmitter to the computer we need to know how the movement of the stick is transformed in a signal that is sent to the receiver. This works as follows: The stick is...
2.2.3 Adjustment for Computer Control

If the transmitter needs to be modified for computer control, the variable voltage that is produced by the stick can be replaced by a variable voltage from the computer. Between the original wires from the stick, a switch is inserted. The input wires from the computer are also connected to the switch. Now the old configuration with the sticks is still operable. The switch can be turned to disconnect the sticks and activate the signal from the computer. This is done for the two channels separately. A combination of one channel by hand and one by computer is also possible. In figure 2.2 the new layout of the transmitter is displayed.

2.2.4 Practical Notes

Some practical aspects of the connection between the transmitter and the computer still need to be discussed. Things like the voltage that the computer must supply for the two channels, the position of the switches and the color of the plugs and sockets are important to know.

The input voltage for the two sticks must be determined. This is done by measuring the voltage supplied by the variable resistance at different positions of the stick. This is done for both sticks and the results are displayed in table 2.1. Neutral and maximum are the same for the throttle since it doesn’t have reverse. Throttle is from motor off to full speed whereas the rudder goes from neutral to full left and from neutral to full right. It is also verified that the voltage changes linear from minimum to maximum.

The switches on the transmitter are placed on top next to the antenna. If the two switches are pointing to the antenna the transmitter is set for manual control. The two switches to the outside is computer control. And if the switches are in the middle the manual and computer circuits are both disconnected. Always make sure that the power

<table>
<thead>
<tr>
<th></th>
<th>CH1 (rudder)</th>
<th>CH2 (throttle)</th>
</tr>
</thead>
<tbody>
<tr>
<td>neutral</td>
<td>2.56</td>
<td>2.80</td>
</tr>
<tr>
<td>max.</td>
<td>3.03</td>
<td>2.80</td>
</tr>
<tr>
<td>min.</td>
<td>2.10</td>
<td>2.45</td>
</tr>
</tbody>
</table>

Table 2.1: Determined radio control input voltage.
Figure 2.2: Schematic representation of the modified transmitter circuit. The position of the switch in the figure connects the sticks represented by the variable resistance $R_v$. If the switches are changed the sticks are disconnected and the computer control is activated.
of the transmitter and receiver are turned off before the wires from the DAC board are removed. When the wires are disconnected the input voltage is 0 V. This can cause unwanted reactions of the hovercraft. Another thing is that the receiver should only be on when the transmitter is. If the receiver is on and the transmitter is not there is no signal, which can cause the receiver to react to noise. Again this leads to unwanted reactions of the hovercraft.

The sockets on the DAC connection terminal are marked by colors. This way it is clear how the wires from the transmitter need to be connected to the connection terminal. The wires that are used are black and yellow and are fixed together per channel. They have a label telling which pair of wires is which channel. The black wires need to be connected to the black sockets and the yellow to the blue or yellow socket (depending on the channel).

2.3 The Data Acquisition Board

The data acquisition board (DAC board) AX5411H is used as interface between the computer and the transmitter. It is a general Purpose DA&C Board by AXIOM Technology with multifunction analog/digital, input/output channels. The card has 24 digital input and 24 digital output lines, 16 analog input and 2 analog output channels.

Analog input characteristics of the A5411H is designed to sample data at high throughput, high-speed sample/hold and A/D converter allow input sampling speeds up to 60 kHz. With programmable gains of 1, 2, 4, 8 and 16, and full scale range of ±5V and ±10V, a particular range for each input corresponding to the signal level connected to that channel can be defined (resolution 12 bit).

In addition to the data acquisition channels of AX5411H, the board contains two independent analog voltage output channels. Each channel has it’s own 12-bit D/A converter. These two channels can be individually set to output a voltage within the range of 0 to 5V or 0 to 10V. The DAC board is connected to a terminal by a parallel cable. With this terminal it is easy to connect cables to the DAC board. Every channel has two sockets in which banana plugs fit.

On the DAC board the correct base address is set by DIP switch to avoid contending other device. Six bit are equipped to specify the base address in hexadecimal notation. The used base address is 0X320. This address also needs to be specified in the software that will operate the DAC board. This software will be discussed in the next chapter. The DIP switch position for 0X320 is: off on on off on

The full range setting for both the A/D and the D/A can be set by 2 jumpers on the DAC board. A/D is not used at the moment because first open loop control is established. The D/A range is set to +5V. The input voltage for the transmitter is determined to be between 0V and +5V.
CHAPTER 2. HARDWARE

For more information about the DAC board see the user manual on:

2.4 The Camera

The camera that is going to be used for vision and control applications with the model sized hovercraft, is an X10 camera. It is the XCam2, a wireless color video camera that can operate on power from an adapter or from a battery pack. The set contains a camera with transmitter. A receiver that sends the video signal to an adapter that converts it to USB. The transmitter has a receiving range of 100 feet. The chip is a CMOS sensor with a format of 1/3" and an NTSC array size of 510 X 492. It has a minimum illumination of 3 lux (f1.9) and a field of view of 60 degrees. The frame rate is 30 frames/sec. The adapter that converts the video signal into USB is a VA11A adapter. It has a Sunplus Technology Co, Ltd type SPCA506A1 chip. The user manual for this chip can be found at the following internet address:

X10 provides the possibility to place more XCam2 cameras and switch between them. This way it is possible to switch for example from the onboard camera to a surveying camera above the hovercraft. To establish this configuration addressable power supplies, a transceiver and a remote controller are necessary. The remote controller sends a signal, to turn on/off a certain power supply, to the transceiver. The transceiver sends the signal to the appropriate power supply. The power supply of that camera switches on/off. The same way another camera can be turned on/off. The camera that is switched on sends his signal to the video receiver and that is the signal that becomes visible on the computer screen. The X10 equipment uses the American standard for power, plugs and sockets. Therefore a transformer is used to transform the Australian power so it can be used for X10 devices.

As already mentioned an X10 camera is installed on the hovercraft. The position and orientation of the camera can be seen in figure 1.1. More information about the camera and applications can be found at www.x10.com.
As a conclusion of this chapter the total layout of the hardware is schematically displayed in figure 2.3. The PC part is subdivided into two software parts. A vision part and a controller part. The computer sends the two output signals to the D/A part of the DAC board. The two signals are represented by a single arrow because they both follow the same route. From the D/A board the signals are translated into a radio signal by the radio transmitter. The radio signal is received by the hovercraft and transformed into the appropriate action by the speed controller and the servo connected to the receiver on the hovercraft. The movement and orientation of the hovercraft data is collected by the camera which passes it through to the PC via the Video to USB adapter. The software part that connects the input of the PC with its output will be discussed in the next chapter.
Chapter 3

Software

In the previous chapter all hardware components, needed to control the hovercraft, are mentioned. The computer plays an important role in the total system. It gets information from sensors (camera) and sends out data to the hovercraft via the DAC board. Internally the input and output of the computer need to be connected too. This can be done with a software interface (device drivers and control software), which gets data from, and can send data to the hardware interface (Video/USB adapter and DAC board). The control software can contain various algorithms to evaluate incoming data and to calculate outgoing signals. The control software is programmed using the C programming language. It will be explained in the following sections and parts of the code are displayed for clarity. In appendix A tot J the complete code of the control software is displayed.

The computers in the laboratory of the Systems and Control Research Group at the UNSW have two operating systems, Windows and Linux. Windows is easy to use and the most common operating system but not ideal for real-time control applications. Only soft real-time operations could be possible under certain conditions. Linux, in contrast to Windows, is an open source system so it can be guaranteed that real-time control is possible. Another advantage of Linux's open source is the fact that adjustments can be made very easily. Therefore Linux is used as the operating system to control the hovercraft.

3.1 Device Drivers

In figure 2.3 can be seen that the computer is connected to the camera (via the video/USB adapter) and the DAC board. To communicate with these devices the computer needs information about the form of information it gets from the device or what information the device expects from the computer. Therefore device drivers are needed.
CHAPTER 3. SOFTWARE

Camera
The X10 camera (www.x10.com) described in section 2.4 is supplied with a driver for windows. A Linux driver is not available. On the internet it becomes clear that a driver is not available but that a group of Linux enthusiasts in Europe is working on one. As with all Linux programs the development of a driver depends on the achievements of volunteers. The development is not as fast as expected and at the moment it looks like it will take a while before they will have a working driver. An alternative is to program an own driver but that is difficult because there is not much data available about the camera. The manufacturer of the X10 camera doesn’t supply the data of the chipset of the camera and the Video-USB adapter. The available information is that the adapter that converts the video signal into USB is a VA11A adapter. More info in the hardware section about the camera 2.4. The fact that no driver is available for the camera means that the loop of figure 2.3 can’t be closed. Only open loop control is possible until the vision part becomes available.

DAC Board
The DAC board doesn’t need a special driver. The DAC board itself has switches to change its settings, discussed in the hardware section about the DAC board (2.3). In the control software the base address has to be specified and the addresses of the two output channels (input channels are not used). To send data to the two outgoing signals the signal must be scaled by the voltage set on the DAC board (+5V) and translated into a 12 bit integer.

3.2 RTLinux

For real-time control in Linux a special real-time kernel needs to be installed. Therefore RTLinux kernel 3.0 must be installed. But before this is possible the Linux kernel needs to be updated to kernel 2.2.18. See the installation instructions in appendix K. Now the RTLinux kernel can be installed following the instructions in appendix L

3.3 Manual Control Program

The program HOVERCRAFT is software that is developed to control the model hovercraft off a computer. The original program on which HOVERCRAFT is based, is RTCON\textsuperscript{1} by Ray Eaton. It is used to control SISO systems on the AX5411H DAC board. It’s structure is set up in such a way that the program can operate under RTLinux. RTCON is expanded to use is for MIMO purposes with two outgoing signals since the hovercraft has a two channel radio control transmitter, to control both the rudder angle and the lift motor.

\textsuperscript{1}RTCON is the program, used in the laboratory of the Systems and Control Research Group at the school of EE&T at the UNSW, to control experimental setups.
In this section, first will be explained how the program works and how it needs to be setup to operate under RTLinux. This is followed by an explanation of several parts of the C-code that are of particular interest. Finally an example on how to run HOVERCRAFT will be given.

### 3.3.1 Structure of HOVERCRAFT

The program structure is single user real-time control with multiple threads. This means that there are no controllers acting parallel on the same time but only one that is divided in several tasks that are performed sequentially. There are three threads. The first one is the Analog to Digital thread (A/D) which samples the output of the sensors. When this thread is finished it wakes up the second thread and suspends itself until it is activated again. The second one does control-calculation. It transforms the measured data and the desired setpoint into a new output signal. When this thread is finished the Digital-to-Analog thread (D/A) is activated and the control-calculation thread is suspended. The D/A thread sends out the new calculated controller output to the RC-transmitter. The A/D thread is activated on a fixed time interval $\Delta t$. The other threads are activated by the previous thread and not on a fixed time. A schematic representation of the thread scheduling is presented in figure 3.1.

The control program uses a Shared-Memory in which, during the sampling period, the new measured data is stored. In RTLinux this Shared-Memory is provided by the MBUFF module. (For the source code of mbuff.c see appendix A). It enables that both the user space and the kernel space can interact with each other. To ease the programming, a common interface for sharing an array of long datatype in both user and kernel space (called CDSM) has been provided in cdsm.c and cdsm.h (see the source code in appendix B and C respectively). Each element in the data array is called a channel. Interfaces are: initialize CDSM: CDSMinit(void), destroy CDSM: CDSM_done(void), set channel data: CDSM_set(int chan, long data), get channel data CDSM_get(int chan).

During the control-calculation thread the new signals to send out to the D/A ports are computed and stored in the Shared-Memory which are actually sent out during the D/A thread. The threads are placed in a module CONTROL_MOD.O which organizes the scheduling of the three threads. In the future the sampling thread will be used to store the data from the digital camera in the shared memory structure. Parallel to the threads a Graphical User Interface (GUI) is active which records the adjustment of the setpoint by the user and stores it in the shared memory. The interaction between the user and kernel space via the shared memory is displayed in figure 3.1. It is also possible to start and stop the program via a shared variable. Although the threads are still executed periodically until the control module is removed, the sampling and control-calculation threads are ignored when the stop button is pressed. The last updated values in the shared memory are sent out until the start button is pressed again and the sampling and control-calculation threads are no longer ignored.
CHAPTER 3. SOFTWARE

Figure 3.1: Threads of control program HHOVERCRAFT. D/A is sampling, C is control calculation and during D/A data is send to the DAC board.

At this moment, since the camera isn’t working in RTLinux, the sampling period does nothing and the controller is bypassed. The setpoint can be updated permanently, but only during the D/A thread the new values for the lift motor speed and rudder angle are send out. This means that HOVERCRAFT is an open-loop control program but the infrastructure to make it closed-loop is already there. It should be no problem to close the loop once the image processing data of the X-10 camera is available and the control thread can be used to implement a derived controller.

Using threads ensures that these tasks have higher priority than other tasks in the user space like for example the GUI. Only the sampling period is scheduled at a fixed time. The other threads occur when they become activated by the previous thread.

The shared memory structure cloop is the core of the control program. It contains all the data about the state variables, the control parameters and the status of the program like start, stop and data logging. It is available for reading and writing data from the user and the kernel space. The structure cloop is defined in the file cloop.h. It also contains the definition of the functions in the file cloop.c (appendix D). The source code is displayed in appendix E. The most important elements in the structure are:

- setpt0, value to send out to DAC board channel 0.
- setpt1, value to send out to DAC board channel 1.
- cstate, control program state (start/stop)
- datalog, data logging on/off
- datastart, time to start data logging
- dataend, time to stop data logging

In cloop.h the base address is defined to be 0X320.
3.3.2 Real-time Control Module

In rtmodule.c the contents of the real-time control module CONTROL_MOD.0 is coded. The c-file is displayed in appendix F. It contains an initialization function init_module, which sets up the shared memory and sends the specified initial values out to the DAC board when the module is inserted into the memory. A function cleanup_module which ends the threads when the module is removed again. Further the prior discussed structure of the three threads can be recognized; a sampling, control-calculation and digital-to-analog thread are defined. The threads are scheduled in the initialization function.

Note that the actual sampling, control-calculation and digital to analog actions during the threads are coded in a different file that will be discussed later. In rtmodule.c the threads are only specified, their priority is defined, the order in which they wake up and suspend is declared and the status of the program is checked to see if the threads need to be (de-)activated.

The threads in rtmodule.c call the sample, control-calculation and D-to-A functions in thread_code.c. (see appendix G for the complete source code) The function used for open loop control of the hovercraft is dtoa:

```c
/* Function to send to dtoa */
/* Called directly by 'dtoa' thread */
void dtoa(volatile cloop_t *cloop)
{
    float ut0,ut1;
    int uint0,uint1,stat;

    ut0 = (get(cloop,0)).u0;
    uint0 = scale_ftoi(ut0, 0.0, 5.0, 0, 4096);
    stat = io_dacout(cloop, 0, uint0);
    ut1 = (get(cloop,0)).u1;
    uint1 = scale_ftoi(ut1, 0.0, 5.0, 0, 4096);
    stat = io_dacout(cloop, 1, uint1);
}
```

From the shared memory structure the stored values for the two output channels are retrieved and they are scaled between 0 and 5 V and transformed into a 12 bit digital number. These values are send to the RC-transmitter by calling the function io_dacout. io_dacout and scale_ftoi are functions that are coded in cloop.c (see appendix D). io_dacout is the driver function to send data to the ax5411h DAC board. It contains the address for the two channels. (IO_BASE+4: D/A 0 output low byte, IO_BASE+5: D/A 0 output high byte and IO_BASE+6: D/A 1 output low byte, IO_BASE+7: D/A 1 output high byte).
CHAPTER 3. SOFTWARE

/* Function to send data to D/A */
int io_dacout(volatile cloop_t *cloop, int chan, int data)
{
    unsigned int low = (data << 4) & 0x00f0;
    unsigned int hi = (data >> 4) & 0x00ff;

    if( data < 0 || 4095 < data){
        #ifdef __RTL__
            rtl_printf("ax5411: dac out value (%d) out of range\n",data);
        #else
            printf("ax5411: dac out value (%d) out of range\n",data);
        #endif
        return -1;
    }

    switch(chan) {
    case 0:
        outb(low, IO_BASE+4);
        outb(hi, IO_BASE+5);
        break;

    case 1:
        outb(low, IO_BASE+6);
        outb(hi, IO_BASE+7);
        break;

    default:
        return -1;
        break;
    }
    return 0;
}

The channel and the output value need to be specified when io_dacout is called. First
the digital number is checked to see if it is a 12 bit integer. Then the correct channel is
chosen to send the data to.

The link between the user and the shared memory structure is the GUI. It enables the
user to start and stop and quit the control program. Data logging can be turned on or off. See figure 3.2 for a picture of the GUI. The actual controlling of the hovercraft can
be done with the four buttons in the middle of the GUI. Right/left turns the rudder so the hovercraft goes right or left. Up gives the lift motor more thrust and so the
hovercraft goes up. Down decreases the lift thrust. The resolution can be adjusted for
every button separately if the control of the hovercraft turns out to be to sensitive or
not quick enough. In the right part of the GUI window the status of the up/down and
left/right button is displayed. The value is the voltage send out to the DAC board.
The file hovercraft.c (in appendix H) contains all the code for the GUI and some other
functions to arrange callback for the various buttons. The main GUI function is main
CHAPTER 3. SOFTWARE

Figure 3.2: The graphical user interface of the control program HOVERCRAFT

in which the GUI is initialized by calling the function gui_init. If certain conditions are not satisfied (if for example the control module is not inserted) this function will cancel the GUI startup. The DAC board is reset by sending out the initial values to the separate channels. gui_main is called and in gui_main, gui_lab is called which contains the actual code for the GUI. And finally the shared memory is freed. The structure of the callback function for the four control buttons (up, down, left, right) is the same.

lookup_sat is the function that contains the maximum, minimum and initial values for the two output channels. It is used to initialize the shared memory and to verify if the user defined input is within the specified boundaries before the memory is updated. Also during initialization and if stop is pressed the initial (neutral) values for the two channels need to be looked up. Since these boundary values are needed in several places in the C-code it is easier to store them in a central place and call them via a function. This way it is also easy to adjust them if that is necessary. The values for this function are determined in section 2.2 and displayed in table 2.1.

When a buttons (up, down, left, right) is pressed the setpoint is adjusted by $\pm 0.01$ V and then checked if it isn't out of the range specified by the boundary values in sat. The resolution of $0.01$V means that the rudder can be adjusted in steps of 1.29 degrees from -60 to 60 degrees. The new setpoint values are adjusted in the status part of the GUI and send to the shared memory structure. In appendix I the corresponding header file for hovercraft.c is displayed.
3.4 HOVERCRAFT

To run the control program HOVERCRAFT start up RTLinux as root. If the program is already build the control module can be inserted and then the program can be run. A typical session looks like:

```
# cd /home/hovercraft/hover
# insmod control_mod.o
# ./hovercraft
```

Now the GUI appears and the hovercraft can be controlled.

To build the program the Makefile (see appendix J) needs to be executed by typing

```
# make
```

in the command prompt. This only needs to be done if something in the source code is changed and the program has to be compiled again.

When the GUI appears click start and the hovercraft can be controlled by the up, down, left, right buttons. When the stop button is clicked the output values for the two signals are reset. This means the rudder is back in the center position and the lift motor is switched off. Note that the values in the shared memory structure for the two channels can be updated by clicking the control buttons while in stop mode. The output doesn’t change then but when start is pressed the new values are sent out which can result in a step input for the rudder or lift motor. When log is clicked data is written to the textfile: hcraft. Columns are: *time*, *r*, *y*, *u0*, *u1*. *r* and *y* are not important at this moment. *r* is normally used as a reference input for a closed loop controller. In thread_code.c, in function sample, the reference input state is checked which default value is 0. In that case *r* is the same as the neutral signal on channel0. If input is set to be 1 the reference signal *r* is a ramp input with an increment value ramp_increment but also a step or other kind of input can be specified which can be read from the shared memory by the control thread. *y* is an input signal of the I/O DAC connection terminal, which is not connected at this moment. *time*, *u0* and *u1* are of interest because with this output it is possible to compare the movement of the hovercraft in time with the output signal of the control program to see if the desired effect is reached. Clicking log again will stop the data logging. Also stop button stops the data logging if that is active. Quit doesn’t reset the output values, the last updated values will be sent until the control module is removed. It only closes the GUI, the control program will continue in kernel space. Always push stop before quit so the two output values are set to neutral.
Chapter 4

Experiments with the Hovercraft

The hardware and software are ready thus open loop control experiments are possible. Everything must be connected properly and the control software started.

From the first experiments it becomes clear that the hovercraft is not properly balanced. The extra weight of the second battery shifts the center of gravity too much to the front. The result is that the front of the hovercraft touches the ground in the lift force is reduced. The hovercraft starts to turn around the point that touches the ground if the rudder is turned.

In full lift the hovercraft is not stable. The force of the lift motor and the center of gravity do not completely coincide. The hovercraft tends to float in a certain direction where the weight is least. With small masses made out of lead the hovercraft is stabilized as much as possible. The weight is put at the side of the hovercraft in which it starts to move. By trial and error this is optimized. This balancing process must be repeated every time adjustments are made to the hovercraft. If for example the batteries are charged, they are never replaced at the exact same position. Thus after recharging, balancing is necessary.

The circumstances during the experiment with the hovercraft are important. The space must be large enough to operate the hovercraft because it can float off very easily and has a relatively large turn radius, despite the fact that the lift force is variable. Because the hovercraft is very light and it floats, it can’t be operated if there is an airflow like for example the wind outside. The hovercraft is too sensitive for that. The underground must be as horizontal and smooth as possible. On an inclination the hovercraft floats away. The corridor is a good place except that the safety of the passing people must be considered. The propellers can do serious damage therefore a part of the corridor could be separated by lines to do the experiments.

It is difficult to do measurements on the hovercraft in motion. The only data that is collected is with a digital camera. A film clip is recorded with a certain frame rate. It can be used to determine the trajectory of the hovercraft. The camera records the
movement from above but not really perpendicular to the plane the hovercraft moves in and the camera does not stand completely still. This results in a distortion of the image. A circle becomes an ellipse but if this is taken into account it is no problem.

To determine the maximum surge speed of the hovercraft an experiment is performed with full lift force and a rudder angle of zero. With some Matlab software the distance in the picture is related to the distance in reality and with the frame rate this gives a speed of 3.0 m/s in surge direction. Another experiment was performed with a half power lift force and a rudder angle of 10 degrees. The result is given in figure 4.1. Not only the trajectory but also some hovercrafts (at random time intervals) are displayed. This is done because not only the position of the hovercraft but also its orientation is relevant.
Chapter 5

Modelling of the Hovercraft

In this chapter a model of the hovercraft will be derived by studying statics and dynamics. This study is divided into two parts. First the kinematics of the hovercraft are discussed, covering the geometrical aspect of the dynamics. After that the dynamics of the hovercraft are examined. This results in the equations of motion which represent the acceleration of the hovercraft as a result of the forces acting on it. The modelling process is based on the one used by Thor I. Fossen [1] to model the dynamics of ocean vehicles. The general approach is described which is applicable to all kinds of ocean vehicles. In the end, the derived general relations are adjusted to fit the special case of the model sized hovercraft.

5.1 Kinematic Model Development

In this subsection the kinematic equations of an ocean vehicle are derived. In figure 5.1 a picture of an ocean vehicle is presented with its degrees of freedom. It is clear that the vehicle has six degrees of freedom, DOF. Three translational: surge, sway, heave, and three rotational: roll, pitch, yaw, in the body-fixed reference frame. It is convenient to define two coordinate systems. A body-fixed reference frame $X_0Y_0Z_0$, and an earth-fixed reference frame $XYZ$ is defined, as is indicated in figure 5.1. The origin $O$ of the body-fixed frame is usually chosen to coincide with the center of gravity of the body. For the hovercraft the axes of the body-fixed reference frame coincide with the principal axes of inertia. The body-fixed frame $X_0Y_0Z_0$ is defined as:

- $X_0$ - longitudinal axis (directed from aft to fore)
- $Y_0$ - transverse axis (directed to starboard)
- $Z_0$ - normal axis (directed from top to bottom)
The earth-fixed reference frame $XYZ$ is assumed to be inertial so the movement of the earth can be neglected. This means that no forces caused by the motion of the earth act on the hovercraft. Because of this the position and orientation of the hovercraft should be described relative to the inertial earth-fixed reference frame while the linear and angular velocities are considered in the body-fixed frame. The notation used to describe the position, velocity and force in the two different reference frames are depicted in table 5.1. It is convenient to use a vector notation to describe the flight path of the hovercraft.

<table>
<thead>
<tr>
<th>DOF</th>
<th>forces and moments</th>
<th>linear and angular vel.</th>
<th>positions and Euler angles</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>surge</td>
<td>$X$</td>
<td>$x$</td>
</tr>
<tr>
<td>2</td>
<td>sway</td>
<td>$Y$</td>
<td>$v$</td>
</tr>
<tr>
<td>3</td>
<td>heave</td>
<td>$Z$</td>
<td>$w$</td>
</tr>
<tr>
<td>4</td>
<td>roll</td>
<td>$K$</td>
<td>$p$</td>
</tr>
<tr>
<td>5</td>
<td>pitch</td>
<td>$M$</td>
<td>$q$</td>
</tr>
<tr>
<td>6</td>
<td>yaw</td>
<td>$N$</td>
<td>$r$</td>
</tr>
</tbody>
</table>

Table 5.1: Notation used to describe position, velocity and force of the hovercraft
Therefore the following vectors are defined:

\[ \eta = [\eta_1^T, \eta_2^T]^T \]
\[ \nu_1 = [u, v, w]^T \]
\[ \nu_2 = [p, q, r]^T \]
\[ \sigma_1 = [X, Y, Z]^T \]
\[ \sigma_2 = [K, M, N]^T \]

(5.1)

With \( \eta \) the position and orientation vector of the hovercraft (earth-fixed frame). \( \nu \) is a vector containing the linear and angular velocities (body-fixed frame). \( \tau \) is a vector used to describe the forces acting on the hovercraft (body-fixed frame). In marine guidance and control subjects, Euler angles are often used to describe the orientation of a body. The method can be used to relate the body-fixed reference frame with the earth-fixed frame.

### 5.2 Rigid Body Dynamics

The equations of motion for the center of mass (COM) of a hovercraft can be derived with the Newton-Euler method. The method is based on Newton's second law:

\[ m\ddot{v}_c = f_c \]  

(5.2)

With \( m \) the mass and \( \ddot{v}_c \) the acceleration of the body. \( f_c \) is the force acting on the body. Euler proved that equation 5.2 can be separated in a linear, \( \dot{p}_c \), and an angular momentum part, \( \dot{\mathbf{h}}_c \), known as Euler's first and second axioms respectively:

\[ \dot{p}_c \triangleq f_c \]
\[ \dot{\mathbf{h}}_c \triangleq M_c \]
\[ \mathbf{h}_c \triangleq I_c \omega \]

(5.3)

\( f_c \) and \( M_c \) are the forces acting on the body's center of gravity. \( m \) is the mass and \( I_c \) is the inertia tensor of the body. \( \dot{v}_c \) and \( \omega \) are the linear and angular velocity respectively. With \( \dot{v}_c = \nu_1 \) and \( \omega = \nu_2 \)

Equation 5.3 can be used to derive the rigid body equations of motion of the hovercraft. First a few assumptions will be made.

1. The hovercraft is rigid thus no forces act between individual elements of mass.
2. The mass and moment of inertia of the hovercraft are assumed to be constant: \( \dot{m} = 0, \dot{I}_c = 0 \)
3. The earth-fixed reference frame is inertial. This means that non of the forces acting on the hovercraft are a result of the movement of the earth.
4. The origin of the body-fixed reference frame is chosen to coincide with the center of gravity.
Consider any given rigid body with a body-fixed coordinate system, $X_0Y_0Z_0$ rotating with an angular velocity, $\omega = [\omega_1 \ \omega_2 \ \omega_3]^T$ about an earth-fixed coordinate system, $XYZ$. The rigid body with the two coordinate systems is displayed in figure 5.2. The body’s inertia tensor, $I_0$ with respect to the body-fixed coordinate system can be defined as:

$$I_0 = \begin{bmatrix}
I_x & -I_{xy} & -I_{xz} \\
-I_{yx} & I_y & -I_{yz} \\
-I_{zx} & -I_{zy} & I_z
\end{bmatrix} ; \quad I_0 = I_0^T > 0 \quad (5.4)$$

with $I_x$, $I_y$, and $I_z$ the moments of inertia about the $X_0$, $Y_0$ and $Z_0$ axis. $I_{xy}$, $I_{yx}$, $I_{xz}$, $I_{zx}$, $I_{yz}$, $I_{zy}$ are the products of inertia. The separate components of the inertia tensor are defined in the following way:

$$
I_x = \int_V (y^2 + z^2) \rho_A dV \\
I_y = \int_V (x^2 + z^2) \rho_A dV \\
I_z = \int_V (x^2 + y^2) \rho_A dV \\
I_{xy} = \int_V xy \rho_A dV = \int_V yx \rho_A dV = I_{yx} \quad (5.5) \\
I_{xz} = \int_V xz \rho_A dV = \int_V zx \rho_A dV = I_{zx} \\
I_{yz} = \int_V yz \rho_A dV = \int_V yz \rho_A dV = I_{zy}
$$

with $\rho_A$, the mass density of the rigid body.

As a consequence the inertia tensor, $I_0$ can be written in vectorial form:

$$I_0\omega = \int_V r \times (r \times \omega) \rho_A dV \quad (5.6)$$

The mass of the body is:

$$m = \int_V \rho_A dV \quad (5.7)$$

Figure 5.2: Earth-fixed and body-fixed coordinate system
As already mentioned it is assumed that the mass is constant. Therefore the distance from the origin \( O \) of the rigid-body to the center of gravity is:

\[
r_g = \frac{1}{m} \int_V r \rho_A dV \quad (5.8)
\]

To derive the equations of motion for an arbitrary point in a local body-fixed rotating coordinate system we need the formula:

\[
\dot{c} = \dot{\hat{c}} + \omega \times c
\]
relating the time derivative of a vector \( c \) in the body-fixed reference frame, \( X_0Y_0Z_0 \) with the earth-fixed frame, \( XYZ \). \( \dot{c} \) is the time derivative of \( c \) in earth-fixed coordinates and \( \dot{\hat{c}} \) the time derivative of \( c \) in the moving reference frame. Equation 5.9 yields:

\[
\dot{\omega} = \dot{\hat{\omega}} + \omega \times \omega = \hat{\omega}
\]

In the next section only the translational movement will be considered. In section 5.4 the rotational modes will be discussed.

### 5.3 Translational Motion

If figure 5.2 is used again, it can be seen that:

\[
r_c = r_0 + r_g
\]

thus the velocity of the center of gravity is:

\[
v_c = \dot{r}_c = \dot{r}_0 + \dot{r}_g
\]

In the enumeration on page 35 we assumed that the hovercraft is a rigid body. This means that the two points \( O \) and \( CG \) in figure 5.2 maintain the same orientation relative to each other so \( v_0 = \dot{r}_g \) and \( \dot{r}_g = 0 \) thus:

\[
v_c = v_0 + \omega \times r_g
\]

For vector \( v \) the same procedure is followed resulting in the following expression:

\[
v = v_0 + \omega \times r
\]

Now the acceleration vector is:

\[
\dot{v}_c = \dot{v}_0 + \dot{\omega} \times r_g + \omega \times \dot{r}_g
\]

With equation 5.9, 5.10 and \( \dot{r}_g = 0 \) yielding:

\[
\dot{v}_c = \dot{v}_0 + \omega \times v_0 + \dot{\omega} \times r_g + \omega \times (\omega \times r_g)
\]
When this expression is substituted in equation 5.3 the following relation is found:

$$m \left( \dot{v}_0 + \omega \times v_0 + \dot{\omega} \times r_g + \omega \times (\omega \times r_g) \right) = f_0$$  \hspace{1cm} (5.17)

Recall that the origin of the body-fixed reference frame $X_0Y_0Z_0$ of the rigid-body is chosen to coincide with the center of gravity. Therefore the vector $r_g = [0,0,0]^T$. Applying this to equation 5.17 together with the fact that $f_0 = f_c$ and $v_0 = v_c$ yields:

$$m \left( \dot{v}_c + \omega \times v_c \right) = f_c$$  \hspace{1cm} (5.18)

### 5.4 Rotational Motion

The approach used in the previous section can also be used to derive the rotational equations of motion. Euler's second axiom from equation 5.3 will be used. The rotational equations of motion of the rigid body in figure 5.2 are discussed with respect to the origin O. The angular momentum can be described as:

$$h_0 = \int \mathbf{r} \times \rho A dV$$  \hspace{1cm} (5.19)

This expression can be differentiated with respect to time yielding:

$$\dot{h}_0 = \int \mathbf{r} \times \dot{\rho} A dV + \int \mathbf{r} \times \rho A dV$$  \hspace{1cm} (5.20)

The first term of the right hand side of equation 5.20 is the standard expression for the mass of a rigid body:

$$m_0 = \int \rho A dV$$  \hspace{1cm} (5.21)

This equation can be rewritten as follows:

$$h_0 = m_0 - v_0 \times \int \rho A dV$$  \hspace{1cm} (5.22)

using the fact that $v \times v = 0$. And in figure 5.2 it can be seen that:

$$v = \dot{r}_0 + \dot{r} \quad \Rightarrow \quad \dot{r} = v - \dot{r}_0$$  \hspace{1cm} (5.23)

Now use $v = \dot{r} + \dot{v}_0$ from equation 5.23 to replace $v$ in equation 5.22. The result is:

$$\dot{h}_0 = m_0 - \dot{v}_0 \times \int (\dot{v}_0 + \dot{r}) \rho A dV = m_0 - \dot{v}_0 \times \int \dot{r} \rho A dV$$  \hspace{1cm} (5.24)

Using equation 5.8, a substitution for $\int \dot{r} \rho A dV$ in equation 5.24 can be derived. The time derivative of equation 5.8 is:

$$m \ddot{r}_g = \int \dot{r} \rho A dV$$  \hspace{1cm} (5.25)
CHAPTER 5. MODELLING OF THE HOVERCRAFT

with \( \dot{r}_g = \omega \times r_g \), equation 5.25 becomes:

\[
\int_V \dot{r}_g \rho_A dV = m (\omega \times r_g) \tag{5.26}
\]

Now it is clear that equation 5.24 with 5.26 can be written as:

\[
\dot{h}_0 = m_0 - m v_0 \times (\omega \times r_g) \tag{5.27}
\]

When we go back to equation 5.19 and recall from equation 5.14 that \( v \) can be substituted by \( v_0 + \omega \times r \) resulting in:

\[
h_0 = \int_V r \times v_0 \rho_A dV = \int_V r \times v_0 \rho_A dV + \int_V r \times (\omega \times r) \rho_A dV \tag{5.28}
\]

The right-hand side of this relation consists of two parts. The first part can be rewritten by using equation 5.8 and the second part by definition 5.6:

5.8: \( \int_V r \times v_0 \rho_A dV = (\int_V r \rho_A dV) \times v_0 = m r_g \times v_0 \)

5.6: \( \int_V r \times (\omega \times r) \rho_A dV = I_0 \omega \)

Equation 5.28 is now simplified to:

\[
h_0 = I_0 \omega + m r_g \times v_0 \tag{5.30}
\]

When this equation is differentiated according to equation 5.9, assuming that \( I_0 \) is constant with respect to time, the following relation is obtained.

\[
\dot{h}_0 = I_0 \dot{\omega} + \omega \times I_0 \omega + m \left( \omega \times r_g \right) \times v_0 + m r_g \times (\dot{v}_0 + \omega \times v_0) = M_0 \tag{5.31}
\]

Combining equation 5.31 and 5.27, knowing that \( (\omega \times r_g) \times v_0 = -v_0 \times (\omega \times r_g) \) yields:

\[
I_0 \dot{\omega} + \omega \times I_0 \omega + m r_g \times (\dot{v}_0 + \omega \times v_0) = M_0 \tag{5.32}
\]

The same as for the translational motion, the origin of the body-fixed reference frame \( X_0 Y_0 Z_0 \) of the rigid-body is chosen to coincide with the center of gravity. Therefore the vector \( r_g = [0, 0, 0]^T \). Applying this to equation 5.32 yields:

\[
I_0 \dot{\omega} + \omega \times I_0 \omega = M_c \tag{5.33}
\]

5.5 Adjustment for Model Sized Hovercraft

Equation 5.18 and 5.33 are in vector form and can be separated in 6 equations for the 6 degrees of freedom. In section 5.1 we defined the variables used to denote the various degrees of freedom, their time derivatives and the forces working in the corresponding direction. The vector \( v_c \) is the translational velocity and corresponds with \( v_1 \) from
section 5.1. In the same way also holds: $\omega = \nu_2$, $f_c = \tau_1$ and $M_c = \tau_2$. Substituting these vectors in equations 5.18 and 5.33 for translational and rotational movement respectively and working out the vector product yields:

**translational:**

$$m (\dot{u} - vr + uq) = X$$
$$m (\dot{v} - wp + ur) = Y$$
$$m (\dot{w} - up + vp) = Z$$

And **rotational:**

$$I_x \dot{p} + (I_z - I_y) qr - (\dot{r} + pq) I_{xz} + (r^2 - q^2) I_{yz} + (pr - q) I_{xy} = K$$
$$I_y \dot{q} + (I_x - I_z) rp - (\dot{p} + qr) I_{xy} + (p^2 - r^2) I_{xz} + (qp - r) I_{yz} = M$$
$$I_z \dot{r} + (I_y - I_x) pq - (\dot{q} + rp) I_{yz} + (q^2 - p^2) I_{xy} + (qp - p) I_{xz} = N$$

The model is very complex and the more difficult the model the harder it will be to design a controller. It is not necessary to use all 6 degrees of freedom. For the hovercraft we can neglect the heave, roll and pitch mode. The heave is not relevant since the hovercraft is operated on an even surface. Although there is some displacement in Z direction when the lift motor is turned on or off but that has no influence on the behavior of the hovercraft. The roll and pitch of the hovercraft are negligible for the same reason; the surface on which the hovercraft is operated is even. The hovercraft is very stable and therefore the roll and pitch modes are hardly present on an even surface. They can be present very shortly if the lift force is adjusted but are small very fast so they can be neglected. Only the surge, sway and yaw displacements remain in the model.

Equation 5.34 and 5.35 reduce to:

$$m (\dot{u} - vr) = X$$
$$m (\dot{v} + ur) = Y$$
$$I_z \dot{r} = N$$

### 5.6 Body-fixed to Earth-fixed Transformation

The completed model of equation 5.36 is in body-fixed coordinates. To transform the speed of the vehicle in its body-fixed reference frame to a position in the earth-fixed reference frame a coordinate transformation is used. The transformation follows from figure 5.3 and is:

$$u = \cos(\psi) \dot{x} + \sin(\psi) \dot{y}$$
$$v = -\sin(\psi) \dot{x} + \cos(\psi) \dot{y}$$
$$r = \dot{\psi}$$

And three states are added to go from velocities to positions in the earth-fixed coordinate system:

$$z_1 = \cos(\psi) x + \sin(\psi) y$$
$$z_2 = -\sin(\psi) x + \cos(\psi) y$$
$$z_3 = \psi$$
These states will be added to the three states that we already had in the model of 5.36. Also damping and friction are added to the model to get a more realistic model. Yielding a new state vector:

\[
\begin{bmatrix}
  z_1 \\
  z_2 \\
  z_3 \\
  u \\
  v \\
  r
\end{bmatrix}
\begin{bmatrix}
  \dot{z}_1 \\
  \dot{z}_2 \\
  \dot{z}_3 \\
  \dot{u} \\
  \dot{v} \\
  \dot{r}
\end{bmatrix}
= \begin{bmatrix}
  u + z_2 r \\
  v - z_1 r \\
  r \\
  ur - \frac{d_{11}}{m} u + X - F_{w1} \\
  -ur - \frac{d_{22}}{m} v + Y - F_{w2} \\
  -\frac{d_{33}}{I_y} r + N - F_{w3}
\end{bmatrix}
\]  

(5.39)

with vector \((x, y, \psi)\) the positions and angle in earth-fixed and \(uvw\) velocities in the body-fixed reference frame. \(d_{11}, d_{22}\) and \(d_{33}\) are damping coefficients. \(F_{w1}, F_{w2}\) and \(F_{w3}\) are friction forces. When simulations are done the results can be easily transformed to earth coordinates since the relation between \(xyz\) and \(uvw\) is straightforward.

The forces \(X, Y\) and \(N\) can be rewritten as a function of the lift force and the back thrust force. With the position of the rudder and the constant back thrust \(F_{ax}\) in figure 5.3 it can be seen that half of \(F_{ax}\) always passes the rudder. The other half is deflected depending on the angle of the rudder. The deflected force can be divided in a part that acts in surge direction and a part that acts is sway that also causes a moment in yaw.
CHAPTER 5. MODELLING OF THE HOVERCRAFT

5.7 Hovercraft Parameter Estimation and Identification

5.7.1 Mass

The mass of the hovercraft is measured using a balance. It has a weight of 2.1 kg.

5.7.2 Moment of Inertia

The moment of inertia \( J \) can be determined both experimentally and mathematically. In the following section, first the experimental approach will be explained followed by the mathematical approach.

Experimental

First the basic idea behind the inertia measurements will be explained followed by a description of the test-rig used to perform the experiments. Finally the collected data is used to calculate the moment of inertia.
From previous sections it is clear that the moment of inertia of interest is the one around the z-axis. To determine this moment of inertia a moment $M$ is applied onto the hovercraft causing it to have an angular acceleration $\dot{\omega}$. The hovercraft needs to be supported in such a way that only rotation around the z-axis through the center of gravity of the hovercraft is possible. It is assumed that there is no friction and from experiments that were performed on the test-rig before, it shows that friction influences can be neglected.

In figure 5.4 the test-rig's schematic is displayed. For this electromechanical system the dynamic model can be derived (see also Nise [3] and Ogata [4] for more information).

The relationship for the voltages in the armature circuit is:

$$V = IR_a + L_a \frac{dI}{dt} + V_b$$

(5.42)

With $V$ the applied armature voltage, $I$ the armature current, $R_a$ the armature resistance, $L_a$ the armature inductance, $V_b$ the back electromotive force (back emf). The relation between $V_b$ and the rotational speed of the motor $\omega$ is

$$V_b = K_b \omega$$

(5.43)

$K_b$ is a constant of proportionality called the back emf constant. The torque $T_m$ developed by the motor is proportional to the armature current:

$$T_m = K_t I \quad \Rightarrow \quad I = \frac{T_m}{K_t}$$

(5.44)

$K_t$ is a constant of proportionality called the motor torque constant which depends on the motor and magnetic field characteristics. To find the transfer function of the motor the Laplace transformed equations 5.43 and 5.44 are substituted into equation 5.42, yielding:

$$\frac{(R_a + L_a s) T_m}{K_t} + K_b \omega = V$$

(5.45)
Chapter 5. Modelling of the Hovercraft

The next step is to find $T_m$ in terms of $\omega$. From figure 5.4:

$$J \omega + b \omega = T_m$$

(5.46)

In the Laplace domain this yields:

$$(Js + b) \omega = T_m$$

(5.47)

with $J$ the moment of inertia and $b$ the viscous damping coefficient. Substituting this result in 5.45 gives:

$$\frac{(R_a + L_a s)(Js + b) \omega}{K_t} + K_i \omega = V$$

(5.48)

The transfer function from $V$ to $\omega$ is now:

$$\frac{\omega}{V} = \frac{K_t}{(R_a + L_a s)(Js + b) + K_b K_t}$$

(5.49)

In order to determine $J$ the step response of the system in figure 5.4 is measured. Both with and without the hovercraft to be able to determine the inertia of the hovercraft without the inertia of the test-rig. The inertia of the hovercraft is the inertia of the hovercraft and rig together minus the inertia of the rig alone. The inertia of the hovercraft has to be multiplied by the gearbox ratio to get the real inertia because all measurements are done at the input axis and we are interested in the inertia at the output axis of the gearbox. Next the step response is derived analytical. For this expression all values are known except for $J$, thus fitting this relation to the measured step response supplies the desired moment of inertia.

To calculate the step response of the transfer function in equation 5.49, it is possible to transform this equation into

$$\frac{\omega}{V} = \frac{K_t}{La J (s + p_1)(s + p_2)}$$

(5.50)

With $-p_1$ and $-p_2$ the poles of the transfer function of 5.50. The poles are

$$p_1, p_2 = -\frac{(R_a J + L_a b) \pm \sqrt{(R_a J + L_a b)^2 - 4 L_a J (R_a b + K_b K_v)}}{2L_a J}$$

(5.51)

The Laplace representation of the step response with stepsize $a$ is

$$\frac{\omega}{V} = \frac{a K_t}{La J s(s + p_1) (s + p_2)}$$

(5.52)

Transforming this equation back to the time domain yields the step response:

$$\omega(t) = \frac{a K_t}{La J p_1 p_2} \left( 1 + \frac{p_2}{p_1 - p_2} \exp(-p_1 t) - \frac{p_1}{p_1 - p_2} \exp(-p_2 t) \right)$$

(5.53)
For these formulas we need the parameters $L_a$, $R_a$, $K_b$, $K_t$, b. These will be determined:

The armature inductance $L_a$ of the motor was given in the data sheet.

$L_a = 62c - 3$ H.

To determine the other parameters, two experiments can be performed. During the first experiment, done to determine the armature resistance $R_a$, different voltages $V$ are supplied to the motor applying a dummy load to it to make sure the motor doesn't rotate. Then the current is measured. Because the motor doesn't rotate the back emf $V_b$ is zero. The equation 5.42 reduces to:

\[ V = I \cdot R_a + L_a \frac{dI}{dt} \]  

(5.54)

The experiments to determine the $R_a$ need to be done in steady state so the current is constant and the last term of equation 5.54 cancels out of the equation. Thus from equation 5.54 we can write:

\[ R_a = \frac{V}{I} \]  

(5.55)

The results of this experiment are displayed in appendix M table M.1

During the second experiment the other parameters are determined. The motor is allowed to rotate with constant angular velocity and again the voltage and the current are measured and also the steady state angular velocity $\omega$ (see the columns 2 3 5 in table M.2 of appendix M) Recall from equation 5.42 that the formula for the voltages in the circuit is:

\[ V = I \cdot R_a + L_a \frac{dI}{dt} + V_b \quad \Rightarrow \quad V_b = V - I R_b \]  

(5.56)

The inductance term can be neglected because $L_a$ is relatively small compared to the other terms. The voltage $V$ and the current $I$ are measured and the resistance $R_a$ is known so the back emf can be determined. The back emf constant $K_b$ is defined as:

\[ K_b = \frac{V_b}{\omega} \]  

(5.57)

The torque $T_m$ can be determined from the torque constant $K_t$ which is equal to the back emf constant

\[ K_t = K_b \]  

(5.58)

and the current $I$, because:

\[ T_m = K_t \cdot I \]  

(5.59)

Finally $b$ can be determined from equation 5.46

\[ J \ddot{\omega} + b \omega = T_m \quad \Rightarrow \quad b = \frac{T_m}{\omega} \]  

(5.60)
Again because these experiments were done during steady state $\dot{\omega}=0$. The data for the experiment with and without the hovercraft can be seen in appendix M.

To determine the moment of inertia experimentally, a test rig in the laboratory of mechanical engineering is used. It is a Maxon DC motor. It can be controlled of a computer and the input shaft has an encoder with a resolution of 8192 counts per revolution, so the position of the input-shaft can be measured. Connected to the motor is a gearbox with a transmission of 19.2:1. The output-shaft of the gearbox has a flange with three holes on which a load can be attached.

To do the experiment with hovercraft it needs to be mounted on the flange of the test-rig. The center of gravity is very important. It must be above the center of the turning axis. During the experiment the hovercraft undergoes high accelerations. Therefore it must be fitted very tight to the test-rig. A plate with two disks is made. The two holes in the bottom of the hovercraft fit exactly around the disks. The hovercraft only rotates around the Z axis thus the two disks are enough to secure the hovercraft to the test-rig. In appendix N a drawing of the plate is displayed. In this picture the position and diameter of the holes of the flange can be seen. During first stage of acceleration the motor must first go through the backlash in the gear box. This has a negative effect on the accuracy of the measurements. By turning the motor a little bit so the gear box already goes through the backlash before the experiment, it is tried to reduce the effects as much as possible.

All parameters of equation 5.53 are known so now the two step responses of the motor with the hovercraft and without it can be measured. To that data equation 5.53 is fitted. The measurement data and the fits for the two experiments are shown in figure 5.5. The first part of the measurement data represents clearly the backlash in the gearbox. Especially for the test with hovercraft. The fit is performed using the MATLAB function \texttt{fminsearch}. \texttt{fminsearch} minimizes the output of a function specified in a separate function-file. When performing a fit the difference between the fit and the measured data should be as minimal as possible. Therefore a function \texttt{Fit.J.m} (see appendix O) is written which computes the quadratic error depending on the inertia $J$. In \texttt{Determine.J.m} (see appendix O) first the fit conditions are specified before the \texttt{fminsearch} algorithm is called. It starts with the initial value $J = 0.001$. Finally the measured data and the fit are compared by plotting them both in the same figure. Of course the parameters $a$, $k_t$ and $b$ need to be adjusted to fit the moment of inertia of the test-rig without the hovercraft or with the hovercraft on it. It yields a moment of inertia of:

\begin{equation}
J = 0.000552 - 0.000295 = 0.000257 \quad (5.61)
\end{equation}

After the gearbox this is 0.0948 kgm².
The moment of inertia \( J \) can also be calculated. This is done to verify the value that is found during the experiments in the previous section. The Hovercraft is divided into components for which the moment of inertia is calculated. The total moment of inertia can be calculated by translating the separate moments of inertia to the center of gravity of the hovercraft with the use of Steiner. The hovercraft is supposed to be build up out of 5 components: The main board, the Battery pack, the lift motor, the back thrust motor and the camera. The Main board, the Battery pack and the Back thrust motor are supposed to be a rectangular whereas the Lift motor and the Camera are supposed to be cylinders. In appendix P table P.1 the measures of the components are given. Figure P.1 gives a schematic representation of the position of the separate parts of the hovercraft on the Main plate.

The formula used to compute the moment of inertia of a beam around the \( z \)-axis:

\[
J_r = \frac{m}{12} (a^2 + b^2) + ml_z^2
\]

(5.62)

with \( l \) the length and \( b \) the width of the beam and \( m \) the mass \( l_z \) is the distance from the center of gravity of the beam to the center of gravity of the hovercraft.

And for a cylinder around the \( z \)-axis:

\[
J_c = \frac{m}{2} r^2 + ml_z^2
\]

(5.63)
CHAPTER 5. MODELLING OF THE HOVERCRAFT

Figure 5.6: Beam and cylinder to compute the moment of inertia

\[ J_c = \frac{m}{12} (3r^2 + h^2) + m l_z^2 \]  

(5.64)

with \( m \) mass, \( r \) radius \( l_z \), the distance from the center of gravity of the cylinder to the center of gravity and \( h \) the height of the cylinder. In table 5.7.2 the results of the moment of inertia calculations of the various parts of the hovercraft, are displayed.

<table>
<thead>
<tr>
<th>Part</th>
<th>Moment of inertia with respect to CG hovercraft [kgm^2]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Main plate</td>
<td>0.0642</td>
</tr>
<tr>
<td>Battery pack</td>
<td>0.0116</td>
</tr>
<tr>
<td>Lift motor</td>
<td>1.1250e-005</td>
</tr>
<tr>
<td>Back thrust motor</td>
<td>0.0055</td>
</tr>
<tr>
<td>Camera</td>
<td>0.0036</td>
</tr>
<tr>
<td>Total hovercraft</td>
<td>0.0849</td>
</tr>
</tbody>
</table>

Table 5.2: Moment of inertia calculated for the separate parts of the hovercraft

Comparison

If we compare the value obtained via the empirical way with the mathematical one, we see that the mathematical is approximately 10%. This can be explained by the fact that only a few components of the hovercraft are taken into account with the mathematical approach to calculate the moment of inertia. If all other components like receiver, servos and rudder are included in the calculation, the moment of inertia found by the mathematical approach, is expected to be much higher and therefore closer to the value
found via the empirical way. Therefore we assume that the value for the moment of inertia for the hovercraft is more accurately determined via the empirical way and will use this value from now on for the model of the hovercraft.

\[ J = 0.0948 \text{ kgm}^2 \] (5.65)

There are more reasons why the results of the mathematical approach should only be used as an indication to verify the empirical method. The sources that cause an error in determining the moment of inertia mathematical are:

- All components that are included in the calculation are approximated by a beam or a cylinder.
- All components that are included in the calculation are assumed to have a homogenous mass distribution
- All dimensions and distances to the centre of gravity of the components need to be measured.

On the other hand the empirical method doesn’t deliver a correct moment of inertia either. It has also a great deal of uncertainty since there is backlash in the gearbox (as already indicated this can be seen in the figure 5.5) and there are sensors that can have noise etc. But we assume that the empirical method is much closer to the actual moment of inertia of the hovercraft than the mathematical method.

### 5.7.3 Forces of the Motors

The two input forces that can be used to control the hovercraft are a force of the thrust motor in the horizontal plane and of the lift motor in the vertical plane. In the current configuration the back thrust has a constant magnitude of approximately 1.8 N. This is determined with a modern type of steelyard. With full lift and therefore almost no friction and a rudder angle of zero, the static force of the hovercraft on the steelyard is measured. The direction of the thrust force can be adjusted by turning the rudder in a desired direction by a certain amount. A maximum of about 60 degrees in both directions from the neutral position. In the body-fixed reference frame the horizontal force can work in both the X and the Y direction depending on the magnitude of the angle of the rudder \( \delta \). As a result the back thrust force is used, not only to control the surge velocity, but also the direction the hovercraft goes to. In the model of 5.40 this force is called \( F_{x,x} \). In the third equation of equation 5.40 the arm \( a \) is the distance between the point the force of the turned rudder acts on to the center of gravity. This way the moment around the z-axis produced by the force at an arm \( a \) can be calculated. \( a = 0.4 \text{ m} \).

The second, vertical force, is adjustable to determine the degree in which the hovercraft is lifted off the ground. If the lift thrust is small, the hovercraft has a lot of friction.
The amount of friction can be used to control the speed of the hovercraft. This way the hovercraft responds quicker to desired changes in speed. If a hovercraft with a normal control configuration (adjustable back thrust), needs to stop, the back thrust is turned off. Because of the very low friction of the hovercraft it takes a while to stop. If the lift thrust is turned off the hovercraft stops almost immediately. The lift force $F_l$ in the model of 5.41 is chosen to be relative to the gravity force of the hovercraft $mg = \pm 21$ N if $F_l$ is 21 that means that the friction is zero (see equation 5.41). Changing $F_l$ from 0 to 21 the frictional resistance can be controlled.

5.7.4 Damping

Forces that can occur as a result of current, waves or wind are neglected since the hovercraft is operated in a closed room on a flat surface. It is assumed that there are no obstacles in the surface. The only remaining force is a damping force.

The damping $d_{11}$ can be determined by the results of the first experiments of section 4. The hovercraft has a constant speed only in surge direction with maximum lift. With equation 5.39 we see that:

$$\dot{u} = v_x - \frac{d_{11}}{m} u + X - F_{w1}$$

This can be, together with equation 5.40, written as:

$$\frac{d_{11}}{m} u = \frac{1}{2m} (1 + \cos(\delta)) F_{xx}$$

Therefore, because $\delta = 0^\circ$ and $F_{xx} = 1.8$ N and the surge velocity is 3 m/s, $d_{11} = 1.8/3 = 0.6$ Ns/m.

The damping in sway velocity can’t be determined this way because it is not possible to have a force that only acts in sway direction. It is assumed that the damping is mostly due to drag of the hovercraft. Since the surface determines the amount of drag and the side surface of the hovercraft is bigger than the front surface, $d_{22}$ is estimated to be 0.8 Ns/m.

$d_{33}$ is not known and isn’t determined empirical therefore a reasonable value of 0.1 is chosen.

5.7.5 Friction

The friction forces $F_{w1}$, $F_{w2}$ and $F_{w3}$ depend on the lift force $F_l$ but are scaled by a factor $\mu$ for the three different degrees of freedom (equation 5.41). These coefficients are determined during simulations. The following values were found:

$\mu_1 = 0.100$

$\mu_2 = 0.010$

$\mu_3 = 0.004$
Chapter 6

Simulations with the Hovercraft Model

The model is implemented in an m-file to simulate the behavior of the hovercraft. Matlab 6.1 is used to perform the simulation calculations. Modelsim.m contains the model and a vector of inputs for the various time periods of the simulation. Model.m has the solving and plotting statements. Shippplot.m is only used to plot hovercraft shapes in the trajectory to see the orientation of the hovercraft at a certain point in the XY plane at a certain time.

For the simulations the same inputs are chosen as for the experiments except that the rudder angle is now -10 instead of 10 degrees. The reason for this is discussed further on. These inputs are presented together with the results of the simulations in figure 6.2. This way the response can be related to the input in a straightforward way. Only the time path can differ from the experiments. The first 5 seconds of the simulation lift force is maximum and rudder angle is zero. From 5 till 40 seconds lift maximum and rudder angle is -10 degrees. From 40 till 50 seconds lift is reduced to half power and rudder angle is -10 degrees. From 50 till 60 seconds lift is reduced to zero and rudder angle is -10 degrees. It is possible to compare the results of the simulations with those of the experiments to verify the model. Attention has to be given to the fact that the positive y axis is defined pointing down (see figure 5.3) during the determination of the hovercraft model. Matlab however always plots the positive y axis pointing up (see figure 6.1) thus in mind this has to be flipped. Therefore we now use a rudder angle of -10 degrees to get a hovercraft that moves down turning clockwise as with the experiments. Now it is easier to compare the two figures. Since the hovercraft is assumed to be symmetrical with respect to the surge axes, this is no problem.

Validation of the Model

In the future the hovercraft is going to be controlled using Vision and Control. A good model is necessary to design a controller so we need to know the quality of the model. We look especially at the trajectory that the hovercraft describes since it is difficult
CHAPTER 6. SIMULATIONS WITH THE HOVERCRAFT MODEL

Figure 6.1: Movement of the hovercraft in earth-fixed reference frame. Blue: full lift thrust and rudder angle of 0°. Red: full lift thrust and rudder angle of -10°. Black: lift thrust is reduced to zero and angle is -10°.

Figure 6.2: Results of the simulations. Body fixed velocities surge, sway and yaw are presented separately and the movement of the hovercraft in earth plane is given without hovercraft. The surge, sway and yaw plots each have three axes. On the left the velocity axis and on the right the input axes of the lift force and the rudder angle.
to measure the surge, sway and yaw velocities during an experiment. In figure 4.1 the picture of the experiments is presented which can be compared to the result of the simulation in figure 6.1. It is clear that the two trajectories are very much the same. The dimensions in the experiment are not completely comparable with those of the simulations because the camera is not completely perpendicular to the plane in which the hovercraft moves. Nevertheless the dimensions match relatively good. The speed on the straight end of the simulation goes to 3 m/s in figure 6.2-surge. The same order as found during the experiments. Also the orientation of the hovercraft is the same during experiment and simulations. A difference between experiment and simulations is the size of the hovercraft. The size in Shipplot.m (appendix Q) is chosen to get a clear figure where the hovercrafts don't overlap but during the experiments the hovercraft seems a lot bigger and therefore the circle a lot smaller. In fact the results of the simulations match those of the experiment relatively good. It can be concluded that the model can be trusted to describe the behavior of the model hovercraft.
Conclusion and Recommendations

Conclusion
The goal to connect the hovercraft to a PC is realized. The hovercraft is now open loop controllable of a computer. The software and hardware have been modified to make this possible. An attempt to make the hovercraft closed loop controllable was done, but because a driver for the camera was not available, this attempt failed. The camera is of vital importance because no other way to get data of the position and orientation of the hovercraft is available. The open loop test setup has been used to do experiments with the hovercraft to learn about its behavior. Results in the form of pictures are included in the report. From these experiments data like the maximum speed of the hovercraft in surge were collected.

Although no controller can be implemented as long as the camera and image processing part are not finished, a dynamical model for the hovercraft has been derived. The model can be used to simulate the behavior of the hovercraft and to design and test a controller. The Newton-Euler Method has been used for the modelling. The parameters of the hovercraft in the model have been determined by a combination of experiments and calculations. The 3 DOF dynamical model of the hovercraft has been used to simulate the movement of the hovercraft. The results of the experiments and the simulations have been compared with each other. It can be concluded that the model produces a similar trajectory as the experiments. Also the orientation and the speed in surge direction match. Although this method is very rough and it is hard to give a value about the quality of the model, it is concluded that the derived model is satisfactory.

Recommendations
An issue that needs attention in the future is the driver for the XiO camera. Once this is available, working on image processing and controller tuning can be started. To get a camera driver the progress on the internet can be followed to see if one becomes available shortly. On the other hand the choice to make an own driver can be made. To support this second possibility all available information about the camera is written down in this report.

The design of the hovercraft can be improved. The mass distribution is not optimal.
Conclusion and Recommendations

The front of the hovercraft touches the ground because the two batteries are too heavy causing more friction and making it behave differently. The incorrect mass distribution is responsible for the instability of the hovercraft, which is compensated by pieces of lead. To improve the situation one of the two batteries could be replaced to the back of the hovercraft.

Maybe the total design can be reconsidered to make the hovercraft more rigid and more reproducible. A different material can make it more rigid but also weight aspects must be considered. Now the batteries are replaced in a different place every time they are recharged. It might be an option to make a construction that ensures that the batteries are replaced at the same place every time. A possible solution can be to switch to fuel motors. But this has other disadvantages like noise and pollution. The camera is not thought about during the design of the hovercraft therefore it has no good place where it can be attached. In a new design the camera can be taken into account.

The model is maybe to rough for the vision and control task it is going to be used for. Other degrees of freedom can be included if necessary or it may turn out that other terms must be added apart from damping and friction. The parameters of the model have been determined as accurate as possible but perhaps other methods are possible. Especially for the damping and friction coefficients which were estimated.
Bibliography


Appendix A

c-program: mbuff.h

#ifndef MBUFF_H
#define MBUFF_H
#ifdef __cplusplus
extern "C" {
#endif

#define MBUFF_NAME_LEN 32
/* max number of attached mmaps per one area */
#define MBUFF_MAX_MMAPS 16

#ifndef SHM_DEMO
#define MBUFF_DEV_NAME "./mbuff"
#else
#define MBUFF_DEV_NAME "/dev/mbuff"
#endif

#ifdef __KERNEL__
#include <linux/types.h>
#include <linux/fs.h>
#else
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
#include <sys/ioctl.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <sys/stat.h>
#endif
#define MBUFF_VERSION "0.7.2"

/*
   All ioctl()s are called with name filled in with the appropriate
   name for the mbuff to be referenced. Calls to any ioctl() makes
   that mbuff "active", i.e., read(), write(), and mmap() use that
   mbuff. I didn't do this yet.

ioctl()s:

ALLOCATE:
   Call with size=0 to just find out if the area exists; no
   mbuff will be allocated. Otherwise, allocate an mbuff with
   that size.
DEALLOCATE:
   Decrease reference count for an mbuff.

issues:
   - using this method, it is *really* easy to get dangling
     mbuffs, i.e., mbuffs that nobody owns. When you close
     /dev/mbuff, it would be a good idea to decrease the ref
     count of the active mbuff.
*/
#define IOCTL_MBUFF_INFO 0
#define IOCTL_MBUFF_ALLOCATE 1
#define IOCTL_MBUFF_DEALLOCATE 2
#define IOCTL_MBUFF_SELECT 3
#define IOCTL_MBUFF_LAST IOCTL_MBUFF_SELECT

struct mbuf_request_struct{
    unsigned int flags;
    char name[MBUFF_NAME_LEN+1];
    size_t size;
    unsigned int reserved[4];
};

#endif __KERNEL__

/* you can use mbuf_alloc several times, the buffer
   will be deallocated when mbuf_free was called the same number of times
   AND area is not mmaped anywhere anymore
   AND it is not used in the kernel as well */
/* if you have a need to mmap the area at the specific address, use
   * mbuf_alloc_at */
inline void * mbuff_alloc_at(const char *name, int size, void *addr) {
  int fd;
  struct mbuff_request_struct req={0,"default",0,{0}};
  void * mbuf;

  if(name) strncpy(req.name,name,sizeof(req.name));
  req.name[sizeof(req.name)-1]='\0';
  req.size = size;
  if(( fd = open(MBUFF_DEV_NAME,0_RDWR) ) < 0 ){
    perror("open failed");
    return NULL;
  }
  size=ioctl(fd,IOCTL_MBUFF_ALLOCATE,&req);
  if(size<req.size) return NULL;
  /* the type of first mmap's argument depends on libc version? This really
   * drives me crazy. Man mmap says "void * start" */
  mbuf=mmap(addr, size, PROT_WRITE|PROT_READ, MAP_SHARED|MAP_FILE, fd, 0);
  if( mbuf == (void *) -1)
    mbuf=NULL;
  close(fd);
  return mbuf;
}

inline void * mbuff_alloc(const char *name, int size) {
  return mbuff_alloc_at(name, size, NULL);
}

inline void mbuff_free(const char *name, void *mbuf) {
  int fd;
  struct mbuff_request_struct req={0,"default",0,{0}};
  int size;

  if(name) strncpy(req.name,name,sizeof(req.name));
  req.name[sizeof(req.name)-1]='\0';
  if(( fd = open(MBUFF_DEV_NAME,0_RDWR) ) < 0 ){
    perror("open failed");
    return;
  }
  size=ioctl(fd,IOCTL_MBUFF_DEALLOCATE,&req);
  if(size > 0) munmap( mbuf, size);
  close(fd);
  /* in general, it could return size, but typical "free" is void */
  return;
}
/* mbuff_attach and mbuff_detach do not change usage counters - * area allocated using mbuff_attach will be deallocated on program exit/kill if nobody else uses it - mbuff_detach is not needed - the only lock keeping area allocated is mmap */

inline void * mbuff_attach_at(const char *name, int size, void * addr) {
  int fd;
  struct mbuff_request_struct req={0, "default", 0, {0}};
  void * mbuf;

  if(name) strncpy(req.name, name, sizeof(req.name));
  req.name[sizeof(req.name)-1]='\0';
  req.size = size;
  if(( fd = open(MBUFF_DEV_NAME, 0_RDWR) ) < 0 ){
    perror("open failed");
    return NULL;
  }
  ioctl(fd, IOCTL_MBUFF_ALLOCATE, &req);
  mbuf=mmap(addr, size, PROT_WRITE|PROT_READ, MAP_SHARED|MAP_FILE, fd, 0);
  /* area will be deallocated on the last munmap, not now */
  ioctl(fd, IOCTL_MBUFF_DEALLOCATE, &req);
  if( mbuf == (void *)-1)
    mbuf=NULL;
  close(fd);
  return mbuf;
}

inline void * mbuff_attach(const char *name, int size) {
  return mbuff_attach_at(name, size, NULL);
}

inline void mbuff_detach(const char *name, void * mbuf) {
  int fd;
  struct mbuff_request_struct req={0,"default",0,0};
  int size;

  if(name) strncpy(req.name,name,sizeof(req.name));
  req.name[sizeof(req.name)-1]='\0';
  if(( fd = open(MBUFF_DEV_NAME,0_RDWR) ) < 0 ){
    perror("open failed");
    return;
  }
  size=ioctl(fd, IOCTL_MBUFF_SELECT,&req);
  if(size > 0) munmap( mbuf, size);
  close(fd);
  /* in general, it could return size, but typical "free" is void */
  return;
}

#else
APPENDIX A. C-PROGRAM: MBUFF.H

```c
struct mbuf{
    struct mbuf *next;
    struct mbuf *prev;

    char name[MBUFF_NAME_LEN+1];

    struct vm_area_struct *(vm_area[MBUFF_MAX_MMAPS]);
    struct file *file;

    unsigned char *buf;
    unsigned long size;
    int count; /* number of allocations from user space */
    int kcount; /* number of allocations from kernel space */
    int open_cnt; /* #times opened */
    int open_mode;
};

extern struct mbuf * mbuf_list_lookup_name(const char *name,int priority);
extern struct mbuf * mbuf_list_lookup_buf(void *buf);
extern int shm_allocate(const char *name,unsigned int size, void **shm);
extern int shm_deallocate(void * shm);

static inline void * mbuf_alloc(const char *name, int size) {
    void *tmp=NULL;
    if( shm_allocate(name, size, &tmp) > 0 )
        return tmp;
    else
        return NULL;
}

static inline void mbuf_free(const char *name, void * mbuf) {
/* it would be no problem to deallocate using only name */
    shm_deallocate(mbuf);
}
/* in kernel space implementing "nonlocking" attach and detach
    would be very unsafe (deallocation from user space possible at any time) */
#define mbuff_attach(name,size) mbuf_alloc(name,size)
#define mbuf_detach(name,mbuf) mbuf_free(name,mbuf)

extern char mbuff_default_name[];
extern int mbuf_ioctl(struct inode *inode, struct file *file, unsigned int cmd,
unsigned long arg);
#ifdef LINUX_V22
extern int mbuf_mmap(struct file *file, struct vm_area_struct *vma);
#else
extern int mbuf_mmap(struct inode *inode, struct file *file,
struct vm_area_struct *vma);
#endif
```
extern int mbuf_open_with_name( struct inode *inode, struct file *file, const char *name);

#endif
#endif
#endif
Appendix B

c-program: cdsm.c

/*
 * File name: cdsm.c
 * Data Sharing interface
*/
#include "cdsm.h"
#ifdef __KERNEL__
#include <mbuff.h>
#include <linux/malloc.h>
#endif

inline void CDSM_init(void)
{
    CDSM_data = (volatile long*)mbuff_alloc(CDSM_DATA_SI,
                                            CDSM_NUMBER_OF_CHANNEL*sizeof(long));
}

inline void CDSM_done(void)
{
    mbuf_free(CDSM_DATA_SI,(void*)CDSM_data);
}

inline void CDSM_set(int chan, long data)
{
    *(CDSM_data+chan) = data;
}

inline long CDSM_get(int chan)
{
    return *(CDSM_data+chan);
}
Appendix C

c-program: cdsm.h

/*
 CDSM - Common Data Sharing Mechanism

 written by: Linh Vu
 (C) 2002
 */

#ifndef __CDSM_H__
#define __CDSM_H__

// maximum of 16 channel, can change to any number
#define CDSM_NUMBER_OF_CHANNEL 128

// data in string id
#define CDSM_DATA_SI "CDSM_DATA_SI"

// mechanism:
// common data sharing mechanism (CDSM)
//
// data will be shared among modules
// by read/write to an array of long type elements (also pointer)

volatile long *CDSM_data;

inline void CDSM_init(void);
inline void CDSM_done(void);
inline void CDSM_set(int chan, long data);
inline long CDSM_get(int chan);

#endif
Appendix D

c-program: cloop.c

```c
#ifndef __KERNEL__
#ifndef __RTL__
#include <tl_printf.h>
#endif /* __RTL__ */
#else
#include <stdio.h>
#endif /* __KERNEL__ */
#include <sys/io.h>
#include <linux/ioport.h>
#include "cloop.h"

/* Functions to operate on the cloop structure */

/* signal initialisation function */
void sig_init(volatile sig_t *signal)
{
    signal->y = 0.0;
    signal->u0 = lookup_sat(5);
    signal->u1 = lookup_sat(3);
    signal->r = 0.0,
    signal->time = 0.0;
}

void sig_copy(volatile sig_t *source, volatile sig_t *dest)
{
    dest->y = source->y;
    dest->u0 = source->u0;
    dest->u1 = source->u1;
    dest->r = source->r;
    dest->time = source->time;
}
```
APPENDIX D. C-PROGRAM: CLOOP.C

/* initialisation function (default) */
void cloop_init(volatile cloop_t *cloop)
{
    int i;

    for (i=0; i<MAX_DATA; i++) sig_init(&(cloop->cloop_sig[i]));
    cloop->time_index = 0; cloop->last = 0; cloop->num_data = 0;
    cloop->dt = DEF_INTERVAL;

    cloop->sig_low = -5; cloop->sig_hi = 5;
    for (i=0; i < 20; i++) {
        cloop->x[i] = 0.0;
        cloop->param[i] = 0.0;
    }

    cloop->setpt0 = lookup_sat(5);
    cloop->setpt1 = lookup_sat(3);
    cloop->ramp_increment = 0.0; cloop->rampcount = 0;
    cloop->input = 0;
    cloop->filter = 0.5;
    cloop->adc_chan = cloop->dac_chan = 0;
    cloop->inchannel = 0;
    cloop->cmode = 0;
    cloop->cstate = 0;
    cloop->initialised = 1;
    cloop->simulation = 0;
    cloop->datalog = 0;
    cloop->datastart = 0;
    cloop->dataend = -1;
    setup_io(cloop);
}

/* reset cloop signal values */
void cloop_reset(volatile cloop_t *cloop)
{
    int i;
    for (i=0; i<MAX_DATA; i++) sig_init(&(cloop->cloop_sig[i]));
}

/* clear cloop signal values */
void cloop_clear(volatile cloop_t *cloop)
{
    cloop_init(cloop);
    cloop->time_index = 0; cloop->last = 0; cloop->num_data = 0;
    cloop->datalog = 0; cloop->datastart = 0; cloop->dataend = -1;
    cloop->rampcount = 0;
}
 Appendix D. C-Program: Cloop.c  

/* Set the controller gains */
void set_params(volatile cloop_t *cloop, float *params, int num_params)
{
    int i;

    for (i=0; i < num_params ; i++) cloop->param[i] = params[i];
    cloop->num_params = num_params;
}

/* 3rd order Runge-Kutte integration */
float rk3(volatile cloop_t *cloop, fp f, float x, float u, float dt)
{
    int j;
    float s1, s2, s3, xx;
    float params[20];

    for (j = 0; j < 20; j++) params[j] = cloop->param[j];
    s1 = f(params,x,u);
    xx = x + dt * s1;
    s2 = f(params,xx,u);
    xx = x + dt * (s1 + s2) / 4.0;
    s3 = f(params,xx,u);
    xx = x + (s1 + 4.0 * s3 + s2) * dt / 6.0;
    return xx;
}

/* Function to sample A/D */
int io_adcin(volatile cloop_t *cloop, int channel)
{
    int low = 0, hi = 0, temp = 0, i;
    int data = 0, chan;

    chan = (channel) & 0x000F;
    chan = chan + (chan << 4);

    // Select our channel by writing our value to the MUX
    // => we start and finish on the same channel

    for (i=0; i<200; i++) outb(chan, IO_BASE+2);

    // clear to A/D register 1st
    outb(0, IO_BASE);

    // wait until the A/D conversion is complete (shouldn't
    // be necessary
while (inb(IO_BASE+8) & 0x80) {

    temp = inb(IO_BASE);
    low = (temp >> 4) & 0x000F;
    hi = (inb(IO_BASE+1) << 4) & 0x00FF;
    data = (hi | low); /* 12 bits */

    outb(0, IO_BASE+9);
    return data;
}

/* Function to send data to D/A */
int io_dacout(volatile cloop_t *cloop, int chan, int data)
{
    unsigned int low = (data << 4) & 0x00F0;
    unsigned int hi = (data >> 4) & 0x00FF;

    if(data < 0 || 4095 < data) {
        #ifdef _RTL_
        rtl_printf("ax5411: dac out value (%d) out of range\n", data);
        #else
        printf("ax5411: dac out value (%d) out of range\n", data);
        #endif
        return -1;
    }

    switch(chan) {
        case 0:
            outb(low, IO_BASE+4);
            outb(hi, IO_BASE+5);
            break;

        case 1:
            outb(low, IO_BASE+6);
            outb(hi, IO_BASE+7);
            break;

        default:
            return -1;
            break;
    }

    return 0;
}
APPENDIX D. C-PROGRAM: CLOOP.C

/* Scale a 12 bit int between low and hi */
float scale_itof(int data, int ilow, int ihi, float low, float hi)
{
    return (((float)data - ilow) / (ihi - ilow)) * (hi - low) + low;
}

/* Asking function for saturation range values */
float lookup_sat(int i)
{
    float a=0;
    if(i == 1)
        a=2.10; // lowest value channel1
    if(i == 2)
        a=3.03; // highest value channel1
    if(i == 3)
        a=2.56; // init value channel1
    if(i == 4)
        a=2.45; // lowest value channel0 (max throttle)
    if(i == 5)
        a=2.80; // highest/init value channel0

    return a;
}

/* Scale a float between low and hi to a 12bit int */
int scale_ftoi(float data, float low, float hi, int ilow, int ihi)
{
    float temp;
    float il, ih;
    il = (float) ilow; ih = (float) ihi;
    temp = ((data - low) / (hi - low)) * (ih - il) + il;
    return (int)temp;
}

/* Filter data using coefficient a */
float filter_data(float ydata, float udata, float a)
{
    return ((1-a)*udata + a*ydata);
}

/* Function to implement saturation */
float sat(float x, float xlo, float xhi)
{
    return (x < xlo) ? xlo : ((x > xhi) ? xhi : x);
}
/* Setup counter on ax5411 card */
void set_io_counters(unsigned int n1, unsigned int n2)
{
/* load counter 2 */
outb(0xb4, IO_BASE+15);
outb((n2 & 0x00ff), IO_BASE+14);
outb((n2 >> 8), IO_BASE+14);
/* load counter 1 */
outb(0x74, IO_BASE+15);
outb((n1 & 0x00ff), IO_BASE+13);
outb((n1 >> 8), IO_BASE+13);
}

/* Setup the IO (ax5411) card */
int setup_io(volatile cloop_t *cloop)
{
int ad_chan;
unsigned int n1=0, n2=0;
unsigned int temp, temp1=1;
/* request access to the i/o region */
#ifdef __KERNEL__
release_region(IO_BASE, 16);
request_region(IO_BASE, 16, "AX5411");
#endif
#ifndef __KERNEL__
ioperm(IO_BASE,16,1);
#endif
/* reset control and status */
outb(0, IO_BASE+9);
outb(0, IO_BASE+8);

    temp = (unsigned int) (1000000.0 * cloop->dt);
    while (temp > temp1) {
        n1 = temp; n2 = temp1;
        temp = (unsigned int) (temp / 5);
        temp1 = temp1 * 5;
    }
    cloop->dt = n1*n2 / 1000000.0;

set_io_counters(n1, n2);
outb(0x00d3, IO_BASE+9);

ad_chan = (cloop->adc_chan << 4) & 0x00f0;
APPENDIX D. C-PROGRAM: CLOOP.C

/* set mux control for channel */
outb(ad_chan, IO_BASE+2); // conversion for channel 0-chan
return 0;
}

/* Insert a new node into the circular buffer */
void insert_new(volatile cloop_t *cloop)
{
    int old;
    old = clcop->time_index;
    clcop->time_index = (clcop->time_index + 1) % MAX_DATA;

    if (clcop->time_index == clcop->last) {
        clcop->num_data--;
        clcop->last = (clcop->last + 1) % MAX_DATA;
    }
    clcop->cloop_sig[clcop->time_index].time = clcop->cloop_sig[old].time;
    clcop->cloop_sig[clcop->time_index].y = clcop->cloop_sig[old].y;
    clcop->cloop_sig[clcop->time_index].u0 = clcop->cloop_sig[old].u0;
    clcop->cloop_sig[clcop->time_index].ui = clcop->cloop_sig[old].ui;
    clcop->cloop_sig[clcop->time_index].r = clcop->cloop_sig[old].r;
    clcop->num_data++;
}

/* Write a new signal into the newest node of the circular buffer */
void store_data(volatile cloop_t *cloop, sig_t new_sig)
{
    sig_copy(&new_sig, &(cloop->cloop_sig[clcop->time_index]));
}

/* Read off the oldest 'num' signal nodes from the end of the buffer */
int read_data(volatile cloop_t *cloop, sig_t *data_store, int num)
{
    int i;

    if (num > clcop->num_data) {
        #ifdef __RTL__
        rtl_printf("Not enough data\n");
        #endif
        return 0;
    }
for (i = 0; i < num; i++) {
    sig_copy(&(cloop->cloop_sig[cloop->last][i]), &data_store[i]);
    cloop->last = (cloop->last + i) % MAX_DATA;
    cloop->num_data--;
}

return num;
}

/* Pull off the signal with delay 'delay' */
sig_t get(volatile cloop_t *cloop, int delay)
{
    int index;
    sig_t temp;

    sig_init(&temp);
    if (cloop->num_data == 0) {
#ifdef __RTL__
        rtl_printf("Not enough data\n");
#endif
        return temp;
    }

    index = cloop->time_index - delay;
    if (index < 0) index += MAX_DATA;
    sig_copy(&(cloop->cloop_sig[index]), &temp);
    return temp;
}

/* Display 'num' past signal values */
void display(volatile cloop_t *cloop, int num)
{
    int i;
    sig_t temp;

    for (i=0; i<num; i++) {
        temp = get(cloop,num-i-1);
#ifdef __KERNEL__
        printf("y : %d \t y1 : %d \t u : %d \t r : %d \t time : %d \n",temp.y,
                temp.u0,temp.ui,temp.r,temp.time);
#endif
#ifdef __RTL__
        rtl_printf("y : %d \t y1 : %d \t u : %d \t r : %d \t time : %d \n",(int)(100*temp.y),
                    (int)(100*temp.u0),(int)(100*temp.ui),(int)(100*temp.r),(int)temp.time);
#endif
    }
}
/* function to log data to a file 'fname' */
int store_log(volatile cloop_t *cloop, char *fname)
{
    #ifndef __KERNEL__
    int i, start, end;
    FILE *fd_open;
    int fd_write;
    sig_t temp;
    float u0,u1,y,r;
    float time;
    float oldu,oldy;

    start = cloop->datastart;
    if (cloop->dataend < cloop->datastart) end = cloop->num_data-1;
    else end = cloop->dataend;

    fd_open = fopen(fname, "w");
    if (fd_open == NULL) {
        printf("Error opening file \
");
        return -1;
    }

    for ( i=start ; i<=end ; i++) {
        temp = get(cloop,end-i);
        time = temp.time; u0 = temp.u0; y = temp.y; u1 = temp.u1; r = temp.r;
        fprintf(fd_open, "%4.2f %4.3f %4.3f %4.3f %4.3f
",time,y,u0,u1);
    }

    fclose(fd_open);
    #endif
    return 0;
}
Appendix E

c-program: cloop.h

#define MAX_DATA 1000 // Maximum data points
#define DEF_INTERVAL 0.1
#define CAL_PERIOD 3
#define IO_BASE 0x320

typedef float (fp)(float *, float, float);

/* Signal structure */
typedef struct sig {
    float y;
    float u0;
    float ui;
    float r;
    float time;
} sig_t;

/* Control Loop Structure */
typedef struct {
    sig_t cloop_sig[MAX_DATA]; // control loop signals
    int time_index, last; // array index and oldest data
    int num_data; // number of valid data
    int sig_low, sig_hi; // signal limits
    float dt; // sampling interval
    float param[20]; // control gains, 0-P, 1-I, 2-D, 3-N, 4-low, 5-hi
    int num_params; // number of control gains
    float x[20]; // state variables
    float setpt0, setpt1; // step value and ramp increment for setpoint
    int ramp_increment; // counter for ramp input
    int rampcount; // 0-step, 1-ramp
    int input;
    int adc_chan; // A/D channel
    int inchannel;
} cloop_s;
int dac_chan;  // D/A channel
int filter;   // low pass filter coefficient
int initialised; // 0-uninitialised, 1-initialised
int simulation; // 0-process, 1-simulation
int cmode; // 0-Manual, 1-PID, 2-OTHER, 3-OTHER
int cstate; // 0-stop, 1-start
int idle_tag;
int datalog; // 0-not data logging, 1-data logging
int datastart; // time to start data logging
int dataend; // time to end data logging
}
cloop_t;

/* signal initialisation function */
void sig_init(volatile sig_t *signal);

/* copy signal */
void sig_copy(volatile sig_t *source, volatile sig_t *dest);

/* Functions for the cloop structure */

/* Initialise the cloop structure */
void cloop_init(volatile cloop_t *cloop);

/* Function to clear the cloop */
void cloop_clear(volatile cloop_t *cloop);

/* Function to reset signal nodes */
void cloop_reset(volatile cloop_t *cloop);

/* Function to sample/scale and store data */
void sample(volatile cloop_t *cloop);

/* Calculate the control */
void control(volatile cloop_t *cloop);

/* D/A function - send control to D/A */
void dtoa(volatile cloop_t *cloop);

/* Set the controller gains */
void set_params(volatile cloop_t *cloop, float *params, int num_params);

/* Function to sample from the A/D */
int io_adcin(volatile cloop_t *cloop, int channel);

/* Function to send data to D/A */
```c
int io_dacout(volatile cloop_t *cloop, int chan, int data);

/* Scale a 12 bit int between low and hi */
float scale_itof(int data, int ilow, int ihi, float low, float hi);

/* Asking function for saturation range values */
float lookup_sat(int i);

/* Scale a float between low and hi to a 12bit int */
int scale_ftoi(float data, float low, float hi, int ilow, int ihi);

/* Filter data using a coefficient of a */
float filter_data(float ydata, float udata, float a);

/* Function to store data in 'fname' */
int store_log(volatile cloop_t *cloop, char *fname);

/* Setup counters on ax5411 card */
void set_io_counters(unsigned int nl, unsigned int n2);

/* Setup the I0 (ax5411 card */
int setup_io (volatile cloop_t *cloop);

/* Insert a new node in the circular buffer */
void insert_new(volatile cloop_t *cloop);

/* Write a signal into the newest node of the buffer */
void store_data(volatile cloop_t *cloop, sig_t new_sig);

/* Read of the last 'num' signal nodes from the buffer */
int read_data(volatile cloop_t *cloop, sig_t *data_store, int nun);

/* Pull of the signal with delay 'delay' */
sig_t get(volatile cloop_t *cloop, int delay);

/* Display signal nodes */
void display(volatile cloop_t *cloop, int nun);

/* Function to calculate manual control */
float manual(volatile cloop_t *cloop);

/* Function to calculate pid control */
float pid(volatile cloop_t *cloop);

/* state equation 1 for pid control */
float pidf1(float *params, float x, float u);

/* state equation 4 for pid control */
float pidf4(float *params, float x, float u);
```
APPENDIX E. C-PROGRAM: CLOOP.H

/* Function to implement saturation */
float sat(float x, float xlo, float xhi);

/* 3rd Runge-Kutta integration */
float rk3(volatile cloop_t *cloop, fp f, float x, float u, float dt);
Appendix F

c-program: rtmodule.c

```c
#include <rtl.h>
#include <pthread.h>
#include "mbuff.h"
#include "cloop.h"
#include "cdsm.h"

void *sample_code(void *arg);
void *control_code(void *arg);
void *dtoa_code(void *arg);

pthread_t sample_thread;
pthread_t control_thread;
pthread_t dtoa_thread;

volatile cloop_t *cloop;

int init_module(void)
{
    int module_status=0;
    float a,b;

    /* initialise shared memory */
cloop = (volatile cloop_t*) mbuf_alloc("control_loop",sizeof(cloop_t));
    if (cloop == NULL) {
```
rtl_printf("Shared Memory Allocation Problem : Returning\n");
return -1;
}

CDSM_init();

if (1) {
cloop_init(cloop);
cloop->simulation = 0;
}
a=lookup_sat(5);
b=lookup_sat(3);
io_dacout(cloop,0,a/5*4095);
io_dacout(cloop,1,b/5*4095);

module_status = pthread_create(&sample_thread,NULL,sample_code,0);
if (module_status != 0) {
rtl_printf("Thread initialisation failed: sample status %d\n",module_status);
return module_status;
}

module_status = pthread_create(&control_thread,NULL,control_code,0);
if (module_status != 0) {
rtl_printf("Thread initialisation failed: control status %d\n",module_status);
return module_status;
}

module_status = pthread_create(&dtoa_thread,NULL,dtoa_code,0);
if (module_status != 0) {
rtl_printf("Thread initialisation failed: dtoa status %d\n",module_status);
return module_status;
}
return 0;
}

/* module destroy */
void cleanup_module(void)
{
 pthread_delete_np(sample_thread);
pthread_delete_np(control_thread);
pthread_delete_np(dtoa_thread);
mbuff_free("control_loop", (void *)cloop);
CDSM_done();
}

/* sampling thread code */
void *sample_code(void *arg)
struct sched_param p;
hrtime_t now;
long interval;

now = gethrtime();
p.sched_priority = 1;
pthread_setschedparam(pthread_self(),SCHED_FIFO,&p);
interval = (long)(cloop->dt * 1000000000.0);
while(1) {
    pthread_make_periodic_np(pthread_self(),now,interval);
    if (cloop->cstate) sample(cloop);
    //rtl_printf("debug message : %d %d\n",cloop->cstate,cloop->calibrate_count);
    pthread_wakeup_np(control_thread);
}
return 0;
*/
void *control_code(void *arg)
{
    struct sched_param p;
    p.sched_priority = 1;
    pthread_setschedparam(pthread_self(),SCHED_FIFO,&p);
    while(1) {
        pthread_suspend_np(pthread_self());
        // run 'control' if not in stop mode
        if (cloop->cstate) control(cloop);
        //rtl_printf("debug message : control thread\n");
        pthread_wakeup_np(dtoa_thread);
    }
    return 0;
}
/* dtoa thread code */

void *dtoa_code(void *arg)
{
  struct sched_param p;

  p.sched_priority = 1;
pthread_setschedparam(pthread_self(),SCHED_FIFO,&p);
pthread_setfp_np(pthread_self(),1);

  while(1) {
    pthread_suspend_np(pthread_self());
    if (cloop->cstate) dtoa(cloop);

    //rtl_printf("debug message : dtoa thread\n");
  }

  return 0;
}
Appendix G

c-program: thread_code.c

/*****************************/
/* thread_code.c */
/* contains functions to sample, control */
/* and drive D/A converter */
/*****************************/

#include <rtl_printf.h>
#include "cloop.h"
#include "cdsm.h"

/* Function to sample/scale and store data */
/* Called directly from 'sample' thread */
void sample(volatile cloop_t *cloop)
{
  int data=0;
  float sdata, fdata=0.0, fdatal=0.0;
  static float yold, yfold;
  float setpt=0.0;
  sig_t new_signal;

  /* get copy of old data */
  new_signal = get(cloop,0);

  /* create space for new data */
  insert_new(cloop);

  if (cloop->simulation) {
    fdata = 0.95*new_signal.y + 0.05*new_signal.u0;
    //fdata = yold + 0.35*fdata;
  }
else {
    data = io_adc(cloop, 0);
    sdata = scale_itof(data, 0, 4096, -10.0, 10.0);
    fdata = filter_data(yold, sdata, cloop->filter);

    data = io_adc(cloop, 1);
    sdata = scale_itof(data, 0, 4096, -10.0, 10.0);
    fdata1 = filter_data(yold, sdata, cloop->filter);
}

// update old values
yold = fdata; yiold = fdata1;

/*@ update setpoint */
switch (cloop->input) {
    case 0: setpt = cloop->setpt0;
            break;
    case 1: setpt = new_signal.r + cloop->ramp_increment;
            if (setpt >= 1.0) {
                setpt = -1.0;
                cloop->rampcount++;
            }
            if (cloop->rampcount == 5) setpt = 0;
            break;
    default: break;
}

/*@ update new signal */
new_signal.y = fdata;
new_signal.r = setpt;
new_signal.time += cloop->dt;

/*@ store in CDSM */
CDSM_set(0, (long)(100*fdata));
CDSM_set(1, (long)(100*setpt));

/*@ store the value in y buffer */
store_data(cloop, new_signal);
}

/*@ Calculate the control */
/*@ Called directly from 'control' thread */
void control(volatile cloop_t *cloop) {
    float cont0 = 0.0, cont1 = 0.0;
APPENDIX G. C-PROGRAM: THREAD_CODE.C

float a, b, c, d;

// define some temporary static variables to use for memory
//static float x[10];

// Declare a 'signal' node. Each node has setpoint (r), output (y),
// input (u), and time (time) elements. To retrieve a node, you use
// the 'get' function. That is, 'get(cloop, delay)' where delay is
// the signal delay in discrete-time. Example: to retrieve the control
// data from 2 samples back use 'get(cloop, 2).u'

sig_t curr_signal;

we are in run mode

// control mode: 0 - manual control
// control mode: 1 - PID control
// can simply add in another case statement and define another control scheme
// the control action is returned to the variable 'cont'

switch (cloop->cmode) {
    case 0:
        cont0 = clloop->setpt0;
        cont1 = clloop->setpt1;
        break;
    case 1:
        cont0 = pid(cloop);
        break;
    default: break;
}

#ifdef _RTL_
rtl_printf("mode = %d \d\n", clloop->cmode, (int)(100*cont0));
#endif

/* apply signal limits */
a = lookup_sat(1);
b = lookup_sat(2);
c = lookup_sat(4);
d = lookup_sat(5);
cont0 = sat(cont0, c, d);
cont1 = sat(cont1, a, b);

curr_signal = get(cloop, 0);
curr_signal.u0 = cont0;
curr_signal.u1 = cont1;

CDSM_set(2, (long)(100*cont0));
CDSM_set(3, (long)(100*cont1));
store_data(cloop, curr_signal);
}
/* Function to send to dtoa */
/* Called directly by 'dtoa' thread */
void dtoa(volatile cloop_t *cloop)
{
    float ut0, ut1;
    int uint0, uint1, stat;

    ut0 = (get(cloop, 0)).u0;
    uint0 = scale_ftoi(ut0, 0.0, 5.0, 0, 4096);
    stat = io_dacout(cloop, 0, uint0);
    ut1 = (get(cloop, 0)).u1;
    uint1 = scale_ftoi(ut1, 0.0, 5.0, 0, 4096);
    stat = io_dacout(cloop, 1, uint1);
}

/* Function to calculate pid control */
float pid(volatile cloop_t *cloop)
{
    float kp, ti, td, N;
    sig_t curr_signal;

    N = cl00p->param[3];
    kp = cl00p->param[0];
    ti = cl00p->param[1];
    td = cl00p->param[2];
    curr_signal = get(cloop, 0);

    if (td > 0.0) {
        cl00p->x[4] = rk3(cl00p, pidf4, cl00p->x[4], curr_signal.y, cl00p->dt);
        cl00p->x[3] = curr_signal.y + N * (curr_signal.y - cl00p->x[4]);
    } else {
        cl00p->x[3] = curr_signal.y;
    }

    cl00p->x[2] = kp * (curr_signal.r - cl00p->x[3]);
    if (ti >= 20) cl00p->x[1] = 0;
    else {
        cl00p->x[1] = rk3(cl00p, pidf1, cl00p->x[1], cl00p->x[2], cl00p->dt);
    }

    return (cl00p->x[1] + cl00p->x[2]);
}
/* state equation 1 for pid control */
float pidf1(float *params, float x, float u)
{
    float temp, ti;
    ti = params[1];
    temp = ((sat(x+u,params[4],params[5]) - x) / ti);
    return temp;
}

/* state equation 2 for pid control */
float pidf4(float *params, float x, float u)
{
    float td, N;
    td = params[2]; N = params[3];
    return (10 * (u - x));
}
Appendix H

c-program: hovercraft.c

#include <stdio.h>
#include <unistd.h>
#include <gtk/gtk.h>
#include <gtk/gtkhscale.h>
#include <gtk/gtkvscale.h>
#include "cloop.h"
#include "hovercraft.h"
#include "mbuff.h"

/* shared memory variable */
volatile cloupt_t *cloop;
volatile unsigned char *cch;
volatile int *flag;

/* gui object */
guiobj cgui;

/* This callback quits the program */
gint delete_event(GtkWidget *widget, GdkEvent *event, gpointer data)
{
    gtk_main_quit();
    return(FALSE);
}

/* function to initialise gui */
int gui_init(void)
{
    /* initialise gtk */
    gtk_init(0,0);

    /* init shared memory */
cloop = (volatile cloop_t*) mbuff_alloc("control_loop",sizeof(cloop_t));
if (cloop == NULL) {
    printf("Shared Memory Allocation Failed!\n");
    return -1;
}
ch = (volatile unsigned char *) mbuff_alloc("char",sizeof(char));
flag = (volatile int *) mbuff_alloc("flag",sizeof(int));
*flag = 0;
/* test to see if cloop is initialised */
if (!cloop->initialised) {
    printf("Control Loop Not Initialised \n");
    return -1;
}
cloop->cstate = 0;
cloop->cmode = 0;

return 0;

/* callback for start button */
void start_function( GtkWidget *widget, int *arg)
{
    /* start the control loop */
cloop->cstate = 1;
}

/* callback for stop button */
void stop_function( GtkWidget *widget, GtkLabel *label)
{
    float a,b;
    char s0[14],s1[14];
a=lockup_sat(5);
b=lockup_sat(3);

    /* stop the control loop */
cloop->cstate = 0;
io_dacout(cloop,0,a/5*4095);
io_dacout(cloop,1,b/5*4095);
cloop->setpt0=a;
cloop->setpt1=b;
sprintf(s0,"UP-DOWN : %3.2f",a);
gtk_label_set_text((GtkLabel *) (cgui.label[3]),s0);
sprintf(s1,"LEFT-RIGHT : %3.2f",b);
gtk_label_set_text((GtkLabel *) (cgui.label[4]),s1);
APPENDIX H. C-PROGRAM: HOVERCRAFT.C

store_log(cloop, "hcraft");
cloop_clear(cloop);
gtk_label_set_text(label, "Logging OFF");
}

/* callback for up button */
void up_button(GtkWidget *widget, int *arg)
{
  float setpt0 = cloop->setpt0;
  char s[14];
  float a, b;
  a = lookup_sat(4);
  b = lookup_sat(5);
  setpt0 -= 0.01;
  setpt0 = sat(setpt0, a, b);
  sprintf(s, "UP-DOWN : %3.2f\n", setpt0);
  gtk_label_set_text((GtkWidget *)(cgui.label[3]), s);
  cloop->setpt0 = setpt0;
  printf("u0 = %3.2f\n", setpt0);
}

/* callback for down button */
void down_button(GtkWidget *widget, int *arg)
{
  float setpt0 = cloop->setpt0;
  char s[14];
  float a, b;
  a = lookup_sat(4);
  b = lookup_sat(5);
  setpt0 += 0.01;
  setpt0 = sat(setpt0, a, b);
  sprintf(s, "UP-DOWN : %3.2f\n", setpt0);
  gtk_label_set_text((GtkWidget *)(cgui.label[3]), s);
  cloop->setpt0 = setpt0;
  printf("u0 = %3.2f\n", setpt0);
}

/* callback for left button */
void left_button(GtkWidget *widget, int *arg)
{
  float setpt1 = cloop->setpt1;
  char s[14];
APPENDIX H. C-PROGRAM: HOVERCRAFT.C

```c
float a, b;

a = lookup_sat(1);
b = lookup_sat(2);
setpt1 -= 0.01;
setpt1 = sat(setpt1, a, b);
sprintf(s, "LEFT-RIGHT: %3.2f", setpt1);
gtk_label_set_text((GtkWidget *)(cgui.label[4]), s);
cloop->setpt1 = setpt1;
printf("ul = %3.2f\n", setpt1);
}

/* callback for right button */
void right_button( GtkWidget *widget, int *arg)
{
float setpt1 = cloop->setptl;
char s[14];
float a, b;

a = lookup_sat(1);
b = lookup_sat(2);
setpt1 -= 0.01;
setpt1 = sat(setpt1, a, b);
sprintf(s, "LEFT-RIGHT: %3.2f", setpt1);
gtk_label_set_text((GtkWidget *)(cgui.label[4]), s);
cloop->setpt1 = setpt1;
printf("ul = %3.2f\n", setpt1);
}

/* callback for data log button */
void data_log_function( GtkWidget *widget, GtkWidget *label)
{
/* start datalog flag */
if (cloop->datalog) {
cloop->datalog = 0;
cloop->dataend = cloop->time_index;
}
else {
cloop->datalog = 1;
cloop->datastart = cloop->time_index;
}
if (cloop->datalog) gtk_label_set_text(label, "Logging ON");
else gtk_label_set_text(label, "Logging OFF");
```
APPENDIX H. C-PROGRAM: HOVERCRAFT.C

/* callback for quit button */
void quit_function(GtkWidget *widget, int *arg)
{
    float a,b;
    a=lookup_sat(5);
    b=lookup_sat(3);

    /* stop the control loop */
    cloop->cstate = 0;
    io_dacout(cloop,0,a/5*4096);
    io_dacout(cloop,1,b/5*4096);
    gtk_idle_remove( clloop->idle_tag );
    store_log(cloop,"hcraft");
    clloop_clear(cloop);
    gtk_main_quit();
}

/* idle function */
int hover_idle(GtkLabel *label)
{
    /* sig_t curr;
    char s0[4],s1[4];
    GtkWidget *label0, *label1;
    
    if (*flag) {
        label0 = (GtkWidget *) (cgui.label[3]);
        label1 = (GtkWidget *) (cgui.label[4]);
        curr = get(cloop,0);
        if (*cch == 'i') curr.u0 += 0.1;
        if (*cch == 'm') curr.u0 -= 0.1;
        if (*cch == 'j') curr.u1 -= 0.1;
        if (*cch == 'l') curr.u1 += 0.1;
        curr.u0 = sat(curr.u0,0.0,5.0);
        curr.u1 = sat(curr.u1,0.0,5.0);
        sprintf(s0,"%3.2fl1,",curr.u0);
        gtk_label_set_text(label0,s0);
        sprintf(s1,"%3.2fN,",curr.u1);
        gtk_label_set_text(label1,s1);
        printf("Char = %c \t %f\t %f\n",*cch,curr.u0,curr.u1);
    }
    store_data(cloop, curr);
    *flag = 0;
    */
}
APPENDIX H. C-PROGRAM: HOVERCRAFT.C

return TRUE;
}

/* gui for lab */
void gui_lab(GtkWidget *window)
{
    /* remove initial window */
    /* Create the new window */
    cgui.window[0] = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_widget_set_usize(cgui.window[0], 500, 300);
    gtk_widget_set_uposition(cgui.window[0], 0, 0);

    /* Set the window title */
    gtk_window_set_title(GTK_WINDOW(cgui.window[0]), "HOVERCRAFT");

    /* Set a handler for delete-event that immediately exits GTK. */
    gtk_signal_connect(GTK_OBJECT(cgui.window[0]), "delete-event",
        GTK_SIGNAL_FUNC (delete-event), NULL);

    /* Sets the border width of the window. */
    gtk_container_set_border_width(GTK_CONTAINER(cgui.window[0]), 20);

    /* Create a 5x1 table */
    cgui.table[0] = gtk_table_new(6, 7, TRUE);
    gtk_table_set_row_spacings(GTK_TABLE(cgui.table[0]), 0);
    gtk_table_set_col_spacings(GTK_TABLE(cgui.table[0]), 25);

    /* Put the table in the main window */
    gtk_container_add(GTK_CONTAINER(cgui.window[0]), cgui.table[0]);

    /* create controller label */
    cgui.label[0] = gtk_label_new("HOVERCRAFT CONTROL");
    gtk_label_set_justify(GTK_LABEL(cgui.label[0]), GTK_JUSTIFY_LEFT),
    gtk_table_attach_defaults(GTK_TABLE(cgui.table[0]), cgui.label[0], 1, 5, 0, 1);
    gtk_widget_show(cgui.label[0]);

    /* include seperator */
    cgui.separator = gtk_hseparator_new();
    gtk_table_attach_defaults(GTK_TABLE(cgui.table[0]), cgui.separator, 0, 6, 1, 2);
    gtk_widget_show(cgui.separator);

    /* put in buttons at bottom - up button */
    cgui.button = gtk_button_new_with_label("UP");
    gtk_signal_connect(GTK_OBJECT(cgui.button), "clicked",
        GTK_SIGNAL_FUNC (up_button), NULL);
    gtk_table_attach_defaults(GTK_TABLE(cgui.table[0]), cgui.button, 2, 3, 2, 3);
APPENDIX H. C-PROGRAM: HOVERCRAFT.C

gtk_widget_show (cgui.button);

/* put in buttons at bottom - down button */
cgui.button = gtk_button_new_with_label ("DOWN");
gtk_signal_connect(GTK_OBJECT (cgui.button), "clicked",
GTK_SIGNAL_FUNC (down_button), NULL);
gtk_table_attach_defaults (GTK_TABLE(cgui.table[0]), cgui.button, 2, 3, 4, 5);
gtk_widget_show (cgui.button);

/* put in buttons at bottom - left button */
cgui.button = gtk_button_new_with_label ("LEFT");
gtk_signal_connect(GTK_OBJECT (cgui.button), "clicked",
GTK_SIGNAL_FUNC (left_button), NULL);
gtk_table_attach_defaults (GTK_TABLE(cgui.table[0]), cgui.button, 1, 2, 3, 4);
gtk_widget_show (cgui.button);

/* put in buttons at bottom - right button */
cgui.button = gtk_button_new_with_label ("RIGHT");
gtk_signal_connect(GTK_OBJECT (cgui.button), "clicked",
GTK_SIGNAL_FUNC (right_button), NULL);
gtk_table_attach_defaults (GTK_TABLE(cgui.table[0]), cgui.button, 3, 4, 3, 4);
gtk_widget_show (cgui.button);

/* create up-down label */
cgui.label[3] = gtk_label_new("UP-DOWN : 2.80");
gtk_label_set_justify(GTK_LABEL(cgui.label[3]), GTK_JUSTIFY_LEFT);
gtk_table_attach_defaults (GTK_TABLE(cgui.table[0]), cgui.label[3], 4, 6, 2, 3);
gtk_widget_show (cgui.label[3]);

/* create left-right label */
cgui.label[4] = gtk_label_new("LEFT-RIGHT : 2.56");
gtk_label_set_justify(GTK_LABEL(cgui.label[4]), GTK_JUSTIFY_LEFT);
gtk_table_attach_defaults (GTK_TABLE(cgui.table[0]), cgui.label[4], 4, 6, 4, 5);
gtk_widget_show (cgui.label[4]);

/* include seperator */
cgui.separator = gtk_hseparator_new ();
gtk_table_attach_defaults (GTK_TABLE(cgui.table[0]), cgui.separator, 0, 6, 5, 6);
gtk_widget_show(cgui.separator);

/* put in buttons at bottom - start button */
cgui.button = gtk_button_new_with_label ("Start");
gtk_signal_connect(GTK_OBJECT (cgui.button), "clicked",
GTK_SIGNAL_FUNC (start_function), NULL);
gtk_table_attach_defaults (GTK_TABLE(cgui.table[0]), cgui.button, 0, 7, 6, 7);
gtk_widget_show (cgui.button);
APPENDIX H. C-PROGRAM: HOVERCRAFT.C

/* create logging status label */
cgui.label[5] = gtk_label_new("Logging OFF");
gtk_label_set_justify(GTK_LABEL(cgui.label[5]), GTK_JUSTIFY_LEFT);
gtk_table_attach_defaults (GTK_TABLE(cgui.table[0]), cgui.label[5], 4, 6, 6, 7);
gtk_widget_show (cgui.label[5]);

/* put in buttons at bottom - stop button */
cgui.button = gtk_button_new_with_label("Stop");
gtk_signal_connect(GTK_OBJECT(cgui.button), "clicked",
GTK_SIGNAL_FUNC (stop_function), GTK_LABEL(cgui.label[5]));
gtk_table_attach_defaults (GTK_TABLE(cgui.table[0]), cgui.button, 1, 2, 6, 7);
gtk_widget_show (cgui.button);

/* put in buttons at bottom - log button */
cgui.button = gtk_button_new_with_label("Log");
gtk_signal_connect(GTK_OBJECT(cgui.button), "clicked",
GTK_SIGNAL_FUNC (data_log_function), GTK_LABEL(cgui.label[5]));
gtk_table_attach_defaults (GTK_TABLE(cgui.table[0]), cgui.button, 3, 4, 6, 7);
gtk_widget_show (cgui.button);

/* put in buttons at bottom - quit button */
cgui.button = gtk_button_new_with_label("Quit");
gtk_signal_connect(GTK_OBJECT(cgui.button), "clicked",
GTK_SIGNAL_FUNC (delete_event), NULL);
gtk_table_attach_defaults (GTK_TABLE(cgui.table[0]), cgui.button, 2, 3, 6, 7);
gtk_widget_show (cgui.button);

/* show window and table*/
gtk_widget_show(cgui.table[0]);
gtk_widget_show(cgui.window[0]);

gtk_main();
}

/* main gui function - run from main */
void gui_main(guiobj *gui)
{
    gui_lab(gui->window[0]);
    /* main gtk function */
    /*gtk_main();*/
}

int main( int argc, char *argv[] )
{
    float a,b;
    /* initialisation */
    if (gui_init() == -1) {
        printf("Initialisation Problem - EXITING \n");
        return -1;
    }

    /* reset card */
    a=lookup_sat(6);
    b=lookup_sat(3);
    ioperm(IO_BASE,16,1);
    io_dacout(cloop,0,a/5*4095);
    io_dacout(cloop,1,b/5*4095);

    /* Create main gui */
    gui_main(&cgui);

    io_dacout(cloop,0,a/5*4095);
    io_dacout(cloop,1,b/5*4095);
    /* free shared memory */
    mbuff_free("control_loop", (void *)cloop);

    return 0;
}
/* gui end */
Appendix I

c-program: hovercraft.h

// type definition of our gui

typedef struct gtk_gui
{
  GtkWidget *window[3];
  GtkWidget *table[2];
  GtkWidget *label[6];
  GtkWidget *button;
  GtkWidget *radio[8];
  GSList *group[3];
  GtkWidget *adjust[10];
  GtkWidget *separator;
  GtkWidget *spin[4];
  GtkWidget *textbox;
  GtkWidget *hbox;
  GtkWidget *vscrollbar;
} guiobj;
Appendix J

c-program: Makefile

all: hovercraft control_mod.o

include rtl.mk

control_mod.o: cloop.o thread_code.o rtmodule.o cdsn.o
    ld -r -o control_mod.o rtmodule.o cloop.o thread_code.o cdsn.o

cloop_nrt.o: cloop.c
    gcc -c -O2 -o cloop_nrt.o cloop.c

hovercraft: hovercraft.c cloop_nrt.o
    gcc -O2 -c -o hovercraft.o hovercraft.c 'gtk-config --cflags'
    gcc -o hovercraft hovercraft.o cloop_nrt.o 'gtk-config --libs'

clean:
    rm -f *.o hovercraft
Appendix K

Linux Installation Manual

Linux kernel release 2.2.xx

These are the release notes for Linux version 2.2. Read them carefully, as they tell you what this is all about, explain how to install the kernel, and what to do if something goes wrong.

However, please make sure you don't ask questions which are already answered in various files in the Documentation directory. See DOCUMENTATION below.

WHAT IS LINUX?

Linux is a Unix clone written from scratch by Linus Torvalds with assistance from a loosely-knit team of hackers across the Net. It aims towards POSIX compliance.

It has all the features you would expect in a modern fully-fledged Unix, including true multitasking, virtual memory, shared libraries, demand loading, shared copy-on-write executables, proper memory management and TCP/IP networking.

It is distributed under the GNU General Public License - see the accompanying COPYING file for more details.

ON WHAT HARDWARE DOES IT RUN?

Linux was first developed for 386/486-based PCs. These days it also runs on ARM, DEC Alphas, SUN Sparcs, M68000 machines (like Atari and Amiga), MIPS and PowerPC, and others.

DOCUMENTATION:

- There is a lot of documentation available both in electronic form on the Internet and in books, both Linux-specific and pertaining to
general UNIX questions. I'd recommend looking into the documentation subdirectories on any Linux FTP site for the LDP (Linux Documentation Project) books. This README is not meant to be documentation on the system: there are much better sources available.

- There are various README files in the Documentation/ subdirectory; these typically contain kernel-specific installation notes for some drivers for example. See ./Documentation/GO-INDE for a list of what is contained in each file. Please read the Changes file, as it contains information about the problems, which may result by upgrading your kernel.

INSTALLING the kernel:

- If you install the full sources, do a

  cd /usr/src
  gzip -cd linux-2.2.XX.tar.gz | tar xfv -

  to get it all put in place. Replace "XX" with the version number of the latest kernel.

- You can also upgrade between 2.2.xx releases by patching. Patches are distributed in the traditional gzip and the new bzip2 format. To install by patching, get all the newer patch files and do

  cd /usr/src
  gzip -cd patchXX.gz | patch -p0

  or

  cd /usr/src
  bzip2 -dc patchXX.bz2 | patch -p0

  (repeat xx for all versions bigger than the version of your current source tree, _in_order_) and you should be ok. You may want to remove the backup files (xxx" or xxx.orig), and make sure that there are no failed patches (xxx# or xxx.rej). If there are, either you or me has made a mistake.

  Alternatively, the script patch-kernel can be used to automate this process. It determines the current kernel version and applies any patches found.

  cd /usr/src
  linux/scripts/patch-kernel

  The default directory for the kernel source is /usr/src/linux, but can be specified as the first argument. Patches are applied from the current directory, but an alternative directory can be specified
as the second argument.

- Make sure you have no stale .o files and dependencies lying around:

```bash
cd /usr/src/linux
make mrproper
```

You should now have the sources correctly installed.

SOFTWARE REQUIREMENTS

Compiling and running the 2.2.xx kernels requires up-to-date versions of various software packages. Consult ./Documentation/Changes for the minimum version numbers required and how to get updates for these packages. Beware that using excessively old versions of these packages can cause indirect errors that are very difficult to track down, so don't assume that you can just update packages when obvious problems arise during build or operation.

CONFIGURING the kernel:

- Do a "make config" to configure the basic kernel. "make config" needs bash to work: it will search for bash in $BASH, /bin/bash and /bin/sh (in that order), so one of those must be correct for it to work.

Do not skip this step even if you are only upgrading one minor version. New configuration options are added in each release, and odd problems will turn up if the configuration files are not set up as expected. If you want to carry your existing configuration to a new version with minimal work, use "make oldconfig", which will only ask you for the answers to new questions.

- Alternate configuration commands are:
  "make menuconfig" Text based color menus, radiolists & dialogs.
  "make xconfig" X windows based configuration tool.
  "make oldconfig" Default all questions based on the contents of your existing ./.config file.

NOTES on "make config":

- Having unnecessary drivers will make the kernel bigger, and can under some circumstances lead to problems: probing for a nonexistent controller card may confuse your other controllers
- Compiling the kernel with "Processor type" set higher than 386 will result in a kernel that does NOT work on a 386. The kernel will detect this on bootup, and give up.
- A kernel with math-emulation compiled in will still use the coprocessor if one is present: the math emulation will just never get used in that case. The kernel will be slightly larger,
but will work on different machines regardless of whether they have a math coprocessor or not.
- the "kernel hacking" configuration details usually result in a bigger or slower kernel (or both), and can even make the kernel less stable by configuring some routines to actively try to break bad code to find kernel problems (kmalloc()). Thus you should probably answer 'n' to the questions for "development", "experimental", or "debugging" features.

- Check the top Makefile for further site-dependent configuration (default SVGA mode etc).
- Finally, do a "make dep" to set up all the dependencies correctly.

COMPILING the kernel:

- Make sure you have gcc-2.7.2 or newer available. It seems older gcc versions can have problems compiling newer versions of Linux. This is mainly because the older compilers can only generate "a.out"-format executables. As of Linux 2.1.0, the kernel must be compiled as an "ELF" binary. If you upgrade your compiler, remember to get the new binutils package too (for as/ld/nm and company).

  Please note that you can still run a.out user programs with this kernel.

- Do a "make zImage" to create a compressed kernel image. If you want to make a boot disk (without root filesystem or LILO), insert a floppy in your A: drive, and do a "make zdisk". It is also possible to do "make zlilo" if you have lilo installed to suit the kernel makefiles, but you may want to check your particular lilo setup first.

- If your kernel is too large for "make zImage", use "makebzImage" instead.

- If you configured any of the parts of the kernel as 'modules', you will have to do "make modules" followed by "make modules_install". Read Documentation/modules.txt for more information. For example, an explanation of how to use the modules is included there.

- Keep a backup kernel handy in case something goes wrong. This is especially true for the development releases, since each new release contains new code which has not been debugged. Make sure you keep a backup of the modules corresponding to that kernel, as well. If you are installing a new kernel with the same version number as your working kernel, make a backup of your modules directory before you do a "make modules_install".

- In order to boot your new kernel, you'll need to copy the kernel
image (found in /usr/src/linux/arch/i386/boot/zImage after compilation) to the place where your regular bootable kernel is found.

For some, this is on a floppy disk, in which case you can "cp /usr/src/linux/arch/i386/boot/zImage /dev/fd0" to make a bootable floppy. Please note that you cannot boot a kernel by directly dumping it to a 720k double-density 3.5" floppy. In this case, it is highly recommended that you install LILO on your double-density boot floppy or switch to high-density floppies.

If you boot Linux from the hard drive, chances are you use LILO which uses the kernel image as specified in the file /etc/lilo.conf. The kernel image file is usually /vmlinuz, or /zImage, or /etc/zImage. To use the new kernel, save a copy of the old image and copy the new image over the old one. Then, you MUST RELOAD LILO to update the loading map!! If you don't, you won't be able to boot the new kernel image.

Reinstalling LILO is usually a matter of running /sbin/lilo. You may wish to edit /etc/lilo.conf to specify an entry for your old kernel image (say, /vmlinuz.old) in case the new one does not work. See the LILO docs for more information.

After reinstalling LILO, you should be all set. Shutdown the system, reboot, and enjoy!

If you ever need to change the default root device, video mode, ramdisk size, etc. in the kernel image, use the 'rdev' program (or alternatively the LILO boot options when appropriate). No need to recompile the kernel to change these parameters.

- Reboot with the new kernel and enjoy.

IF SOMETHING GOES WRONG:

- If you have problems that seem to be due to kernel bugs, please check the file MAINTAINERS to see if there is a particular person associated with the part of the kernel that you are having trouble with. If there isn't anyone listed there, then the second best thing is to mail them to me (torvalds@transmeta.com), and possibly to any other relevant mailing-list or to the newsgroup. The mailing-lists are useful especially for SCSI and networking problems, as I can't test either of those personally anyway.

- In all bug-reports, *please* tell what kernel you are talking about, how to duplicate the problem, and what your setup is (use your common sense). If the problem is new, tell me so, and if the problem is old, please try to tell me when you first noticed it.
- If the bug results in a message like

unable to handle kernel paging request at address C0000010
Oops: 0002
EIP: 0010:XXXXXXXX
eax: xxxxxxxx ebx: xxxxxxxx ecx: xxxxxxxx edx: xxxxxxxx
esi: xxxxxxxx edi: xxxxxxxx ebp: xxxxxxxx
ds: xxxx es: xxxx fs: xxxx gs: xxxx
Pid: xx, process nr: xx
xx xx xx xx xx xx xx xx xx xx

or similar kernel debugging information on your screen or in your
system log, please duplicate it *exactly*. The dump may look
incomprehensible to you, but it does contain information that may
help debugging the problem. The text above the dump is also
important: it tells something about why the kernel dumped code (in
the above example it's due to a bad kernel pointer). More information
on making sense of the dump is in Documentation/oops-tracing.txt

- You can use the "ksymoops" program to make sense of the dump. Find
the C++ sources under the scripts/ directory to avoid having to do
the dump lookup by hand:

- In debugging dumps like the above, it helps enormously if you can
look up what the EIP value means. The hex value as such doesn't help
me or anybody else very much: it will depend on your particular
kernel setup. What you should do is take the hex value from the EIP
line (ignore the "0010:"), and look it up in the kernel namelist to
see which kernel function contains the offending address.

To find out the kernel function name, you'll need to find the system
binary associated with the kernel that exhibited the symptom. This is
the file 'linux/vmlinux'. To extract the namelist and match it against
the EIP from the kernel crash, do:

```
ns vmlinux | sort | less
```

This will give you a list of kernel addresses sorted in ascending
order, from which it is simple to find the function that contains the
offending address. Note that the address given by the kernel
debbuging messages will not necessarily match exactly with the
function addresses (in fact, that is very unlikely), so you can't
just 'grep' the list: the list will, however, give you the starting
point of each kernel function, so by looking for the function that
has a starting address lower than the one you are searching for but
is followed by a function with a higher address you will find the one
you want. In fact, it may be a good idea to include a bit of
"context" in your problem report, giving a few lines around the
interesting one.
If you for some reason cannot do the above (you have a pre-compiled kernel image or similar), telling me as much about your setup as possible will help.

- Alternately, you can use gdb on a running kernel. (read-only; i.e. you cannot change values or set break points.) To do this, first compile the kernel with -g; edit arch/i386/Makefile appropriately, then do a "make clean". You'll also need to enable CONFIG_PROC_FS (via "make config").

After you've rebooted with the new kernel, do "gdb vmlinux /proc/kcore". You can now use all the usual gdb commands. The command to look up the point where your system crashed is "1 *0xXXXXXXX". (Replace the XXXes with the EIP value.)

gdb'ing a non-running kernel currently fails because gdb (wrongly) disregards the starting offset for which the kernel is compiled.
Appendix L

RTLinux Installation Manual

RTLinux INSTALLATION INSTRUCTIONS
FSM Labs, Inc.
http://www.fsmlabs.com

DOWNLOADING THE APPROPRIATE LINUX KERNEL:
-----------------------------------------------

In order to compile the RTLinux kernel, you first need to download the
kernel for which RTLinux was built. To do so, note that there are two
patches in the top-level directories:
kernel_patch-2.2, and
kernel_patch-2.4

Where:

kernel_patch-2.2 is for 2.2.18 (x86 only):
http://ftp.kernel.org/pub/linux/kernel/v2.2/linux-2.2.18.tar.gz

kernel_patch-2.4 is for 2.4.0-test1 (x86, PowerPC, Alpha):
http://ftp.kernel.org/pub/linux/kernel/v2.4/linux-2.4.0-test1.tar.gz

Choose which of the two Linux kernels you would like to run, and
download it from the appropriate website, as described above.

PREPARING FOR INSTALLATION:
-----------------------------

Make sure you have gcc 2.7.2.3 or egcs-1.1.2 or egcs-2.91 installed.
You can verify that with
gcc -v

On Debian, it is enough to install the "gcc272" package. On RedHat systems, one needs to install the "kgcc" RPM. You may be able to use other compiler versions, but this is not recommended.

RTLINUX INSTALLATION:
___________________________

If you have downloaded the RTLINUX distribution with a prepatched kernel, skip steps 1 and 2. Quick check: if your kernel contains file arch/i386/kernel/rtlinux.c, you do not need to patch the kernel.

1. put a fresh copy of the Linux kernel in the /usr/src/linux directory:
   
   cd /usr/src
   tar xzf linux-2.2.18.tar.gz
   cd linux

1.b. If you haven’t done so already, put a fresh copy of the RTLINUX kernel in the /usr/src/rtlinux directory:

   cd /usr/src
   tar xzf rtlinux.tar.gz

1.c. Create a symbolic link from within the rtlinux directory to the linux directory:

   cd /usr/src/rtlinux
   ln -sfl /usr/src/linux ./linux

2. Patch the kernel with the RTLINUX patch:

   cd /usr/src/linux
   patch -p1 < /usr/src/rtlinux/kernel_patch-2.2

OR, if you’re using a 2.4.xx kernel:

   patch -p1 < /usr/src/rtlinux/kernel_patch-2.4

3. Now, configure the Linux kernel:

   cd /usr/src/linux
   make config
   or
   make menuconfig
   or
make xconfig

Note: Enabling APM support is not recommended. APM BIOS calls may have unpredictable effect on real-time performance.

Note: On Alpha, you need to enable RTLinux Support (CONFIG_RTLINUX). On i386 and PPC, this is done automatically.

Note: Please make sure to specify the correct CPU type for the target machine.

4. After you are finished configuring the Linux kernel, type:

make dep

Note. Steps 5 through 7 are x86-specific.

5. Compile the Linux kernel and modules:

make bzImage
make modules

5.b Install the Linux modules:

make modules_install

 cp arch/i386/boot/bzImage /boot/rtzImage

6. Configure LILO. To do so, edit /etc/lilo.conf to contain the following piece (you only need to do this once):

image=/boot/rtzImage
label=rtlinux
read-only
root=/dev/hdal

WARNING: replace /dev/hdal in the above with your root filesystem. The easiest way to find out which filesystem it should be, take a look at the existing entry in your /etc/lilo.conf for "root=". Alternatively, type "df", and look for the line for "/" in the "mounted on" column. The corresponding entry in the "Filesystem" column is your root filesystem.

7. Install LILO. To do so, type:

/sbin/lilo

7.b. Restart the computer:

/sbin/shutdown -r now
7. Load the RTLinux kernel: At the LILO prompt, press "Shift" or "Tab". This will give you a listing of the available kernels. Enter:

```
rtlinux
```

RTLlinux should boot.

8. Configure RTLlinux:
   cd /usr/src/rtlinux
   make config OR make menuconfig OR make xconfig

9. Compile RTLlinux:
   make
   make devices
   make install

The last step will create the directory:

```
/usr/rtlinux-xx (xx denotes the version)
```

which contains the default installation directory for RTLinux which is needed to create and compile user programs (that is, it contains the include files, utilities, and documentation). It will also create a symbolic link:

```
/usr/rtlinux
```

which points to /usr/rtlinux-xx. In order to maintain future compatibility, please make sure that all of your own RTLinux programs use /usr/rtlinux as its default path.

POST INSTALLATION AND RUNNING RTLINUX PROGRAMS:
------------------------------------------------------

To be able to run any programs, you must first load the rtlinux modules. To do so, type:

```
/usr/rtlinux/bin/rtlinux start
```

Or

```
/usr/rtlinux/bin/rtlinux start <programname>
```

where <programname> is the name of the rtlinux program/module you want to run.

You can also try running the examples. To do so, simply go to the appropriate directory under /usr/rtlinux/examples and type:
make test

For example:

cd /usr/rtlinux/examples/sound
make test

SPECIAL NOTES:
-------------
If you change any Linux kernel options, please don't forget to do:

cd /usr/src/rtlinux
make clean
make
make install

DOCUMENTATION AND SOURCES OF HELP:
--------------------------------------

The docs/html/GettingStarted document contains a brief introduction to RTLinux. Additional documents in docs/html also provide information about other aspects to RTLinux such as web installation, CD installation, FAQ, and RTiC-Lab.

In case of problems, please consult the FAQ first, available in the docs/ directory.

If all of the above fails, you can obtain help -- free of charge -- from your peers via the rtlQrtlinux.org mailing list for which you can un/subscribe to via http://www.rtlinux.org/mailing_lists.html.

FSM Labs further provides commercial support, development, and training. Please contact FSM Labs at

business@fsmlabs.com for

additional information or visit their website at

Appendix M

J-experiment Data

<table>
<thead>
<tr>
<th>Voltage [V]</th>
<th>Current [A]</th>
<th>Resistance [Ω]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.333</td>
<td>0.1833</td>
<td>1.814</td>
</tr>
<tr>
<td>0.368</td>
<td>0.2216</td>
<td>1.661</td>
</tr>
<tr>
<td>0.403</td>
<td>0.2375</td>
<td>1.697</td>
</tr>
<tr>
<td>0.477</td>
<td>0.2444</td>
<td>1.952</td>
</tr>
<tr>
<td>0.474</td>
<td>0.2506</td>
<td>1.891</td>
</tr>
<tr>
<td>1.327</td>
<td>0.6790</td>
<td>1.954</td>
</tr>
<tr>
<td>1.470</td>
<td>0.8200</td>
<td>1.793</td>
</tr>
<tr>
<td>1.466</td>
<td>0.8694</td>
<td>1.686</td>
</tr>
<tr>
<td>1.310</td>
<td>0.7580</td>
<td>1.728</td>
</tr>
<tr>
<td>1.510</td>
<td>0.9350</td>
<td>1.615</td>
</tr>
</tbody>
</table>

Maximum Measured Armature Resistance 1.954
Minimum Measured Armature Resistance 1.615
Average Measured Armature Resistance 1.779

Table M.1: Armature resistance test results
### Inertia experiments

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>experiments without hovercraft</td>
<td>23.895</td>
<td>0.235</td>
<td>23.477</td>
<td>220.3</td>
<td>0.1066</td>
<td>0.0250</td>
<td>0.000114</td>
<td>0.000295</td>
</tr>
<tr>
<td>experiments with hovercraft</td>
<td>23.650</td>
<td>0.618</td>
<td>22.551</td>
<td>212.6</td>
<td>0.1061</td>
<td>0.0656</td>
<td>0.000308</td>
<td>0.000552</td>
</tr>
</tbody>
</table>

Table M.2: Results of the second tests with constant angular velocity to determine the parameters of the inertia test-rig
Appendix N

Specifications of the Board for Inertia Test

Figure N.1: MDF board to attach the hovercraft to the test-rig.
Appendix O

m-files: DataFit

Determine_J.m

clear all
load data
t    % time vector
om    % measured angular velocity

a=23.9; % step input amplitude
kt=0.1065; % torque constant
kv=kt; % back emf constant
b=0.000114; % damping constant
la=0.62/1000; % inductance
ra=1.779; % resistance

options=optimset('TolFun',[1e-18],'TolX',[1e-18]);
J=fminsearch('Fit_J',0.001,options,t2,om2)

yfit=[];
p1=(ra+la*b)/2/la/J-sqrt((ra+la*b)^2-4*la*J*(ra*b+kt*kv))/2/la/J;
p2=(ra+la*b)/2/la/J+sqrt((ra+la*b)^2-4*la*J*(ra*b+kt*kv))/2/la/J;
for i=1:length(t)
    t2=t(i);
    yfit=[yfit; kt/J/la*a/p1/p2*(1+p2/(p1-p2)*exp(-p1*t2)-p1/(p1-p2)*exp(-p2*t2))];
end
hold on
plot(t,om,t,yfit,'r')

Fit_J.m

function sse=Fit_J(params,input,output)
t2=input;
J=params;

113
APPENDIX O. M-FILES: DATAFIT

a=23.5;
kt=0.1065;
kv=kt;
b=0.000308;
la=0.62/1000;
ra=1.779;
p1=(ra*J+la*b)/2/la/J-sqrt((ra*J+la*b)^2-4*la*J*(ra*b+kt*kv))/2/la/J;
p2=(ra*J+la*b)/2/la/J+sqrt((ra*J+la*b)^2-4*la*J*(ra*b+kt*kv))/2/la/J;

fitted_curve=[];
for i=1:length(t2)
    t=t2(i);
    fitted_curve=[fitted_curve; kt/la*a/p1/p2*(1+p2/(p1-p2)*exp(-p1*t)
                        -p1/(p1-p2)*exp(-p2*t))];
end

error_y=fitted_curve-output;
sse=sum(error_y.^2);
Appendix P

Dimensions of the Hovercraft

Figure P.1: Dimensions of the hovercraft. With C.G.: the center of gravity of the hovercraft, C.P.: center of gravity of the main plate, $b$: the Battery pack, $m_1$: the lift motor, $m_2$: the back thrust motor and $C$: the camera.
### Table P.1: Measures of the hovercraft and components necessary to determine the moment of inertia

<table>
<thead>
<tr>
<th>Component</th>
<th>( m ) [kg]</th>
<th>( l_z )</th>
<th>( l ) [m]</th>
<th>( b ) [m]</th>
<th>( r ) [m]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Main plate</td>
<td>0.955</td>
<td>( l_z_1 = 0.08 )</td>
<td>0.76</td>
<td>0.39</td>
<td>-</td>
</tr>
<tr>
<td>Battery pack</td>
<td>0.625</td>
<td>( l_z_4 = 0.13 )</td>
<td>0.13</td>
<td>0.05</td>
<td>-</td>
</tr>
<tr>
<td>Lift motor</td>
<td>0.100</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0.015</td>
</tr>
<tr>
<td>Back thrust motor</td>
<td>0.135</td>
<td>( l_z_3 = 0.20 )</td>
<td>0.07</td>
<td>-</td>
<td>0.015</td>
</tr>
<tr>
<td>Camera</td>
<td>0.285</td>
<td>( l_z_2 = 0.11 )</td>
<td>-</td>
<td>-</td>
<td>0.035</td>
</tr>
</tbody>
</table>
Appendix Q

m-files: Simulation

Modelsim.m

clear all
close all

%Starting point
x0=0;
y0=0;
fio=0;
u0=0;
v0=0;
ro=0;

z10=cos(fio)*x0+sin(fio)*y0;
z20=-sin(fio)*x0+cos(fio)*y0;
z30=fio;

[t,sol]=ode45('modelsim7ehv',[0,60],[x0,z20,z30,u0,v0,ro]);
z1=sol(:,1);z2=sol(:,2);z3=sol(:,3);u=sol(:,4);v=sol(:,5);w=sol(:,6);

x=[];
y=[];
fi=[];
for hihi=1:length(sol(:,1))
    trans=inv([cos(z3(hihi,1)) sin(z3(hihi,1));
               -sin(z3(hihi,1)) cos(z3(hihi,1))]);
    x=[x trans(1,:)*z1(hihi,1) z2(hihi,1)]';
    y=[y trans(2,:)*z1(hihi,1) z2(hihi,1)]';
    fi=[fi z3(hihi,1)];
end

figure(),plot(t,u),title('surge');
xlabel('t [ s ]'),ylabel('velocity [ m/s ]')
hold on
plot([0 40], [2.25 2.25], 'k', [40 40], [2.25 1.125], 'k',
     [40 50], [1.125 1.125], 'k', [50 50], [1.125 0], 'k', [50 60], [0 0], 'k')
plot([0 5], [0 0], 'r', [5 5], [0 1.5], 'r', [5 60], [1.5 1.5], 'r')
figure(2), plot(t, v), title('sway');
xlabel('t [ s ]'), ylabel('velocity [ m/s ]')
figure(3), plot(t, w), title('yaw');
xlabel('t [ s ]'), ylabel('angular velocity [ rad/s ]')
figure(4)
okj = find(t(5>5 & t>0));
plot([x(okj) x(length(okj)+1)], [y(okj) y(length(okj)+1)], 'b')
hold on
kleurtjes = ['r' 'r' 'r' 'r' 'r' 'k' 'k' 'k' 'k']
for ay = 1:11
    okj = find(t(5>5 & t>ay*5));
    plot([x(okj) x(length(okj)+1)], [y(okj) y(length(okj)+1)], kleurtjes(ay))
    hold on
end
title('Movement of the hovercraft in earth coordinates');
hold on
x2 = x(find(t==45));
y2 = y(find(t>=45));
[i, j] = size(sol);
ll1 = 0.3; % length of the ship
n = [30 34 82 99 105 112 114 116 119
     124 131 138 150 190 227 245 259 578 460];
colortjes = ['b' 'b' 'r' 'r' 'r' 'r' 'r' 'r' 'r' 'r'
               'r' 'r' 'r' 'r' 'r' 'r' 'r' 'k' 'k' 'k' 'k']
for nij = 1:length(n)
    shipplot(x(1, n(nij)), y(1, n(nij)), fi(1, n(nij)), ll1, colortjes(nij));
end
xlabel('X [ m ]'), ylabel('Y [ m ]')
figure(5), subplot(2, 4, 1:16);
plot(t, u), hold on, title('surge');
ylabel('velocity [ m/s ]'), plot([0 40], [2.25 2.25], 'k', [40 40], [2.25 1.125], 'k', [40 50],
     [1.125 1.125], 'k', [50 50], [1.125 0], 'k', [50 60], [0 0], 'k')
plot([0 5], [0 0], 'r', [5 5], [0 1.5], 'r', [5 60], [1.5 1.5], 'r')
subplot(2, 4, 18), title('Fl'), subplot(2, 4, 20), title('Rudder');
subplot(2, 4, 23:38) plot(t, v), hold on,
title('sway'), xlabel('t [ s ]'), ylabel('velocity [ m/s ]'),
plot([0 40], [2.25 2.25], 'k', [40 40], [2.25 1.125], 'k', [40 50],
     [1.125 1.125], 'k', [50 50], [1.125 0], 'k', [50 60], [0 0], 'k')
plot([0 5], [0 0], 'r', [5 5], [0 1.5], 'r', [5 60], [1.5 1.5], 'r')
APPENDIX Q. M-FILES: SIMULATION

```matlab
subplot(2,4,40),title('Fl'),subplot(2,4,42),title('Rudder');
subplot(2,4,43:58),plot(t,w),hold on,title('yaw'),
plot([0 40],[2.25 2.25],'k',[40 40],[2.25 1.125],'k',[40 50],
[1.125 1.125],'k',[50 50],[1.125 0],'k',[50 60],[0 0],'k')
plot([0 5],[0 5],'r',[5 5],[0 1.5],'r',[5 60],[1.5 1.5],'r')
xlabel('t [ s ]'),ylabel('angular velocity [ rad/s ]'),

subplot(2,4,60),title('Fl'),subplot(2,4,62),title('Rudder');

subplot(2,4,66:84)
okj=find(t<5*5&t>0);
plot([x(okj) x(okj(length(okj))+1)],y(okj) y(okj(length(okj))+1),b')
hold on
for ay=1:11
  okj=find(t<(ay+1)*5&t>ay*5);
  plot([x(okj) x(okj(length(okj))+1)],
  [y(okj) y(okj(length(okj))+1)],kleurtjes(ay))
  hold on
end
title('Movement of the hovercraft in earth coordinates');
hold on
[i,j]=size(sol);
xlabel('X [ m ]'),ylabel('Y [ m ]')

set(4,'Position',[0 50 530-30 530-50])
set(8,'Position',[500 50 530-30 530-50])

Model.m

function [xdot]=modelsim7ehv(t,x)

% Defining parameters
m=2.1;
d11=0.6;
d22=0.8;
d33=.1;
a=0.4;
g=10;
mu1=0.1;
mu2=0.01;
mu3=0.004;
Jz=0.0948;

T1=5;
Fx=[1.8 1.8 1.8 1.8 1.8 1.8 1.8 1.8 1.8 1.8];
ang=[0 -10 -10 -10 -10 -10 -10 -10 -10 -10];
```
F1=[21 21 21 21 21 21 21 21 10 10 0]; % lift force

% Simulation intervals
if t<=T1
    Fxx=Fx(1); % in Newton
    d=ang(1)/360*2*pi; % in rad
    Fll=F1(1);
elseif t>T1&t<=2*T1
    Fxx=Fx(2);
    d=ang(2)/360*2*pi;
    Fll=F1(2);
elseif t>2*T1&t<=3*T1
    Fxx=Fx(3);
    d=ang(3)/360*2*pi;
    Fll=F1(3);
elseif t>3*T1&t<=4*T1
    Fxx=Fx(4);
    d=ang(4)/360*2*pi;
    Fll=F1(4);
elseif t>4*T1&t<=5*T1
    Fxx=Fx(5);
    d=ang(5)/360*2*pi;
    Fll=F1(5);
elseif t>5*T1&t<=6*T1
    Fxx=Fx(6);
    d=ang(6)/360*2*pi;
    Fll=F1(6);
elseif t>6*T1&t<=7*T1
    Fxx=Fx(7);
    d=ang(7)/360*2*pi;
    Fll=F1(7);
elseif t>7*T1&t<=8*T1
    Fxx=Fx(8);
    d=ang(8)/360*2*pi;
    Fll=F1(8);
elseif t>8*T1&t<=9*T1
    Fxx=Fx(9);
    d=ang(9)/360*2*pi;
    Fll=F1(9);
elseif t>9*T1&t<=10*T1
    Fxx=Fx(10);
    d=ang(10)/360*2*pi;
    Fll=F1(10);
else
    Fxx=Fx(11);
    d=ang(11)/360*2*pi;
    Fll=F1(11);
end
APPENDIX Q. M-FILES: SIMULATION

function y=shipplot(x,y,fi,length,kleur)

% y=shipplot(x,y,fi,length,kleur)
% give the coordinates, angle, length of the ship and color of the ship

lll=length; % length of the ship

Ax=x-cos(fi)*1.4*lll; Ay=y-sin(fi)*1.4*lll;
Ax2=x-cos(fi)*0.8*lll; Ay2=y-sin(fi)*0.8*lll;
Bx=x+cos(fi)*1.4*lll; By=y+sin(fi)*1.4*lll;
Dx=Ax-sin(fi)*0.5*lll; Dy=Ay+cos(fi)*0.5*lll;
Ex=Ax+sin(fi)*0.5*lll; Ey=Ay-cos(fi)*0.5*lll;
Cx=Bx-sin(fi)*0.5*lll*1+tan(40/360*2*pi)*cos(fi);
Cy=By+sin(fi)*0.5*lll*1+tan(40/360*2*pi)*sin(fi);
Fx=Bx*sin(fi)*0.5*lll*1+tan(40/360*2*pi)*cos(fi);  
Fy=By*cos(fi)*0.5*lll*1+tan(40/360*2*pi)*sin(fi);

plot([Ax3 Ax2], [Ay3 Ay2], kleur, [Dx Ex], [Dy Ey], kleur, [Bx Cx], [By Cy], kleur, [Fx Ex], [Fy Ey], kleur)