École Polytechnique Fédérale
de Lausanne

Semester project

# Hardware development for the riderless motorcycle

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

*Student:*
Quentin Herzig

*Assistants:*
Harsh Shukla
Luca Fabietti

*Teacher:*
Colin Jones

June 10, 2016

# Contents

# List of Figures

# List of Tables

# Chapter 2

# Hardware

The Riderless Motorcycle hardware consists of many parts :

- A ThunderTiger SB-5 motorbike
- A Raspberry Pi 2 Model B
- A EMLID Navio2
- An Arduino Nano
- A remote control emitter
- A remote control receiver
- A EMLID Reach RTK station
- A EMLID Reach RTK rover
- A PC or smartphone

A full diagram of the entire hardware is presented in Fig. 2.1. The upcoming subsections describe each part separately.

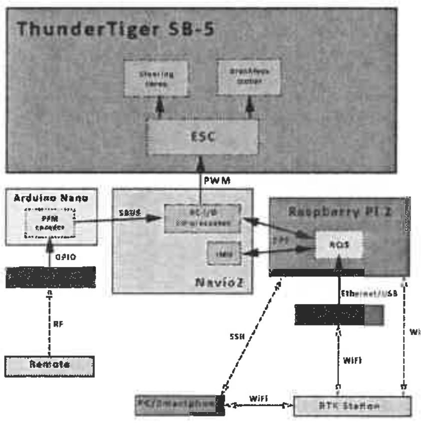


Figure 2.1: Hardware diagram.

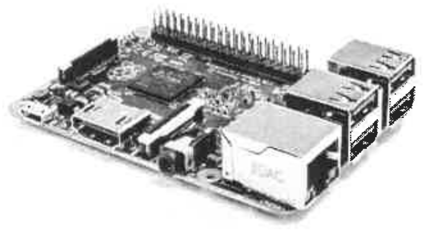


Figure 2.3: Raspberry Pi 2 Model B.

## 2.3 Navio2

The Navio2 (Fig. 2.4) is a board from EMLID that plugs in on top of a Raspberry Pi 2 or 3. It includes 2 Inertial Measurement Units (IMU), a barometer, a Global Naviguation Satellite System (GNSS) receiver, a RC I/O co-processor that accepts PPM/SBUS[1] and 14 Pulse Width Modulation (PPM) outputs. A list of all features and specifications is shown in Table 2.2. It is used for this project for its compatibility with the Raspberry Pi 2 Model B, its already written drivers in both C++ and Python, and its large community support.

| **Features** | **Specifications** |
|---|---|
| • MPU9250 9DOF IMU | • 14 PWM servo outputs |
| • LSM9DS1 9DOF IMU | • PPM/S.Bus input |
| • MS5611 Barometer | • Triple redundant power supply |
| • U-blox M8N Glonass/GPS/Beidou | • Power module connector |
| • RC I/O co-processor | • UART, I2C, ADC for extensions |
| • HAT EEPROM | • Size: 55x65mm |
| • RGB LED | • Weight: 23gr |

Table 2.2: Navio2 specifications and features.

[1]https://en.wikipedia.org/wiki/Pulse-position_modulation

## 2.5 Arduino Nano

As mentionned in the previous section, the Navio2 only accepts PPM signal on SBUS, and the receiver outputs PWM signals. Thus, the Arduino is used as a PWM to SBUS converter. The Arduino Nano (Fig. 2.6) is a small programmable microcontroller development board, frequently used in the Do It Yourself (DIY) community, thanks to its low price, and quick and easy code and sensors implementation. An existing project found on Github[2] did exactly what we required : a PWM to SBUS conversion. Thus, it was used at this purpose for this project. It simply takes as input the PWM output channel from the remote control receiver, converts the signals into one PPM/SBUS connected directly onto the Navio2. *Note: Eric Unnervik's work on the nonlinear system identification for the Riderless Motorcycle also includes a speed encoder on the back wheel, which is connected to and processed by the Arduino Nano.*

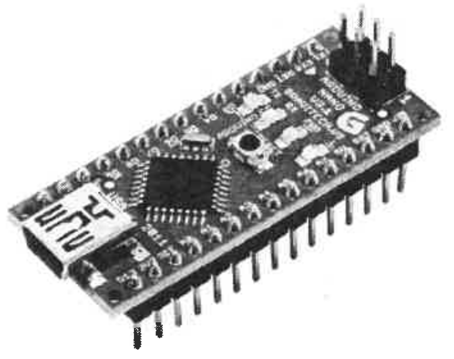


Figure 2.6: Arduino Nano.

## 2.6 Reach RTK

### 2.6.1 What is RTK ?

Real Time Kinematics (RTK) is a differential GPS measurement. It consists of a stationnary base, which coordinates are precisely known. The base (or the station) measures the position it gets from the satellites and, since its position is known, it can compute corrections terms (taking into account error sources such as satellite clock and ephemerides delays, and ionospheric and tropospheric delays) using a process called *ambiguity resolution*[3] that are then send to a mobile receiver, called the *rover.* The precision therefore obtained is in the order of the centimeter. Fig. 2.7 shows a diagram of the RTK operation.

[2]https://github.com/davidbuzz/BuzzsArduinoCode/tree/master/buzz_8pwm_to_ppm328

[3]http://www.novatel.com/an-introduction-to-gnss/chapter-5-resolving-errors/real-time-kinematic-rtk/

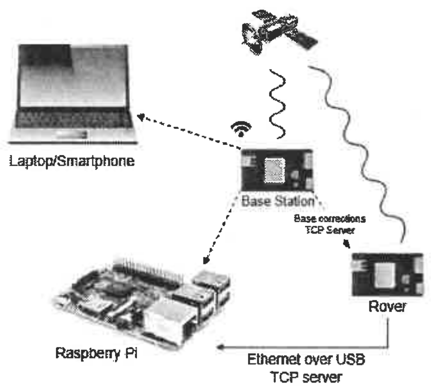
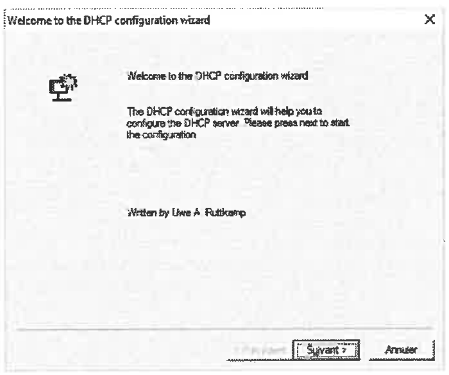


Figure 2.9: Emlid Reach RTK system diagram for the Riderless Motorcycle.

1. Run *dhcpwiz.exe* (Fig. 3.2).
2. Select *Ethernet* as network interface (Fig. 3.3).
3. Enable and set DNS server with address 8.8.8.8 (Fig. 3.4).
4. Set the IP pool to be 192.168.137.1/254 (Fig. 3.5).
5. Click on *Advanced...* to open advanced settings and set the subnet mask to be 255.255.255.0, the DNS server 8.8.8.8 and the gateaway 192.168.137.1 (Fig. 3.6).
6. Click *OK* and *Next->*, and write the INI file (Fig. 3.7).
7. You can now run your DHCP server. Install and start the service, and configure the firewall exception (Fig. 3.8).

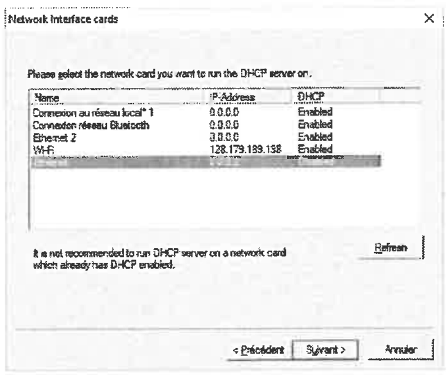


Welcome to the DHCP configuration wizard

The DHCP configuration wizard will help you to configure the DHCP server. Please press next to start the configuration

Written by Uwe A. Ruttkamp

Figure 3.2: DHCP server configuration

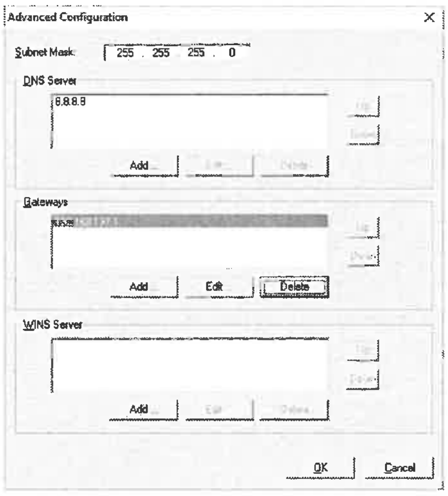


Please select the network card you want to run the DHCP server on.

| Name | IP-Address | DHCP |
|---|---|---|
| Connexion au réseau local* 1 | 0.0.0.0 | Enabled |
| Connexion réseau Bluetooth | 0.0.0.0 | Enabled |
| Ethernet 2 | 0.0.0.0 | Enabled |
| Wi-Fi | 128.179.189.158 | Enabled |
| [illegible] | [illegible] | [illegible] |

It is not recommended to run DHCP server on a network card which already has DHCP enabled.

Figure 3.3: DHCP server configuration: Network interface selection.



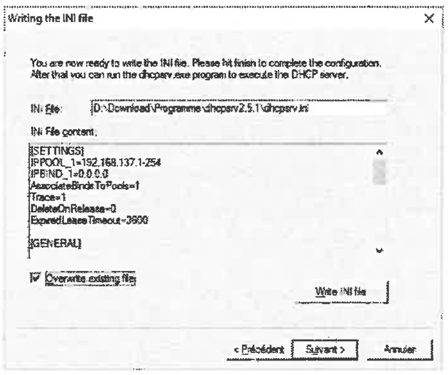
Subnet Mask: 255 . 255 . 255 . 0

DNS Server: 8.8.8.8

Gateways: [illegible]

WINS Server:

Figure 3.6: DHCP server configuration: advanced configuration (subnet mask, DNS server and gateaway).



You are now ready to write the INI file. Please hit finish to complete the configuration. After that you can run the dhcpsrv.exe program to execute the DHCP server.

INI File: D:\Download\Programme\dhcpsrv2.5.1\dhcpsrv.ini

INI File content:

```
[SETTINGS]
IPPOOL_1=192.168.137.1-254
IPBIND_1=0.0.0.0
AssociateBindsToPools=1
Trace=1
DeleteOnRelease=0
ExpiredLeaseTimeout=3600

[GENERAL]
```

[✓] Overwrite existing file

Figure 3.7: DHCP server configuration: INI configuration file writing.

You're then prompted for the user password. Enter *raspberry*:

```
$ ssh pi@192.168.137.2
The authenticity of host '192.168.137.2 (192.168.137.2)' can't be established.
ECDSA key fingerprint is SHA256:g9rwSbfnAOvhLaVdDNKJ3 nPYSM+pEdEy/zWeOW5K2NM.
Are you sure you want to continue connecting (yes/no)? yes
pi@192.168.137.2's password:
```

You should now be connected to your raspi:

```
The programs included with the Debian GNU/Linux system are free
software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
pi@navio:~ $
```

Test the internet connection:

```
pi@navio:~ $ sudo ping -c 5 google.ch
PING google.ch (195.176.255.237) 56(84) bytes of data.
64 bytes from 195.176.255.237: icmp_seq=1 ttl=58 time=9.04 ms
64 bytes from 195.176.255.237: icmp_seq=2 ttl=58 time=8.20 ms
64 bytes from 195.176.255.237: icmp_seq=3 ttl=58 time=8.09 ms
64 bytes from 195.176.255.237: icmp_seq=4 ttl=58 time=8.01 ms
64 bytes from 195.176.255.237: icmp_seq=5 ttl=58 time=8.16 ms

--- google.ch ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4007ms
rtt min/avg/max/mdev = 8.013/8.305/9.048/0.393 ms
```

If you don't get any response, you should have done something wrong in the previous steps.

## 3.3 Setup a WiFi connection

You can also connect your Raspberry Pi to a WiFi network using a WiFi dongle. WiFi networks can be configured by editing */etc/wpa_ supplicant.conf* file. To add your network, simply add the following lines to it:

```
network={
  ssid="yourssid"
  key_mgmt=WPA-PSK
  psk="yourpasskey"
}
```

## 3.4 Expand filesystem

In order to get the full range of SD card memory for the system, a filesystem expand is required. This can be done using *raspi-config* (Fig 3.9) command by choosing **Expand Filesystem** option:

```
$ sudo raspi-config
$ sudo apt-get update
```

Install bootstrap dependencies

```
$ sudo apt-get install python-pip python-setuptools python-yaml
python-distribute python-docutils python-dateutil python-six
$ sudo pip install rosdep rosinstall_generator wstool rosinstall
```

Initializing Rosdep

```
$ sudo rosdep init
$ rosdep update
```

### 3.5.3 Installation

Now, we will download and build ROS Indigo. Create a catkin workspace. In order to build the core packages, you will need a catkin workspace. Create one now:

```
$ mkdir ~/ros_catkin_ws
$ cd ~/ros_catkin_ws
```

Next we will want to fetch the core packages so we can build them. We will use wstool for this.

```
$ rosinstall_generator ros_comm --rosdistro indigo --deps --wet-only
--exclude roslisp --tar > indigo-ros_comm-wet.rosinstall
$ wstool init src indigo-ros_comm-wet.rosinstall
```

Once you have completed downloading the packages and have resolved the dependencies, you are ready to build the catkin packages.
Invoke catkin_make_isolated:

```
$ sudo ./src/catkin/bin/catkin_make_isolated --install
-DCMAKE_BUILD_TYPE=Release --install-space /opt/ros/indigo -j2
```

Now ROS should be installed! Remember to source the new installation:

```
$ source /opt/ros/indigo/setup.bash
```

Or optionally source the setup.bash in the /.bashrc, so that ROS environment variables are automatically added to your bash session every time a new shell is launched:

```
$ sudo nano ~./bashrc
```

And add *source /opt/ros/indigo/setup.bash* at the end of the file.

### 3.5.4 Adding released packages

You may add additional packages to the installed ros workspace that have been released into the ros ecosystem. First, a new rosinstall file must be created including the new packages.

```
$ cd ~/ros_catkin_ws
$ rosinstall_generator ros_comm ros_control joystick_drivers common_msgs
--rosdistro indigo --deps --wet-only --exclude roslisp --tar >
indigo-custom_ros.rosinstall
```

Next, update the workspace with wstool:

- **navio2_imu**: this node is written in C++ and uses the provided Navio2 libraries[9]. It handles IMU readings and publishes them on the *imu_readings* and *mag_readings* topics.
- **navio2_remote**: this node is also written in C++ and also uses the provided Navio2 libraries. It handles the remote readings as well as the PWM signal gereration to drive the motors and publish the remote readings on the *remote_readings* topic.
- **gps_rtk**: also written in C++, this node handles the RTK GPS readings and publish them on the *gps_readings* topic.

The ROS architecture for this project is shown in Fig. 3.11.

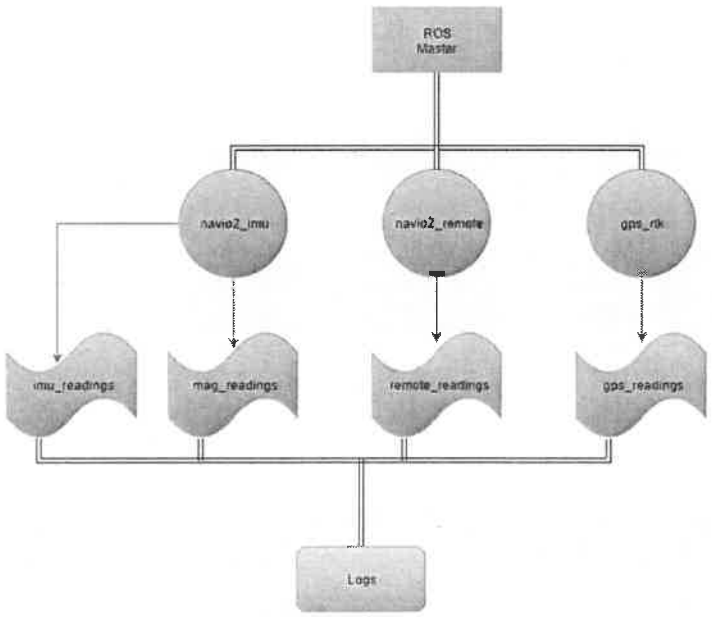


Figure 3.11: Riderless Motorcycle ROS architecture.

### 3.6.3 Run the packages

A bash script (App. A.1) was created in order to launch the roscore and all the packages at once in a tmux environment (Fig 3.12). This script is located in */home/pi/catkin_ws* and called *ros.bash*.

[9]https://github.com/emlid/Navio2.git

Chapter 4

# RTK configuration

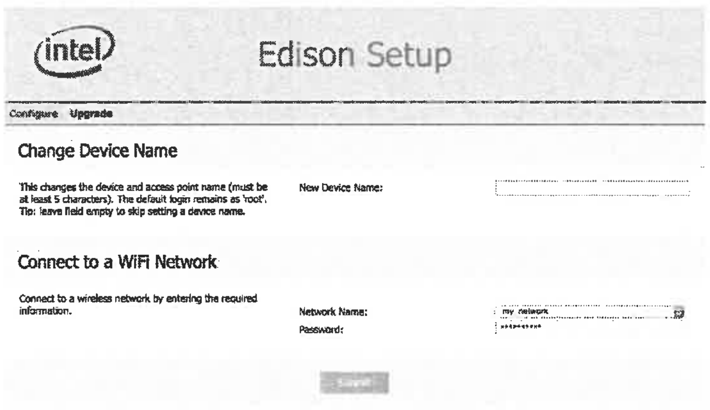
## 4.1 Configuration

### 4.1.1 Connect to Reach

When the module is powered for the first time, it automatically create a WiFi hotspot. The network SSID should be named *reach:PART_ OF_ MAC_ ADDRESS*. Connect to this network with *emlidreach* as password.

### 4.1.2 Setting up WiFi

After connecting to the hotspot, open a web browser and connect to **192.168.42.1:5000**. This will open a confuguration interface (Fig 4.1) where you can set the device's name and the WiFi which you want it to connect to (if required). Note that while changing the device's name, the new hotspot will also change its name to be *DEVICE_ NAME:PART_ OF_ MAC_ ADDRESS*. Unfortunately, the hotspot passphrase cannot be modified and will stay *emlidreach*. By asking the module to connect to a specific WiFi network, it won't create a hotspot anymore.



Change Device Name

This changes the device and access point name (must be at least 5 characters). The default login remains as 'root'. Tip: leave field empty to skip setting a device name.

New Device Name:

Connect to a WiFi Network

Connect to a wireless network by entering the required information.

Network Name: my network

Password: **********

Figure 4.1: Reach WiFi setup.

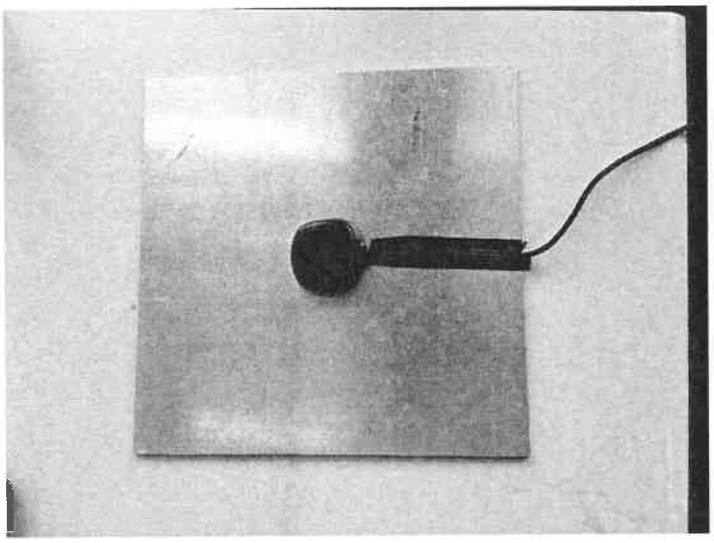


Figure 4.2: Station ground plane made of a 200 mm × 200 mm aluminium plate.

### 4.1.6 Setting up the rover with RTK mode

Connect to your reach you want to use as a station. Go to **Config** tab and choose **Rover** mode. To enable RTK positionong, select **Kinematics** in **Positioning mode. Input source for base corrections** needs to be set as *tcpcli* on the station IP address (by default 192.168.42.1 if you are using the recommended WiFi setup described in Section 4.1.2) on port 9000.

### 4.1.7 Viewing rover results

To view your rover coordinates, open a web browser and connect to your rover's IP. In **Status** tab, you can see a bar chart showing the satellites levels, the current position and the status (Tab 4.1).

| Status | Meaning |
|---|---|
| - | No information. Check the antenna placement. |
| single | No base corrections taken into account. |
| float | Base corrections considered, but integer ambiguity is not resolved. |
| fixed | High level of positioning precision reached. |

Table 4.1: Rover status meanings.

## 4.2 Results

Some tests have been run on field to check the accuracy of the RTK GPS. The results showed that the *fixed* precision was really difficult to reach. Indeed, almost the entire time, the precision was *float*. This can be due to the fact

# Chapter 5

# Position and speed estimation

In order to later do path following Model Predictive Control (MPC), we need to precisely know the position and speed of the motorcycle to make it track a determinate reference. The fusion of the Real Time Kinematics (RTK) GPS, the Inertial Measurement Unit (IMU) and the speed encoder can be used to estimate the state of the motorcycle in presence of measurement and process noise. This chapter proposes, describes and analyzes an approach using an Extended Kalman Filter to do this state estimation.

## 5.1 Asymptotic observer

An observer is a filter that can estimate the state of a system using the input and output measurements. Consider the Linear Time Invariant (LTI) state-space model from Equation (5.1), where $x(k) \in \mathbb{R}^n$ is the state vector, $u(k) \in \mathbb{R}^m$ the input vector, and $y(k) \in \mathbb{R}^l$ the output vector [4]. Matrix $A \in \mathbb{R}^{n \times n}$ is called the state matrix, $B \in \mathbb{R}^{n \times m}$ the input matrix, $C \in \mathbb{R}^{l \times n}$ the output matrix and $D \in \mathbb{R}^{l \times m}$ the feedthrough matrix.

$$x(k+1) = Ax(k) + Bu(k) \tag{5.1a}$$
$$y(k) = Cx(k) + Du(k) \tag{5.1b}$$

According to Luenberger's work [2], the state can be estimated using the measured output $y$ and the the model ouptut $\hat{y}$. The estimation is based on a correction term coming from the difference between $y$ and $\hat{y}$, and a gain matrix $L$. Equation (5.2) represents the resulting system. Equation (5.2a) is called an *asymptotic observer.*

$$\hat{x}(k+1) = A\hat{x}(k) + Bu(k) + L(y(k) - \hat{y}(k)) \tag{5.2a}$$
$$\hat{y}(k) = C\hat{x}(k) + Du(k) \tag{5.2b}$$

The gain matrix $L$ must be chosen in a appropriate way, to minimize the error $\tilde{x}$ between the real state $x$ and the estimated state $\hat{x}$, so its limit at $k \to \infty$ tends to 0, as shown in Equation (5.3).

$\mathbb{R}^l$ the *measurement noise.* Both are zero-mean gaussian noise. Equation (5.6) describes the model including noise without feedthrough term ([4]).

$$x(k+1) = Ax(k) + Bu(k) + w(k) \tag{5.6a}$$
$$y(k) = Cx(k) + v(k) \tag{5.6b}$$

Taking the observer from (5.2a), the estimates $\hat{x}$ becomes:

$$\hat{x}(k) \quad = \quad A\hat{x}(k-1) + Bu(k-1) + K(y(k-1) - C\hat{x}(k-1)) \tag{5.7}$$

However, the presence of noise will not lead $\tilde{x}$ to 0, even if $K$ is chosen such that $(A - KC)$ is asymptocally stable. The idea with the Kalman filter is to make the error as small as possible, to be close to the real state $x(k)$. To do so, we want to make the state-error covariance matrix $P(k|k) = \mathbb{E}[(x(k) - \hat{x}(k|k))(x(k) - \hat{x}(k|k))^T]$, where $\mathbb{E}$ denotes the expectation (Equation (5.8)), the smallest possible, in order to get a minimum-error variance estimate.

$$\mathbb{E}(x) = \frac{1}{N}\sum_{i=1}^{N} x_i, \ \ where\ x \in \mathbb{R}^N \tag{5.8}$$

Let's define the noise covariances to be:

$$\mathbb{E}(w(k)w(k)^T) = Q \tag{5.9a}$$
$$\mathbb{E}(v(k)v(k)^T) = R \tag{5.9b}$$
$$\mathbb{E}(w(k)v(k)^T) = S \tag{5.9c}$$

Then, the state-error covariance matrix $P$ is derived by solving the discrete Ricatti equation:

$$P = APA^T + Q - (S - APC^T)(CPC^T + R)^{-1}(S + APC^T)^T \tag{5.10}$$

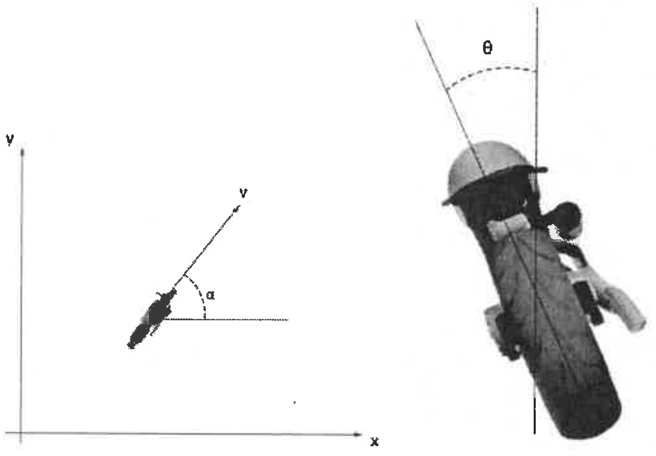
And the Kalman gain $K$ as:

$$K = (S + APC^T)(CPC^T + R)^{-1} \tag{5.11}$$

This is the steady-state Kalman filter. The time-varying Kalman filter is a generalization of the steady-state filter for time-varying systems or LTI systems with nonstationary noise covariance.

The time-varying Kalman filter is given by the following equations [4]:

$$\hat{x}(k|k) = \hat{x}(k|k-1) + K(k)(y(k) - C\hat{x}(k|k-1)) \tag{5.12a}$$
$$K(k) = P(k|k-1)C^T(CP(k|k-1)C^T + R(k))^{-1} \tag{5.12b}$$
$$P(k|k) = (I - K(k)C)P(k|k-1) \tag{5.12c}$$
$$\hat{x}(k+1|k) = A\hat{x}(k|k) + Bu(k) \tag{5.12d}$$
$$P(k+1|k) = AP(k|k)A^T + Q(k) \tag{5.12e}$$



(a) $x$, $y$, $v$ and $\alpha$ states. (b) $\theta$ state.

Figure 5.2: Definition of the motorbike state vector $\mu$

The state evolution relies mainly on the radius of the curve that the motorbike is following (see Fig. 5.3) which can be computed as in Equation (5.15a) where $g$ is the Earth gravitationnal constant. $w(v, R)$ denotes the angular velocity around the curve and is given by Equation (5.15b).

$$R(v,\theta) = \frac{v^2}{gtan(\theta)} \tag{5.15a}$$

$$w(v,R) = \frac{v}{R} \tag{5.15b}$$

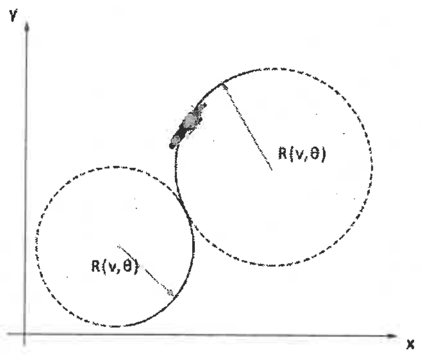
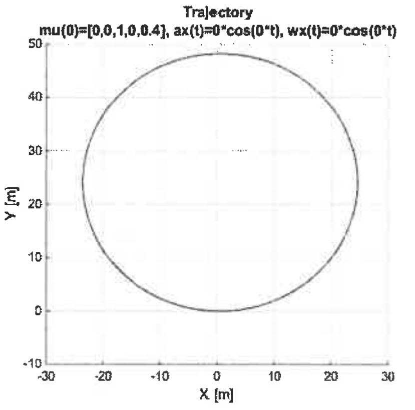


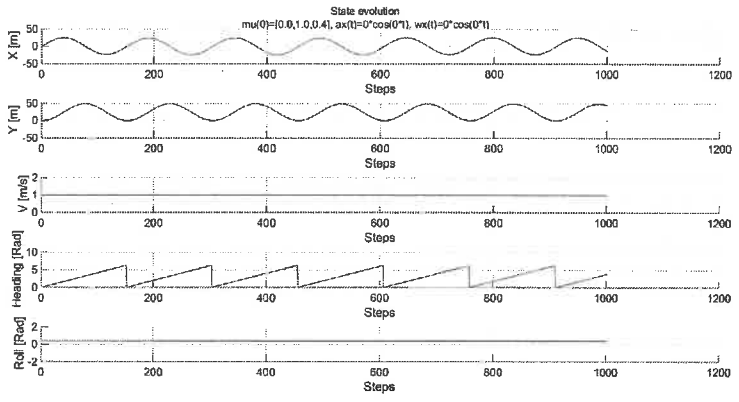
Figure 5.3: Radius $R(v, \theta)$ of the curve that the motorbike is following.

Therefore, the nonlinear model of the motorbike can be described by $g(\mu(k), u(k))$ with Equation (5.16). And its Jacobian by $G(\mu(k), u(k))$ with Equation (5.17).

The model was implemented in a MATLAB script to validate it. Fig. 5.4 and Fig. 5.5 show the state evolution as well as the motorcycle trajectory for various initial states and control inputs. Simulations show that the motorcycle behave as expected with this model. Indeed, whenever the roll angle is positive, the motorcycle is turning left and turns right when the roll is negative. Moreover, the more the motorbike is tilting, the sharper is the turn, and the faster it goes, the smoother it turns. Therefore, it will be kept for the next steps of the state estimation.



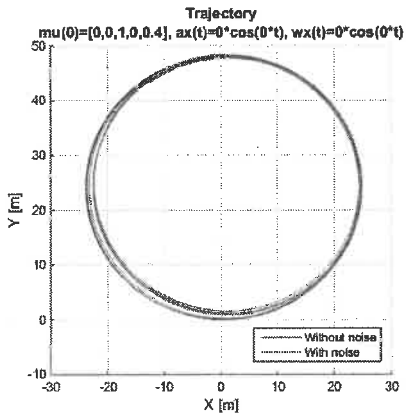
(a) Resulting trajectory.



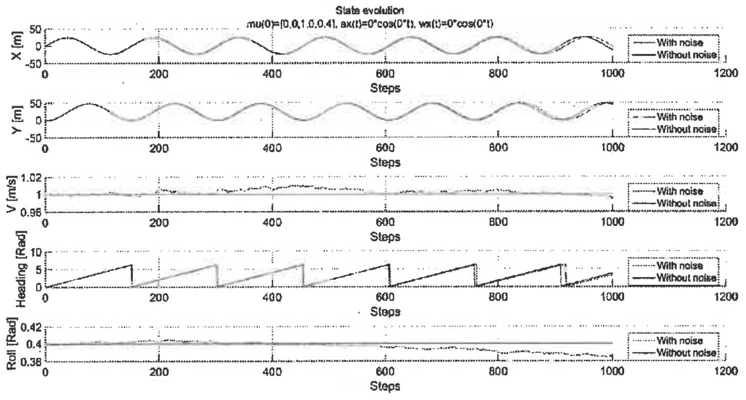
(b) State evolution.

Figure 5.4: Simulation of the motorcycle's model described in 5.3.1, with an initial state $\mu(0) = [0, 0, 1, 0, 0.4]^T$ and no control inputs.

and Fig. 5.7 show the state evolution as well as the motorcycle trajectory for various initial states and control inputs including process noise. Simulations show that the noisy state diverges from the real state. Therefore, it needs to be corrected with measurements, thus the extended Kalman filter presented in 5.3.3 will be used.



(a) Resulting trajectories.



(b) State evolution.

Figure 5.6: Simulation of the motorcycle's model described in 5.3.1, with an initial state $\mu(0) = [0, 0, 1, 0, 0.4]^T$ and no control inputs. Including process noise with $var(a_x) = 0.05$ and $var(\omega_x) = 0.02$.

can be made to estimate the current state to be as close as possible to the real state.

As mentionned in 5.3.2, only the noise on the control input is considered, and propagates among all the states. The process covariance matrix $Q$ is then defined by Equation (5.21).

$$Q = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & var(a_x)^2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & var(\omega_x)^2 \end{bmatrix} \tag{5.21}$$

Since the sampling frequency from the GPS and the speed encoder are both different from the IMU sampling frequency, a new measurement of either position or speed is not available at each step, thus the measurement updates can't be done the same way at each step. It is then required to treat each case separately:

- Only the GPS measurements are available at this step: only the position of the motorbike is updated. The output vector becomes (5.22), the output matrix (5.24) and the measurement covariance matrix (5.23).
- Only the speed encoder measurements are available at this step: only the speed of the motorbike is updated. The output vector becomes (5.25), the output matrix (5.27) and the measurement covariance matrix (5.26).
- Both GPS and speed encoder measurements are available at this step: both position and speed are updated. This is the most accurate measurement update available for this filter. The output vector becomes (5.28), the output matrix (5.30) and the measurement covariance matrix (5.29).
- No measurements are available at this step: no measurement update is done. The estimated state is only propagated and the state-error covariance matrix updated. This case makes the estimate diverge from the real state, but as soon as a new measurement comes, it will be pulled towards the real state.

### Only the GPS measurements are available at this step

$$z = \begin{bmatrix} x \\ y \end{bmatrix} \tag{5.22}$$

$$R_{GPS} = \begin{bmatrix} var(gps)^2 & 0 \\ 0 & var(gps)^2 \end{bmatrix} \tag{5.23}$$

$$H_{GPS} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{bmatrix} \tag{5.24}$$

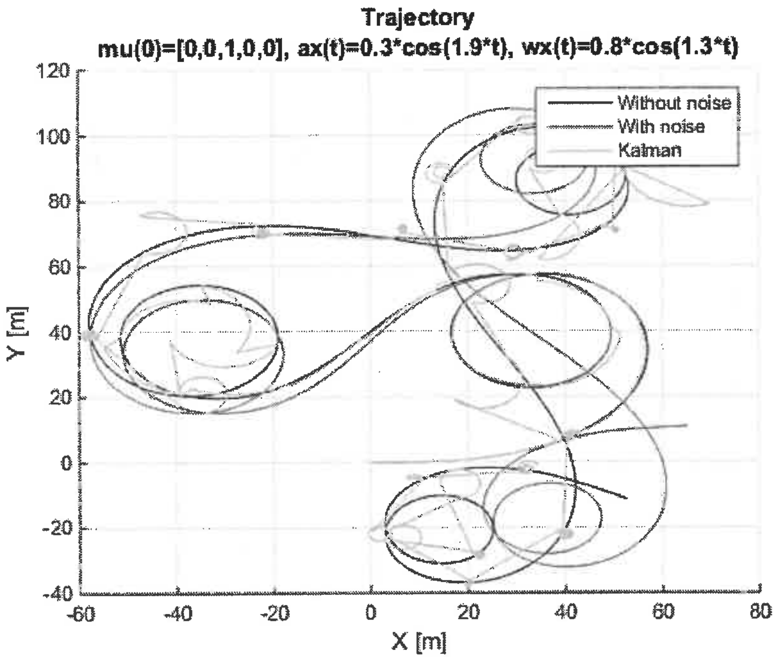
### Only the speed encoder measurements are available at this step

$$z = \begin{bmatrix} v \end{bmatrix} \tag{5.25}$$

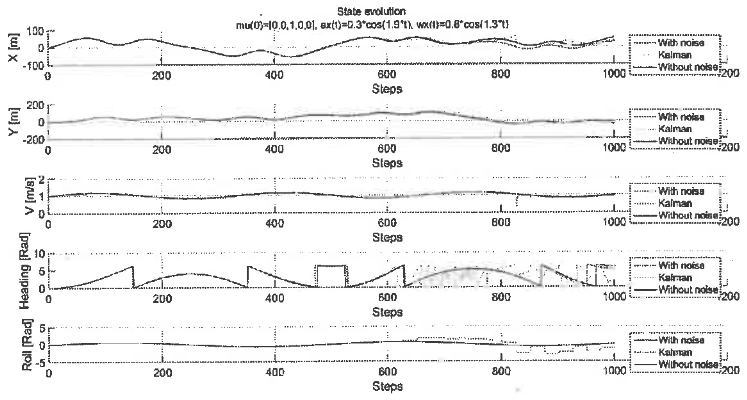
$$R_{encoder} = var(encoder)^2 \tag{5.26}$$

$$H_{encoder} = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 \end{bmatrix} \tag{5.27}$$

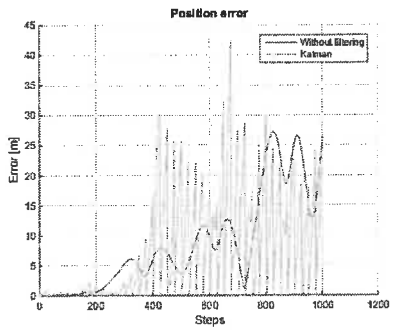
Results then show that it is prefered to have a low noise on the IMU, particulary on the roll rate $\omega_x$ to make the Kalman estimation effective. The speed encoder measurement rate can be increased to yield better results, but this is not sufficient to cancel error position coming from higher roll rate noise. A solution would be to have a measurement of the heading of the motorbike, but due to lack of time, this is left for future work.



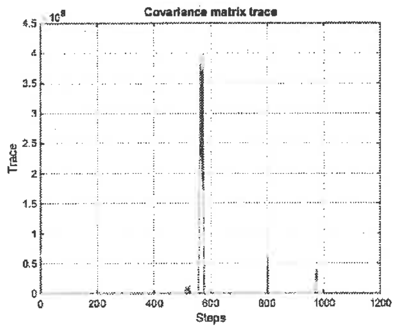
(a) Resulting trajectories.



(b) State evolution.

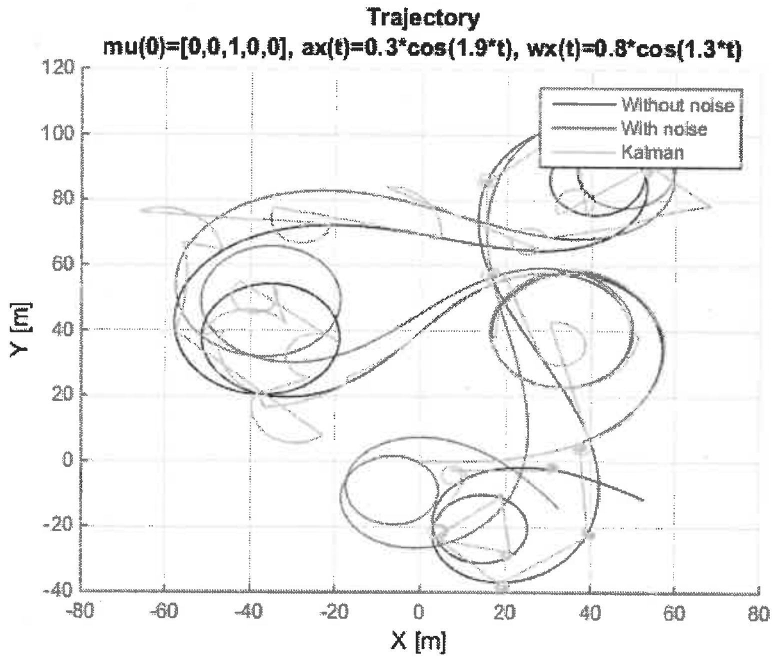


(c) Position error.

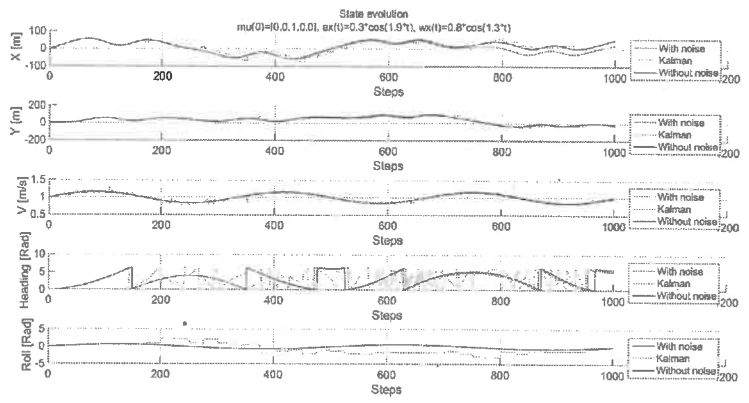


(d) Covariance matrix trace evolution.

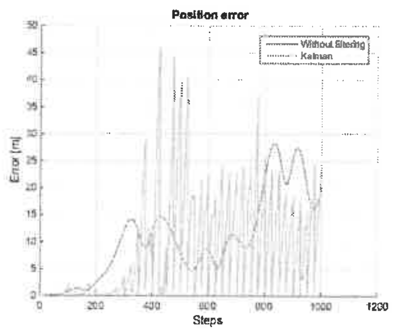
Figure 5.9: Simulation of the motorcycle's model described in 5.3.1, with an initial state $\mu(0) = [0, 0, 1, 0, 0]^T$, and sinusoidal control inputs defined as $a_x(t) = 0.3cos(1.9t)$ and $\omega_x(t) = 0.8cos(1.3t)$. Including process noise with $var(a_x) = 0.05$ and $var(\omega_x) = 0.03$, and measurement noise with $var(gps) = 0.1$ and $var(encoder) = 0.05$. Speed encoder frequency is 50 Hz. Implementation of the extended Kalman filter presented in 5.3.3.



(a) Resulting trajectories.



(b) State evolution.



(c) Position error.

(d) Covariance matrix trace evolution.

Figure 5.11: Simulation of the motorcycle's model described in 5.3.1, with an initial state $\mu(0) = [0, 0, 1, 0, 0]^T$, and sinusoidal control inputs defined as $a_x(t) = 0.3cos(1.9t)$ and $\omega_x(t) = 0.8cos(1.3t)$. Including process noise with $var(a_x) = 0.05$ and $var(\omega_x) = 0.03$, and measurement noise with $var(gps) = 0.1$ and $var(encoder) = 0.05$. Speed encoder frequency is 100 Hz. Implementation of the extended Kalman filter presented in 5.3.3.

# Appendix A

# Appendices

## A.1 ROS bash script

```
#!/bin/bash
SESSION=$USER

if [ "$#" -ne 1 ]
then
  tmux -2 new-session -d -s $SESSION
  tmux new-window -t $SESSION:1 -n 'ROS'
  tmux split-window -v
  tmux select-pane -t 0
  tmux send-keys "source /home/pi/catkin_ws/devel/setup.bash" C-m
  tmux send-keys "roscore" C-m
  tmux select-pane -t 1
  tmux send-keys "sleep 5" C-m
  tmux send-keys "sudo -i" C-m
  tmux send-keys "source /home/pi/catkin_ws/devel/setup.bash" C-m
  tmux send-keys "rosrun navio2_remote remote_pub_sub" C-m
elif [ "$1" == "-log" ]
then
  tmux -2 new-session -d -s $SESSION
  tmux new-window -t $SESSION:1 -n 'ROS'
  tmux split-window -h
  tmux select-pane -t 0
  tmux send-keys "source /home/pi/catkin_ws/devel/setup.bash" C-m
  tmux send-keys "roscore" C-m
  tmux select-pane -t 1
  tmux send-keys "sleep 5" C-m
  tmux send-keys "source /home/pi/catkin_ws/devel/setup.bash" C-m
  tmux send-keys "rosrun navio2_imu imu_pub" C-m
  tmux split-window -v
  tmux send-keys "sleep 5" C-m
  tmux send-keys "rosrun navio2_gps gps_pub" C-m
  tmux select-pane -t 0
```

```
Q_encoder = [0,0,1,0,0];         %Measurement matrix for encoder

R = [0,0,0,0,0;...                        %Process covariance matrix
0,0,0,0,0;...
0,0,var_ax^2,0,0;...
0,0,0,0,0;
0,0,0,0,var_wx^2];

R_gps = [var_gps^2,0;...         %Measurement covariance matrix for GPS
0,var_gps^2];

R_encoder = [var_encoder^2]; %Measurement covariance matrix for encoder

%System
% x : x-position
% y : y-position
% v : speed
% alpha : heading
% theta : roll
%
% ax : acceleration in x (motorcycle coordinates)
% wx : roll rate
syms x y v alpha theta ax wx dt

%System for theta != 0
Radius(v,theta) = (v^2)/(9.81*tan(theta));        %Radius of curvature while turning
w(v) = v/Radius(v,theta);        %Angular velocity around the curve

g1(x,y,v,alpha,theta,ax,wx,dt) =  x + v*cos(alpha) + ((ax*cos(alpha))^2)*(dt/2);
g2(x,y,v,alpha,theta,ax,wx,dt) =  y + v*sin(alpha) + ((ax*sin(alpha))^2)*(dt/2);
g3(x,y,v,alpha,theta,ax,wx,dt) = v + ax*dt;
g4(x,y,v,alpha,theta,ax,wx,dt) = alpha + w(v)*dt;
g5(x,y,v,alpha,theta,ax,wx,dt) = theta + wx*dt;

G = [g1;g2;g3;g4;g5];
J = jacobian(G,[x,y,v,alpha,theta]);        %Linearization

%System for theta == 0 or v == 0
gs1(x,y,v,alpha,theta,ax,wx,dt) = x + v*cos(alpha) + ((ax*cos(alpha))^2)*(dt/2);
gs2(x,y,v,alpha,theta,ax,wx,dt) = y + v*sin(alpha) + ((ax*sin(alpha))^2)*(dt/2);
gs3(x,y,v,alpha,theta,ax,wx,dt) = v + ax*dt;
gs4(x,y,v,alpha,theta,ax,wx,dt) = alpha;
gs5(x,y,v,alpha,theta,ax,wx,dt) = theta + wx*dt;

Gs = [gs1;gs2;gs3;gs4;gs5];
Js = jacobian(Gs,[x,y,v,alpha,theta]);        %Linearization

%Simulation
mu = [0;0;1;0;0];                        %Real state
```

```
end


% 0 <= alpha < 2pi
% -pi < theta <= pi
mu_noise(4) = mod(mu_noise(4),2*pi);

if mu_noise(5) > pi
        mu_noise(5) = -((2*pi)-mu_noise(5));
elseif mu_noise(5) < -pi
        mu_noise(5) = 2*pi + mu_noise(5);
end

mus_noise = [mus_noise mu_noise];

%Kalman state propagation
x=mus_kalman(1,i);y=mus_kalman(2,i);v=mus_kalman(3,i);
alpha=mus_kalman(4,i);theta=mus_kalman(5,i);

if theta == 0 || v == 0
        mubar = eval(Gs);
        sigmabar = eval(Js)*sigma*eval(Js)' + R;
else
        mubar = eval(G);
        sigmabar = eval(J)*sigma*eval(J)' + R;
end


% 0 <= alpha < 2pi
% -pi < theta <= pi
mubar(4) = mod(mubar(4),2*pi);

if mubar(5) > pi
        mubar(5) = -((2*pi)-mubar(5));
elseif mubar(5) < -pi
        mubar(5) = 2*pi + mubar(5);
end

%KALMAN
%Both GPS and encoder data are measured
if((mod(i-1,freq_imu/freq_gps) == 0) && (mod(i-1,freq_imu/freq_encoder)) == 0)
        Q = [Q_gps; Q_encoder];
        R = [R_gps,[0;0];[0,0,R_encoder]];

        K = sigmabar*Q'*((Q*sigmabar*Q' + R)^(-1));

        z = [mu(1) + normrnd(0,var_gps);...
        mu(2) + normrnd(0,var_gps);...
        mu(3) + normrnd(0,var_encoder)];
```

```
figtitle = sprintf(['State evolution\nmu(0)'...
'=[%0.2g,%0.2g,%0.2g,%0.2g,%0.2g], ax(t)=%0.2g*cos(%0.2g*t),'...
' wx(t)=%0.2g*cos(%0.2g*t)'],...
mus(1,1), mus(2,1), mus(3,1), mus(4,1),...
mus(5,1), ax_A, ax_w, wx_A, wx_w);
figure(1)
suptitle(figtitle)
subplot(5,1,1)
plot(mus_noise(1,:),'r-');grid on; hold on;
plot(mus_kalman(1,:),'g-');grid on; hold on;
plot(mus(1,:),'k-');grid on; hold on;
legend('With noise','Kalman', 'Without noise');
xlabel('Steps');
ylabel('X [m]');

subplot(5,1,2)
plot(mus_noise(2,:),'r-');grid on; hold on;
plot(mus_kalman(2,:),'g-');grid on; hold on;
plot(mus(2,:),'k-');grid on; hold on;
xlabel('Steps');
ylabel('Y [m]');
legend('With noise','Kalman', 'Without noise')

subplot(5,1,3)
plot(mus_noise(3,:),'r-');grid on; hold on;
plot(mus_kalman(3,:),'g-');grid on; hold on;
plot(mus(3,:),'k-');grid on; hold on;
legend('With noise','Kalman', 'Without noise')
xlabel('Steps');
ylabel('V [m/s]')

subplot(5,1,4)
plot(mus_noise(4,:),'r-');grid on; hold on;
plot(mus_kalman(4,:),'g-');grid on; hold on;
plot(mus(4,:),'k-');grid on; hold on;
legend('With noise','Kalman', 'Without noise')
xlabel('Steps');
ylabel('Qeading [Rad]');

subplot(5,1,5)
plot(mus_noise(5,:),'r-');grid on; hold on;
plot(mus_kalman(5,:),'g-');grid on; hold on;
plot(mus(5,:),'k-');grid on; hold on;
legend('With noise','Kalman', 'Without noise')
xlabel('Steps');
ylabel('Roll [Rad]');

figure(2);hold on;grid on;
figtitle = sprintf(['Trajectory\nmu(0)=[%0.2g,%0.2g,%0.2g,%0.2g,%0.2g],'...
' ax(t)=%0.2g*cos(%0.2g*t), wx(t)=%0.2g*cos(%0.2g*t)'],...
```

# Bibliography

[1] J. Leimer. Identification and control for riderless motorcycle. Technical report, EPFL, 2016.

[2] D. G. Luenberger. Observing the state of a linear system. *IEEE Transactions on Military Electronics*, 8(2):74–80, April 1964.

[3] S. Thrun, W. Burgard, and D. Fox. *Probabilistic Robotics (Intelligent Robotics and Autonomous Agents).* The MIT Press, 2005.

[4] M. Verhaegen and V. Verdult. *Filtering and System Identification, A Least Squares Approach.* Cambridge University Press, 2007.