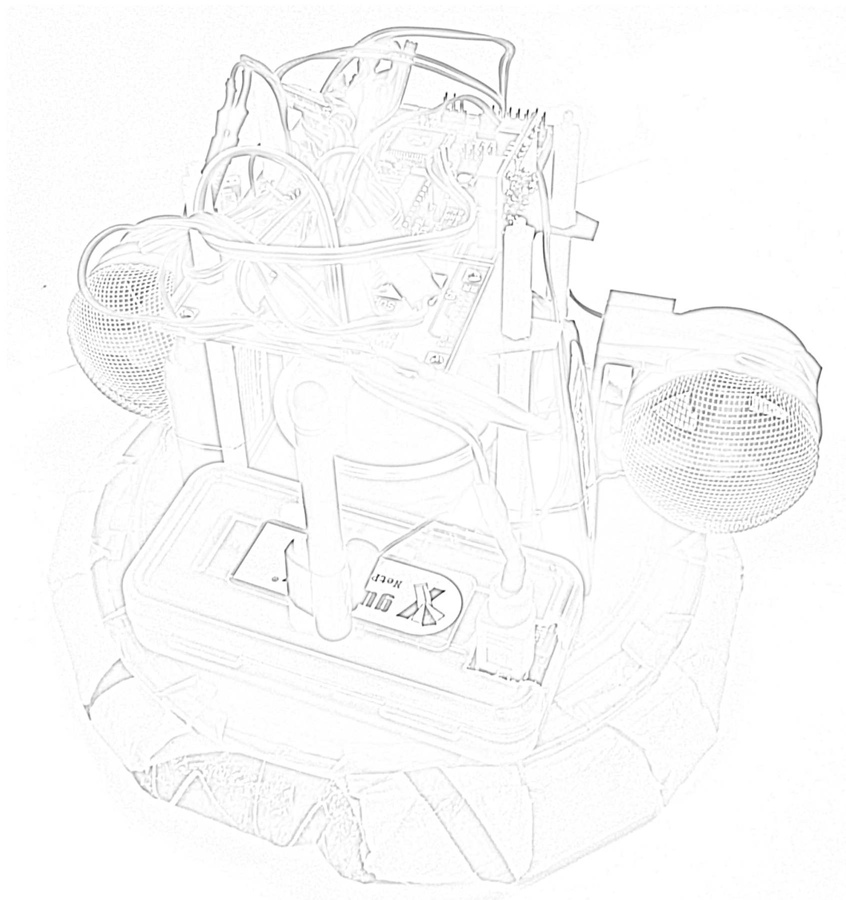
Relaunch The MVWT Hover Craft

Stephan Pleines Aldo Zgraggen

January 9, 2009



Contents

1 Introduction **1**

2 Hardware **2**

2.1 How To Build A Hovercraft . . . 2

2.1.1 Components . . . 2

2.1.2 Assembly . . . 3

3 Gumstix **8**

3.1 Set Up The Gumstix . . . 8

3.1.1 Assembly . . . 8

3.1.2 Getting started . . . 10

3.1.3 Serial connection to gumstix . . . 12

3.1.4 Replacing file system . . . 12

3.1.5 Package maintenance . . . 14

3.1.6 Bitbake . . . 15

3.2 Cross Compiling Libraries . . . 16

3.2.1 Cross compiling Berkeley database . . . 17

3.2.2 Cross compiling ncurses . . . 17

3.2.3 Cross compiling SPARROW . . . 18

3.2.4 Cross compiling SPREAD . . . 18

4 ATMEL **19**

4.1 Program The ATMEL ATmega128 . . . 19

4.1.1 Prerequisites . . . 19

4.1.2 Debugging and simulation . . . 20

4.1.3 Flashing the ATmega128 . . . 20

4.2 Changes . . . 21

4.3 Possible Improvements . . . 22

4.3.1 Watchdog for battery voltage . . . 22

4.3.2 Replace the MVWT circuit board . . . 22
4.3.3 Replace crystal . . . 23

5 Sparrow Drivers **24**
5.1 Serial Driver . . . 24
5.1.1 Reading the sensor data stream . . . 24
5.1.2 The driver . . . 26
5.2 Vision Driver . . . 27
5.2.1 Start the vision system . . . 27
5.2.2 Data format and protocol . . . 27
5.2.3 The driver . . . 27
5.3 Command Driver . . . 28
5.3.1 Data format and protocol . . . 29
5.3.2 The driver . . . 29
5.4 Trajectory Driver . . . 30

6 Model **31**
6.1 The model . . . 31
6.1.1 Equations of motion . . . 31
6.1.2 Discussion . . . 34
6.2 Parameter identification . . . 35
6.2.1 Weight . . . 35
6.2.2 Inertia . . . 35
6.2.3 Translational friction . . . 36
6.2.4 Rotational friction . . . 37

7 Hovercraft Local Controller **41**
7.1 Devices . . . 41
7.2 Binary . . . 41
7.2.1 Display disabled . . . 41
7.2.2 The dynamic display . . . 42
7.3 The really really simple PID controller . . . 42
7.4 Another controller approach . . . 44

8 Command Center **47**
8.1 Drivers . . . 47
8.2 Binary . . . 48

8.3 Algorithms 48
8.4 What works and what does not 48

A Timeline **50**

B The Hidden Treasures (SVN) **52**

C Starting The System **53**

D Doxygen **54**

E Hardware and Parts **55**

F Documentation Of Previous Projects **56**

Chapter 1

Introduction

This 3 month, 2 persons project aimed at reactivating the Multi Vehicle Wireless Testbed (MVWT) and especially improving the hovercraft vehicles. The objectives were to

1. set up the testbed in the basement,
2. build a fleet of 3 hovercrafts that can follow a path within a certain error,
3. implement a control structure using open source control software (SPARROW & FALCON),
4. increase the onboard computational power of the hovercrafts,
5. document everything properly.

Objective 1 was achieved. The lab was cleaned, the testbed is operational. The vision system is still operational, it was not altered.

Objective 2 was not achieved. Read the details in chp. 8.

Objective 3 was achieved. The local controller and the command station as well is now based on SPARROW and FALCON, which had to be slightly adapted to the new computational platform.

Objective 4 was achieved. The old PDA's were replaced with gumstix embedded computers. The code running on the ATMELS on the existing PCBs had to be altered because of problems with the serial connection, which has been a major bug in previous MVWT projects.

The question whether objective 5 was achieved is left to be answered by the reader. Please also refer to the code documentation generated by DOXYGEN (cp. app. D).

Chapter 2

Hardware

2.1 How To Build A Hovercraft

2.1.1 Components

The following components are needed to assemble a hovercraft:

Chassis and fan

The chassis and the fans were not altered in this project. Two different chassis were used, with and without a skirt. The skirtless chassis has the major drawback that the the lift fan will cause a steady rotation of the hovercraft, which can not be compensated. This effect is caused by the bars that connect the motor to the housing of the fan. It can not hover in steady state. This is less of a problem with the skirt version, because the skirt causes more friction which keeps the hovercraft from rotating. This does not fully remove the effect, but it is easier to control.

Figure 2.1: The hovercraft chassis.

Batteries

New lithium polymer batteries were bought: 2 cells, capacity 2000 mAh, nominal voltage 7.4 V, manufactured by Thunderpower (cp. fig. 2.2). Do not discharge below 6.8 V nor

charge above 8.4 V. For charging, use the Graupner LiPo Charger 4. Each hovercraft is powered by 2 batteries, there is a total of 6 batteries and 3 chargers.

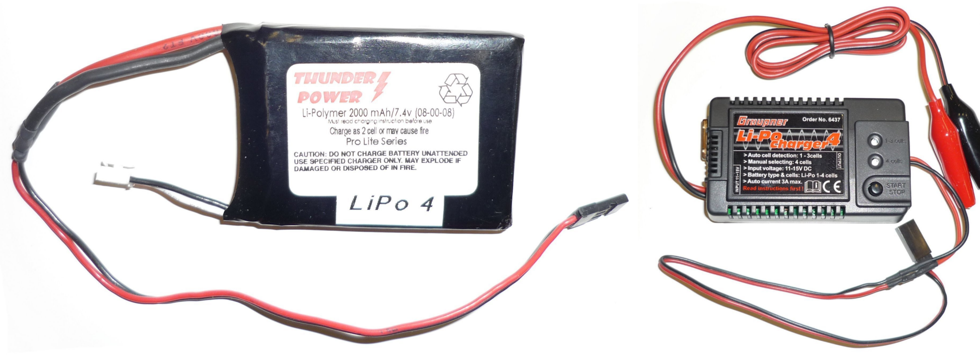


Figure 2.2: The new charger and LiPo battery: 2 cells, capacity 2000 mAh, 7.4 V.

The PCB

The PCB was modified in the way that a cable to connect the gumstix to the 5 V voltage regulator on the board was added (cp. fig. 2.3). Spare plugs to modify more boards are in stock. Also, the Atmel was reflashed, see sec. 4.

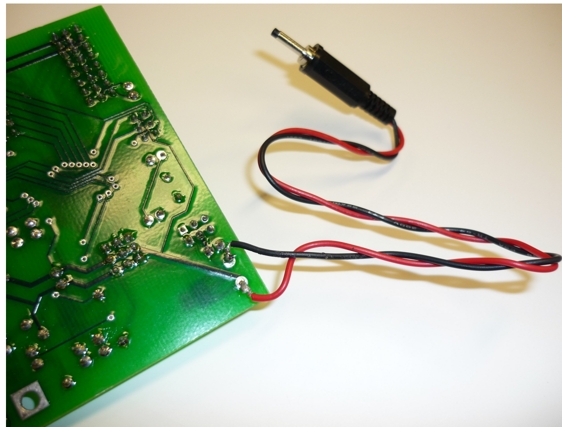


Figure 2.3: This cable provides 5V to the gumstix.

Serial Cable

The gumstix serial port has a mini DIN 8 connector, the PCB has a simple 3 pin connector. Connect the pins as show in fig. 2.4. Figure 2.5 shows how to connect the cable to the gumstix and PCB respectively.

2.1.2 Assembly

The thrust fans are mounted on the U bars with either elastic strap, tape, or wire. Their chassis is not very stiff - avoid deforming them or the blades will touch the housing. Mount the PCB on the hexagonal rods above the thrust fan. Use Velcro to mount the gumstix in its box on the nose of the hovercraft. The batteries go to the side, attached by a string

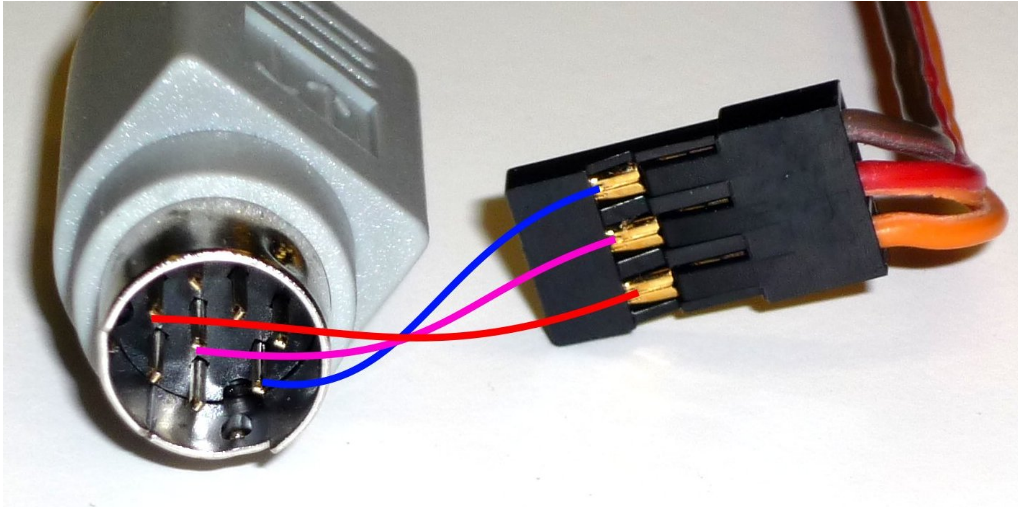


Figure 2.4: Serial cable to connect the gumstix to the PCB.

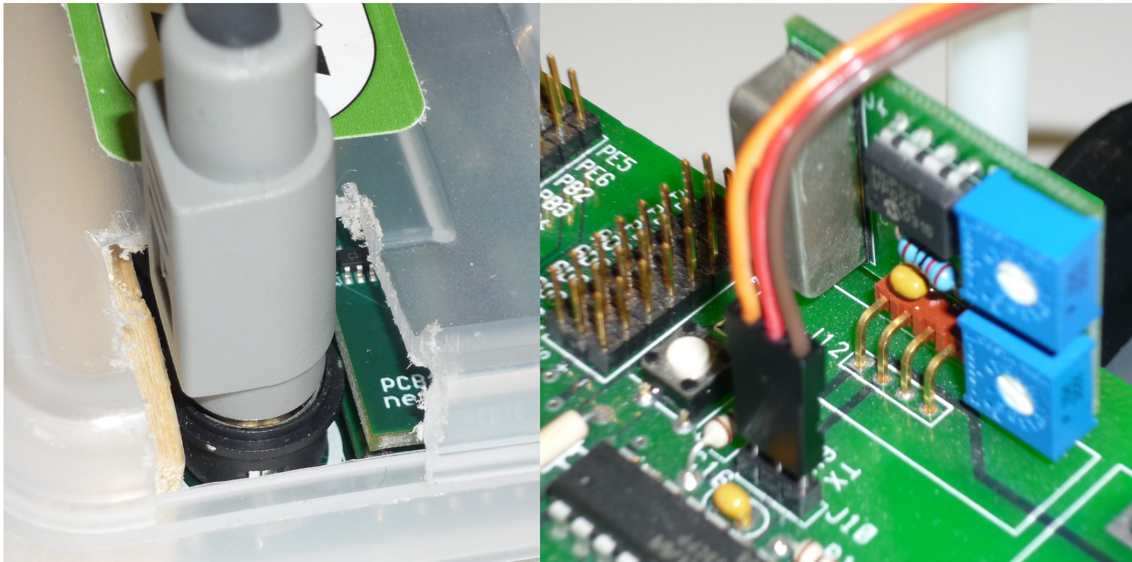


Figure 2.5: How to connect the serial cable to the gumstix and to the PCB.

for example. If necessary, use weights to balance the hovercraft. If it is not balanced, it will not hover, but move, even if the thrust fans are turned off. The assembled hovercraft (without the vision system head) is shown in fig. 2.8.

Cable Connections

Connect the batteries to the fans and to the board via the motor controllers and additional Y cables as shown in fig. 2.6. Due to the design of the PCB it might be that both batteries power the board through the motor controllers.

The serial cable goes to port ttyS2 on the gumstix, this is the port on the corner of the LCD console extension board. The other end goes to the PCB. The side of the connector where one can see the 3 metal connectors faces the gyroscope. Don't worry, there is a 50%

chance to do it right.

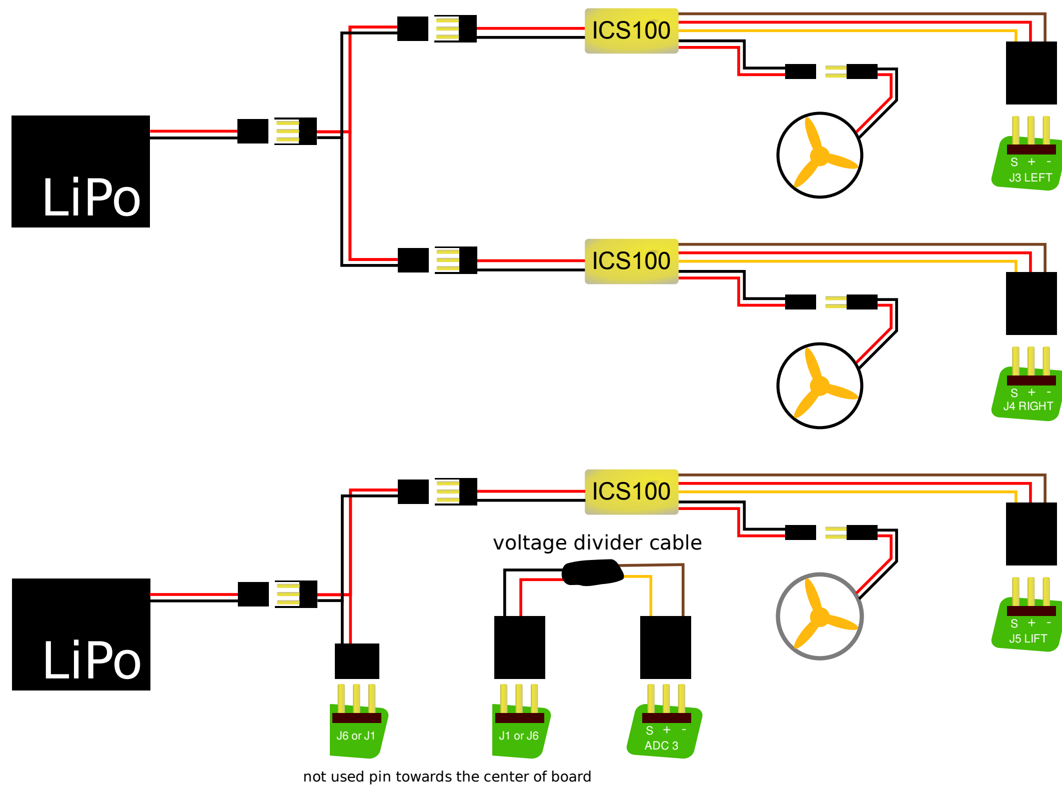


Figure 2.6: How to connect the electrical components of the hovercraft.

Gumstix Mounting

One of the boxes that came with the gumstix, containing an extension board, is used to mount the gumstix. It has been modified, so connectors can be attached. The box is mounted on the front of the chassis with Velcro (cp. fig. 2.7).

Figure 2.7: The gumstix in the box that is used to mount it on the hovercraft.

Figure 2.8: The assembled hovercraft (without the vision system hat).

Chapter 3

Gumstix

This chapter is an introduction on how to us the gumstix. One section is about setting up the gumstix and how to add new packages. Other sections deal with cross compiling important libraries for the ARM architecture. Everything described in this chapter was done using a linux box running Ubuntu 8.04.1LST Hardy Heron (kernel release 2.6.24-22-generic). This is also true for the subsequent chapter about the ATMEL programming[1].

3.1 Set Up The Gumstix

This section should tell you how you get started with your gumstix verdex pro XL6P (gumstix) with the netpro-vx, the consoleLCD16-vx and the wifi module FCC (incl. u.fl antenna) extension kits.

3.1.1 Assembly

It is a little bit tricky and unfortunately there is no casing available yet. First you should attach the wifi antenna to the wifi module and then the wifi module to the netpro-vx part (the one with the RJ-45 socket). You can fix the wifi module with one screw if you like to but be aware that on one side the screw head would collide with the lower module. Next you should attach the LCD screen to the consoleLCD16-vx. To do so pull the tiny black spots on the outside of the 45-pin connector forward. Then you should be able to plug in the flexible connector of the LCD and push back the black spots. The next thing to do is to screw all three parts together. Make sure you attach the netpro-vx to the verdex pro XL6P (XL6P) first and don't forget to insert a middle length screw in the lower left corner (RJ-45 socket on top) of the XL6P (cp. fig. 3.4). Then attach both to the consoleLCD16-vx and pull the three boards tight together with two long screws and the black spacers. Now you are ready to go. The fully assembled gumstix can be seen in fig. 3.1, 3.2, 3.3 and 3.4.

[1] Two linux boxes in the MVWT lab were set up with Ubuntu: Beijing: user&pwd: `mvwt` and Sydney: user&pwd: `mvwt2008`. Use this logins to create your own account on these local machines

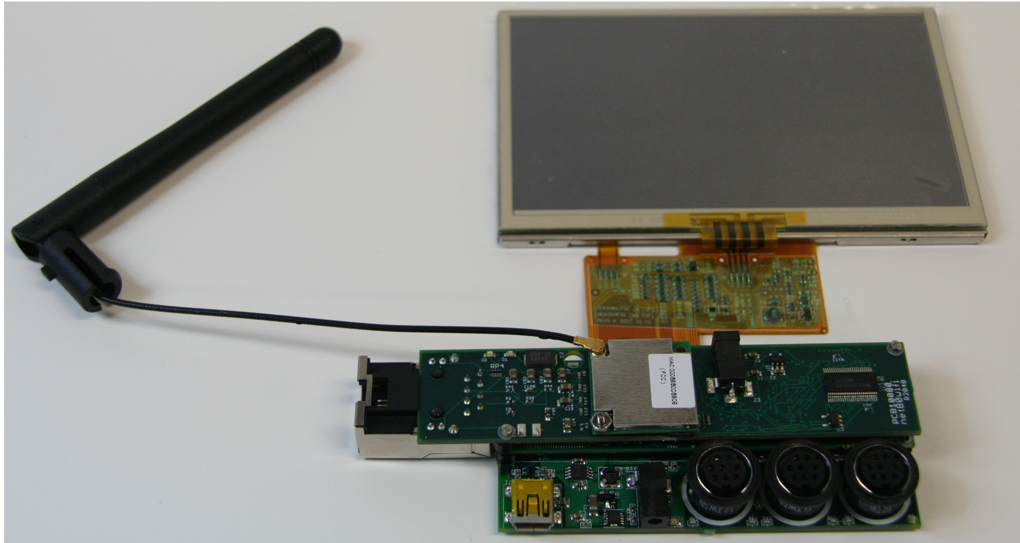


Figure 3.1: The assembled gumstix.

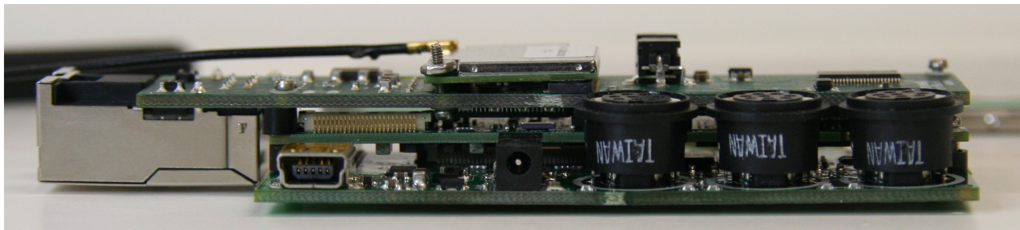


Figure 3.2: The assembled gumstix - front side.

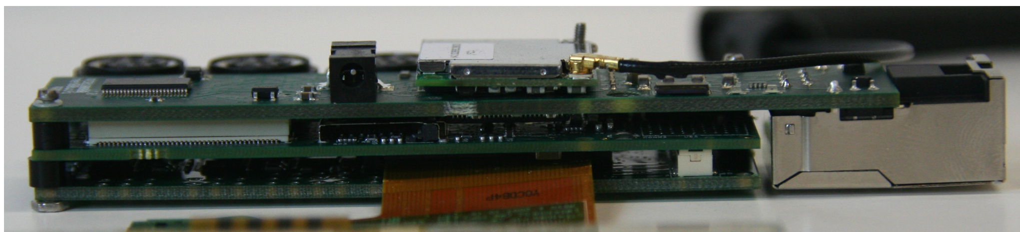


Figure 3.3: The assembled gumstix - back side.

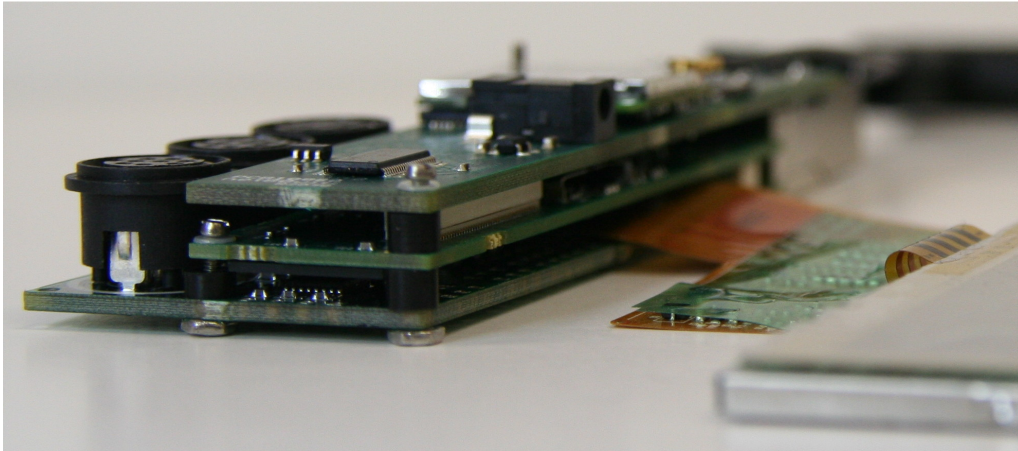


Figure 3.4: The assembled gumstix - right side.

3.1.2 Getting started

Here you can find lots of useful information: Getting Started[2]. The steps described here are based on the manual from exactly this site.

1. Check out the OpenEmbedded source code SVN:

```
$ mkdir ~/gumstix
$ cd ~/gumstix
$ svn co https://gumstix.svn.sourceforge.net/svnroot/gumstix/
         trunk gumstix-oe
```

   This may take a little while (about 30min). If it fails during the checkout process just try 'svn up'.

2. Environment setup

```
$ cat gumstix-oe/extras/profile >> ~/.bashrc
```

   This will include some used path for gumstix in your profile

3. Source code caching

```
$ sudo groupadd oe
$ sudo usermod -a -G oe your_username
$ sudo mkdir /usr/share/sources
$ sudo chgrp oe /usr/share/sources
$ sudo chmod 0775 /usr/share/sources
$ sudo chmod ug+s /usr/share/sources
```

[2] http://www.gumstix.net/Software/cat/Getting-started/

This will create a group called oe and 'your_username' will be added as a member of this group. If this group already exists just add your username. `CHMOD 0775` gives your user and group read/write access. `CHMOD ug+s` sets the rights for the compiled package[3]. After that restart your X-server by pressing 'Ctrl+Alt+Backspace' and log in again in order to let the changes in the user groups take effect.

4. Packages that you need are listed below. This is depending on your machine and there may be other packages missing as well. The naming can be slightly different from distribution to distribution.

   - gcc
   - g++
   - patch (these first 2 are often bundled with other developer tools in the build-essential package)
   - help2man (Centos 5 package available from atrpms repository)
   - diffstat
   - texi2html (texinfo on SUSE)
   - makeinfo (texinfo on Ubuntu)
   - ncurses-devel (libncurses5-dev on Ubuntu)
   - cvs
   - gawk
   - python-dev
   - python-pysqlite2 (python-sqlite2 on SUSE)

   On Ubuntu just use `sudo apt-get install 'package name'` to install missing packages or if you are still trying to avoid console based instructions use your synaptic package manager GUI. There is also a recommendation that you install the Psyco compiler from psyco.sf.net. To do so just load the svn repository on your machine

   ```
   $ svn co http://codespeak.net/svn/psyco/dist/ psyco-dist
   ```

   And install with the top-level script setup.py. Make sure that the `python-dev` package is installed.

   ```
   $ python setup.py install
   ```

5. Now we finally start building the new environment.

   ```
   $ bitbake gumstix-basic-image
   ```

   This may take a while or two to get all online resources and compile it. If the compiling fails which occurred often on gentoo machines (not yet sure if it was because of the global user setting or not) you can begin from scratch with

   ```
   $ rm -rf ~/gumstix/gumstix-oe/tmp
   ```

[3]The need of this command `CHMOd ug+s` is not clear. This will give any user executing a binary from this source the rights from the user who compiled it, if I'm not wrong. So take care what you are doing.

3.1.3 Serial connection to gumstix

First you have to get a serial cable with the right plugs for you computer and the gumstix. There should be one in the gumstix package which is pretty short. So just get an extension. Use the middle serial port on the console expansion board. If this is set install the Kermit software on your linux machine if it is not yet installed.

- Install Kermit (minicom is another option)

  ```
  $ sudo apt-get install ckermit
  ```

- Plug in the serial cable to your PC and gumstix.
- Launch kermit (`/dev/ttyS0` for serial and `/dev/ttyUSB0` for usb connection).

  ```
  $ sudo kermit -l /dev/ttyS0

  C-Kermit> take ~/gumstix/gumstix-oe/extras/kermit-setup
  C-Kermit> connect
  ```

- Now power up the gumstix by plugging in the power cable in any of the power jacks.
- You will see the u-boot sequence followed by the normal gumstix boot sequence. After that you can login remotely (you do not need the LCD) with user: `root` and password: `gumstix`.

3.1.4 Replacing file system

- First of all complete all steps of the section 3.1.3. Then type `reboot` and the gumstix will begin to reboot after a few seconds. On one point the u-boot script, visible on your PC screen, will wait for a moment. At this time you should hit any key to interrupt the u-boot sequence. You will then see a new console called ‘GUM>’.
- Tell u-boot to prepare to write on the RAM location a2000000 with

  ```
  GUM> laodb a2000000
  ```

- Then hit ‘Ctrl+Backslash’ and then c. You will break the serial connection and end up in the kermit terminal. There you provide gumstix with the new file system.

  ```
  C-Kermit> cd ~/gumstix/gumstix-oe/tmp/deploy/glibc/images/
            gumstix-custom-verdex/
  C-Kermit> send gumstix-basic-image-gumstix-custom-verdex.jffs2
  ```

  The names of the directory and the file *.jffs2 can vary. Adapt it to your system. You can use tab completion within kermit. After executing the send command you will see a screen with the progress of the transfer. After that the filesystem is stored in the RAM so do not unpower the gumstix. The contents of FLASH is still unchanged.

- Reconnect to the gumstix

```
C-Kermit> connect
```

- Now as we are reconnected to the gumstix we have to replace the FLASH with the content of the RAM. BE VERY CAREFUL with the next step. DO NOT FORGET to protect the first two FLASH sectors!

```
GUM> protect on 1:0-1
```

  You should see that the protection was successfully applied and we can proceed with erasing the FLASH and rewrite it.

```
GUM> erase all
GUM> cp.b a2000000 40000 ${filesize}
```

  The variable *filesize* was set by the previous file transfer.

- Now we load the kernel

```
GUM> loadb a2000000
```

  The terminal will show you that the gumstix is now ready to receive the data. So press 'Ctrl+backslash and' c again to return to the kermit terminal.

```
C-Kermit> send uImage-2.6.21-r1-gumstix-custom-verdex.bin
```

  And again data is transmitted. Afterwards we can reconnect to the gumstix.

```
C-Kermit> connect
```

  Back on the gumstix terminal type

```
GUM> katinstall 100000
GUM> katload 100000
GUM> bootm
```

  To get back to the normal boot just type

```
GUM> boot
```

3.1.5 Package maintenance

This part was copied from gumstix.net[4]. You will need a network connection on your gumstix to utilize ipkg. To update all installed packages on your gumstix to the latest version, type in the gumstix terminal:

```
$ ipkg update
$ ipkg upgrade
```

The first command downloads a directory of all packages currently in the repository. The second command will scan the list of all currently installed packages, determine which have upgrades available, and then finally download and install those upgrades.

Your gumstix ships from the factory with just a basic set of software pre-installed. The ipkg repository contains a large number of additional packages. To see a list of those packages:

```
$ ipkg update
$ ipkg list
```

Installing a new package is quite simple. For example, to install the perl interpreter package:

```
$ ipkg install perl
```

The package management system will automatically install any other packages that are required for proper functioning of your chosen package.

While the gumstix ipkg repository contains a large number of packages, OpenEmbedded itself has recipes for an even greater number of packages. You can see the available package recipes by browsing the `org.openembedded.snapshot/packages` directory in your build tree.

If you find that a package you need is not available as a pre-built binary in the repository, it is a simple matter to build it. For example, to build the strace package on your linux box:

```
$ bitbake strace
```

After the build completes, the ipkg file for the strace package will be placed in the gumstix-oe tree in `.../tmp/deploy/glibc/ipk/armv5te` (assuming you have selected glibc builds). To install the strace package you would transfer the strace ipgk file (in this case strace_4.5.14-r4_armv5te.ipk) to your gumstix using scp (or compact flash, mmc, microSD, USB drive, etc) and then type:

```
$ ipkg install strace_4.5.14-r4_armv5te.ipk
```

If you want to remove the installed package from the gumstix just type

```
$ ipkg remove strace
```

The package name you use with this command is the same as you would type to execute it and not the full package name with the version and revision number.

[4] http://www.gumstix.net/Software/view/Getting-started/Updating-and-adding-packages-via-ipkg/111.html

3.1.6 Bitbake

The gumstix open embedded build system has its own directory layout which is explained on gumstix.net[5]. The `user.collection` directory has to be created first and can be used for any purpose but it is a good idea to follow the same structure as in the official packages. In Fig 3.1.6 you see how the directory tree of the MVWT project looks like.

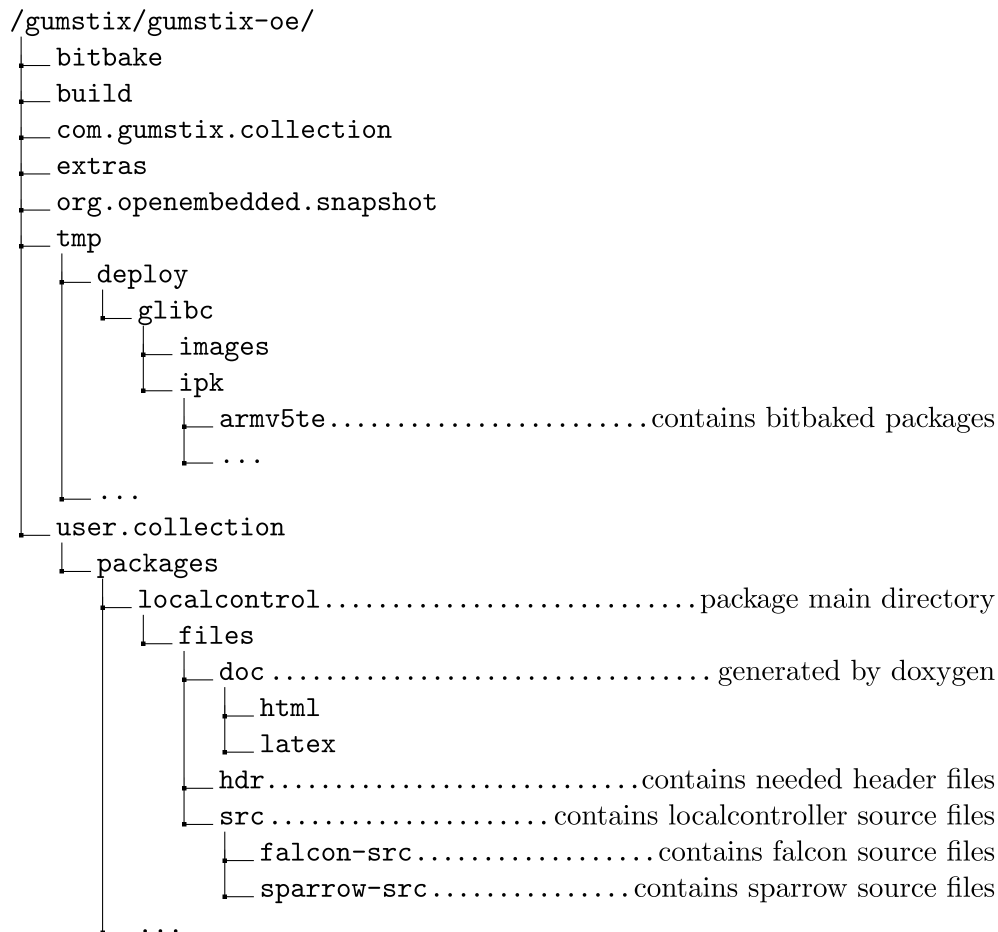


Figure 3.5: Directory structure used for the gumstix build environment

Since the directory `user.collection` is made by the user it is not part of the subversion of the gumstix-oe which means that it should be included in its own subversion tree. The packages in the `user.collection` directory have higher priority than the packages from the `com.gumstix.collection` which again have higher priority than those from the `org.openembedded.snapshot` directory.

There are mainly two ways to install new packages on the gumstix. Either you install all packages by hand or include all needed packages in the file system image and replace the whole image. Our gumstix will just need the `localcontrol` packages and thus replacing the file system image is not explained here. In section 3.1.5 is stated how to install new packages and this also applies for self made packages.

Bitbake needs its own "Makefile" which has the ending `*.bb`. This file tells bitbake what the name, the version and the revision of this package is and is usually placed

[5]http://www.gumstix.net/Software/view/Build-system-overview/Directory-layout/111.html

in the package main directory. A simple example can again be found on gumstix.net[6]. The bitbake file for the localcontrol package looks a little bit more complex but is still pretty straight forward[7]. Firstly, all needed compiler flags and library flags are defined. Below there is a list of all files used for building the package. Keep in mind that bitbake will simply copy all files into another directory without the original directory hierarchy. This can cause conflicts if there is a header file inclusion from a local subdirectory. The next part is the `do_compile()` command which compiles all source files. In our case it is a rather complicated sequence because we are also compiling the Sparrow and the Falcon library besides the localcontrol files. This is because the libraries were altered during our development and as soon as the libraries are finished this should be changed.

To bitbake your package just type

```
$ bitbake localcontrol
```

You don't have to be in the directory where the `*.bb` file is located, bitbake will find it for you. Bitbake just needs the package name and not the full name with the version number. If you changed the `*.bb file` and added other resources it is a good idea to clean up the build directory for this package. This is done by

```
$ bitbake -c rebuild localcontrol
```

3.2 Cross Compiling Libraries

On some systems, when using `bitbake` to compile an application containing Sparrow, it might be necessary to cross compile the ncurses library first. To cross compile ncurses for the gumstix, it is necessary to crosscompile the Berkeley database. Both is explained here, allthough it hopefully won't be necessary. Besides that we need to cross compile the Spread library for the gumstix.

Figure 3.6: Sparrow has successfully been installed on the gumstix.

[6] http://www.gumstix.net/Software/view/Build-system-overview/Hello-world-tutorial/111.html
[7] One can also include "Makefiles" directly in the `*.bb` file but this can get a little bit confusing

3.2.1 Cross compiling Berkeley database

1. Download and extract the Berkeley database source (oracle.com[8]).
2. Copy the cross compiler `arm-angstrom-linux-gnueabi-gcc` from your `gumstix-oe/tmp/cross/bin/` directory to the `/bin/` directory, so you can use the command in the makefile. This will be undone later.
3. `cd` into the db source directory, then `cd build_unix`.
4. Execute (change directories as necessary)

```
../dist/configure \
--host=arm-angstrom-linux-gnueabi-gcc \
--prefix=<gumstix-oe>/tmp/staging/arm-angstrom-linux-gnueabi/
```

5. Run `make`.
6. Run `make install`.

`db.h` should now be found in `/tmp/staging/arm-angstrom-linux-gnueabi/include/`.

3.2.2 Cross compiling ncurses

1. Download ncursrs from `http://ftp.gnu.org/pub/gnu/ncurses/` , untar it.
2. `cd` to the ncurses source directory.
3. Replace `<gumstix-oe>` with the path to your gumstix-oe directory, typically `~/gumstix/gumstix-oe` and run

```
CC=arm-angstrom-linux-gnueabi-gcc \
./configure arm-linux \
--host=arm-angstrom-linux-gnueabi \
--target=arm-angstrom-linux-gnueabi \
--disable-database \
--with-fallback=linux \
--enable-termcap \
--with-normal \
--with-shared \
--prefix=<gumstix-oe>/tmp/staging/arm-angstrom-linux-gnueabi
```

4. Run `make`.
5. Run `make install`.
6. Now this might seem slightly weird: The cross compiler can be found in `gumstix-oe/tmp/staging/` and `gumstix-oe/tmp/cross/` (or part of it is in either place). Replace `gumstix-oe/tmp/cross/include` with a symbolic link to `gumstix-oe/tmp/staging/include`.
7. Remove the cross compiler from `/bin/` if you like.

[8] http://www.oracle.com/technology/software/products/berkeley-db/index.html

3.2.3 Cross compiling Sparrow

The provided `bitbake` files can now be used to compile a program which is using SPARROW.

3.2.4 Cross compiling Spread

Download the source directory spread 4.0.0 from spread.org and untar it anywhere suitable. To install on a local linux box just go to that directory and do

```
$ ./configure
$ make
$ make install
```

There is a `sample.spread.conf` file in the `docs` directory of the source directory of spread. As long as you are using the spread daemon on *sitka* you won't need that but if you would like to run your own daemon locally you have to place that file in your `/etc/` folder and rename it to `spread.conf`. If you just use it locally you won't have to change anything in it.

To cross compile the library we need to use some special arguments

```
$ CC=~/gumstix/gumstix-oe/tmp/cross/bin/arm-angstrom-linux-gnueabi-gcc
$ RANLIB=~/gumstix/gumstix-oe/tmp/cross/bin/arm-angstrom-linux-gnueabi-ranlib
$ AR=~/gumstix/gumstix-oe/tmp/cross/bin/arm-angstrom-linux-gnueabi-ar
$ LD=~/gumstix/gumstix-oe/tmp/cross/bin/arm-angstrom-linux-gnueabi-gcc
$ ./configure \
     --host=arm-angstrom-linux-gnueabi \
     --prefix=~/gumstix/gumstix-oe/tmp/cross/include
```

```
$ make
```

Don't hit `make install` for cross compiled libraries, this will mess up your system. This will create you the libraries you need for the localcontroller to bitbake it. Bitbake will look for the files in `.../user.collection/packages/localcontroller/files/hdr/spread/`. Put all header files and all *lib files* in this folder.

Chapter 4

ATMEL

This chapter describes in a hopefully simple way how to make changes to the software running on the ATMEL ATmega128 microprocessor which is mounted on the PCB. As mentioned in the previous chapter the linux distribution used for all the steps described below is Ubuntu 8.04.1LST Hardy Heron (kernel release 2.6.24-22-generic)[1].

4.1 Program The Atmel ATmega128

A lot of useful information can be found on psychogenic.com[2], including a neat Makefile for AVR projects which can be easily adapted to almost any project. More on that later.

4.1.1 Prerequisites

To cross compile the source code written in C for the ATMEL ATmega128 microprocessor a bunch of tools need to be installed. These are standard tools and should not be hard to find with your package manager. Install the following packages:

- binutils-avr (binary utilities for files for the ATMEL's AVR architecture)
- binutils-dev
- binutils-doc
- gcc-avr (AVR cross compiler)
- avr-libc (standard C library for AVR architecture)
- avrdude (ATMEL's AVR CPU programmer)
- avrdude-doc
- simulavr (a simulator for the ATMEL AVR family of microcontrollers)

Once all these tools are installed place the AVR Project Makefile template[3] from Psychogenic[4] and all source files in any suitable folder. Before you can make your project you need to adapt the Makefile for your purposes. In the case of the MVWT2 hovercraft:

- On line 68: MCU=atmega128

[1] Two linux boxes in the MVWT lab were set up with Ubuntu: Beijing: user&pwd: `mvwt` and Sydney: user&pwd: `mvwt2008`. Use this logins to create your own account on these local machines

[2] http://electrons.psychogenic.com/modules/arms/sec/1/AVR/

[3] http://electrons.psychogenic.com/modules/arms/view.php?w=art&idx=8&page=1

[4] `http://electrons.psychogenic.com`

- On line 76: PROGRAMMER_MCU=m128
- On line 80: PROJECTNAME=MVWTpcb
- On line 87: PRJSCR=accel.c adc.c fan.c gyro.c mvwt2.c parse.c scheduler.c serial.c servo.c
- On line 114: AVRDUDE_PROGRAMMERID=stk500v2
- On line 119: AVRDUDE_PORT=/dev/ttyS0

4.1.2 Debugging and simulation

Once your code is written you need to compile and debug it. This can be a little tedious on a microchip because you cannot always debug the code on the microchip itself. Using `simulavr` we have a tool to simulate different microchip architecture on a Intel CPU and execute our code on it. The `simulavr` tool makes use of the `avr-gdb` tool which is equivalent to `gdb` for the gcc compiler. Make sure you include `-g` in your CFLAGS during compiling otherwise `gdb` will not get the needed debug information. If you are using the Makefile template from above you do not have to worry about this. First of all compile your code by using the Makefile from above. Just type `make all`. Once the code compiles successfully we can start the simulation and compile the code for debugging:

```
$ simulavr --device atmega128 --gdbserver
```

Now you should see some output messages, one says it is waiting for a gdb client. For this open another shell terminal, go to the directory where the source code and the Makefile is placed and do:

```
$ make gdbinit
$ avr-gdb -x gdbinit-MVWTpcb
```

The program will start and be halted at the first command in the `main()` function. For command explanation please see the avr-gdb man page or type help. The most used commands are `step`, `next` and `quit`.

4.1.3 Flashing the ATmega128

Once these changes are done and compilable it is pretty easy to flash the Atmel ATmega128 on the hover craft board. Connect the STK500 programmer board to any 12 V power source and connect it with a serial cable to your computer to the serial port defined above (AVRDUDE_PORT). Then take your hover craft circuit board, power it up with approximately 5 V and connect the STK500 and the circuit board with a 6 wire cable for ISP. Now the status LED and the LED on the white part on the STK500 should turn green. The next step is to flash the microchip with your compiled program using the Makefile from Psychogenic. A picture of the setup can be seen in fig. 4.1.

```
$ make all
$ make writeflash
```

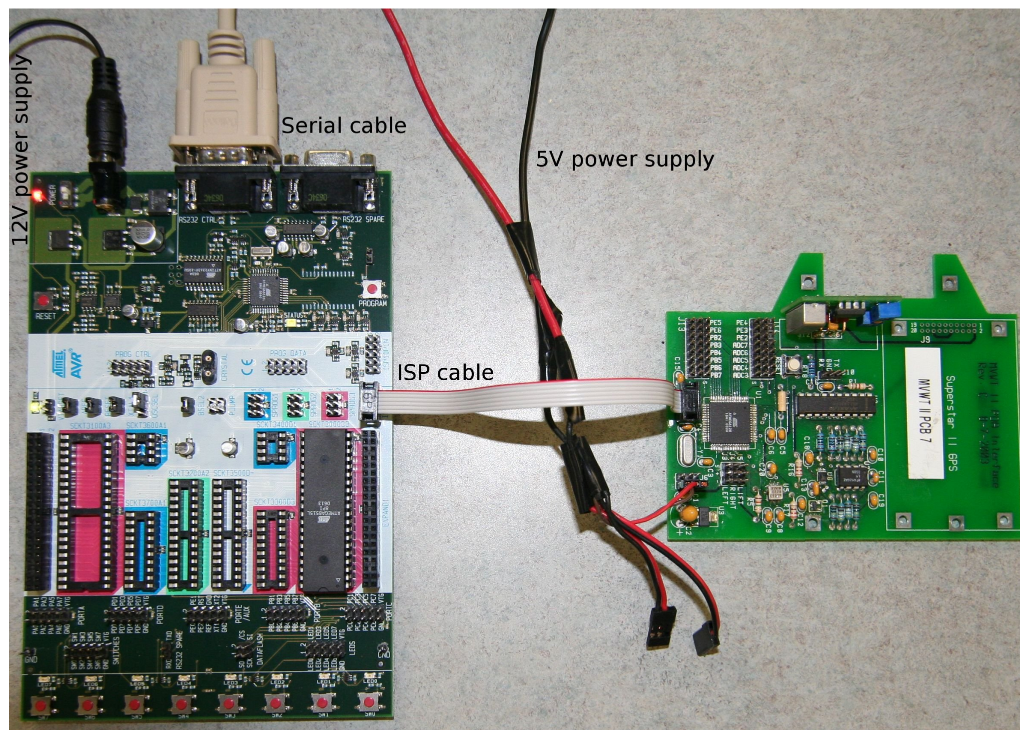


Figure 4.1: Setup and cable connections of the circuit board with the programmer

4.2 Changes

Only some minor changes were made to the source code of the ATMEL microschip but it is worth mentioning them here to be able to backtrace all changes. First of all all changes are commented in the source code so if you have a look at the source code you will see the changes. Mainly three changes in the source code were made. Firstly, the header include `avr/signal.h` was replaced with `avr/interrupt.h`. Secondly, the sending of the heading and the accelerometer data were disabled and instead the battery voltage is now read (from ADC3) and send with the header byte B. And last but not least the baud rate for the serial connection was lowered. The header replacement is due to changes in the header files. `interrupt.h` now covers all the functionality of the `signal.h` and thus it became obsolete. The data sending could be disabled because there is no need for this sensor data in our setup and it just slows down the process but if at one point someone needs this data again just uncomment the corresponding section in the `main()` function in the `mvwt2.c` source file. The baud rate settings in the code caused some very strange behavior with the board using the gumstix and it took us a long time to figure it out. The 16 MHz crystal used on the MVWT board and the used baud rate (115.2 kbps) are not a real good combination and this will cause an error of about 3.6% in the clock timing. The UBRR (UART Baud Rate Register) and the down-counter function as a baud rate generator. The down-counter runs at the system clock ($f_{osc} = 16$ MHz) and counts the UBRR value down and a clock is generated each time the value reaches zero. In our configuration this clock is additionally divided by 16. Since the UBRR value has to be a natural number there is no exact match for certain baud rates.

$$
\begin{aligned}
BAUD &= \frac{f_{ocs}}{16 * UBRR + 1} \\
&= \frac{16\ \text{MHz}}{16 * 8 + 1} = 111111.11\ \text{kbps} \approx 115200\ \text{kbps} \qquad \text{or} \\
&= \frac{16\ \text{MHz}}{16 * 12 + 1} = 76923.08\ \text{kbps} \approx 76800\ \text{kbps} \qquad \text{or} \\
&= \frac{16\ \text{MHz}}{16 * 25 + 1} = 38461.54\ \text{kbps} \approx 38400\ \text{kbps}
\end{aligned}
$$

The examples from above induce a certain error in the clock timing which should not exceed a limit which can be found in the ATMEL documentation [4](on page 187). In our case the error should be less than ± 1.5%. If we look at the errors from above we get

$$
\begin{aligned}
e_{115.2\ \text{kbps}} &\approx 3.55\% \\
e_{76.8\ \text{kbps}} &\approx 0.16\% \\
e_{38.4\ \text{kbps}} &\approx 0.16\%
\end{aligned}
$$

One would expect that we use 76.8 kbps as our new baud rate but this is not a standart baud rate and just for the sake of simplicity we are now using 38.4 kbps as our new serial baud rate which is still fast enough. In the case this baud rate is not sufficient one can either implement the 76.8 kbps baud rate or a more time-consuming solution would be to exchange the 16 MHz crystal with for example a 14.746 MHz crystal which would allow a baud rate up to 230.4 kbps.

4.3 Possible Improvements

Improvements are always possible and therefore just the ones with the highest priority for future MVWT projects are mentioned below.

4.3.1 Watchdog for battery voltage

With the new software watchdog, at least the voltage of 1 battery can be monitored. Typically, the battery of the lift fan, which is also powering the gumstix and the PCB, will be depleted first. However, using a hardware watchdog and monitoring both batteries would be a valueable improvement.

4.3.2 Replace the MVWT circuit board

The circuit board is relatively old and large and most parts of it are not used. There is a place for a GPS module which was never mounted and implemented and the sensors on board are not really useful due to noise and inaccuracy. One could replace the sensors with better but probably much more expensive ones or one could replace this circuit board with the `robostix` expansion board from gumstix. This expansion board features 6 PWM outputs and several ADC's and could also be used to read data from the gyro. Unfortunately, in order to mount this board onto the gumstix motherboard we have to remove the `consoleLCD16-vx` expansion board which is currently used for the serial connection and the LCD. This should not be a big problem since we do not really need a LCD screen and with the new expansion board the serial connection would be obsolete and the `robostix` expansion board features also a serial port (in TTL logic).

4.3.3 Replace crystal

As mentioned before the currently used crystal induced some problems with the old baud rate. One should try to replace the 16 MHz crystal with a 14.746 MHz crystal. If this works much higher baud rates could be achieved.

Chapter 5

Sparrow Drivers

In order to make the communication with the hardware as easy as possible we decided to use the SPARROW library. SPARROW needs a driver for each device and this driver has to be able to handle certain tasks. Once this driver is written for all devices the programmer does not have to care about what kind of hardware he is dealing with. He can handle every piece of hardware in the same way. Each driver has to be able to handle certain actions which will be called from SPARROW. The most important ones are `Init`, `Read`, `Write`, `HandleFlag` and `DeInit`. For a more detailed documentation on SPARROW please read [2].

5.1 Serial Driver

Interfacing the sensors and actuators on the hovercraft is done through an RS232 interface. This section describes the SPARROW device driver for this specific purpose developed during the project.

5.1.1 Reading the sensor data stream

The ATMEL board sends the raw sensor output as a stream of bytes. This data has to be converted to meaningful sensor data. The format of the raw data is a header byte, followed by 1, 2, 4 or 6 data bytes. The header byte can only be decimal 65 (character A) for the accelerometer, decimal 71 (character G) for the gyroscope, decimal 72 (character H) for the heading sensor or decimal 66 (character B) for the battery sensor. The accelerometer data is contained in 6 data bytes, the gyroscope data in 2 data bytes, the heading data in 4 data bytes and the battery data in 1 byte. The battery voltage sensor was added during this project. Eventually, the accelerometer and heading sensors were disabled.

When reading the data on the serial port, one looks for header bytes to then process the following data bytes in an adequate way. When reading the data bytes, it is important to not blindly read the next n bytes and consider them data bytes. Analysis of the data stream has shown that some data bytes get lost. If any decimal 0 is found in the data stream, the rest of the data read in this call can be discarded. There are only 0s following. Note that some of the sensor boards will not transmit any useful data, others will only read some of the sensors. You might be able to reenable a non functioning board by reflashing it.

Gyroscope data conversion

The gyroscope data is stored in 2 bytes (buffer[0 to 1]). The following formula was used to convert these bytes to actual angular velocity measured in previous MVWT projects:

$$\text{angvel} = \text{scale} \cdot (\text{buffer}[0] + 128 \cdot \text{buffer}[1]) - \text{offset}$$

This formula contains a major bug. It seems that as the angular velocity increases, buffer[0] increases. When buffer[0] hits 128, buffer[1] increases by 1 and buffer[0] continues at 0. Imagine buffer[1] and buffer[0] as one integer. As buffer[0] hits 128, the 9th bit is set to 1 (this is the first bit of buffer[1] now) and buffer[0] continues at 0.

Now the bug is caused by the fact that a uint8_t can take values from 0 to 255, not 127. Also, what the ATMEL seems to do is increasing buffer[1] every time buffer[0] hits 128 or 255. The above formula therefore results in a discontinuous mapping. To correct this, the following has to be done before using the above formula[1]:

$$\text{buffer}[0] = \begin{cases} \text{buffer}[0] & \texttt{if} \ \text{buffer}[0] \leq 127 \\ \text{buffer}[0] - 127 & \texttt{if} \ \text{buffer}[0] > 127 \end{cases}$$

Battery data conversion

The voltage divider scales the battery voltage from 0 - 8.4 V down to 0 - 1.0659 V.

$$\frac{680 \ \Omega + 1.5 \ \text{k}\Omega}{680 \ \Omega + 1.5 \ \text{k}\Omega + 15 \ \text{k}\Omega} \cdot 8.4 \ \text{V} = 1.0659 \ \text{V}$$

The ADC3 will return a value of 255 at 1.23 V (1/4 of the supply voltage of 4.93 V $\simeq$ 5 V). Dividing the ADC3 value by a factor of 26 will return the battery voltage.

$$\begin{aligned} V_{\text{batt}} &= \frac{1.23 \ \text{V}}{255} \cdot \frac{8.4 \ \text{V}}{1.0659 \ \text{V}} \cdot \text{ADC3} \\ &\simeq \frac{\text{ADC3}}{26} \end{aligned}$$

Accelerometer data conversion

The accelerometer data is stored in 6 bytes (buffer[0 to 5]). The following formula converts these bytes to the actual acceleration measured.

$$acc_x = \left(\left(\frac{\text{buffer}[2] + 128 \cdot \text{buffer}[3]}{\text{buffer}[4] + 128 \cdot \text{buffer}[5]} - 0.5 \right) \cdot 8 - \text{offset}_x \right) \cdot \text{factor}_x$$

$$acc_y = \left(\left(\frac{\text{buffer}[0] + 128 \cdot \text{buffer}[1]}{\text{buffer}[4] + 128 \cdot \text{buffer}[5]} - 0.5 \right) \cdot 8 - \text{offset}_y \right) \cdot \text{factor}_y$$

Where acc_i is the acceleration in direction i in [g]. The calibration procedure to determine is as follows: Initialize the offset to 0 and the factor to 1. Measure the acceleration at 0g (place sensor horizontal, do not move it), this is the offset (by subtracting this value the output will be 0g in horizontal position). Now measure the acceleration at 1g (place vertically, do not move). This is the factor (by dividing by this value, the output will be 1g in vertical position).

The accelerometer measurements are not useful after all. There is too little accuracy and too much noise. Therefore, the SPARROW driver does not read the accelerometer.

Note: It seems that the gyroscope conversion bug does not occur here.

[1] The driver is taking care of this now. It would also be possible to change the code of the ATMEL, now that we know how to flash it. Either way, it is working now.

Heading data conversion

Here the same problem as with the gyroscope occurs. The problem with the heading sensor is that the magnetic field in the basement is not strong enough to get any useful data. Therefore it is not described in detail here, nor does the SPARROW driver read the heading sensor.

5.1.2 The driver

Load the driver (.dev file)

The serial driver will be loaded when the following line is present in the `.dev` loaded by SPARROW:

```
device: serialdriver 5 /dev/ttyS0 -port=/dev/ttyS -read_sleep=20 -scale=1.0 -offset=360;
```

The name of the device has to be `serialdriver` (this must be listed in `devlut.c`). It has 5 channels: 1 to read the gyroscope data, 3 to send commands to the 3 fans of the hovercraft (left, right, lift in this order) and one to read the battery voltage. The hardware address is `/dev/ttyS0`, but this value is not used by the driver. So one might as well specify `0x00` or anything else here. The `port` flag is used the specify the serial port actually used by the driver. On a gumstix it will typically be `/dev/ttyS2`. The `read_sleep` flag is used to specify a time in milliseconds that the thread which is reading from the serial port will sleep for. The values specified for the standard SPARROW flags `scale` and `offset` are used when converting the raw gyroscope data to a value in degrees per second.

This is a 'silent' driver, meaning it will only print error messages, no success messages. Also refer to DOXYGEN (app. D).

Driver initialization

When the `.dev` file is parsed by SPARROW, the driver will be called to handle the flags. It will save the settings for `port` and `read_sleep`. When the diver is initialized, it will try to open the serial port specified in the `.dev` file and invoke certain settings specific for the MVWT project. It will save the previous settings, so these can be restored when the program is terminated. Finally it will create a posix thread which will read the data coming in on the serial port.

The read thread and reading from the device

The read thread reads the data received and looks for the gyroscope[2] header byte. If this is found, it will read the data bytes coming with it and calculate the actual rotational velocity and store it in a buffer. When data is read from the device, the data will be copied from the buffer to the SPARROW channel. Assuming that the sensor board sends 3 bytes for the gyroscope, 5 bytes for the heading sensor and 7 bytes for the accelerometer, the gyroscope header byte should appear at least every 15 bytes. After disabling the heading sensor and accelerometer and adding the battery voltage measurement, it should appear at least every 5 bytes. The thread will cancel looking for a gyroscope header byte after 30 bytes. Whether it read gyro data or not, the thread will then pause for `read_sleep` milliseconds, to prevent blocking other threads. The thread will exit if `read_sleep` is 0 or smaller.

[2]The same holds for the latter added battery voltage measurement.

Writing commands to the fans

When the write action is performed, the integer values stored in the 3 fan channels are written to the sensor board. The integers may take values from 0 to 255. The fans will start rotating between 55 and 60, their maximum speed is reached at about 200.

Deinitialization

When the device is deinitialized, the serial port will be closed and the original settings will be restored.

5.2 Vision Driver

This driver handels the data comming from the vision system over the ethernet and extracts the needed information from each package. The vision system was used as it was set up and no changes were made. There are certain regions on the testbed where the hover crafts cannot be seen by the vision system but since they are small a recalibration of the camera setup or changing the code itself was not considered.

5.2.1 Start the vision system

The vision system is pretty easy to use. The only thing which has to be done is to boot the windows machine (username: `Administrator`; password: `mvwt2001`) and execute the `vision_static.exe`. You should then see a window with the actual status of the vision system and the image processed should be displayed in the screen next to it. Unfortunately, the system will crash once in a while.

5.2.2 Data format and protocol

The packages sent by the vision system contain the 2D position and velocity of each vehicle[3]. Every package contains also a timestamp which gives us a relative time information when this data was gathered with respect to the time since the vision system is running. The exact data structure can be seen in *MVWTPacket.h*. The packages are sent as a UDP broadcast message and every machine listening to this specific port can read these packages.

5.2.3 The driver

Load the driver (.dev file)

The vision driver will be loaded when the following line is present in the `.dev` loaded by Sparrow:

```
device: mvwt_visiondriver 7 0x00 -HCID=7 -port=2001
        -server_ip=192.168.1.187 -broadcast=1;
```

The name of the device has to be `mvwt_visiondriver` (this must be listed in `devlut.c`). It has 7 channels (used on a hover craft): 1 for the time stamp, 3 for the 2D position (x, y, ϑ) and 3 for the velocity in the plane ($\dot{x}$, $\dot{y}$, $\dot{\vartheta}$). The hardware address is `0x00`, but this value is not used by the driver. The `HCID` flag is used to identify the hovercraft and this

[3] The maximum number of vehicles is hardcoded in *MVWTPacket.h* and is set to 16 because this is the maximum number of vehicles the vision system can handle.

number should correspond to the vision hat number and it also has to be the same in every device driver line in the `.dev` file. The `port` flag is used the specify the TCP/UDP vision port used by the vision system. The `server_ip` flag is used to define a specific vision server IP address. This address will only be used if the subsequent flag `broadcast` is set to 0 which means that a connection orientated socket will be created. The default setting for the broadcast flag is 1 and therefore no IP address is needed. This is also the recommended setting because the connection orientated socket is not properly tested.

If the vision driver should be used for the command center the `HCID` flag has to be set equal zero. This will tell the program that this machine is the command center. Keep in mind that the command center needs 97 channels for all the vision information of all vehicles. 1 for the time stamp and $16 \cdot 6 = 96$ channels for the position and the velocity.

Driver initialization

When the `.dev` file is parsed by SPARROW, the driver will be called to handle the flags. It will store the `HCID` flag, the `port` flag, the `server_ip` flag and `broadcast` flag for further processing. If one of the flags are not set the default values are used: `HCID=1 port=2001 broadcast=1`. Once all flags are handled the driver will then initialize a socket for the connection to the vision server according to the information gathered with the flags. At the moment the socket is configured as a blocking socket which means the program will wait until it receives a new vision package if the SPARROW command `chn_read()` is executed. This can cause problems if the vision system fails for a certain amount of time.

Reading

This task is performed if the SPARROW command `chn_read()` is executed. The driver will then look for a new vision package and extract the data from it. If the machine running the program is a hover craft only the hover craft specific information are collected otherwise the whole struct is extracted and stored in the SPARROW channel table.

Deinitialization

This task is performed if the SPARROW command `chn_close()` is executed. The driver will simply close the vision socket.

Possible Improvements

Currently the vision socket is initialized as a blocking socket which means that the program will wait until it gets a new vision package once the receive command is executed. This is not a problem as long the vision system runs fine but if it goes down for a certain amount of time there is no way to let the hover craft react on that. One could either implement a non blocking socket[4] which increases the amount of dropped packages or one could do this by using threads which would allow to detect such a failure.

5.3 Command Driver

This driver will handle the communication between the hover crafts and the command station. All commands are sent over the wireless LAN to the hover crafts. The fundament of this driver is based on a SPREAD server which will handle all messages comming in and provide it to the expected recipient. For a detailed documentation about SPREAD please

[4]This is already implemented but currently deactivated.

read [3].

5.3.1 Data format and protocol

The packages sent over the command driver contain a command type information (integer) such as velocity control, position control or shut down and then for each vehicle[5] three control values (floats) which can be used depending on the command type. The exact data structure can be seen in *MVWTPacket.h*. The packages are sent to a central SPREAD server to which each hover craft and the command station can connect. Depending on their group identification they can communicate with each other or not.

5.3.2 The driver

Load the driver (.dev file)

The command driver will be loaded when the following line is present in the `.dev` loaded by SPARROW:

```
device: mvwt_commanddriver 4 0x00 -HCID=7 -spread=4803@sitka -group_id=1;
```

The name of the device has to be `mvwt_commanddriver` (this must be listed in `devlut.c`). It has 4 channels (used on a hover craft): 1 for the command type, 3 for command itself (cmd_x, cmd_y, cmd_ϑ). The hardware address is `0x00`, but this value is not used by the driver. The `HCID` flag is used to identify the hovercraft and this number should correspond to the vision hat number and it also has to be the same in every device driver line in the `.dev` file. The `spread` flag is used the specify the SPREAD server. The `group_ip` flag is used to define a spread group which the hover craft should join.

If the command driver is used for the command center the `HCID` flag has to be set equal zero. This will tell the program that this machine is the command center. Keep in mind that the command center needs 49 channels for all the command information for all vehicles. 1 for the command type and $16 \cdot 3 = 48$ channels for the actual commands.

Driver initialization

When the `.dev` file is parsed by SPARROW, the driver will be called to handle the flags. It will store the `HCID` flag, the `spread` flag and the `group_id` flag for further processing. If one of the flags is not set the default values are used: `HCID=1 spread=4803@sitka group_id=1`. Once all flags are handled the driver will then initialize a connection to the defined SPREAD server according to the information gathered with the flags. A unique private_name will be generated for each connected device and the group_id is used to join the right group and connects its SPREAD mailbox to it.

Reading

This task is performed if the SPARROW command `chn_read()` is executed. The driver will then look if new messages for his group have arrived on the SPREAD server and collect them. At the moment this command is a blocking routine. If no new command is sent to the SPREAD server the program will wait until it gets a new command[6]. The driver will extract the commands for the hover craft it is running on. This driver is designed as a one

[5] The maximum number of vehicles is hardcoded in *MVWTPacket.h* and is set to 16 because this is the maximum number of vehicles the vision system can handle.

[6] This issue should be eliminated in further development

way communication. The command center sends commands to the individual hover crafts but not vice versa. This means that there is no read function for the command station and no write routine for the hover crafts[7].

Writing

This task is performed if the SPARROW command `chn_write()` is executed. The driver will then copy all commands stored in the SPARROW channel table into the command struct[8] and sends it to the SPREAD group it is connected to. As mentioned above the write function is only implemented for the command center.

Deinitialization

This task is performed if the SPARROW command `chn_close()` is executed. The driver will simply close the connection and disconnect from the SPREAD server.

Possible improvements

As mentioned above this driver is blocking due the blocking characteristics of the underlying SPREAD function. To avoid this one need to use threads and this would also be the way to implement commands being sent back from the hover craft to the command center. One could also consider to use SKYNET which would give the user some other nice features on top of the SPREAD server. Unfortunately, SKYNET is poorly documented and there is also some inconsistency on the wiki page[9]. I would recommend to stick to the DOXYGEN page[10] which is probably the most recent resource.

5.4 Trajectory Driver

While FALCON was intended to sit on top of SPARROW, both are really independent of each other. For this device driver, SPARROW uses FALCON to read trajectory files. This device should be initialized with as many channels as there are data columns in the `.trj` file, typically 49 (1 time column, 16 x 3 data channels) For further information please refer to the documentation of FALCON [1].

[7] A two way communication can easily be added to this driver
[8] Defined in `MVWTPacket.h`
[9] http://gc.caltech.edu/wiki/index.php/Skynet
[10] http://gc.caltech.edu/yam/DGCdocs/modules/skynet/html/

Chapter 6

Model

6.1 The model

6.1.1 Equations of motion

These equations (cp. (6.1)) were derived and documented in previous projects but just for completeness we state them here again. Figure 6.1 shows a sketch of the model.

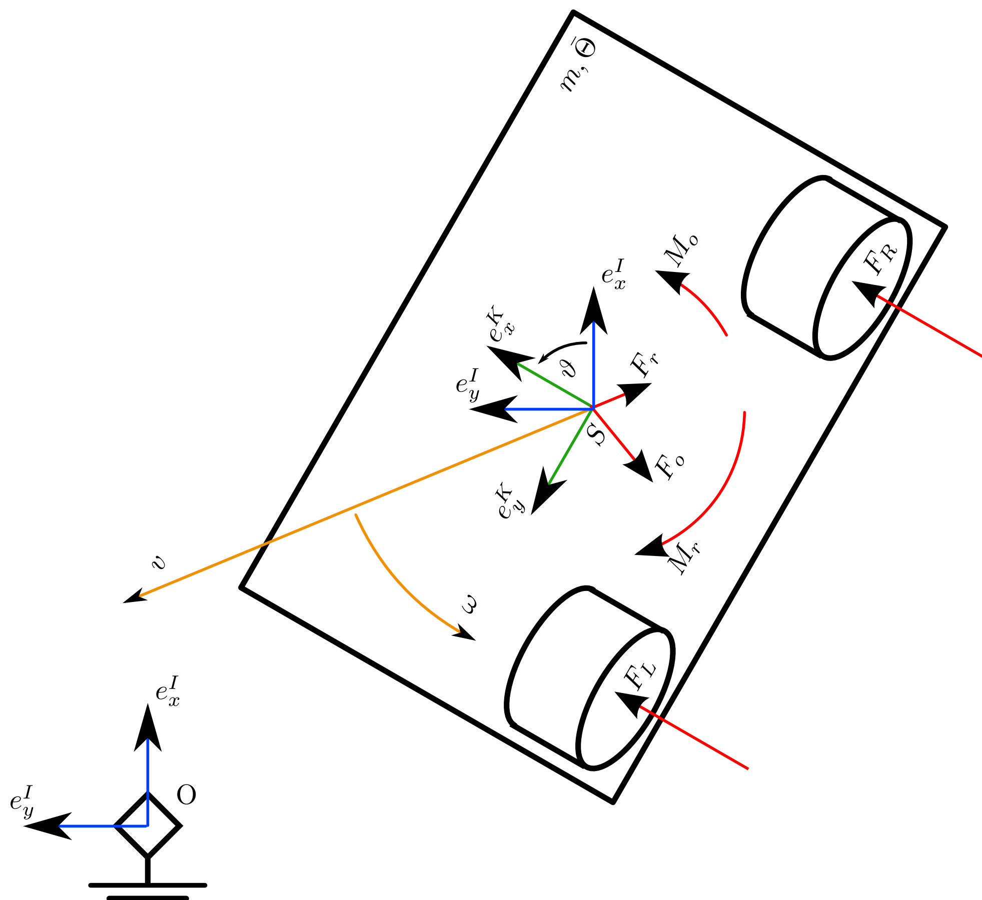


Figure 6.1: Coordinate frames, forces and vectors of the hover craft in a plane

Using the Lagrange II method

$$\frac{d}{dt}\left(\frac{\partial T}{\partial \dot{\vec{q}}}\right)^T - \left(\frac{\partial T}{\partial \vec{q}}\right)^T - \left(\frac{\partial V}{\partial \vec{q}}\right)^T = \vec{f}_{NP}$$

We need to derive the total kinetic energy T for the system and its total potential energy V. Since we use a 2D model V will be zero anyway and can be neglected. $\vec{f}_{NP}$ stands for all non potential forces like external forces (eg. fan thrust) or momentums (eg. momentum induced by friction). Our minimal coordinates are

$$\vec{q} = \begin{pmatrix} x \\ y \\ \vartheta \end{pmatrix}$$

To make sure everyone speaks of the same thing I will shortly explain my notation. There are two simple cases to consider a vector $\vec{r}$ and a rotational velocity $\vec{\omega}$. The notation of these two will look something like this

$$_A\vec{r}_{PQ} \qquad \text{and} \qquad _I\vec{\omega}_{KI}$$

The A and the I on the left hand side stand for the coordinate frame the term is expressed in and the KI on the right hand side of the ω stands for the rotation of the K frame with respect to the I frame. Which means that $_I\vec{\omega}_{KI}$ expresses the rotational velocity of the K frame with respect to the I frame expressed in the I frame.

Back to the derivation of the equation of motions. The kinetic energy of a body in the space is

$$T = \frac{1}{2} m \cdot {}_A\vec{v}_p^T \cdot {}_A\vec{v}_p + m \cdot {}_B\vec{v}_p^T \left({}_B\vec{\Omega} \times {}_B\vec{r}_{PS} \right) + \frac{1}{2} {}_C\vec{\Omega}^T \cdot {}_C\bar{\Theta}_P \cdot {}_C\vec{\Omega}$$

and since we use $P \equiv S$ the middle term will vanish.

$$\begin{aligned} {}_I\vec{r}_{OS} &= \begin{pmatrix} x \\ y \\ 0 \end{pmatrix} \\ {}_I\vec{v}_S &= \begin{pmatrix} \dot{x} \\ \dot{y} \\ 0 \end{pmatrix} \\ \vec{\Omega} = {}_I\vec{\omega}_{IK} &= \begin{pmatrix} 0 \\ 0 \\ \dot{\vartheta} \end{pmatrix} \end{aligned}$$

It is not yet known what $\bar{\Theta}$ exactly will look like but we do know its structure which will simplify the equation.

$$\bar{\Theta} = \begin{pmatrix} * & * & * \\ * & * & * \\ * & * & \Theta_z \end{pmatrix}$$

The values marked by asterisks are not of interested because they will be canceled out by

the zeros of $\vec{\Omega}$. Putting together all information from above we get

$$\begin{aligned}
T &= \frac{1}{2}m\left(\dot{x}^2+\dot{y}^2\right)+\frac{1}{2}\Theta_z\dot{\vartheta} \\
\left(\frac{\partial T}{\partial \vec{q}}\right)^T &= \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \\
\left(\frac{\partial T}{\partial \dot{\vec{q}}}\right)^T &= \begin{pmatrix} m\dot{x} \\ m\dot{y} \\ \Theta_z\dot{\vartheta} \end{pmatrix} \\
\frac{d}{dt}\left(\frac{\partial T}{\partial \dot{\vec{q}}}\right)^T &= \begin{pmatrix} m\ddot{x} \\ m\ddot{y} \\ \Theta_z\ddot{\vartheta} \end{pmatrix}
\end{aligned}$$

Now only the $\vec{f}_{NP}$ are missing. We have two forces F_L and F_R from the fans (left and right), a momentum M_F caused by the two fans, a friction force F_r in the direction of the velocity of the hover craft and a momentum M_r induced by the rotation of the hover craft due to friction. We can also consider the fact that a slightly unbalanced hover craft will tend to drift and turn which will add another momentum M_o and force F_o. If we do not need this degree of accuracy we just let M_o and F_o be equal to zero. We will therefore have

$$\vec{f}_{NP} = {}_K\bar{J}_S^T\cdot\left({}_K\vec{F}_L+{}_K\vec{F}_R+{}_K\vec{F}_r+{}_K\vec{F}_o\right)+{}_K\bar{J}_S^T\cdot\left({}_K\vec{M}_F+{}_K\vec{M}_r+{}_K\vec{M}_o\right)$$

$$ {}_K\vec{F}_L = \begin{pmatrix} F_L \\ 0 \\ 0 \end{pmatrix}, \qquad {}_K\vec{M}_F = \begin{pmatrix} 0 \\ 0 \\ r_{\text{Fan}}(F_R-F_L) \end{pmatrix}$$

$$ {}_K\vec{F}_R = \begin{pmatrix} F_R \\ 0 \\ 0 \end{pmatrix}, \qquad {}_K\vec{M}_r = \begin{pmatrix} 0 \\ 0 \\ -\mu_r\dot{\vartheta} \end{pmatrix}$$

$$ {}_K\vec{F}_o = \begin{pmatrix} a \\ b \\ 0 \end{pmatrix}, \qquad {}_K\vec{M}_o = \begin{pmatrix} 0 \\ 0 \\ c \end{pmatrix}$$

$$ {}_K\vec{F}_r = \begin{pmatrix} -\mu_t(\cos\vartheta\cdot\dot{x}+\sin\vartheta\cdot\dot{y}) \\ -\mu_t(\cos\vartheta\cdot\dot{y}-\sin\vartheta\cdot\dot{x}) \\ 0 \end{pmatrix}$$

where the $\bar{J}$-terms are the Jacobian to transform the forces from the body frame into the

minimal coordinate frame:

$$
\begin{aligned}
{}_K\vec{v}_S &= A_{KI} \cdot {}_I\vec{v}_S = \begin{pmatrix} \dot{x}\cos\vartheta + \dot{y}\sin\vartheta \\ \dot{y}\cos\vartheta - \dot{x}\sin\vartheta \\ 0 \end{pmatrix} = {}_K\bar{J}_S \cdot \begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{\vartheta} \end{pmatrix} \\
{}_K\vec{\Omega} &= \begin{pmatrix} 0 \\ 0 \\ \dot{\vartheta} \end{pmatrix} = {}_K\bar{J}_R \cdot \begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{\vartheta} \end{pmatrix} \\
{}_K\bar{J}_S &= \begin{pmatrix} \cos\vartheta & \sin\vartheta & 0 \\ -\sin\vartheta & \cos\vartheta & 0 \\ 0 & 0 & 0 \end{pmatrix} \\
{}_K\bar{J}_R &= \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}
\end{aligned}
$$

where A_{KI} is the transformation matrix from the K body frame to the inertial frame I. Setting everything together gives us

$$
\begin{pmatrix} m\ddot{x} \\ m\ddot{y} \\ \Theta_z\ddot{\vartheta} \end{pmatrix} = \begin{pmatrix} -\mu_t\dot{x} + \cos\vartheta\,(F_R + F_L + a) + \sin\vartheta\,(-b) \\ -\mu_t\dot{y} + \sin\vartheta\,(F_R + F_L + a) + \cos\vartheta\,(b) \\ -\mu_r\dot{\vartheta} + c + r_{\text{Fan}} \cdot (F_R - F_L) \end{pmatrix} \tag{6.1}
$$

where r_{Fan} is the radial distance from the hover craft center to the fan and a, b and c are the offset forces in positive x direction, positive y direction and the offset momentum in positive direction.

6.1.2 Discussion

These equations of motion are rather obvious and the only difference between the other derivations in previous documents are the offset forces ${}_K\vec{F}_o$ and ${}_K\vec{M}_o$. The idea of these were that we could add them in the simulation to see if it would be helpful to have a startup sequence where we determine the drift and then add it to the model.

6.2 Parameter identification

There are 4 parameters in the system and they were measured, calculated or identified as follows: The weight $m = 659$ g and inertia $\Theta_z = 0.0021\,\mathrm{kg\,m^2}$ of the hovercraft, and the translational $\mu_t = 0.05\,\mathrm{Ns/m}$ and rotational $\mu_r = 0.0013\,\mathrm{Nms}$ friction.

In the directory `../matlab/measfric/` of the svn repository the coresponding files can be found.

6.2.1 Weight

The weight of the hovercraft with the skirt is 659 g (the version without a skirt weights 685 g, we used the version with the skirt). This includes all batteries and the vision system hat. The weight of some of the components are: Batteries 82 g each, thrust fan incl. holding 50 g each, gumstix incl. mounting box 90 g, vision system hat 50 g, lift fan 80 g, base plate and lift fan mounting plate 66 g.

6.2.2 Inertia

The inertia was not measured, but rather calculated. The attempt to measure it using the turntable failed, because of the high ratio of the inertia of the turntable to the inertia of the hovercraft.

A rough approach to estimate the inertia is to treat the hovercraft as a cylinder with a radius $r = 8$ cm and mass $m = 659$ g. The inertia becomes

$$\Theta_z = \frac{m \cdot r^2}{2} = \frac{0.659 \text{ kg} \cdot 0.08 \text{ m}^2}{2} = 0.0021\,\mathrm{kg\,m^2}$$

component	weight	radius/size	quantity
batteries	82 g	4 cm	2
lift fans	50 g	8 cm	2
gumstix	90 g	6 cm	1
lift fan	80 g	1.5 cm	1
vision head	50 g	22.6 cm by 32.7 cm	1
base	66 g	7 cm	1

Table 6.1: Components and parameters for inertia calculation.

A more educated approach is to consider single components as point masses and add up their individual contribution to the inertia. The head is considered as a box, the base

plate and the lift fan as a cylinder. The inertia becomes

$$\begin{aligned}
\Theta_z &= 82\ \mathrm{g} \cdot 4\ \mathrm{cm}^2 \cdot 2 \\
&\quad +50\ \mathrm{g} \cdot 8\ \mathrm{cm}^2 \cdot 2 \\
&\quad +90\ \mathrm{g} \cdot 6\ \mathrm{cm}^2 \\
&\quad +80\ \mathrm{g} \cdot 1.5\ \mathrm{cm}^2 \cdot \frac{1}{2} \\
&\quad +50\ \mathrm{g} \cdot ((22.6\ \mathrm{cm})^2 + (32.7\ \mathrm{cm})^2) \cdot \frac{1}{12} \\
&\quad +66\ \mathrm{g} \cdot 7\ \mathrm{cm}^2 \cdot \frac{1}{2} \\
&= 0.0021\,\mathrm{kg\,m}^2
\end{aligned}$$

6.2.3 Translational friction

To identify the translational friction parameter μ_t, 4 runs (fig. 6.2) of the hovercraft were measured. In every run, the lift fan was turned to 180 and the hovercraft was accelerated manually, so the only force acting on the hovercraft was friction. This was preferred over using the thrust fans to accelerate the hovercraft, because the force produced by the fans is uncertain. The rotation was neglected, the hovercraft was going more or less straight and effort was taken to balance it properly.

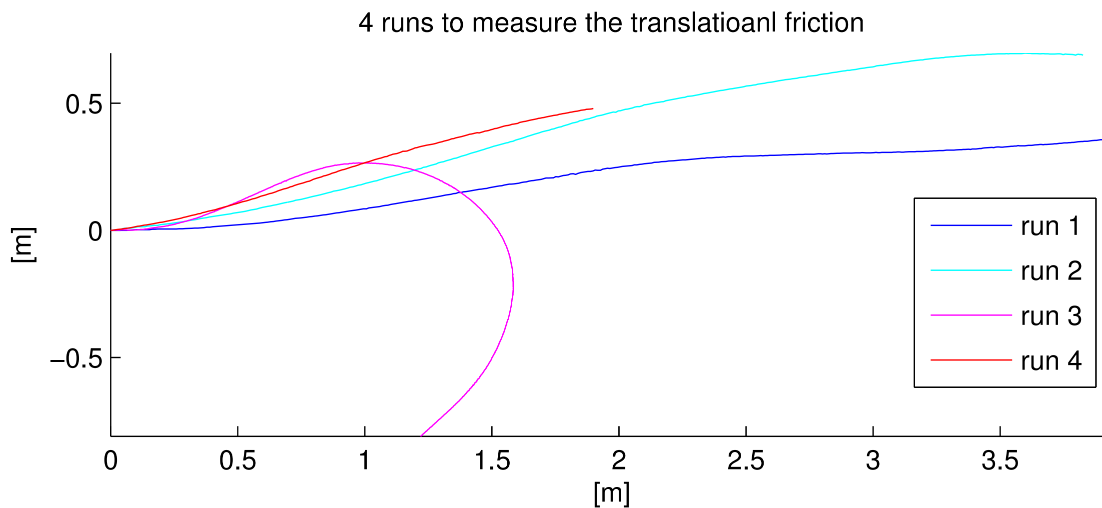


Figure 6.2: The 4 runs that were used to determine the translational friction.

Just looking at the distance the hovercraft traveled, that is combining x and y to the distance σ, the dynamics can be described as (no force, since fans are turned off):

$$m \cdot \frac{\mathrm{d}^2}{\mathrm{d}t^2}\sigma = -\mu_t \cdot \frac{\mathrm{d}}{\mathrm{d}t}\sigma$$

The solution with initial position $\sigma(0) = 0$ and initial velocity $\dot{\sigma} = v_0$ is

$$\sigma(t) = \frac{m \cdot v_0}{\mu_t} \cdot \left(1 - e^{-\frac{\mu_t}{m} \cdot t}\right)$$

$$\dot{\sigma}(t) = v_0 \cdot e^{-\frac{\mu_t}{m} \cdot t}$$

The transfer function is (if there is an input force u)

$$\frac{\Sigma}{U} = \frac{1}{s} \cdot \frac{\mu_t}{\frac{m}{\mu_t} \cdot s + 1}$$

Both the gain and the pole depend on μ_t, so they can not be fitted to the measurements independently.

The results can be seen in fig. 6.3. The choice of $\mu_t = 0.05\,\mathrm{Ns/m}$ yields good consistency of measurement and model for all 4 measured runs. It has to be said that only a limited range of velocities were covered in the measurements and the gain (how far the hovercrafts travel before coming to a rest) cannot be measured.There are two main reasons for this, first of all for high initial velocities the testbed is not large enough and secondly the hovercraft will never really come to a rest. For control this might not matter since one can simply turn off the lift fan to stop any time.

6.2.4 Rotational friction

To identify the rotational friction parameter μ_r, 4 runs of the hovercraft were measured. In every run, the lift fan was turned to 180 and an initial rotational velocity was invoked manually, so the only moment acting on the hovercraft was friction. This was preferred over using the thrust fans to rotate the hovercraft, because the force produced by the fans is uncertain. The translational movement was neglected, the hovercraft was more or less staying in place and effort was taken to balance it properly.

Just looking at the rotation of the hovercraft, the dynamics can be described as (no momentum, since fans are turned off):

$$\Theta_z \cdot \frac{\mathrm{d}^2}{\mathrm{dt}^2}\vartheta = -\mu_r \cdot \frac{\mathrm{d}}{\mathrm{dt}}\vartheta$$

The solution with initial angle $\vartheta(0) = 0$ and initial angular velocity $\dot{\vartheta} = \alpha_0$ is

$$\sigma(t) = \frac{\Theta_z \cdot \alpha_0}{\mu_r} \cdot \left(1 - e^{-\frac{\mu_r}{\Theta_z} \cdot t}\right)$$

$$\dot{\sigma}(t) = \alpha_0 \cdot e^{-\frac{\mu_r}{\Theta_z} \cdot t}$$

The transfer function is (if there is an input force u, where T is the Laplace transform of ϑ and Θ_z is the inertia)

$$\frac{T}{U} = \frac{1}{s} \cdot \frac{\mu_r}{\frac{\Theta_z}{\mu_r} \cdot s + 1}$$

Both the gain and the pole depend on μ_r, so they can not be fitted to the measurements independently.

Fitting the time constant to the measurements does not yield good results as can be seen in fig. 6.4. Fitting the gain gives in slightly better results. Note that for each run μ_r was calculated separately, the results are 0.0013 Nms, 0.0013 Nms, 0.0018 Nms, 0.0007 Nms and the average is 0.0013 Nms.

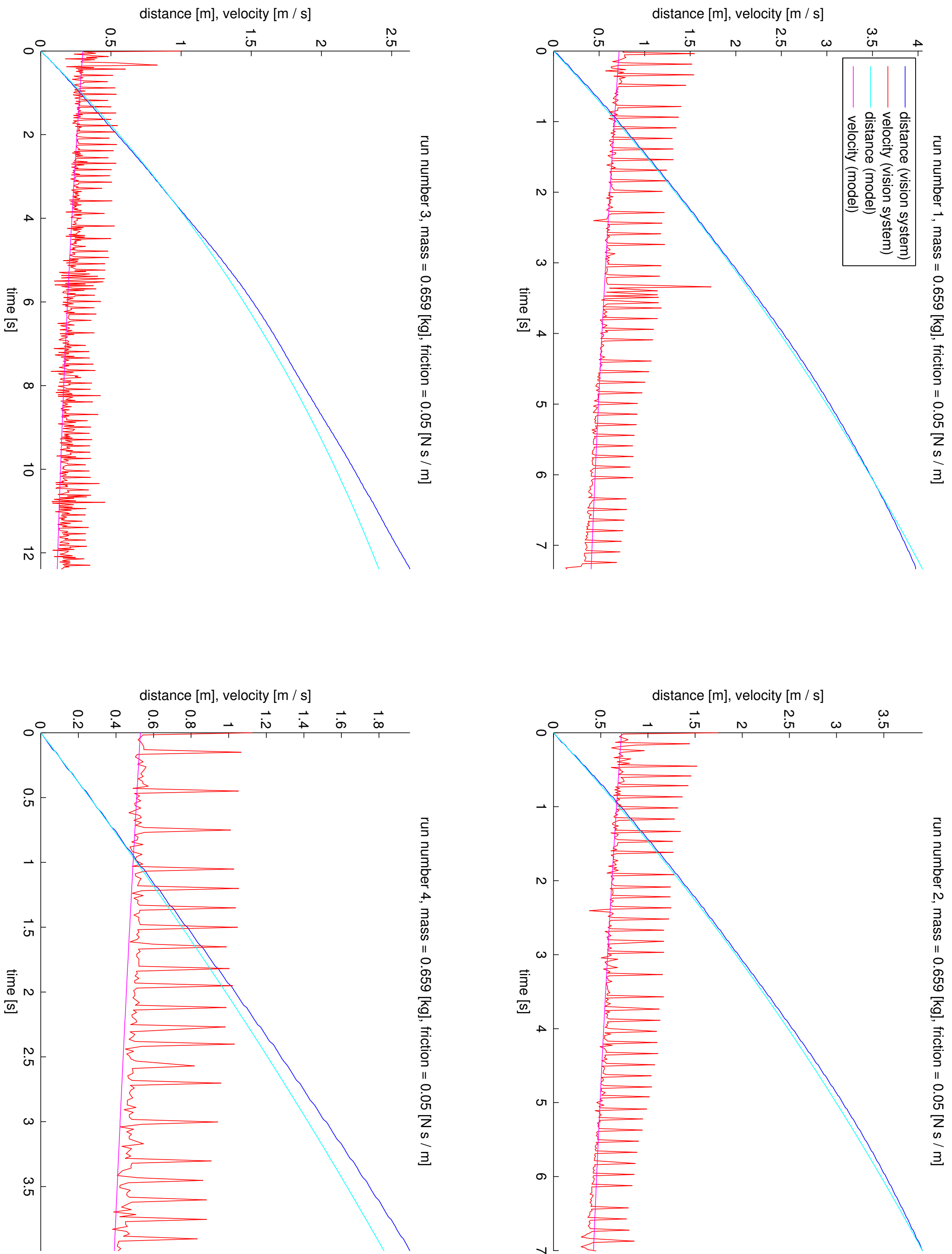


Figure 6.3: Identification of the translational friction parameter: Comparison between measurements and model. The spikes are produced by the vision system (reason unknown). All plots with the same friction parameter.

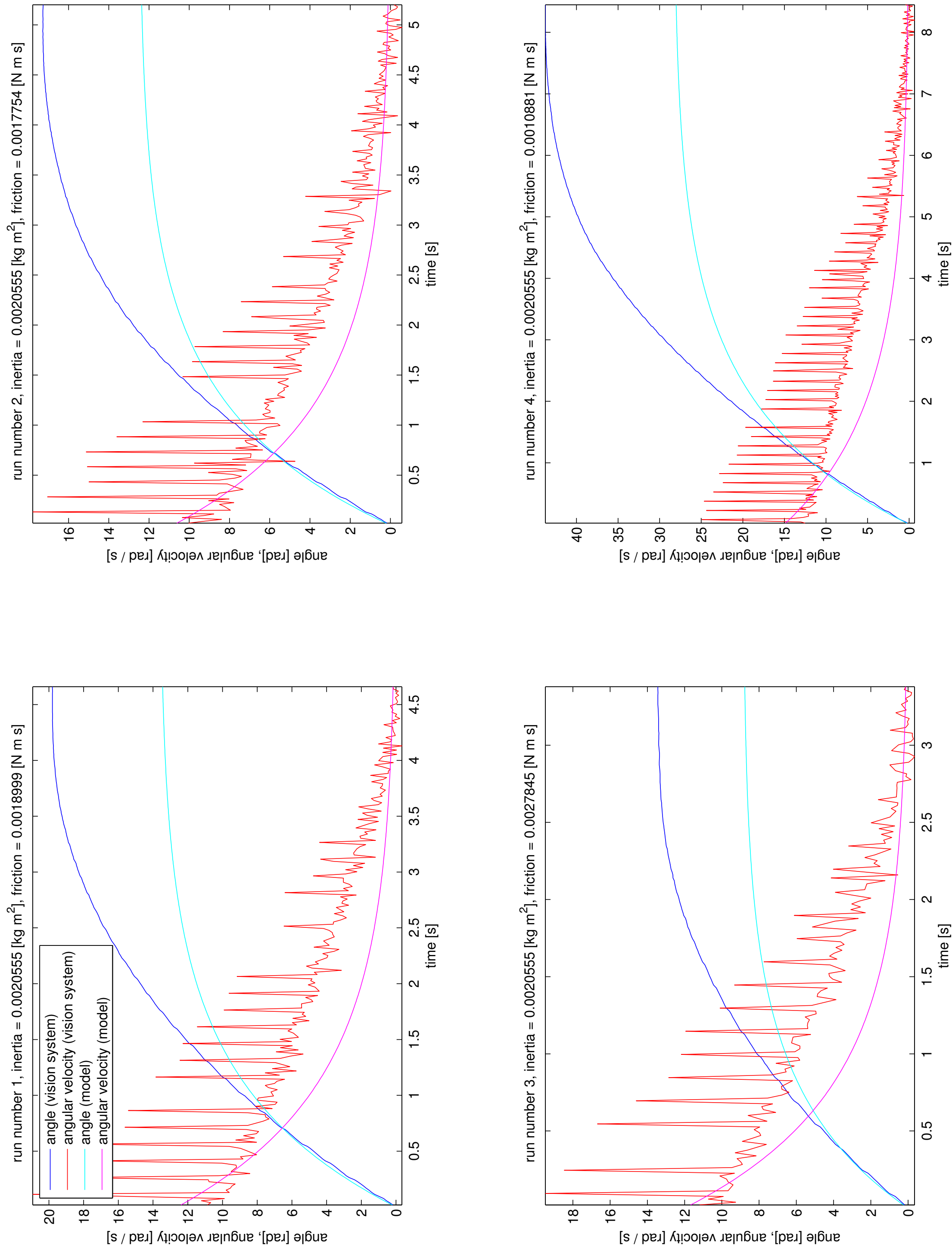


Figure 6.4: Identification of the rotational friction parameter: Comparison between measurements and model. The spikes are produced by the vision system (reason unknown). The friction parameter was chosen such that the time constant of the system matches the measurement, resulting in a mismatch with the gain. The parameter was calculated separately for each measurement.

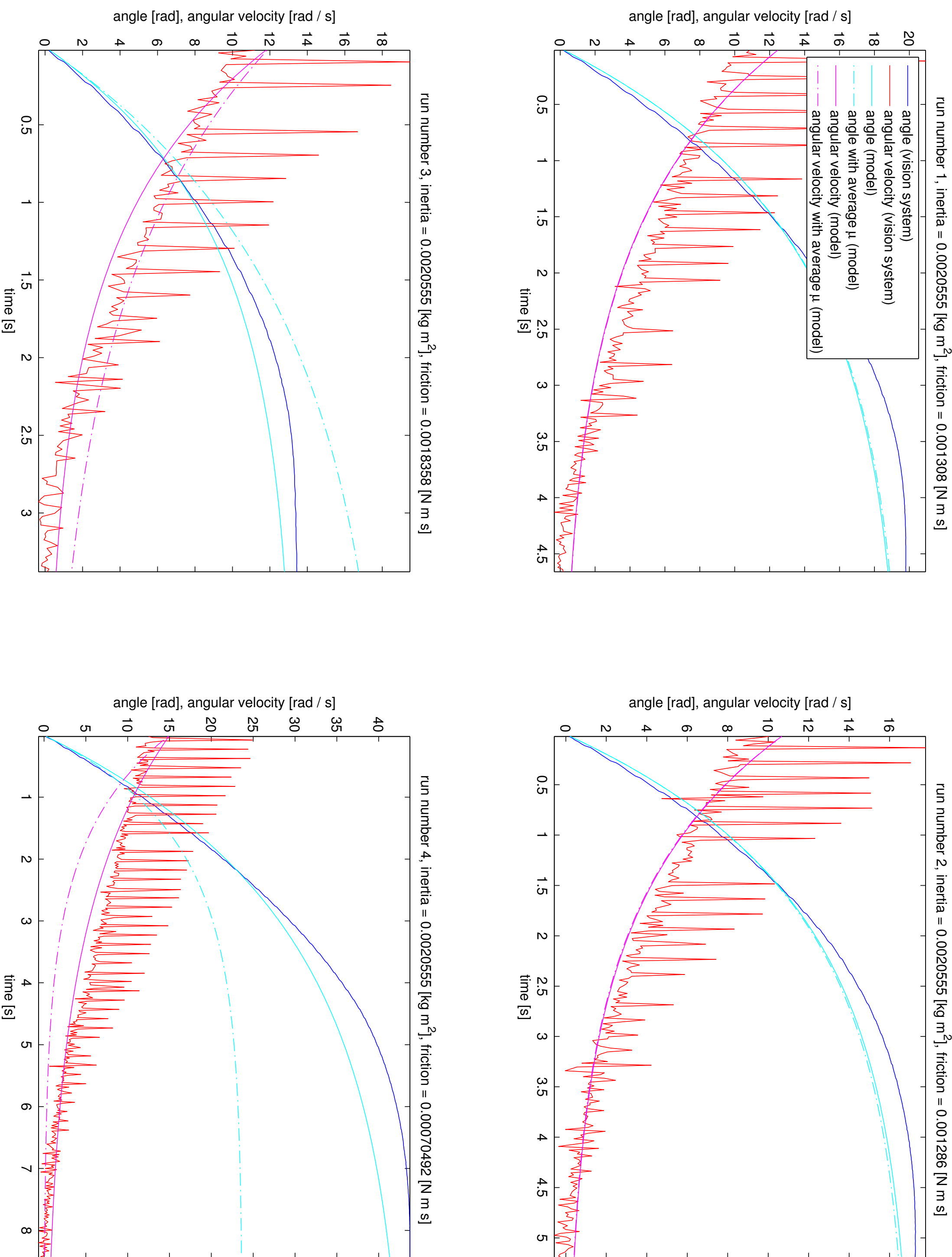


Figure 6.5: Identification of the rotational friction parameter: Comparison between measurements and model. The spikes are produced by the vision system (reason unknown). The friction parameter was chosen such that the gain of the system matches the measurement, resulting in a mismatch with the time constant. The parameter was calculated separately for each measurement. When using the average friction, the model does not fit for runs 3 and 4.

Chapter 7

Hovercraft Local Controller

7.1 Devices

The local controller uses the devices introduced in the previous chapter. The serial driver is used to communicate with the PCB in order to read the gyroscope measurements and control the fans. The vision driver is used to get the hovercrafts position as seen by the vision system. The command driver receives the commands send by the command station. All these devices are specified in the `.dev` file, here is what it might look like for the hovercraft ID 3.

```
device: serialdriver 4 /dev/ttyS2 -port=/dev/ttyS2 -read_sleep=20 -scale=1.0 -offset=360;
device: mvwt_visiondriver 7 0x00 -HCID=3 -port=2001 -server_ip=192.168.1.187 -broadcast=1;
device: mvwt_commanddriver 4 0x00 -HCID=3 -spread=4803@sitka -group_id=1;
```

7.2 Binary

The binary `localcontrol` has two modes: One with the dynamic display enabled (intended for manual control and debugging) and one without the display (faster). To start the display, simply run `localcontrol`. To disable the display, run `localcontrol nodisplay`. If you run `localcontrol help` or pass any other arguments, a help message will appear.

7.2.1 Display disabled

If the local controller is run without the display, it will right away start the control algorithm. To manually terminate the program, just hit enter.

	serial driver		vision driver		command driver	
c	gyroscope	[°/s]	time[a]	[s]	command type	[integer]
h	left fan	[integer]	x position	[m]	value 1	[double]
a	right fan	[integer]	y position	[m]	value 2	[double]
n	lift fan	[integer]	ϑ orientation	[rad]	value 3	[double]
n			x velocity	[m/s]		
e			y velocity	[m/s]		
l			angular velocity	[rad/s]		

[a] Time elapsed since start of the vision system.

Table 7.1: Channels of the drivers used by the local controller of the hovercraft.

7.2.2 The dynamic display

If the dynamic display is enabled (default) the following will appear on the screen (see fig. 7.1). Sensor values will be show in blue (light blue if they are editable). Buttons will appear in green. The following keys are bound: `e` will perform `chn_write()`, `r` will perform `chn_read()`, `w,a,s,d` are used to manually control the hovercraft, `space` will toggle the lift fan, `c` will start the controller, `l` will start a thread that performs `chn_read()` every second. Pressing `q` will exit the program (standard SPARROW binding). Please refer to the SPARROW documentation [2] for further information on the dynamic display.

```
LOCAL HOVERCRAFT CONTROLLER

CMD     type  value1  value2  value3             GYRO   0.00     [deg/sec]
        000   +0.00   +0.00   +0.00              BATT   0.00     [V]

VISION ID    x      y      th     dx/dt  dy/dt  dth/dt  time      dropped
       12    0.000  0.000  +0.000 +0.00  +0.00  +0.00   0.00        0
             [m]    [m]    [rad]  [m/s]  [m/s]  [rad/s] [s]
     Note: ID range 0 to 15

FANS    left   lift   right           space: toggle lift fan
          0      0      0             w,a,s,d: manual control
        e: invoke                     r: read channels  l: thread

CONTROL ref = +0.000   u = 000        rot     c: turn on controller
        err = +0.000   u = 000        trans
        x_r = +4.770 y_r = +5.000    reference position



CONTROL LOOP execution time = 0.000      period time = 0.000

[READ]                                                            [QUIT]
```

Figure 7.1: The dynamic display of the local controller. All numerical values are in blue. Light blue values can be changed manually. Buttons are green.

7.3 The really really simple PID controller

This controller will move the hovercraft to a desired target location. It consists of two parts: A PID controller that controls the angle, and a second controller that controls the translational velocity of the hovercraft.

The rotational controller is a PID controller that uses only the angle measured by the vision system as an input. The desired angle is the angle that is pointing from the hovercrafts current position to the desired target location. It is running at 10 Hz and has some 'extra features' which make it rather fuzzy: The output is limited to 80 mN (the I-part will not wind up). The controller will determine whether it has to turn left or right, if the direction changes from one iteration to the next, the controller is reset. Finally, if the hovercraft is looking in the roughly right direction, that is as the angle enters an interval of a desired orientation $\pm 30°$, the lift fan will be shut off to stop the hovercraft from rotating. The translational controller will simply turn both fans on to 80 mN, as

long the hovercraft is looking in this roughly right direction. Finally, the controller will stop the hovercraft once it gets closer than 20 cm to the target location.

This controller is ugly, but it works.

Figure 7.3 shows run 6 in detail. Denoting time on the z-axis, one can see what the controller is doing. In the beginning, the hovercraft is looking in the wrong direction, so the controller starts rotating it which takes about 4 seconds. Then, the translational controller starts moving it towards the desired location. After 8 and 10 seconds, the hovercraft slows down, because first it was no longer looking in the right direction and the translational controller stops the hovercraft, and second the liftfan is turned off as the rotational controller turns the hovercraft back in the right direction. Finally, the hovercraft get close to the target location and shuts down.

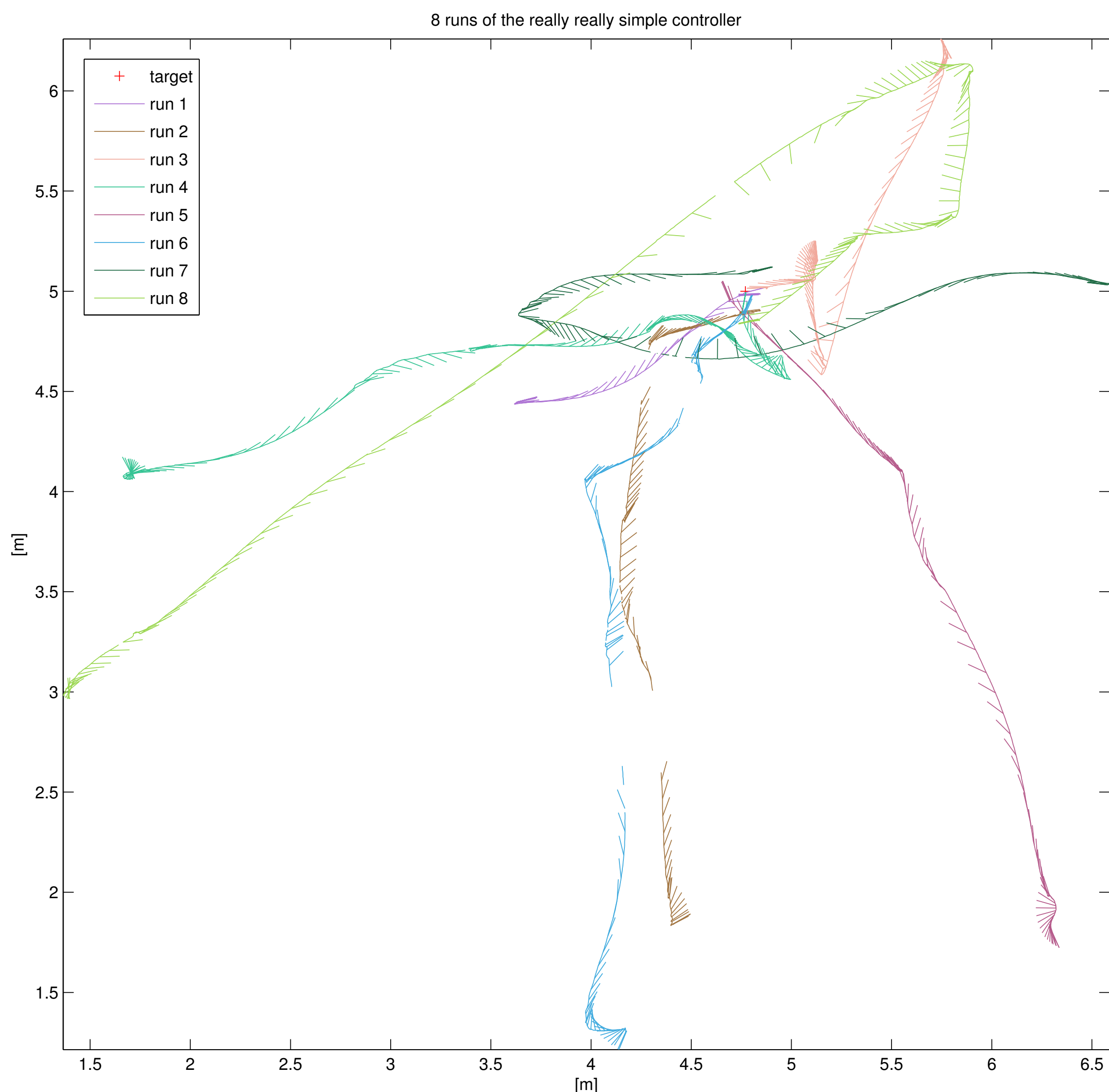


Figure 7.2: Eight runs of the really simple controller. The hovercraft was moved manually to an arbitrary starting location and then 'released'.

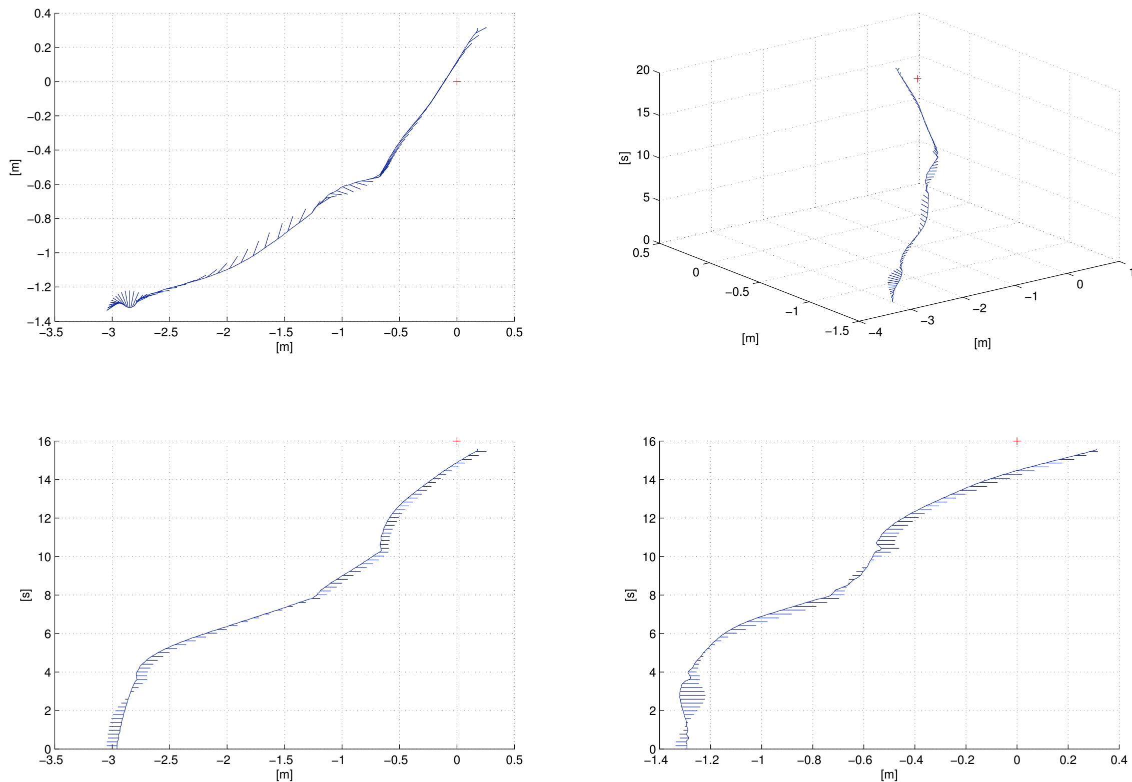


Figure 7.3: Run number 6. Time on the z-axis.

7.4 Another controller approach

Time was short so this controller got never really out of the simulation status and only a few tests were performed with the actual hover craft. The idea was to design an H_∞ controller using the S/KS/T schema. As a start only the orientation Θ should be controlled. To prevent oscillations a PI part was appended to the plant model for the controller design which is obviously not present in the real plant. A step response for the orientation of 1 rad can be seen in fig. 7.5. The controller needs final tuning and maybe one should consider to account for the delay in the system as well. The controller is now configured to act very slow due to testing reasons.

For the controller derivation a linear approximation of the rotational dynamics was used. The weightings of the H_∞ controller make sure that the cross over frequency is around 3 rad per second and the sensitivity and complementary sensitivity stay below 3 dB at all frequencies.

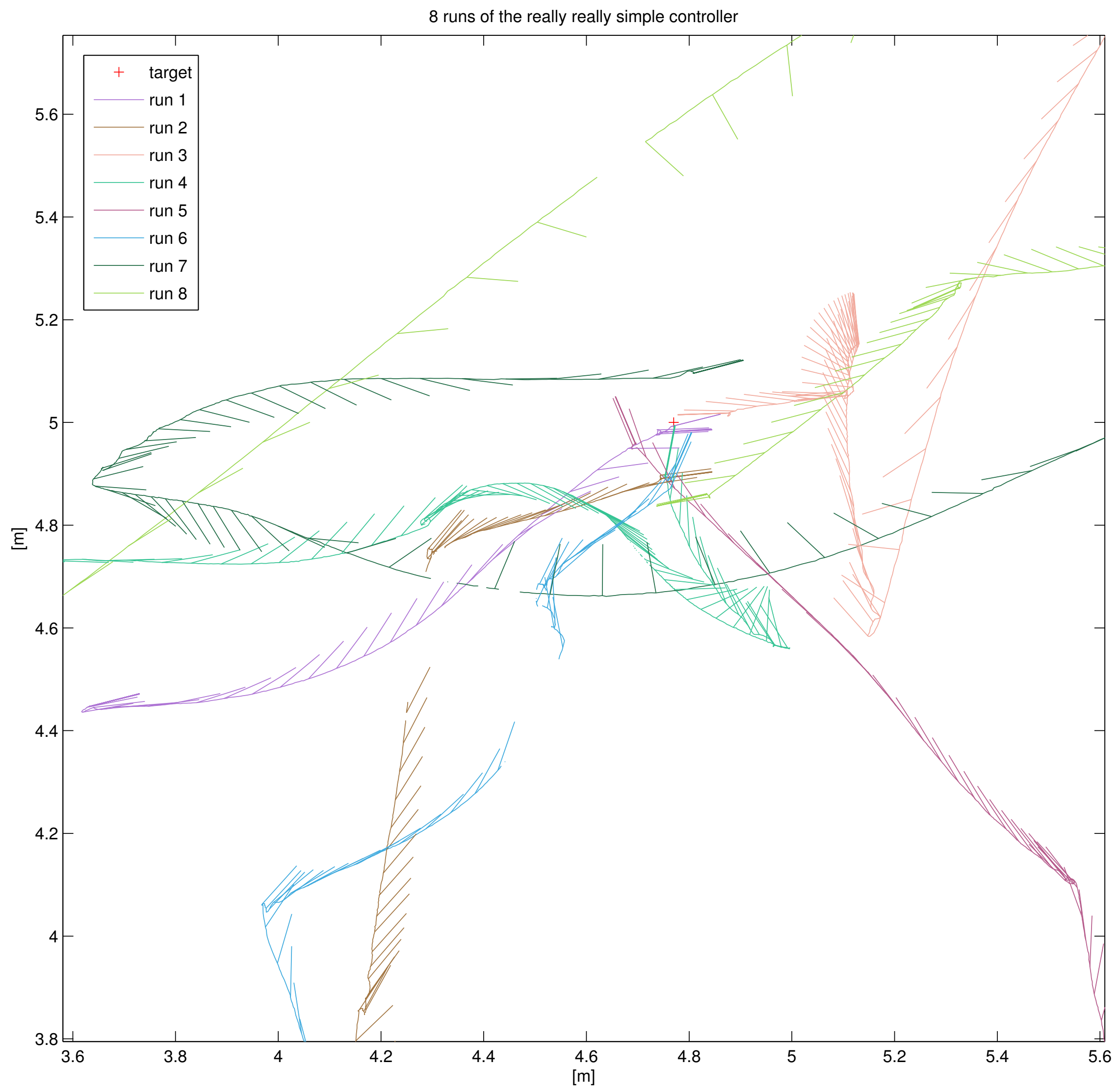


Figure 7.4: Close up of the target location for the same 8 runs.

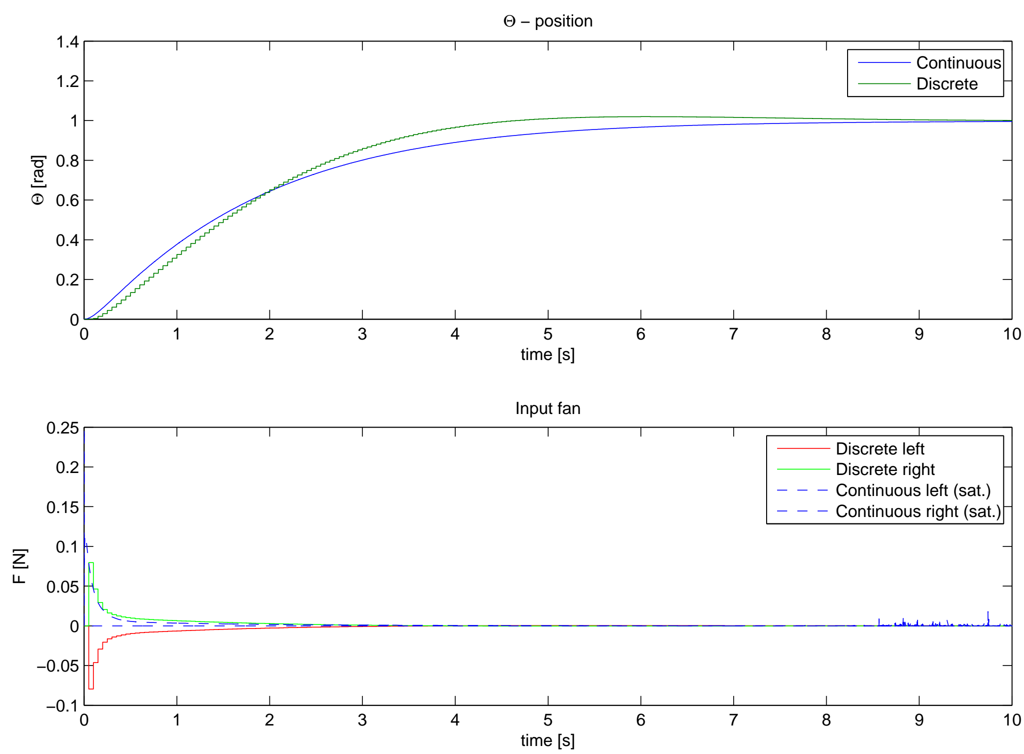


Figure 7.5: Step Response of the hover craft for a step in the orientation with H_∞

Chapter 8

Command Center

The command center is the application that will tell the hovercrafts what to do. It receives the position of all vehicles from the vision system and can read trajectory files, either in the SPARROW trajectory format (with timestamp) or from a simpler SPARROW matrix. It can furthermore send commands to the vehicles. So, for example it might right reference trajectories and pass them on to the vehicles, or it might run them through some kind of planner and then send commands.

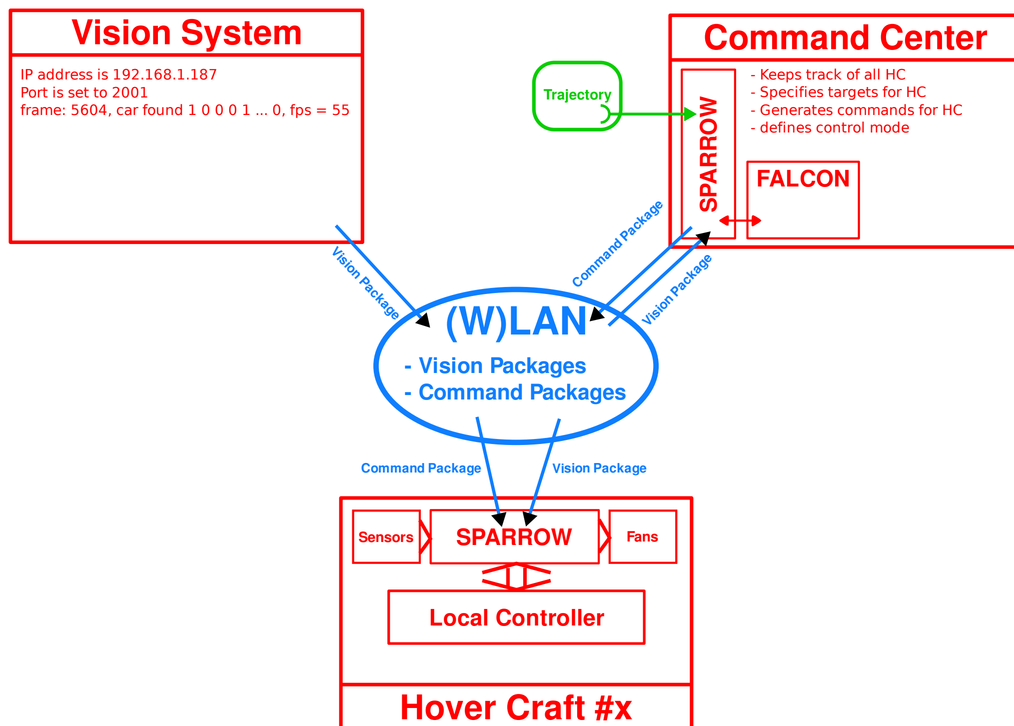


Figure 8.1: High level software architecture of the MVWT testbed

8.1 Drivers

The vision and command drivers are used in their command center version. Also, the trajectory driver is used. The following `.dev` file is used:

```
device: mvwt_commanddriver 49 0x00 -HCID=0 -group_id=1;
device: mvwt_visiondriver 97 0x00 -HCID=0;
device: trajdriv 49 0x00 -filename=sample.trj;
```

8.2 Binary

The command center will bring up a display like shown in fig. 8.2. Pressing `r` will perform `chn_read`, pressing `l` will start a thread that performs `chn_read` and also starts logging the vision system output to a file.

8.3 Algorithms

The command center is currently programmed to read a file containing an n by 2 matrix, containing n target positions. One hovercraft is specified to be the ‘leader’, it will be sent a command containing the location of the closest point of the trajectory. Another hovercraft will act as a wingman, trying to follow the leader with a certain offset.

8.4 What works and what does not

The command station will correctly identify the next point of the trajectory and send it to the leader. It will also correctly calculate the wingmen position and send it as well.

Unfortunately, there seems to be a bug in either the command station or the spread system. Whether the hovercraft receives its commands sometimes depends on whether the command center or the hovercraft controller was started first. For some strange reason, the target position for the wingmen will not be transmitted correctly once the wingmen has reached its target position once - the command station will then produce Spread error messages and/or not send the commands correctly (95% of the time). Tracking down this bug was not successfull.

MVWT COMMAND CENTER															
VISION									COMMAND				TRAJECTORY	0.00	
ID	x	y	t	xd	yd	td	time	dropped	type	value1	value2	value3	value1	value2	value3
01	0.000	0.000	+0.000	+0.00	+0.00	+0.00	0.00	0	0	+0.00	+0.00	+0.00	+0.00	+0.00	+0.00
02	0.000	0.000	+0.000	+0.00	+0.00	+0.00	0.00	0	0	+0.00	+0.00	+0.00	+0.00	+0.00	+0.00
03	0.000	0.000	+0.000	+0.00	+0.00	+0.00	0.00	0	0	+0.00	+0.00	+0.00	+0.00	+0.00	+0.00
04	0.000	0.000	+0.000	+0.00	+0.00	+0.00	0.00	0	0	+0.00	+0.00	+0.00	+0.00	+0.00	+0.00
05	0.000	0.000	+0.000	+0.00	+0.00	+0.00	0.00	0	0	+0.00	+0.00	+0.00	+0.00	+0.00	+0.00
06	0.000	0.000	+0.000	+0.00	+0.00	+0.00	0.00	0	0	+0.00	+0.00	+0.00	+0.00	+0.00	+0.00
07	0.000	0.000	+0.000	+0.00	+0.00	+0.00	0.00	0	0	+0.00	+0.00	+0.00	+0.00	+0.00	+0.00
08	0.000	0.000	+0.000	+0.00	+0.00	+0.00	0.00	0	0	+0.00	+0.00	+0.00	+0.00	+0.00	+0.00
09	0.000	0.000	+0.000	+0.00	+0.00	+0.00	0.00	0	0	+0.00	+0.00	+0.00	+0.00	+0.00	+0.00
10	0.000	0.000	+0.000	+0.00	+0.00	+0.00	0.00	0	0	+0.00	+0.00	+0.00	+0.00	+0.00	+0.00
11	0.000	0.000	+0.000	+0.00	+0.00	+0.00	0.00	0	0	+0.00	+0.00	+0.00	+0.00	+0.00	+0.00
12	0.000	0.000	+0.000	+0.00	+0.00	+0.00	0.00	0	0	+0.00	+0.00	+0.00	+0.00	+0.00	+0.00
13	0.000	0.000	+0.000	+0.00	+0.00	+0.00	0.00	0	0	+0.00	+0.00	+0.00	+0.00	+0.00	+0.00
14	0.000	0.000	+0.000	+0.00	+0.00	+0.00	0.00	0	0	+0.00	+0.00	+0.00	+0.00	+0.00	+0.00
15	0.000	0.000	+0.000	+0.00	+0.00	+0.00	0.00	0	0	+0.00	+0.00	+0.00	+0.00	+0.00	+0.00
16	0.000	0.000	+0.000	+0.00	+0.00	+0.00	0.00	0	0	+0.00	+0.00	+0.00	+0.00	+0.00	+0.00

Figure 8.2: The dynamic display of the command station. All numerical values are in blue. Light blue values can be changed manually. Buttons are green.

Appendix A

Timeline

Week 1

In the first week we decided to work on the MVWT project after being given an introduction by Vanessa. Also, Nok gave as an introduction on Alice. Also, we gathered all the documentation on the MVWT we could find.

Week 2

Together with Richard we came up with a first version of a gotcha chart, defining the objectives of the project. We spend 3 days cleaning the lab. Also, we ordered the gumstix.

Week 3

Vanessa helped us to dig out the code from the subversion repository. She showed us to assemble the hardware. We were able to get one hovercraft to work. We found 4 working PCBs. We were able to cross compile code for the PDA and hooked the PCB up to a PC. Finally, we ordered new batteries. On the weekend, we figured out how to compile Sparrow.

Week 4

We figured out that the gyroscope worked pretty well after fixing some major bugs in the data conversion. However the other sensors are not producing useful data. We set up the build environment for the gumstix and run our first programs on the gumstix. We also started to try to communicate with the hover craft circuit board from the gumstix via a RS232 connection to control the fans and read the sensor data.

Week 5

This week was all about getting Sparrow to run on the gumstix. Crosscompiling Sparrow requires crosscompiling ncurses which requires crosscompiling Berkeley database. Finally, the dynamic display of Sparrow was working on the gumstix. The RS232 connection between the gumstix and the hover craft hardware remains unresolved.

Week 6

We figured out how to access the data being send by the vision system. The serial driver can now read, write and uses threads. We wrote a vision driver using the Sparrow library which gathers the data packages from the vision system and extracts the needed information for each device.

Week 7

The gumstix can not control the fans. We spend a lot of time searching for the reason. The serial booster we bought does not solve the problem. We successfully cross compiled FALCON. We also wrote a command driver to be able to send commands from the command station to the individual hover crafts. For this reason we are using the SPREAD library and we also cross compiled it for the gumstix.

Week 8

The hovercraft is run on a leash, controlled by a PC. A very basic 'proof of concept' controller will cause the hovercraft to navigate to a specified point. We tried to estimate the model parameters, such as friction and inertia of the hovercraft. FALCONis used to implement a state space controller. We were still trying to get the RS232 connection to work but without any success so far.

Week 9

In the week of Thanksgiving we refined the command station.

Week 10

The trajectory driver which uses FALCON to read a trajectory file has been written. The command center can now monitor all vehicles and send commands to them, which might come from a trajectory file for example. After reading through the code of the ATMEL ATmega128 microprocessor and its documentation from ATMEL itself in the past weeks we found the possible error which caused all these strange effect with the RS232 connection. After reflashing the ATMEL with the corrected code, the gumstix can finally control the fans. The speed had be lowered significantly. We started writing this very document.

Week 11

We spent most time on the documentation.

Week 12

We implemented a really really simple controller but failed to achieve the task of formation flight.

Appendix B

The Hidden Treasures (SVN)

Here the location of the files on a CDS subversion should be noted.

Appendix C

Starting The System

Compile the command station and make sure the files `simple.trj` and `test.dat` are present in its working directory.

For the localcontroller, use bitbake and ipkg to install it on the gumstix. You need the files `cont_tra.dat`, `cont_rot.dat`, and `cont_rot_sample_time.dat`.

Appendix D

Doxygen

In the `../packages/localcontrol` directory is a Doxygen settings file called `lcdoxconfig`. To generate the Doxygen documentation just run the command `doxygen lcdoxconfig`. This will compile the Doxygen documentation of the code and place it in `./files/doc`.

Appendix E

Hardware and Parts

All hardware is stored in boxes in the MVWT lab. All boxes are either labeled or transparent.

Appendix F

Documentation Of Previous Projects

This chapter describes the documentation of previous MVWT projects (all we could find). It is actually included here. May it never be lost again.

MVWT 2003

Filename: `Hovercraft-design.pdf`

Date: September 11, 2003

Authors: Beth Wildanger, Mike Lammers, Dave Held, Hans Scholze, Peter Foley, Rob Christy, Tina Hsu

This document describes a previous version of the hovercrafts where the lift fan was mounted eccentric on the plate. It contains information on how to manufacture the mechanical parts and how to set up the Zaurus PDA and transfer software onto it. The PCB is described in detail, allthough the document contains no schematics. Also the assembly of the hovercraft is explained. Contains part lists.

MVWT 2003

Beth Wildanger, Mike Lammers, Dave Held, Hans Scholze
Peter Foley, Rob Christy, Tina Hsu

Control and Dynamical Systems
California Institute of Technology
Pasadena, CA 91125

September 11, 2003

Contents

1 Introduction **3**

2 Hovercraft **3**
- 2.1 Design Summary . . . 3
- 2.2 Specifications and Parameters . . . 4

3 Component Construction **4**
- 3.1 The Plate . . . 4
- 3.2 Lift Fan Holder . . . 5
- 3.3 U-channel Fan Mounts . . . 6
- 3.4 Elastic Straps . . . 6
- 3.5 Fan Safety Cages . . . 7
- 3.6 Flexifoam . . . 7
- 3.7 Sensor Hat . . . 8

4 Zaurus Setup **9**
- 4.1 Preparing the SecureDigital Card . . . 9
- 4.2 Installing software . . . 9

5 Atmel Interface Board **10**
- 5.1 Overview . . . 10
- 5.2 Hardware Description . . . 11
  - 5.2.1 Power Supply . . . 11

5.2.2 ATMega128 12
5.2.3 RS-232 12
5.2.4 ISP Header 12
5.2.5 Gyro 12
5.2.6 Magnetic Heading Sensor 12
5.2.7 Accelerometers 13
5.3 PCB Fabrication 13
5.4 Assembly 13
5.4.1 Zaurus Serial Cable 14
5.4.2 Soldering the Surface Mount Components 14
5.5 Programming the ATMega128 15
5.5.1 Development Tools 15
5.5.2 Programming the Fuse Bits 15
5.5.3 Writing the Program Flash 16

6 Force Map Generation 16
6.1 Overview 16
6.2 Data Format 17
6.3 Force Mapping Board Software 17
6.4 Procedure 17

7 Hovercraft Assembly 18
7.1 Thrust Fans 18
7.2 Lift Fan 18
7.3 Zaurus 19
7.4 PCB 23
7.5 Restraining Bolts 23

8 Testbed: Roof 24

9 Design Considerations 25
9.1 Lift Fan 25
9.2 Thrust Fan 26
9.3 Batteries 26
9.4 Base 27

10 Future Work 27

A Bill of Materials **28**
A.1 Physical . . . 28
A.2 Computational . . . 28
A.3 Electrical . . . 29

B CNC Code **30**
B.1 Hovercraft Plate . . . 30
B.2 Lift Fan Holder . . . 31
B.3 Sensor Hat . . . 33

C Testbed Materials **37**

D Data Acquisition **38**
D.1 Coefficient of Friction . . . 38
D.2 Moment of Inertia . . . 38

1 Introduction

MVWT 2003 has seen much progress toward the construction of twenty-four hovercraft and their surrounding environment. This documentation was written to assist future developers towards completion of the project. This information includes the hovercraft construction procedures and design plans for the roof testbed. In addition, information has been included detailing the different considerations that led to the current design of the hovercraft. It is our hope that this information will make it possible for anyone to make as many of these hovercraft as they might want.

2 Hovercraft

2.1 Design Summary

The hovercraft base is a round $7\frac{1}{2}$" in diameter plastic plate turned upside down, so that the rim of the plate forms a plenum chamber. The pressure rise is provided by a downward thrusting electric ducted fan acting through a two-inch hole near the edge of the plate. Two more electric ducted fans on either side of the lift fan provide thrust for the craft. The on board processing is powered by a 200 MHz Sharp Zaurus, a Linux based handheld computer.

2.2 Specifications and Parameters

Parameter	*Hovercraft value*
Mass	800 g
Moment of intertia	31,603 g cm^2
Distance between thrust fans	17.8 cm
Maximum fan thrust	0.7 N
Cost (without GPS)	$860
Lift Fan Battery lifetime	35-40 minutes

3 Component Construction

3.1 The Plate

The CNC machine referred to is the one in the ME shop, in the subbasement of Spalding. Please ask John or Rodney, the staff, before attempting to use it on your own.

Figure 1: CNC Machine and John

1. Use a sander to grind off the lip on the bottom of the plastic plate.
2. Insert the hovercraft plate fixture into the CNC.
3. Find the center of the fixture. *Note: The center finder is very expensive. John or Rodney should be present for this step.*
4. Fasten the plate to the CNC hovercraft plate fixture by clamping the fixture ring around the lip of the plate.
5. Tool 1: # 28 straight flute drill
   Tool 2: $\frac{1}{4}$" four flute end mill
6. Set tool heights for drill and end mill.
7. Run the program (located in Appendix B).

4

8. This process may be repeated without any need to reset the center location or drill heights as long as the fixture and the tools are not adjusted.

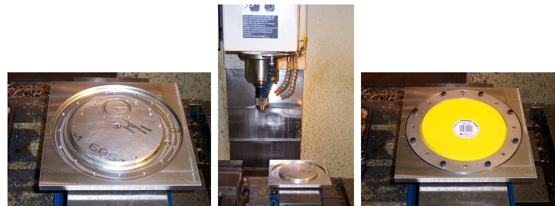


Figure 2: Plate on CNC Machine

3.2 Lift Fan Holder

1. Use a bandsaw to cut a piece of $\frac{1}{16}$" sheet aluminum into a square $2\frac{1}{4}$" on a side. Use a grinder or file to remove sharp edges.

2. Insert the lift fan holder fixture into the CNC.

3. Find the center of the fixture. *Note: The center finder is very expensive. John or Rodney should be present for this step.*

4. Use fixture clamping arms to clamp the $2\frac{1}{4}$" square aluminum sheet into the fixture.

5. Set up tools in the same manner as when CNCing the plate, being sure to set tool heights.

6. Run the program (located in Appendix B).

7. When the program pauses, bolt the sheet down through the freshly drilled holes.

8. Remove the clamping arms.

9. Hit "start" and the program will continue to completion.

10. This process may be repeated without any need to reset the center location or drill heights as long as the fixture and the tools are not adjusted.

11. Use tin snips to cut through one side of the lift fan holder. This gives it the flexibility to fit around the rim of the fan.

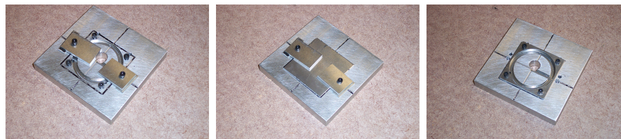


Figure 3: Lift Fan Holder

3.3 U-channel Fan Mounts

1. Use a bandsaw to cut U-channel into sections with a width slightly less than that of the fan.
2. Use a grinder or file to remove any burrs on the PVC.
3. Press U-Channel section against the inner wall of the U-channel Drill Fixture and C-clamp it in place.
4. Drill holes in the U-channel using a hand drill with a #28 drill. Use the hole guides in the U-channel Drill Fixture to drill the holes in the proper locations.

3.4 Elastic Straps

1. Cut a 6” length of elastic strap.
2. Cut a small hole in one end of the elastic strap.

Figure 4: U-Channel Drill Fixture

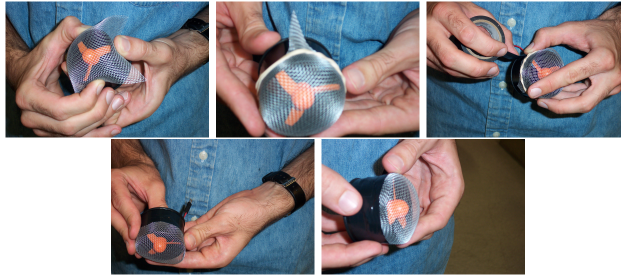


Figure 5: Safety Cage Assembly Process

3. Attach grommet through this hole.
4. Cut two small holes next to each other in the other end of the elastic strap.

3.5 Fan Safety Cages

1. Cut a square of aluminum mesh approximately 4 inches on a side.
2. Pull the four flat sides of the mesh against the side of the fan.
3. Push a rubber band down around the mesh to hold it in place against the fan.
4. Cut off any large sections of aluminum mesh sticking out the other side of the rubber band.
5. What portion of the aluminum remains sticking out should be attached using electrical tape. Once affixed, remove the rubber band and apply more tape to hold the aluminum mesh more firmly.
6. Put safety cages on all the fans: the thrust fans and the lift fan.

3.6 Flexifoam

Foam padding is necessary to protect and cushion certain parts of the hovercraft. Please cut a sheet of flexifoam into the following shapes:

1. A 2.5" x 4" rectangle.

2. A circle approximately 2.5” in diameter with holes aligning with the holes in the lift fan holder.

3.7 Sensor Hat

The hovercraft was designed to be highly compact, cramming all of the essential components onto a small surface. The sensor hat solves this problem by providing an extra mounting surface. It is a $\frac{1}{8}$” plexiglass plate measuring 7” by 9”. It attaches to the bolts that hold the elastic for the thrust fans. It should be noted that it relies on the lift battery for a third support location and should not be heavily loaded when the lift battery is not present. There is a clearance hole for a $\frac{1}{4}$-20 bolt at it’s center (this allows a standard webcam to be mounted) and an array of clearance holes for $\#4-40$ bolts spaced at $\frac{5}{8}$”, providing mounting locations for sonar and IR sensors.

It should be noted that the sensor hat is currently in the earliest stages of prototyping. It appears that the $\frac{5}{8}$” spacing between the #4-40 clearance holes is not quite appropriate for the sonar mounts (although appears as if it will just barely work).

Immediately visible problems are:

1. Blocked Airflow: Future designs should remove more material above the lift fan to improve air flow.
2. Sensor Blocking: When used in conjunction with a vision system hat, it appears that some sensors may be peripherally blocked.
3. Battery Replacement: The sensor hat currently makes it much more difficult to change batteries.
4. Vision System Hat: Currently the spacers between the vision system hat and the sensor hat are simply 4 bolts sticking straight up. Each bolt has 2 nuts tightened down against each other that provide the base of support for the vision system hat. In the future a more permanent riser should be manufactured to more firmly hold the vision system hat in place.

The manufacture of the sensor hat is simple. The CNC code can be found at the end of this document. $Careful$: the code is currently flawed. The $\frac{1}{4}$-20 bolt must be removed at two points during the procedure to avoid damaging the tools. To CNC a sensor hat, simply clamp a 7” by 9” piece of $\frac{1}{8}$” plexiglass into the CNC fixture such that the four #6-32 threaded holes are under the corners of the plexiglass. Run the program until pauses,

8

remove the clamps, bolt it to the fixture through the $\frac{1}{4}$-20 and #6-32 holes. Allow it to run again, being very wary of the $\frac{1}{4}$-20 bolt at the center.

4 Zaurus Setup

4.1 Preparing the SecureDigital Card

This is only necessary if updating or replacing the card. If several identical Zauri are to be prepared, this step only needs to be done once.

1. Open the folder "M:/MVWT/zaurus files" where M: refers to \\pcfiles.cds.caltech.edu\mvwt.
2. Copy the folder "rootfs" onto the SecureDigital card.
3. Open the folder "SDRoot" and copy all the contained files into the base directory of the SecureDigital card.

4.2 Installing software

1. Turn on the Zaurus and go through its setup process (calibrate the screen, set times, etc.).
2. Insert the SecureDigital card into the Zaurus.
3. Go to the "Documents" tab, click the "qpe-terminal_1" package, click the "Install" button, and install to RAM. When the installation is complete, close all open windows by clicking the X button in the upper-right corner.
4. Go to the "Applications" tab and open the "terminal" program. Type `cp -R /mnt/card/rootfs/* /` to copy over configuration files, install the wireless network card driver, install minicom, and allow telnet connections over the network.
5. In order to allow remote access, the root password must be set. Still in terminal, type the command `passwd`. When prompted for the new password, type `mvwt2003`, using the Zaurus's function key to type the numbers. Confirm the new password and close the terminal when finished.
6. Open the "Wireless LAN Setting" under the "Settings" tab. Set "Specific ESSID" to "caltechmvwt," and set "Network Type" in "Infrastructure". Press OK to save the settings and close the window.

7. Open “Network & Sync” under the “Settings” tab. Under “Services”, click the “Add” button. Select “LAN - TCP/IP” and click “Add”. Select the “Specify TCP/IP Information” radio button, then go to the “TCP/IP” tab and input the following settings:

IP	–	192.168.1.2## where ## is the number of the Zaurus
Subnet Mask	–	255.255.255.0
Broadcast	–	192.168.1.255
Gateway	–	192.168.1.1

   Now under the ”DNS” tab, input the nameservers below:

DNS Nameservers	–	131.215.42.28
	–	131.215.9.49

   Under the “Proxies” tab, select “No Proxies.” Click “OK” in the upper-right corner to save the settings.

8. Still under the ”Settings” tab, click ”Light & Power.” Turn off the backlight and set the Zaurus to suspend after 3600 seconds.

9. Now install the FTP daemon by clicking the troll-ftpd_package under ”Documents”. The installation process is identical to that of the terminal.

10. Unmount the SecureDigital card by clicking the ”SD” icon in the lower right and selecting ”Eject SD-card”.To remove the card, press it until it clicks, then release and it will pop out like a push-on, push-off switch. Reboot the Zaurus (Settings→Shutdown→Reboot) to load the wireless card drivers and configuration settings.

5 Atmel Interface Board

5.1 Overview

The Atmel interface board provides the low-level interface between the Zaurus PDA and the various actuators and sensors on the hovercraft. The Atmel microcontroller communicates with the Zaurus via the RS-232 serial port and is responsible for reading the gyro, magnetic heading sensor, and accelerometers as well as generating PWM control signals for the fan speed controllers.

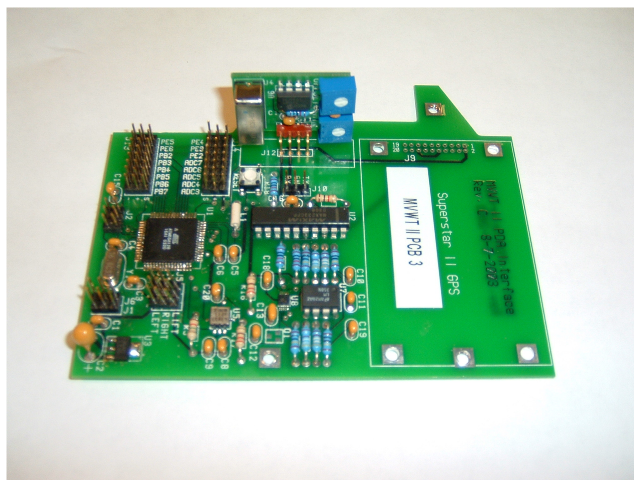


Figure 6: Completed Interface Board

5.2 Hardware Description

The interface board is built around a Atmel ATMega128. The ATMega128 is a microcontroller using Atmel's AVR 8-bit RISC architecture running at 16Mhz. It includes a UART for serial communication, an 8-channel 10-bit ADC, 8 external interrupts, and a number of 8-bit and 16-bit timer/counters. More information on the ATMega128 can be obtained from the Atmel web site, www.atmel.com. The web site www.avrfreaks.org has a wealth of information about programmers and compilers for AVR microcontrollers.

5.2.1 Power Supply

The interface board is powered from the same 7.2V battery that powers the lift fan. The regulated 5V supply is provided by the LM2940IMP (U3). The choice of output filter capacitor (C2) is important. The LM2940 requires the ESR of the cap to be within a certain range. Refer to the LM2940 datasheet for more information.

5.2.2 ATMega128

The ATMega128 requires very few external components to operate. L1, C5, and C6 filter noise from the ADC's supply and internal reference, as recommended by Atmel.

5.2.3 RS-232

RS-232 level conversion is done with a Maxim MAX233.
IMPORTANT: R1 is necessary because the ISP and UART share the same I/O pins on the ATMega128. R1 allows the programmer to override the output of the MAX233 during programming. Because of this, the serial port will not work when to programmer is connected.

5.2.4 ISP Header

The interface board includes Atmel's standard 6-pin in-circuit programming header (J2) to allow the ATMega128 to be easily programmed without removing it from the board.
IMPORTANT: R1 is necessary because the ISP and UART share the same I/O pins on the ATMega128. R1 allows the programmer to override the output of the MAX233 during programming. Because of this, the serial port will not work when to programmer is connected.

5.2.5 Gyro

The gyro circuit is designed as a separate 4-pin module so that it can mount vertically on the main PCB with a right angle connector. The sensor used is a Tokin CG-16D ceramic gyro. The output of the gyro module is 1.1mV/deg/sec ±20% referenced around 2.4V. The offset at zero angular rate can vary as much as ±300mV from gyro to gyro. To compensate for this and provide a voltage between 0V and 5V as required by the ADC, the signal from the gyro sent through a simple differential amplifier built around an MCP601 op amp. R2 adjusts reference input to the amplifier to correct offset in the output of the gyro module. R4 allows the gain to be varied from 1 to 10. C7 filters out high-frequency noise in the gyro module's output.

5.2.6 Magnetic Heading Sensor

The magnetic heading sensor allows the hovercraft to get an absolute heading to complement the GPS position data when running without the vision

12

system. The sensor used is the HMC1052, a 2-axis magnetoresistive sensor made by Honeywell. The compass circuit is taken directly from Honeywell *AN214 Reference Design: Low-Cost Compass*. Currently, the interface board only returns the raw ADC readings from each axis. The functions to calibrate the compass and calculate the heading will run on the Zaurus, but have not yet been written.

5.2.7 Accelerometers

The interface board includes an Analog Devices ADXL202E 2-axis accelerometer to measure acceleration in the forward/backward and side-to-side axes. The ADXL202E can measure ±2g acceleration. The output of the ADXL202E is a fixed-frequency square wave with duty cycle corresponding to the acceleration. ±1g of acceleration corresponds to approximately ±12.5% duty cycle. The time high for each channel and the total period (which is the same for both channels) are measured by the ATMega128 and sent to the Zaurus, where the actual acceleration is calculated. C8 and C9 set the bandwidth of the accelerometers. R5 sets the period of the output square waves. Details on choosing C8, C9, and R5 are in the ADXL202E datasheet.

5.3 PCB Fabrication

The PCBs for the interface board were fabricated by Advanced Circuits, www.4pcb.com. The schematics and PCB layouts are all on the M: drive. Advanced Circuits requires that you submit a zip file containing gerber files for each copper, soldermask, and silkscreen layer. The zip file of the most recent revision of the board is mvwtrevc.zip. If more boards need to be ordered Advanced Circuits should still have the tooling and artwork on file. The order information is:

Customer Number: 21310
Quote Number: 68973
Part Number: mvwt2
Revision: C

Our salesperson at Advanced Circuits is Lydia Arriaga, (800)289-1724 x302, lydia@4pcb.com.

5.4 Assembly

Assembly of the interface board is straightforward. The gyro board is tab routed so it will snap off of the main PCB. It mounts vertically to the main

PCB with a 4-pin right-angle header.

5.4.1 Zaurus Serial Cable

Cut the serial down so its just long enough to reach from the Zaurus to the 3-pin RS-232 header on the interface board. The pinout of the cable is:

Red Data from the Zaurus to the ATMega128. Connects to RX on the interface board.

White Data from the ATMega128 to the Zaurus. Connects to TX on the interface board.

Brown Ground.

5.4.2 Soldering the Surface Mount Components

Soldering the surface mount components can be difficult. Here's a method that is fairly quick and easy:

1. Apply a thin coat of flux to the pads. The liquid flux with a brush applicator sold in the EE stockroom is ideal for this.
2. Place the component in approximately the right location with tweezers or needle-nose pliers. The flux will help keep it from sliding around.
3. Carefully poke at the component with something small (a piece of wire works well) until it is aligned on the pads.
4. Make sure the soldering iron is clean and tinned and place a small dab of solder on the tip.
5. Holding the component in place with your finger, touch the tip of the iron to one of the pads at a corner of the component. The solder should wick into the joint. Be careful because the blob of solder can pull the component out of alignment as it cools.
6. Do the same for a pad at the opposite corner, so the component is held securely in place.
7. Apply more flux over the pins of the component to ensure they are completely covered.
8. Fill some solder wick with flux. Filing the last half inch to inch or so seems to work well. Put enough solder on it so that is is completely soaked, but not so much that forming a blob of solder.

9. Gently press the end of the solder wick against the board with the soldering iron so that the solder is molten. Using the soldering iron, carefully slide the solder wick up against the pads/pins of the component.

10. If all goes well, a small amount of solder will wick into each joint. Move down the row of joints while repeatedly sliding the wick up against them. Do not drag the wick across the joints, as this will form solder bridges.

11. When finished, inspect the joints carefully to make sure they are all soldered and there are no solder bridges. If there are any suspicious looking spots, just apply more flux and go over them again as before.

5.5 Programming the ATMega128

5.5.1 Development Tools

For developing software for the ATMega128 we used the free WinAVR set of tools and Atmel's AVRISP programmer. WinAVR includes a port of gcc to the AVR architecture which runs under windows. It also includes avrdude, command-line software to program the microcontroller. The WinAVR web site is winavr.sourceforge.net. The AVRISP is a simple, inexpensive programmer that allows programming of the microcontroller in-circuit through the 6-pin ISP header.

5.5.2 Programming the Fuse Bits

Before the interface board will boot for the first time, the fuse bits of the ATMega128 need to be programmed. The fuse bits are used to configure peripherals, set the clock source, etc... For more information on the fuse bits, refer to the programming section of the ATMega128 datasheet. The values of the fuse bytes for use with the interface board are:

Low Fuse Byte 0xFF
High Fuse Byte 0xC9
Extended Fuse Byte 0xFF

The procedure to program the fuse bits with avrdude and the AVRISP programmer is:

1. Setup the programmer as in the section on writing the program flash.

2. Use `avrdude -p ATMEGA128 -c avrisp -t` to open the programmer in terminal mode.
3. `w lfuse 0 0xff` to write the low fuse byte.
4. `w hfuse 0 0xc9` to write the high fuse byte.
5. `w efuse 0 0xff` to write the extended fuse byte.

5.5.3 Writing the Program Flash

1. Connect the AVRISP programmer to the serial port on the computer.
2. Power the board and plug the programmer in to the ISP header. Make sure pin 1 on the programmer (indicated by red wire/small arrow on the connector) goes to pin 1 on the PCB (the square pad).
3. When the programmer is plugged in its light should blink a couple times then remain green (the programmer draws its power from the board). If not, try turning the power off and on again.
4. Compile the source to an srec object file that avrdude can recognize. With the standard WinAVR makefile this can be done with `make srec`.
5. Use `avrdude -p ATMEGA128 -c avrisp -e` to erase the flash. `-p` specifies the device being programmed, `-c` specifies the programmer to use, and `-e` erases the flash.
6. Program the flash with `avrdude -p ATMEGA128 -c avrisp -i mvwt2.srec`. `-i` specifies the object file to use.

6 Force Map Generation

6.1 Overview

The force mapping setup measures actual fan thrust as a function of control input to the fan controller. A modified interface board (the force mapping board) is used to slowly ramp up the fan speed while measuring the output from a load cell connected to the fan.

6.2 Data Format

At each power level the force mapping board sends an ASCII data point in the format `command, thrust,` to the serial port. `command` is dimensionless power level the board is sending to the fan. `thrust` is the value the board read from the ADC for that particular command. Each data point is on a new line. The output of the force mapping board can be copied directly from the terminal window into a *.csv (comma-separated value) text file which can be read by Excel.

6.3 Force Mapping Board Software

The force mapping program starts as soon as the ATMega128 boots. It does the following:

1. Set the current power level of the fan controller.
2. Wait a certain amount of time to allow the thrust to stabilize.
3. Take 8 ADC readings of the load cell output.
4. Average together the ADC readings and send them to the serial port.
5. Increase the current power level and repeat. Stop when the fan is at max power.

There are `#define`'s in the source code to setup:

- Time to allow the thrust to stabilize.
- Period of time to take the 8 readings over.
- Size of each step in power level.

6.4 Procedure

1. Connect the load cell amplifier to the `ADC3` input of the force mapping board.
2. Connect the fan under test to `LEFT` output of the thrust mapping board.
3. Connect the thrust mapping board serial port to the computer's serial port.

4. Place the fan to test in the force mapping jig.
5. Open a terminal window on the computer.
6. Apply 28V to the load cell amp.
7. Apply 7.2V power to the thrust mapping board and fan controller.
8. The ATMega128 should boot and begin sending data points. If not, reset it.
9. The thrust map will show up in the terminal window in comma-separated value format. Copy it into a text file and save it as *.csv.
10. Run the thrust mapper a few more times with the fan disconnected and weights on the jig to get some data points to convert the ADC values to actual thrusts.

7 Hovercraft Assembly

7.1 Thrust Fans

1. Bolt the u-channel and the elastic to the plate. The bolts should go through the two holes in the elastic and the elastic should be sandwiched between the u-channel and the plate. Bolt in holes 7 and 8 or 9 and 10 using $\frac{1}{2}$"#6-32 bolts.
2. Attach the other end of elastic to the top of the u-channel through the grommet with a $1\frac{1}{4}$"#6-32 bolt and a nut.
3. Repeat for a second u-channel and elastic strap on the other side of the plate.
4. Slide the fans into the u-channel, allowing the elastic to keep them in place.

7.2 Lift Fan

1. Slide the lift fan holder around the bottom rim of the lift fan.
2. Push the flexifoam circle around the bottom rim of the lift fan.
3. Align the bolt holes in the flexifoam and the lift fan holder.

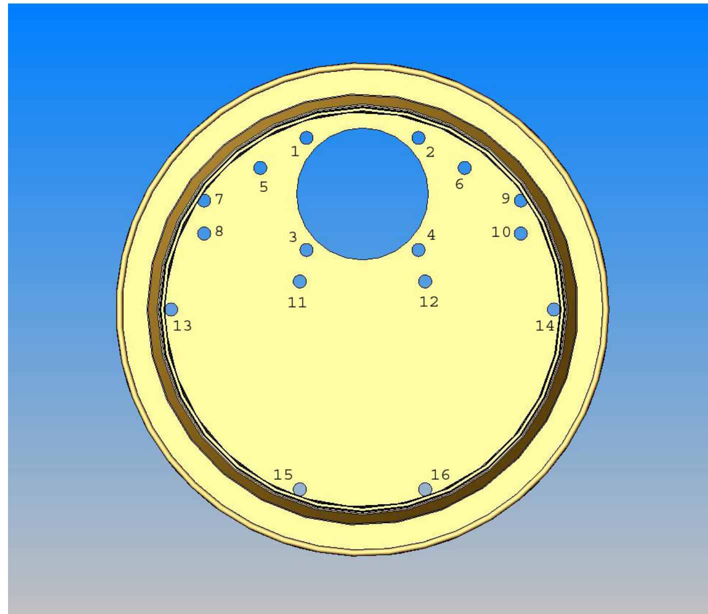


Figure 7: Hole referencing for the plate.

4. Using $\frac{1}{2}$" #6-32 bolts, bolt the lift fan holder to the plate through holes 1 - 4.

7.3 Zaurus

1. Configure the Zaurus using the instructions in Section 4.
2. Label the Zaurus with its number. Label all its corresponding parts, as well.
3. Leaving the flip-top on, position the Zaurus on the hovercraft.
4. Put the flexifoam rectangle between the Zaurus and the PCB.

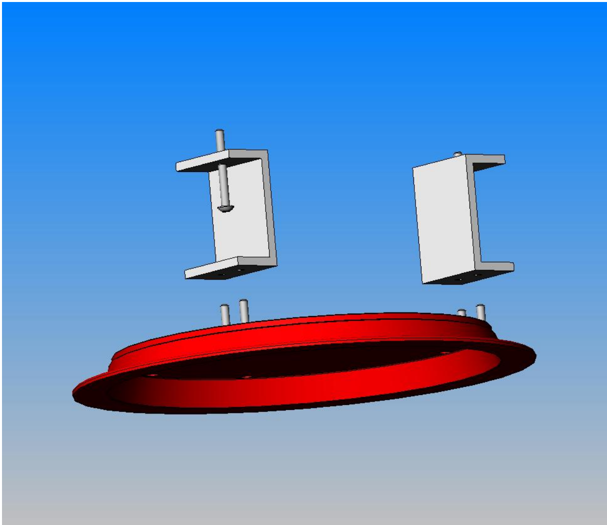


Figure 8: Attaching U-channels to the plate. Note: Elastic not shown here.

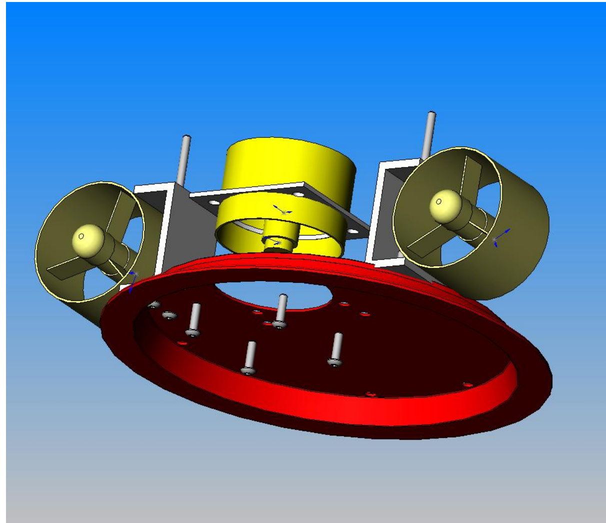


Figure 9: Attaching lift fan.

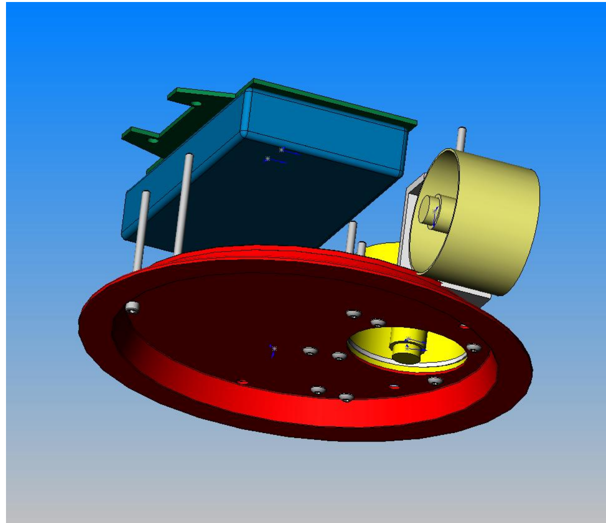


Figure 10: Attaching the PCB and Zaurus.

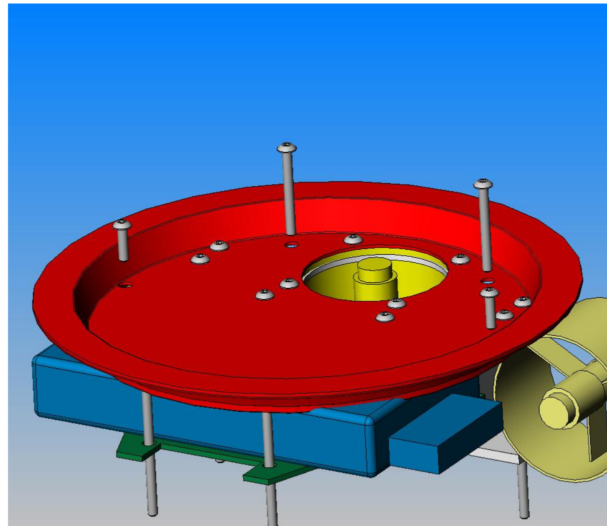


Figure 11: Restraining Bolts.

7.4 PCB

1. After the PCB has been fabricated, bolt it on to the plate with two 1 $\frac{1}{4}$"#6-32 bolts and two 2"#6-32 bolts, gently sandwiching the Zaurus and flexifoam in between. The 2" bolts should be a the rear of the hovercraft (holes 15 and 16). The 1 $\frac{1}{4}$" bolts should go through holes 11 and 12.
2. Attach two rubberbands between the bolts on the back of the hovercraft- one above the PCB and one below. These rubberbands are used to hold on the lift fan's battery.

7.5 Restraining Bolts

Bolts sticking up from the base of the hovercraft are used to keep the Zaurus from slipping out on either side, and to hold the thrust batteries in place. Bolt two 1$\frac{1}{4}$"#6-32 bolts in holes 5 and 6, and two $\frac{1}{2}$"#6-32 bolts in holes 13 and 14.

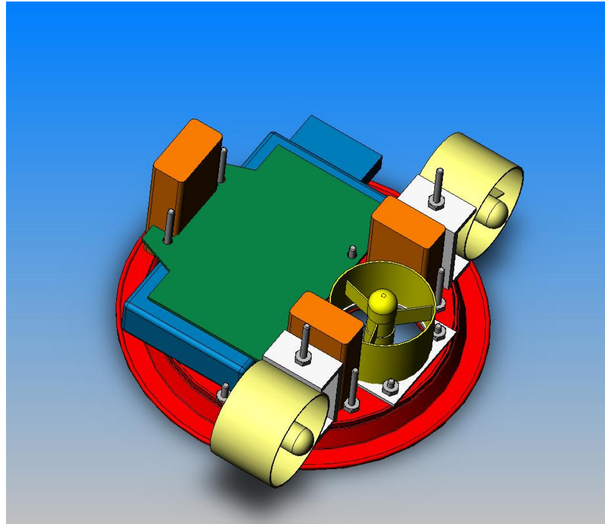


Figure 12: Finished Hovercraft

8 Testbed: Roof

The size of the roof made it the optimal location for a large game of Roboflag. However, its dirty, rocky and highly uneven surface was a problem for the hovercraft. To solve this problem the MVWT II team designed a simple and easily removable testbed for use on the roof.

A type of carpet padding called Berber Max was determined to be the optimal material. It is relatively smooth and can easily be rolled up to be moved aside. Its only problems are a susceptibility to ripping and an inability to even out larger bumps. While tedious, it appears that putting tape along the edge of each roll is enough to take care of the ripping problem. The problem with large bumps is much more serious. The best solution we could come up with was to cut holes in the areas with larger bumps and cover the Berber Max with black plastic. This has only been tested on a small scale, but there is no reason to believe that it wouldn't work if implemented on the entire testbed.

Wooden boards with thin strips of carpet padding are used as bumpers around the perimeter of the testbed. Once the carpet padding is unrolled,

they are placed along the outer perimeter. The bumpers protect the hovercraft from leaving the playing area as well as hold the carpet padding in place.

9 Design Considerations

As with all projects, there are tradeoffs. Optimizing the design requires researching the available options and choosing the best one.

One of the most crucial decisions for MVWT II was what kind of lift fan to use. How could the hovercraft hover high enough to clear the gravelly surface on top of the roof? How could it also be large enough to hold the necessary computing components? Could the hovercraft be made to hover for 40 minutes? Can we overcome these problems simply by finding a smoother roof? An extensive survey of the roofs of campus buildings determined that smoother roofs have large bumps between roofing panels . Any hovercraft at a reasonable hovering height would get caught on these bumps. Increasing the hover height would require higher air flow from the lift fan. In addition to a high hoverheight, we looked into the option of having a flexible and durable skirt that would assist in the gliding over the bumps. Unfortunately, while this helped the hovercraft get over the bumps, there was still a significant increase in friction.

These problems led us to develop a testbed on the roof that could support a simple, lightweight hovercraft. Most importantly this testbed needs to be removable to keep it from being a fire hazard. It would be smoother than the roof and would compensate for the bumpiness.

The main advantage of a lightweight hovercraft is the cost. Less weight means we can use less costly fans that require less power. Less power means that fewer batteries are needed. Our primary limit in shrinking the craft was the size of the computer platform. A handheld computer seemed to work best for this purpose.

9.1 Lift Fan

Selecting the right fan turned out to be more difficult than we had anticipated. Stronger fans require more power and money. More power requires more batteries. More batteries weigh and cost more, and once again, more

weight requires a stronger fan. With some fans we were able to make hovercraft hover a centimeter above the ground, but this would have been very expensive. After deciding on a smooth testbed, the goal was then to make the hovercraft as small as it could be. The fan we decided to use is ≈ 2" in diameter. There were only two fans available in this size. We chose the EDF-50 fan for it's low power consumption.

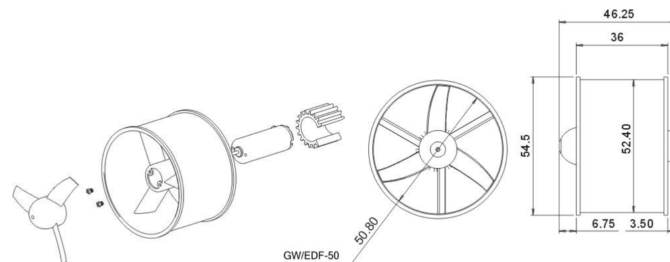


Figure 13: Lift and Thrust Fans (measurements in millimeters)

9.2 Thrust Fan

We chose to use the same fan for thrust as for lift. It was lightweight, and provided adequate thrust (some could even argue there was too much thrust!).

9.3 Batteries

After deciding on the fans, we knew we had to find the densest, lightest, smallest batteries we could find. The fans run on 7.2 Volts and pull about 3 Amps on full thrust. We wanted the lift fan to run continuously for 40 minutes. We also wanted to avoid ordering a custom battery to save on cost. As such we decided to go with mass produced Lithium Ion Battery packs designed for the model airplane industry.

$$40minutes \times \frac{1hour}{60minutes} = \frac{2}{3}hour$$
$$3A \times \frac{1000mA}{A} = 3000mA$$
$$\rightarrow \frac{2}{3}hour \times 3000mA = 2000mAh$$

26

9.4 Base

Determining the shape of the base was our main focus for the beginning of the summer. We wanted to maximize the distance between the thrust fans. At the same time, we wanted the hovercraft to be stable, and simple to build. It would have been simple to try different design variations with a rapid prototyping system, but sadly Caltech does not have one. Because of this, we chose a simple base design using plastic plates in which holes were created using a CNC machine. Full size hovercraft used for racing are usually designed to be twice as long as they are wide. This would give us a smallish moment arm (between the thrust fan and the center) in comparison with the length of the hovercraft. The Kelly's are about twice as wide as they are long to make use of a large moment arm.

10 Future Work

Unfortunately, there are things that the MVWT 2003 team could not complete over the course of the summer.

1. Determine if the Zaurus are an effective as a means of controlling the hovercraft.
2. Determine if current fans are strong enough to work as lift fans. Fans have appeared to get weaker after running for 40 minutes as a lift fan.
3. Build an additional 12 hovercraft.
4. Thrust map all fans.
5. Characterize all gyros.
6. Make improvements to sensor hat. (see 3.7)

A Bill of Materials

A.1 Physical

Qty	*Description*	*Distributor*	*Part Number*
1	Zaurus 5500, Handheld Computer	Sharp	SL-5500
1	Serial Cable for Zaurus	Sharp	CE-170TS
1	Compact Flash Wireless LAN Card	Linksys	WCF12
3	Electric Ducted Fan	GWS	EDF-50
3	Fan Speed Controller	GWS	ICS-100
3	Aluminum Mesh, 3.5" Square	Physical Plant Stockroom	
	Flexifoam	JoAnneFabrics	
2	7.2 V, 950 mAh battery	Duralite Batteries	PPD-7092
1	7.2 V, 1800 mAh battery	Duralite Batteries	PPD-7092
1	7.5" Plastic Plate (Primary Plate)	Walmart (Arrow Plastic)	290
2	1 3/8" wide PVC U-channel, 1.88 base by 0.79 leg length	McMaster-Carr	85065K49
2	Washer Grommets	McMaster-Carr	9604K22
2	1" Wide Elastic Strap	JoAnne Fabrics	
1	2.5" square of 1/16" 6061 Al sheet metal	Materials Warehouse	
2	1/2" of 1" x 3/8" Brass Stock	Materials Warehouse	
2	2" #6-32 bolts	Physical Plant Stockroom	
8	1 1/4" #6-32 bolts	Physical Plant Stockroom	
8	1/2" #6-32 bolts	Physical Plant Stockroom	
18	#6-32 Nuts	Physical Plant Stockroom	

A.2 Computational

Qty	*Description*	*Distributor*	*Part Number*
1	Zaurus 5500, Handheld Computer	Sharp	SL-5500
1	Serial Cable for Zaurus	Sharp	CE-170TS
1	Compact Flash Wireless LAN Card	Linksys	WCF12

A.3 Electrical

Part Number	*Schematic Ref.*	*Qty*	*Description*	*Manufacturer*
ATMega128-16AI	U1	1	Microcontroller	Atmel
MAX233CPP	U2	1	RS232 Transceiver	Maxim
102976-3	J1, J3-6, J10	6	3-pin Header	AMP/Tyco
103186-3	J2	1	6-pin Header	AMP/Tyco
CFR-25JB-1K0	R1	1	1k 5% 1/8W Resistor	Panasonic
LM2940IMP-5.0	U3	1	5V 1A Regulator	
T356K686K016AS	C2	1	68uF 16V Tantalum Cap	Kemet
C315C104M5U5CA	C1,5-9,12,14-21	15	0.1uF Ceramic Cap	Kemet
HC49US16.000MABJ	Y1	1	16MHz Crystal	Citizen America
C320C150J2G5CA	C3, C4	2	15pF 200V Cap	Kemet
1025-44K	L1	1	10uH Choke	API Delevan
CG-16D	U4	1	Ceramic Gyro	Tokin
ADXL202AE	U5	1	2 Axis Accelerometer	Analog Devices
3362P-1-104	R2, R4	2	100k Pot	Bourns
CFR-25JB-10K	R3, R18	5	10k Resistor	Yageo
CFR-25JB-1M2	R5	5	1.2M Resistor	Yageo
MCP601-I/P	U6	1	Single Op Amp	Microchip
LM358N	U7	1	Dual Op Amp	National Semi.
HMC1052	U8	1	Magnetic Sensor	Honeywell
MFR-25FBF-4K99	R6-9	5	4.99k 1% Resistor	Yageo
MFR-25FBF-1M00	R10-13	5	1.00M 1% Resistor	Yageo
C315C102K1R5CA	C10, C11	2	1000pF Ceramic Cap	Kemet
MFR-25FBF-10K0	R14, R15	5	10.0k 1% Resistor	Yageo
C320C224M5U5CA	C13	1	0.22uF Ceramic Cap	Kemet
MMBT2222A	Q1	1	NPN Transistor	
CFR-25JB-220R	R16, R17	5	220 Resistor	Yageo
DF11-20DS-2DSA	J9	1	20-pin Receptacle	Hirose
22-12-2041	J11, J12	1	Connector for gyro board	Molex
104344-6	J13, J14	6	8-pin Single Header	AMP/Tyco
B3F-1000	S1	1	Reset Switch	Omron

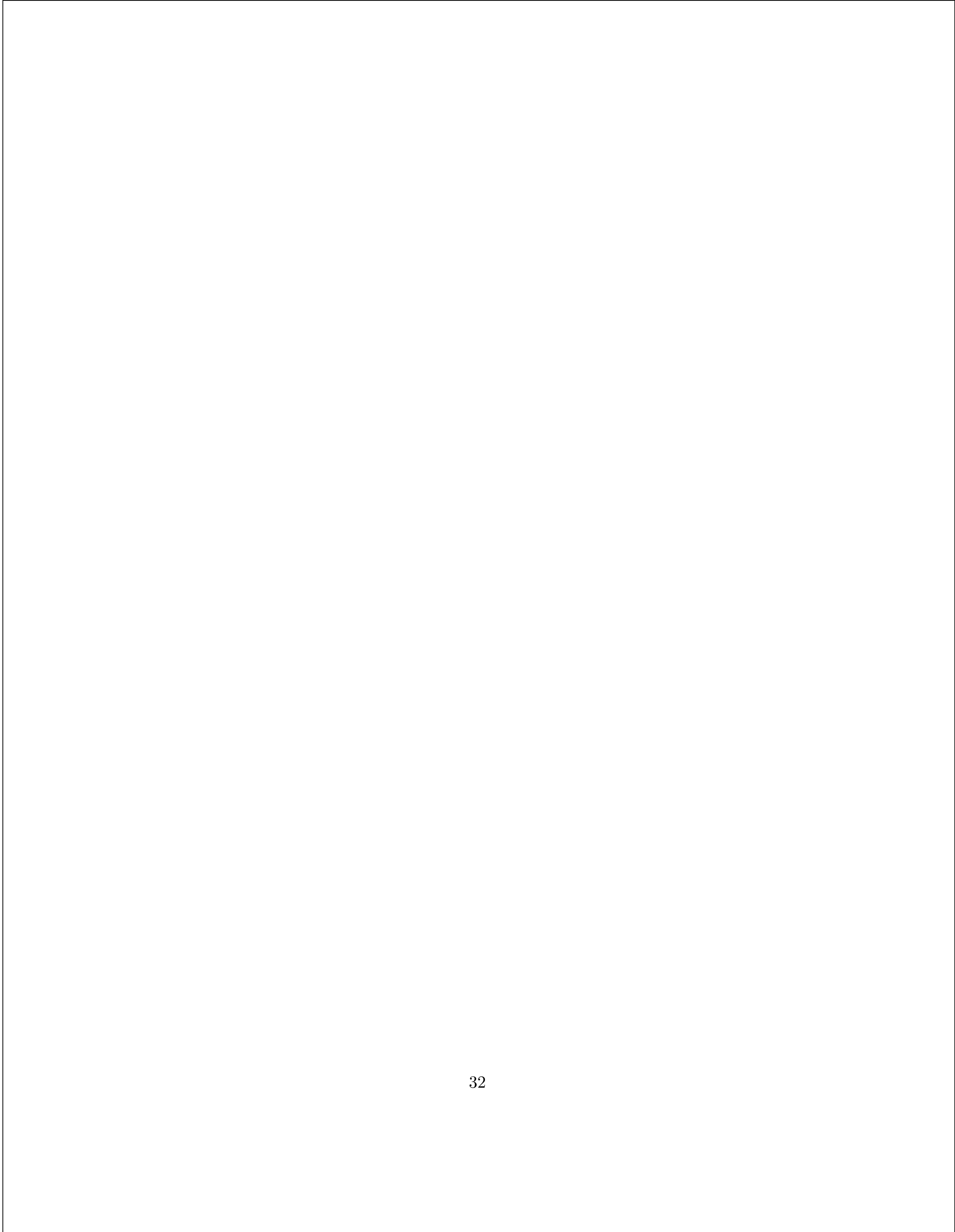
B CNC Code

B.1 Hovercraft Plate

```
N0   O17 ⋆ HOVERCRAFT PLATE
N1   T1M6 ⋆ #28 DRILL
N2   G0G40G80G90 X0Y0 E1S2500M3
N3   X0.95 Y-2.725
N4   Z1. H1
N5   G81G99 X0.95 Y-2.725 Z-0.2 R+0.1 F15.
N6   X-0.95 Y-2.725
N7   X-2.9 Y0
N8   X-2.4 Y1.15
N9   X-2.4 Y1.65
N10  X-1.55 Y2.147
N11  X-0.849 Y2.599
N12  X-0.849 Y0.901
N13  X-0.95 Y0.425
N14  X0.95 Y0.425
N15  X0.849 Y0.901
N16  X0.849 Y2.599
N17  X1.55 Y2.147
N18  X2.4 Y1.65
N19  X2.4 Y1.15
N20  X2.9 Y0
N21  G80G0Z0H0
N22  T2M6⋆ 3/16 ENDMILL
N23  G0G40G80G90 X0Y0 E1S2500M3
N24  X0 Y1.75
N25  Z1.H2
N26  G1 Z0.05 F25.
N27  Z-0.07 F5.
N28  X0.906 Y1.75 F15.
N29  G3 X0.906 Y1.75 I-0.906J0
N30  G1 X0.9
N31  G0 Z0H0
N32  M2
```

B.2 Lift Fan Holder

```
N10   O20 ⋆ LIFT FAN CLAMP FIXTURE
N20   T1M6 ⋆ DRILL
N30   G0G40G80G90X0Y0E3S3000M3
N40   X0.8475Y0.8475
N50   Z1.H1M8
N60   G81G99X0.8475Y0.8475Z-0.1R+0.5F5.
N70   Y-0.8475
N80   X-0.8475
N90   Y0.8475
N100  G0Z0G80M9H0
N110  Y7.
N120  T2M6⋆3/16 EM
N130  M0
N140  G0G40G80G90X0Y0E3S3000M3
N150  Z1.H2M8
N160  G1Z0.1F30.
N170  G1Z-0.1F5.
N180  X0.944Y0F10.
N190  G3X0.944Y0I-0.944J0
N200  X0.94Y0
N210  G0Z0M9H0
N220  Y7.
N230  M0
N240  M2
```



Relaunch The MVWT Hover Craft

B.3 Sensor Hat

```
N10   O18 ⋆ HOVERCRAFT HAT
N20   T1M6
N30   G0G40G80G90X0Y0E1S2500M3
N40   Z1.H1
N50   G81G99X0Y0Z-0.05R+0.1F5.
N60   G0G80Z3.
N70   X-4.25Y3.25
N80   G81G99X-4.25Y3.25Z-0.05R+0.1F5.
N90   G0G80Z3.
N100  Y-3.25
N110  G81G99Z-0.05R+0.1F5.X-4.25Y-3.25
N120  X4.25
N130  G0G80Z3.
N140  Y3.25
N150  G81G99Z-0.05R+0.1F5.X4.25Y3.25
N160  X2.4Y1.35
N170  X-2.4
N180  G0G80Z3.
N190  T2M6
N200  G0G40G80G90X0Y0E1S2500M3
N210  Z1.H2
N220  G81G99X0Y0Z-0.25R+0.1F5.
N230  G0G80Z0H0
N240  T3M6
N250  G0G40G80G90X0Y0E1S2500M3
N260  X-4.25Y3.25
N270  Z1.H3
N280  G81G99X-4.25Y3.25Z-0.2R+0.1F5.
N290  G0G80Z3.
N300  Y-3.25
N310  G81Z-0.2R+0.1X-4.25Y-3.25
N320  X4.25
N330  G0G80Z3.
N340  Y3.25
N350  G81Z-0.2R+0.1X4.25Y3.25
N360  X2.4Y1.35
N370  X-2.4
N380  G0G80Z0H0
N390  M0
```

```
N400  T4M6
N410  G0G40G80G90X0Y0E1S2500M3
N420  X-3.75Y3.125
N430  Z1.H4
N440  G81G99X-3.75Y3.125Z-0.2R+0.1F5.
N450  X-3.125
N460  X-2.5
N470  X-1.875
N480  X-1.25
N490  X-0.625
N500  X0
N510  X0.625
N520  X1.25
N530  X1.875
N540  X2.5
N550  X3.125
N560  X3.75
N570  X-3.75Y2.5
N580  X-3.125
N590  X-2.5
N600  X-1.875
N610  X-1.25
N620  X-0.625
N630  X0
N640  X0.625
N650  X1.25
N660  X1.875
N670  X2.5
N680  X3.125
N690  X3.75
N700  X-3.75Y1.875
N710  X-3.125
N720  X-2.5
N730  X-1.875
N740  X-1.25
N750  X-0.625
N760  X0
N770  X0.625
N780  X1.25
```

34

```
N790   X1.875
N800   X2.5
N810   X3.125
N820   X3.75
N830   X-3.75Y1.25
N840   X-3.125
N850   X-2.5
N860   X-1.875
N870   X-1.25
N880   X-0.625
N890   X0
N900   X0.625
N910   X1.25
N920   X1.875
N930   X2.5
N940   X3.125
N950   X3.75
N960   X-3.75Y0.625
N970   X-3.125
N980   X-2.5
N990   X-1.875
N1000  X-1.25
N1010  X-0.625
N1020  X0
N1030  X0.625
N1040  X1.25
N1050  X1.875
N1060  X2.5
N1070  X3.125
N1080  X3.75
N1090  G0G80Z1.
N1100  X-3.75Y0
N1110  G81G99X-3.75Y0Z-0.2R+0.1F5.
N1120  X-3.125
N1130  X-2.5
N1140  X-1.875
N1150  X-1.25
N1160  X-0.625
N1170  G0G80Z1.
```

```
N1180  X0.625
N1190  X0.625G81G99Y0Z-0.2R+0.1F5.
N1200  X1.25
N1210  X1.875
N1220  X2.5
N1230  X3.125
N1240  X3.75
N1250  X-3.75Y-0.625
N1260  X-3.125
N1270  X-2.5
N1280  X-1.875
N1290  X-1.25
N1300  X-0.625
N1310  X0
N1320  X0.625
N1330  X1.25
N1340  X1.875
N1350  X2.5
N1360  X3.125
N1370  X3.75
N1380  X-3.75Y-1.25
N1390  X-3.125
N1400  X-2.5
N1410  X-1.875
N1420  X-1.25
N1430  X-0.625
N1440  X0
N1450  X0.625
N1460  X1.25
N1470  X1.875
N1480  X2.5
N1490  X3.125
N1500  X3.75
N1510  X-3.75Y-1.875
N1520  X-3.125
N1530  X-2.5
N1540  X-1.875
N1550  X-1.25
N1560  X-0.625
```

36

```
N1570   X0
N1580   X0.625
N1590   X1.25
N1600   X1.875
N1610   X2.5
N1620   X3.125
N1630   X3.75
N1640   X-3.75Y-2.5
N1650   X-3.125
N1660   X-2.5
N1670   X-1.875
N1680   X-1.25
N1690   X-0.625
N1700   X0
N1710   X0.625
N1720   X1.25
N1730   X1.875
N1740   X2.5
N1750   X3.125
N1760   X3.75
N1770   X-3.75Y-3.125
N1780   X-3.125
N1790   X-2.5
N1800   X-1.875
N1810   X-1.25
N1820   X-0.625
N1830   X0
N1840   X0.625
N1850   X1.25
N1860   X1.875
N1870   X2.5
N1880   X3.125
N1890   X3.75
N1900   G0G80Z0H0
N1910   M2
```

C Testbed Materials

The type of carpet padding used on the testbed is called Berber Max. It is a dense, but smooth type of carpet padding. We found that Carousel Custom

Floors (on Green street in Pasadena) was very reliable and reasonable. Talk to Marv at (626) 795-8085.

D Data Acquisition

D.1 Coefficient of Friction

In the past the coefficient of friction was measured using this basic method: the vehicle was placed in a box and attached to the sides of the box with springs of known k. A force was applied by a human to start it into oscillation and the vision system was then used to measure the decay of the oscillation amplitude. That data was then analyzed to find the coefficient of kinetic friction.
To measure the coefficient of kinetic friction of the Bat, the Bat (wearing a hat) was placed on the testbed. The vision system was turned on. The Bat was given an initial velocity, and then vision system data was recorded. Analysis of the data was made easier by a Matlab program.

D.2 Moment of Inertia

The given value for the moment of inertia of the vehicle is approximate. It was approximated by modelling the vehicle in SolidWorks, a computer program used for solid modelling. Each component on the hovercraft was given a weight. From that model (which assumed constant density throughout each part) we approximated the hovercraft's moment of inertia.

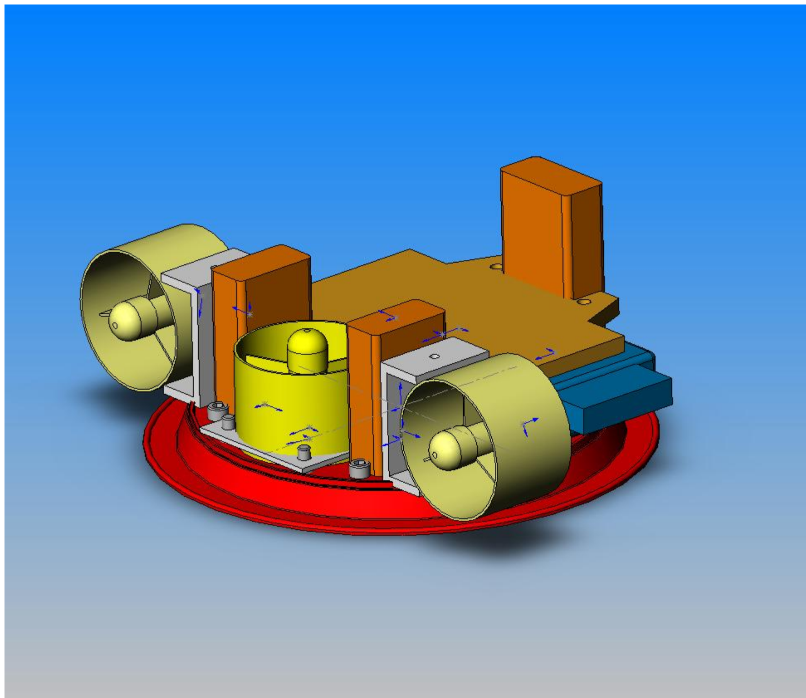


Figure 14: The Bat in SolidWorks

39

Control of the MVWT II Hovercraft

Some notes on the dynamics and a possible control approach.

Filename: `Hovercraft-dynamics.pdf`

Date: August 12, 2003

Authors: Stephen Waydo

Control of the MVWT II Hovercraft

S. Waydo

August 12, 2003

1 Model

The equations of motion can be written as

$$\begin{aligned} m\ddot{x} &= -\mu\dot{x} + f_x \\ m\ddot{y} &= -\mu\dot{y} + f_y \\ J\ddot{\theta} &= -\psi\dot{\theta} + T, \end{aligned}$$

where m and J are the mass and moment of inertia of the vehicle, μ and ψ are the linear and rotational friction coefficients, f_x and f_y are the x and y forces exerted by the fans, and T is the torque exerted by the fans. The physical parameters of the vehicles should be read in from a parameter file and should be allowed to vary from vehicle to vehicle. Reasonable parameters to start with are $m = 0.749$kg, $J = 0.0031$kg m^2, $\mu = 0.15$kg/s, and $\psi = 0.005$kg m^2/s.

In matrix form, the equations become:

$$\frac{d}{dt}\begin{bmatrix} x \\ y \\ \theta \\ \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & -\frac{\mu}{m} & 0 & 0 \\ 0 & 0 & 0 & 0 & -\frac{\mu}{m} & 0 \\ 0 & 0 & 0 & 0 & 0 & -\frac{\psi}{J} \end{bmatrix} \begin{bmatrix} x \\ y \\ \theta \\ \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} + \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ \frac{1}{m} & 0 & 0 \\ 0 & \frac{1}{m} & 0 \\ 0 & 0 & \frac{1}{J} \end{bmatrix} \begin{bmatrix} f_x \\ f_y \\ T \end{bmatrix}$$

2 Control

The controller takes the state $\mathbf{x} = (x, y, \theta, \dot{x}, \dot{y}, \dot{\theta})$ and a reference velocity $\mathbf{v}_{ref} = (u, v)$ and calculates the desired thrust F and torque T. As this happens on board the hovercraft, the simulator will need to perform this

1

step. It first transforms everything into "error coordinates" $(\theta_e, \dot{\theta}, \xi_1, \xi_2)$ that represent how far away the vehicle is from the desired velocity. The equations to do this are:

$$\begin{aligned} \theta_e &= \theta - \tan^{-1}(v/u) \\ \xi_1 &= \frac{\dot{x}u + \dot{y}v}{\|\mathbf{v}\|} - \|\mathbf{v}\| \\ \xi_2 &= \frac{-\dot{x}v + \dot{y}u}{\|\mathbf{v}\|} \end{aligned}$$

We then calculate the controls by

$$\begin{bmatrix} F \\ T \end{bmatrix} = -K \begin{bmatrix} \theta_e \\ \dot{\theta} \\ \xi_1 \\ \xi_2 \end{bmatrix}.$$

The 2×4 gain matrix K will be calculated offline and should be readable from a parameter file to allow us to test different controllers.

We now need to transform these control inputs to the forces f_r and f_r to be sent to the fans. Ideally,

$$\begin{aligned} f_r &= \frac{1}{2}(F + T + F_0) \\ f_l &= \frac{1}{2}(F - T + F_0), \end{aligned}$$

where F_0 is a "feed-forward" force that should also be readable from a parameter file. In reality, however, we need to remember that the fans have a minimum and maximum possible thrust. The minimum thrust is 0, i.e. the fans can't thrust backwards. The maximum force should be readable from a parameter file as well. The current estimate of this maximum force is $f_{max} = 0.706N$.

The forces in global coordinates to be used by the simulator are calculated from

$$\begin{aligned} f_x &= (f_r + f_l)\cos\theta \\ f_y &= (f_r + f_l)\sin\theta \\ T &= (f_r - f_l)r_f, \end{aligned}$$

where f_r and f_l are the right and left fan forces and r_f is the distance from the center of mass of the vehicle to the axis of the fan. The parameter r_f again should be readable from a file and is $r_f = 0.089m$.

2

3 Simulation

The simulator needs to take the current state and reference information, calculate the controls as above, then use them as the input to the differential equation given in the first section. This will give the derivative of the state which can be integrated forward for a short period of time to calculate the new state for the next step.

MVWT-II: The Second Generation Caltech Multi-Vehicle Wireless Testbed

Conference paper, summing up the results of the MVWT 2003 project.

Filename: `jin+04-acc.pdf`

Date: 2004

Authors: Zhipu Jin, Stephen Waydo, Elisabeth B. Wildanger, Michael Lammers, Hans Scholze, Peter Foley, David Held, Richard M. Murray

MVWT-II: The Second Generation Caltech Multi-vehicle Wireless Testbed

Zhipu Jin♮*, Stephen Waydo*, Elisabeth B. Wildanger*, Michael Lammers*,
Hans Scholze*, Peter Foley*, David Held**, Richard M. Murray***

***Abstract*— The Caltech Multi-Vehicle Wireless Testbed is an experimental platform for validating theoretical advances in multiple-vehicle coordination and cooperation, real-time networked control system, and distributed computation. This paper describes the design and development of an additional fleet of 12 second-generation vehicles. These vehicles are hovercrafts and designed to have lower mass and friction as well as smaller size than the first generation vehicles. These hovercrafts combined with the outdoor wireless testbed provide a perfect hardware platform for RoboFlag competition.**

I. INTRODUCTION

The Caltech Multi-Vehicle Wireless Testbed (MVWT) [1] is a tool for validating theoretical advances in multiple-vehicle coordination and cooperative control, networked control systems, real-time networking and high confidence distributed computation. The first-generation MVWT vehicles consist of a laptop computer mounted to a chassis that rolls on three omni-directional casters, with a pair of model aircraft ducted fans for actuation. A unique feature of this testbed is that the vehicles are underactuated and exhibit nonlinear second-order dynamics. These nontrivial dynamics force us to actively stabilize the vehicles while also trying to accomplish cooperative tasks in a manner analogous to the operation of Uninhabited Aerial Vehicles (UAV's). The MVWT vehicles run in a laboratory environment with localization achieved using an overhead camera system called the Lab Positioning System (LPS).

While the MVWT has proven useful in experimentally verifying theoretical results in nonlinear and cooperative control, several factors have limited its utility. First among them is the size of the vehicles relative to the laboratory space available. Experiments with more than 3 or 4 vehicles running on the floor at the same time have proved crowded and difficult to conduct. The use of an overhead vision system, while convenient from an implementation standpoint, has limited us to running the vehicles only within the laboratory and thus has prevented us from using larger spaces for more complex experiments, such as the RoboFlag competition [3].

This work was partially supported by DARPA program

♮Corresponding author, Electrical Engineering, California Institute of Technology, Pasadena, CA 91125, USA jzp@caltech.edu

*Graduate and undergraduate students of MVWT-II project, California Institute of Technology, Pasadena, CA 91125, USA

**Undergraduate student of MVWT-II SURF project, Massachusetts Institute of Technology, Cambridge, MA 02139, USA

***Professor of Mechanical Engineering, California Institute of Technology, Pasadena, CA 91125, USA

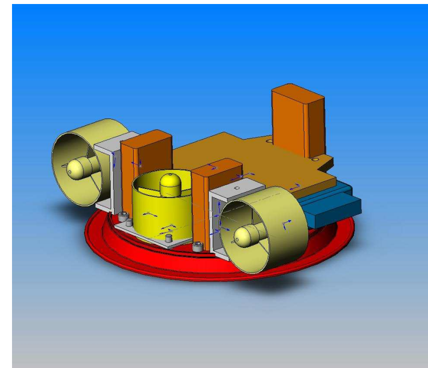


Fig. 1. Design of MVWT-II Hovercraft

During the summer of 2003, several graduate and undergraduate students worked together at Caltech to develop an additional fleet of 12 second-generation vehicles designed to address these issues which we call the "MVWT-II". The paper describes their work and is organized as follows: In Section 2, we briefly talk about the RoboFlag game and the design consideration of the second-generation vehicles. Section 3 lists the details of the hovercraft development including the mechanical design, embedded electrical system, and simple local controller. Section 4 gives the new hovercraft's parameters and performance measurements. Summary and future work are discussed in Section 5.

II. MOTIVATIONS AND DESIGN CONSIDERATION

The RoboFlag competition [3] is a powerful opportunity to use the MVWT as an experimental platform for research challenges in distributed control, sensor fusion, and human-centered control in a realtime dynamic environment. The RoboFlag game has been formulated at Cornell University over the last few years. It uses more complex scoring rules and a more specialized field than RoboCup. Roughly speaking, this game is based on "capture the flag". Two teams of 6 to 12 robots commanded by 1 or 2 human players play the game. The number of robots depends on the playing field size and the game complexity. Each team tries to attack the other team's territory, capture the other team's flag and bring it back to its own home zone. Because of the realtime dynamic environment, the complex offense/defense strategies, and the cooperation between robots, this game

helps us to understand some fundamental issues in realtime and high confidence distributed control.

Caltech and Cornell have worked on the RoboFlag competition together since 2001. In 2002, two groups of undergraduate students, team Pasadena and team Ithaca from Caltech and Cornell University respectively, joined the RoboFlag Summer Undergraduate Research Fellowship program (SURF). They spent ten weeks together developing a software system, fully completing the rules, and designing defence/offence strategies. At the end of the summer, they successfully competed with each other three times. Based on their experience and feedback, RoboFlag needs a larger, easily configurable playing field with faster, smaller vehicles to increase the feasibility and challenge. At the same time, vehicles should have second order dynamics so that the coordination control algorithms we develop will greatly rely on the advanced control techniques. With this motivation, a hovercraft design was developed in the summer of 2003 for MVWT-II vehicles shown in Fig. 1.

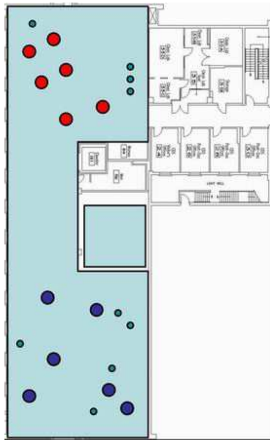


Fig. 2. Outdoor testbed on the roof. The small square in the middle is the original MVWT, the large area is the roof testbed.

There are couple of design considerations. First, the vehicles can run outdoor since developing a outdoor testbed will be a good solution to get a large playing filed. Fig. 2 is the outdoor testbed on the roof which is under development at Caltech. This will give us approximately 10 times more area than the indoor MVWT. The positioning system can be differential GPS or infrared positioning system (IR). Second, to reduce mass and friction and thus increase performance relative to the original MVWT vehicles, we need a very compact design which can enable us to reduce the mass greatly while improving the control authority (in terms of thrust/weight and torque/moment-of-inertia ratios) and reducing friction to nearly zero. Third, each vehicle should have an independent computation unit by which we can implement certain local controllers. Finally, we need to keep the cost of each hovercraft as low as possible since we will build a fleet with 12 vehicles. Also, how to make the assembly and maintenance job easy is another important issue for large number of vehicles.

III. Design and Implementation

A. Mechanical Design

The basic concept behind the new vehicles is the same as for the original MVWT vehicles. There is a forward-facing thrust fan on either side of the vehicle, and the vehicle is free to move in all directions. Beyond this, however, the two designs diverge dramatically. The previous MVWT vehicles were used on a smooth plastic surface, and the casters on the bottom were able to decrease the friction to a useable level. The new designs needed to be able to slide with very low friction over a surface that could be prepared on the roof of a building. Since the surface possibilities were quite limited, the only possible design was some sort of hovercraft.

a) Computation Unit: We selected Sharp Zaurus SL-5500 PDA as the computation unit. The dimensions of Zaurus is $74mm \times 138mm \times 18mm$. The weight is $212g$. Please refer to the electrical and computation issues discussed in Section III-B.

b) Skirtless Design: Considering the main design goals of low cost, simplicity, small size, stability, and ruggedness, the skirtless hovercraft design was motivated. While a skirt design could almost certainly give better performance, the labor consideration and financial constraint simply cut off this possibility. Also, after testing several prototypes with and without skirt, we found that the skirtless prototype worked very well.

c) Fans Selection and Location: While centrifugal fans are more efficient for the high-pressure, low-flow lift fan application, it was very hard to find one that was both powerful enough to lift the vehicle and compact enough to fit the space available. For this reason, an axial design was chosen, and the best fan for the application ended up being the same one used for thrust. The fans (GWS EDF-50), and the motor controllers (GWS ICS-100) were chosen with a focus on price, costing $15 and $20 per item, respectively. Each thrust fan gives approximately 0.7N thrust, which is enough to accelerate the vehicle quite quickly. To protect users and to keep foreign objects out of the fans, safety covers were fashioned from aluminum mesh and placed on the intakes to both thrust and lift fans. The dimensions of the Zaurus made overall layout rather difficult. We put the lift fan on one side of the plate and two thrust fans evenly to either side of the lift fan. The off-center lift fan keeps the Zaurus from hanging over the plate edges, protecting it in the event of inevitable collisions. Fig. 3 is the fan force map measured in MWVT lab.

d) Batteries Selection: The batteries used were rechargeable Lithium types, with one $1800mAh$ battery for the lift fan and two $950mAh$ batteries for thrust fans. The Lithium batteries are more expensive than NiMH or NiCd batteries, but they can hold much more power

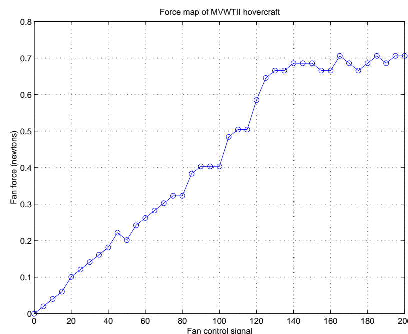


Fig. 3. Force map of the hovercraft fans

considering their size and weight. Since the batteries make up a substantial portion of the total vehicle weight, it was decided that the small size was worth the extra cost.

e) Balance: A major design issue for the hovercraft was balance, especially with the skirtless design and the off-center location of the fans. If the vehicle's base tilted, air would escape unevenly around the edges, and the craft would be propelled in the direction it tilted. Also it was originally thought that the off-center fan would cause uneven hovering height between front and back. These problems were largely avoided by strategically positioning batteries and by adding small brass weights to various bolts around the craft.

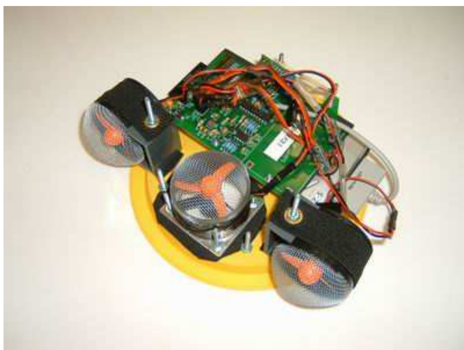


Fig. 4. MVWT-II hovercarft

The hovercraft was designed and modelled by using SolidWorks, which greatly eased vehicle layout. Also, it left us with very precise measurements of where to cut or drill. In order to drill holes quite accurately on the base plate and to decrease labor per vehicle, the base was drilled on a CNC machine. A jig was milled for drilling holes in the fan-mounting brackets. Since most of the time-consuming labor was automatic, the actual assembly was quite quick, consisting almost entirely of bolting parts together and connecting cables. Two people could build a vehicle from prefabricated parts in about 30 minutes. Altogether, the mechanical parts (including fans, speed controllers, and batteries) cost $300 per hovercraft.

B. Embedded System

The embedded computing system in the hovercraft is split into two sections. The Sharp Zaurus PDA provides wireless Ethernet connectivity and processing power to run simple local controllers onboard. The bridge between the Zaurus and the hovercraft fans is a custom interface board built around an Atmel micro-controller.

The Zaurus runs a custom version of Linux 2.4 on a 206MHz Strong ARM processor. It has 16MB FLASH and 64MB SDRAM. There are two main reasons to choose Zaurus. First, the Sharp Linux-based OS is easy to develop custom software with standard tools (there is a cross compiler of GCC available for the Strong ARM processor). Second, it is much cheaper than comparable specialized single-board computers, such as PC104. The Zaurus connects to the network in the vehicles lab through a wireless Ethernet card in the Compact Flash slot. The network connection allows the hovercraft to receive commands from more powerful controllers running on off board computers and communicate with other vehicles.

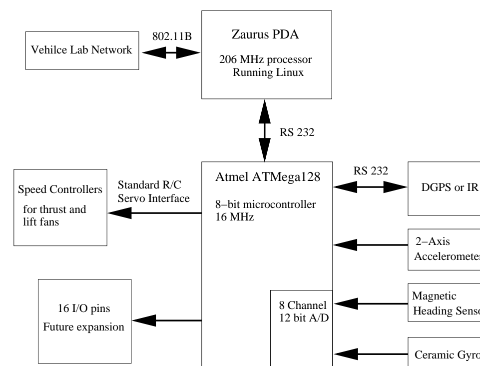


Fig. 5. Diagram of the embedded system

There is a limited amount of I/O available on the Zaurus, so all the low-level control of the hovercraft hardware is handled by an Atmel ATMega128 micro-controller. The Zaurus communicates with the micro-controller through one of ATMega's RS-232 serial ports.

The ATMega128 is an 8-bit RISC micro-controller running at 16MHz. It includes an 8-channel 12-bit A/D converter, 8 external interrupts, two RS-232 ports, and 4 timers/counters. The ATMega128 is incorporated into a custom PCB with a power supply, a ceramic gyro, a two-axis accelerometer, a magnetic heading sensor, a external module reserved for differential GPS or IR system, three outputs

to the lift and thrust fan speed controllers, and 16 general purpose I/O lines for future expansion. The ATMega128 reads the sensors continuously and sends packets of sensor data back to the Zaurus at approximately 50 Hz. It also receives fan-speed control commands from the Zaurus and sets the speed controller outputs appropriately.

For developing the software of the ATMega128 we used the free WinAVR set of tools and Atmel's AVRISP programmer. WinAVR includes a port of gcc and a cross-compiler for the AVR architecture which runs under Microsoft Windows OS. The AVRISP is a simple programmer that allows in-circuit programming of the Atmel through the 6-pin ISP connector.

The gyro we are using is the Tokin CG-16D ceramic gyro, which is mounted vertically on the interface board, a product of the Coriolis effect on an internal vibrating ceramic column printed with electrodes. The output of the gyro is 1.1 mV/deg/sec $\pm 20\%$ referenced around 2.4V. The offset at zero angular speed can vary as much as $\pm$300mV. A simple differential amplifier is built to compensate this and modify the gyro output such that it can be easily processed by A/D converter. The gyro is linear in the range ± 840 degrees/s according to our tests and the Atmel can sample the gyro outputs with a frequency up to 100 Hz.

The accelerometer is an Analog Devices ADXL202 two-axis MEMS accelerometer which measures acceleration in the forward/backward and side-to-side directions. Its output is a pulse-width modulated (PWM) signal which is measured with the ATMega128 timer/counters. Every $\pm$1g acceleration corresponds to approximately $\pm 12.5\%$ duty cycle. The ADXL202 can measure $\pm$2g acceleration with a sample frequency of the accelerometer up to 100 Hz.

Absolute heading is measured by a heading sensor which is composed by a Honeywell HMC1052 two-axis magneto-resistive sensor, two A/D converter, and a compassing circuit. This sensor provides two analog outputs and can be used to calculate the heading with about 1.5 degree orientation accuracy. Currently, the interface board only returns the raw A/D data from each axis. The functions to calibrate the compass and calculate the orientation run on the Zaurus.

Absolute position is read from the module reserved for the differential GPS or IR system. The module interfaces through a serial port, which is connected to the second UART port on the ATMega128.

The lift and thrust fan speed controllers on the hovercraft are controlled through the standard R/C servo interface. The ATMega128 generates the PWM signal at 50Hz to set the speed of each fan. The interface board's electrical power comes from the battery of the lift fan.

C. Local Controller Design

The MVWT-II hovercraft has the same propulsion principle as the first-generation MVWT vehicle. According to [1], the equations of the hovercraft motion can be written as

$$\begin{aligned} m\ddot{x} &= -\mu\dot{x} + (F_R + F_L)\cos\theta \\ m\ddot{y} &= -\mu\dot{y} + (F_R + F_L)\sin\theta \\ J\ddot{\theta} &= -\psi\dot{\theta} + (F_R - F_L)r_f \end{aligned}$$

These equations are derived by observation from the simple schematic of the vehicle shown in Fig. 6 and are the same as motion equations of first-generation vehicle.

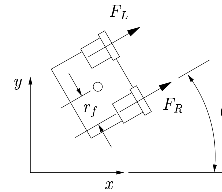


Fig. 6. Schematic of hovercraft. The coordinate frame is inertial and the forces F_L and F_R are applied at the fan axes.

These equations include four physical parameters: the mass m, the moment of inertia J, and the linear and rotational viscous friction coefficients μ and ψ. Linear and rotational friction coefficients μ and ψ depend on the the lift fan thrust and the surface of the field. This makes the the lift fan thrust another possible control force.

For the RoboFlag game, the local controller takes the state $X = (x, \dot{x}, y, \dot{y}, \theta, \dot{\theta})$ and a reference velocity $V = (v_x, v_y)$, calculates the desired thrust $F = F_R + F_L$ and torque $T = F_R - F_L$. (We assume the lift fan thrust is constant.) The equations to transfer everything into error coordinates which is represented by $(\theta_e, \dot{\theta}, \xi_1, \xi_2)$ are

$$\begin{aligned} \theta_e &= \theta - \tan^{-1}(v_y/v_x) \\ \xi_1 &= \frac{\dot{x}v_x + \dot{y}v_y}{\|V\|} - \|V\| \\ \xi_2 &= \frac{-\dot{x}v_y + \dot{y}v_x}{\|V\|} \end{aligned}$$

and the control law is

$$\begin{bmatrix} F \\ T \end{bmatrix} = -K \begin{bmatrix} \theta_e \\ \dot{\theta} \\ \xi_1 \\ \xi_2 \end{bmatrix}.$$

The 2×4 gain matrix K is calculated off-line for different controllers and the fan thrusts are

$$\begin{aligned} F_R &= 1/2(F + T) + F_0 \\ F_L &= 1/2(F - T) + F_0 \end{aligned}$$

where F_0 is a "feed-forward" force that should be re-configurable by on-line software or parameter files. In reality, however, the fans are saturation units which have the minimum thrust and maximum thrust. The minimum thrust is 0N, i.e. the fan cannot thrust thrust backwards. The maximum thrust is about 0.7N.

TABLE I
PARAMETERS OF FIRST-GENERATION VEHICLE AND MVWT-II HOVERCRAFT

Vehicle Parameters	First-generation vehicle	MVWT-II hovercraft
Plane Dimensions	rectangle with 37.00 cm ×27.00 cm	round disk with diameter 20.00 cm
Height	18.00cm	7.50 cm
Mass	5.05 ± 0.05 kg	0.75kg
Moment of Inertia	0.05 kg m^2	0.00316 kg m^2
Distance between Thrust Fans	24.6 cm	17.8 cm
Maximum Fan Thrust	5.1 N	0.7N
Linear Friction on Indoor Testbed	4.5 kg/s	0.15 kg/s with maximum lift fan thrust
Rotational Friction on Indoor Testbed	0.064 kg m^2/s	0.005 kg m^2/s with maximum lift fan thrust
Maximum Speed	1.2 m/s	$>$ 2.5 m/s
Unit Cost	More than $2000	Less than $860
Lift Fan Battery Lifetime	$20 - 25$ minutes	$35 - 40$ minutes
Hover Height	N/A	$< 2mm$
Computation Unit	Dell Latitude L400	Sharp Zaurus SL-5500
Processor	Intel 700 MHz	Strong ARM 206 MHz
Wireless Network	Yes	Yes
Onboard Sensor	Gyro	Gyro, accelerometer, and heading sensor
Positioning System	Overhead cameras	Overhead cameras, DGPS or IR

IV. PARAMETERS AND EXPERIMENTS

The final hovercraft design resulted in an effective yet low cost device. It was impossible to achieve a high hover height with the low cost limit in ten weeks, so the hovercraft were unable to operate on rough surfaces such as lawn or pavement. However, they are still effective for coordinated control tests on a relative smooth field such as MVWT, gymnasium, and building roof covered by BerberMax carpet padding.

Parameters of MVWT-II hovercraft and first-generation vehicle are listed in Table I. We conducted some performance tests on Caltech MVWT to obtain these data. The overhead vision system provided the necessary position for these experiments.

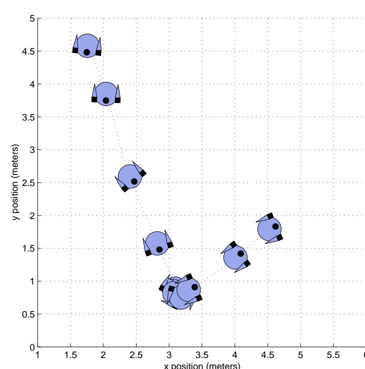


Fig. 7. A simple experiment of the hovercraft

Fig. 7 shows a top view of a simple experiment. The hovercraft goes straight and then makes a left turn. Fig. 8

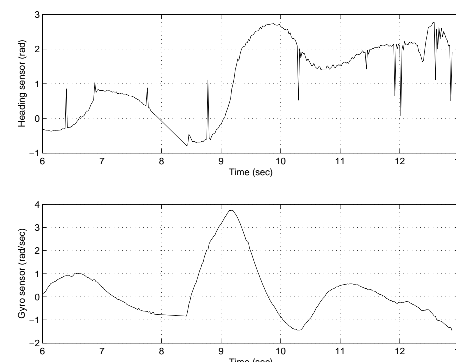


Fig. 8. Heading sensor data and gyro data

is the heading sensor data and gyro data. In this experiment, we use a simple PID controller for the heading and also the lift fan's output is controlled as a "brake" to slow down the hovercraft before it turns.

The heading sensor is based on a two-axis magnetic filed sensor and it easily suffers from the nearby magnetized devices such as CRT monitors, power supplies, etc. So it's not a good idea to use it in a small and crowded lab, but it can provide good orientation data in a open space such as a gymnasium or a outdoor field.

V. SUMMARY AND FUTURE WORKS

The MVWT-II vehicle faced and overcame many design challenges. These included the requirements of low weight, relative low cost and simplicity. The small size of the Zaurus greatly contributed to the compact size of the vehicle. A larger computational platform would have required greater

power and greater lift strength. The lightweight solution facilitated the advantageous downsizing of the hovercraft towards lower power consumption and a smaller lift fan.

There are some design challenges that the MVWT-II vehicle has not yet overcome, but its flexibility will allow for continual development. For greatest flexibility, the MVWT-II vehicle provides the DGPS or IR interface. This will allow use of the MVWT-II vehicle outdoors or in other testbeds. The rooftop testbed has not reached its full potential for at this time, for there are not yet hovercraft that can make full use of it. The flexibility of the MVWT-II vehicle will make it a functional testbed on which to test and develop new algorithms for distributed control. The flexibility in adding new sensors such as GPS module, Sonic ranger, etc, onto the Atmel interface board will support new developments on the MVWT-II vehicle for varied purposes. Also, the flexibility of the MVWT-II vehicle makes each vehicle a small platform on which to effectively test and develop new control algorithms for non-linear systems.

Currently, the local controller and low level software is under testing. Once the current fleet of vehicles has been proven experimentally, we will construct an additional group of 12 vehicles to enable cooperative control experiments, such as RoboFlag, using up to 24 total vehicles. Other future work includes implementing the Computation and Control Language (CCL) [6] on the Zaurus PDAs to complement work on using CCL for cooperative control currently in progress. We will also develop a system by which trajectories can be computed on a server using the Nonlinear Trajectory Generation (NTG) software developed at Caltech [7]. This server will communicate to the vehicles using the wireless network.

References

[1] T. Chung, L. Cremean, W. B. Dunbar, Z. Jin, E. Klavins, D. Moore, A. Tiwari, D. van Gogh, and S. Waydo. A platform for cooperative and coordinated control of multiple vehicles: The Caltech Multi-Vehicle Wireless Testbed. In *Proc. of the 3rd Conference on Cooperative Control and Optimization*, Dec. 2002.

[2] L. Cremean, W. Dunbar, D. van Gogh, J. Hickey, E. Klavins, J. Meltzer, and R. M. Murray. The caltech multi-vehicle wireless testbed. In *Proc. of the 41st IEEE Conference on Decision and Control*, volume 1, pages 86–88, 2002.

[3] Raffaello D'Andrea and Richard M. Murray. The RoboFlag competitions. In *Proc. of the American Control Conference*, 2003.

[4] R. D'Andrea and R. M. Murray. The roboflag testbed. In *Proc. of the American Control Conference*, pages 656 – 660, 2003.

[5] J. P. Desai, J. P. Ostrowski, and V. Kumar. Modeling and control of formations of nonholonomic mobile robots. *IEEE Transactions on Robotics and Automation*, 17(6):905–908, 2001.

[6] Eric Klavins. A language for modeling and programming cooperative control systems. In *Proceedings of the International Conference on Robotics and Automation*, 2004. To Appear.

[7] M.B. Milam. *Real-Time Optimal Trajectory Generation for Constrained Dynamical Systems.* Ph.d thesis, California Institute of Technology, 2003.

Addendum to MVWTII Hovercraft Documentation (MVWT 2003) by Wildanger, et. al.

MVWT status in early 2008. Detailed information on hardware. Describes LQR control approach, which could not be implemented due to hardware failure.

Filename: `MVWT2Hovercraft_Addendum.pdf`

Date: 21 Mar. 2008

Authors: Nam Nguyen

Addendum to MVWTII Hovercraft Documentation (MVWT 2003) by Wildanger, et. al.

Nam Nguyen

Control and Dynamical Systems
California Institute of Technology
1200 E. California Blvd.
Pasadena, CA 91125

21 Mar. 2008

Relaunch The MVWT Hover Craft

Table of Contents

I. Pertinent information omitted in original documentation
  1. Mechanical information
    i. Motor data
    ii. Battery information
    iii. Measuring moment of inertia about center and side fan force at a point
    iv. Hats for the vision system
  2. Electronics information
    i. Electrical connections
    ii. Gyro calibration
  3. Software information
II. Information specific to 2008 CDS110b hovercraft restoration and control project
  1. Project description
  2. Hardware Status
    i. Old hardware
      A) PC boards
      B) Zaurus PDAs
      C) Hovercraft Chasses
      D) Batteries
      E) Vision system
    ii. New hardware
      A) Hovercraft skirt design
      B) Test cables
      C) The new batteries
  3. Software
    i. Trajectory generation and simulation
    ii. Control code
    iii. Test code
    iv. Unwritten code
  4. Control
    i. Control scheme description
    ii. Note about constraints
  5. Project results

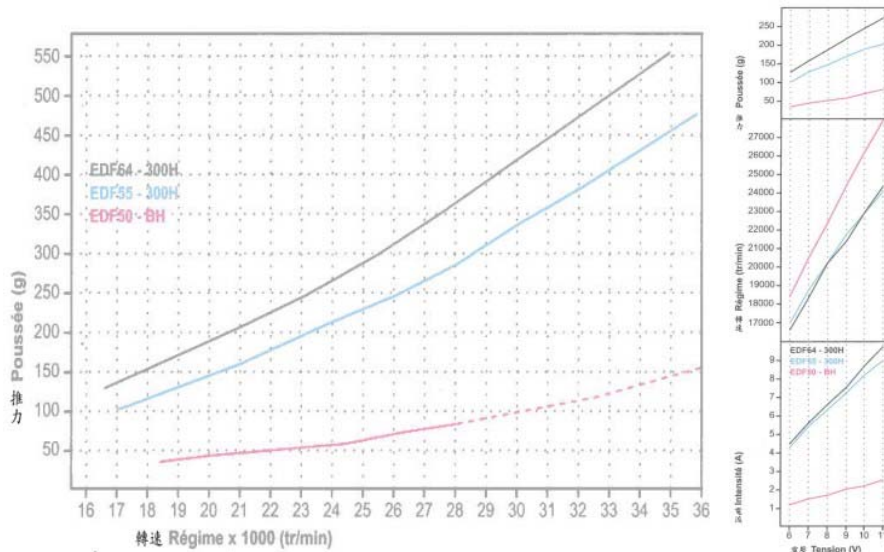
I. Pertinent information omitted in original documentation

1. Mechanical information

i. Motor data

There are actually three different ducted fan models being used, not one. Two of the models, EDF-50 and EDF-50AH, are nearly indistinguishable, and are used for the side fans as well as the lift fan on older manifestations of the hovercraft (design outlined in the original documentation). The third, stronger model, EDF-64-300H, is currently being used as the lift fan. Information is available on GWS's website, http://www.gws.com.tw/English/english.htm . The EDF-50 models do not seem to be supported anymore. Data for the EDF-64-300H is reproduced below:

The following test datas have been noted all by applying continual high RPMs for 5 minutes respectively.							
GW/EDF-64(EP2540X6)	MOTOR : EM300H		Weight : 73g				
ROTOR	Volts (v)	Amps (A)	Thrust		Power (w)	Efficiency	
			(g)	(oz)		(g/w)	(oz/kw)
EP2540X6	7.2	4.6	133	4.69	33.12	4.02	142
EP2540X6	8.4	5.7	167	5.89	47.88	3.49	123
EP2540X6	9.6	6.6	190	6.7	63.36	3	106
EP2540X6	10.8	7.7	218	7.69	83.16	2.62	92



The lab has a moderate supply of new EDF-50 fans (pictured below):

ii. *Battery information*

Important fact: Do not overcharge or over-discharge the batteries. Some battery models are not protected, meaning that if they are charged or discharged too much, they can be rendered inoperable. According to Duralite's website, their current models are protected.

The chargers require 12V input. This can be achieved by hooking it up to any of the various adjustable power supplies in the lab and setting the power supply to 12V while making sure it is not current-limited. When the green LEDs are on, the batteries are charging. When they are finished, the lights go out. As is evident, connect the batteries' yellow connector to the charger's.

iii. *Measuring moment of inertia about center and side fan force at a point*

One method of measuring the hovercraft's moment of inertia about its center is to use the turntable and a calibrated gyro (see gyro calibration). First, the moment of inertia of the turntable must be calculated analytically. The hovercraft is spun up using one of the side fans while the gyro records angular velocity with respect to time. In order to eliminate the nonlinearity caused by friction in the turntable, consider only the low angular velocity regime which is nearly linear. Thus, a weak fan force should be used to make the acceleration slow in order to get as much data from the linear regime as possible. One then repeats the process with known weights added symmetrically to the turntable. The result is two linear equations with two unknowns, moment of inertia of the hovercraft and force of the fan. (The slope of the graph of angular velocity vs time is torque from the fan divided by moment of inertia.) Because the fan can also be characterized, the fan force measurement is a convenient side effect if one uses a linear model for the fan force, which is not too unreasonable.
The turntable is shown below:

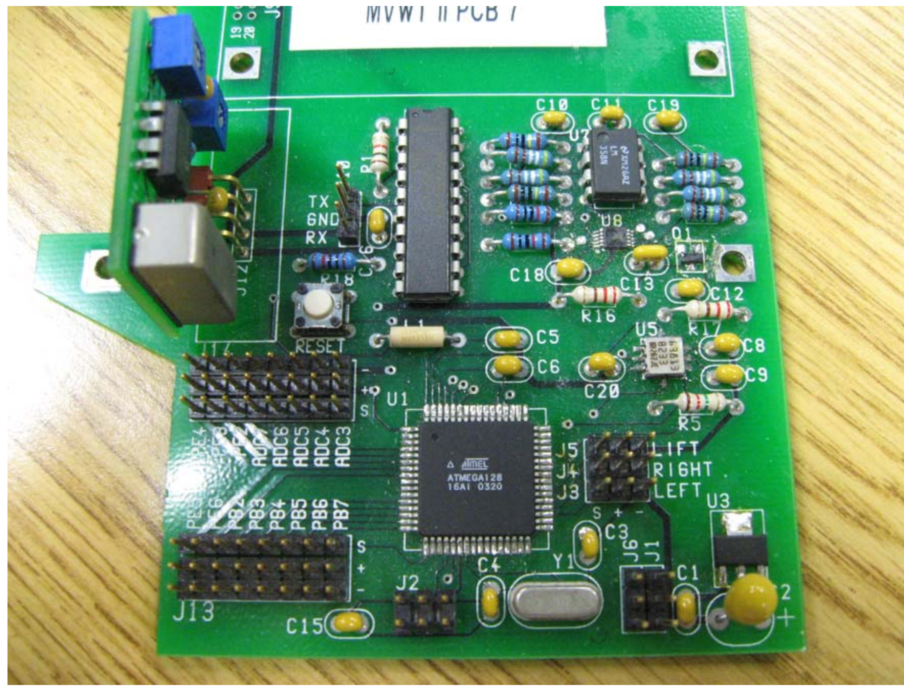
iv. *Hats for the vision system*

A powerpoint file called Hats.ppt contains the hovercraft hat images that can be printed and mounted.

2. Electronics information

i. Electrical connections

The PC board is pictured below:

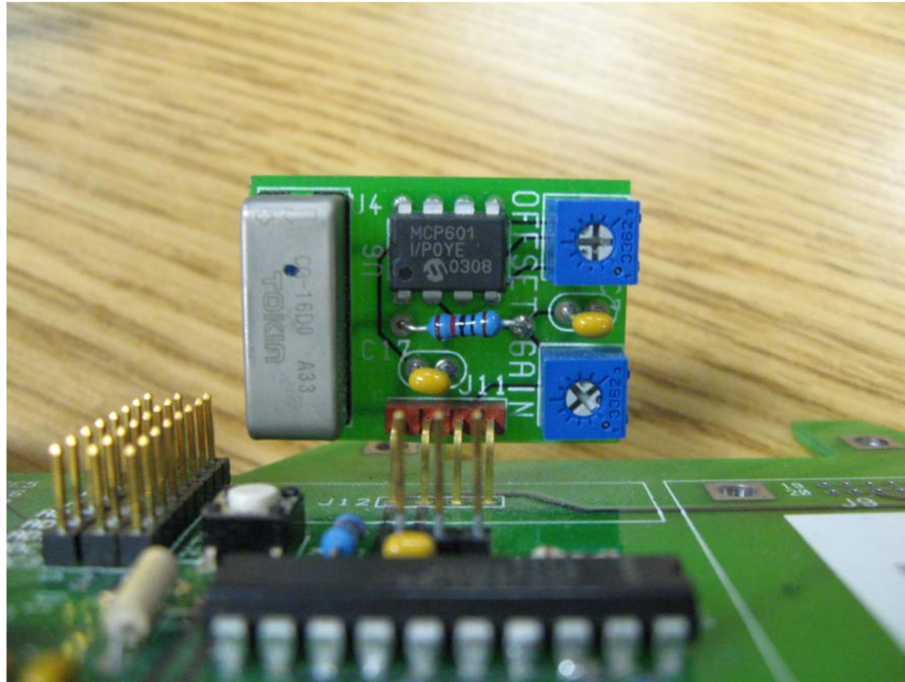


Original 3 battery configuration:

Connect the cable from the Zaurus serial port adapter to TX/GND/RX on the board. Connect the leg of the Y on each motor controller to each fan. Connect the signal cables from the three motor controllers to J3, J4, and J5, marked left, right, and lift, respectively. Connect each of the side fan motor controller power cables to a small battery pack. Connect the large battery pack to J6 with the two wires facing the edge of the board and the unused pin being nearer to the center of the board. Finally, connect the power cable from the lift fan motor controller to J1 in the same way.

ii. Gyro calibration

There are two potentiometers for the gyro on the small PC board, which is pictured below:



As labeled on the board, one controls the offset and one controls the gain on the op-amp amplifying the gyro signal. Therefore, calibration depends on these settings and must be performed every time the settings are changed.

The offset and gain must first be setup while observing the real-time output of the gyro using a program such as "gyrotest." One must first determine an upper bound for angular speed the gyro will be expected to sense. One then modifies the offset potentiometer until the output range is centered at 0 velocity. Finally, the gain must be changed until the output range extends up to the chosen bounds for angular velocity. This maximizes the resolution of the digital output given a maximum angular speed.

One way to characterize the gyro is to use the vision system in conjunction with the turntable in the lab. One can spin the hovercraft using one side fan to steady state (making sure the steady state angular velocity is within the gyro's measurement range) and then measure the angular velocity using the vision system as well as the value outputted by the gyro. The ratio between these two numbers is then the conversion factor between gyro output and angular velocity.

3. Software information

The SD card mentioned in the original documentation is pictured below:

As of the time of writing, the SD card is located in the hovercraft used for the 2008 CDS110b MVWTII Hovercraft Restoration and Control Project. The contents of the SD card should be organized in the same location as the body of MVWTII documentation.

Moreover, various test codes remain from the original project:
acceltest.c
gyrotest.c
headingtest.c
serialtest.c
timertest.c
thrust_test.c
visiontest.c

All tests should be cross-compiled and run remotely on the Zaurus. Rudimentary control codes were also written, such as controller.c, democon.c, and democtlr.c.

II. Information specific to 2008 CDS110b hovercraft restoration and control project

1. Project description

Project team: Chris Schantz, Vanessa Carson, Kenny Oslund, Nam Nguyen

Project goals:

- Restore at least one hovercraft unit to working order
- Ensure sensors (both onboard and vision system) can be read and useful data can be extracted
- Ensure communication interfaces are functional
- Implement an LQR controller in code
- Implement a Kalman Filter in code
- Take data and show that the hovercraft can be controlled stably, robustly, and with good performance
- Make improvements to existing design as seen fit
- Designate unusable equipment
- Research undocumented items

Unfulfilled goals:

- We were not able to control the hovercraft in a straight line for an entire run, but

we were able to make it run in a straight line with self-correction clearly visible.

- Kalman filter was written but not tested again due to PC board failure
- Accelerometers remain unused because they are malfunctional or unreliable
- Compass unused because of reliability concern (abundance of computers)

2. Hardware Status

i. Old hardware

A) PC boards

All seem to carry some flaw. The one used for this project had the least number of flaws until it was no longer able to send signals to the fans. Most have onboard sensors that are completely defunct. There is another board with impaired gyro adjustability which is suspected to be due to worn potentiometers as no signals are received when the potentiometer for gain is set to certain bands.

B) Zaurus PDAs

At least three communicate with the wireless network and are ready to accept software, including the one used for this project. That unit is fully setup. Also, there is one labeled “broken”.

C) Hovercraft Chasses

The hovercraft chasses are in various states of disarray. Most are in pieces and a few are capable of mounting fans, the zaurus, and the PC board. There are many plates which are remnants of older versions of the hovercraft, some still with the weaker side fan motors mounted as lift fans off center.

D) Batteries

Most of the yellow Duralite batteries for the hovercraft in the lab are inoperable and have been separated from the working ones.

Moreover, an alternate 2 battery configuration was developed which is nearly the same as the original 3 battery configuration except one small battery powers both side fans using a Y-cable we constructed.

E) Vision system

It seems that we can only retrieve data from half of the vision system area even though the display monitor shows the cameras capturing images from across the entire area. This half is the left half of the vision area as one walks into the lab.

ii. New hardware

A) Hovercraft skirt design

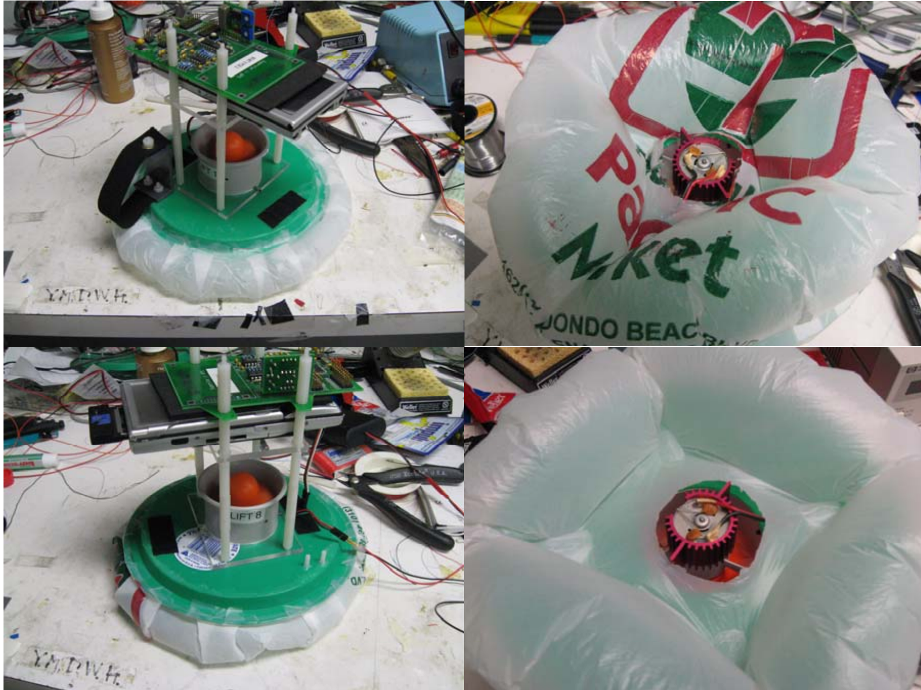
The original project by Wildanger, et. al, used plastic tubing instead of a skirt because it was decided that skirts would be too expensive and that plastic tubing would suffice. It is evident from one of the chasses that some experimentation was done with using plastic grocery bags as skirts. We determined that the performance increase, namely stability and decreased friction, was well worth half an hour and half a plastic bag.

Our skirt design is as follows. A flat circle is cut from a plastic grocery bag that is approximately 3 inches larger in radius than the chassis plate. Four small pieces of scotch tape are looped (so that they become essentially double sided tape) and placed radially at moderate distance from the center. The plastic circle is placed concentrically on top, sticking to the four pieces of tape without the center becoming taut. The sides of the circle are then methodically folded down and taped to the One should first tape down the 0, 90, 180, and 270 degree positions, then the 45, 135, 225, and 315 degree positions. Finally, the eight sections in between are taped down to completely seal the skirt. In addition, the folds made in the plastic should be taped to maintain the shape of the skirt and prevent it from ballooning upward (and making the center of gravity even higher). The last step is to run the lift fan on the test bench while cutting out a hole in the center of the plastic circle about the size of the lift fan motor's heatsink. The result is four balloons of air at the 45, 135, 225, and 315 degree positions and a strong air jet in between.

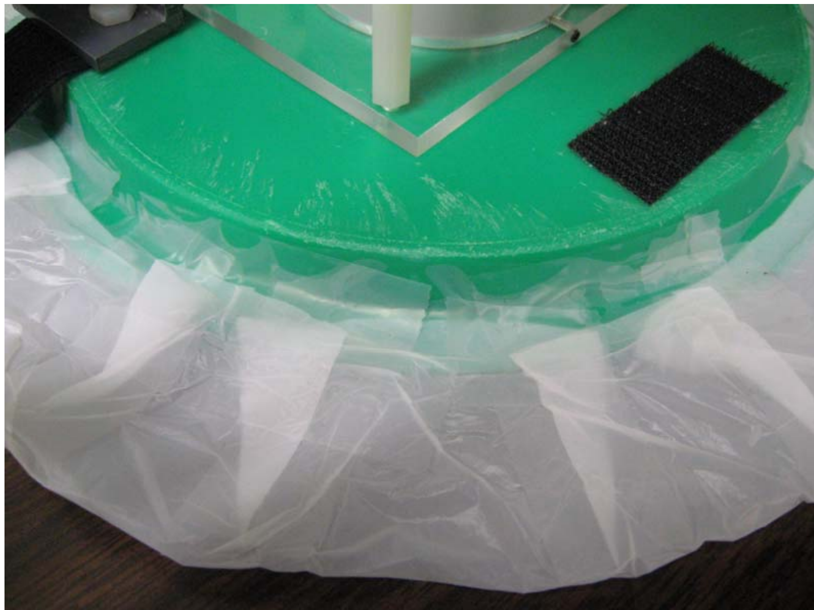
In general, we found that a vent hole at the bottom is necessary for two reasons. First, if too much pressure is put on the lift fan, air will merely be back-driven through it. Second, the air jet is the primary means by which the hovercraft is held up just as in the skirtless design. However, just enough air must flow into the four balloons to maintain pressure such that they can perform the job of stabilizing the hovercraft. Moreover, if the skirt is too small, it is unable to provide enough of a moment to counteract the highly unstable center of gravity. On the other hand, if it is too large, it may balloon upwards or lose its stabilizing ability.

Two skirt sizes of this design were fabricated, but we were unable to test any of the fabricated skirts under read conditions before the PC board malfunctioned. However, the skirts were tested using the power supplies on the test bench. The larger of the two was closer to the optimal size, as it provided stronger stabilization with respect to the unstable high center of gravity. The smaller skirt would cause the chassis to fall forward and bounce back when the hovercraft was pushed forward and allowed to coast. The

small (top) and large (bottom) skirts are pictured below:

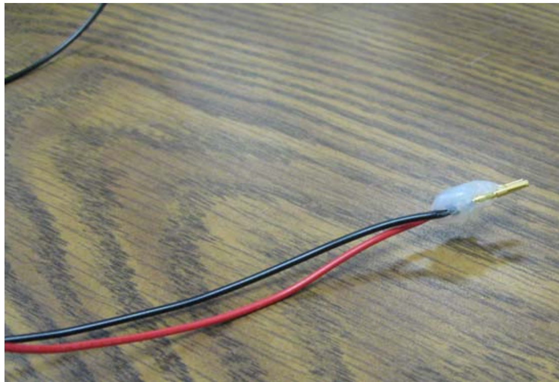


Close-up of the large skirt construction:



B) Test cables

Cables were fabricated as a simple way to connect fan motors to the power supplies (pictured below). With three power supplies on a test bench, one can power all three fans on a hovercraft. However, note that the maximum power that can be delivered to the fans this way is less than with the batteries. The fans can be expected to be more powerful when using the batteries.



C) The new batteries

New lithium-polymer batteries which are lighter and stronger than the lithium-ion Duralite batteries (but quite expensive) were meant to serve as an upgrade, but we unfortunately did not realize the danger of over-discharging until after they were no longer usable.

3. Software

i. Trajectory generation and simulation

The collection of files with main MATLAB script Sim_GUI written by Chris Schantz can be used to generate trajectories as well as run simulations.

ii. Control code

The main code consists of two components: estimation and control software (ECS) and local control software (LCS).

The former consists of:
MvwtFollower.h: Header file for estimator and controller class
MvwtFollower.c: Implementation of estimator and controller class
Hover.c: Instantiation of the above class. Should be cross-compiled for ARM/linux and

executed on Zaurus remotely by SSH.

And the latter consists of:

LocalController.c: main program and implementation of inner loop controller

iii. Test code

We modified thrust_test.c to make it better suited to testing the hovercraft rather than just the fans.

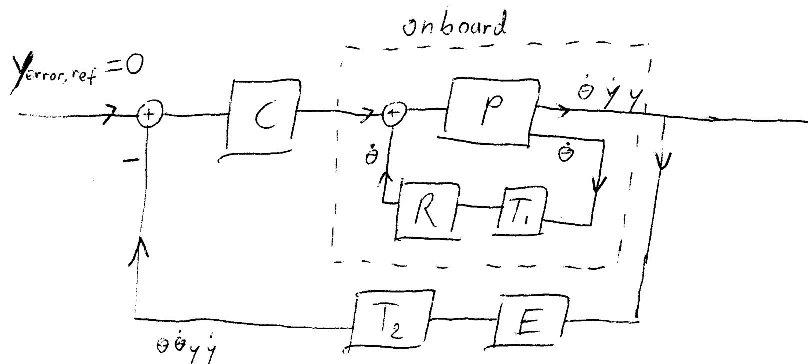
iv. Unwritten code

Automatic calibration software remains unimplemented in the bathw library.

4. Control

i. Control scheme description

The following diagram summarizes the control scheme employed:



Onboard transfer function

$$B = \frac{P}{1 + PRT_1}$$

Transfer function from $y_{error,ref}$ to output

$$\frac{CB}{1+ET_2CB} = \frac{\frac{CP}{1+PRT_1}}{1+\frac{ET_2CP}{1+PRT_1}} = \boxed{\frac{CP}{1+PRT_1+ET_2CP}}$$

Matrix transfer functions (multiple input multiple output):
C = outer loop controller transfer function
P = process transfer function
R = inner loop controller transfer function
E = estimator transfer function
T1 = inner loop time delay (dependent on zaurus computation time)
T2 = outer loop time delay (dependent on vision system, network communication, and remote computation time)

The main reason for using the inner-outer loop scheme is to avoid putting too much computational strain on the Zaurus and having to deal with high time delays. If communication with a remote computer is faster than local computation, it makes sense to move those computations off board. When splitting tasks among local and remote controllers, one allocates the ones that need to be performed the fastest on the local controller. Therefore, angular velocity dampening must run on the inner (local) controller because in order for the dampening to have an effect on control signals, it must run at a much higher frequency (at least an order of magnitude fast).

The inner loop handles constantly forcing angular velocity to zero with the outer loop the same as in a basic control scheme. If a simple proportional gain controller is used, this is a sort of artificial friction whose effects are conveniently limited to the rotational physics without affecting the translational physics. More importantly, it can be accounted for in the dynamics by simply augmenting the angular friction coefficient. Although angular velocity dampening is very effective for stabilization, there is a penalty paid in performance because dampening inherently slows down the hovercraft rotationally.

Thus, the execution remains simple. With the inner loop employing an LQR controller based on the linearized dynamics, the outer loop (also utilizing an LQR controller) continues about its usual business forcing lateral displacement to zero. In the general case beyond a simple straight line path, we require another superseding outer loop to control the forward velocity.

Moreover, we use a Kalman Filter to estimate the full state from sensing only theta dot onboard while sensing the full state x, y, theta, and theta dot using the vision system. Theta is estimated from the gyro's reading of theta dot and fused with the vision system's reading of theta.

ii. Note about constraints

Although we enjoy the simplicity of the LQR controller, we are limited by the

fact that it cannot handle input saturation. As the side fans cannot thrust backwards, we have an asymmetric input saturation about zero. If it were symmetric, simply avoiding the saturations would be much easier. However, with the controller's current manifestation, we must let the controller's robustness deal with this shortcoming. At the most basic level, we must rely entirely on friction to decelerate (of which hovercrafts have relatively little). It is conceivable that one can turn off the lift fan to brake instantly, but this control scheme is far beyond the scope of LQR controllers.

5. Project results

Although the project has yet to bear fruit in terms of control, most of the work needed to reach that point has been done.

It is unfortunate that the PC board malfunctioned at the very end of the project time span. As it heated up significantly during testing, we suspect that the digital to analog converter (DAC) on the board overheated and was damaged, perhaps because of faulty circuit design or because a heat sink was needed. It is likely that the boards tend to fail for the same reasons.

RoboFlag for MVWT Software User Guide

RoboFlag guide.

Filename: `Mvwt-users.pdf`

Date: January 27, 2008

Authors: Robert Christy

RoboFlag for MVWT Software Users' Guide

Robert Christy

January 27, 2008

Contents

1 Overview **1**

2 Setup **1**

2.1 Requirements 1
2.2 Setting up the *RoboFlag* software 2
2.3 Setting up the *MVWT* software 2
2.4 Setting up the controllers 3
2.4.1 Steelebot controller 4
2.4.2 Bat controller 6
2.4.3 Kelly controller 7

3 Running the game **7**

A Checking *RoboFlag/MVWT* software compatibility **8**

B The `sendcommand` utility **9**

1 Overview

This document is intended to give the reader a thorough understanding of what infrastructure has been developed to allow Cornell-developed *RoboFlag* games to be played on the Caltech-developed *MVWT* testbed. It should be noted that in this document, *MVWT* refers specifically to "*MVWT I*," the testbed located in room 12 of the Steele building and is not necessarily relevant to the *MVWT II* testbed.

2 Setup

2.1 Requirements

In order to run a *RoboFlag* game on the *MVWT* testbed, you need the following things:

- 1 *MVWT* testbed
- 2[1] or more of the following types of robots: "Steelebot" kinematic robots, "Bat" hovercraft, or "Kelly" fan-driven vehicles.
- 1 or more computers running *Windows®* *2000* or *Windows®* *XP*. These computers are necessary to run the *RoboFlag* software. Since this software can be rather computationally intensive, multiple computers may be necessary for larger numbers of robots.
- 1 computer running Linux. The software intended for this computer has only been tested under Redhat 9, but should work for any installation using the 2.4 kernel.
- A copy of the *RoboFlag* software. The testbed has been tested with the 2.1 beta version of the *RoboFlag* code. To test compatibility between the a version of the *RoboFlag* software and the *MVWT* software, see appendix A.

2.2 Setting up the *RoboFlag* software

Clearly, a working version of the *RoboFlag* software must be acquired and setup in order to run an *RoboFlag* game. The instructions for setup differ per *RoboFlag* version, however, the basic instructions should be present on the *RoboFlag* website: `http://roboflag.mae.cornell.edu/RoboFlag.html`. The only *MVWT* specific directions are below, otherwise the *RoboFlag* software should be configured normally.

1. In ConnectionParameters.txt, set the parameters *REAL_WORLD_HOST* to the hostname or IP address of the computer where the "Real world server," *MVServer*, will be running (ie. the Linux machine mentioned in section 2.1). Also, be sure to set the *REAL_WORLD_PORT* parameter to the port on which *MVServer* will listen for the Arbiter. The default port is 4545.
2. In RoboflagConstants.txt, make sure that the constant *UseWireless* is turned off (ie. has a value of zero). Although setting this value should not have any adverse effects, the RF-based wireless communication system used by the Cornell *RoboFlag* testbed is not used by the Caltech *MVWT* testbed, and therefore should be turned off.

2.3 Setting up the *MVWT* software

To set up the *MVWT*-specific software, first copies of the *MVWT* CVS module must be checked out of the central CDS CVS repository located at `mojave.-cs.caltech.edu:/cvsroot/`. To get access to this repository, see whomever it

[1] The game can be played with only one vehicle, but it will, most likely, not be a very interesting game

is that is in charge of it[2]. Once a local copy of the module has been checked out, the source code relevant to the remainder of this document can be found in the directory `mvwt/users/roboflag/`. Note that, in fact, one copy of the CVS module will need to be checked out on each computer where code will be compiled. At the moment, almost all of the code can be compiled natively on a Linux machine[3]. The only exceptions being the Bat controller, which can be compiled on a Linux machine with the appropriate cross-compiler, and the Kelly controller which requires a QNX 6.1 machine to compile. See section 2.4 for more on compiling specific controllers.

The first thing that needs to be done with a fresh checkout of the *MVWT* module is to build the *MVServer*, as the *RoboFlag* Arbiter cannot interface with the *MVWT* testbed without it. To do this, change to the directory `mvwt-/users/roboflag/MVServer` and run the command `make`. This will automatically compile *MVServer* and link it into an executable. Additionally, several utilities will be built. These include a utility to interface with the robots once they are running *RoboFlag* controllers (see appendex B for more information on the `sendcommand` utility) as well as several debugging tools. For more information on these utilities, read the `README` file located in that directory or run the command in question with the `-h` command-line option.

Next, *MVServer*'s configuration file `vehicle.conf` must be modified to indicate which vehicles will be used in the game, which (vision system) hats they are wearing, and which *RoboFlag* team they should be assigned to. The file is whitespace independent and comments begin with a #. In the file, each non-empty line represents a single vehicle with two integers and a hostname or IP address. The first of the two integers is the hat number (currently the hat numbers range 1-16), which is the number by which the vision system will identify that robot. The second integer is the number of the *RoboFlag* team to which that robot will be assigned. Currently, *RoboFlag* supports two teams of players, plus a third, "obstacle" team. The *RoboFlag* "agent id," which is used by *RoboFlag* entities, is simply determined by the order in which vehicles are specified in the file. See figure 1 for an example.

2.4 Setting up the controllers

In *RoboFlag* for *MVWT*, the programs that run on the individual robots are known as controllers. In order to play a *RoboFlag* game on the *MVWT* testbed, the controller for each type of robot to be used must be compiled and a copy of the resulting executable must be copied to each robot of that type that will be used. The code for all of the *RoboFlag* controllers can be found in subdirectories of the `mvwt/users/roboflag/controllers` directory.

Since each type has its own, intricate software infrastructure, each robot type has its own setup procedure. The setup procedure for each robot type is

[2]I have no idea who this person is. I gather that none of the grad students do either. In fact, I do not even have my own CVS user; I use Steve Waydo's

[3]See 2.1. It is assumed that this machine is of the x86 architecture

```
# vehicle.conf  -  Vehicle/team configuration for MVServer

# Team 1 (first robot team)
#   Vehicle     Team        Hostname/IP Address

    2           1           steelebot7      # agent id 1
    15          1           steelebot5      # agent id 2
    1           1           bat3            # agent id 3

# Team 2
#   Vehicle     Team        Hostname/IP Address

    13          2           steelebot2      # agent id 1
    14          2           bat1            # agent id 2
    16          2           steelebot4      # agent id 3

# Obstacles (team 3)
#   Vehicle     Team        Hostname/IP Address
    3           3           mvwt3           # obstacles don't
    6           3           mvwt6           # have agent ids
```

Figure 1: This is an example `vehicle.conf` file. The comments on the far right of each line indicate which agent id each vehicle gets as dictated by the ordering of the file.

outlined below. Typically, the setup procedure will need to be repeated for each robot, however, the controller will only need to be compiled once.

2.4.1 Steelebot controller

To compile the Steelebot controller and the necessary setup utilities, change to the directory `mvwt/users/roboflag/controller/Steelebot/` and type `make`. This step need only be done once. This step should take place on an x86-architecture computer running Linux with gcc.

The following steps should be performed for each Steelebot to be used in the *RoboFlag* game. There are several notes on the procedure located at the end of this section, be sure to read over them before you begin, they may save you some time and trouble.

1. Turn on the Steelebot to be set up and, when it has booted, login using either telnet or ssh. If, in the home directory, the executable `RFController` and the data file `anglesteps.data` already exist, then no further setup is necessary.

2. If `RFController` is not already present, copy it from the build machine to the Steelebot using `scp`. This executable is the *RoboFlag* controller.

It can be found among the results of the `make` performed earlier in this section. If, at this point, both `RFController` and `anglesteps.data` are present on the robot, no further setup is necessary.

3. If the executable `stepmap` is not already present, copy it from the build machine to the Steelebot using `scp`. This program is a utility that will generate the `anglesteps.data` file. It can be found in the results of the `make` as well.

4. Run the battery monitor program found on the Steelebot by typing `run battread`. The number on the far left should indicate the approximate battery power remaining. Make sure that there is enough battery power for another 20 minutes of operating time (ie. the battery power should be > 2150 or so). If there is not, power down (by running `run /sbin/halt`, waiting for one minute, then switching it off), replace the battery and boot again.

5. Now generate the `anglesteps.data` with the command `run stepmap`. This will take a while. The `stepmap` utility generates a look-up table that assists the *RoboFlag* controller in making precise, in-place rotations. To do this, however, the utility will cause the Steelebot to rotate, in sequence, 360 rotations of increasing angle. Be patient. When this step completes, a new file `anglesteps.data` should have been created and the Steelebot setup should be complete.

Notes:

- Each Steelebot is different. When choosing which Steelebots to use in your game, be sure to get an accurate listing of which are currently in a working state (this list is everchanging). Also make sure that you have the correct username and password for each robot. While the usernames and passwords are largely consistent among the robots, they are not entirely so.

- The software present on the Steelebot is not necessarily consistent either. Specifically, certain versions of `battread` operate differently than others. In some cases, the battery numbers will scroll by rather quickly and the program will terminate after 100 readings and, in others, the readings will progress slowly and `battread` will take an infinite number of them. In this latter case, feel free to use CTRL-C to terminate the program whenever you please.

- If you find yourself feeling lazy and not wanting to wait 20 minutes for a `stepmap` to complete, it is generally safe to copy an `anglesteps.data` file from another Steelebot. While, strictly speaking, `stepmap` generates a rotation profile accurate for each specific robot, most of the motor controllers across the Steelebots are consistent enough to get by with the same one.

- You must prefix most of your commands with `run` as demonstrated above. This is because many commands require root privileges to run, none of the programs have the setuid bit set and running them as root would be a faux pas. Therefore, the `run` command, which does have the setuid bit set (and is owned by root), exists to run whatever command it is passed as root.

2.4.2 Bat controller

To compile the Bat (hovercraft) controller, the ARM Linux cross compiler must first be installed. This must only be done once.

1. Check to make sure the cross compiler, libraries, header files and binary utilities have not already been installed. The standard target directory for their installation (on a Red Hat machine, anyway) is `/opt/Embedix/-tools/`. If they have, you may skip ahead to step 4.

2. The necessary software can be downloaded from `http://www.zaurus.com-/dev/tools/downloads/tools/` as RPM files. Listed here the are the versions of the necessary packages that are current as of the writing of this document:

   - The cross compiler, `gcc-cross-sa1100-2.95.2-0.rpm`
   - The standard C library, `glibc-arm-2.2.2-0.rpm`
   - Linux header files, `linux-headers-arm-sa1100-2.4.6-3.rpm`
   - Binary utilities, `binutils-cross-arm-2.11.2-0.rpm`

   Download and install these packages using `rpm` (see the `rpm` documentation for usage).

3. Modify your *PATH* environment variable to include the new cross-compiler binary directory `/opt/Embedix/tools/bin/`. This is necessary in order for the `Makefile` to work. The precise method for doing this varies depending on which shell you use.

   For users of `sh`, `bash`, or another `sh` variant : Add the line
   `export PATH=$PATH:/opt/Embedix/tools/bin/`
   to the `.profile` or `.bash_profile` file in your home directory. Then type that very same line at the command prompt and execute it (since the `.*profile` file won't be reread until the next time you login).

   For users of `csh`, `tcsh`, or another `csh` variant : Add the line
   `set path=($path /opt/Embedix/tools/bin/)`
   to the `.login` file in your home directory. Also type this same line into the command prompt and execute it (since the `.login` file won't be reread until the next time you login).

6

4. Change to the directory `mvwt/users/roboflag/controllers/Bat/` and run `make`. The Bat *RoboFlag* controller as well as associated utilities and debugging tools should build.

After building the controller. Copy the following files to each Bat to be used in the *RoboFlag* game:

- The *RoboFlag* controller executable, `RFController`
- The controller gains file, `gains.txt`
- The vehicle parameters file, `parameters.txt`

In order to perform this copy, make use of the ftp server that each Bat runs. See the documentation for the ftp client, `ftp`, for help copying these files. While it is not important where, specifically, you put these files, only a subset of the Zaurus' filesystem is writeable (much of the filesystem is actually on the Zaurus' ROM, thus making it read-only). One potential place to put the files is `/tmp`. Also, all subdirectories of `/home` are writeable. Finally, before the controller can be run, it must be made executable (as ftp often drops this file attribute). To accomplish this, log in to the Zaurus, change to the directory to where the files have been moved and type `chmod 755 RFController`. The Bat should now be ready to play *RoboFlag*.

2.4.3 Kelly controller

This controller requires special treatment as it is based upon the *RHexLib* modules developed for the *MVWT* testbed. As such, the code can only be compiled on a QNX 6.1 system with an *MVWT* system installation. See the *MVWT* documentation on how to perform this installation, if it is necessary. Since the *RoboFlag* controller is based upon the modules found in `libmvwt.a`, the *RHexLib* `Makefile`s are used to build it. Essentially, building the controller consists of changing to the directory `mvwt/users/roboflag/controllers/Kelly` and running `make`. For more detailed information, see the *MVWT* documentation.

The controller executable, `RFShell`, may be copied to the vehicle with `rcp` but not `scp` as the Kellys do not run ssh servers. Like the Bats, it is not of particular importance where the controller executable is placed, provided it is executable from that location. Also, the controller will require one parameters file, `vehicle_params.rc`, which should already be present on the vehicle. Simply make sure that a copy of (or symbolic link to) the file is in the same directory as the controller executable.

3 Running the game

In large part, running a *RoboFlag* game is simply a matter of following the instructions as laid out on the *RoboFlag* website. There are, however, several important steps that must be performed before the *RoboFlag* software may be started in order for *RoboFlag* to properly interface with the testbed.

1. Start all of the robots. Turn on each robot, making sure that all of the necessary batteries are charged and attached. In the case of the Kelly robots, make sure that the fan switches are switched on.

2. Login to each vehicle with `ssh`, `telnet` or `rsh` (typically each in a different window) and run the appropriate controller. For the Bats and Steelebots, the controller takes the vehicle's hat number as a mandatory command line argument. This is not the case for the Kellys as they determine their own hat number by assuming that it is correlated to their IP. Therefore, always make sure that Kelly 1 (mvwt1) is wearing hat 1, Kelly 2 (mvwt2) is wearing hat 2, etc. Also, do not forget to use the `run RFController ...` syntax on the Steelebots.

3. In the `mvwt/users/roboflag/MVServer/` directory, run *MVServer*. It should report a message indicating that it is waiting for the *RoboFlag* Arbiter to connect. Once the *RoboFlag* software is started, the message should change to indicate that the server is running.

4. Start the *RoboFlag* software as directed on the website, skipping the step that starts the simulator.

5. Play *RoboFlag* as usual.

Notes:

- *MVServer* will run until either 'q' is pressed on the keyboard or the *RoboFlag* Arbiter disconnects. In either case, the *RoboFlag* game will effectively be over (since one cannot play a *RoboFlag* game without seeing or controlling any robots) and *MVServer* will have to be restarted in order to restart the game.

- It is common to have many ssh/telnet/rsh clients open for the duration of the game as there should be one connection to each robot in play at all times. For the most part, this connection will be silent, that is, the *RoboFlag* controller will generally not report anything back to the user unless an error occurs. Furthermore, the *RoboFlag* controllers will not terminate upon the closing of *MVServer*. To terminate the *RoboFlag* controllers at the end of the game, use the `sendcommand` utility found in the `mvwt/users/roboflag/MVServer/` directory. See appendix B for more on `sendcommand`.

A Checking *RoboFlag/MVWT* software compatibility

*MVServer*interfaces with the *RoboFlag* Arbiter over the network to perform two tasks:

1. To forward vision information from the vision computer to the Arbiter

2. To distribute commands, received in bulk from the Arbiter, to each vehicle

To accomplish these tasks, *MVServer* and the Arbiter have to use the same data representation when exchanging data. This representation is given by the *RawVision* and *ObjectCommands* struct definitions found in `DataTypes.h`. In order for *MVServer* to work with a given version of the Arbiter, the definitions of *RawVision* and *ObjectCommands* used by *MVServer* and the Arbiter must match. Check `mvwt/users/roboflag/MVServer/RoboFlag/DataTypes.h` in the *MVWT* module against `datatypes/DataTypes.h` in your copy of the *RoboFlag* release. Be sure to check that the base data types and the data types of the elements match as well.

Additionally, when connecting, the *RoboFlag* Arbiter performs a "handshake" of sorts with the server. The constants and data types in the file `CommType.h` are needed to do this. Again, in order for *MVServer* to work with the Arbiter, the version of this file used by *MVServer* must be equivalent to the version used by the *RoboFlag* Arbiter. However, typically this file does not come with the *RoboFlag* release since the release does not include the Arbiter code, just the executable. This file seems unlikely to change and therefore it is not really necessary to check this file. However, if you have access to the source code of the *RoboFlag* Arbiter compare the *MVServer* file `mvwt/users/roboflag/-MVServer/RoboFlag/CommType.h` against `_common\network\CommType.h` in the Arbiter code.

B The `sendcommand` utility

One of the utilities that accompanies *MVServer* is `sendcommand`. It is useful for sending commands directly to vehicles without relying on *MVServer* or the *RoboFlag* Arbiter. From the command-line you specify which command to send and the vehicles to which the command. The syntax is:
`sendcommand [-h] [-i | -h | -x XVEL -y YVEL] [-p PORT] HOST [HOST...]`
Options:

`-i` Command the vehicle to go into idle mode, if possible

`-s` Command the vehicle controller to shutdown (this is the default)

`-x` Command the vehicle to move with an x-velocity of *XVEL*

`-y` Command the vehicle to move with a y-velocity of *YVEL*

`-h` Print a usage message

`-p` Set the port to which the command is to be sent (the default is 2020)

Notes:

- `sendcommand` only sends one command. Therefore, if multiple command types are specified on the command-line then only the last one is used.

The exception to this rule is that both `-x` and `-y` may be used to specify both components of the velocity command. If only one of these two is used, then the other component is set to zero.

- *MVServer* does not shutdown the controllers when it terminates. To terminate the controllers running on the individual vehicles, you must use `sendcommand`. Since the default action is to send a shutdown command, simply typing:
  `sendcommand` *`robot1 robot2 ...`*
  where *robot1*, *robot2*, etc. are the hostnames of the robots currently in use.

The Zaurus Software Development Guide

Information on the hovercraft software, namely `bathw.c`. Generally useful, but there were quite some bugs in the functions reading and converting the data from the ATMEL.

Filename: `Zaurus.pdf`

Date: August 29, 2003

Authors: Robert Christy

The Zaurus Software Development Guide

Robert Christy

August 29, 2003

Contents

1 Overview **1**

2 Writing Software for the *Zaurus* **2**

3 Using the *bathw* Library **3**
3.1 Using the fans 3
3.2 Getting sensor data 4
3.2.1 Enabling sensors 4
3.2.2 Updating sensors 4
3.2.3 Disabling sensors 6

4 Completing and Extending *bathw* **7**
4.1 Finishing existing sensor support 7
4.2 Adding additional sensor support 7

A The serial problem **8**

1 Overview

The Bat computing platform is comprised of two layers: the upper layer, a *Sharp*® *Zaurus* and the lower layer, an *ATMEL*® *ATmega128*. There are also two controller schemes being used by the *MVWT* group for the Bat. The first of these controllers is specific to *RoboFlag*. It should provide the control and interface necessary to run the Bat as a vehicle in a *RoboFlag* game. This controller is fairly small and effectively controls for velocity using the vision system and the onboard gyro[1] However, the *RoboFlag* controller is not sufficient for most control research applications and, moreover, it is unlikely that the *Zaurus* itself is sufficiently powerful for these applications. For this reason, there is planned[2] a force controller that will be used in conjunction with an

[1]Gyro support for the *RoboFlag* controller is not yet supported because of the ongoing troubles with the *Zaurus*'s ability to receive serial data. See appendix A for more on the state of sensor data support.

[2]This controller has not yet been implemented *at all*

offboard control platform. Under this scheme, sophisticated control laws (with trajectory generation, etc.) may be implemented on a computer separate from the Bat and the output of these controllers (fan forces, specifically) will be transmitted over the network to the vehicle. Once there, the force controller will use these values as reference to control the output forces of the individual fans by closing a loop around the onboard accelerometer data.

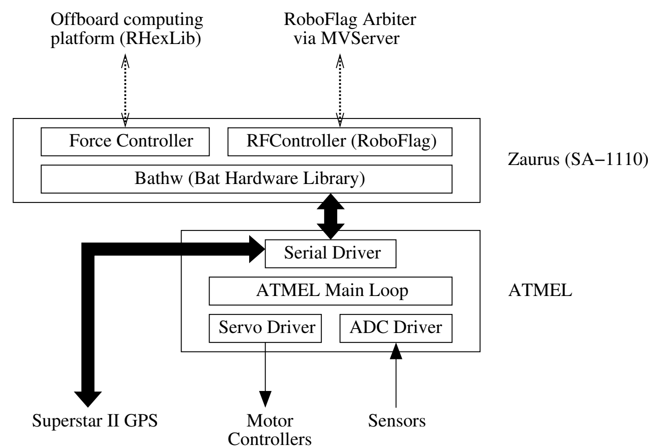


Figure 1: This diagram lays out the general structure of the Bat software. The items at the top are offboard, that is, not located on the Bat. The items at the bottom are components located on the PCB. The layers in between represent the *Zaurus* and the *ATMEL* and are labeled as such. The heavy solid, lines represent RS232 serial connections, the dotted lines represent 802.11b wireless ethernet connections, and the thin, solid lines represent PCB trace connections.

The relationship among the control schemes and computation layers is depicted in figure 1. The remainder of this document should serve to explain the operation of and, where relevant, interfaces to the various components in the "*Zaurus*" block of this figure. If developing additional sensor support or looking for information about the "*ATMEL*" block in the figure, be sure to also consult Hans Scholze's documentation for the *ATMEL* microcontroller code.

2 Writing Software for the *Zaurus*

The *Zaurus* runs the *Lineo Embedix Linux*™ operating system. Therefore, developing software for the *Zaurus* is very similar to developing software for any other Linux-based platform. Since the *Zaurus* uses an *Intel*® *StrongARM*

2

SA-1110 processor (rather than an *Intel*® x86 processor), a cross compiler is necessary to compile programs for the *Zaurus* on a different (x86 architecture) computer. The cross compiler recommended for writing *Zaurus* programs is `arm-linux-gcc`. It can be obtained from the *Zaurus* website: `http://www.zaurus.com-/dev/tools/downloads/tools/` as an RPM. In addition to the cross compiler itself, three other packages are necessary. Namely the binary utilities, glibc standard C library, and the standard C header files are required to compile programs for the *Zaurus*. For more instructions on how to install the cross compiler, see the *RoboFlag for MVWT Users' Guide* section on setting up the Bat controller.

3 Using the *bathw* Library

Since the *Zaurus* interfaces to the hardware through a serial connection, basic Linux device I/O will suffice to control the fans and read sensor data. However, for the sake of good programming practice, an abstraction layer exists to obviate the use of Linux I/O calls in programs that interact with the Bat hardware. This abstraction layer is manifested in the "Bat hardware library," *bathw*.

The *bathw* library is designed to provide the programmer with a comprehensive interface to the Bat hardware. In addition to basic intialization, cleanup and error handling functions, the *bathw* library contains an interface to all three fan outputs as well as an extensible interface to sensor data. While this document is not intended as a comprehensive reference (the header file is very thoroughly commented, and should be used as for that purpose), an outline of these interfaces will be provided below.

3.1 Using the fans

The programmer is provided access to the fans is provided through a pair of functions: *bathw_setfanoutputs* and *bathw_getfanoutputs*. Each of these functions take three arguments corresponding to the left, right and lift fans respectively.

The *bathw_setfanoutputs* function takes three integers, ranging between 0 and the defined constant *FAN_MAX*. This call will power each of the fans to the level specified by the given argument. If the programmer wishes to change only some subset of the fan values, he or she may opt to pass the constant *FAN_NO_CHANGE* as the argument for the fans whose levels should remain unchanged.

The *bathw_getfanoutputs* function takes pointers to three integers and stores the last values output to the fans at the location referenced by the pointers. Note that this function does *not* actually read values from the hardware. Rather it simply recalls cached values of previous *bathw_setfanoutputs* commands. As a result, if no previous calls to *bathw_setfanoutputs* have been made, then the values *FAN_UNDEF* will be returned in place of actual fan outputs. For this reason, it is good practice to always set zero fan outputs immediately after initialization.

That way, *bathw_getfanoutputs* will always return meaningful results. No values are stored for NULL arguments.

3.2 Getting sensor data

The *bathw* library contains a fairly robust interface to the sensors onboard the Bat. While at this time, the sensor interface is mostly implemented[3], it is largely untested. This is because of an ongoing dilemma with the serial communication onboard the *Zaurus*. For more information on this problem, please consult appendix A. This section, therefore, lays out the existing infrastructure and the intended functionality of the completed interface.

3.2.1 Enabling sensors

Before a sensor may be used, it must be enabled by calling the appropriate *bathw_enable* function. The existing sensors are enabled by the functions:

bathw_enable_gyro Enables the gyro

bathw_enable_accel Enables the accelerometers

bathw_enable_heading Enables the heading sensors (magnometers)

Each enable function typically takes a filename and one or more pointers to "buffers." as arguments. The filename refers to a file containing calibration data for that particular sensor. Each enable function must know how to read and interpret the data found in that file. The other arguments are pointers to variables accessable locally by the caller. When the sensors are updated (this will be covered in the next section), the updated sensor data will be stored in these variables.

3.2.2 Updating sensors

After the programmer has enabled all of the sensors he or she intends to use, the sensors must be periodically updated. This is done by calling *bathw_update_sensors*. After a call to *bathw_update_sensors*, the most recent sensor values will be stored in the buffers of each of the enabled sensors. See figure 3.2.2 for an example.

There is something very important to note about the *bathw_update_sensors* function. It performs a blocking read. This means that once it is called, the function will not return until sensor data is available. This behavior is undesirable for many applications, specifically those that perform several tasks asynchonously. Dynamic controllers and programs with networking both fall under this category. Three ways of working around this problem are presented here. The last and most elegant of these solutions is not currently supported, however, it is nevertheless described in the hopes that support may be added in the future.

[3]Several areas of code, such as the loading of sensor calibration data are missing, at the time of the writing of this document. See section 4 for more info.

4

```c
/* Example code snippet to demonstrate enabling and updating sensors */
#include <stdio.h>
#include "bathw.h"

int main()
{
    float gyro_value;

    /* Initialization code */
    if (bathw_init(DEFAULT_COMM_DEVICE, DEFAULT_COMM_SPEED) == -1)
    {
        fprintf(stderr, "Could not initialize Bat hardware: %s\n",
                bathw_stderr());
        return 1;
    }

    if (bathw_enable_gyro("gyro_calib.data", &gyro_value) == -1)
    {
        fprintf(stderr, "Could not enable gyro sensor: %s\n",
                bathw_stderr());
        return 1;
    }

    /* Main loop goes here ... */

        bathw_update_sensors();

        /* At this point gyro_value will contain the most recent gyro sensor
         * value */

    /* ... End of main loop */

    /* Cleanup code */
    bathw_disable_gyro();
    bathw_cleanup();

    return 0;
}
```

Figure 2: This program sample program demonstrates how the programmer can enable sensors and update them.

1. Use the *bathw_get_serial_fd* to obtain the file descriptor for the serial port and use the libc function *fcntl* to set the *O_NONBLOCK* flag. This will cause all serial port reads to be non-blocking. As a result, when *bathw_update_sensors* is called when no serial data is ready, -1 will be returned with a *bathw*error code of *BATERDERR*. This option is undesirable for several reasons:
   - It violates the stated goal of abstracting the programmer from Linux I/O
   - The programmer cannot distinguish the unavailablity of new sensor data from a genuine I/O error
   - Non-blocking I/O often requires polling on the part of the programmer, which is generally bad practice.
2. Use the *bathw_get_serial_fd* to obtain the file descriptor for the serial port and use the libc function *select* to determine when data is available on the serial port. In this manner, the programmer can ensure that *bathw_update_sensors* is only called when the read will not block. While this method still violates the abstraction layer (as the previous method did), this method has the added advantage that a *select* can be performed on many file descriptors simultaneously, thus providing a simple means for performing I/O with many devices simultaneously. This is very useful for networking applications. In general, this method is the best of the supported methods.
3. Use POSIX threads to set up a separate "sensor thread" whose sole responsibility is to retrieve sensor data while the main thread is remains available to perform other tasks. While this is clearly the most elegant solution (it does not violate the abstraction layer), it is not currently supported since the standard C I/O routines are not thread-safe. This means that having a sensor thread read sensor data while the main thread sends fan commands could have unpredictable results. Also POSIX threads carry extra computational overhead which should be considered a minor detractor of this method.

3.2.3 Disabling sensors

Sensors may be disabled as well. This is useful if you want to top using a particular sensor part of the way through the execution of a program. Sensors should also be disabled at the end of the program. At the moment, there is no true need to do this, in the future, neglecting to do this may affect the performance of subsequently running programs. Therefore, it is good programming practice to do so.

4 Completing and Extending *bathw*

As has already been stated, the *bathw* library is not entirely complete. Moreover, in the future, sensors will be added to the vehicle, which will require the addition of support for those sensors. For both of these reasons, modification of the *bathw* code will be necessary. This section is to make the developer's job as painless as possible in doing either of these tasks.

4.1 Finishing existing sensor support

The only thing missing from *bathw* currently is the code to load calibration data for each of the existing sensors and the code to use that calibration data to convert the raw *ATMEL* output values to meaningful, phyical values. The code changes must be made directly to the `bathw.c` source file and the locations where changes must be made are marked with the comment "FIXME." Once the serial dilemma[4] is resolved and the format of the calibration data determined, these pieces of code should be fairly straight forward to implement. At this time, I do not forsee the need to add any more than 20 lines of code to the existing *bathw* codebase.

4.2 Adding additional sensor support

The *bathw* sensor interface was designed with the intent that it be relatively simple to add support for new sensors. There are five steps to adding support for a new sensor device to the *bathw* code[5].

1. In `bathw.h`, add a constant of the form *SENSOR_*sensorname (where *sensorname* is an abbreviated name for your sensor) to the list of existing sensor constants found in the anonymous enumeration (`enum` in C). The constant for the new sensor should be inserted immediately above the line containg the constant *SENSOR_COUNT*. Note that since the constants are actually defined through an enumeration there is no need to specify an integer, the next integer in the sequence will automatically be assigned to your sensor.

2. Near the top of the file `bathw.c`, add a line to the array *sensors* containing information about your sensor. *sensors* is an array of type `struct sensor_info`. This structure has the following elements (note the order does matter):

   `char` ***mnemonic*** This field should be set to the character or byte that the *ATMEL* sends just before it sends data from the sensor. That is,

[4] See appendix A

[5] This does not include any steps necessary to add support for the sensor to the *ATMEL* code. There will, invariably, need to be changes to that code in order to get new sensor devices to work on the Bat. Please consult Hans Scholze's documentation for instructions on how to do that.

this byte is the header byte that is used to inform the *bathw* as to which type of sensor data is waiting on the serial port.

`char` ***enabled*** This character is a boolean value that represents whether the sensor is currently enabled. In the structure definition, this value should always be 0, since all sensors are, at startup, disabled.

int (*handler`)(uint8_t *)` This is a pointer to the handler function that *bathw_update_sensors* will call when data of the appropriate sensor type arrives. The argument to this function is the raw sensor data as read off the serial port. This function will be written at step 4.

`uint8_t` ***datasize*** This value is the length of sensor data in bytes. More specifically, it is the number of bytes that *bathw_update_sensors* should read off of the serial port and pass to *handler* when a byte equal to *mnemonic* is read.

The element should be added at the end of the array (ie. it should be the *SENSOR_*sensorname'th element in *sensors*).

3. Write an enable function. The convention used thus far for naming this function is *bathw_enable_*sensorname where *sensorname* is the abbreviated name of the sensor being added. This function must do several things:
   - Load any calibration data necessary for the sensor and or prepare any precalcuated values necessary for the sensor.
   - Send any commands to the *ATMEL*that are necessary to start the *ATMEL*sending the sensor data. At the moment, no such commands exist.
   - Set the variable *sensors*`[`*SENSOR_*sensorname`].`*enabled* equal to a non-zero value (1 will do just fine).
   - Record "buffer" pointer(s) so that the caller may access the processed sensor data.
4. Write the update function. This is the function to which *handler* in step 2 must be set. This function takes one argument, `uint8_t *`*buffer*, which is a pointer to the raw sensor data. The handler must then take this data, convert it into a meaningful value, and store that in locations referenced by the "buffer" pointer(s) recorded by the enable function.
5. Write a disable function. This function must simply reset the *enabled* flag that was set by the enable function and send any commands necessary to shut the sensor down on the *ATMEL*. At the moment, no such command exist.

A The serial problem

At the time of the writing of this document, there is a very curious problem with serial communication on the *Zaurus*. On a certain subset of the Bats, the

Zaurus can send data over the serial port but it cannot receive. That is, fan commands are sent without trouble yet peculiar behavior obscures the sensor data. The strange behavior is this: for most of the time, the serial port appears to be receiving no data. Every once in a while a read event occurs on the serial port (as indicated by the *select* libc call) but when a *read* call is attempted it blocks indefinitely. It should be pointed out that this *read* call only attempts to read a single byte, and therefore, if data is in fact available, the *read* should never block.

The behavior seems to follow the *Zaurus*'s rather than the *ATMEL*. The broken *Zaurus*'s have also been tested against a computer and have exhibited the same faults. Moreover, the *ATMEL*'s paired with broken *Zaurus*'s have been connected to working *Zaurus*'s and shown to work. This seems to indicate that the problem is localized to the *Zaurus*.

The exact same program is used on each vehicle, therefore it does not seem likely to be a high-level software issue. The status of a *Zaurus*, however, has been known to change after reinstalling all of the software (see the *MVWT II* documentation for the *Zaurus* installation procedure). Namely, a working *Zaurus* has become broken after a reinstallation. Therefore it seems unlikely that the *Zaurus* hardware is at fault. Additionally, the ailment seems to affect at least three *Zaurus*'s, which makes a hardware problem seem further unlikely.

While we have made remarkably little headway in tracking down the origins of this problem, there are still a few experiments that may be worthwhile. For example, it may be educational to try to get the *Zaurus* communicating with a computer at a baud rate slower than 115.2kbps. This should be attempted first with a standard piece of software where confidence that the serial I/O code is correct is high. And then should be tried with custom code, such as `democtlr`, where the confidence is not as high.

MVWT II Hovercraft Project Report

Contains information on trajectory generation, analysis of the vision system data, derivation of a gain scheduled LQR controller, a MATLAB simulation of the MVWT.

Filename: `MVWT II Hovercraft Project Report.pdf`

Date: not specified, presumably early 2008

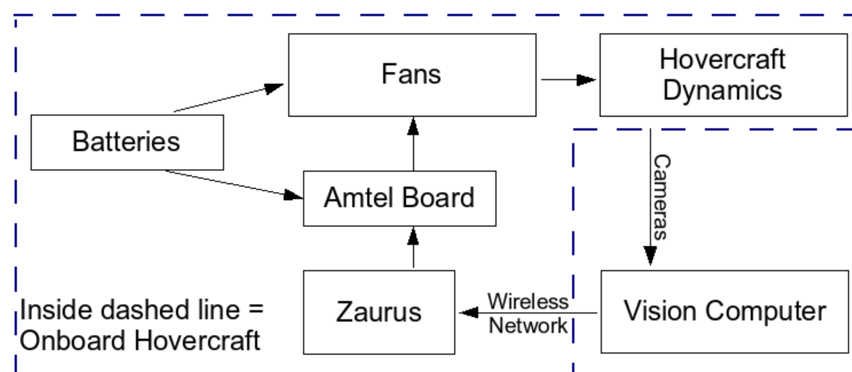
Authors: Chris Schantz (Kenny, Vanessa, Nam)

MVWT II Hovercraft Project Report
-Chris Schantz (Kenny, Vanessa, Nam)

This report details the efforts and progress of our group in designing a 2 degree of freedom controller for the MVWT II hovercraft. This process was a great learning experience for myself. While the hardware did not always work as intended, and in the end was abandoned in favor of simulation, sufficient progress and insight was gained that I strongly recommend that the CDs department hires a few EEs to remake a fleet of interface boards to allows this platform to continue to serve as such a good tool for learning control design. I believe the rest of the platform is sound, and combined with the tools developed by our team it can continue to be improved. Its second order dynamics significantly increase the challenge as well as reward in success.

Besides formulating the controller, a major contribution to the platform was made in the form of Matlab based simulation environment. This simulator's design and use is described in detail, and will ensure it is made part of the MVWT II repository. The report is composed of a series of chapters whose headings should be fairly self explanatory. This format was chosen as I found it easier to write up certain aspects all at once and then move on/ develop more of the project. This results in some amount of redundancy, but the chapters are designed to be largely self sufficient in explaining their own areas. The chapter list follows a description of the MVWT II platform.

The hovercraft are light weight battery powered vehicles consisting of left, right, and lift fans as well as an onboard Zaurus SL-5500 with wireless network capability for computation and various sensors. The main sensor in the system is an overhead vision system that broadcasts the state of the vehicle wirelessly. A block schematic of the set up is shown below.



The chapters that follow:

1. Trajectory Generation
2. Vision Data Analysis
3. Controller Derivation
4. Experimental Trials and Results
5. Simulator Design and Use
6. Simulation Based Results

Trajectory Generation Document for Hovercraft

The six dynamic equations of motion for the hovercraft are written to the right. It is noted that the system is differentially flat with the flat outputs $z=(x,y)$. Originally when constructing the remainder of the state and inputs from the flat outputs, the friction terms were left out, as their inclusion leads to immensely more complex formulas. It was believed that it was a reasonable assumption to demand that the controller be robust enough so that the un-modeled disturbances of friction are accounted for. This turned out to be a bad assumption. It was found that in the event of any curvature in the path the desired heading of the hovercraft would point directly at the center of curvature. This would cause the hovercraft to quickly loose its velocity component along the trajectory and accelerate towards the center of curvature. Needless to say there would be no standard controllers capable of overcoming this issue. Thus the much more complicated differentially flat formulas that included the friction terms were used for trajectory generation. The fully correct formulas for the remainder of the states in terms of the flat outputs are shown below. Note that the temporary variables x1…3, y1…3, and temp1….4 are used to allow readability and compact display of these extremely large formulas.

$$\dot{x}=\dot{x}$$
$$\dot{y}=\dot{y}$$
$$\dot{\theta}=\dot{\theta}$$
$$m\ddot{x}=(u_1+u_2)\cos(\theta)-\mu\dot{x}$$
$$m\ddot{y}=(u_1+u_2)\sin(\theta)-\mu\dot{y}$$
$$J\ddot{\theta}=(u_1-u_2)r_f-\psi\dot{\theta}$$

$$x_1=\mu\dot{x}+m\ddot{x}$$
$$x_2=\mu\ddot{x}+mx^{(3)}$$
$$x_3=\mu x^{(3)}+mx^{(4)}$$
$$y_1=\mu\dot{y}+m\ddot{y}$$
$$y_2=\mu\ddot{y}+my^{(3)}$$
$$y_3=\mu y^{(3)}+my^{(4)}$$
$$\theta(t)=\arctan 2(\frac{y_1}{x_1})$$
$$\dot{\theta}(t)=\frac{-y_1x_2+x_1y_2}{x_1^2\left(1+\frac{y_1^2}{x_1^2}\right)}$$

$$Temp1=\frac{1}{x_1^3\left(1+\frac{y_1^2}{x_1^2}\right)^2}$$
$$Temp2=-2\left(1+\frac{y_1^2}{x_1^2}\right)x_2(-y_1x_2+x_1y_2)$$
$$Temp3=\frac{1}{x_1^2}(2y_1(\mu\dot{y}x_2-\mu\dot{x}y_2+m^2(\ddot{y}x^{(3)}-\ddot{x}y^{(3)}))^2)$$
$$Temp4=x_1\left(1+\frac{y_1^2}{x_1^2}\right)(-y_1x_3+x_1y_3)$$

$$\ddot{\theta}(t)=Temp1(Temp2-Temp3+Temp4)$$
$$u_1(t)=\frac{m(\ddot{x}\cos(\theta)+\ddot{y}\sin(\theta))+\mu(\dot{x}\cos(\theta)+\dot{y}\sin(\theta))}{2}+\frac{J\ddot{\theta}+\psi\dot{\theta}}{2r_f}$$
$$u_2(t)=\frac{m(\ddot{x}\cos(\theta)+\ddot{y}\sin(\theta))+\mu(\dot{x}\cos(\theta)+\dot{y}\sin(\theta))}{2}-\frac{J\ddot{\theta}+\psi\dot{\theta}}{2r_f}$$

Needless to say the full formula for $u_1(t)$ and $u_2(t)$ is quite long due to the various derivatives of $\theta(t)$. For my investigations with the SNOPT solver I chose to parameterize $\theta(t)$ explicitly as well as $x(t)$ and $y(t)$. This leads to an equality constraint in the non-linear specification. This equality constraint must be enforced for the paths to remain feasible, but if you do this you can just take derivatives of $\theta(t)$ directly. The full list of constraints to satisfy includes the equality constraint, the input constraints, and the start and ending position constraints:

$$x(0) = x_0$$
$$y(0) = y_0$$
$$\theta(0) = \theta_0$$
$$\dot{x}(0) = 0$$
$$\dot{y}(0) = 0$$
$$\theta(t) = \arctan\left(\frac{\ddot{y}}{\ddot{x}}\right)$$
$$\dot{\theta}(0) = 0$$
$$x(t_f) = x_f$$
$$y(t_f) = y_f$$
$$\theta(t_f) = \theta_f$$

$$0 \le \frac{m(\ddot{x}\cos(\theta) + \ddot{y}\sin(\theta)) + \mu(\dot{x}\cos(\theta) + \dot{y}\sin(\theta))}{2} + \frac{J\ddot{\theta} + \psi\dot{\theta}}{2r_f} \le 0.7N - \delta$$

$$0 \le \frac{m(\ddot{x}\cos(\theta) + \ddot{y}\sin(\theta)) + \mu(\dot{x}\cos(\theta) + \dot{y}\sin(\theta))}{2} - \frac{J\ddot{\theta} + \psi\dot{\theta}}{2r_f} \le 0.7N - \delta$$

Of core initial velocities can be non zero, but this is unlikely in practice. The δ above is set to some small value less than 0.7 in order to ensure that the controller has some margin of velocity left over to execute control as well as to provide a buffer to reduce the chance that the trajectory violates the input constraints between the collocation points. Each of $\theta(t)$, $x(t)$ and $y(t)$ are parameterized using B-splines and the constraints are enforced at collocation points along path. The question of finding a good t_f makes the current formulation risky, as a t_f that is too small will not allow any feasible trajectories to be found and a t_f that is too large will be inefficient. In addition, given the simple and relatively cheap and quick nature of our control input, varying voltages to an electric motor, there is no reason to make the inputs part of our cost function. Accordingly it makes sense to minimize t_f as our objective function to drive the nonlinear optimization. This cam be accomplished by parameterize time as a scalar using a new normalized time variable $\tau = \frac{t}{t_f}$. Time derivatives get scaled as $\frac{d}{dt} = \frac{1}{t_f}\frac{d}{d\tau}$ and the rewritten constraint equations follow:

$$x(0) = x_0 \qquad t_f > 0$$
$$y(0) = y_0 \qquad x(1) = x_f \qquad \frac{dy}{d\tau}(0) = 0$$
$$\theta(0) = \theta_0 \qquad y(1) = y_f \qquad \frac{d\theta}{d\tau}(0) = 0$$
$$\frac{dx}{d\tau}(0) = 0 \qquad \theta(1) = \theta_f$$
$$\theta(\tau) = \arctan\left(\frac{d^2y / d\tau^2}{d^2x / d\tau^2}\right)$$

$$0 \le \frac{m(\ddot{x}\cos(\theta) + \ddot{y}\sin(\theta)) + \mu(\dot{x}\cos(\theta) + \dot{y}\sin(\theta))t_f}{2t_f^2} + \frac{J\ddot{\theta} + \psi\dot{\theta}t_f}{2r_f t_f^2} \le 0.7N - \delta$$

$$0 \le \frac{m(\ddot{x}\cos(\theta) + \ddot{y}\sin(\theta)) + \mu(\dot{x}\cos(\theta) + \dot{y}\sin(\theta))t_f}{2t_f^2} - \frac{J\ddot{\theta} + \psi\dot{\theta}t_f}{2r_f t_f^2} \le 0.7N - \delta$$

The splines that parameterize $\theta(t)$, $x(t)$ and $y(t)$ need to be at minimum C^3 everywhere to have continuous second derivatives. This necessities a smoothness condition of 3 and an order of 4 or higher for the splines. It turns out that the SNOPT solver is very sensitive to initial guesses, as is does not guarantee global convergence. The cost function now seeks to minimize t_f . It's a good idea to feed it as feasible a trajectory as can be found for its initial guess. It was found that an initial guess trajectory composed of two ninth order polynomials and solved for the by standard matrix method outlined below. This method is unable to find a guess satisfying the input constraints, and it also relies on a pre chosen t_f , but one can make this pre chosen t_f long and end up with generally lower inputs than with a short t_f . From these two polynomials it is possible to generate the initial guesses for each of $\theta(t)$, $x(t)$, $y(t)$ and t_f . The coefficients of the polynomials are calculated using $a = M^{-1}X$, as shown on the next page. Using an initial guess generated in this manner was found to be the only way to get the SNOPT solver to compute. Hover, the results using the equations above and the optragen Matlab toolbox written by Raktim Bhattacharya and the associates SNOPT solver have been unsatisfactory to date. In all cases the solver is unable to find trajectories that satisfy the input constraints. In rare cases the feed forward inputs are kept below the maximum of 0.7N, but in no instances thus far were the inputs always positive.

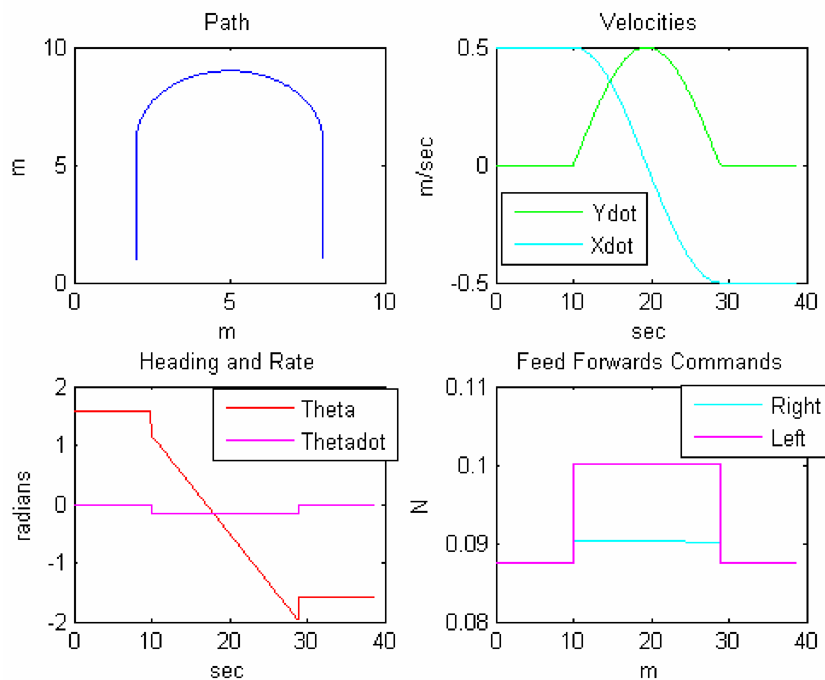
$$M =$$

$$\begin{bmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 24 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & t_f & t_f^2 & t_f^3 & t_f^4 & t_f^5 & t_f^6 & t_f^7 & t_f^8 & t_f^9 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 2t_f & 3t_f^2 & 4t_f^3 & 5t_f^4 & 6t_f^5 & 7t_f^6 & 8t_f^7 & 9t_f^8 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 2 & 6t_f & 12t_f^2 & 20t_f^3 & 30t_f^4 & 42t_f^5 & 56t_f^6 & 72t_f^7 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 6 & 24t_f & 60t_f^2 & 120t_f^3 & 210t_f^4 & 336t_f^5 & 504t_f^6 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 24 & 120t_f & 360t_f^2 & 849t_f^3 & 1680t_f^4 & 3024t_f^5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 6 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 24 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & t_f & t_f^2 & t_f^3 & t_f^4 & t_f^5 & t_f^6 & t_f^7 & t_f^8 & t_f^9 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 2t_f & 3t_f^2 & 4t_f^3 & 5t_f^4 & 6t_f^5 & 7t_f^6 & 8t_f^7 & 9t_f^8 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 & 6t_f & 12t_f^2 & 20t_f^3 & 30t_f^4 & 42t_f^5 & 56t_f^6 & 72t_f^7 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 6 & 24t_f & 60t_f^2 & 120t_f^3 & 210t_f^4 & 336t_f^5 & 504t_f^6 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 24 & 120t_f & 360t_f^2 & 849t_f^3 & 1680t_f^4 & 3024t_f^5
\end{bmatrix}$$

$$a = \begin{bmatrix} a_1 & a_2 & a_3 & a_4 & a_5 & a_6 & a_7 & a_8 & a_9 & a_{10} & b_1 & b_2 & b_3 & b_4 & b_5 & b_6 & b_7 & b_8 & b_9 & b_{10} \end{bmatrix}^T$$

$$X = \begin{bmatrix} x_0 & \dot{x}_0 & \ddot{x}_0 & x_0^{(3)} & x_0^{(4)} & x_f & \dot{x}_f & \ddot{x}_f & x_f^{(3)} & x_f^{(4)} & y_0 & \dot{y}_0 & \ddot{y}_0 & y_0^{(3)} & y_0^{(4)} & y_f & \dot{y}_f & \ddot{y}_f & y_f^{(3)} & y_f^{(4)} \end{bmatrix}^T$$

It was decided to create the straight line trajectories and the U shaped trajectory used in the simulation and experimental tests with the fully differentially flat approach. The straight line trajectories are given a constant velocity, and a constant theta, greatly reducing the complexity of the resulting flatness equations by making many of the higher derivatives zero. The Curve of the U shaped trajectory is constructed from a perfect half circle, and while none of the X or Y derivatives are zero, they are simple trigonometric functions. $\dot{\theta}$ is constant and $\ddot{\theta}$ is zero, and this also makes things easier. The trajectories are represented as a sequence of points spaced so that at the correct velocity the hover craft will encounter them every tenth of a second. One not however: there are two discontinuities where the curvature changes on the U shaped trajectory, as can be seen in the figure below in the Heading subplot, and these discontinuities cause small but noticeable tracking errors when the hovercraft encounters them.



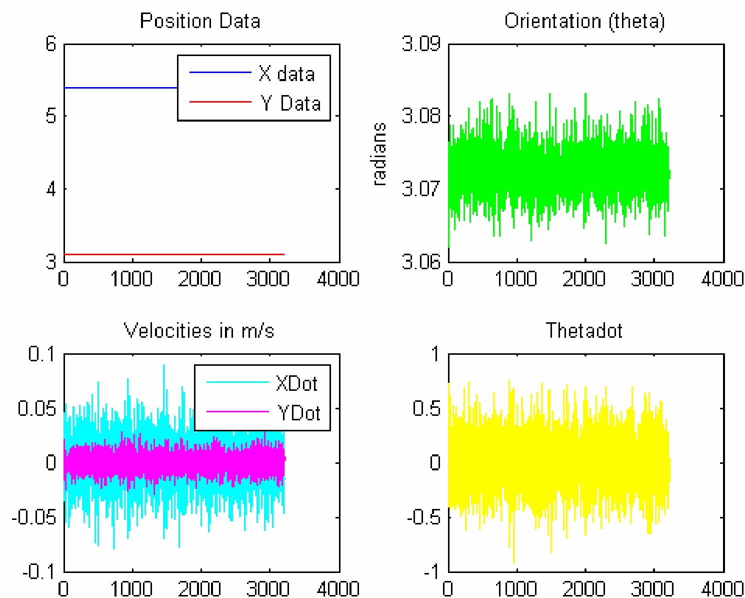
Vision Data Analysis:

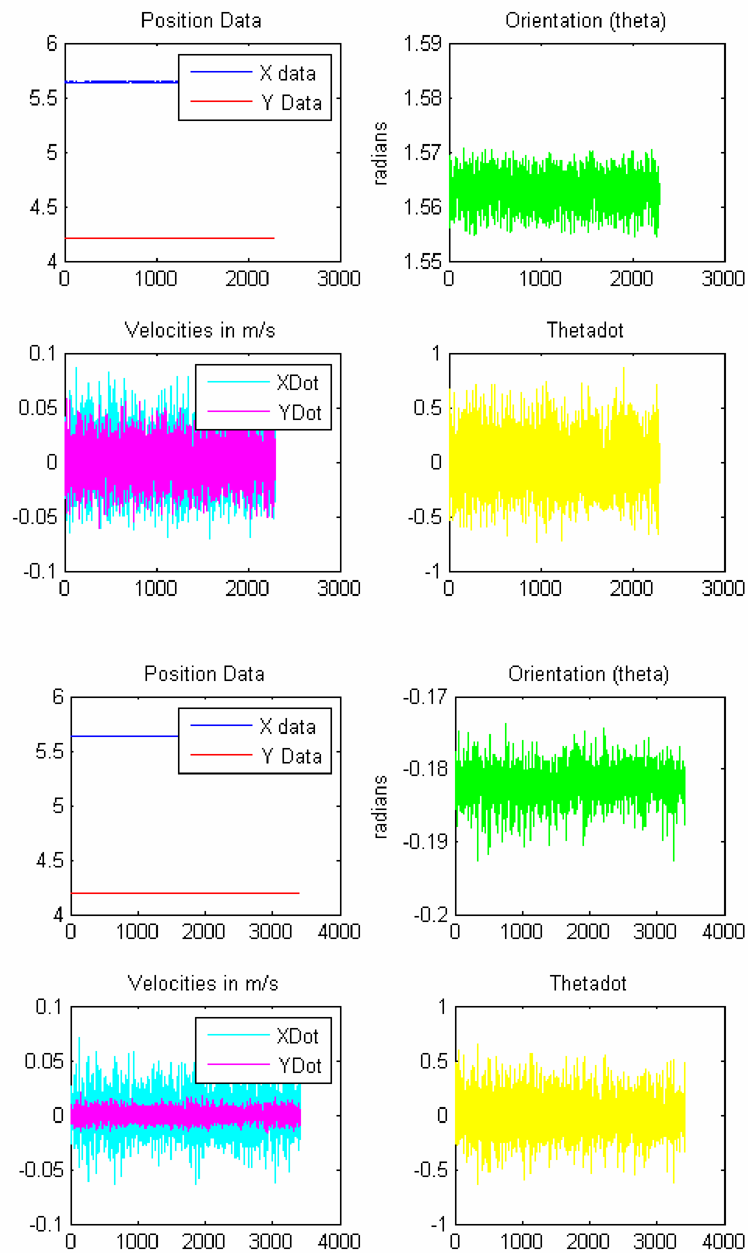
Three sets of vision data were taken at different orientations and positions to qualify the noise characteristics of the vision sensors. This was done with a still vehicle based on the assumption that we will never achieve velocities that will lead to motion blur, and the assumption that the vision algorithm uses only frame by frame information and has minimal reliance on previous frame data when examining the current frame. Thee plots of the taken data are below, along with the standard deviations of the values plotted. Based on our modeling of the noise as a Gaussian process with mean zero, we will simply use the square of the standard deviations as the variances of the sensor noise.

Standard Deviations of Vision Data:

Data Set	X	Y	Theta	Xdot	Ydot	Thetadot
1	0.00025882	0.000095666	0.0028256	0.021598	0.0079263	0.22927
2	0.00029827	0.00023202	0.0029033	0.02556	0.018989	0.24625
3	0.00019648	0.00006082	0.0020149	0.016221	0.0051505	0.16293

Data Plots:




Position Data
X data
Y Data
Orientation (theta)
radians
Velocities in m/s
XDot
YDot
Thetadot
Position Data
X data
Y Data
Orientation (theta)
radians
Velocities in m/s
XDot
YDot
Thetadot

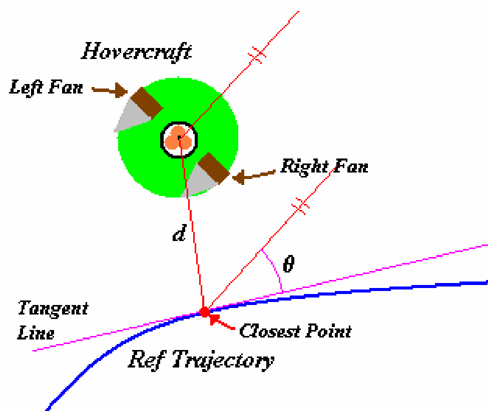
Controller Derivation Document for Gain Scheduled LQR Controller Version 2

The six dynamic equations of motion for the hovercraft are shown to the right. The hovercraft has two side fans, one mounted to the left of the center of mass, and one mounted to the right of the center of mass. They are mounted the same distance away from the center of mass and that distance is known as the constant r_f. The mass of the hovercraft is m, J is the rotational inertia, μ is the coefficient of sliding friction and ψ is the coefficient of rotating friction. It should be noted that the side fans are unable to thrust in reverse. The controller structure used strives to decouple the forward dynamics of the hovercraft as much as possible from the longitudinal dynamics. This was done by treating the control inputs separately as a forward thrust (F) and a torque (T), and then converting the thrust and torque into individual motor commands. An inner loop controller onboard the hovercraft was also utilized to stabilize the most unstable state in the system: the hovercrafts heading. The natural instability of the heading was confirmed both in experiment and in simulation.

$$\dot{x} = \dot{x}$$
$$\dot{y} = \dot{y}$$
$$\dot{\theta} = \dot{\theta}$$
$$m\ddot{x} = (u_1 + u_2)\cos(\theta) - \mu\dot{x}$$
$$m\ddot{y} = (u_1 + u_2)\sin(\theta) - \mu\dot{y}$$
$$J\ddot{\theta} = (u_1 - u_2)r_f - \psi\dot{\theta}$$

$$F = u_1 + u_2$$
$$T = u_1 - u_2$$

Interaction with the Reference Trajectory:



The basic reference path around which the controller was linearized was a straight line path parallel to the x axis with a heading of zero. The decision was made that the hovercraft would not attempt to strictly adhere to the path through space and time specified by the reference trajectory. Spatial accuracy was given the higher priority, and because of this it was determined that the hovercraft should strive to travel at the specified velocity of the closest part of the trajectory, and not try to catch up with where it should be based on the nominal elapsed travel time. This allows one to remove spatial dependence in the longitudinal/ velocity controller.

Lateral Dynamics and Controller:

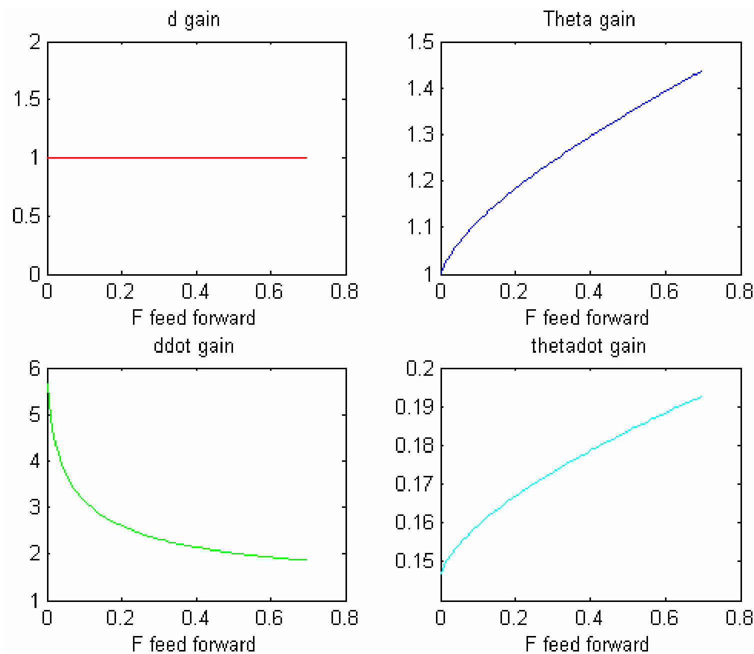
$$\dot{d} = \dot{d}$$
$$\dot{\theta} = \dot{\theta}$$
$$m\ddot{d} = (F)\sin(\theta) - \mu\dot{d}$$
$$J\ddot{\theta} = (T)r_f - \psi\dot{\theta}$$

Using the nominal reference trajectory, the y state was transformed to a lateral distance d from the trajectory, and the $\dot{y}$ state represents the closing speed of this lateral distance. The θ and $\dot{\theta}$ state equations are kept relatively unchanged. In the derivation of the lateral dynamics the torque term T is used as the input to the lateral dynamics, while the force F is treated as a gain scheduling parameter. The gain scheduled linearization of the lateral dynamics is shown.

$$A(t) = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & \frac{F_{ff}\mu\cos(\theta)}{m} = \frac{F_{ff}\mu}{m} & \frac{-\mu}{m} & 0 \\ 0 & 0 & 0 & \frac{-\psi}{J} \end{bmatrix}$$

The controller used to control the torque T is an LQR controller that generates proportional gains for all four states in the lateral dynamics. However, due to the bad quality of our ability to sense $\dot{\theta}$ with the vision system, the weighting on that state is kept at one, and the generated gain on that state is zeroed out. The three remaining proportional gains act as a PD controller on d and a P controller on θ. Since the A matrix is dependent on the feed forward force F_{ff}, the infinite time LQR gains are solved for at every time step in the simulation to account for the possible change in F_{ff}. A plot of the gains on the four states using the identity matrix for the weightings follows.

$$B(t) = \begin{bmatrix} 0 \\ 0 \\ 0 \\ \frac{r_f}{J} \end{bmatrix}$$



Longitudinal Dynamics and Controller:

$$m\ddot{x} = (F)\cos(\theta) - \mu\dot{x}$$

$$A = \frac{-\mu}{m}$$

$$B = \frac{\cos(\theta)}{m} = \frac{1}{m}$$

The longitudinal controller is responsible for maintaining the velocity of the hovercraft. Not much experimental testing was done to ensure the performance of the longitudinal controller due to the lateral performance taking priority as long as the feed forward commands were able to move the hovercraft at all. Had the hardware not failed when it did focus would have returned to the longitudinal controller. Right before the hardware failure we had been using a proportional plus integral controller to regulate the longitudinal performance of the hovercraft. In simulation the longitudinal controller was made to be a simple proportional controller with a single gain chosen by trial and error. Given that the weakest point in the simulation is probably its treatment of the linear sliding friction of the hovercraft, and that this friction bears greatly on the velocity control, the longitudinal controller used in the simulation is very likely insufficient for application to reality. The output of this controller was the forward force term *F*.

Conversion of Torque and Force to Left and Right Motor Thrusts:

Given the constraints of the fan actuators, care had to be taken in this step to insure that significant lateral error resulting in large torques did not cause unwanted and uncontrolled acceleration of the hovercraft. This was a frequent problem with our experimental trials, as any deviation above a certain magnitude would cause this run away behavior. This was due to the previous controller designs we were using that calculated the command for the left and right fan separately and in parallel for both of the longitudinal and lateral controllers, and then combined the outputs of both controllers with the feed forward commands form the trajectory. This combined value for each fan would simply be truncated to the range (0, 0.7)N and then sent to the fans. In any event where the truncation step converted a negative thrust command to a zero, an imbalance in the sum of the two fans would occur that would rapidly accelerate the craft in an undesired manner.

The present method as tested extensively in simulation is shown in the Matlab code fragment below. It takes the force from the longitudinal controller *F* and spreads it equally between the left and right fans and then sums in the feed forward commands. Then the torque term *T* is applied by boosting the correct fan and decreasing the other by an equal amount. In the event that there is not enough feed forward and force command already being sent to the fan that is to be decreased, the algorithm will zero that fan and only increase the other fan by the amount that the first fan was decreased. This avoids the problem of relying on negative fan commands, but introduces another problem: if the hovercraft is being commanded to move very slowly, it will have a hard time turning. This issue rarely comes up in practice though, as all the trajectories tested are meant to be traveled as speeds of .25m/s or greater. A parameter called coupling_factor is present and nominally set to zero. Setting this parameter to one will make the algorithm behave just like the older version outlined in the previous paragraph.

```
coupling_factor = 0;
left_fan = F/2 + left_ff;
right_fan = F/2 + right_ff;
T=T/r_f;
if (T > 0)
    if(left_fan > (T/2))
        right_mod = T/2;
        left_mod = -T/2;
    else
        left_mod = -left_fan;
        right_mod = -left_mod + (coupling_factor*(T -
left_fan));
    end
else
    T = T * -1;
    if (right_fan > (T/2))
       left_mod = T/2;
       right_mod = -T/2;
    else
       right_mod = -right_fan;
       left_mod = -right_mod + (coupling_factor*(T-
right_fan));
    end
end

left_fan = left_fan + left_mod;
right_fan = right_fan + right_mod;
```

Inner Loop Controller:

The most unstable state of the hovercraft is the rotational mode, and this problem is aggravated because by the vision system because the heading and heading rate information is nosier than the other states of the same order. However the hovercraft has an onboard gyroscope with insignificant time delay and low noise. This gyroscope gives the heading rate of change, and this sensed input is used to artificially increase the rotational friction as seen by the outer loop controller. The gyro reading in rad/sec is multiplied by a small "artificial rotational friction" parameter to calculate the counter torque value, and this value is then divided by r_f to calculate the fan command to be added into the outer loop fan commands before finally being sent to the fan motors. A local copy of the algorithm outlined above is used when adding in the resisting torque force. For the design of the outer loop controller and other things needing the rotational friction of the hovercraft, the natural value ψ is always added to the artificial value. This inner loop controller turned out to be an effective method of keeping the hovercraft stable in the rotational mode and allows good performance in both experiment and simulation.

Experimental Trials and Results

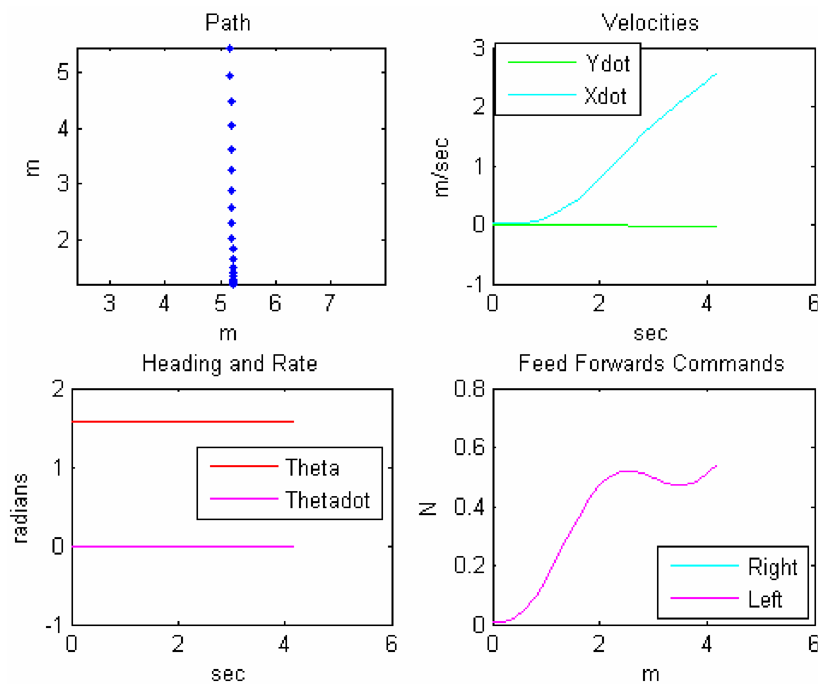
There were three main periods of data collection and experimental trials conducted over during the term. The first set of trial will not be covered in this document for reasons mentioned below. The second set of trials and the third set of trials represent major testing sessions after large improvement campaigns and architecture changes in the code. Numerous runs were conducted for the express purpose of debugging the controller itself, and most of those short runs were recorded, but are unworthy of analysis because of the sometimes major mistakes in the code at the time of their run. This document does not examine every successful run, but it does cover the trials with a meaningful result.

The first period was testing the initial implementation of the controller. This trial took place before the midterm reports and was reported on in those reports. It was subsequently discovered that any appearance of correcting action by the controller was with a very high likelihood just a fluke due to a number of serious bugs in the code. For example the controller was mistaking the reference trajectory's time stamp column for its x position data, x data column for the y data, the y data for the heading data and so on. Also the fan outputs were inverted from what their correct setting. Accordingly there will be no analysis of those trials in this document.

The second set of trials was the result of sorting all bugs out of the initial controller implementation. Te evolution of the controller software and test conditions will be laid out along with the results of the trials. During the third major set of runs the hovercraft hardware failed in such a way that it was no longer able command its actuators. This event ended our testing and subsequent controller improvements were developed and tested in simulation.

Experimental Reference Trajectories:

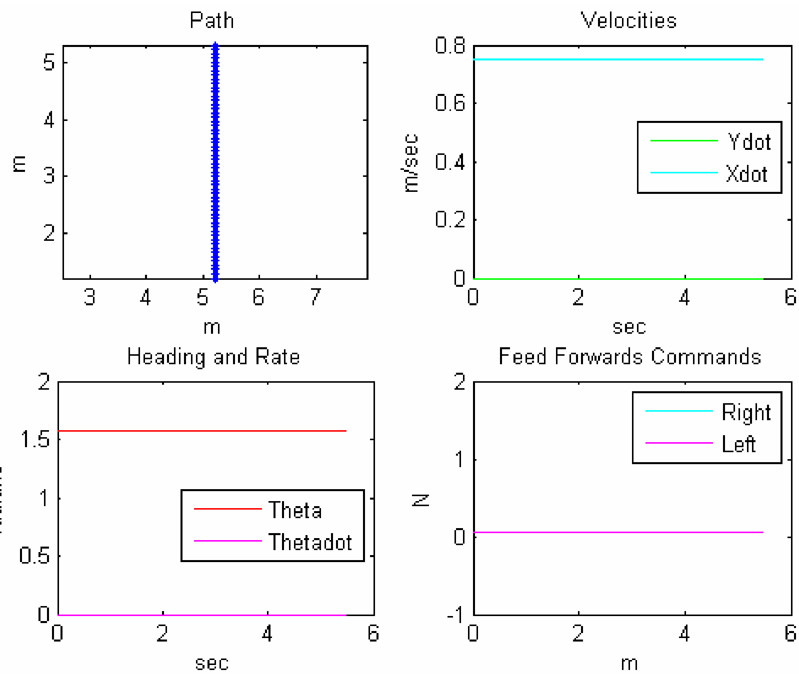
There were two reference trajectories used in the experimental trials. Both of them were meant to be simple straight line trajectories, with their difference being the reference speed along the trajectory. The first trajectory was an attempt to find a feasible trajectory that began with zero velocity and steadily accelerated down the path. This trajectory, while much more complicated that a straight line constant velocity trajectory, was created because all of our testing began with the hovercraft having no initial velocity. The trajectory was represented as a series of points containing the full state of the vehicle as well as the feed forward commands and these points were spaced out so the hovercraft would encounter them every fifth of a second assuming perfect velocity tracking. This spacing proved to be an unwise choice given the 20hz loop time of the controller, and due to slower than nominal velocity tracking the real rate of encountering new trajectory points was sporadic and would increase as the flight went on. A plot of the Simple.traj trajectory data is shown below.



Simple.traj Trajectory Data

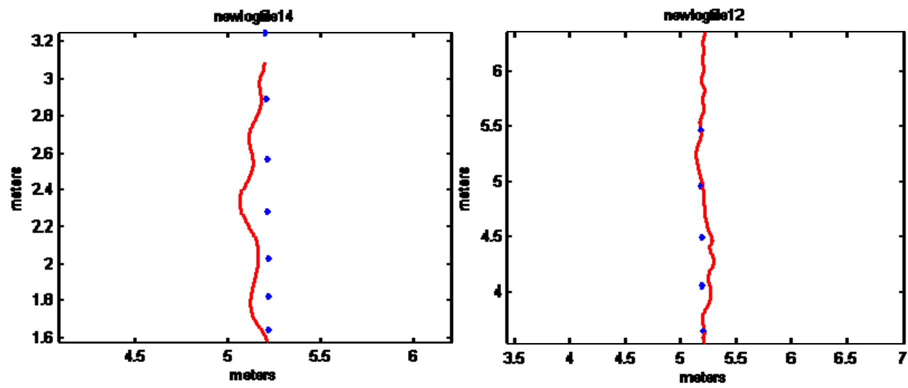
After the shortfalls of the Simple.traj trajectory were realized and understood a new baseline reference trajectory called Straight.traj was created. The majority of our meaningful experimental trials were conducted using this trajectory. This trajectory assumed a constant uniform velocity of 0.75 m/s and was perfectly straight. The points on this trajectory were spaced to be encountered ten times a second. A plot is also shown of this trajectory. A cursory comparison can be made by viewing the spacing and quantity of the blue dots representing the trajectory points in the upper left hand subplot of the previous and following figures. It was found that the assumption of a non-zero starting velocity did not significantly affect the performance of the hovercraft when using Straight.traj. The simplicity of this trajectory also aided the analysis of the logged data from the runs and led to more rapid improvements in the controller design and gains.

The controller structure used for the trials consisted of a simple proportional gain on the lateral error, the heading, and the heading rate, and a proportional gain for the velocity. The velocity error was being calculated incorrectly however, leading to exceptionally bad tracking. Fortunately the feed forward terms were sufficient to move the hovercraft in most cases. An anti wind-up integrator component was added to assist with the velocity tracking, but this was not enabled during any of the runs. The controller had a different set of gains for each fan, and these gains were negative of the other. This scheme was later abandoned because it gives bad performance in any case involving significant error.



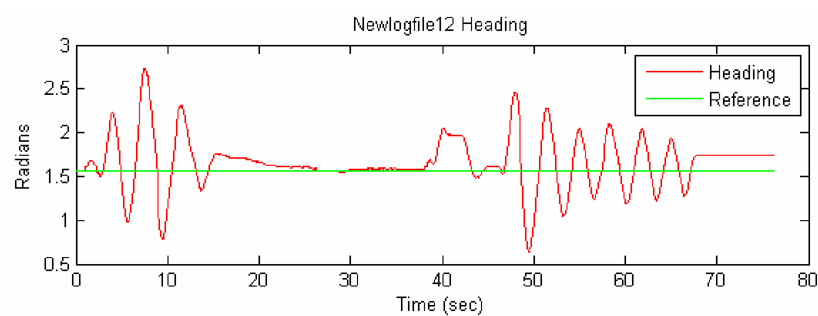
Straight.traj Trajectory Data

Simple.traj Results:

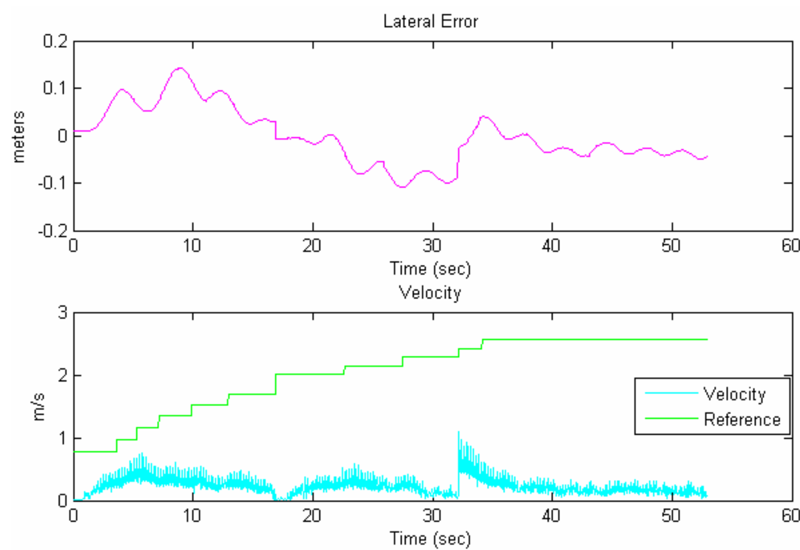


The trials above represent the best performance achieved on the Simple.traj trajectory. The blue dots are the data points of the trajectory, and the red line is the path traced out by the hovercraft. All plots show the hovercraft starting with correct x, y, and heading initial conditions. The left plot above ends where it does because of a piece of

tape on the testing surface that would catch the hovercraft skirt as it went by. This piece of tape would otherwise lie flat and was not discovered as the culprit causing the hovercraft to stall midway down the reference trajectory until late during the third set of trials. It was initially believed that the reference velocity of the trajectory was too low in this region to sustain forward flight, and accordingly the right plot shows the completion of the reference trajectory with initial conditions located after the stall point (the piece of tape.) One can notice the waviness of the hovercraft's path. This was caused by large instabilities in the heading of the hovercraft as shown in the plot below.



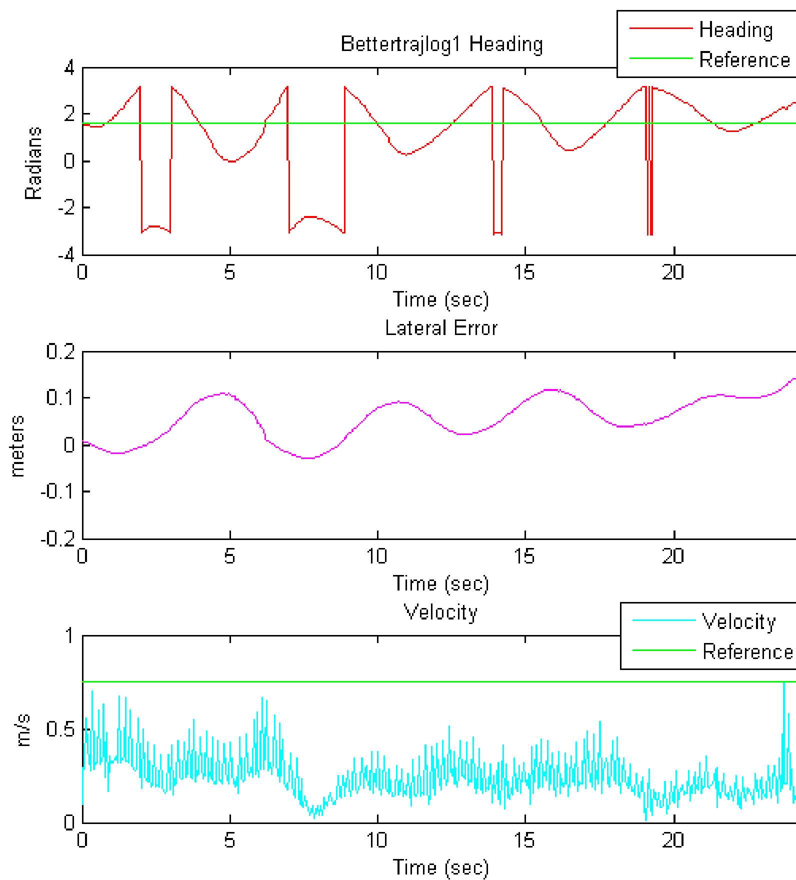
The period between 16sec and 48sec is when the craft was stalled. It was freed by a couple taps administered starting at 40sec. At ~68sec the vision system lost sight of the craft when it went out of view of the cameras.



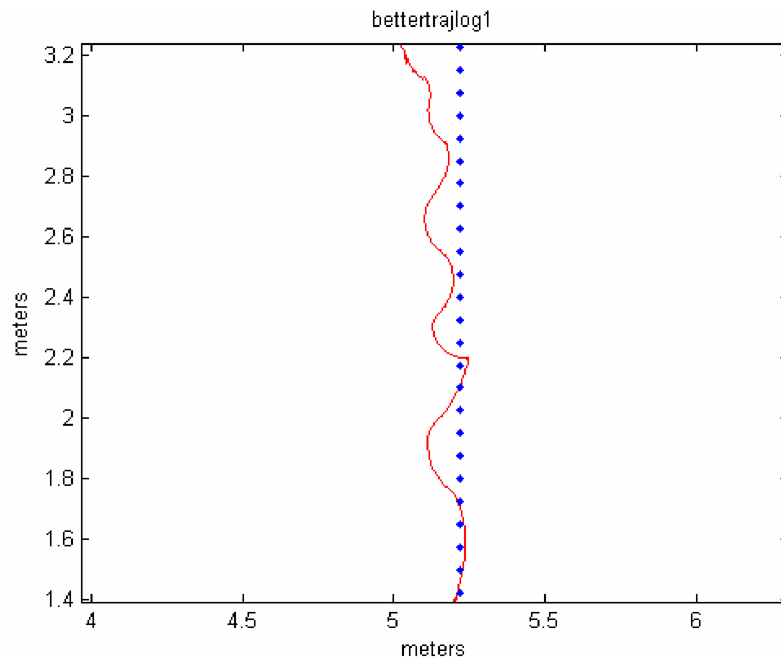
Combined lateral error and velocity performance for the two runs. The discontinuities ~ 17sec and ~32sec are where data collected while the craft was stalled is cut out of the plot. Notice especially bad velocity tracking. The spike in the velocity at ~32sec was from an external kick to budge the craft from its stalled state

Straight.traj Results:

We noticed that the waves occurred with approximately the same frequency of the data points as they were encountered spatially, and one of the reasons behind the switch to the Straight.traj trajectory was to test if this data point frequency had any effect on the heading instability. As can be seen in the following plots, switching to Straight.traj did not fix the issue. The same mid trajectory stalling because of the piece of tape also occurred during this test.



Performance of bettertrajlog1; times when the hovercraft was stalled were removed from the data for ease of analysis. Note instabilities in heading persist. The jumps in the heading are because heading was constrained to lie between plus and minus pie.



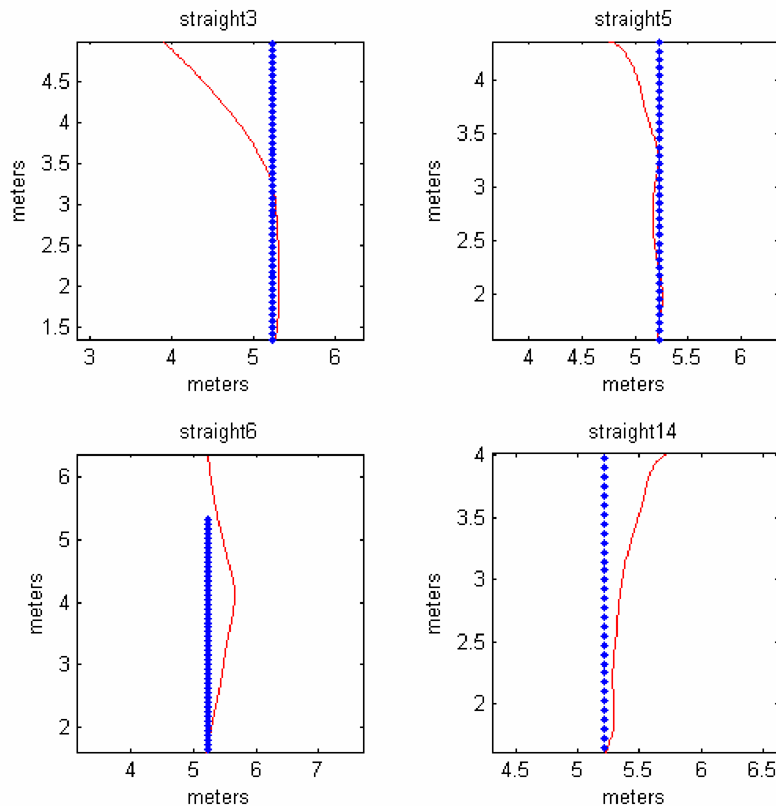
Controller Improvements:

One can clearly notice the noise in the velocity signal, and to deal with this as well as fulfill the requirements of the project an extended Kalman filter in predictor corrector format was implemented in the code. While some limited offline testing was done of the EKF, the hardware failure occurred before it was integrated into the main program flow, and no experimental records of its use were made. The other major improvement was the implementation of an inner loop controller to dampen out the heading instability of the hovercraft, as well as the shifting of the rest of the control code to a desktop. Previously all control code had run on the Zaurus.

The trials and the dynamic equations show that the rotational mode of the hovercraft is the most unstable. Due to the large time delays and noise in the vision system information involving the heading and its rate, the any controller relying only on the vision system will be unable to stabilize the heading sufficiently. However, the hovercraft has an onboard gyroscope with insignificant time delay and low noise. This gyroscope gives the heading rate of change, and this sensed input is used to artificially increase the rotational friction as seen by the outer loop controller. The gyro reading in radians/sec is multiplied by a small "artificial rotational friction" parameter to calculate the counter torque value, and this value is then divided by the moment arm of the fans to calculate the fan command to be added into the outer loop fan commands before finally being sent to the fan motors. This inner loop controller turned out to be an effective

method of keeping the hovercraft stable in the rotational mode as will be seen in the results below.

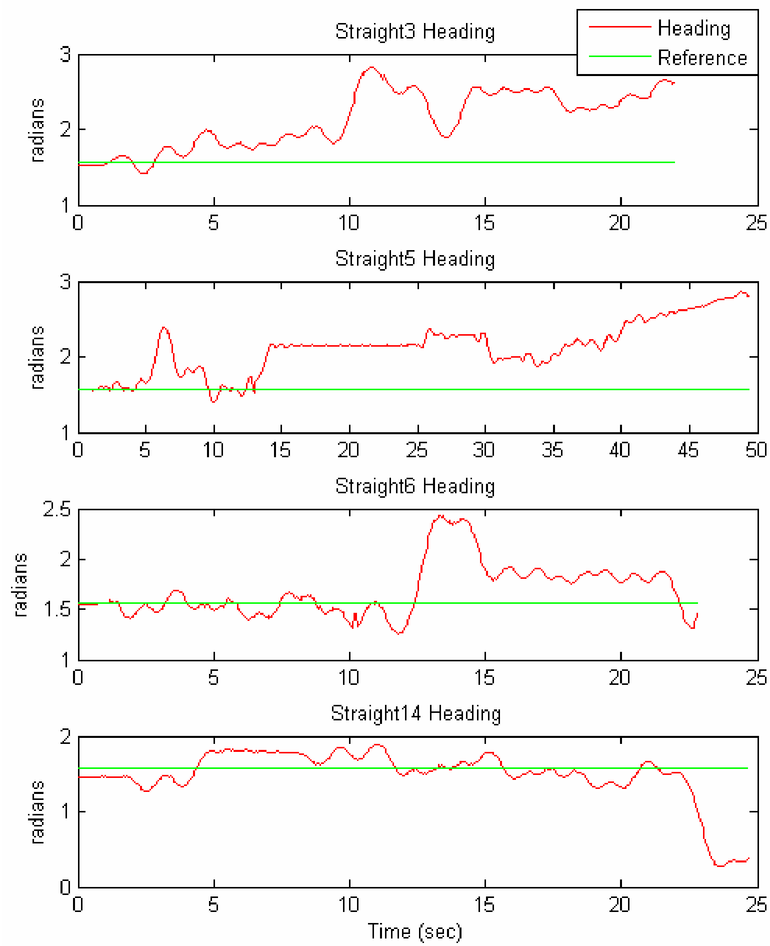
Inner loop Controller Results:



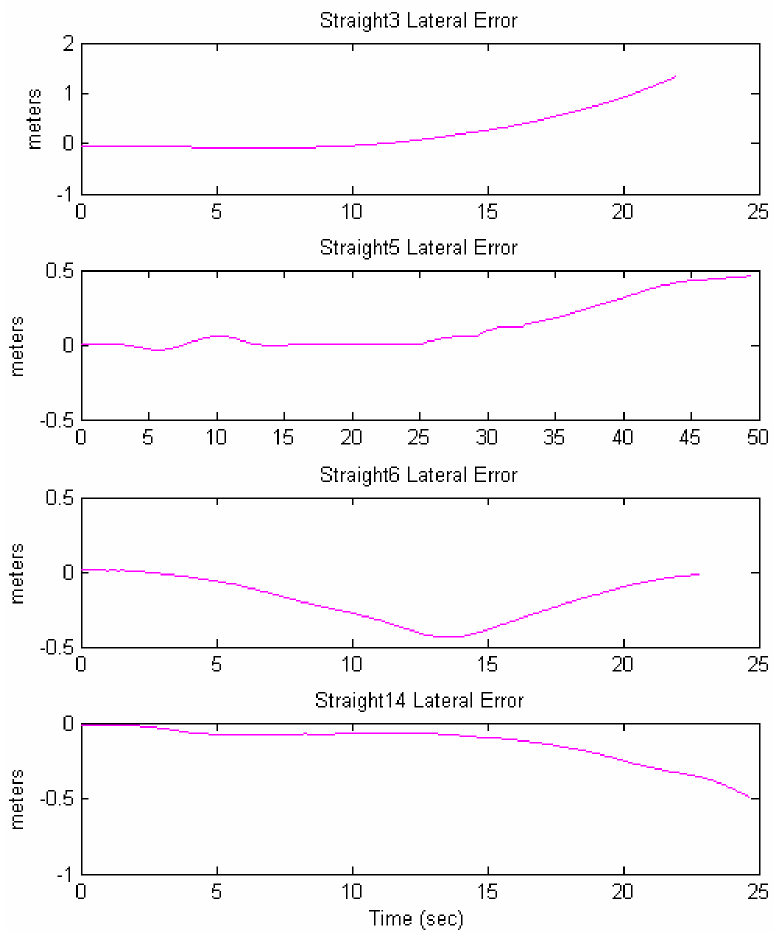
These runs lack the waviness of previous runs, and show the effectiveness of the inner loop controller.

The lateral tracking performance of the third set of trials was worse than that of the second set of trials, but this was due to a number of factors. Primarily the testing time available to correctly tune the inner loop controller was cut short after 15 runs due to hardware failure. The outer loop logs show evidence of the inner loop overpowering the outer loop and keeping the craft from being able to correct itself once it had wondered away from the reference trajectory. No logging was done in the inner loop controller to keep its loop time as fast as possible. The other factor was a problem with the way the fan outputs were constructed. This problem would cause the outer loop commands to essentially saturate one fan or the other when the lateral error exceeded a small range and combined with the strong inner loop compensating for the rotation with the other fan the

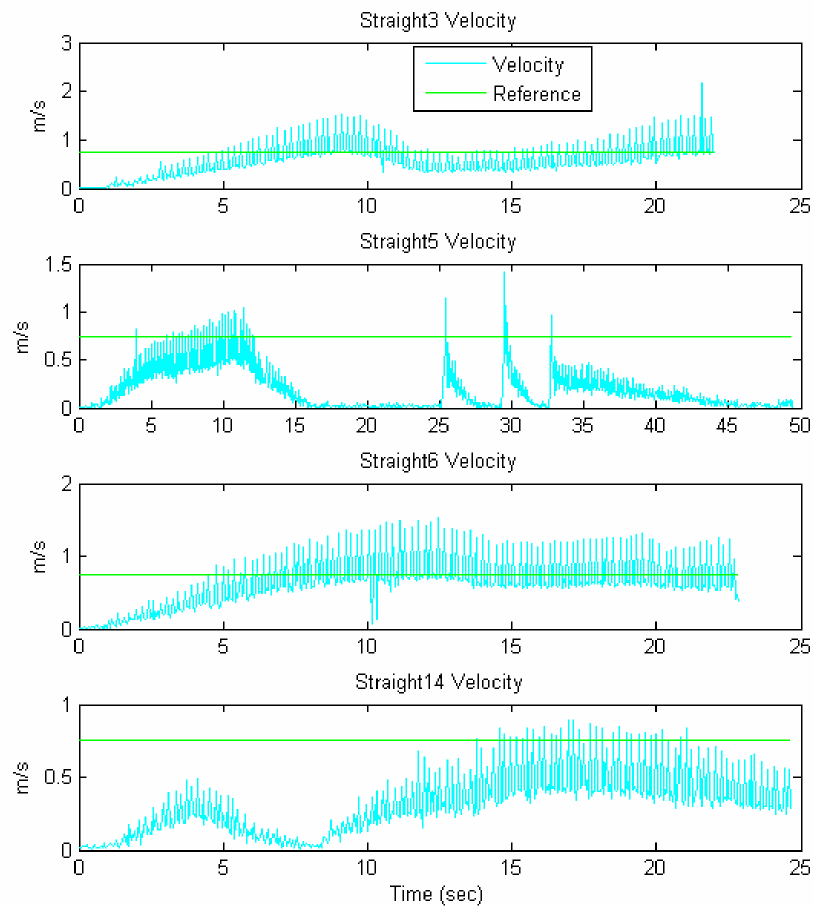
hovercraft would rapidly accelerate out of control. Plots of the heading over the four runs are shown below to compare with the plots above.



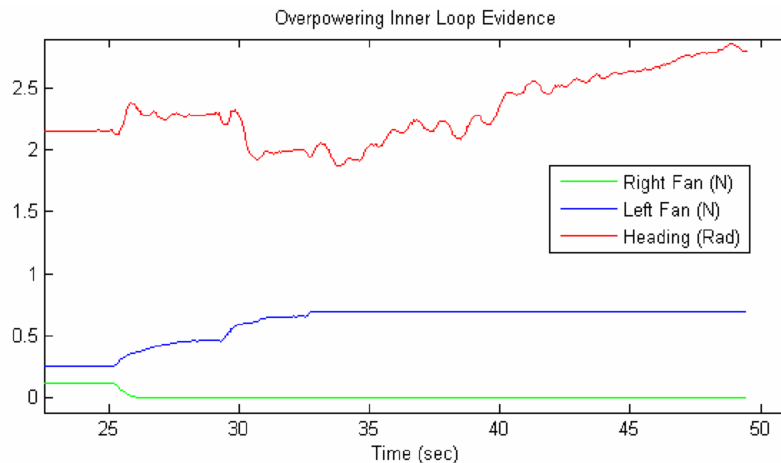
The constant heading between 15sec and 30sec in Straight5 was the result of a stalling event freed with a few taps.



The bad lateral tracking has been explained as a result of an overpowering inner loop and an error in the method of calculating the fan forces.



The velocity tracking was much more uniform with these trials. The three spikes in velocity at ~25, ~30, and ~33sec in Straight5 were the taps that freed it form being stalled on the piece of tape.



Zero heading is pointing to 3 o'clock on a clock face or along the positive x axis on a standard top down view of the hovercraft. Increasing heading corresponds with a counterclockwise turn.

The plot above is a close up of the fan commands being generated by the outer loop controller alongside the heading of the hovercraft for the same period of time. The outer loop controller is maxing out the left fan and zeroing the right fan to try to correct for a substantial lateral and heading error towards the end of the run called Straight5. If one notices the heading and also the position of the hovercraft in the final part of the run (see the figures 4 pages back) it is obvious that the correct action is to turn clockwise and thrust back towards the reference trajectory. The outer loop controller is attempting to do just that with its choice of fan commands. However the inner loop controller's artificial rotational friction term was too large in magnitude and it kept the clockwise turn from happening. In fact the hovercraft gradually turns counterclockwise over this time period, directly contrary to the commands of the outer loop controller. The too strong inner loop was the main culprit preventing good lateral performance. The correct initial conditions for the four featured runs were the primary reason for the initially correct trajectory tracking. As mentioned before the hardware failed before we were able to adequately tune down the inner loop performance to allow good tracking.

Analysis of the controllers above is absent form this document due to a number of factors. The gains and parameters in these runs were unrecorded, the programming was later found to contain errors that render traditional analysis methods unusable, and the structure of the controllers themselves used in the trials did not attempt to decouple the lateral and longitudinal dynamics, resulting in complicated mimo system that is also difficult to analyze.

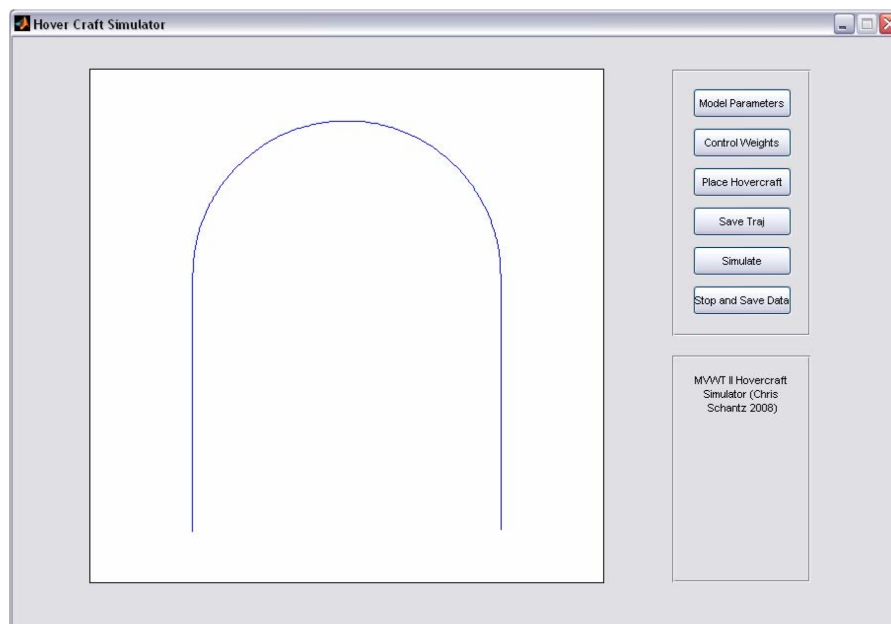
Matlab Simulator Design and Documentation

Introduction:

A Matlab based simulator was designed to serve as a tool to aid in control design and testing for the MVWT II platform. This document details the features and use of the simulator in its current form as of the end of March. The simulation environment is accessed through a graphical user interface with the idea of its future employment or adaptation for uses with other vehicles. It uses recorded samples of sensor noise and a reconstructed noisy fan force map, as well as a settable time delay on the measured state to achieve a high level of similarity with recorded experimental results.

Interface:

To open the simulator, put all files in the Matlab_hovercraft_simulator file into a directory and add them to the Matlab path. Then open and run the file Sim_GUI.m. The main screen is shown below and it will automatically generate an inverted U-shaped trajectory and show that trajectory in blue on the environmental axes plot. The characteristics of this trajectory are discussed in its own section, but the environmental axes plot represents a 10m by 10 m top down view of the hovercraft's world with the origin in the bottom left of the plot.



There are six buttons to enter model and simulation parameters, modify the LQR weights for the controller, place the hovercraft into the simulation, save the u-shaped traj to a file, activate the simulation, or stop the simulation. The details of these six actions are explained further below, and then the internal features of the simulation are discussed as well as an explanation of the controller architecture employed within the simulation. Comparisons with real experimental data are done where possible, mainly to illustrate the validity of the simulation in non-nominal situations. This is due to the limited experimental data obtained before hardware failure became an obstacle to further experimentation. However in the author's opinion the simulation so successfully reproduces the qualitative behavior of the real hardware that a direct transfer of controller architecture and gains from the simulation to the real hardware would result in very good performance without any significant amount of tuning.

Model and Simulation Parameter Window:

Model and Simulation Parameters

Parameter	Value
Mass (kg):	0.75
Rotational Inertia (kg m^2):	0.00316
Rotational Friction (kg m^2/s):	0.005
Linear Friction (kg/s):	0.15
Fan Separation (m):	0.178
Max Fan Thrust (N):	0.7
Artificial Rotation Friction (kg m^2/s):	0.01
Coupling Factor (0-1):	0
Loop Frequency (Hz):	20
Time Delay (miliseconds):	30

Accept Default

Clicking the "Model Parameters" button on the main screen will bring up the window at left. Each text input box has limited protection against the entering of non numbers or negative constants where this would not make physical sense. Note that the Artificial Rotational Friction term can be made negative if one desires, but expect unstable behavior. Of note are two terms that may not be immediately obvious: the Artificial Rotational Friction term and the Coupling Factor term. These terms are most easily explained in the later section on the controller structure. Excepting the Loop Frequency term, which defines how fast the control loop runs, and the two terms previously mentioned, the remainder of the terms are physical constants of the simulation and are derived by measurement of the hovercraft and network timing. Pressing the Default button will restore the default parameters that describe the system MVWT II system. Using a different set of parameters it should be possible to simulate the original MVWT ball caster craft.

Control Weights Window:

The simulation uses an LQR controller to control the lateral dynamics and a simple proportional controller to regulate the velocity of the hovercraft. The weights for the four sub-states involved in the lateral dynamics as well as the single gain for the velocity controller can be set in this window reached by clicking on the Control Weights button. Not a number protection is also present on these input boxes, but any parameter here can have any sign.

Control Parameters

Lateral Distance	10
Theta Weighting:	15
Lateral-dot	5
Thetadot Weighting:	0
Velocity Gain:	4

Accept

Place Hovercraft Window:

Place Hovercraft

X position:	0
Y Position:	0
X Velocity:	0
Y Velocity:	0
X Acceleration:	.0001
Y Acceleration:	0

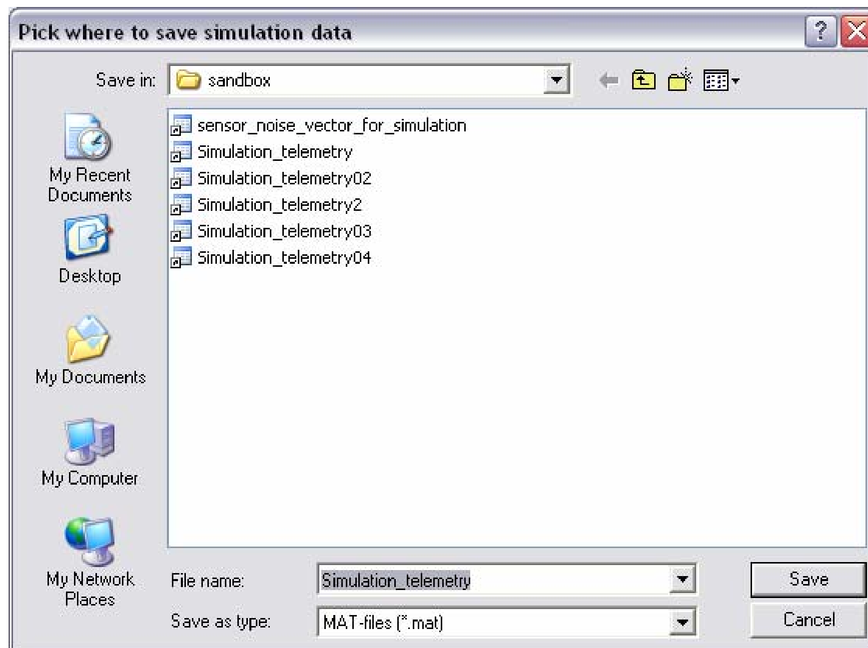
Accept   Use Mouse Cursor

The place hovercraft window is how one introduces the hovercraft to the simulation environment and gives it initial conditions. There are two ways to do this. If exact initial conditions are desired they can be entered right into the window and then the “Accept” button can be pressed. The default small non zero number for the X acceleration is to insure that the heading does not become undefined. For cases where ease of use is important or the exactness of initial conditions is not required, one can click the “Use Mouse Cursor” window and use the three click placement method. Clicking this button will close the place hovercraft window and ignore any values specified in it. The simulator will then record the position of the mouse cursor during the next three clicks of the mouse inside of the simulator window. The three click placement method works as follows: the first click specifies the coordinates of the hovercraft. The second click is used to create a vector between the location of the first click and the second click and this vector defines the direction of the initial velocity of the hovercraft. The magnitude of the velocity is proportional to the distance between the first and second click. The third click is treated in a similar way to the second click and it is used to define the heading of the vehicle, and its distance from the first click gives the magnitude of the initial acceleration of the hovercraft. The hovercraft we are using can only accelerate in the direction they are facing, but this initial acceleration has little effect because it is dependent directly on the fan commands, and after the controller has had a chance to run once through its loop it

will rapidly vary the fan commands. See the section titled Visualization Details for an explanation of the animated graphics of the hovercraft directly after placement and during "flight".

Save Trajectory and Simulate:

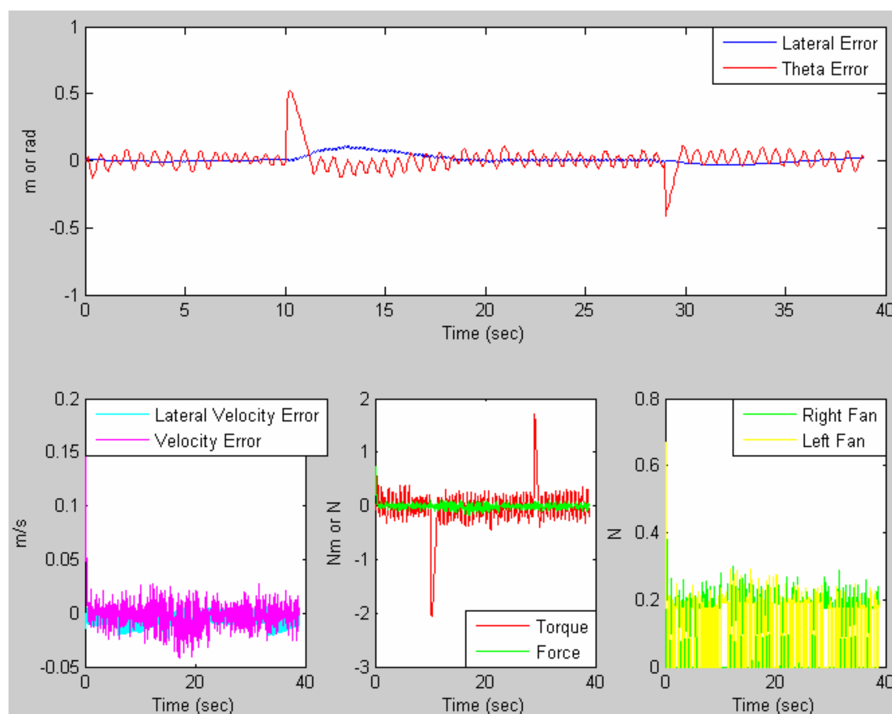
Both of these buttons are fairly simple. They both open a standard save file dialogue box and request a file name to save either the data for the U-shaped trajectory or the logged data from the simulation. If the trajectory is being saved it is saved in a format directly useable by our software running on the hardware, though one might wish to adjust the parameters of the trajectory in the code to make it fit on the actual 6m by 6m testing surface of the MVWT.



The trajectory is saved directly after the file name is chosen. If one is selecting a log file name after clicking the "Simulate" button, the name is stored by the program and not used to save the simulation logs until the "Stop and Save Data" button is pressed. After picking a log file name the simulation begins immediately. A warning: do not ever overwrite the "sensor_noise_vector_for_simulation.mat" file. This file contains the recorded samples of sensor noise from the vision system and the reconstructed noisy fan force map data and the simulation will not run without it.

Stop and Save Data:

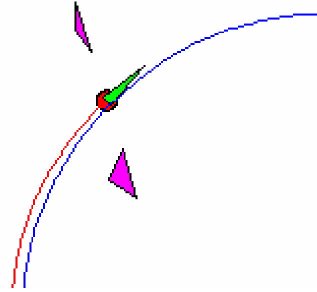
This button will end a simulation and save the logged data to the .mat file specified when the simulation began. It will also open a plot of the error logs so that one can quickly judge the quantitative performance of the flight. An example plot is given below. Simulation runs can also finish by completing the given trajectory, but even after this the "Stop and Save Data" button needs to be pressed to record the logged data and display the chart.



Visualization Details:

The simulation provides a cartoon hovercraft display and animates it as the simulation progresses. At any given moment animation visualizes the magnitude of the command being sent to each fan, the heading of the hovercraft, the velocity of the hovercraft and the magnitude of the velocity, as well as the traveled path of the center of the hovercraft. The default U-shaped trajectory is also always displayed. More than one simulation can be done sequentially. Simulation runs can end by wither completing the trajectory or by pressing the "Stop and Save Data" button. After this a new simulation can be started from the most recently uses initial conditions by pressing the simulate button again, or the hovercraft can be placed elsewhere in the environment before re-

simulating. In either case the traveled path of the previous simulations are preserved for comparison reasons. The only way to clear a build up of old traveled paths is to quit the simulator and restart it. To the right is a picture of the hovercraft mid flight. The red circle represents the hovercraft. The green triangle points in the direction of the instantaneous velocity, with a length proportional to the magnitude of the velocity. The magenta/ pinkish triangles both point to the rear of the hovercraft, and their lengths are proportional to the command being sent to the left or right fan at that moment respectively. The red path is the traveled path of the hovercraft and the blue line is the trajectory it is trying to follow. The direction that the green triangle is pointing should not be confused with the "front" of the hovercraft. Use the magenta triangles to determine that information as the hovercraft can only accelerate in the direction it is facing. The magenta triangles will collapse to being only a line when the fan command being sent is zero.

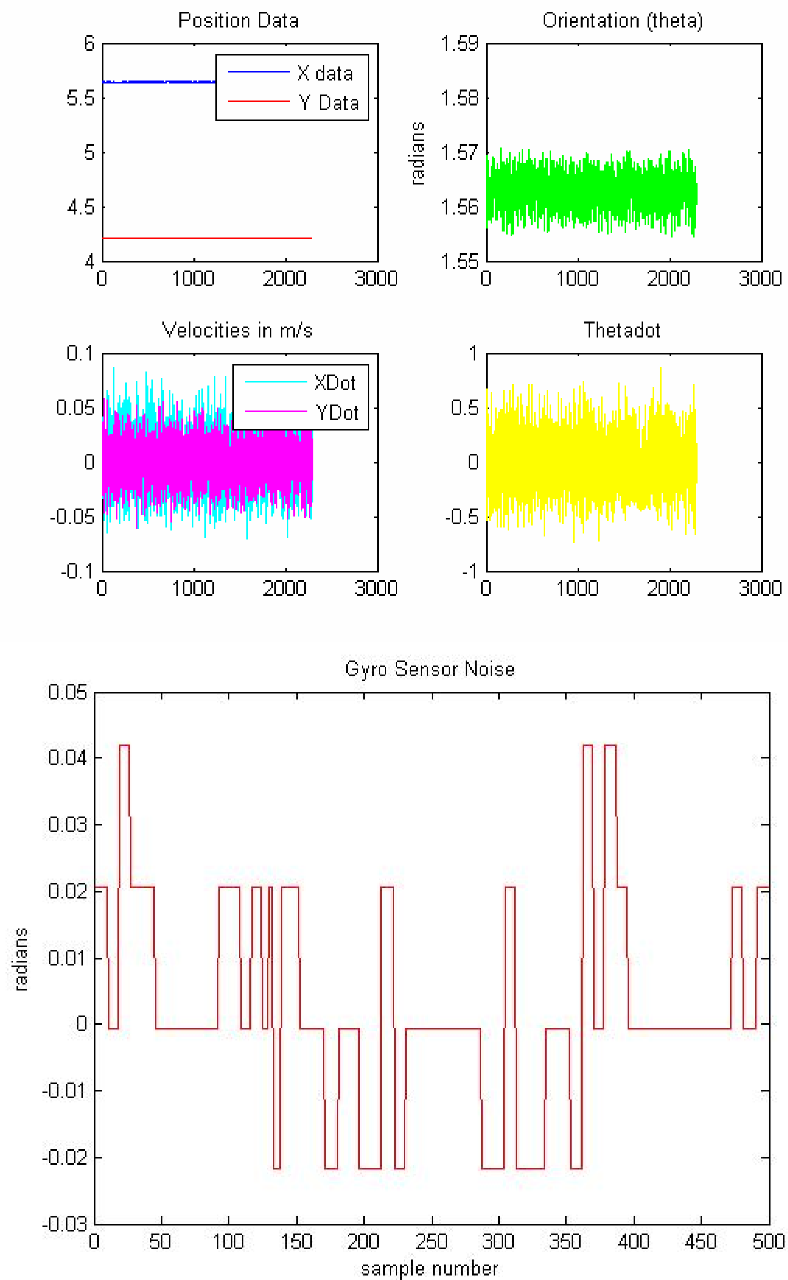


Simulation of Sensor Noise:

Real recorded noise samples from both the vision system and the onboard gyroscope are used to simulate the sensor information being received by the controller in the hovercraft. Three sets of vision data were taken at different orientations and positions on the test floor to qualify the noise characteristics of the vision sensors. This was done with a still vehicle based on the assumption that we will never achieve velocities that will lead to motion blur, and the assumption that the vision algorithm uses only frame by frame information and has minimal reliance on previous frame data when examining the current frame. The noise characteristics of the three data sets are displayed below. The second data set contained the widest standard deviation. The mean was reliably computed by a simple average over the entire set, as the craft being observed was still during the entire time of observation. This average was treated as the true value and subtracted from each individual data stream to leave a zero mean true sample of sensor noise. The same process was repeated for the gyroscope by recording a length of data under a constant angular rate. The zero mean sensor noise samples are matched with an estimated time stamp series based on the update rates of the vision system and the loop time of the gyro reading code respectively, and sample noise values are generated from it by linear interpolation given a specified sample time. These noise values are added to the simulators internal exact state before use by the control code. Plots of the second data set from the vision system, as well as the gyro sample noise are on the next page.

Data Set	X	Y	Theta	Xdot	Ydot	Thetadot
1	0.00025882	0.000095666	0.0028256	0.021598	0.0079263	0.22927
2	0.00029827	0.00023202	0.0029033	0.02556	0.018989	0.24625
3	0.00019648	0.00006082	0.0020149	0.016221	0.0051505	0.16293

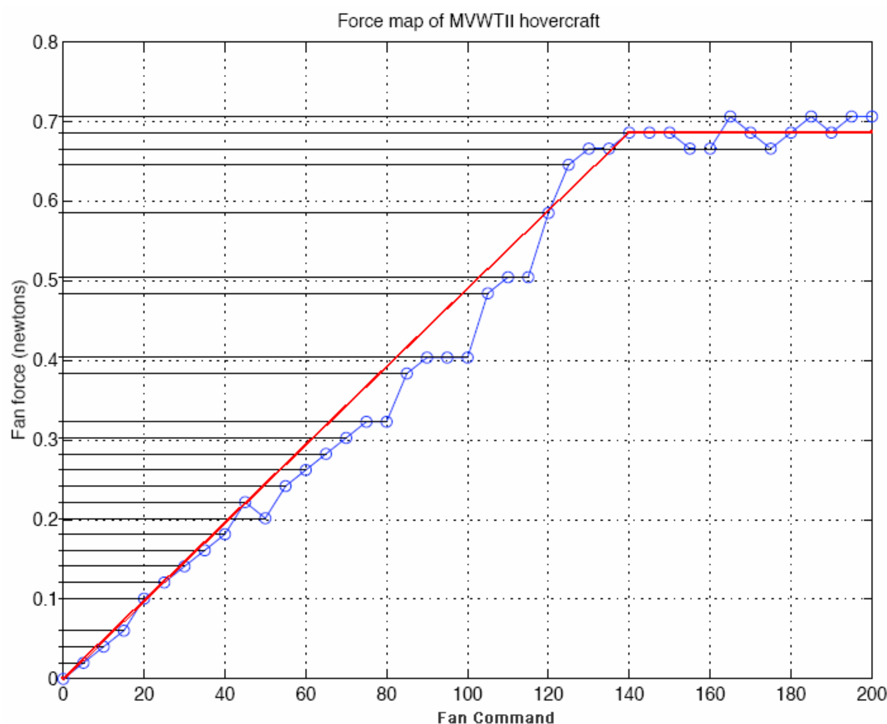
Vision Sensor Standard Deviations

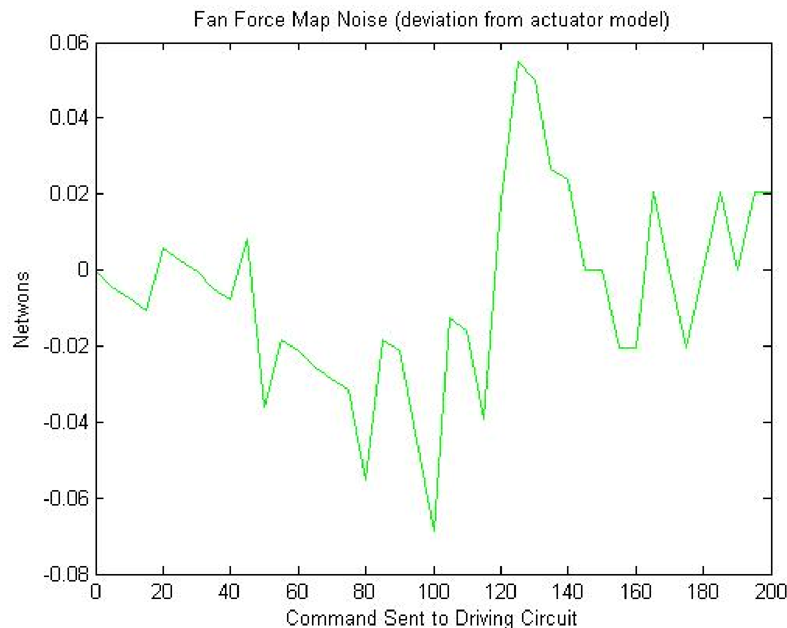


Position Data
X data
Y Data
Orientation (theta)
radians
Velocities in m/s
XDot
YDot
Thetadot
Gyro Sensor Noise
radians
sample number

Stochastic Fan Force Map Reconstruction:

The following plot was copied from the "MVWT-II: The Second Generation Caltech Multi-vehicle Wireless Testbed" document. It represents the measured force from one of the side mounted ducted fans compared to the numerical command sent out by the onboard microcontroller to the motors driving circuit. The measurements were performed with a load cell apparatus and hardware sometime presumably in the summer of 2002 or 2003. The blue line represents the measured data from the original plot. The horizontal black lines were added by the author to aid in accurately determining the data set underlying this plot. The information was used to construct a stochastic fan force map for the simulator. The red line is a model of a linear and saturating fan force map that was used subtracted from the blue line data to approximate the noisy fan force map. A plot of the noisy fan force map is also included. The standard deviation of the resultant line was taken and a vector of zero mean Gaussian noise with the requisite characteristics was generated. This noise vector is used to add noise to the fan forces generated by the controller before those forces are integrated by the simulator each time step to update the simulators internal true state. A random small time constant is added to the look up timed in the interpolation algorithm to insure that the left fan and right fan noises are not identical.





Simulator Time Delays:

delay (ms)	Event	elapsed time (ms)
0	LED on/off	0
8	Camera takes picture	8
33	Vision broadcasts data	41
6	Laptop receives data	47
1	Controller receives data	48
6	PIC receives control signal	54
11	LED on/off	65

The vision system represents the most significant time delay in the MVWT II. Again referring to past documentation a chart characterizing the time delays was utilized to quantify their magnitude. As detailed in “A Platform for Cooperative and Coordinated Control of Multiple Vehicles; the Caltech Multi-Vehicle Wireless Testbed,” the chart at right below was generated by replacing a blackened block on one of the vision system vehicle identification “hats” with a bank of LEDs that were programmed to turn on when the vision system detected the presence of the hat on the floor, and turn off when the vision system lost. The resulting cycle of flashing LEDs will have twice the length as the tome delay in the vision system. The simulator’s default time delay is set to 40 milliseconds. The simulator’s time delay code is able to handle time delays longer than the loop time of the controller by evaluating the true internal state at the requisite times interior to the time step at which the controller runs and storing these interior evaluations to a buffer. The “measured” state is then read off from the buffer of old states sensor noise is added to it as detailed above. Only the measurements coming from the vision system are given a time delay. It is assumed that the onboard gyroscope measurements have essentially no time delay.

Kalman Filtering and Simulated Controller Performance Analysis

A Kalman filter was added to the simulator to filter the simulated noisy sensor data. For the development of this filter sensor noise characteristics were taken from the vision data analysis covered elsewhere, and the process disturbances were assumed to be zero. Much effort was expended finding an extended Kalman filter formulation that was numerically stable. Due to the very small noise levels in the vision system for the position and heading states, the integration algorithms were rarely able to converge for full EKF demonstrations. Even discrete time EKFs were not stable. Finally the Schmidt formulation of the EKF was found to be stable for the lateral dynamics.

An attempt was made to add the insert the Schmidt style (shown to right) EKF into the simulator. This attempt was also plagued with numerical stability issues. It was realized after much futile effort that the sensor noise characteristics were no longer valid with the addition of the simulators time delays. Because sensor noise was characterized on a stationary target, the effect of the time delays did not appear. However, when the hovercraft is moving, the sensor measurements will exhibit inaccuracies as a function of both their internal noise levels, and the fact that the data will be on average 40 milliseconds out of date and the hovercraft will have changed position in that time. This realization led to a re-characterization of the sensor and process disturbances. A prediction solver was implemented to update the estimated state of the hovercraft according to the ideal equations of motion. This predicted state was then compared to the stochastic true state maintained by the simulator and the error was found. The standard deviation of this error was taken and squared to generate the process disturbance variance. A similar comparison was done between the time-delayed measured state and the internal simulator's values to regenerate the sensor noise variance.

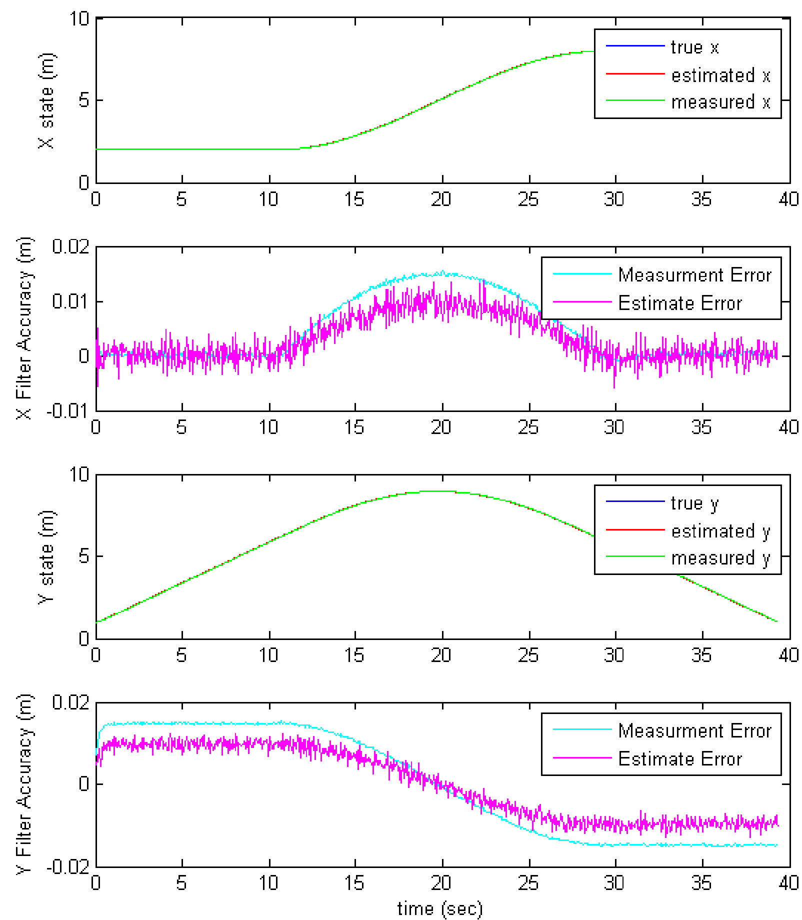
$$\dot{\hat{X}} = f(\hat{X}, U, 0) + L(Y - C\hat{X})$$
$$L = PC^T R_W^{-1}$$
$$\dot{P} = (\tilde{A} - LC)P + P(\tilde{A} - LC)^T + \tilde{F}R_V\tilde{F}^T + LR_W L^T$$
$$P(t_o) = E\{X(t_0)X^T(t_0)\}$$

However even with these more accurate values, numerical precision issues persisted. However I do believe that these values generated above were optimal, as the results of the ode45 function calls would stay bounded and stable and converge rabidly towards the correct values right up until the integrator would complain loosing accuracy and quit. Integration to generate the new *P* matrix was abandoned. In its place the Matlab lqe function is called at every time step with freshly linearized dynamics matrices around the predicted state. This formulation combined with some enlargements of the noise and disturbance characteristics was found to be a stable combination and give good results.

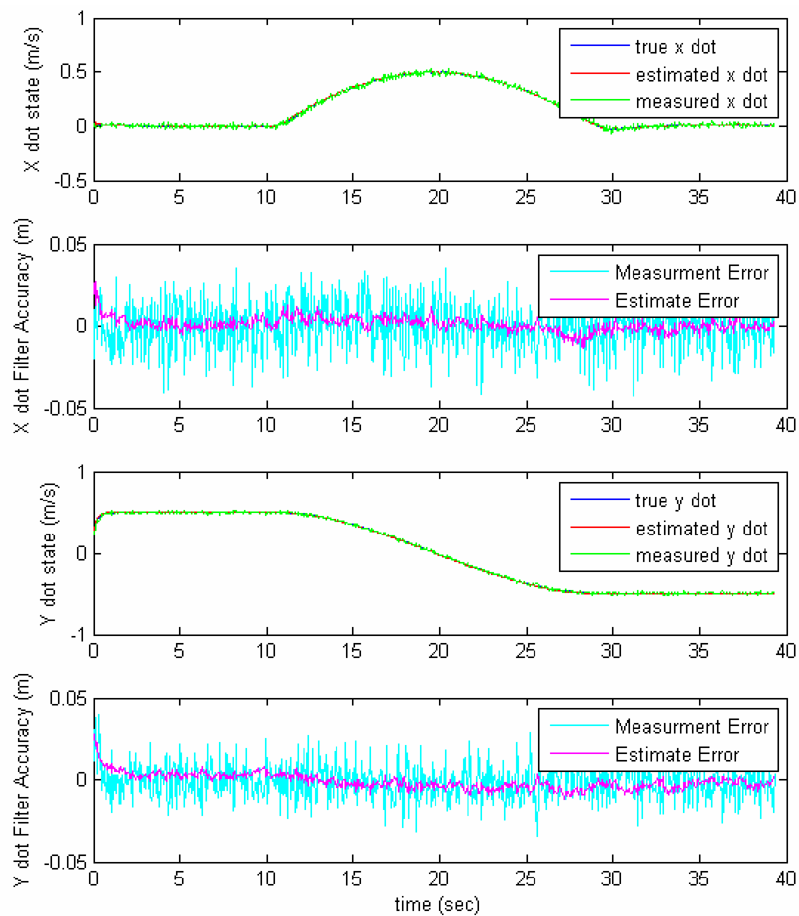
Variances	X	Y	θ	$\dot{X}$	$\dot{Y}$	$\dot{\theta}$
Sensor:	5e-4	5e-4	1e-3	2.5e-3	2.5e-3	N/A
Process:	5e-5	5e-5	1e-3	5e-4	5e-4	0.025

Note: the vision system did such a bad job sensing the $\dot{\theta}$ state that this information is ignored by the filter. Instead the predicted $\dot{\theta}$ state is utilized and this results in much more accurate performance for every state in the filter/ estimator. The noisy $\dot{\theta}$ measurements had been corrupting all the other states.

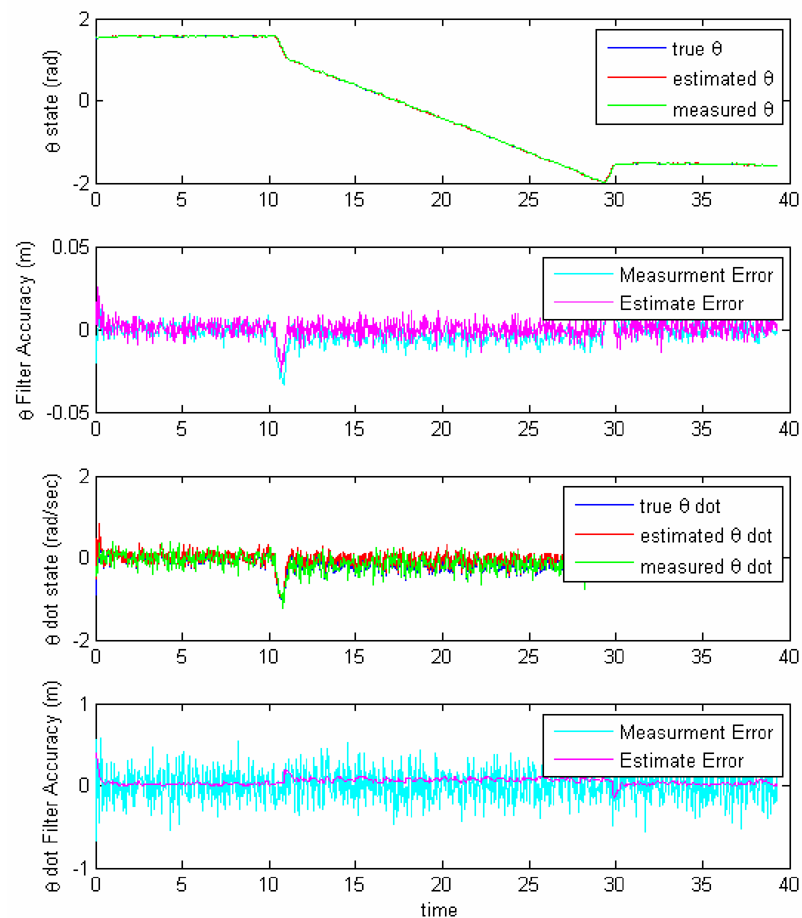
Plots of the filters performance follow:



The performance of the filter here seems lacking. However this was deemed acceptable as it allowed much better performance in the other states of the system. Then effect of the time delays can clearly be seen here as the measurement error essentially switches sign depending on of the value is measuring is increasing or decreasing. What is important here is the relative quality of the estimate compared to the measurement.



Here it can be seen that the filter does a good job smoothing and correcting the velocity measurements.

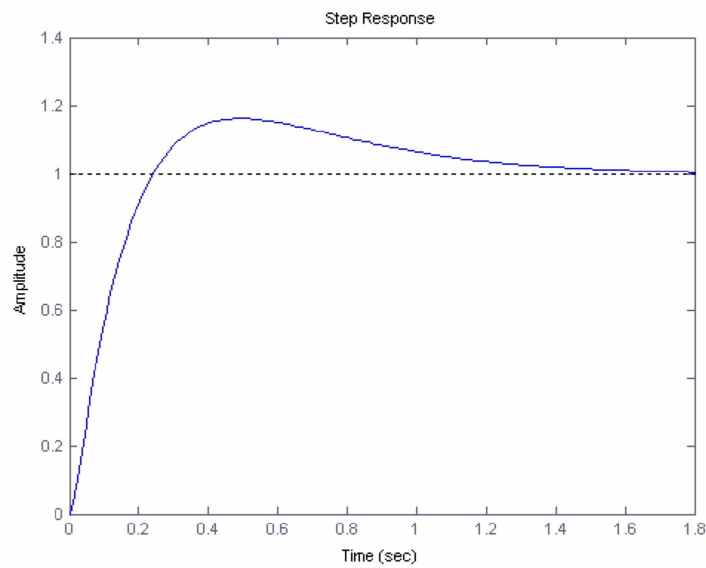


The theta estimate and measurement show similar quality, while the filtered theta dot measurement is much more useful than previously. The spikes here are due to the discontinuities in the trajectory.

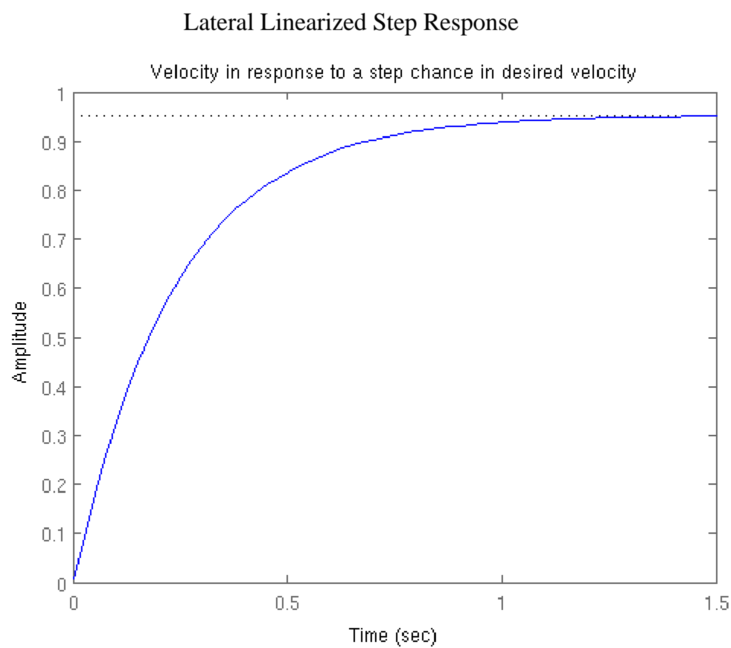
Controller Analysis:

The multiple input multiple output nature of the gain scheduled LQR controller implemented in the simulator makes traditional gang of 4/6 transfer function analysis as well as nyquist criteria difficult to do/ ascertain. However the step response of the linearized lateral dynamics paired with an LQR controller gains with the same weightings as used in the simulator and the average schedule is shown below. While it is possible to remove the overshoot form this response by manipulating the LQR weightings, the real

nonlinear system is quite a bit slower than the linearized system, and these LQR weights were found to work well in simulation.



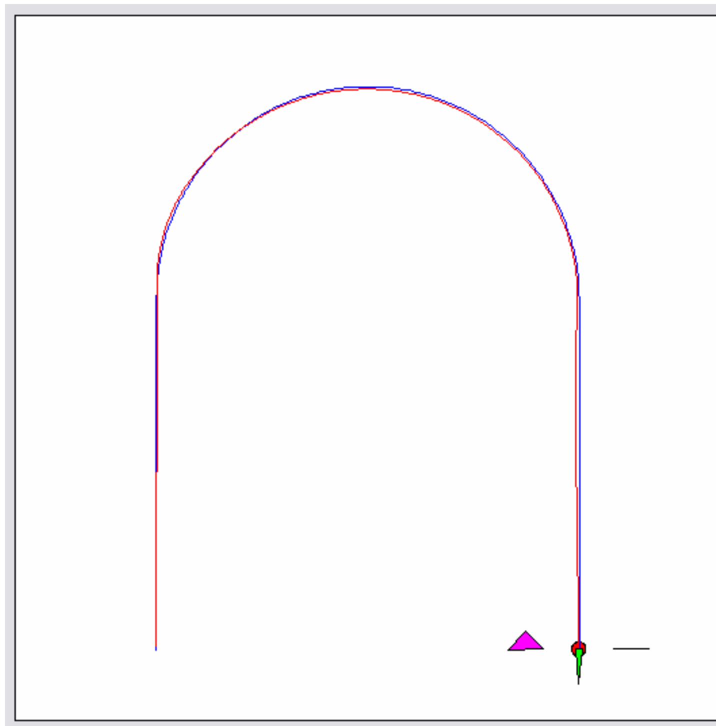
Lateral Linearized Step Response

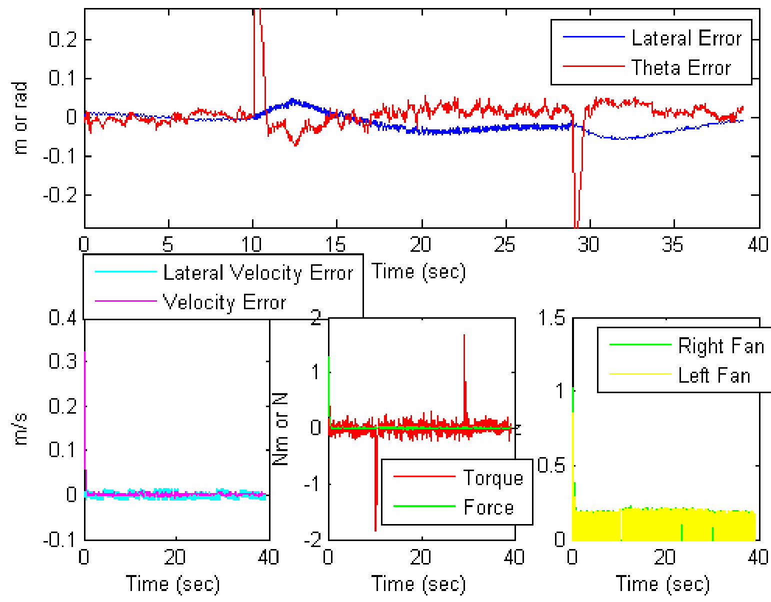


The longitudinal controller used is the weakest controller as not much effort was made to model the varying and random translational friction. Its step response is on the previous page is form a more refined longitudinal controller that shows promise in prototype, but was not needed for the simulation to work well.

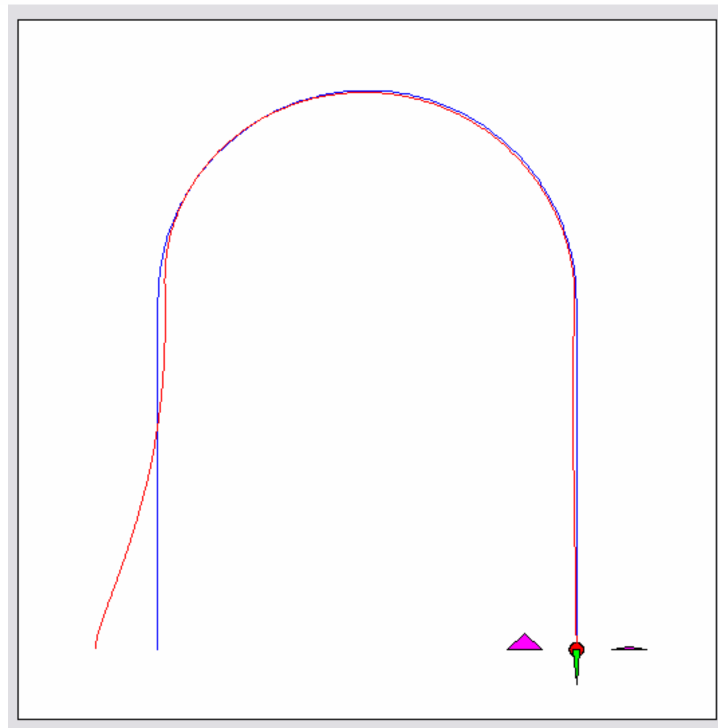
Simulation Runs:

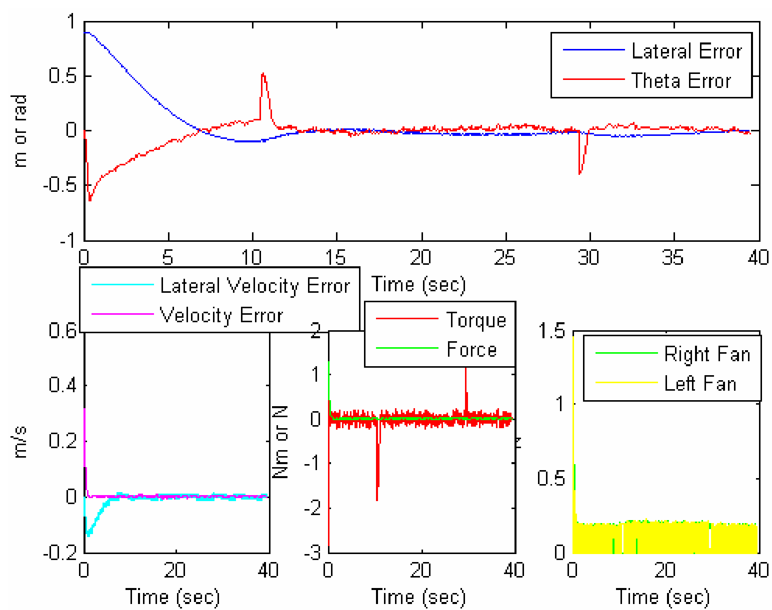
What follows are two runs conducted with the simulator. The LQR weights for the lateral dynamics were: (10, 15, 5, 1) for the four states, and the control weight for the lateral velocity controller was 4. The control cost was one. A screen shot of the overall path is shown, followed by the performance of that path and other information. The accurate filtering of thetadot allowed the controller to virtually eliminate the small wobbles in heading present in the experimental trials. (A sample of the wobble from before the filter was added to the simulation can be seen in the section describing the simulator in detail.) The first trial starts with very little initial error, and the second trial showcases what happens when there is initial error. The two spikes in theta error are a result of the discontinuity of the reference trajectory.





Lateral Error
Theta Error
m or rad
Time (sec)
Lateral Velocity Error
Velocity Error
m/s
Time (sec)
Nm or N
Torque
Force
Time (sec)
Right Fan
Left Fan
Time (sec)





Run with Initial Offset.

Bibliography

[1] Michael J. Kantner. *Falcon Reference Manual*. Division of Engineering and Applied Science - California Institute of Technology - Pasadena, CA 91125, 1995.

[2] Richard M. Murray. *Sparrow Reference Manual*. Division of Engineering and Applied Science - California Institute of Technology - Pasadena, CA 91125, 2004.

[3] Jonathan R. Stanton. *A Users Guido to Spread*. Spread Concepts LLC, 2002.

[4] w/o. *8-bit AVR Microcontroller with 128K Bytes In-System Programmable Flash*. ATMEL, 2008.