Learning Parameterized Maneuvers from Multiple Demonstrations

Martin Møller Sørensen, Master Thesis

University of California at Berkeley
Department of Electrical Engineering
and Computer Sciences

Aalborg University
Section for Automation and Control
Department of Electronic Systems

8^{th} June, 2009

University of California at Berkeley
Department of Electrical Engineering and Computer Sciences
253 Cory Hall, Berkeley, CA 94720
Telephone (+1) 510 642-3214
http://eecs.berkeley.edu

Aalborg University
Intelligent Autonomous Systems
Department of Electronic Systems
Fredrik Bajers Vej 7, C3
Telephone (+45) 99 40 87 02
http://es.aau.dk

Title: Learning Parameterized Maneuvers from Multiple Demonstrations

Project period:
9^{th} semester, fall 2008

Author:
Martin Møller Sørensen

Supervisors:
Pieter Abbeel
Jakob Stoustrup
Morten Bisgaard

Copies: 4

Pages: 96

Appendices: 4

Printed: June 8, 2009

Abstract:

Hand-specifying trajectories for control tasks can be very time-consuming, and often near impossible as good target trajectories for control should satisfy the system dynamics.
This project has concerned with the problem of planning arbitrary trajectories by stitching together small pieces of different parameterized maneuvers. The maneuvers are found by using interpolation-based algorithms and probabilistic model-based algorithms.
An algorithm is presented which uses a few waypoints with partial state information and a large corpus of random demonstrations. It then plans a larger trajectory by looking up good demonstrations in the data set based on the waypoints, and then interpolating between the demonstrations picked.
This makes is possible to automatically generate target trajectories for control by learning parameterized maneuvers from multiple demonstrations of the maneuvers.
As test platform a Drift-R Sedan 4WD 1/10 RC car has been used. A Differential Dynamic Programming controller have been used for successfully controlling the car around the planned trajectory.

Preface

This report is my master thesis and is written in the fall 2008 and spring 2009 while studying abroad at the University of California at Berkeley in the department of Electrical Engineering and Computer Sciences (EECS). I would like to say a special thanks to assistant professor Pieter Abbeel who have supervised me in my project, and have provided me with much help for this project.

Also a big thanks to Professor Edward A. Lee, EECS department UC Berkeley, and Professor Jakob Stoustrup, Aalborg University, who made it possible for me to attend UC Berkeley.

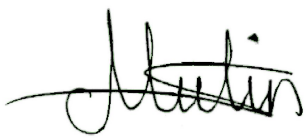
Pieter Abbeel was newly hired at UC Berkeley one month before this project started. Therefore, all hardware and most of the software have been setup and implemented from scratch. This is also why there throughout the report are various implementations sections and appendices.

Citations throughout the report are denoted [Surname of author, Publication year].

An enclosed CD is available, containing the following:

- The report in PDF format
  `./report/parameterized_maneuvers_using_reinforcement_learning_2009.pdf`
- The source code
  `./code/`

University of California at Berkeley the 7^{th} of June 2009



Martin Møller Sørensen

Contents

Chapter 1

Introduction

Trajectory following is a fundamental building block for many robotics tasks. By reducing the control problem to trajectory following, one can often suffer less from the curse of dimensionality as it becomes sufficient to consider a relatively small part of the state space during control policy design. Unfortunately, specifying the desired trajectory and building an appropriate model for the robot dynamics along that trajectory are often non-trivial tasks which are tightly coupled. Indeed, for the control design to benefit from being reduced to a trajectory following task, it typically requires that the target trajectory is physically feasible. Specifying a physically feasible target trajectory can be highly challenging. For example, what would be the correct state sequence (position, heading and its derivatives) for a car performing an aggressive sliding turn?

Therefore, the apprenticeship learning setting is often used, where an expert is used to provide expert demonstrations, where is natural to request a demonstration of the desired trajectory as specifications of the target trajectory.

However, rarely will an expert be able to demonstrate exactly the desired trajectory to execute autonomously. Repeated expert demonstrations together often do capture a desired maneuver, as different demonstrations deviate from the intent in different ways. Indeed, in [Coates et al., 2008], the authors described a generative probabilistic model that enabled them to extract an expert helicopter pilot's intended aerobatic trajectory from multiple suboptimal demonstrations. They also show how the multiple demonstrations can be leveraged to obtain a high accuracy dynamics model, which is specifically tuned to the particular maneuver in consideration.

Building a maneuver library directly based upon [Coates et al., 2008] would require collecting a set of demonstrations for each maneuver. For example, one might have a set of demonstrations of 90 degree left turns, a set of demonstrations of 90 degree right turns, a set of demonstrations of 80 degree left turn, a set of demonstrations of 80 degree right turn, etc.

This report presents interpolation-based algorithms and probabilistic model-based algorithms (which build upon [Coates et al., 2008]) which, rather than learning a discrete set of maneuvers, make more efficient use of expert demonstrations by learning *parameterized maneuvers*, which continuously index into a certain maneuver (for example a right turn) based upon a set of demonstrations spanning the range of executions of that maneuver, see Figure 1.1.

While the parameterization can be along any quantitative property of the maneuver, in practical settings it is of particular interest to consider parameterizations by start and end state of the maneuver, as this enables sequencing several maneuvers. In Chapter 7 on page 63 an algorithm

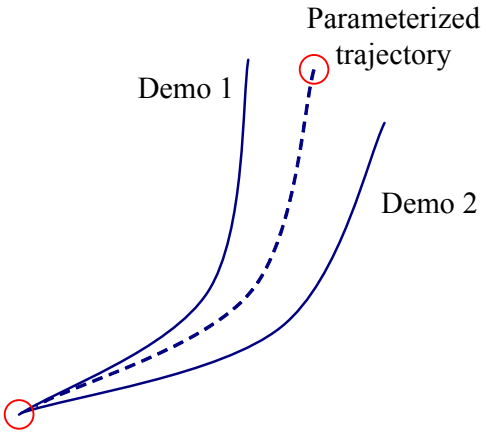


Figure 1.1: Illustration of planning a parameterized maneuver from two demonstrations. The parameterization of the planned trajectory is done based on the start and end state indicated by the circles

is describe for sequencing the learned parameterized trajectories.

The algorithm is applied to learn trajectories for high-precision, aggressive driving of a remotely controlled car. The experimental results show that our algorithms reliably generate good parameterized maneuvers from a relatively small number of training examples: (i) The generated trajectories closely match hold-out trajectories. (ii) Our approach allows the car to reliably perform sequences of learned parameterized maneuvers.

1.1 Related Work

The work by [Coates et al., 2008] is the most closely related. They consider the setting of learning a single intended trajectory and a high-precision dynamics model along that trajectory from several demonstrations. Our probabilistic approach extracts such a single intended trajectory as a side product. The approach presented here also leverage their observation that, for a specific maneuver, it is possible to obtain a very high fidelity dynamics model by combining a crude low-order dynamics model with corrections specific to that trajectory. Our approach extends this towards learning parameterized biases, corresponding to the parameterized trajectories. A minor difference is the handling of time warping: [Coates et al., 2008] use dynamic time warping, a discrete time approach for aligning trajectories. In our setting, the parameterized trajectories are considered of shorter duration (which can be sequenced together to build longer trajectories). Over the shorter durations, uniform time warping (uniformly shrinking or stretching) has been sufficient.

The computer graphics literature has a long history of algorithms that leverage motion capture data in various ways to generate realistic looking animations. While their approaches have enabled significant simplification of the animation generation process, their final objective is not high precision control. (See, e.g., [Lee et al., 2002, Arikan and Forsyth, 2002, Fang and Pollard, 2003, Rose et al., 1996, Unuma et al., 1995, Wiley and Hahn, 1997].)

[Atkeson and Schaal, 1997] use multiple demonstrations to learn a model for a robot arm, and then find an optimal controller in their simulator, initializing their optimal control algorithm with one of the demonstrations.

The work of [Calinon et al., 2007] considered learning trajectories and constraints from demonstrations for robotic tasks. They do not consider the system's dynamics or provide a clear mechanism for the inclusion of prior knowledge.

Among others, [An et al., 1988] and, more recently, [Abbeel et al., 2006a, Coates et al., 2008] have exploited the idea of trajectory-indexed model learning for control.

Our work also has some similarities with recent work on inverse reinforcement learning, which extracts a reward function (rather than a trajectory) from the expert demonstrations. See, e.g., [Ng and Russell, 2000, Abbeel and Ng, 2004, Ratliff et al., 2006, Neu and Szepesvari, 2007] [Ramachandran and Amir, 2007, Syed and Schapire, 2008].

Chapter 2

System Description

This chapter will outline the development platform used in the project. This includes a overview of the system setup, the specifications of the Drift-R Sedan RTR car, the Phasespace motion tracking system and the take-over box for sending and receiving control signals. For each of the sections a short description will be made about the appertaining software which all has been developed as a part of this project. In Chapter 6 on page 59 all the software will be summarized to describe how the different programs work together.

2.1 System Overview

An overview of the overall system setup is illustrated in Figure 2.1.

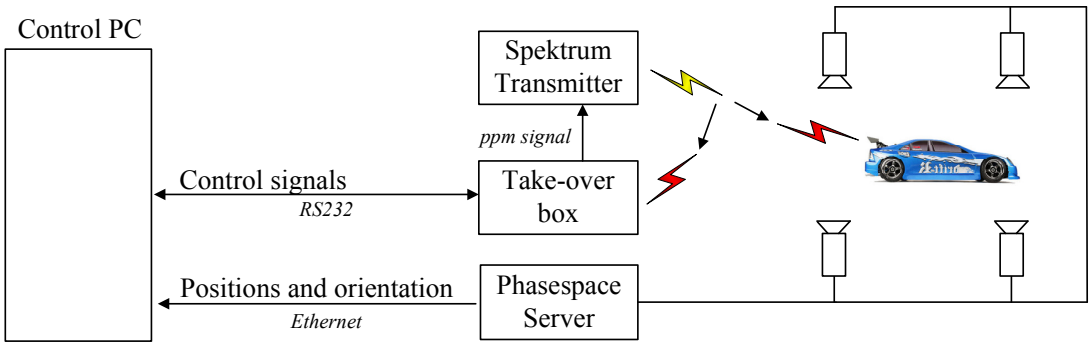


Figure 2.1: An overview of the system setup which include the platform, Phasespace motion capture system and the takeover box for sending and receiving radio signals. The arrows indicates the direction of communication. The yellow lightning indicates wireless transmission, and the red lightnings indicates wireless reception

The car is controlled within the field of the motion tracking system enabling the Phasespace server to find the position and orientation of the system and sends it back to the control PC over a Ethernet connection. A DX6i Spektrum radio transmitter shown in Figure 2.2, is used to transmit signals to the system. For control and modeling purposes the control signals are also received by the take-over box, which sends them to the control PC over a RS232 line. As illustrated the take-over box also makes it possible to send out control signals from the control computer to the system through the Spektrum transmitter, hereby controlling the system autonomously.

Figure 2.2: Spektrum DXi6 transmitter used to transmit control signals [Spektrum, 2008]

2.2 Drift-R Sedan 4WD Car

The Drift-R Sedan RTR is a 1/10 small scale 4WD radio controlled car which is build specific for drifting, and in Figure 2.3 the various main parts is highlighted.

Figure 2.3: System overview of the Drift-R car without shield

Unless otherwise noted all the parts listed below, including the car itself, are from [Losi, 2008]

1. **Phasespace LEDs:** Active tracking LED's [Phasespace, 2008]
2. **LED Controller:** Controller which communicates with the Phasespace server and activate the LED's [Phasespace, 2008]
3. **Battery:** Venom 7.2V 6 cell 5000mAh NiMH battery [Venom, 2008]
4. **Servo:** Z-590 high torque steering servo for controlling the angle of the front wheels
5. **Receiver:** Spektrum AR6200 Pulse Position Modulation (PPM) receiver used to receive the signal from the radio transmitter [Spektrum, 2008]
6. **Main Engine:** High performance DC-motor for the propulsion of the car
7. **Governor:** Novak-engineered Losi ESC speed controller for the main engine

The position and heading of the Drift-R car is tracked by the using the Phasespace motion tracking system, which will be described in the following section.

2.3 Phasespace Motion Tracking System

To track the car the motion tracking system Phasespace is used. Opposite to e.g. the Vicon motion capture system the Phasespace system works by using active LED's which are driven by a wireless controller connected via a basestation to the Phasespace server as illustrated in Figure 2.4. The LED's are then tracked by the Phasespace cameras which allows the Phasespace server to triangulate the position of the LED markers. The system can be used to either return the position of each marker or define a rigid body using three or more markers and output the position and orientation of the specified rigid body, both with rates up to 480 Hz. [Phasespace, 2008]

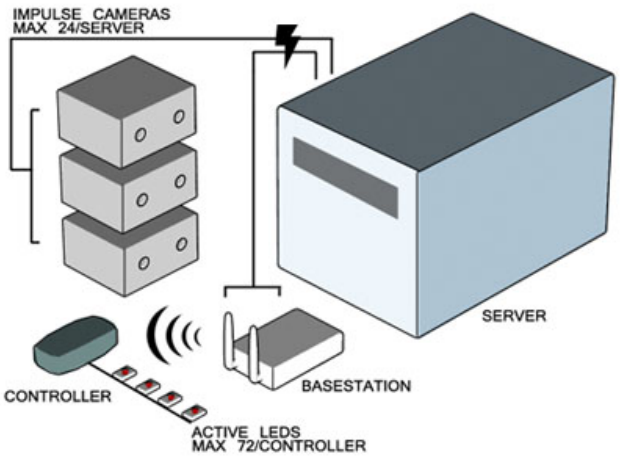


Figure 2.4: Overview of the Phasespace motion tracking system [Phasespace, 2008]

To ease the use of the Phasespace system a C++ class has been developed as part of this project along with the nine public functions listed in Table 2.1

Function	Description
`PhaseSpace_api`	Constructor of the class which sets up the connection to the Phasespace server either for tracking a specified rigid body or individual markers.
`~PhaseSpace_api`	De-constructor of the class which closes down the connection to the Phasespace server and free the allocated memory
`GetSetOfMarkersNonBlocking`	Gets a specified set of markers as a non-blocking call
`GetSetOfMarkersBlocking`	Gets a specified set of markers as a blocking call
`GetAllMarkersNonBlocking`	Gets all 32 markers as a non-blocking call, even though not all of them are active
`GetAllMarkersBlocking`	Gets all 32 markers as a blocking call, even though not all of them are active
`GetMarkerINonBlocking`	Gets a single specified marker as a non-blocking call
`GetMarkerIBlocking`	Gets a single specified marker as a blocking call
`GetAllMarkersMeanBlocking`	Gets the mean value of all markers over a specified number of samples
`GetRigidBodyNonBlocking`	Gets the position and orientation of a specified rigid body as a non-blocking
`GetRigidBodyBlocking`	Gets the position and orientation of a specified rigid body as a blocking call

Table 2.1: The C++ class PhaseSpace_api and the nine public functions associated with it

When the constructor of the class is called, unless otherwise specified in a input, it scales and rotates the coordinates to match a specified inertial frame in the lab. This extrinsic calibration data consist of a scaling factor, a 3D offset and a rotation quaternion. To ease the usage and to improve the accuracy, the extrinsic calibration pose is automatically found by using a homemade calibration tool formed as a triangle with three LED's placed in known positions. A program, `extrinsic_calibration`, has then been programmed which finds the mean position of the three LED's and then calculates the extrinsic calibration data which is saved in a file the constructor loads in when called. A more through description of the extrinsic calibration can be found in appendix A.

2.4 Take-over of Radio Control Signals

To send and receive Pulse Position Modulated (PPM) radio signals to and from the application a take-over box has been build. The control signals are received for modeling and logging purposes. The take-over box consists of two ATmega128 microcontrollers [Atmel, 2008] and a AR6200 radio receiver similar to the one on the car, see Figure 2.6. As illustrated in Figure 2.5 the control PC communicates with both microcontrollers through a RS232 connection.

The data consists of a channel number and a value for that channel which is send over the RS232 connection, in both directions, as two consecutive bytes. The format of the two bytes is illustrated in Figure 2.7. The two microcontrollers then takes care of either converting the two bytes to a PPM signal or to convert the received PPM signal to two bytes.

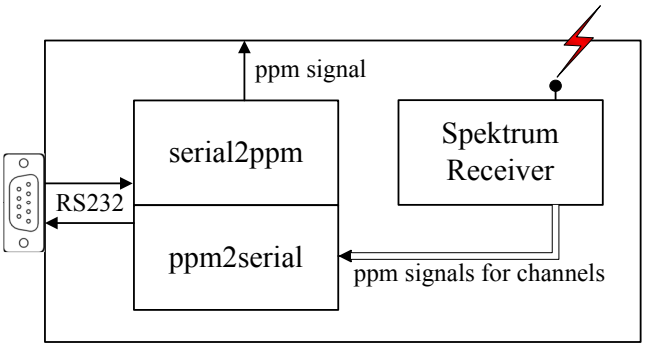


Figure 2.5: Illustration of the take-over box used by the control PC to send and receive signals to and from the transmitter

Figure 2.6: Spektrum AR6200 receiver used to receive the transmitted signals

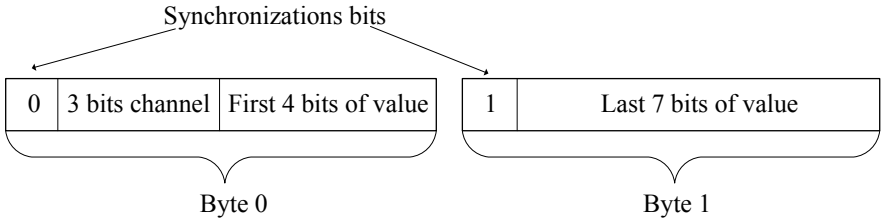


Figure 2.7: Illustration of communication protocol for RS232 connection between the control PC and the take-over box

To ease the use of the takeover system a C++ class has been made with the functions listed in Table 2.2

Function	Description
`ControlsRxTx`	Constructor of the class which sets up the RS232 connection, as inputs it takes in the min, max and default value for each channel used
`~ControlsRxTx`	De-constructor of the class which sends out the default values before it closes down the connection
`RxControls`	Receives control signals from the transmitter
`TxControls`	Sends control signals to the transmitter
`TxControlsWithAck`	Sends control signals to the transmitter and waits until the send signal has been received which makes it a blocking call

Table 2.2: The C++ class ControlsRxTx and the three public functions associated with it

2.4.1 Mapping of I/O Signals

When doing control it is important to make sure that the mapping of the I/O control signals are as precise as possible to minimize the feedback due to a mismatch in the signals. I.e. if the system works well in open loop the feedback is minimized. To simplify the modeling a mapping is done so that the signals send out are as close to those received. Therefore a test has been made where all possible control values in the range of 500-1500 [μs] are send out through the takeover box and the Spektrum transmitter and received again by the take-over box. Both the send and the corresponding received values are then logged in a file which is plotted in Figure 2.8.

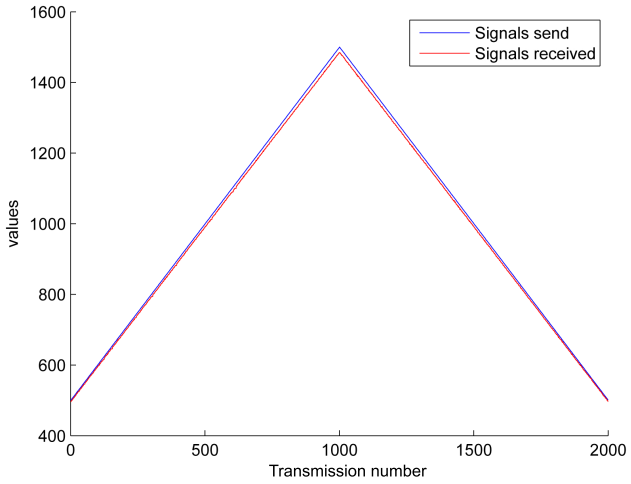


Figure 2.8: The values send out and the corresponding values received. Due to the mismatch in the signals the two curves do not coincident well enough especially at the higher values

To match the I/O control signals the function in Equation (2.1) has been found using linear regression on the collected data.

$$S_2 = 1.0101069 \cdot S_1 - 0.9482705 \tag{2.1}$$

where:
S_1 is the control signal intended to be received
S_2 is the corrected control signal

By first look at the fudge factor it could seem that it is almost insignificant. However, due to a non-linearity in the range of the ppm signals it has a significant influence. This can be seen in Figure 2.9 where the same inputs are plotted with and without the fudge factor along with the corresponding values for the translatory velocity in Figure 2.10

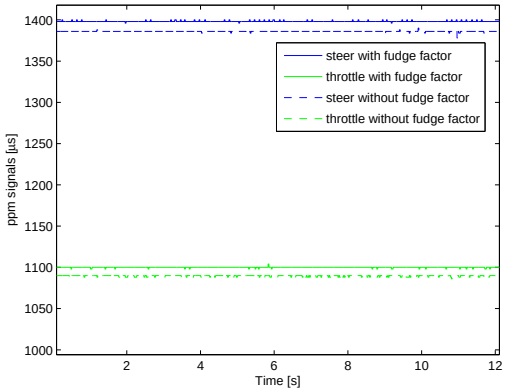


Figure 2.9: Plot of input signals with and without the fudge factor. The values sent out are steer = 1400 and throttle = 1100

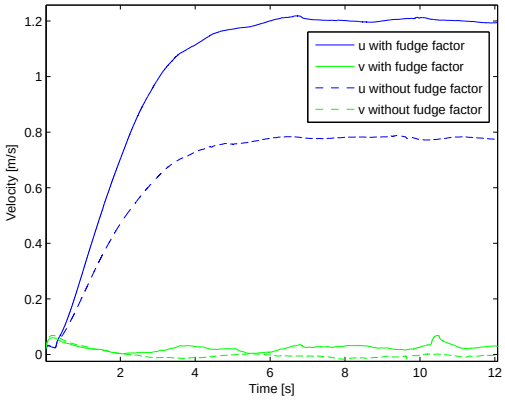


Figure 2.10: Plot of the translatory velocities u and v for the corresponding input signals

Chapter 3

Modeling

For this project two different models have been explored for the car. The first model used where based on system identification from recorded data where the car where driven around to try and explore as much of the state space as possible. From the data a locally weighted model where trained and used in the simulations. The model where based on the k-nearest neighbors from a current state, which where used to estimate the translatory and angular accelerations. However, due to problems with the removals of outliers the model never really became good enough. And due to the time limitations of the project the model where never improved enough to be used even though it where showing promising results.

Instead a more simple model have been developed based on the steady state values of the car. But before the modeling of the car some basic definitions are first defined to ease the modeling. These definitions includes the velocities of the car and the different frames which will be used throughout the rest of the report.

3.1 Frames

To ease the description and modeling three different frames are defined in Table 3.1.

Name	**Notation**
Earth inertial frame	$\{\mathbb{E}\}$
Phasespace frame	$\{\mathbb{P}\}$
Body frame	$\{\mathbb{B}\}$

Table 3.1: The three Cartesian frames used throughout this report

When wanting to specify which frame a coordinate or vector is defined in, a frame designation will be used as a superscript before the coordinate or vector. Hence, the vector v defined in frame $\{\mathbb{E}\}$ will be denoted by ${}^{\mathbb{E}}v$

Earth Inertial Frame $\{\mathbb{E}\}$
In order to make use of Newton's laws of dynamics, an Newtonian reference or earth inertial frame $\{\mathbb{E}\}$ needs to be defined [Bak, 2002, p. 8]. An inertial frame is a non-accelerated coordinate system, and all motion, position and attitude of the system will from this point be described relative to this earth inertial frame.

Body Frame $\{\mathbb{B}\}$
The body frame has its origin in the point of rotation of the system used. For helicopters and other systems without ground contact it is the Center of Mass (CM), but for the car the body frame is placed in the middle between the back wheels, since these can not turn.

Phasespace Frame $\{\mathbb{P}\}$
This frame is defined from the first camera connected to the Phasespace system. Therefore, the transformation from $\{\mathbb{P}\}$ to $\{\mathbb{E}\}$ has to be found, which is done in appendix A

3.2 Definition of Velocities in the Cars Body Frame

The velocities of the car captured with the Phasespace motion tracking system are, due to the extrinsic calibration found in appendix A, returned as velocities in the inertial frame $\{\mathbb{E}\}$. To simplify the modeling the translatory velocities are often transformed to the velocities u and v defined in the body frame $\{\mathbb{B}\}$ as depicted in Figure 3.1.

As can be seen the point of rotation is placed between the two back wheels. This might seem wrong but imagine the extreme case where the front wheels are steered orthogonal to the side. In this case (given the car is front wheel driven) the car would drive in circles around the point of rotation.

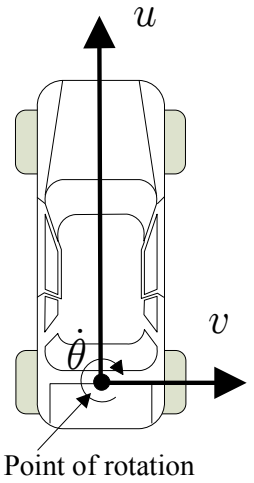


Figure 3.1: Illustration of car where the directions of the translatory velocities u and v are defined along with the angular rate $\dot{\theta}$ around the point between the back wheels

This transformation is done by a simple rotating the velocities around the point of rotation based on the cars heading θ as described by Equation (3.1),

$$^{\mathbb{B}}\begin{bmatrix} u \\ v \end{bmatrix} = \mathcal{R}_{\mathbb{BE}}(\theta) \; ^{\mathbb{E}}\begin{bmatrix} \dot{n} \\ \dot{e} \end{bmatrix} \tag{3.1}$$

where:

u is the translatory forward velocity in $\{\mathbb{B}\}$	[m/s]
v is the translatory sideways velocity in $\{\mathbb{B}\}$	[m/s]
n is the north position in $\{\mathbb{E}\}$	[m]
e is the east position in $\{\mathbb{E}\}$	[m]
$\mathcal{R}_{\mathbb{BE}}(\theta)$ is the rotation matrix from $\{\mathbb{E}\}$ to frame $\{\mathbb{B}\}$	[-]
θ is the orientation in $\{\mathbb{E}\}$	[rad]

where,

$$\mathcal{R}_{\mathbb{BE}}(\theta) = \begin{bmatrix} cos(\theta) & -sin(\theta) \\ sin(\theta) & cos(\theta) \end{bmatrix} \tag{3.2}$$

3.3 Model Based on Steady State Values

In this Section a simulation model, f, is estimated which can predict the next state based on the current state and the inputs,

$$x_{t+1} = f(x_t, u) \tag{3.3}$$

where:
x is the state of the system [-]
u is the control signals applied to the system [-]

Since the simulation model later will be used to find different trajectories it has been chosen to include the controls in the state,

$$\boldsymbol{x} = \begin{bmatrix} uv \\ ne \\ \dot{\theta} \\ \theta \\ u \end{bmatrix} \tag{3.4}$$

Where $uv = [u\ v]^T$ is a simplified notation which at times will be used throughout this report for the forward and sideways velocity. And $ne = [n\ e]^T$ is the simplified notation for the position in north. Since the controls are included in the state Equation (3.3) becomes,

$$x_{t+1} = f(x_t) \tag{3.5}$$

The idea of the model is to estimate the steady state velocities of the car for different values of the inputs steering and throttle. The accelerations are then estimated based on the difference in the current velocity and the steady state. The model consists of three 2D lookup tables containing the steady state values of the translatory velocities, u and v, and the angular rate, $\dot{\theta}$, along with a time constant, τ, for each table. The steady state values are found by applying different combinations of constant inputs and then estimate the velocities using the Phasespace motion tracking system and the Kalman filter described in Chapter 4.1 on page 27. In Table 3.2 the lookup table with the steady state values of the forward velocity are shown. Notice the deadzone when the throttle is between 970 and 1070.

Throttle \ Steer	596	1020	1420
596	-13	-20	-13
920	-1.44	-2.16	-1.44
946	-1.07	-1.6	-1.07
964	-0.76	-1.14	-0.76
970	0	0	0
1000	0	0	0
1070	0	0	0
1080	0.92	1.32	1.03
1100	1.62	2.1	1.67
1116	1.96	2.47	1.96
1256	1.05	6	1.05
1420	0.8	13	0.8

Table 3.2: 2D lookup table for the steady state values of the forward velocity u [m/s] based on the inputs throttle and steer

An 2D interpolation is then used to find the steady state values for arbitrary inputs. After having found the steady state values for the current inputs they are used to find the accelerations of the car,

$$a_x = \frac{x_{ss} - x}{\tau_x} \tag{3.6}$$

where:
x is a variable exchangeable with the velocities u, v or the angular rate θ [m/s], [rad/s]
a_x are the translatory or the angular acceleration [m/s^2], [rad/s^2]
τ_x are the time constants of the system [s]

After having found the translatory and angular accelerations, Euler integration is performed to find the next state,

$$\begin{bmatrix} u_{t+1} \\ v_{t+1} \\ \dot{\theta}_{t+1} \end{bmatrix} = \begin{bmatrix} u_t \\ v_t \\ \dot{\theta}_t \end{bmatrix} + dt \begin{bmatrix} \dot{u}_t \\ \dot{v}_t \\ \ddot{\theta}_t \end{bmatrix} \tag{3.7}$$

where:
dt is the time interval [s]

The position and orientation is then found using Euler integration on the velocities and angular rate. However, since the translatory velocities, u, v, are given relative to the body frame they are first transformed into velocities in the inertial frame, before Euler integration is applied to find the next position of the car.

$$\begin{bmatrix} n_{t+1} \\ e_{t+1} \end{bmatrix} = \begin{bmatrix} n_t \\ e_t \end{bmatrix} + dt \cdot \mathcal{R}_{\mathbb{EB}}(\theta) \begin{bmatrix} u \\ v \end{bmatrix} \tag{3.8}$$

where,

$$\mathcal{R}_{\mathbb{EB}}(\theta) = \mathcal{R}_{\mathbb{BE}}(\theta)^T \tag{3.9}$$

3.3.1 Validation of Steady State Values

To give a visualization of the steady state values reached, a validation is done for all combinations of the inputs to make sure that the interpolation of the lookup tables works. In Figure 3.2 and 3.3 the steady state values for u and v are plotted, respectively, as a function of the two inputs with a granularity of one in the inputs.

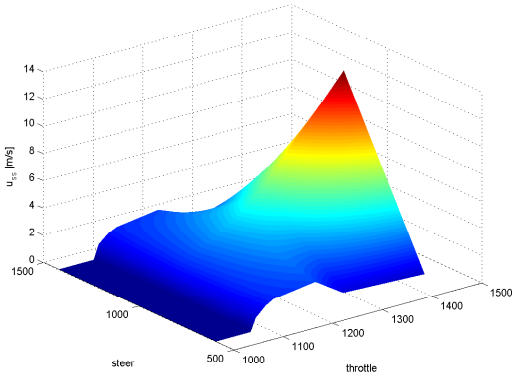


Figure 3.2: 3D plot of u as a function of the different combinations of controls

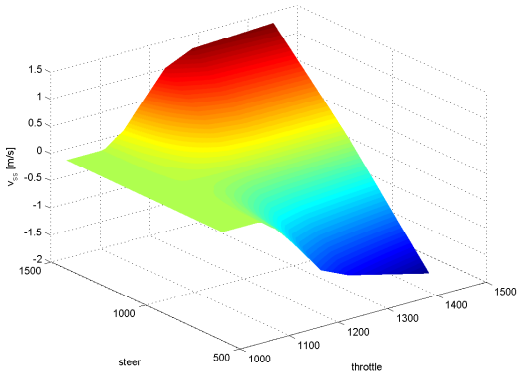


Figure 3.3: 3D plot of v as a function of the different combinations of controls

Notice in Figure 3.2 how the steady state value of the forward velocity, u_{ss}, goes close to zero when the car starts to drift, and the sideways velocity goes up or down depending on which side the car is drifting. In Figure 3.4 the same 3D plot are shown for the angular rate, $\dot{\theta}$. The very high/low values for the angular rate for maximum throttle and minimum/maximum steering are caused by the car drifting around it self causing the forward velocity to be close to zero.

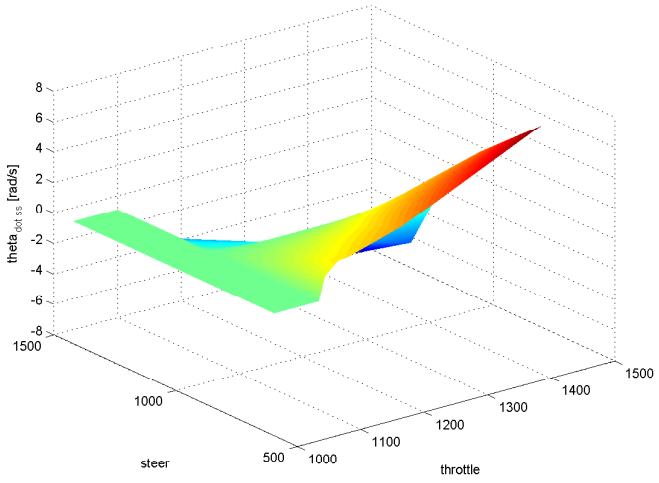


Figure 3.4: 3D plot of $\ddot{\theta}$ as a function of the different combinations of controls

The time constants in Equation (3.6) are in next subsection estimated and then optimized.

3.3.2 Estimation and Optimization of Time Constants, τ

Since the car has different dynamics for the three accelerations and depending on if the car is accelerating or de-accelerating, six different time constants are defined for the car. The time constants are found by recording the response of the car to a step input, and the time it takes to

reach 63% of the steady state value are found by visual inspection. The different time constants used can be seen in Table 3.3.

τ [s]	u	v	θ
Acceleration	2.8	0.5	0.2
De-acceleration	1.1	0.5	0.2

Table 3.3: Initial time constants for the car

The time constants found are then evaluated by recording a log with a lot of stops and steering. A simulation is then performed using the time constants, to compare the simulated state with the real state. Since the model is not perfect the simulation state is reset to the target state every 5 seconds, illustrated with the yellow dashed vertical lines.

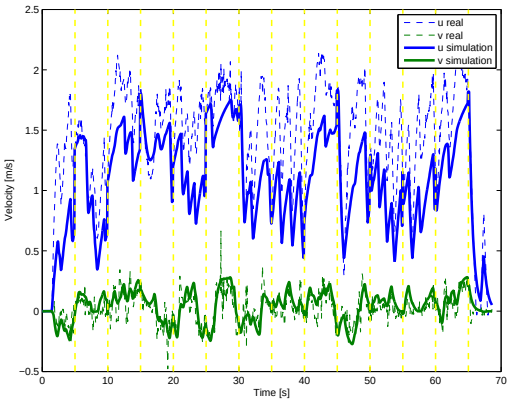


Figure 3.5: Plot of the real uv from a demonstration and a open loop simulation using the controls from the demonstration, before the optimization of τ. The dashed yellow lines indicates that the state is reset to the actual state

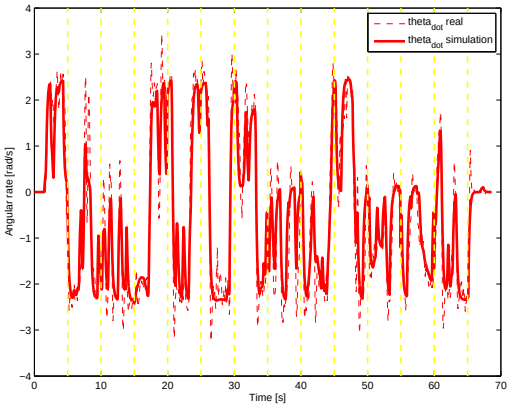


Figure 3.6: Plot of the real $\dot{\theta}$ from a demonstration and a open loop simulation using the controls from the demonstration, before the optimization of τ. The dashed yellow lines indicates that the state is reset to the actual state

As it is clear from Figure 3.5 and 3.6 the time constants, τ, can be improved, especially the acceleration of the forward velocity, u. Therefore, a line search has been performed to optimize the time constants. The optimization is done by simulating using the controls from the target and then compare the simulated state with the target state. A score is then computed to tell if the model is improved as the time constants are changed. The score is computed by the norm of the squared difference between the state and the target. To keep a good basis for comparing the state with the target throughout the simulation, the simulation state is again reset to the target state every 5 seconds. In Table 3.4 the time constants can be seen after the optimization. As expected the time constant for the forward velocity has been lowered significantly.

τ [s]	u	v	θ
Acceleration	1.0259	0.4798	0.1762
De-acceleration	0.5834	0.3211	0.1818

Table 3.4: Estimated time constants for the car found by performing a line search for each constant

In Figure 3.7 and 3.8 the same simulation is done using the new optimized time constants. It is clear that the optimization has improved the model significantly.

In Figure 3.8 is can be seen that there are some overshoot which is more clear in Figure 3.9 where it can be seen that the angular rate follows the measurement very nice except that the model fails to capture the clear second order dynamics with a overshoot. This 2. order dynamic is assumed to be caused mainly by the Kalman filter and are discarded in the model.

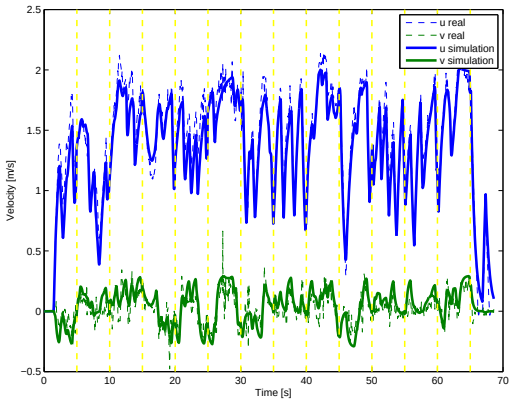


Figure 3.7: Plot of the real uv from a demonstration and a open loop simulation using the controls from the demonstration, after the optimization of τ. The dashed yellow lines indicates that the state is reset to the actual state

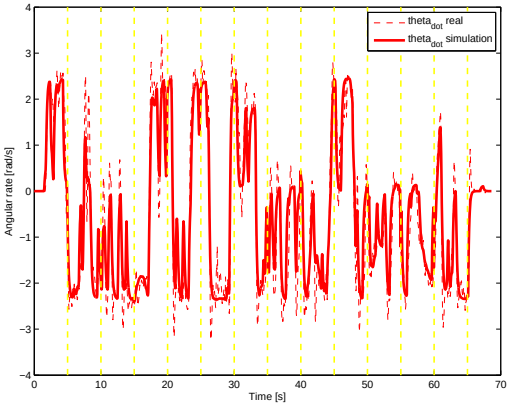


Figure 3.8: Plot of the real $\dot{\theta}$ from a demonstration and a open loop simulation using the controls from the demonstration, after the optimization of τ. The dashed yellow lines indicates that the state is reset to the actual state

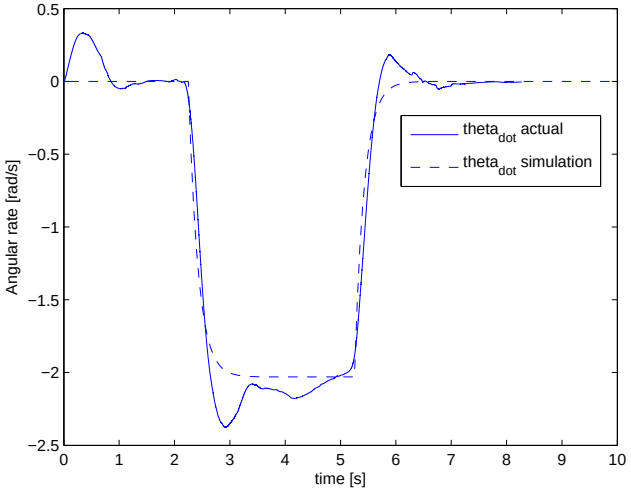


Figure 3.9: Plot of simulated and recored values of u, v for steer = 972 and throttle = 1100 where a steep is applied for the steering to the maximal value of 1420 and then back to 972 after having reached steady state

3.3.3 Implementation

The model is implemented in Matlab and works as an online simulation model which can be used to perform open or closed loop simulation or to be used by the Differential Dynamic Programming algorithm to compute a controller as described in chapter 5 on page 41. However, the 2D interpolation in Matlab are relatively time consuming, and therefore the model is discretized to speed up the simulation. This is done by precomputing all combinations of the inputs to avoid the 2D interpolation at runtime. The inputs could then be rounded off to the nearest precomputed value, however, this would result in problems later when the model is linearized by numerically computing the Jacobian, since a small change in the input then won't result in a change in the steady state value. Therefore, the inputs are rounded up and down and a linear regression is performed to find a more precise representation of the steady state value for that specific combination of controls, see Figure 3.10.

A linear regression are then found by solving Equation (3.10)

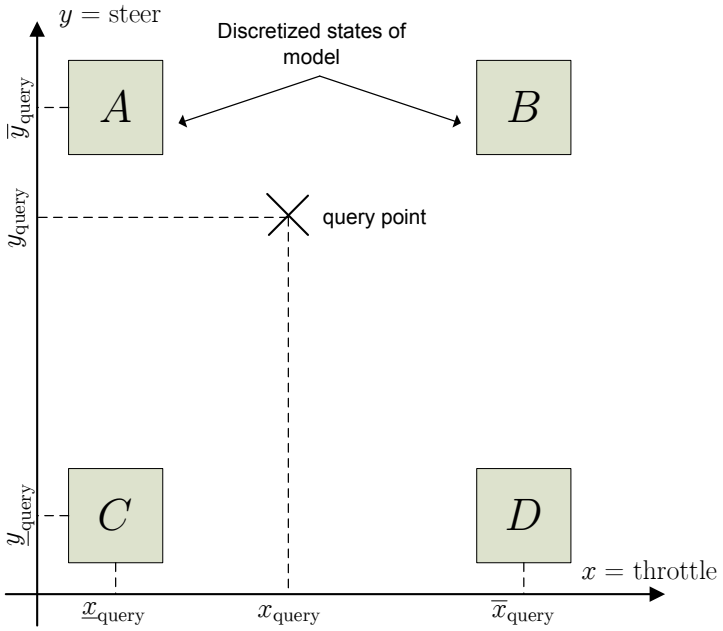


Figure 3.10: Illustration of discretized model with a query point and the four closest states in the model

$$\begin{bmatrix} 1 & x_A & y_A \\ 1 & x_B & y_B \\ 1 & x_C & y_C \\ 1 & x_D & y_D \end{bmatrix} \phi = \begin{bmatrix} z_A \\ z_B \\ z_C \\ z_D \end{bmatrix} \tag{3.10}$$

where:

A, B, C, D are the discretized states of the model for different combinations of controls	[-]
x is the steering value for a given state	[-]
y is the throttle value for a given state	[-]
z is the steady state value for a given state	[m/s^2], [rad/s^2]
ϕ is a vector that solves the equation	[-]

Given a query point linear regression is performed and the solution, ϕ, are used to compute the steady state value. This is done for each of the three tables, one for each velocity/angular rate.

$$z_{\text{query}} = [1 \; x_{\text{query}} \; y_{\text{query}}]\phi \tag{3.11}$$

where:

z_{query} is the steady state value for the query point	[m/s], [rad/s]

To verify that the discretized model is working a series of different open loop simulation has been performed with both models running for 50 time steps. When running the discretized model the computation time where 10-11 times faster, and the maximum difference in the estimated states where only 8.88e-6 which is acceptable.

3.4 Model Verification

To verify the model the car is driven around with a lot of stops and go and steering to make sure the dynamics of the car are used, while the cars state is then logged along with the controls sent to the car. Since it is the verification none of the parameters in the model has been optimized using this log. An open loop simulation is then performed using the discretized model with the control signals recorded. In Figure 3.11 and 3.12 the translatory velocities, uv, and the angular rate, $\dot{\theta}$, are plotted respectively. Since the model is not perfect the state of the simulation is reset to the actual state every 10 seconds indicated by the vertical yellow dashed lines in the Figures.

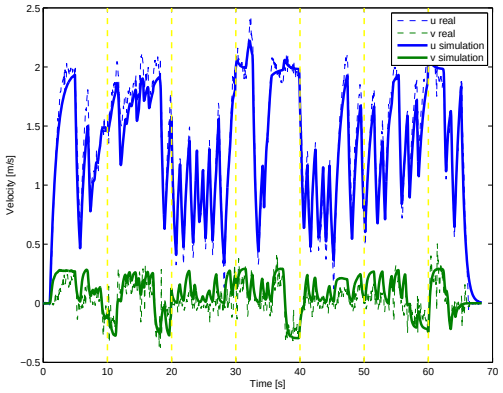


Figure 3.11: Verification of uv by comparing the state of a demonstration with a open loop simulation using the demonstrations controls. The dashed yellow lines indicates that the state is reset to the actual state

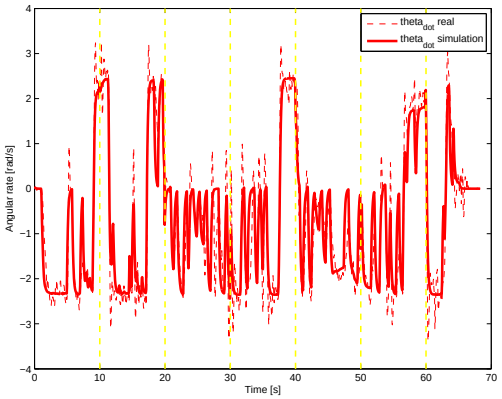


Figure 3.12: Verification of $\dot{\theta}$ by comparing the state of a demonstration with a open loop simulation using the demonstrations controls. The dashed yellow lines indicates that the state is reset to the actual state

As can be seen in Figure 3.11 and 3.12 the model captures the velocities and angular rate of the car are very good. A model of the car based on steady state values, has now been presented and verified. Before a controller can be designed the cars state has to be estimated which is the subject of the next chapter.

Chapter 4

Estimation of States

Even though the Phasespace system is capable of outputting the position and orientation of a defined rigid body, it has been chosen to implement the Kalman filter to track the state, since this gives the possibility of tuning the filter depending of the applications. Also the log likelihood is implemented to evaluate how well the Kalman filter estimates the state. And since the model is non-linear the linear Kalman filter is modified to an Extended Kalman Filter (EKF). A smoother is presented which can estimate the state better since it runs an extra pass over all the data. An Exploitation Maximization (EM) algorithm is then presented for estimating the covariance matrices used in the Kalman filter and smoother. Lastly in the Chapter there is a Section about the implementation of the Kalman filter, and a Section about how to modify the EKF such that it works with the marker positions, found by the Phasespace system, as observations.

4.1 Kalman Filter

The Kalman filter gives a way to calculate an estimate of the state x_t based on a partial output sequence $(y_0, y_1, ..., y_t)$. This is done by calculating or predicting the probability distribution of x_t given all past measurements of y up to and including the time t, denoted $P(x_t|y_{0:t})$. The probability distribution can then be used to estimate the state at time t. This means that the filter can be used to estimate or track the states in real time. Equation (4.1) and (4.2) are the state space representation of a linear system with process noise w_t and observation noise v_t. It is assumed that both the process and observation noise are Gaussian.

$$x_{t+1} = Ax_t + Bu_t + w_t \tag{4.1}$$
$$y_t = Cx_t + v_t \tag{4.2}$$

where:
w_t is a Gaussian noise term independent of x_t, $\quad w_t \sim \mathcal{N}(0, \Sigma_w)$
v_t is a Gaussian noise term independent of y_t, $\quad v_t \sim \mathcal{N}(0, \Sigma_v)$

As derived in Appendix B the states of a linear system can be estimated using a Kalman filter consisting of the Equations (B.22) to (B.26),

$$\hat{x}_{t+1|t} = A\hat{x}_{t|t} + Bu_t \tag{4.3}$$
$$P_{t+1|t} = AP_{t|t}A^T + \Sigma_w \tag{4.4}$$
$$K_{t+1} \triangleq P_{t+1|t}C^T(CP_{t+1|t}C^T + \Sigma_v)^{-1} \tag{4.5}$$
$$\hat{x}_{t+1|t+1} = \hat{x}_{t+1|t} + K_{t+1}(y_{t+1} - C\hat{x}_{t+1|t}) \tag{4.6}$$
$$P_{t+1|t+1} = P_{t+1|t} - K_{t+1}CP_{t+1|t} \tag{4.7}$$

Equation (4.6) is used to estimate the state x_{t+1}. The power of the Kalman filter is that it operates on-line, by updating the state and uncertainty with every new measurement. And as can be seen from the equations only the last state is needed for computing the optimal estimate, due to the Markov property. It can be seen that the predicted estimate, $\hat{x}_{t+1|t}$, is corrected by a term proportional to the error between the observed output and the prediction of the output, $(y_{t+1} - C\hat{x}_{t+1|t})$.

4.1.1 Extended Kalman Filter (EKF)

Since the system is non-linear an Extended Kalman Filter (EKF) is used instead. The EKF can be implemented in different ways but the method described in this section allows the use of the equations from the linear case. When the system is non-linear the system equation and output equation becomes,

$$x_{t+1} = f(x_t) + w_t \tag{4.8}$$
$$y_t = h(x_t) + v_t \tag{4.9}$$

The goal is to linearize the functions f and h such that,

$$f(x_t) \approx A_t x_t \tag{4.10}$$
$$h(x_t) \approx C_t c_t \tag{4.11}$$

For each time index the non-linear functions f and h can be linearized by calculation the Jacobian of f and h.

$$\mathcal{D}f(x_t) = \left.\frac{\delta f}{\delta x}\right|_{\hat{x}_{t|t}} \tag{4.12}$$
$$\mathcal{D}h(x_t) = \left.\frac{\delta h}{\delta x}\right|_{\hat{x}_{t|t}} \tag{4.13}$$

where:
$\mathcal{D}$ is the Jacobian of a function

The Jacobian can numerically be computed by performing a forward or backward step or by central difference. For precision the central difference method is chosen and are computed by Formula (4.14)

$$\mathcal{D}f(x) = \frac{f(x + \Delta x) - f(x - \Delta x)}{2\Delta x} \tag{4.14}$$

where:
f is the non-linear function
Δx is the step size

4.1.2 Test by Simulation

To test the Kalman filter a trajectory with two states has been generated using the linear system dynamics,

$$A = \begin{bmatrix} 1.1 & 0.1 \\ -0.2 & 1.03 \end{bmatrix} \tag{4.15}$$

However, even though the system dynamics used in the example are linear they are implemented as a function such that the EKF can be tested including the computation of the Jacobian. Process noise, w_t, and measurement noise, v_t, are then added as described in Equations (4.1) and (4.2) respectively. The process and observation noise are randomly generated from a zero mean Gausian distribution with the following covariances,

$$\Sigma_w = \begin{bmatrix} 0.1 & 0.05 \\ 0.05 & 0.3 \end{bmatrix} \qquad \Sigma_v = \begin{bmatrix} 1 & 1.5 \\ 1.5 & 3 \end{bmatrix} \tag{4.16}$$

Figure 4.1 shows one of the states in the trajectory without noise and with process and observation noise added along with the filtered state. As can be seen the filtered state are more smooth than the observations, and follows the real state better except in the beginning where the filtered state are bad due to a bad initialization of the state, x_0, and covariance matrix, P_0.

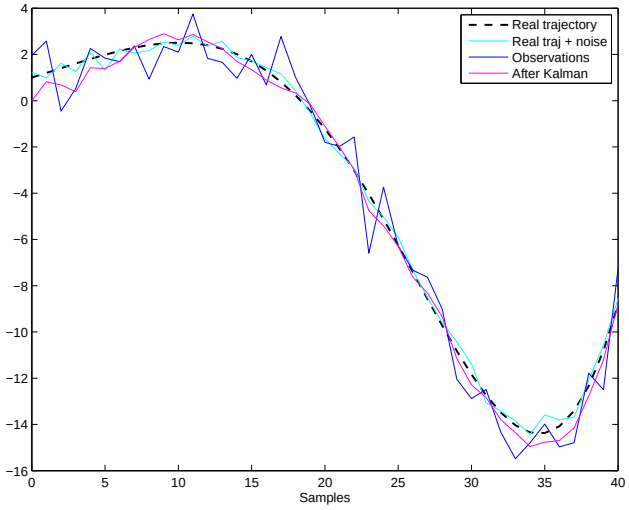


Figure 4.1: Simulation of Extended Kalman Filter where the filtered state are closer to the real state than the observations which the EKF are given

4.2 Log Likelihood of State Estimation

To evaluate the state estimation the likelihood of the estimations are calculated of the observable sequence $y_{0:T}$. This is done based on the observations and the prior knowledge.

$$\begin{aligned}\mathbb{P}(y_0, y_1, \ldots, y_T) =& \mathbb{P}(y_0)\mathbb{P}(y_1|y_0)\mathbb{P}(y_2|y_{0:1})\ldots\mathbb{P}(y_T|y_{0:T-1}) \\ =& \prod_{t=0}^{T} \frac{1}{(2\pi)^{d/2}\,|S|^{1/2}} e^{-\frac{1}{2}(y_{t+1}-\hat{y}_{t+1|t})^T (S)^{-1}(y_{t+1}-\hat{y}_{t+1|t})}\end{aligned} \tag{4.17}$$

where:
d is the dimension of y
$S = CP_{t+1|t}C^T + \Sigma_v$ is the innovation or residual covariance

The log likelihood is a good indication, as a single number, of how well the Kalman filter is tracking, or how well a series of states match some observations. The log likelihood is also used in section 4.4 on the facing page where an Expectation Maximization (EM) algorithm is presented to estimate the covariance matrices from data. There the log likelihood is used as to insure that the estimation is improved for each iteration of the EM algorithm as the covariance matrices gets estimated better.

4.2.1 Implementation

To simplify the implementation of Equation (4.17) the logarithm is taken which yields,

$$\begin{aligned}\mathbb{P}(y_0, y_1, \ldots, y_T) =& \sum_{t=0}^{T} \left\{ log\left(\frac{1}{(2\pi)^{d/2}\,|S|^{1/2}}\right) - \frac{1}{2}(y_{t+1}-\hat{y}_{t+1|t})^T S^{-1}(y_{t+1}-\hat{y}_{t+1|t}) \right\} \\ =& \sum_{t=0}^{T} \left\{ -\frac{d}{2}log(2\pi) - \frac{1}{2}log\left(|S|\right) - \frac{1}{2}(y_{t+1}-\hat{y}_{t+1|t})^T S^{-1}(y_{t+1}-\hat{y}_{t+1|t}) \right\}\end{aligned} \tag{4.18}$$

4.3 Rauch-Tung-Striebel (RTS) Smoother

Using the Kalman filter allows to estimate the states based on observations up to and including the time t. This section will concern the issue of obtaining estimates of the states based on all observations up to the time T, which is called a smoother. The Rauch-Tung-Striebel (RTS) smoother works by first forward filtering as described in section 4.1, followed by a backward pass through all the samples, which is the smoother. All measurements therefore have to be available to be able to run the smoother. And the smoother can therefore not be used in real time applications. The Equations, (4.19)-(4.21), for the Rauch-Tung-Striebel (RTS) smoother are derived in appendix C

$$L_t = P_{t|t}A^T P^{-1}_{t+1|t} \tag{4.19}$$

$$\hat{x}_{t|T} = \hat{x}_{t|t} + L_t(x_{t+1|T} - \hat{x}_{t+1|t}) \tag{4.20}$$

$$P_{t|T} = P_{t|t} + L_t(P_{t+1|T} - P_{t+1|t})L_t^T \tag{4.21}$$

Before running the backward pass the algorithm is initialized using the last values, $\hat{x}_{T|T}$ and $P_{T|T}$ from the forward filtering pass.

4.3.1 Test by Simulation

In Figure 4.2 the test simulation from the EKF are shown along with the smoothed state. As it is evident the smoothed state is much closer to the real trajectory compared to the filtered state. Also it can be seen that the initialization is improved allot compared to the filtered state, since the smoother is run as a backward pass based on all the data.

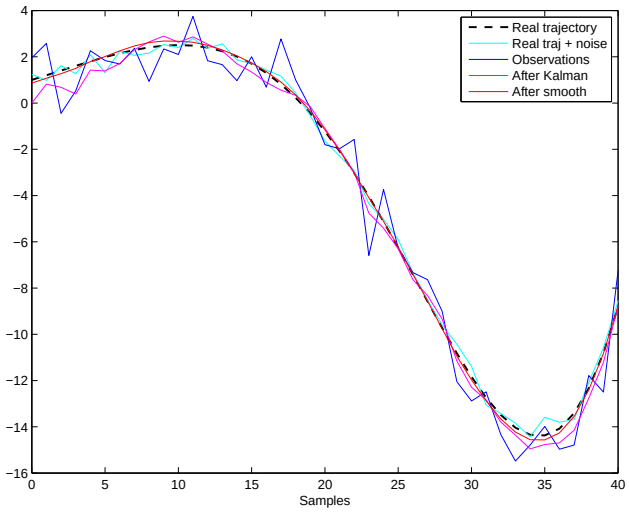


Figure 4.2: Simulation of Extended Kalman Filter where the filtered state are closer to the real state than the observations which the EKF are given

4.4 EM Algorithm for Estimating the Covariance Matrices

Using a Kalman filter to track an object proves to be very efficient if it is setup correct. However, if the Kalman filter is tuned wrong the estimation of the state will be bad. Therefore, the tuning of the Kalman filter is a very important task for getting a good state estimate. The tuning of the Kalman filter is depending on the following things:

- Initial state, x_0
- Initial covariance matrix, P_0
- Covariance matrix for system noise, Σ_w
- Covariance matrix for observation noise, Σ_v

Of course the performance of the filter also depends on the system and output equations, but since the model where verified in Section 3.4, the only thing left is the covariance matrices and the initialization. The initial state is simple since it for most cases will be some position and heading where the car starts, with all velocities and accelerations set equal to zero. And since both the initial state and covariance matrix will be updated in the filter, the initialization of them is not that significant. The covariance matrices Σ_w and Σ_v on the other hand is not being updated and therefore has a very significance influence on how well the Kalman filter is

tracking. It is therefore these two matrices there is referred to when saying that the Kalman need to be tuned, which is the subject of the this section.

Hand tuning the covariance matrices can prove to be a very calling task. Therefore, this section will present an Expectation Maximization (EM) algorithm from [Coates et al., 2008] which estimate the covariances matrices from recorded data by maximizing the following expression,

$$(\Sigma_w, \Sigma_v) = \arg\max_{\Sigma_w, \Sigma_v} \log\ P(y_0, \ldots, y_T | \Sigma_w, \Sigma_v, A, B, C) \tag{4.22}$$

In Equations (4.23)-(4.27) the EM algorithm equations are listed,

$$\delta x_t = \hat{x}_{t+1|T-1} - f(\hat{x}_{t|T-1}) \tag{4.23}$$
$$S_t = P_{t+1|T-1} - P_{t+1|T-1} L_t^T A_t^T - A_t L_t P_{t+1|T-1} \tag{4.24}$$
$$\hat{\Sigma}_w = \frac{1}{T} \sum_{t=0}^{T-1} \delta\mu_t \delta\mu_t^T + A_t P_{t|T-1} A_t^T + S_t \tag{4.25}$$
$$\delta y_t = y_t - h(\hat{x}_{t|T-1}) \tag{4.26}$$
$$\hat{\Sigma}_v = \frac{1}{T} \sum_{t=0}^{T-1} \delta y_t \delta y_t^T + C_t P_{t|T-1} C_t^T \tag{4.27}$$

where:
$\hat{\Sigma}_w$ is the estimated process covariance matrix [-]
$\hat{\Sigma}_v$ is the estimated observation covariance matrix [-]

As can be seen from Equations (4.23)-(4.27) the EM algorithm is depending on both the filtered and the smoothed states. Therefore the algorithm are implemented together with the smoother during the backward pass.

4.4.1 Test by Simulation

Since there is no guarantee that the EM-Algorithm will converge to an globally optimum, it is run multiple times with different initializations each time. This should give an indication of if the optimum found is globally. In figure 4.3 the log likelihood of the estimation is plotted as a function of the iterations of the EM algorithm. The EM-Algorithm has been run 10 times with different initializations but each of them converges to almost the same log likelihood, indicating a globally optimum. For the initialization random numbers between 0 and 1 are generated from a uniform distribution.

From Figure 4.3 it can be seen that for each of the 10 times the EM algorithm are run, the log likelihood is improved for each iteration and that they converges to the same point. Also it can be seen that a few iterations of the algorithm is enough to improves the log likelihood significantly.

Real values of covariance matrices,

$$\Sigma_w = \begin{bmatrix} 0.1 & 0.05 \\ 0.05 & 0.3 \end{bmatrix} \qquad \Sigma_v = \begin{bmatrix} 1 & 1.5 \\ 1.5 & 3 \end{bmatrix} \tag{4.28}$$

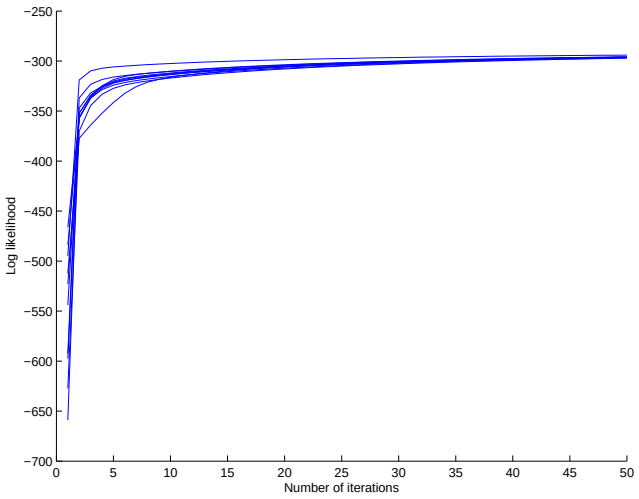


Figure 4.3: Plot showing the development of the log likelihood as 50 iterations of the EM algorithm is run. The different curves is the 10 different initializations of the EM algorithm

Mean values of covariance matrices after running the EM algorithm 10 times,

$$\bar{\Sigma}_w = \begin{bmatrix} 0.0532 & 0.0183 \\ 0.0183 & 0.0489 \end{bmatrix} \qquad \bar{\Sigma}_v = \begin{bmatrix} 0.9172 & 1.3438 \\ 1.3438 & 2.9673 \end{bmatrix} \tag{4.29}$$

Standard deviation of covariance matrices after running the EM algorithm 10 times,

$$\Sigma_{w\_std} = \begin{bmatrix} 0.0201 & 0.0157 \\ 0.0157 & 0.0304 \end{bmatrix} \qquad \Sigma_{v\_std} = \begin{bmatrix} 0.0228 & 0.0278 \\ 0.0278 & 0.0513 \end{bmatrix} \tag{4.30}$$

As can be seen the EM algorithm estimates the measurement covariance matrix, Σ_v, very well with a small standard deviation. When estimating the process covariance, Σ_w, the values are underestimated which could be explained by the low values it is trying to estimate and due to the limited amount of data used.

In figure 4.4 the state from the previous test example are estimated with the Kalman filter and smoother, using the estimated covariance matrices from Equation (4.29). As can be seen the smoothed state is very close to the real state even when the covariance matrices are estimated.

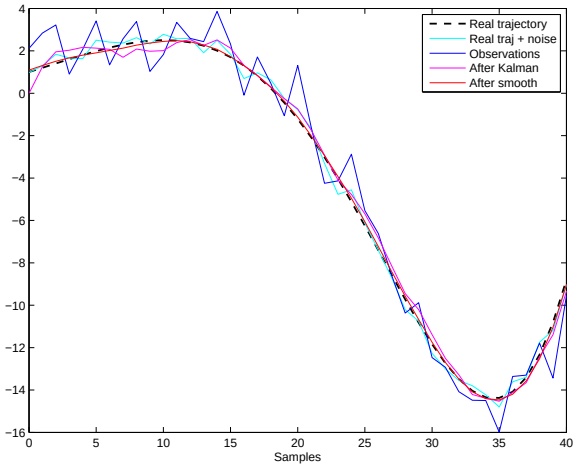


Figure 4.4: Simulation of Extended Kalman Filter using the covariance matrices Σ_w and Σ_v estimates using the EM-algorithm

4.5 Implementation

The Extended Kalman Filter is implemented in Matlab using the dynamics model based on steady state values described in Chapter 3.3. However, to be able to track the system in real time the EKF is also implemented in C++. This implementation uses the marker positions from the Phasespace motion tracking system as observations using the function `GetSetOfMarkersNonBlocking()` from table 2.1 in the system description. This means that the measurement vector contains the position of a marker in $\{\mathbb{E}\}$.

$$Y_{\text{marker ID}} = \begin{matrix} \\ \mathbb{E} \end{matrix} \begin{bmatrix} x_{\text{ID}} \\ y_{\text{ID}} \end{bmatrix} \tag{4.31}$$

where:

Y is the observation vector for the KF, $(Y = Cx)$	[m]
ID is the marker identification	[ID = A,B,C,D]

In Table 4.1 the implementation of the C++ Kalman filter are summarize by listing the C++ class along with the four public functions associated with it.

Function	**Description**
`KalmanFilter`	Constructor of the class which loads the initial state, x_0, and the covariance matrices, Σ_w and P_0 from files along with the position and variance for each marker
`~KalmanFilter`	De-constructor of the class which closes frees the allocated memory
`dynamicsUpdate`	Do a dynamics update using the function `SystemDynamics()`
`measurementUpdateMarker`	Updates the state, x_t, and the covariance matrix, P_t based on measurements from the Phasespace system
`SystemDynamics`	Simple dynamics model described in Section 4.6
`convergence_check`	Function to check if the state, x_t, and the covariance matrix, P_t, has converged

Table 4.1: The C++ class KalmanFilter and the four public functions associated with it

Both versions of the Kalman filer can be found on the enclosed CD.

4.6 Simple Dynamics Model used in C++ Kalman Filter

For the Kalman filter implemented in C++ a more simple dynamics model are used which do not require the control inputs. The model is implemented such that it keeps the velocities constant in $\{\mathbb{B}\}$ instead of propagating them forward by Euler integration in $\{\mathbb{E}\}$. The state of the car is shown in Equation (4.32),

$$\boldsymbol{x} = \begin{bmatrix} \dot{uv} \\ \dot{ne} \\ ne \\ \ddot{\theta} \\ \dot{\theta} \\ \theta \end{bmatrix} \tag{4.32}$$

where:

$\dot{uv}$ is the translatory acceleration in $\{\mathbb{B}\}$	[m/s^2]
$\dot{ne}$ is the translatory velocity in $\{\mathbb{E}\}$	[m/s]
ne is the position in $\{\mathbb{E}\}$	[m]
$\ddot{\theta}$ is the angular acceleration in $\{\mathbb{E}\}$	[rad/s^2]
$\dot{\theta}$ is the angular rate in $\{\mathbb{E}\}$	[rad/s]
θ is the orientation in $\{\mathbb{E}\}$	[rad]

First the translatory velocities, $\dot{ne}$, are transformed to $\{\mathbb{B}\}$ to perform the Euler integration in that frame,

$$uv_t = \mathcal{R}_{\mathbb{EB}}(\theta_t)\dot{ne}_t \tag{4.33}$$

Euler integration is then performed,

$$uv_{t+1} = uv_t + dt \cdot \dot{uv}_t \tag{4.34}$$
$$ne_{t+1} = ne_t + dt \cdot \dot{ne}_t \tag{4.35}$$
$$\dot{\theta}_{t+1} = \dot{\theta}_t + dt \cdot \ddot{\theta}_t \tag{4.36}$$
$$\theta_{t+1} = \theta_t + dt \cdot \dot{\theta}_t \tag{4.37}$$

where:

dt is the time interval since last update	[s]

After having updated the heading the translatory velocities are then transformed back to $\{\mathbb{E}\}$,

$$\dot{ne}_{t+1} = \mathcal{R}_{\mathbb{BE}}(\theta_{t+1})uv_t \tag{4.38}$$

The translatory and angular accelerations are then dampened to ensure that it is the measurements from the Phasespace system which are updating the state. This way the accelerations and velocities go to zero over a short period of time if no measurements are provided.

$$\dot{uv}_{t+1} = \dot{uv}_t \cdot e^{-dt} \tag{4.39}$$
$$\ddot{\theta}_{t+1} = \ddot{\theta}_t \cdot e^{-dt} \tag{4.40}$$

4.7 Estimation of Rigid Body State from Marker Observations

As described in Section 2.3 on page 13 about the Phasespace motion tracking system, the state of the car is estimated using active led markers which are places on the car as shown in Figure 4.5.

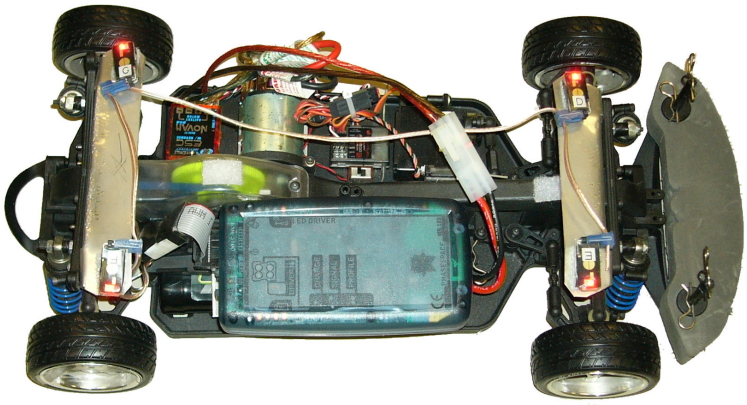


Figure 4.5: Overview of car where the four Phasespace LED markers can be seen

Since the observations from the Phasespace motion tracking system are for the markers the Kalman filter has to be changed a to incorporate these observations. This is done by changing the observation function, h, to the function `state2markers()`, which maps the state to the different marker positions. This way the state, x_t, and uncertainty covariance, P_t, can be updated by firstly changing the state into the different marker positions, and secondly using the Kalman filter equations with the marker observations. To do this transformation from the state to each of the marker positions the markers are defined in the rigid body frame relative to the point of rotation (center of the back wheels), along with the variance of each marker in the ${}^{\mathbb{E}}x$ and ${}^{\mathbb{E}}y$ direction, see Figure 4.6. The position of each marker can be seen in Table 4.2.

Marker ID	${}^{\mathbb{B}}x$	${}^{\mathbb{B}}y$
A	0.23409	-0.0563754
B	0.23653	0.0514787
C	0.00641	-0.0649576
D	0.00809	0.0594779

Table 4.2: Position of markers in the cars rigid body frame

The observation function, $h =$`state2markers(`x_t`)`, is defined as,

$$ {}^{\mathbb{E}}ne_{\text{Marker ID}} = \mathcal{R}_{\mathbb{BE}}(\theta)\ {}^{\mathbb{B}}xy_{\text{Marker ID}} + {}^{\mathbb{E}}ne; \tag{4.41} $$

where,

$$ \mathcal{R}_{\mathbb{BE}}(\theta) = \begin{bmatrix} cos(\theta) & -sin(\theta) \\ sin(\theta) & cos(\theta) \end{bmatrix} \tag{4.42} $$

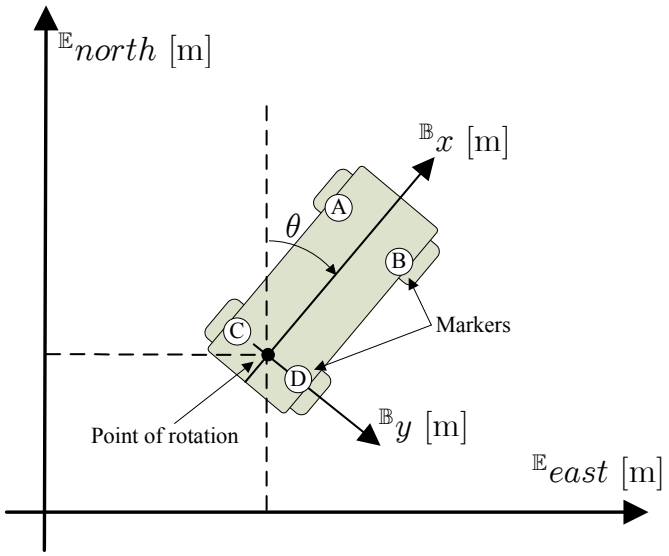


Figure 4.6: Illustration of how the cars state (position and orientation) are transformed to the four marker positions. Since the car is located in $\{\mathbb{E}\}$ with the four markers placed in the cars body frame $\{\mathbb{B}\}$

Algorithm

1. Compute the Jacobian linearization of the system,
   $x_{t+1} = A_t x_t$
2. Predict the state and covariance,
   $\hat{x}_{t+1|t} = A\hat{x}_{t|t} + Bu_t$
   $P_{t+1|t} = AP_{t|t}A^T + \Sigma_w$
3. Compute the Jacobian linearization of the observation function h =`state2markers(`x_t`)`,
   $y = C_t x_t$
4. For all active markers
5. $K_{t+1} = P_{t+1|t}C^T(CP_{t+1|t}C^T + \Sigma_v)^{-1}$
6. $\hat{x}_{t+1|t+1} = \hat{x}_{t+1|t} + K_{t+1}(y_{t+1} - C\hat{x}_{t+1|t})$
   $P_{t+1|t+1} = P_{t+1|t} - K_{t+1}CP_{t+1|t}$

As can be seen from Equations (4.20) - (4.21) no changes has to be made to the smoother when used to estimate the state of a rigid body using multiple marker observations. In Figure 4.7 the position of the markers are plotted along with the filtered and smoothed position of the rigid body.

Since the dynamics model, f, are used in the Kalman filter then the rest of the elements in state are also estimated. In Figure 4.8 the orientation of the car is depicted which corresponds to the position of the markers and the dynamics model. And in Figures 4.9-4.12 the translatory velocity, angular rate and velocities are depicted, respectively.

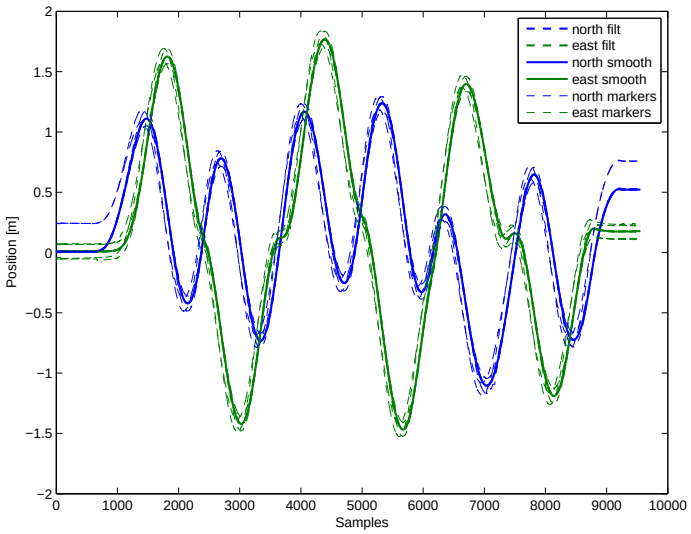


Figure 4.7: Position of the four markers which are triangulated by the Phasespace system, along with the filtered position of the cars rigid body

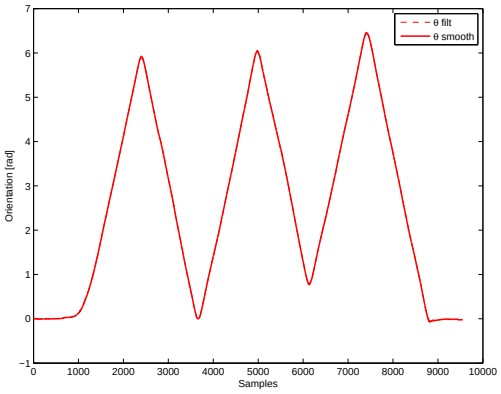


Figure 4.8: Orientation, θ, of the cars rigid body estimated using the Kalman filter

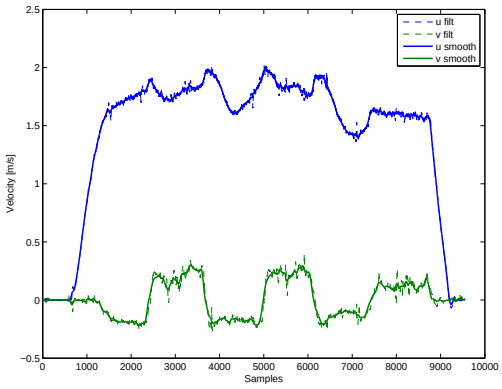


Figure 4.9: Translatory velocities, uv, of the cars rigid body estimated using the Kalman filter

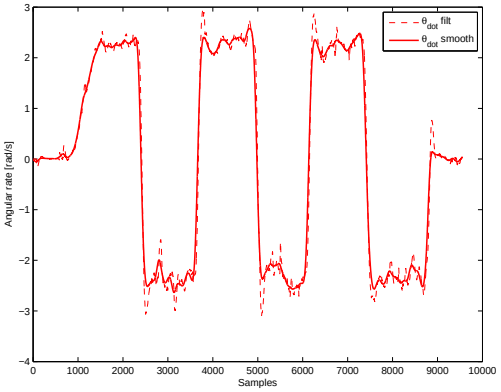


Figure 4.10: Angular rate, $\dot{\theta}$, of the cars rigid body estimated using the Kalman filter

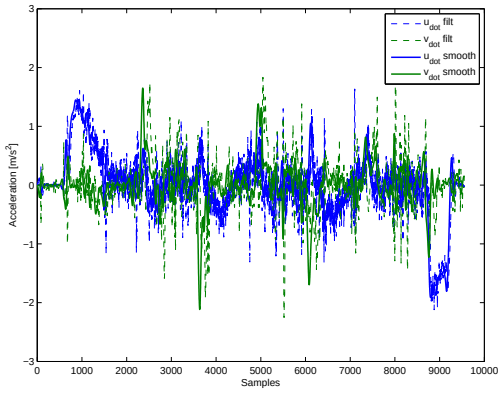


Figure 4.11: Translatory accelerations, $\dot{u}v$, of the cars rigid body estimated using the Kalman filter

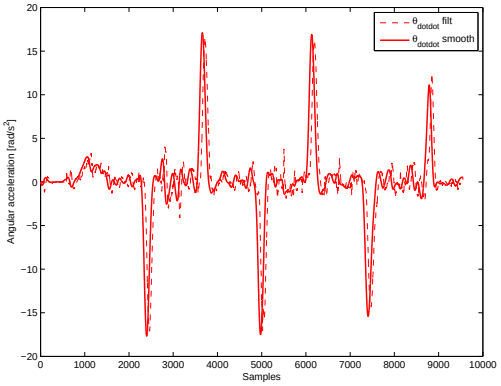


Figure 4.12: Angular accelerations, $\ddot{\theta}$, of the cars rigid body estimated using the Kalman filter

4.7.1 EM Algorithm when using Marker Observations

Since Equation (4.25) from the EM algorithm is independent of the marker observations and hereby also the output matrix C this equation does not change. However, since Equation (4.27) is dependent of the output this equation has to be iterated for each active marker. And a counter has to be introduced to keep track of the total number of active observations used in the EM-algorithm, in order to find the mean. I.e. there is divided with the total number of observations used instead of just T.

For the EM algorithm to estimate good covariance matrices it is important that the data collected contain a lot of aggressive moves, since this will bring out the dynamics. If for example the car in all the data is driven around with a constant speed then the corresponding entries in the covariance matrix will be very small compared to reality when the car moves more aggressively. Therefore, about 3 min of data is recorded with a lot of stops and as much steering as possible, and in about one minute of the data the car is driven very aggressively such that it is drifting around.

To avoid over fitting the data only the main diagonal of Σ_v and the accelerations of Σ_w are updated through each iteration of the EM algorithm. In Table 4.3 and 4.4 the mean values of the main diagonal of Σ_w and Σ_v are shown respectively along with the standard deviation. The values are found by running the EM algorithm with 5 iterations run 10 times with different initializations of the covariance matrices.

State	Mean value	Standard deviation
$\dot{u}$	10	XXX
$\dot{v}$	10	XXX
u	1e-2	0
v	1e-2	0
n	1.5e-5	0
e	1.5e-5	0
$\ddot{\theta}$	170	XXX
$\dot{\theta}$	1e-3	0
θ	1e-4	0

Table 4.3: Mean values and standard deviation of the main diagonal in Σ_w

State	Mean value	Standard deviation
$^{\mathbb{E}}x$	2.4e-3	1.769e-6
$^{\mathbb{E}}y$	3.6e-3	3.481e-6

Table 4.4: Mean values and standard deviation of the main diagonal in Σ_v

XXX COMMENT VARIABLES

An Extended Kalman Filter and smoother has now been presented along with an EM-algorithm to estimate the covariance matrices. In the following Chapter a controller will be found for making the system follow a trajectory based on the estimated state.

EM don't know the meaning of the variables

Chapter 5

Controller

This chapter describes the controller used in the project for trajectory following. First the theory behind the Linear Quadratic Regulator (LQR) is described, where a feedback controller is computed based on the dynamics of the system and a cost function, specified as two weighting matrices. After having found the equations used to compute the optimal controller for a Linear Time Varying (LTV) system the LQR is extended to an Differential Dynamic Programming (DDP) algorithm which will be used for trajectory following, since it computes a resulting trajectory which is feasible given the dynamics model. In order to improve the performance of the controller, feedforward inputs are in Section 5.6 added to the controller. And in Section 5.7 model biases is computed and added to correct for model inaccuracies. The chapter will conclude with simulations performed in order to clarify the performance of the developed controller, and a test on the real system. Throughout the chapter the state space and action space are assumed continuous.

5.1 Linear Quadratic Regulator (LQR) for a Linear Time Varying (LTV) System

The Linear Quadratic Regulator (LQR) control problem is a special class of Markov Decision Processes (MDPs), for which the optimal policy can be computed efficiently. The LQR is an optimal controller which are described in more detail in [Anderson and Moore, 1989]. Consider the following discrete Linear Time Varying (LTV) dynamics,

$$x_{t+1} = A_t x_t + B_t u_t \tag{5.1}$$

where:

$x_t \in \mathcal{R}^n$ denotes the state at time t	[-]
$u_t \in \mathcal{R}^p$ denotes the input at time t	[-]
$A_t \in \mathcal{R}^{n \times n}$ is describing the system dynamics	[-]
$B_t \in \mathcal{R}^{n \times p}$ is the input matrix	[-]

The control law of the system is described such that the feedback is a linear function of the states x and the feedback matrices K

$$u_t = K_t x_t \tag{5.2}$$

where:
$K_t \in \mathcal{R}^{p \times n}$ is the feedback matrix at time t [-]

It is supposed that all states can be measured without error and that the input signal, u_t is unlimited. An illustration of the linear state space system with the feedback matrices K is shown in figure 5.1

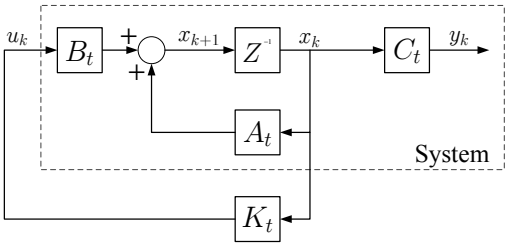


Figure 5.1: Block diagram of the closed loop system

The objective of controller design is to keep the output as close as possible to a reference using a small control signal. By use of the LQR method, the design goals are specified using a performance function. The purpose of using this method is to determine an input signal, u_t, such that the finite horizon performance function in Equation (5.3) is minimized. The discrete performance function is a sum over time of weighted squared states and weighted square inputs.

$$\mathcal{J} = \sum_{t=0}^{H-1} (x_t^T Q_t x_t + u_t^T R_t u_t) + x_H^T P_H x_H \tag{5.3}$$

where:
H is the finite time-horizon [-]
$Q_t \in \mathcal{R}^{n \times n}$ is the weighting matrices for the states and has to be positive semidefinite [-]
$R_t \in \mathcal{R}^{p \times p}$ is the weighting matrix for the inputs and has to be positive definite [-]
P_t is the cost-to-go which is the cost incurred in all future steps if the agent acts optimal [-]

The cost-to-go, P_t, is the total cost, meaning all the penalty which will be accumulated from the time, t, until the horizon, H. Hence, the total cost of being in state x_0 at time $t = 0$ and following the optimal policy is given by $J(x_0) = x_0^T P_0 x_0$. More generally, the optimal cost-to-go for being in state x_t at time t is given by $x_t^T P_t x_t$. Where the cost-to-go is the cost incurred in all future steps, which means that the cost-to-go will decrease to zero as the times go to the horizon H.

Q_t has to be positive semidefinite and R_t has to be positive definite to ensure that all nonzero control signals will give a positive contribution to the performance function. The minimization of the performance function will lead to the optimal controller, by calculating the feedback matrices K_t such that the control law in equation (5.2) minimizes the performance function. The controller that minimizes the performance function is called a Linear Quadratic Regulator (LQR).

$$\begin{aligned} &\min_{u_0 \dots u_{H-1}} \mathcal{J} \\ &= \min_{u_0 \dots u_{H-1}} \sum_{t=0}^{H-1} (x_t^T Q_t x_t + u_t^T R_t u_t) + x_H^T P_H x_H \end{aligned} \tag{5.4}$$

$$= \min_{u_0 \dots u_{H-2}} \sum_{t=0}^{H-2} (x_t^T Q_t x_t + u_t^T R_t u_t) + x_{H-1}^T Q_{H-1} x_{H-1} + \min_{u_{H-1}} u_{H-1}^T R_{H-1} u_{H-1} + x_H^T P_H x_H \tag{5.5}$$

First the minimization is done over u_{H-1} which is the last two expressions of Equation (5.5) and by utilizing Equation (5.1) yields,

$$\begin{aligned} &u_{H-1}^T R_{H-1} u_{H-1} + x_H^T P_H x_H = \\ &\quad u_{H-1}^T R_{H-1} u_{H-1} + (A_{H-1} x_{H-1} + B_{H-1} u_{H-1})^T P_H (A_{H-1} x_{H-1} + B_{H-1} u_{H-1}) \end{aligned} \tag{5.6}$$

This expression is convex quadratic in u_{H-1}, and the minimum can be found by setting the gradient equal to zero,

$$\nabla_{u_{H-1}}(\cdot) = 0 = 2R_{H-1} u_{H-1} + 2B_{H-1}^T P_H A_{H-1} x_{H-1} + 2B_{H-1}^T P_H B_{H-1} u_{H-1} \tag{5.7}$$

This gives the following expression for u_{H-1},

$$\begin{aligned} u_{H-1} &= -(R_{H-1} + B_{H-1}^T P_H B_{H-1})^{-1} B_{H-1}^T P_H A_{H-1} x_{H-1} \\ &= K_{H-1} x_{H-1} \end{aligned} \tag{5.8}$$

where,

$$K_{H-1} = -(R_{H-1} + B_{H-1}^T P_H B_{H-1})^{-1} B_{H-1}^T P_H A_{H-1} \tag{5.9}$$

Inserting this and Equation (5.2) back into Equation (5.5) yields,

$$\min_{u_0 \dots u_{H-1}} J$$

$$= \min_{u_0 \dots u_{H-1}} \sum_{t=0}^{H-1} (x_t^T Q_t x_t + u_t^T R_t u_t) + x_H^T P_H x_H \tag{5.10}$$

$$= \min_{u_0 \dots u_{H-2}} \sum_{t=0}^{H-2} (x_t^T Q_t x_t + u_t^T R_t u_t) + x_{H-1}^T P_{H-1} x_{H-1} \tag{5.11}$$

where,

$$P_{H-1} = Q_{H-1} + K_{H-1}^T R_{H-1} K_{H-1} + (A_{H-1} + B_{H-1} K_{H-1})^T P_H (A_{H-1} + B_{H-1} K_{H-1}) \tag{5.12}$$

As can be see Equation (5.11) is exactly of the same format as the original problem from Equation (5.4), which is written out again in Equation (5.10). Hence, this procedure can be repeated for each time step. This gives the following dynamic programming algorithm to find the optimal controllers for a LTV system with quadratic costs,

for $t = H-1, H-2, ..., 0$

$$K_t = -(B_t P_t B_t + R_t)^{-1} B_t^T P_{t+1} A_t \tag{5.13}$$

$$P_t = Q_t + K_t^T R_t K_t + (A_t + B_t K_t)^T P_{t+1} (A_t + B_t K_t) \tag{5.14}$$

5.2 Differential Dynamic Programing (DDP)

As mentioned the Differential Dynamic Programming (DDP) is an extension of the LQR and where first presented in [Jacobson and Mayne, 1970]. The DDP is a time varying controller which consist of a series of feedback matrices, one for each time instance. It approximately solves general continuous state space MDPs by iterating the following forward and a backward pass.

1. **Backward pass:** Compute a linear approximation to the dynamics and a quadratic approximation to the reward function around the trajectory obtained when using the current policy (controller). Then compute the optimal policy for the LQR problem using Equations (5.13) and (5.14)

2. **Forward pass:** Simulate by setting the current policy equal to the optimal policy found in the LQR problem to obtain the new resulting trajectory. This means that a closed loop simulation are done using the non-linear simulation model in the length of the target trajectory

The linear approximation of the dynamics are found by computing the Jacobian linearization of the non-linear dynamics. More specific this is done by linearizing around the current resulting trajectory, such that for each time instance in the resulting trajectory are used as an equilibrium to compute a linear representation of the system dynamics for that specific time index. In the first iteration the system is linearized around the target trajectory. Since the DDP in this project is used for trajectory following, a small change has to be made to the algorithm.

5.2.1 DDP for Trajectory Following

Since the algorithm is needed for trajectory following the standard formulation presented in Equation (5.3) need to be extended such that the it is a function of the difference between the state and the desired trajectory $x_0^*, \ldots, x_H^*$. This is archived by rewriting the reward function to a function of the error state $e_t = x_t - x_t^*$ rather than the actual state x_t. This way the algorithm can be used to track some desired state reference sequence.

$$\mathcal{J} = \sum_{t=0}^{H-1} \left(-(x_t - x_t^*)^T Q_t (x_t - x_t^*) - u(t)^T R u(t) + (x_H - x_H^*)^T P_H (x_H - x_H^*) \right) \tag{5.15}$$

When the DDP computes a controller it uses the simulation model which means that the target trajectory has to be feasible with the given dynamics model. Hence, if a slightly non-feasible target trajectory is given, the DDP algorithm will change the target trajectory in the forward pass, to a trajectory which is feasible with the simulation model. And if the target trajectory and simulation model are not consistent then the algorithm will not converge.

Since the controller is precomputed by linearizing the system around the resulting trajectory it is important that the car follows the trajectory. Because if the system start deviating to much from the target then the controller is no longer optimal and the deviation from the linearization point may result in very poor performance.

5.3 Implementation

The DDP has been implemented in Matlab where the feedback matrices are precomputed and saved in a file. The main C++ program then computes the control signals using these pre-computed feedback matrices and the current state. The DDP controller has been implemented such that the control law from Equation (5.2) have been modified to give the change in controls instead of the control signals,

$$\Delta u_t = K_t(x_t - x_t^*) \tag{5.16}$$

where:
$\Delta u_t = u_t - u_{t-1}$ is the change in controls from the previous time [-]

Therefore, it has been chosen to have the previous controls, u_{t-1}, in the the DDP state since this means that the actual controls can be computed as the previous controls added with the feedback. The state vector used in the DDP is therefore defined as,

$$\boldsymbol{x}_{\text{ddp}} = \begin{bmatrix} uv \\ ne \\ \dot{\theta} \\ \theta \\ u_{t-1} \\ \Delta u_{t-1} \end{bmatrix} \tag{5.17}$$

As can be seen in the state vector a term has been added to the reward function that penalizes the change in inputs over consecutive time steps,

$$\Delta u_{t-1} = u_{t-1} - u_{t-2} \tag{5.18}$$

In particular, this term will penalize the controller for changing the controls from the controls at the previous time step. This is an implementation choice and means that the weighting matrix R for the inputs is set to zero since a cost for change in inputs are added in the state. In Figure 5.2 a flowchart is illustrating how the precomputed DDP controller is used. As illustrated the controller is read from a file along with the target states, x^*.

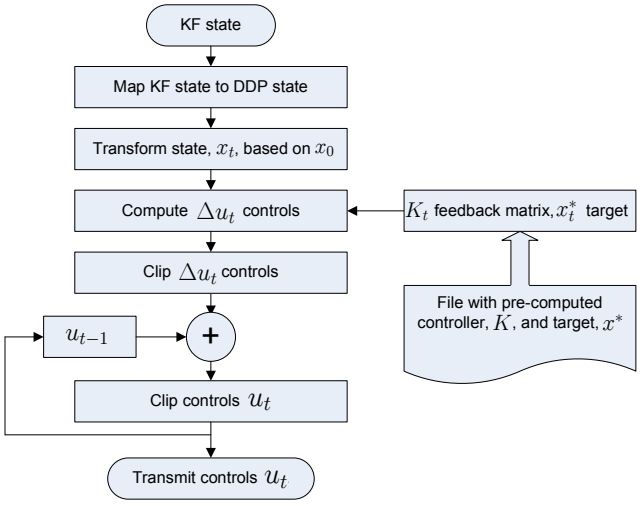


Figure 5.2: Flowchart of the DDP controller

After the Extended Kalman Filter has estimated the state it is mapped into the DDP state. The state is then transformed relative to the initial state, x_0, and the initial target, x_0^*, such that the controller works independent of where the car starts or is orientation, see Figure 5.3.

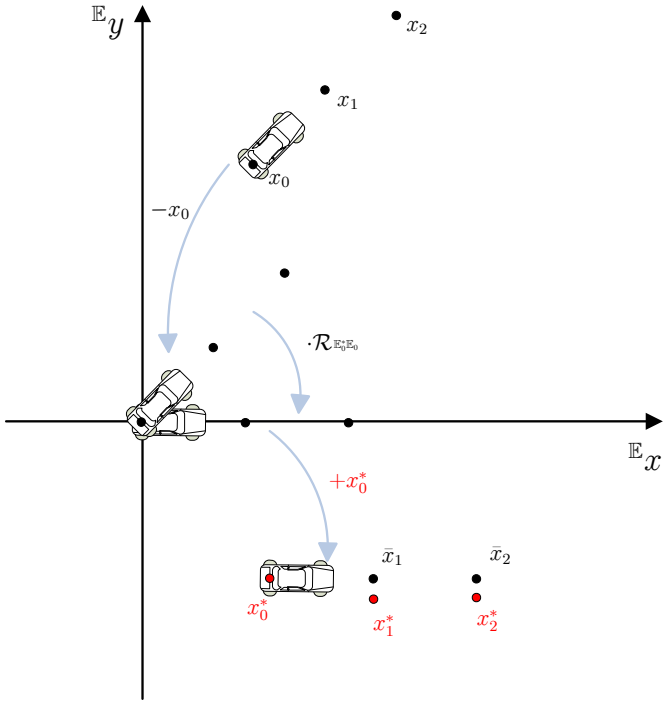


Figure 5.3: Illustration of how the state, x_t, is transformed based on the initial state, x_0, and the initial target, x_0^*

The transformation from the state, x_t, to the transformed state, $\bar{x}_t$, which is comparable with the target independent of the start state and initial target, is described by Equations (5.19) and (5.20). Normally the translatory velocities would also need to be transformed, however since the velocities in the DDP state are given in the cars body frame they do not need to be transformed.

$$\begin{bmatrix} \bar{n}_t \\ e_t \end{bmatrix} = \mathcal{R}_{\mathbb{E}_0^*\mathbb{E}_0}(\Delta\theta_0) \left(\begin{bmatrix} n_t \\ e_t \end{bmatrix} - \begin{bmatrix} n_0 \\ e_0 \end{bmatrix} \right) + \begin{bmatrix} n_0^* \\ e_0^* \end{bmatrix} \tag{5.19}$$

$$\bar{\theta}_t = \theta - \Delta\theta_0 \tag{5.20}$$

where:

$\Delta\theta_0 = \theta_0 - \theta_0^*$ is the difference in heading between the initial state and the initial target	[rad]
$\mathcal{R}_{\mathbb{E}_0^*\mathbb{E}_0}$ is the rotation matrix from the initial frame $\{\mathbb{E}_0\}$ to the targets initial frame $\{\mathbb{E}_0^*\}$	[-]
$\bar{ne}_t$ is the transformed position at time t	[m]
$\bar{\theta}_t$ is the transformed orientation at time t	[rad]

This means that the control law from Equation (5.16) is changed to use the transformed state, $\bar{x}_t$,

$$\Delta u_t = K_t(\bar{x}_t - x_t^*) \tag{5.21}$$

The rotation matrix $\mathcal{R}_{\mathbb{E}_0^*\mathbb{E}_0}$ is defined as,

$$\mathcal{R}_{\mathbb{E}_0^*\mathbb{E}_0} = \begin{bmatrix} cos(\Delta\theta_0) & -sin(\Delta\theta_0) \\ sin(\Delta\theta_0) & cos(\Delta\theta_0) \end{bmatrix} \tag{5.22}$$

Likewise the inverse of this transformation is performed on the target, see Equations (5.23) and (5.24), such that the target visually can be compared to the state, when displayed in a 3D graphics frontend as described further in Chapter 6 on page 59 about the software.

$$\begin{bmatrix} \bar{n_t^*} \\ e_t^* \end{bmatrix} = \mathcal{R}_{\mathbb{E}_0\mathbb{E}_0^*}(\Delta\theta_0) \left(\begin{bmatrix} n_t^* \\ e_t^* \end{bmatrix} - \begin{bmatrix} n_0^* \\ e_0^* \end{bmatrix} \right) + \begin{bmatrix} n_0 \\ e_0 \end{bmatrix} \tag{5.23}$$

$$\bar{\theta}_t = \theta + \Delta\theta_0 \tag{5.24}$$

where,

$$\mathcal{R}_{\mathbb{E}_0\mathbb{E}_0^*} = \mathcal{R}^T_{\mathbb{E}_0^*\mathbb{E}_0} \tag{5.25}$$

After the transformation of the state the feedback controls are computed using Equation (5.21). As illustrated in Figure 5.2 the computation of the feedback controls are done based on the current target and the feedback matrices which are saved in a file. The feedback change in controls, Δu_t, are then clipped at some defined limits. The previous controls are then added to give the current feedback controls which also are clipped. The feedback controls are then send out to the system using the function `TxControls()` from the C++ class `RxTxControls()` which where described in Table 2.2 in the system description.

In Table 5.1 the implementation of the controller are summarize by listing the C++ class along with the its five public functions associated with it.

Function	**Description**
`Controller`	Constructor of the class which loads the target and feedback matrices to memory, and takes in the initial state, x_0, used for transforming the state
`~Controller`	De-constructor of the class which closes the files and frees the allocated memory
`ComputeControls`	Computes the controls based on the current time and state
`GetInitialControls`	Gets the initial controls of the target
`GetTarget`	Gets the target at the current time
`GetFeedforwardControls`	If used it gets the feedforward controls at the current time
`TransformeState`	Rotate and translate the current state, x_t, based on the initial state, x_0, into the frame of the controller. Also it transforms the target into the frame of the initial state

Table 5.1: The C++ class Controller and the five public functions associated with it

When running the DDP algorithm the feedback matrices are computed for a specific time interval which is 50 milliseconds corresponding to 20 Hz. Therefore, it is important that the main loop runs with the same time interval which will be described further in Chapter 6 on page 59 about the overall software setup.

5.4 Simulation using a Circle

To test the DDP a simple trajectory has been defined, where the car in simulation is driving around in a counter clockwise circle. The DDP algorithm are then used to compute an optimal controller, and the performance of the controller are tested in a closed loop simulation. In figures 5.4 - 5.9 the position (n, e), translatory velocity (u, v), heading (θ), angular rate ($\dot{\theta}$), and the controls (steer and throttle) are illustrated along with a Matlab Quiver plot which plots the position and heading in one plot to give a better illustration of the maneuver. In all the plots the target trajectories are plotted along with the resulting trajectories after having run the DDP algorithm.

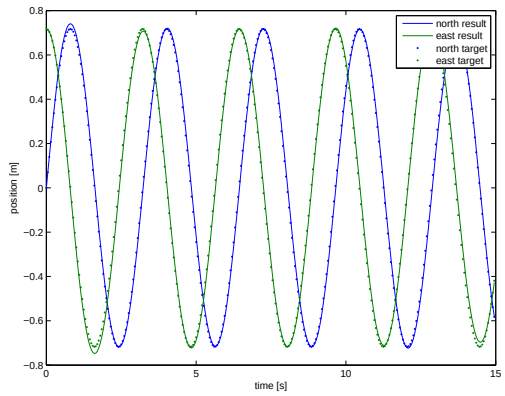


Figure 5.4: Position: north and east given in the earth initial frame

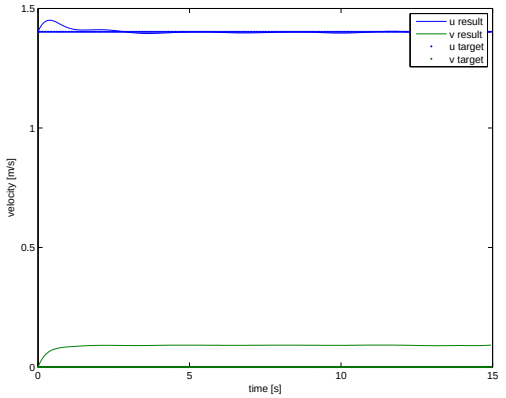


Figure 5.5: Forward velocity, u, and sideways velocity, v, given in the body frame

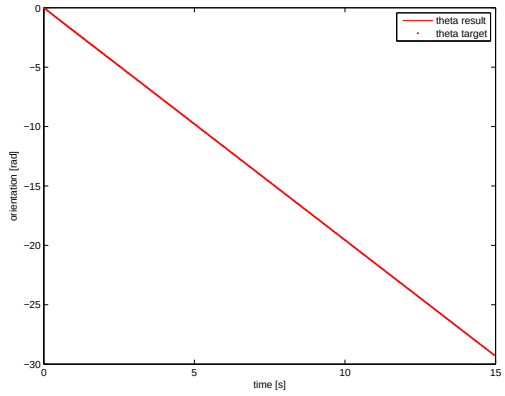


Figure 5.6: Heading θ

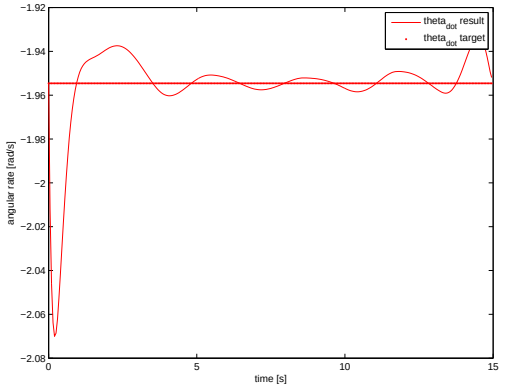


Figure 5.7: Angular rate $\dot{\theta}$

As can be seen from the plots the specified trajectory comply with the dynamic model since the resulting trajectory is lying on top of the specified trajectory in most of the states. However, as can be seen there is a inconsistencies in the sideways velocity since the target is set to zero sideways velocity, and that the DDP have found that to complete the target with the given dynamics model the resulting target has to have a small sideways velocity for it to be feasible. This also results in the change in the angular rate to correct for this sideways velocity.

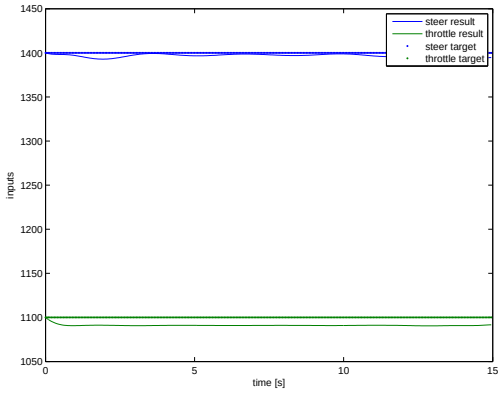


Figure 5.8: Control inputs steer and throttle

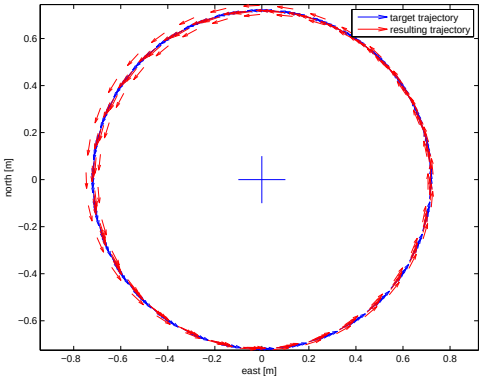


Figure 5.9: Matlab Quiver plot of position and heading

To test the computed controller an closed loop simulation is done where the car is started with a small offset to the trajectory, see Figures 5.10-5.15.

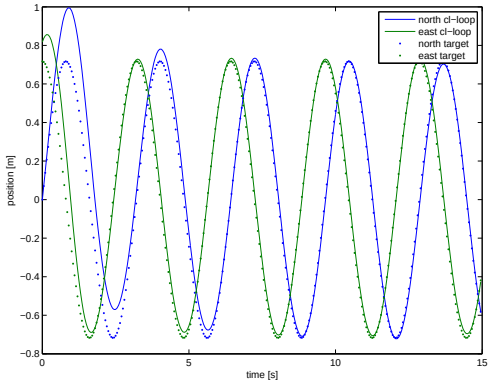


Figure 5.10: Position: north and east given in the earth initial frame in closed loop simulation

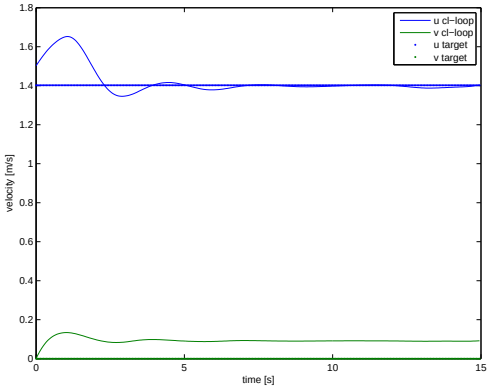


Figure 5.11: Forward velocity, u, and sideways velocity, v, given in the body frame in closed loop simulation

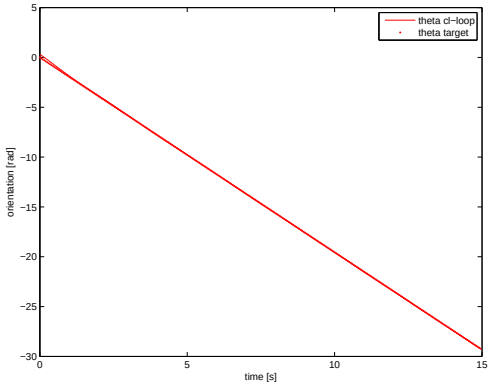


Figure 5.12: Heading θ in closed loop simulation

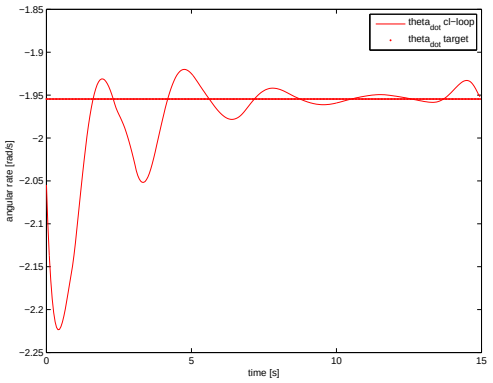


Figure 5.13: Angular rate $\dot{\theta}$ in closed loop simulation

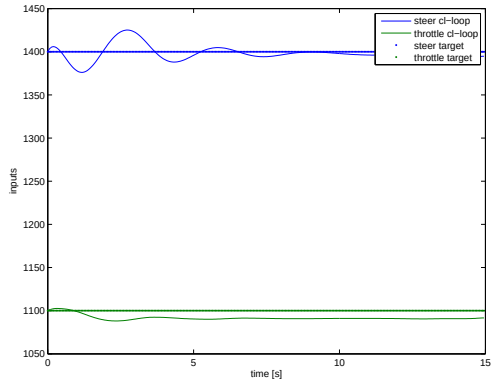


Figure 5.14: Control inputs steer and throttle in closed loop simulation

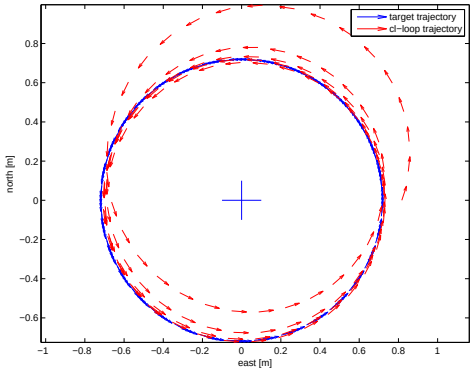


Figure 5.15: Matlab Quiver plot of position and heading in closed loop simulation

As can be seen from Figures 5.10-5.15 the controller is capable of controlling the car around in the specified trajectory, and also corrects the mismatch in the starting state as intended.

5.5 Simulation using an 8-Trajectory from a Demonstration

To test the performance of the DDP controller a more difficult trajectory is used. The trajectory is a 8-trajectory which is obtained from a single demonstration. The target trajectory is obtained by driving the car around in a 8-trajectory while the Kalman filter is tracking the state which is saved to a file. The Kalman filter state is then transformed into the DDP state which is subsampled by interpolation to 20 Hz. In Figures 5.16 - 5.21 the target and resulting trajectories from the DDP are plotted.

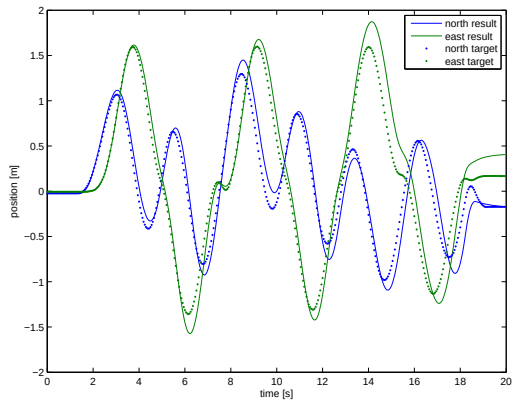


Figure 5.16: Position: north and east given in the earth initial frame

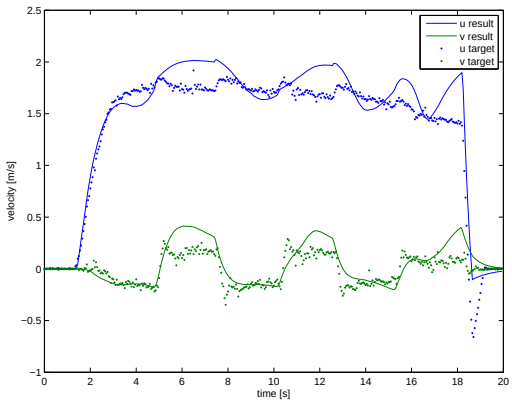


Figure 5.17: Forward velocity, u, and sideways velocity, v, given in the body frame

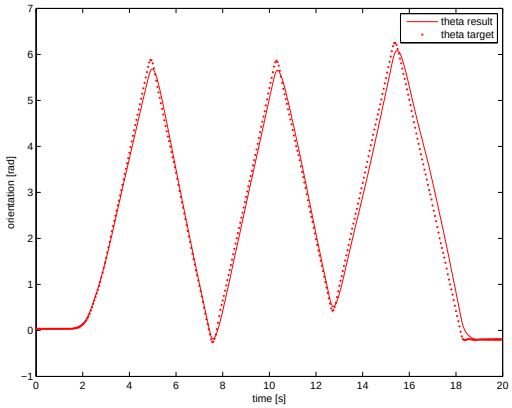


Figure 5.18: Heading θ

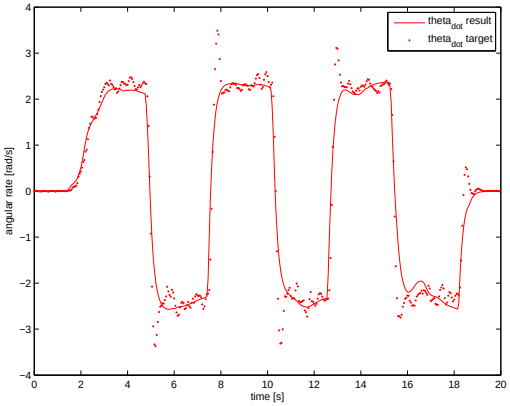


Figure 5.19: Angular rate $\dot{\theta}$

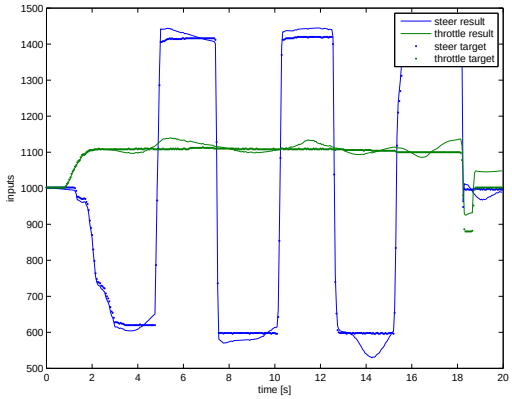


Figure 5.20: Control inputs steer and throttle

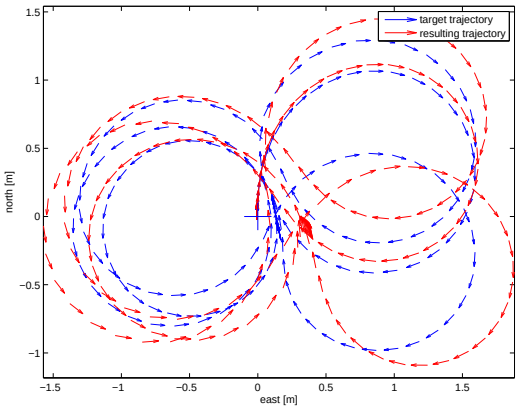


Figure 5.21: Matlab Quiver plot of position and heading

As can be seen from the plots the performance is not that good and in figure 5.22 where the feedback matrices are plotted over time it can be seen that there are some very large spikes which is a problem if the car deviates from the target trajectory in time. The large feedback values appear because the steering in the trajectory is changing very dramatically as can be seen from figure 5.20. This means that no matter how large a penalty there are used for changing in controls the large spikes in the feedback matrices will always be there. Therefore, feedforward controls are introduced such that the controls only are penalized for deviating from the feedforward controls.

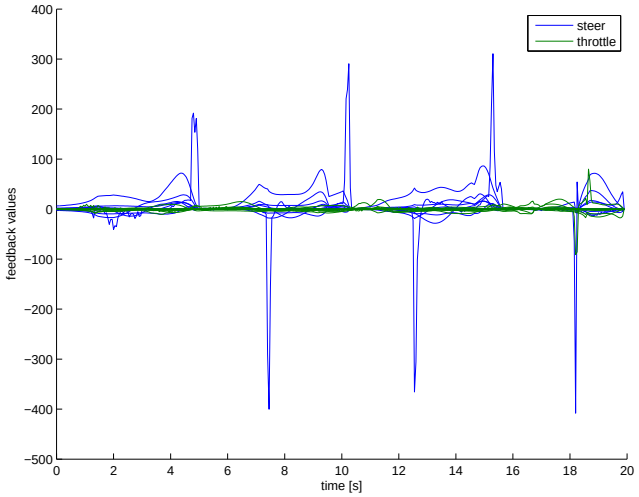


Figure 5.22: Plot of all values of feedback matrices over time

5.6 Introducing Feed-forward Controls

The feedforward controls are introduced by adding them to in the simulation model. In Figure 5.23 the DDP flowchart is updated with the feedforward controls, $\tilde{u}_t$.

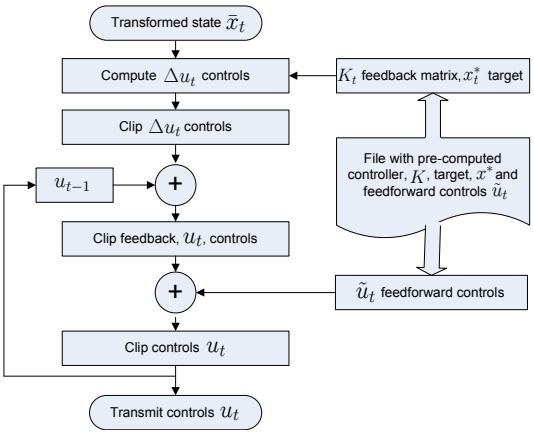


Figure 5.23: Flowchart of the DDP controller with feedforward controls

The effect of adding the feedforward controls can be seen in Figures 5.24 - 5.29. Since the feedforward controls are added in the simulation model the DDP algorithm do not know of them, which means that the target for the controls is all zeros as can be seen in Figure 5.28.

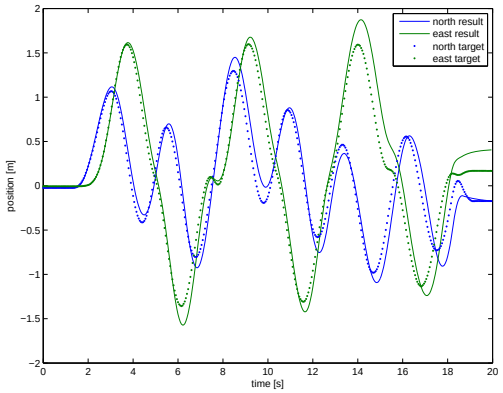


Figure 5.24: Position: north and east given in the earth initial frame

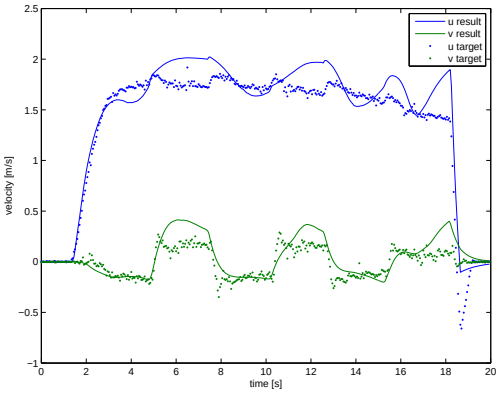


Figure 5.25: Forward velocity, u, and sideways velocity, v, given in the body frame

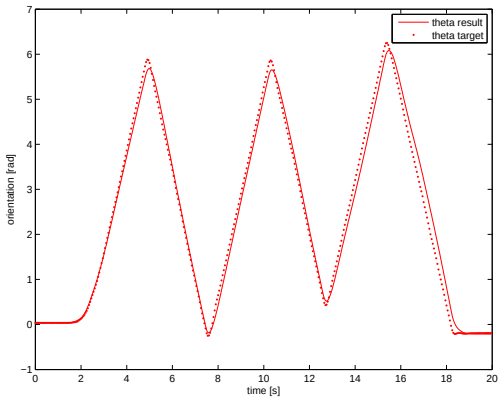


Figure 5.26: Heading θ

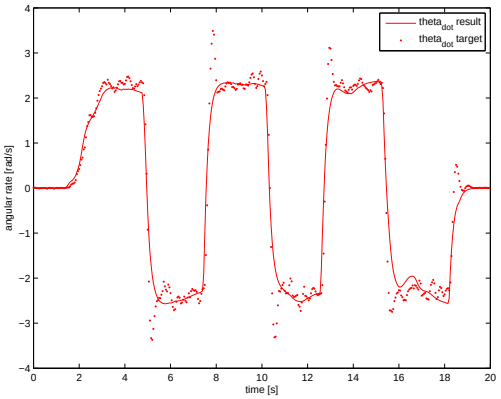


Figure 5.27: Angular rate $\dot{\theta}$

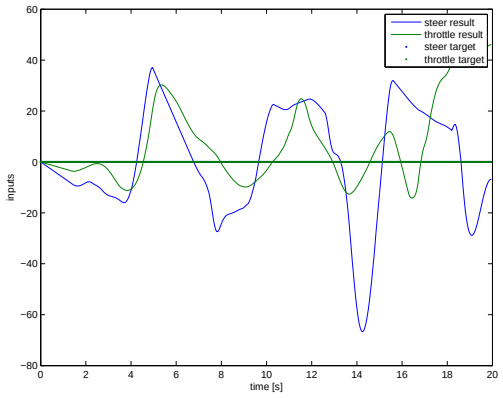


Figure 5.28: Control inputs steer and throttle

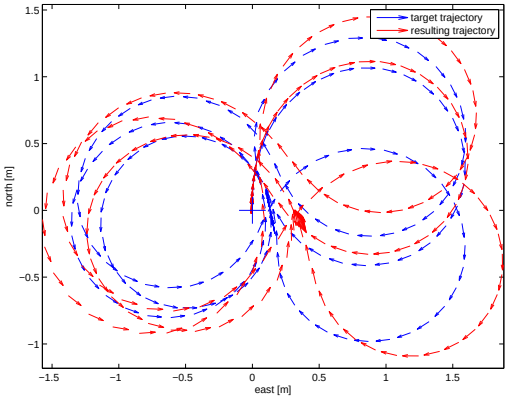


Figure 5.29: Matlab Quiver plot of position and heading

As can be seen in Figure 5.30 the spikes in the feedback matrices are now gone since the controller now only penalizes for changing of controls compared to the feedforward controls. Notice that all the values in the feedback matrices near the end are going to zero. This is called the end-effect and are caused by the cost to go, P_t, being close to the horizon H, which means that it is small and therefore the algorithm will allow small errors in the state resulting in the feedback values going to zero. Therefore, it is important not to use the last few feedback matrices computed.

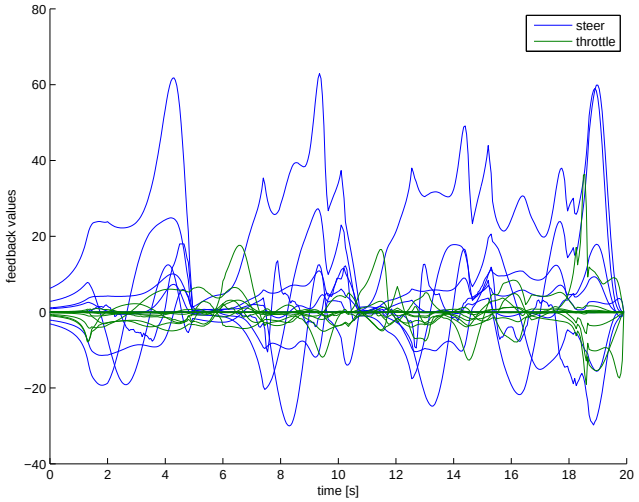


Figure 5.30: Plot of all values of feedback matrices over time

However, as it is evident from the rest of the plots and very clear in the Quiver plot in Figure 5.29 the performance is still poor. This is due to inaccuracies between the simulation model and the real system. To help improve the simulation model bias terms are introduced for each time instance.

5.7 Introducing Model Biases to Correct for Model Inaccurate

Since the states are found by Euler integration of the accelerations it is the accelerations which are the most important terms. Therefore, time dependent model biases is computed and added to the simulation model for the three accelerations. This then works as a feed forward loop which fills in the difference between the demonstration and the open loop simulation,

As shown in Equation (5.26) the bias term, β_t^*, is computed for each time instance by taking the difference in accelerations between the actual next state and the simulated next state.

$$\beta_t^* = \begin{bmatrix} \dot{u}_{t+1} \\ \dot{v}_{t+1} \\ \ddot{\theta}_{t+1} \end{bmatrix}^* - f\left(\begin{bmatrix} \dot{u}_t \\ \dot{v}_t \\ \ddot{\theta}_t \end{bmatrix}^* \right) \tag{5.26}$$

This way a bias term is computed for each time instance. Since the accelerations are not a part of the DDP state they are computed from the velocities by inverse Euler integration. Due to noise in the observations the model biases are smoothed by running a moving average over the computed biases. The model biases for the 8-trajectory are plotted in Figures 5.31 and

5.32 along with the smoothed values. The idea of adding model biases where first presented in [Abbeel et al., 2006b].

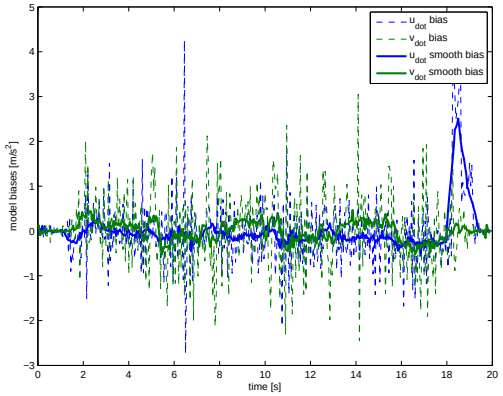


Figure 5.31: Model biases for $\dot{u}$ and $\dot{v}$ along width the mean of the model biases computed by a moving average

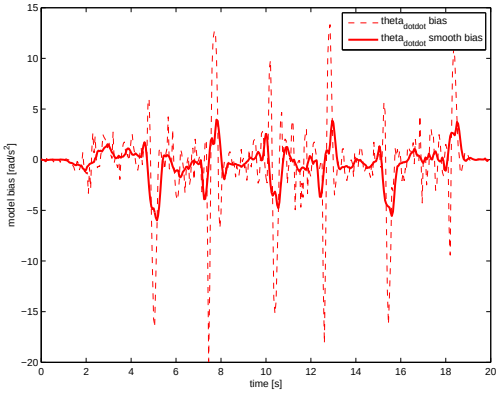


Figure 5.32: Model bias for $\ddot{\theta}$ along width the mean of the model bias computed by a moving average

In Figure 5.33 - 5.38 the new resulting trajectory can be seen compared to the target trajectory. As it is clear the performance of the controller has improved significantly since the difference between the simulation model and the real system has been corrected by adding the model biases. Though the resulting trajectory are still a dampened version of the target which is clear in figure 5.36. This is due to the fact that the model biases are smoothed by meaning out the noise.

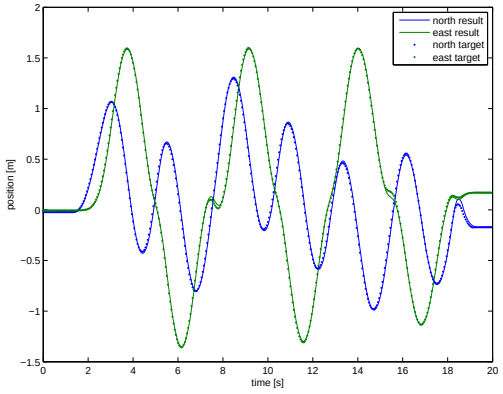


Figure 5.33: Position: north and east given in the earth initial frame

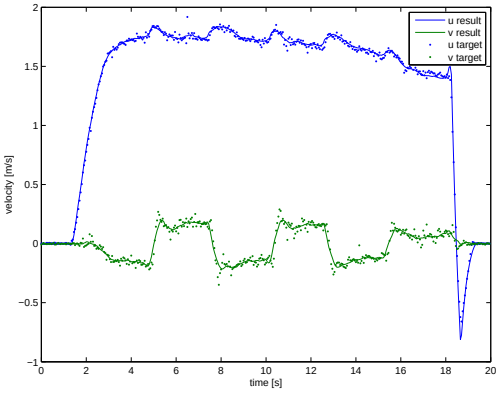


Figure 5.34: Forward velocity, u, and sideways velocity, v, given in the body frame

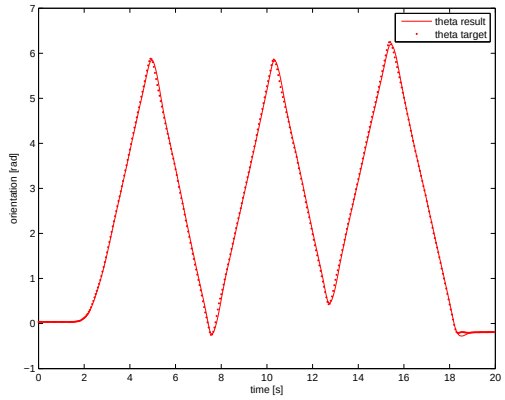


Figure 5.35: Heading θ

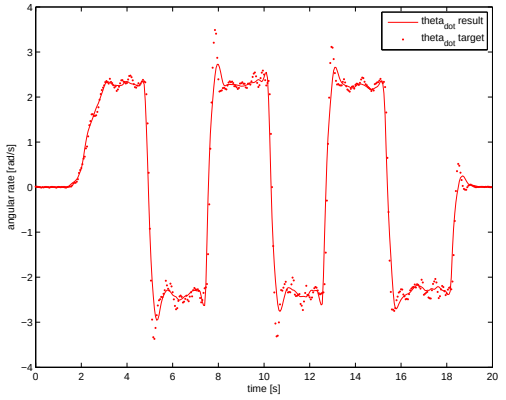


Figure 5.36: Angular rate $\dot{\theta}$

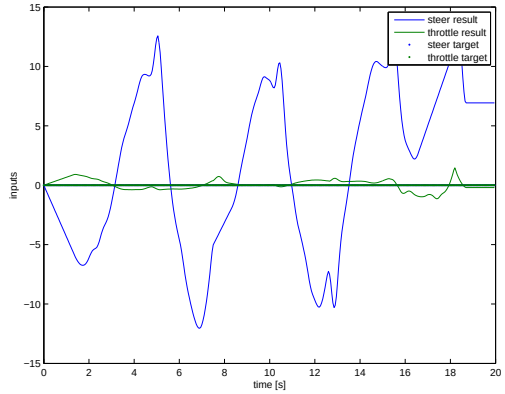


Figure 5.37: Control inputs steer and throttle

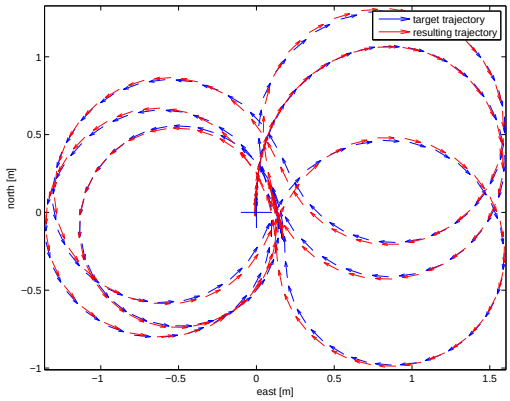


Figure 5.38: Matlab Quiver plot of position and heading

Using the model biases to correct for model accuracies is a very powerful and simple way to improve a model. However, it only works when a demonstration is available, and it only works when the system stays relative close to the target trajectory especially in time. Also it adds on any noise from the demonstration, so it is important that the demonstration is good. Therefore it can be necessary to estimate the target trajectory and model biases from multiple demonstrations, as done in [Coates et al., 2008], to get a target and model biases with less noise.

5.8 Test on Real System

The computed controller are then tested on the real system using the 8-trajectory from the simulations as the target. In Figure 5.39 - 5.44 the real state can be seen compared to the target trajectory. As it is clear especially from the Quiver plot in Figure 5.44 the car is following the target very nicely. And as can be seen in Figure 5.43 where the feedback controls are plotted, the controller is performing feedback and especially changing the steering very aggressively.

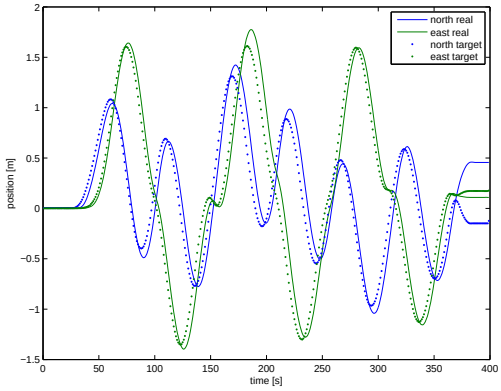


Figure 5.39: Position: north and east given in the earth initial frame

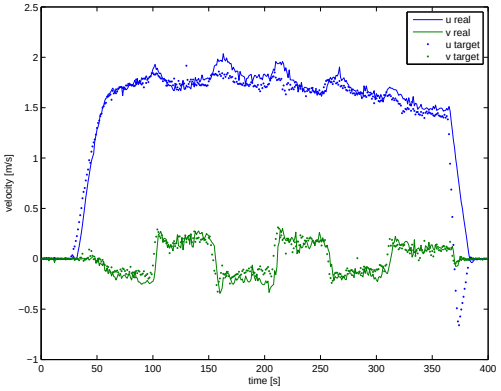


Figure 5.40: Forward velocity, u, and sideways velocity, v, given in the body frame

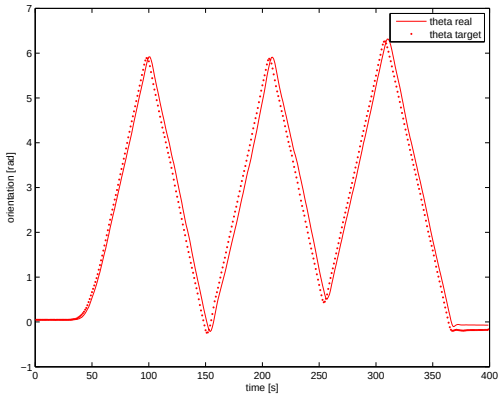


Figure 5.41: Heading θ

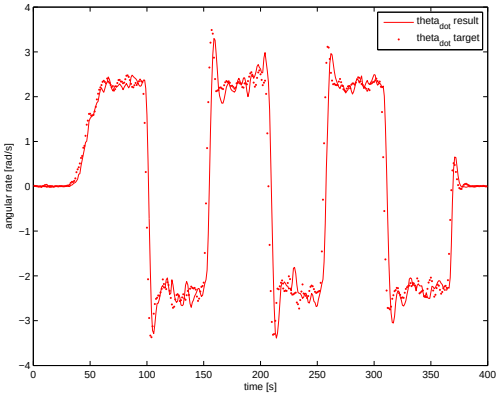


Figure 5.42: Angular rate $\dot{\theta}$

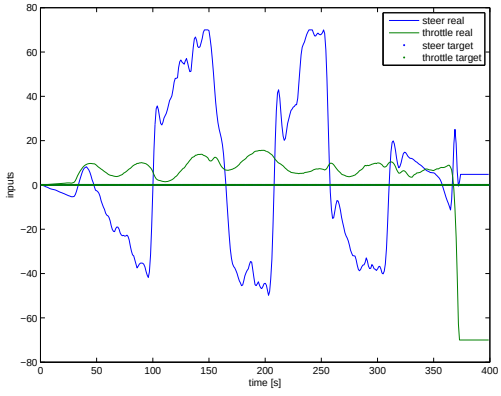


Figure 5.43: Feedback control inputs steer and throttle, where zero feedback correspond to the feedforward controls

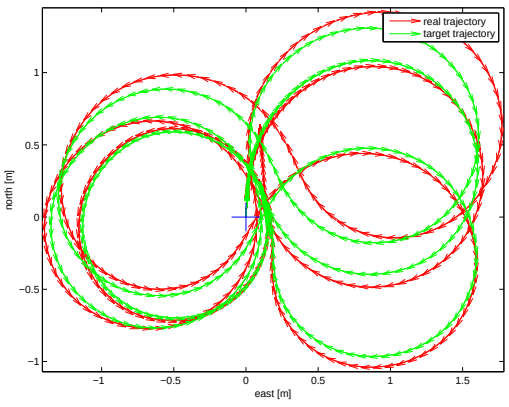


Figure 5.44: Matlab Quiver plot of position and heading

A trajectory following controller has now been presented and tested in simulations. Feedforward controls and model biases have been introduced which improved the performance significantly. Lastly the controller has been tested on the real system which performed very well. In the next Chapter an overview is given of all the software.

Chapter 6

Software

Throughout this report some of the software has been mentioned such as the C++ classes `Phasespace()`, `RxTxControls()`, `KalmanFilter()` and `Controller()`. This Chapter will describe the rest of the software and how all the software interacts. Later in this Chapter the flow of the different functions and classes will be described, but first an overview of the primary functions and classes used in this project are illustrated in Figure 6.1.

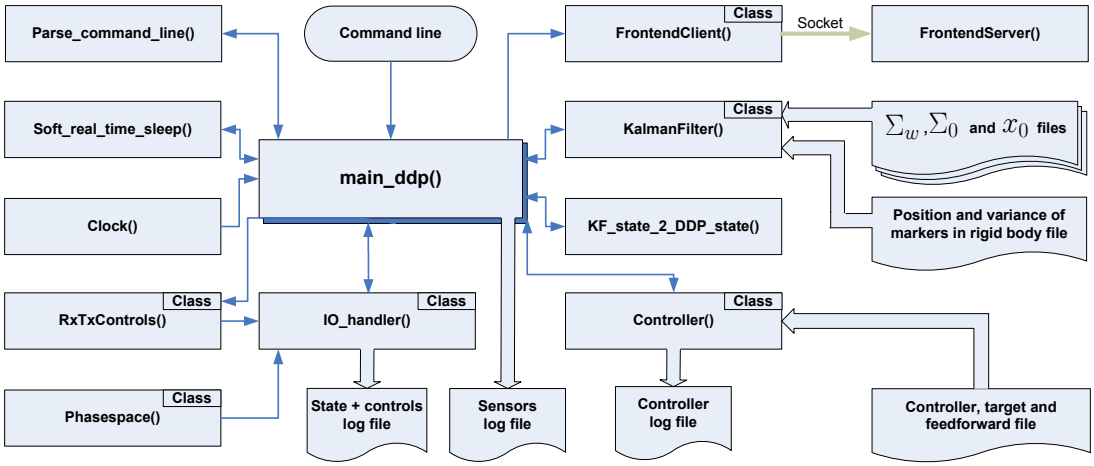


Figure 6.1: Overview of the primary functions and objects of the C++ software. The arrows indicates that a function is called and which directions the communication occur

As can be seen in the bottom of Figure 6.1 there are three files used for logging of the sensor data, controller data and the state and controls. Gathering of new data is done through the IO-handler which provides a way of performing centralized logging of all the raw sensor data from the `Phasespace()` and `RxTxControls()`. This way all the code can be run off line by using an existing sensor file as inputs. This gives the possibility of rerunning an experiment offline, and to change the settings of the Kalman filter before rerunning the code.

As illustrated the Kalman filter reads the settings from four different files, when constructed, such that the filter can be tuned without recompiling the code. And as already described the controller reads the precomputed target and feedback matrices from another file.

The main parts, `IO-handler()`, `KalmanFilter()`, `FrontendClient()`, `RxTxControls()`, `Phasespace()` and `Controller()` are implemented as C++ classes. All the classes are then located in its own

folder and each of the class have a simple example file associated with it which shows how that specific class works in the most simple version. This is a good setup since it is possible to test each of the classes alone. In Figure 6.2 a simplified flowchart is illustrated of the main program.

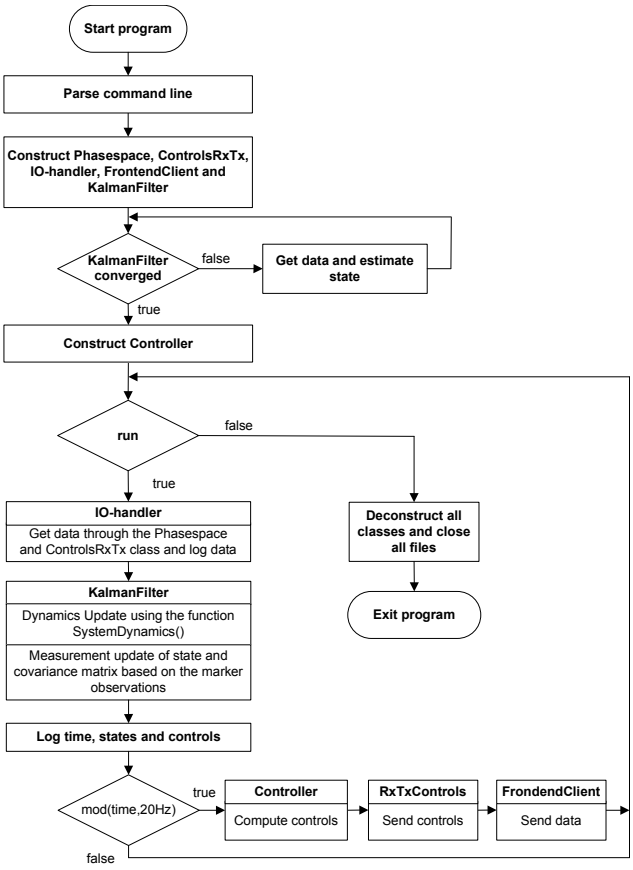


Figure 6.2: Simplified flowchart of main loop

When the program is started any command line arguments are parsed and saved in a struct for easy further use. The `Phasespace()`, `ControlsRxTx()`, `IO-handler()`, `FrontendClient()` and `KalmanFilter()` classes are then initialized. The Kalman Filter is then run until the state, x_t, and covariance matrix, P_t, have converged. Where the converged state is what throughout this report have been denoted, x_0. The `Controller()` class are then constructed with the file containing the precomputed controller and the initial state, x_0, which is used to transform the state and target.

Any new data are then collected and logged by the IO-handler using the `ControlsRxTx()` and `Phasespace()` class. The `KalmanKilter()` is then used to estimate the state which is also logged to a separate file along with the time and the controls. To detect when the buddy switch is pulled (autonomous) the gear is added as an extra channel for the transmitter. This way depending of the gear switch the value of that channel will be 500 or 1500. Therefore by sending out 1000 it is possible to detect when the buddy switch gets pulled indicating autonomous mode.

As described in Chapter 5 about the controller it is important that the feedback controls are computed at the same frequency which the controller are computed for. Therefore, the function `soft_real_time_sleep()` has been implemented to ensure that that control algorithm runs with a specified time interval T. The timer is implemented using the function `usleep()` and is described in detail and tested in Appendix D.

At 20Hz the controls are then computed and sent out to the car using the `RxTxControls()` class. The cars current state and controls are then sent to the `FrontendServer()` through the `FrontendClient()` to give a 3D visualization of the maneuver, see Figure 6.3. Also the controls and the number of active markers are sent to the frontend server along with a flag indicating if the system runs autonomous. To see how well the car follows the target, the target is also sent to the frontend server along with the feedforward controls. The frontend server has been developed as a separate program functioning as a client/server setup over the socket. The server has been written such that it listens for new connections from clients. And if a client connects the server then forks a new thread to handle that client, and the main program returns to listen for new clients. This way the server can handle multiple clients simultaneously. The program are also written in C++ by using the open source OpenGL utility toolkit Glut.

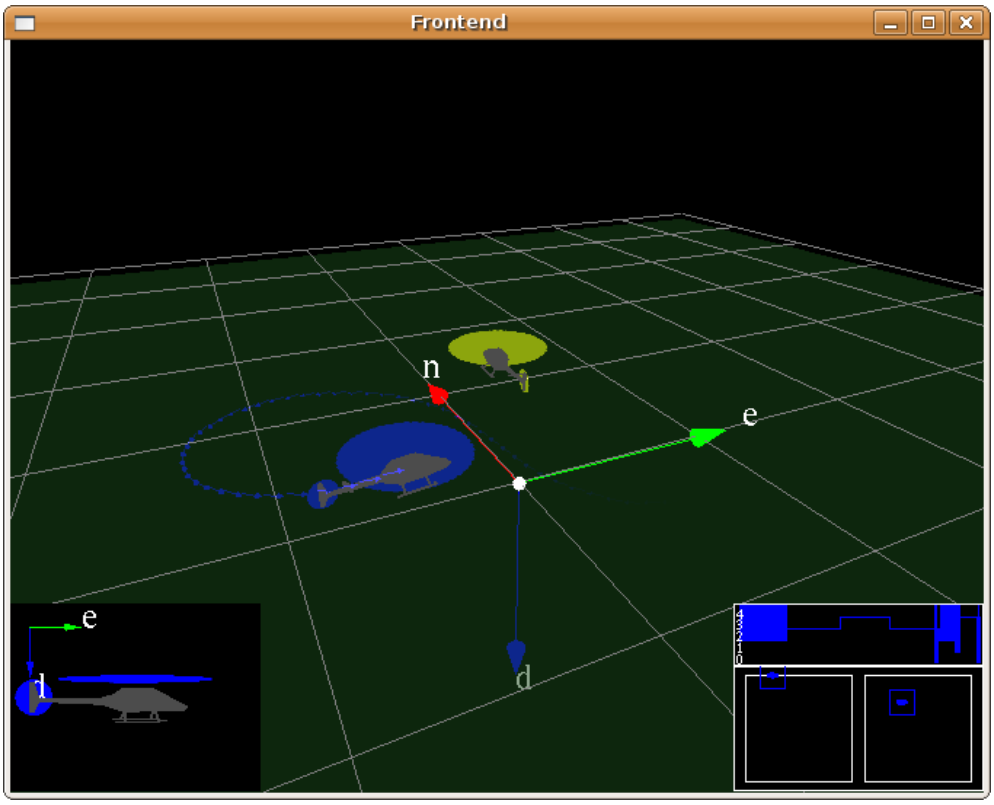


Figure 6.3: Still image of the frontend illustrating a helicopter (blue) and its target (yellow). In the bottom left corner the attitude of the helicopter is illustrated in a subwindow and in the bottom right window the pin positions of the helicopter are illustrated. In the graph on top of the controls subwindow the number of active markers are illustrated over time. If two or less markers are active the space above the graph gets filled out to indicate that the Kalman filter needs more active markers to do an state estimation update.

As described before any command line arguments are parsed through the function `parse_command_line()` function. This gives the following syntax options which are printed with the help argument `-h` or any unrecognized arguments.

```
Syntax is:  ./car_ddp [-option]
options:

I/O handler:
 -f   <FILENAME>       uses sensor data loaded from FILENAME
 -r                    read data real time from sensor file
 -b                    get data from Phasespace as blocking calls
 -br  <BREAK_COUNTER>  break main loop when counter reaches BREAK_COUNTER

KalmanFilter:
 -i                    do NOT init Kalman filter
 -dt  <VALUE>          run Kalman filter with timeinterval VALUE in seconds
 -x0  <FILENAME>       init state vector, x_0, from FILENAME
 -s0  <FILENAME>       init covariance matrix, P_0, from FILENAME
 -sw  <FILENAME>       init sigma_w from FILENAME
 -m   <FILENAME>       read info about markers in rigid body from FILENAME

Controller:
 -tr                   Do NOT transform state and target accordingly to init of KF

FrontEnd:
 -v                    use front end to visualize on Heli2 through port 9000
 -v_s  <IP> <PORT>     use front end to visualize on IP through PORT
 -v_id <ID>            initialize the model to ID
 -v_ty <TYPE>          draw the object as a TYPE ('c'=car, 'h'=gaui, 'q'=quad)
 -v_c  <COLOR>         draw the object in the COLOR (Matlab char, 'b'=blue, etc.)
 -v_tr <NUM_TRAILS>    leave NUM_TRAILS trails behind object
 -v_i  <INIT_VIEW>     init the camera view to INIT_VIEW (0=not, 1=2d, 2=3d)
 -v_1  <PRIMARY_ID>    track the PRIMARY_ID in the subwindows
 -v_2  <SECONDARY_ID>  track the SECONDARY_ID in the subwindows
 -v_a                  activate the attitude subwindow
 -v_m                  activate the number of active markers subwindow
 -v_c                  activate the controls subwindow
```

Using the command line to set different options is a very robust way of using the software since a recompilation can be avoided.

Chapter 7

Learning Parameterized Maneuvers

In Chapter 5 a controller where presented which allowed for controlling the car to follow a single demonstration. But building a maneuver library directly based upon expert demonstrations would require collecting a set of demonstrations for each maneuver. Instead an interpolation-based algorithm and probabilistic model-based algorithm (which build upon [Coates et al., 2008]) which, rather than learning a discrete set of maneuvers, make more efficient use of expert demonstrations by learning *parameterized maneuvers*, which continuously index into a certain maneuver (for example a right turn) based upon a set of demonstrations spanning the range of executions of that maneuver. The approach presented here consists of three main steps,

1. **Learning parameterized maneuvers:**
   Given a set of similar trajectories, learn to generate parameterized maneuvers. The parameterization could but need not be the start and end states

2. **Sequencing parameterized maneuvers:**
   Given a set of way-points or other partial specification of a long trajectory, generate the complete state trajectory sequence by sequencing together (shorter) parameterized maneuvers

3. **Automatic extraction of demonstrations**
   Rather than collecting demonstrations for each parameterized maneuver, collect a large set of "random" demonstrations and automatically picks out similar stretches and then uses them when learning a parameterized maneuver

7.1 Learning Parameterized Maneuvers

It is assumed that M demonstration trajectories of length T are provided. Since the demonstration trajectories are considered relatively short, each demonstration are first uniformly time warped to have a normalized length of T. Each trajectory is a sequence of states, s_t^k, control inputs, u_t^k, and model biases β_t^k composed into a single state vector:

$$y_t^k = \begin{bmatrix} s_t^k \\ u_t^k \\ \beta_t^k \end{bmatrix}, \text{ for } t = 1..T, k = 1..M. \tag{7.1}$$

The goal is to estimate a target trajectory of length T, denoted similarly:

$$z_t = \begin{bmatrix} s_t^\star \\ u_t^\star \\ \beta_t^\star \end{bmatrix}, \text{ for } t = 1..T. \tag{7.2}$$

For simplicity the following notation will often be used: $\mathbf{Y} = \{y_t^k \mid t = 1..T, k = 1..M\}$, $\mathbf{Z} = \{z_t \mid t = 1..T\}$, and similarly for other indexed variables.

To do the parameterization some property of the target trajectory is given, this could be the start and end state or a partial specification thereof. The target trajectory's start and end state will be denoted by z_1^* and z_T^* respectively.

7.1.1 Approach 1: Convex Weighting

Since it is interested to find interpolated versions of the demonstrations, a natural approach is to use a convex weighting of the demos such that this convex combination hits both target start and end states. The convex weights could change throughout the trajectory, and the following convex formulation is proposed,

$$\begin{aligned}
\min_{x_1,x_T} \quad & (z_1^* - z_1)^\top \Lambda_1 (z_1^* - z_1) + (z_T^* - z_T)^\top \Lambda_T (z_T^* - z_T) \\
& + \sum_{k=1}^{M} (x_1)_k \log(x_1)_k + \sum_{k=1}^{M} (x_T)_k \log(x_T)_k \\
s.t. \quad & z_1 = [y_1 \dots y_M] x_1 \\
& z_T = [y_1 \dots y_M] x_T \\
& 0 \le x_1 \le 1 \\
& 0 \le x_T \le 1 \\
& \sum_{k=1}^{M} (x_1)_k = 1 \\
& \sum_{k=1}^{M} (x_T)_k = 1
\end{aligned}$$

The solution x_1 provides the convex weighting for time 1, and x_T provides the convex weighting for the end state. To obtain the intermediate convex weightings a linearly interpolate between x_1 and x_T is performed.

The negative entropy terms $\sum_i (x_1)_i \log(x_1)_i + \sum_i (x_T)_i \log(x_T)_i$ in the objective function ensure that the optimization does not put too much weight on a very small number of trajectories. Without the entropy regularization this happens easily and especially the model bias terms learned for the planned trajectory tend to be too noisy.

If the trajectories are kept short enough, this linear combination can be a good approximation to a true, dynamically feasible trajectory.

7.1.2 Approach 2: A Probabilistic Graphical Model Approach

While for very short trajectories the convex weighting approach can work quite well, it completely ignores the dynamics of the system and might generate trajectories which are not physically feasible.

To incorporate the dynamics model into the planning procedure, the probabilistic model proposed in [Coates et al., 2008] is adapted to being able to plan a new trajectory, rather than simply generating a planned target trajectory which best captures the demonstrator's intent under the assumption all demonstration were executed with the same intent.

The generative model is a standard Hidden Markov Model (HMM) and uses the following dynamics model,

$$z_{t+1} = f(z_t) + \omega_t^{(z)}, \qquad \omega_t^{(z)} \sim \mathcal{N}(0, \Sigma^{(z)}). \tag{7.3}$$

The generative model represents each demonstration as a set of independent "observations" of a hidden, to-be-planned trajectory $\mathbf{Z}$. Specifically, the model assumes,

$$y_t^k = z_t + \omega_t^{(y)}, \qquad \omega_t^{(y)} \sim \mathcal{N}(0, \Sigma^{(y)}). \tag{7.4}$$

In addition, there are added additional observations characterizing the target trajectory: the start and end state. The graphical model described are illustrated in Figure 7.1

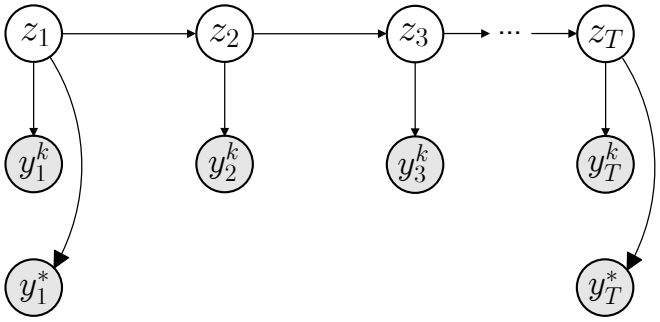


Figure 7.1: Illustration of probabilistic graphical model used to plan a parameterized trajectory. The gray circles represent observations and the arrows represent correlations between the states

In a typical setting between five to ten demonstrations are used. Some of these will more closely match the target start and end states and hence they get assigned a noise model which assumes they deviate less from the planned target trajectory, and hence the inference for the planned target trajectory will more heavily rely on these closer demonstrations.

To find the planned trajectory, an extended Kalman filter/smoother are run over the above described probabilistic model while the EM algorithm described in Section 4.4 are used to estimate the entires in the covariance matrices corresponding to the system dynamics and controls.

Different demonstrations of a parameterized maneuver do not merely differ in independent Gaussian noise, but will often drift around unintentionally. Since these position errors are highly correlated, they are not explained well by the Gaussian noise term in the observation model. To capture such slow drift in the demonstrated trajectories, the latent trajectory's state are augment with a "drift" vector δ_t^k for each time t and each demonstrated trajectory k. The drift is modeled as a zero-mean random walk with a (relatively) small variance. The state observations are now noisy measurements of $z_t + \delta_t^k$ rather than merely z_t. In Figure 7.1 the graphical model described are illustrated with the drift added.

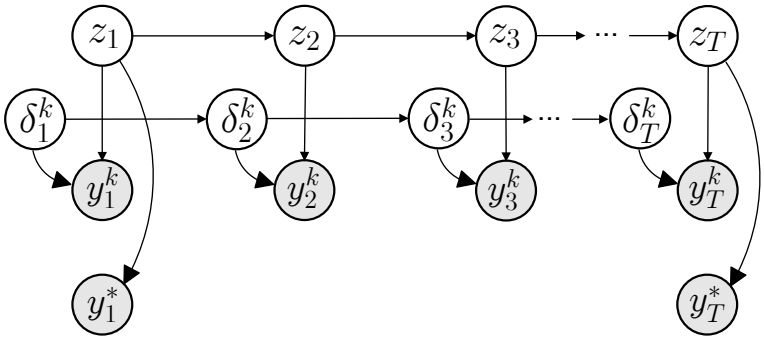


Figure 7.2: Illustration of probabilistic graphical model used to plan a parameterized trajectory, with drift terms added to each demonstration

7.1.3 Tests

To test the different approaches for planning parameterized maneuvers, 10 different demonstrations of a right turn are demonstrated and one of the demonstrations are hold out and tried reconstructed by using its start and end state. The methods used for planning the trajectory are the convex combinations with and without the entropy terms and the probabilistic graphical model. In Figure 7.3 nine demonstrations are plotted along with the leave one out and the planned trajectories from the three different methods.

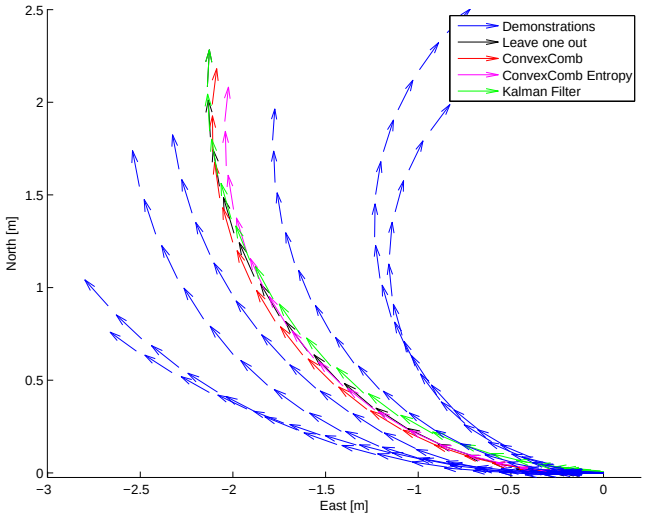


Figure 7.3: Plot of the nine demonstrations along with the leave one out and the planned trajectories from the three different methods

The leave one out verification have been performed on five different trajectories. Each of the planned trajectories have then been executed by using the DDP controller from Chapter 5. The difference between the leave one out trajectory and the executed trajectory have then been calculated as the sum of the squared error of all elements in the state vector (without biases and controls). In Table 7.1 the score of the leave one out validation is listed for 5 different trajectories each executed three times.

Methods	Error when reconstructing 5 demonstrations				
ConvexComb	0.3412	1.5339	3.2470	0.7686	7.4471
	0.4415	1.4566	1.0475	0.7493	5.6761
	0.2567	1.9831	2.0045	1.0429	5.5806
ConvexComb w/ entropy	0.3764	2.8193	0.6015	0.4248	6.0913
	0.3128	1.6912	1.2005	0.4990	7.1451
	0.4601	1.8638	0.8719	0.4057	5.6616
Kalman Filter	1.2561	1.1619	2.1600	1.3862	1.7706
	1.5104	0.9827	2.0753	2.6001	1.7803
	1.2385	1.6371	1.5666	0.8090	1.2209

Table 7.1: Sum of squared error for five different trajectories compared to the planned trajectories executed three times. The planned trajectories are reconstructed from their start and end states

As can be seen in Table 7.1 it is difficult to draw a clear conclusion about the three methods.

7.2 Sequencing Parameterized Maneuvers

Given a set of way-points or other partial specification of a long trajectory, the goal is to generate the complete state trajectory sequence by sequencing together (shorter) parameterized maneuvers. Waypoints are often much easier to specify than entire trajectories, especially so when only being requested to specify a few of the state variables for each of the way-points, such as position and heading. This is illustrated in Figure 7.4(a) where the waypoints used are generated from a synthetic target figure 8 trajectory. The red squares mark the waypoints supplied to the sequencing algorithm.

If the stitches are computed in chronological order, additional state information from the end of the last stitch can be used to give a more smooth transition of the part of the state which are not defined. Therefore, to ensure a smooth transition over the stitch boundaries, the start state z_1^* of a stitch are augmented with a overlap from the previous stitch. Both in the convex combination approach and in the probabilistic model approach, these are readily incorporated.

For each of the intervals between two waypoints, there are a collection of trajectories which are used to interpolate between those two waypoints (call this interval and the associated trajectory a stitch). Stitching makes use of the interpolation methods from the previous section to create a single unified trajectory. In Figure 7.4(b) the 7 nearest demonstrations which are interpolate from for each stitch are plotted along with the final stitched trajectory.

In Figure 7.4(c) an open loop simulation is performed using the dynamics model from Chapter 3.3 and the entire stitched trajectory. And in Figure 7.4(c) an closed loop simulation is done using a the DDP controller where it can be seen that the car can drive the planned trajectory.

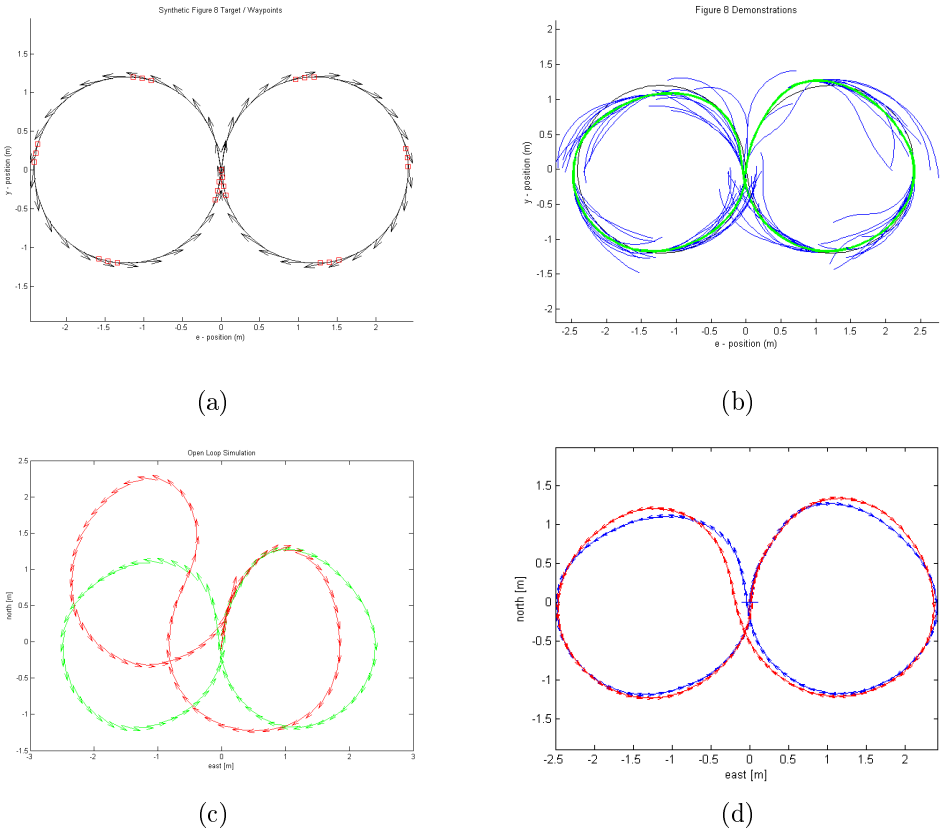


Figure 7.4: (a) The synthetic target figure 8 trajectory. Red squares mark the waypoints supplied to the stitching algorithm. (b) Stitched target figure 8 from demonstrations, using n=7 nearest neighbor trajectories. The blue lines represent the nearest neighbor demonstration trajectories which are interpolate from; the green line is the final stitched trajectory. (c) Open loop simulation of target trajectory, using the naive car dynamics model. The green line is the target, while the red is the simulation (d) Closed loop simulation of target trajectory using learned DDP controller. The blue line is the target trajectory, while the red is the closed loop simulation

In Figures 7.5 - 7.9 the entire state are plotted over time for the 7 demonstrations used to plan the 8 trajectory from eight stitches. The borders between the stitches are illustrated with yellow dashed vertical lines. In the plots the demonstrations used for each stitch are plotted along with the planned trajectory using the convex combination method with entropy.

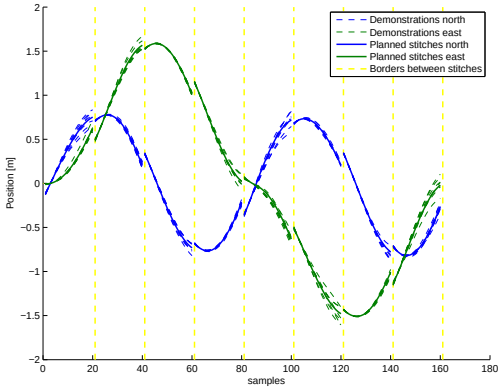


Figure 7.5: Position ne for each of the demonstrations and planned trajectory put together from eight stitches

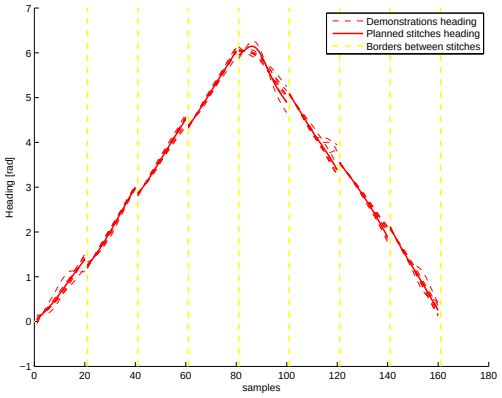


Figure 7.6: Heading θ for each of the demonstrations and planned trajectory put together from eight stitches

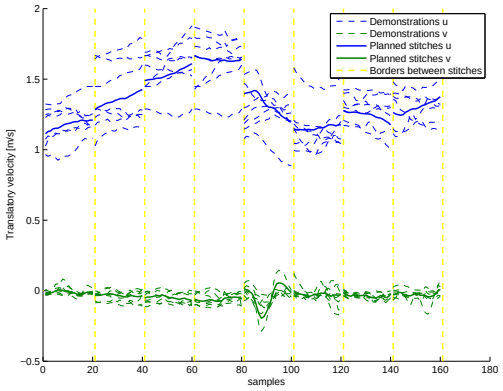


Figure 7.7: Translatory velocity uv for each of the demonstrations and planned trajectory put together from eight stitches

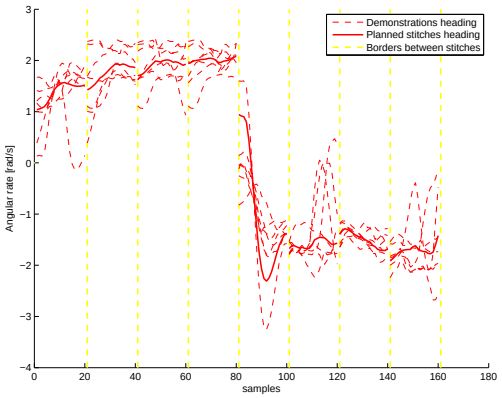


Figure 7.8: Angular rate $\dot{\theta}$ for each of the demonstrations and planned trajectory put together from eight stitches

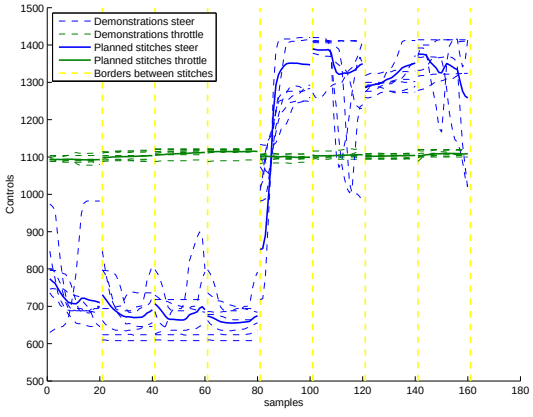


Figure 7.9: Controls, steer and throttle, for each of the demonstrations and planned trajectory put together from eight stitches

As can be seen there is a good match in the overlap between the stitches for the position and heading. But for the velocities and controls the match are not always that good, which can be seen in the open and closed loop performance.

This is because the optimization problem does not account for what happens in the demonstrations after the way-point. As a waypoint typically only contains partial state information, the non-specified state variables and, indeed, possibly unmodeled (hidden) state variables do not match up closely at the end waypoint.

In Figure 7.10 and 7.11 the biases are plotted for the demonstrations and the planned 8-trajectory. As it is evident the overlap of the biases are very smooth as intended.

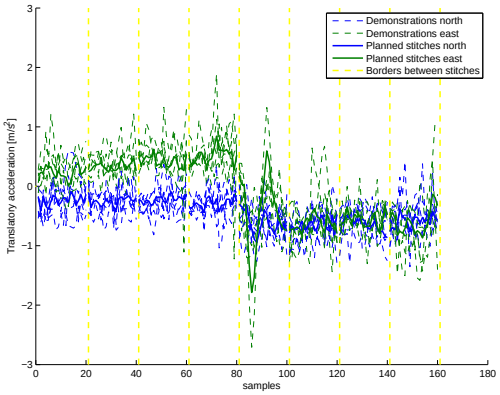


Figure 7.10: Bias terms for the accelerations $\dot{uv}$ for each of the demonstrations and planned trajectory put together from eight stitches

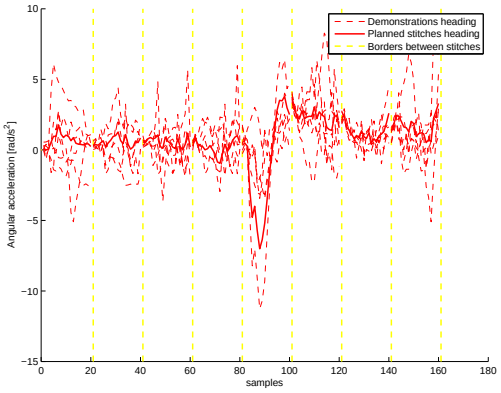


Figure 7.11: Bias terms for the accelerations $\ddot{\theta}$ for each of the demonstrations and planned trajectory put together from eight stitches

7.3 Automatic Extraction of Demonstrations

Thus far the details of obtaining good demonstration trajectories have been glossed over. To simplify the data collection the car is driven around to collect a large dataset. The large corpus of data where gathered by driving the of generic turns, circles, and loops by driving the car for an extended period of time (around 10 minutes). Notably, we do not explicitly produce any figure 8's in this training data.

An algorithm is then presented which can extract stretches from the large dataset which are good demonstrations based on the start and end state. Again the stitches are computed in chronological order, to give additional state information from the end of the last stitch.

When finding good demonstrations for a stitch all the data is searched through and a score is computed for the start state with the augmented overlap and for the end state. To compute this score the data in the large data corpus are transformed to match the start state as described in Section 4.6 on page 35.

The first crude solution where to use weighted least squares to compute the distance between the waypoints and the trajectories seen in the data. I.e., the score of a demonstration trajectory seen in the data corresponds to the result of the optimization problem if that demonstration trajectory were the only demonstration.

In Figure 7.12 the position of the large data set used are plotted, along with the 7 best demonstrations for a specific stitch. In Figure 7.13 the same 7 demonstrations are extracted from the data and transformed to match the starting state.

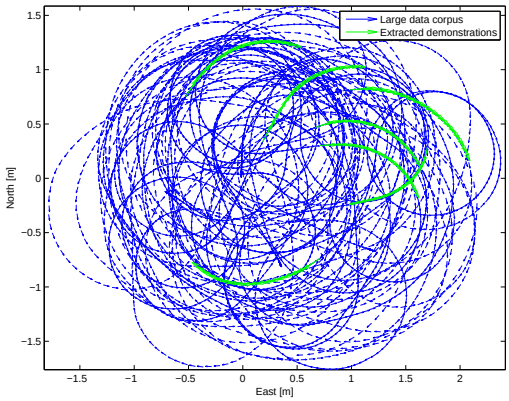


Figure 7.12: Large corpus of data with the 7 stretches that had the best score for a specific stitch

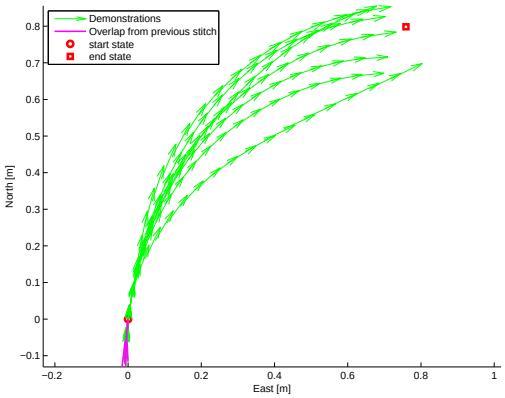


Figure 7.13: Same 7 demonstrations as plotted in Figure 7.12 here plotted after they have been transformed based on the starting state

7.3.1 Time Search

Because the stitches focus on short trajectories, the time scales at which maneuvers occur should be relatively stable across trajectories. Thus one can simply stretch or shrink demonstration trajectories to the proper length (the time distance between waypoints). We run a simple brute-force search over time indexes, allowing each time index to vary by ± 4 ticks either way, and take the indices which minimize the weighted least-squares error between the current stitch with overlap and the desired time indices for the end state.

7.4 Summary

It is perhaps somewhat surprising that the convex combination methods, while crude, perform reasonably when compared against the Kalman-filtering-based approaches. Simple interpolation methods do not respect the dynamics of the problem. However, when interpolating locally between past demonstrations the result cannot deviate too far from dynamically feasible regimes. Short trajectories have the benefit that simple stretching and shrinking are sufficient to obtain a satisfactory alignment between demonstrations.

The highest return techniques are those related to demonstration selection. Adding the maximum entropy weighting condition improves the interpolation. It is important to smooth out the noise in each demonstration trajectory using the entropy criterion, but one must be sure that the trajectories used to are similar enough that it makes sense to do such interpolation.

Chapter 8

Conclusion

This project has concerned with the problem of planning arbitrary trajectories, by stitching together smaller pieces of different parameterized maneuvers found by using interpolation-based algorithms and probabilistic model-based algorithms.

As a test platform a Drift-R Sedan 4WD 1/10 RC car has been used. The car has been modeled based on the cars steady state values and time constants. After having optimized the time constants from recorded data, it have been verified that the model captures the dynamics of the car. To optimize for speed the simulation model have been discretized which reduced the computation time by 10-11 times without a significant loss in precision. Even though a good dynamics model have been presented, model inaccuracies will still occur. To correct for this time dependent model biases have been added to the model which significantly improves the performance when used together with the controller.

A Kalman filter has been designed and implemented for real-time estimation of the cars state. Also a smoother and an EM algorithm have been presented which are used to tune the Kalman filter and in the probabilistic model-based algorithm for planning parameterized maneuvers.

A controller has been implemented using Differential Dynamic Programming (DDP) which are an extension to the (LQR). This controller is used together with the model and a given target trajectory to precompute a series of feedback matrices. Simulations and tests showed that the DDP where capable of making the car follow a specified target trajectory.

To plan the parameterized maneuvers three different approaches have been presented. The first two are to calculate the hidden planned states based on a convex weighting of the start and end state, and then to do a linear interpolate between the solution for the start and end state. This method have been presented with and without an additional entropy term to penalize for picking the same trajectory. The last method presented is a probabilistic model-based algorithm based on a Kalman filter/smoother and an EM algorithm. All methods where capable of planning a trajectory from a start state to a end state. And using the DDP controller the planned trajectories where executed.

An algorithm where then presented for planning longer trajectories by stitching together smaller parameterized maneuvers. To get good data for planning the individual stitches an algorithm where used to searching through a large corpus of data and extracting good demonstrations.

Chapter 9

Future Perspectives

A model based on steady state values has been presented which proved to model the dynamics of the car very well. But for different driving styles the model might have to be updated which takes some time since steady state values have to be recorded for different combinations of the inputs. This is especially a problem for steering straight and full throttle since a very long stretch is needed for the car to reach steady state. Therefore, it would be useful to develop an algorithm which could switch between exploration and exploitation to make the system explore the relevant parts of the state space autonomously in a safe way, and estimate the steady state values used in the table. This could be done by giving a very simple model of the system and then have the algorithm explore states close by and exploit the new data collected to improve the model. The improved model would then be valid in a slightly larger state space allowing the algorithm to explore more states.

Also a different model based on recorded data could be used. This could be a locally weighted nearest neighbor model which would use recorded data to estimate the next state.

As described in the in Chapter 5 the DDP controller only works well when the specified trajectory is followed closely, since it is a series of linear feedback matrices which are precomputed. Hence, if the car at some point deviates to much from the specified trajectory the performance of the DDP might not be sufficient to get the car back on track. Therefore, the DDP could be extended with a DDP controller with receding horizon, which at every time instance performs a forward simulate by solving the DDP into the future from the specific state. By doing so the controller would be able to efficiently correct errors even though the car deviates allot from the specified trajectory.

One problem observed when stitching together a larger trajectory where that while the velocities and control inputs wher bound to be reasonable near the beginning of a stitch, they tended to fluctuate as they get closer to the endpoint. To address this issue "lookahead waypoints" could be added when performing the least-squares match for selecting trajectories. The idea would be to look ahead and add penalty terms for the squared error between the future waypoint and the state of the demonstration trajectory at that future waypoint.

In addition to incorporating a longer state sequence for the start state constraint/observation, the probabilistic model approach can automatically provide a dynamically reasonable interpolation for individual stitches. I.e., the Kalman filter/smoother could be run over the entire trajectory rather than over individual stitches.

When searching for good demonstrations for a specific start and end point a time search is performed on the data demonstrations. However, shrinking or stretching the data will affect the correlation in the states since the velocity then will no longer correspond to the position, making the trajectory non-feasible. Instead the time search could be done over the waypoint in time.

Since only a small subset of the state-space are used at each waypoint (i.e. position and orientation information), the demonstration trajectories which are selected from the data may correspond to very different maneuvers. For example, imagine a maneuver that attempts to avoid an obstacle directly in front of the car. One might avoid the obstacle either to the left or to the right, ending up on the other side. Such demonstrations may have excellent least-squares properties, yet interpolating between them would result in a highly non-optimal collision. Therefore, it is wanted that the demonstration trajectories used in the stitching belong to the same class of maneuver, i.e. either all turn-left paths or turn-right paths but not both.
This problem could be dealt with by clustering the demonstration trajectories after the weighted-least-squares time search. Hierarchical agglomerative clustering could be used for this weighting.

When starting this project the ultimate goal where to perform parameterized drifting maneuvers, where both the front and back wheels are sliding sideways. A lot of time where spend by putting tape on the wheels to minimize the friction and update the dynamics model by recording new steady state values. The initial idea for drifting where to have the car drift around in a circle continuously. But it where noticed that it where very difficult to get the car into the drifting state in closed loop, since the car has to go faster before it starts to slide resulting in the forward velocity to drop significantly. But the model do not know that it need to increase the forward velocity in order to later make it drop. The idea used where to have a small initial trajectory which in open loop drove the car into a state of steady state drifting (steady state forward and sideways velocities), and then have the controller take over and keep the car drifting around in the specified trajectory. However, this where never fully achieved even though it came close.

Bibliography

[Abbeel and Ng, 2004] Abbeel, P. and Ng, A. Y. (2004). Apprenticeship learning via inverse reinforcement learning. In *Proc. ICML*.

[Abbeel et al., 2006a] Abbeel, P., Quigley, M., and Ng, A. Y. (2006a). Using inaccurate models in reinforcement learning. In *Proc. ICML*.

[Abbeel et al., 2006b] Abbeel, P., Quigley, M., and Ng, A. Y. (2006b). Using inaccurate models in reinforcement learning. ICML 2006.

[An et al., 1988] An, C. H., Atkeson, C. G., and Hollerbach, J. M. (1988). *Model-Based Control of a Robot Manipulator*. MIT Press.

[Anderson and Moore, 1989] Anderson, B. D. O. and Moore, J. B. (1989). *Optimal Control: Linear Quadratic Methods*. Prentice-Hall.

[Arikan and Forsyth, 2002] Arikan, O. and Forsyth, D. (2002). Interactive motion generation from examples.

[Atkeson and Schaal, 1997] Atkeson, C. and Schaal, S. (1997). Robot learning from demonstration. In *Proc. ICML*.

[Atmel, 2008] Atmel (2008). http://www.datasheetcatalog.org/datasheet/atmel/2467s.pdf. Internet 11/24-08 time: 09.15 pm.

[Bak, 2002] Bak, T. (2002). *Lecture Notes - Modeling of Mechanical Systems*. Aalborg University, first edition.

[Calinon et al., 2007] Calinon, S., Guenter, F., and Billard, A. (2007). On learning, representing and generalizing a task in a humanoid robot. In *IEEE Trans. on Systems, Man and Cybernetics, Part B*, volume 37.

[Coates et al., 2008] Coates, A., Abbeel, P., and Ng, A. Y. (2008). Learning for control from multiple demonstrations. ICML 2008. http://heli.stanford.edu/icml2008.

[Fang and Pollard, 2003] Fang, A. C. and Pollard, N. S. (2003). Efficient synthesis of physically valid human motion.

[Guennebaud and Jacob, 2008] Guennebaud, G. and Jacob, B. (2008). http://eigen.tuxfamily.org. Internet 10/24-08 time: 6.47 pm.

[Jacobson and Mayne, 1970] Jacobson, D. H. and Mayne, D. Q. (1970). *Differential Dynamic Programming*. American Elsevier Publishing Company, Inc. ISBN: 44-00070-4.

[Jordan, 2003] Jordan, M. I. (2003). *An Introduction to Probabilistic Graphical Models*. Copy Central University of California at Berkeley. Course reader for Statistical Learning Theory for Graphical Models, EECS 281A.

[Lay, 2003] Lay, D. C. (2003). *Linear Algebra and its Applications*. Addison Wesley, third edition. ISBN: 0-321-14992-0.

[Lee et al., 2002] Lee, J., Chai, J., Reitsma, P., Hodgins, J., and Pollard, N. (2002). Interactive control of avatars animated with human motion data. In *Proc. SIGGRAPH*.

[Losi, 2008] Losi, T. (2008). http://www.losi.com/products/features.aspx?prodid=losb0289. Internet 11/24-08 time: 01.13 am.

[Neu and Szepesvari, 2007] Neu, G. and Szepesvari, C. (2007). Apprenticeship learning using inverse reinforcement learning and gradient methods. In *Proc. UAI*.

[Ng and Russell, 2000] Ng, A. Y. and Russell, S. (2000). Algorithms for inverse reinforcement learning. In *Proc. ICML*.

[Phasespace, 2008] Phasespace (2008). http://www.phasespace.com/. Internet 11/24-08 time: 03.15 pm.

[Ramachandran and Amir, 2007] Ramachandran, D. and Amir, E. (2007). Bayesian inverse reinforcement learning. In *Proc. IJCAI*.

[Ratliff et al., 2006] Ratliff, N., Bagnell, J., and Zinkevich, M. (2006). Maximum margin planning. In *Proc. ICML*.

[Rose et al., 1996] Rose, C., Guenter, B., Bodenheimer, B., and Cohen, M. F. (1996). Efficient generation of motion transitions using spacetime constraints. *Computer Graphics*.

[Ross, 1996] Ross, S. M. (1996). *Stochastic Processes*. John Wiley & Sons, Inc, second edition. ISBN: 0-471-12062-6.

[Spektrum, 2008] Spektrum (2008). http://www.spektrumrc.com/. Internet 11/24-08 time: 01.09 am.

[Syed and Schapire, 2008] Syed, U. and Schapire, R. E. (2008). A game-theoretic approach to apprenticeship learning. In *NIPS 20*.

[Unuma et al., 1995] Unuma, M., Anjyo, K., and Takeuchi, R. (1995). Fourier principles for emotion-based human figure animation. *Computer Graphics*.

[Venom, 2008] Venom (2008). http://www.venom-racing.com/. Internet 11/24-08 time: 03.13 pm.

[wikipedia, 2008] wikipedia (2008). http://en.wikipedia.org/. Internet 12/14-08 time: 1.20 am.

[Wiley and Hahn, 1997] Wiley, D. J. and Hahn, J. K. (1997). Interpolation synthesis for articulated figure motion. *Virtual Reality Annual International Symposium*.

Part I

Appendix

Appendix A

Extrinsic Calibration of Phasespace System

When using the Phasespace system the coordinates of the markers are given in the Phasespace frame, $\{\mathbb{P}\}$, defined by camera 1. However, to ease the use of these coordinates they are transformed into coordinates in the inertial frame, $\{\mathbb{E}\}$, defined in the lab. The transformation between these two frames consists of three things: a scaler, a rotation and an offset. The scaling factor is to ensure that the coordinates in $\{\mathbb{E}\}$ is in metrics. When setting up the Phasespace system two API's are available for the transformation, one which sets the scaling, and another which sets a pose consisting of a 3D-offset and a quaternion.

A.1 Initialization of Parameters

To define the position and orientation of the inertial frame relative to the Phasespace frame, three LED's are placed in the three known positions in the earth inertial frame $\{\mathbb{E}\}$ which is ${}^{\mathbb{E}}[0,0,0]^T$, ${}^{\mathbb{E}}[1,0,0]^T$ and ${}^{\mathbb{E}}[0,1,0]^T$, corresponding to the three points ${}^{\mathbb{P}}origin$, ${}^{\mathbb{P}}xyz_1$ and ${}^{\mathbb{P}}xyz_2$ respectively in the Phasespace frame.

The scaling factor between the two frames is found by dividing 1 with the mean distance from the origin to the two other coordinates, since this is suppose to be one meter. All three coordinates in the Phasespace frame is then scaled using this factor, and the scaled origin is subtracted from all coordinates in the Phasespace frame. The scaled origin is then the translatory offset which is the first three values of the transformation pose, though they first have to be rotated. The rotation matrix $\hat{R}$ between the two frames can be found by Equation (A.1)

$$ {}^{\mathbb{E}}\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \hat{R} \; {}^{\mathbb{P}}\begin{bmatrix} x \\ y \\ z \end{bmatrix} \tag{A.1} $$

Given the two coordinates which is used to calculate the rotation matrix Equation (A.1) becomes,

$$
{}^{\mathbb{E}}\begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{bmatrix} = \hat{R} \quad {}^{\mathbb{P}}\begin{bmatrix} x_1 & x_1 \\ y_1 & y_2 \\ z_1 & z_2 \end{bmatrix} \tag{A.2}
$$

$\hat{R}$ is then estimated using Equation (A.2). However, the elements in $\hat{R}$ might satisfy Equation (A.2) but for it to be a rotation matrix it has to be an orthogonal matrix meaning that $\hat{R}\hat{R}^T = I$ [wikipedia, 2008]. Therefore, $\hat{R}$ is projected onto the nearest rotation matrix R, which is done in Equation (A.3), by using the singular value decomposition of $\hat{R} = U\Sigma V^T$, and the fact that both U and V are orthogonal matrices.

$$
\begin{aligned}
\min_{R\,:\,R^TR = I} \left\| R - \hat{R} \right\|_2^2 &= \left\| R - U\Sigma V^T \right\|_2^2 \\
&= \left\| U^T R V - \Sigma \right\|_2^2
\end{aligned} \tag{A.3}
$$

This means that for $\hat{R}$ to be a rotation matrix, Σ has to be the identity matrix, which means that the nearest rotation matrix R can be found by,

$$
\begin{aligned}
R &= UIV^T \\
&= UV^T
\end{aligned} \tag{A.4}
$$

This rotation matrix could be used to first rotate the scaled offset and then converted into a quaternion which completes the transformation pose needed to setup the Phasespace system. However, to optimize it a line search is first performed on the rotation matrix to minimize the error.

A.2 Line Search to Optimize Rotation Matrix

The purpose is to find the orthogonal rotation matrix R that minimize the the squared error,

$$
\min_{R\,:\,R^TR = I} \sum_{i=1}^{2} \left\| {}^{\mathbb{E}}xyz_i - R \cdot {}^{\mathbb{P}}xyz_i \right\|_2^2 \tag{A.5}
$$

To optimize the rotation matrix R by doing a line search means to calculate the squared error denoted the score, and then change one of the elements in R and evaluate if the new matrix R has improved the score. However, to avoid having to evaluating each of the nine elements in the rotation matrix, it is first transformed to the axis of rotation since this describes the rotation matrix using only three elements. It could also have been transformed to a quaternion, but that would have given four elements to evaluate on. To do this transformation some basic theory about skew symmetric matrix is needed.

A.2.1 Skew Symmetric Matrices

Given the two vectors $a = [a_1, a_2, a_3]^T$ and $b = [b_1, b_2, b_3]^T$ then the vector product is defined as,

$$a \times b = \begin{pmatrix} a_2b_3 - b_2a_3 \\ -a_1b_3 - b_3a_1 \\ a_1b_2 - a_2b_1 \end{pmatrix} \tag{A.6}$$

Since this is a linear transformation in a and b, it can be rewritten as,

$$\begin{aligned} a \times b =& \begin{bmatrix} 0 & -a_3 & a_2 \\ a_3 & 0 & -a_1 \\ -a_2 & a_1 & 0 \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} && (A.7) \\ =& A \cdot b \\ =& \hat{a} \cdot b && (A.8) \end{aligned}$$

where:
$\hat{a} = (a_1, a_2, a_3)$ is a simplified notation for the skew symmetric matrix A, $(A^T = -A)$

This means that the vector product of two vectors can be written as the product of a skew-symmetric matrix $(A^T = -A)$ and a vector. This can be utilized when the position of a rotating object is defined by the following deferential equation,

$$\begin{aligned} \dot{q}(t) =& \omega \times q(t) \\ \dot{q}(t) =& \hat{\omega} \cdot q(t) \\ q(t) =& e^{\hat{\omega}} \cdot q(0) \end{aligned} \tag{A.9}$$

From Equation (A.9) it is evident that $e^{\hat{\omega}}$ is a rotation matrix. This means that the rotation matrix R found in Equation (A.4) can be transformed to the skew symmetric matrix represented by $\hat{n}$ and back again using Equation (A.10) and (A.9) respectively.

$$\hat{n} = log(R) \tag{A.10}$$

$$R = e^{\hat{n}} \tag{A.11}$$

A.2.2 Algorithm for Line Search

The algorithm used for the line search is as follows. Initialize three variables containing the step size for each of the elements in $\hat{n}$ and run the following steps, for each of the three elements, until convergence.

1. Step in one direction with the size of that specific step variable
2. Calculate new score and evaluate
3. If step improved score then update score and double that specific step direction and jump to 1

4. Else step in opposite direction
5. Calculate new score and evaluate
6. If step improved score then update score and change sign for that specific step variable and jump to 1
7. Else multiply that specific step variable with .5 and jump to 1

By implementing a varying step size the algorithm is improved for both speed and precision. The algorithm is then run until it has convergence, which is when the norm of the step size is less than some specified value, or until a maximum number of iterations. Figure A.1 shows how the score is improved as the algorithm is run.

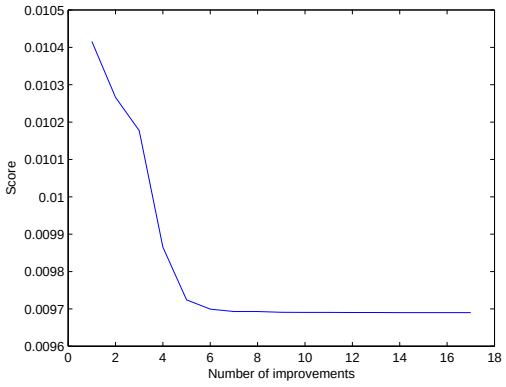


Figure A.1: Improvements of score as the line search algorithm is run. In this specific case the algorithm converged after 17 iterations, which means that the score was improved in every iteration

The evaluation of the score is done in Equation (A.12) by first converting $\hat{n}$ to the rotation matrix R.

$$score = \sum_{i=1}^{2} \left\| {}^{\mathbb{E}}xyz_i - R \cdot {}^{\mathbb{P}}xyz_i \right\|_2^2 \tag{A.12}$$

Matlab Code for Line Search

In the following code *ned* and *xyz* corresponds to ${}^{\mathbb{E}}xyz$ and ${}^{\mathbb{P}}xyz$ respectively.

```
dns = [.01 ; .01 ; .01];
max_dn = .1;
min_dn = .0001;

curr_score = squared_rotation_error(ned, xyz, n);

num_iters = 1000;
for iter = 1:num_iters
    if(sum(abs(dns)) <= 3*min_dn)
        'exit b/c converged'
        break;
    end
    for i=1:3
        test_n = n;
        test_n(i) = n(i) + dns(i);
        test_score = squared_rotation_error(ned,xyz,test_n);
        if( test_score < curr_score)
            n = test_n;
            curr_score = test_score;
            dns(i) = dns(i) * 2;
            if(abs(dns(i)) > max_dn)
                dns(i) = sign(dns(i))*max_dn;
            end
        else
            test_n(i) = n(i) - dns(i);
            test_score = squared_rotation_error(ned,xyz,test_n);
            if(test_score < curr_score)
                n = test_n;
                curr_score = test_score;
                dns(i) = -dns(i);
            else
                dns(i) = .5*dns(i);
                if(abs(dns(i)) < min_dn)
                    dns(i) = sign(dns(i))*min_dn;
                end
            end
        end
    end
end
```

A.3 Implementation

To make the calibration of the Phasespace system more efficient and automated, all the software has been implemented in C++ using the matrix library Eigen2 [Guennebaud and Jacob, 2008].

To ease the use and to improve the precision of the marker position a calibration tool has been produced consisting of a orthogonal triangle where each cathetus is 1 m, and a LED marker placed in each corner of the triangle. Each LED marker is then sampled 2000 times and the mean coordinates is found to mean out the noise. Since the matrix exponential and logarithm does not exist in Eigen2, the four functions in Table A.1 have been implemented. Hence the axis of rotation can be found from a rotation matrix by first calculating the quaternion by means of `matrix2q` and then using the function `axis_rotation_from_quaternion` to find the axis of rotation.

Function	**Description**
`matrix2q`	converts a rotation matrix to a quaternion
`q2matrix`	converts a quaternion to a rotation matrix
`quaternion_from_axis_rotation`	converts the axis of rotation to a quaternion
`axis_rotation_from_quaternion`	converts a quaternion to a the axis of rotation
`pinv`	calculates the pseudo inverse of a matrix using the singular value decomposition

Table A.1: Five functions implemented in C++

To solve Equation (A.2) ($\boldsymbol{b} = \boldsymbol{A} \cdot \boldsymbol{r}$) both the LU method and the pseudo inverse can be used since $\boldsymbol{A}$ is not a $n \times n$ matrix. However since the pseudo inverse is needed in the Kalman update, described in section 4.1 on page 27, the Moore-Penrose Pseudo-Inverse has been implemented in C++.

Appendix B

Kalman Filter

The Kalman filter gives a way to calculate an estimate of the state x_t based on a partial output sequence $(y_0, y_1, ..., y_t)$. This is done by calculate or prediction the probability distribution of x_t given all past measurements of y up to and including the time t, denoted $P(x_t|y_{0:t})$. The probability distribution can then be used to estimate the state at time t. This means that the filter can be used to estimate or track the states in real time. Equation (B.1) and (B.2) are the state space representation of a linear system with process noise w_t and observation noise v_t. Unless other is mentioned the following section is inspired by [Jordan, 2003, Chap. 15]

$$x_{t+1} = Ax_t + Bu_t + w_t \tag{B.1}$$
$$y_t = Cx_t + v_t \tag{B.2}$$

where:
w_t is a Gaussian noise term independent of x_t, $\quad w_t \sim \mathcal{N}(0, \Sigma_w)$
v_t is a Gaussian noise term independent of y_t, $\quad v_t \sim \mathcal{N}(0, \Sigma_v)$

Since the Kalman filter is used for systems assumed to have Gaussian noise, the definition of a Gaussian signal is first defined.

$$x \sim \mathcal{N}(\mu, \Sigma) \Leftrightarrow$$
$$P(x) = \frac{1}{(2\pi)^{n/2}|\Sigma|^{1/2}} e^{-\frac{1}{2}(x-\mu)^T \Sigma (x-\mu)} \tag{B.3}$$

where:
μ is the mean of x
Σ is the covariance of x
n is the dimension of x

The definition of the mean and covariance matrix is given in Equation (B.4) and (B.5) respectively.

$$\mu_x = \hat{x} = \mathbb{E}[x] \tag{B.4}$$
$$\Sigma_{xx} = \mathbb{E}[(x - \mu_x)(x - \mu_x)^T] \tag{B.5}$$

To describe the conditional means and covariance matrix, a simplified notation will be used to emphasize the particular output sequence which it is being conditioned on, thus,

$$\hat{x}_{t|t} \triangleq \mathbb{E}[x_t|y_{o:t}] \tag{B.6}$$
$$P_{t|t} \triangleq \mathbb{E}[(x_t - \hat{x}_{t|t})(x_t - \hat{x}_{t|t})^T|y_{0:t}] \tag{B.7}$$

where:
$y_{0:t}$ is the observations $y_0, y_1, ..., y_t$

First the probability distribution for the next time index is predicted as shown in the prediction step. It is assumed that $\mathbb{P}(x_t|y_{0:t})$ is already calculated, that is, $\hat{x}_{t|t}$ and $P_{t|t}$ is known which are usefully since $x_{t|t} \sim \mathcal{N}(\hat{x}_{t|t}, P_{t|t})$. After the measurement $t+1$ becomes available the predicted distribution is corrected in the correction step.

time update ("Prediction step"):	$P(x_t\|y_{0:t}) \rightarrow P(x_{t+1}\|y_{0:t})$
measurement update ("Correction step"):	$P(x_{t+1}\|y_{0:t}) \rightarrow P(x_{t+1}\|y_{0:t+1})$

Prediction Step
The equations used for the prediction step is now derived, starting with the prediction of the state,

$$\begin{aligned}\hat{x}_{t+1|t} =& \mathbb{E}[x_{t+1}|y_{o:t}] \\ =& \mathbb{E}[Ax_t + Bu_t + w_t|y_{o:t}] \\ =& A\mathbb{E}[x_t|y_{o:t}] + Bu_t + 0 \\ =& A\hat{x}_{t|t} + Bu_t\end{aligned} \tag{B.8}$$

Similar the prediction of the covariance matrix can be found.

$$\begin{aligned}P_{t+1|t} =& \mathbb{E}[(x_{t+1} - \hat{x}_{t+1|t})(x_{t+1} - \hat{x}_{t+1|t})^T|y_{0:t}] \\ =& \mathbb{E}[(Ax_t + Bu_t + w_t - A\hat{x}_{t|t} + Bu_t)(Ax_t + Bu_t + w_t - A\hat{x}_{t|t} - Bu_t)^T|y_{0:t}] \\ =& \mathbb{E}[A(x_t - \hat{x}_{t|t})(x_t - \hat{x}_{t|t})^T A^T + 2w_t^T(x_t - \hat{x}_{t|t})A^T + w_t w_t^T|y_{0:t}] \\ =& A \underbrace{\mathbb{E}[(x_t - \hat{x}_{t|t})(x_t - \hat{x}_{t|t})^T|y_{0:t}]}_{P_{t|t}} A^T + 0 + \Sigma_w \\ =& AP_{t|t}A^T + \Sigma_w\end{aligned} \tag{B.9}$$

Correction Step
The conditional distribution of x_{t+1} is now known and next the conditional distribution of y_{t+1} and the cross correlation between x_{t+1} and y_{t+1} are found. First the mean of y_{t+1} is found,

$$\begin{aligned}\hat{y}_{t+1|t} \triangleq& \mathbb{E}[y_{t+1}|y_{0:t}] \\ =& \mathbb{E}[Cx_{t+1} + v_{t+1}|y_{0:t}] \\ =& C\hat{x}_{t+1|t}\end{aligned} \tag{B.10}$$

Secondly the covariance is found,

$$\begin{aligned}
&\mathbb{E}[(y_{t+1} - \hat{y}_{t+1|t})(y_{t+1} - \hat{y}_{t+1|t})^T | y_{0:t}] \\
&= \mathbb{E}[(Cx_{t+1} + v_{t+1} - C\hat{x}_{t+1|t})(Cx_{t+1} + v_{t+1} - C\hat{x}_{t+1|t})^T | y_{0:t}] \\
&= \mathbb{E}[C(x_{t+1} - \hat{x}_{t+1|t})(x_{t+1} - \hat{x}_{t+1|t})^T C^T | y_{0:t}] + 2\mathbb{E}[v_{t+1}(x_{t+1} - \hat{x}_{t+1|t})^T C^T | y_{0:t}] + 2\mathbb{E}[v_{t+1} v_{t+1}^T | y_{0:t}] \\
&= CP_{t+1|t}C^T + \Sigma_v
\end{aligned} \tag{B.11}$$

And lastly the cross correlation between x_{t+1} and y_{t+1},

$$\begin{aligned}
&\mathbb{E}[(y_{t+1} - \hat{y}_{t+1|t})(x_{t+1} - \hat{x}_{t+1|t})^T | y_{0:t}] \\
&= \mathbb{E}[(Cx_{t+1} + v_{t+1} - C\hat{x}_{t+1|t})(x_{t+1} + w_{t+1} - \hat{x}_{t+1|t})^T | y_{0:t}] \\
&= CP_{t+1|t}
\end{aligned} \tag{B.12}$$

Summarizing, the variables x_{t+1} and y_{t+1} has a Gaussian distribution described by Equation (B.13)

$$\begin{pmatrix} x_{t+1|t} \\ y_{t+1|t} \end{pmatrix} \sim \mathcal{N}\left(\begin{bmatrix} \hat{x}_{t+1|t} \\ C\hat{x}_{t+1|t} \end{bmatrix}, \begin{bmatrix} P_{t+1|t} & P_{t+1|t}C^T \\ CP_{t+1|t} & CP_{t+1|t}C^T + \Sigma_v \end{bmatrix} \right) \tag{B.13}$$

To find the conditional distribution of Equation (B.13) a simplified example is made,

$$\begin{pmatrix} x \\ y \end{pmatrix} \sim \mathcal{N}\left(\begin{bmatrix} \mu_x \\ \mu_y \end{bmatrix}, \begin{bmatrix} \Sigma_{xx} & \Sigma_{xy} \\ \Sigma_{yx} & \Sigma_{yy} \end{bmatrix} \right) \tag{B.14}$$

This can then be written as the joint distribution,

$$p(x,y) = \frac{1}{2\pi^{(n+d)/2} |\Sigma|^{1/2}} e^{-\frac{1}{2} \begin{pmatrix} x-\mu_x \\ y-\mu_y \end{pmatrix}^T \left(\underbrace{\begin{bmatrix} \Sigma_{xx} & \Sigma_{xy} \\ \Sigma_{yx} & \Sigma_{yy} \end{bmatrix}}_{\Sigma} \right)^{-1} \begin{pmatrix} x-\mu_x \\ y-\mu_y \end{pmatrix}} \tag{B.15}$$

[Jordan, 2003, chap 13 p.7] where:
n is the dimension of x
d is the dimension of y

The joint distribution can then be split up into a marginal probability for y and a conditional probability for x,

$$p(x,y) = p(x|y)p(y) \tag{B.16}$$

Applying this property to Equation (B.15) it is possible to split up the joint distribution into the product of a marginal distribution and the following conditional distribution [Jordan, 2003, chap 13 p.7],

$$p(x|y) = \frac{1}{2\pi^{n/2} \left|\Sigma_{x|y}\right|^{1/2}} e^{-\frac{1}{2}(x-\mu_x-\Sigma_{xy}\Sigma_{yy}^{-1}(y-\mu_y))^T (\Sigma/\Sigma_{yy})^{-1} (x-\mu_x-\Sigma_{xy}\Sigma_{yy}^{-1}(y-\mu_y))} \tag{B.17}$$

where:
Σ/Σ_{yy} is the Schur complement of Σ with respect to Σ_{yy} [Lay, 2003, p. 139]

From this the following two equations for the conditional mean and covariance can be extracted,

$$\mu_{x|y} = \mu_x + \Sigma_{xy}\Sigma_{yy}^{-1}(y - \mu_y) \tag{B.18}$$
$$\Sigma_{x|y} = \Sigma_{xx} - \Sigma_{xy}\Sigma_{yy}^{-1}\Sigma_{yx} \tag{B.19}$$

By utilizing equations (B.18) and (B.19) on Equation (B.13) yields,

$$\hat{x}_{t+1|t+1} = \hat{x}_{t+1|t} + P_{t+1|t}C^T(CP_{t+1|t}C^T + \Sigma_v)^{-1}(y_{t+1} - C\hat{x}_{t+1|t}) \tag{B.20}$$
$$P_{t+1|t+1} = P_{t+1|t} - P_{t+1|t}C^T(CP_{t+1|t}C^T + \Sigma_v)^{-1}CP_{t+1|t} \tag{B.21}$$

Summarize
To summarize the two prediction equations (B.8) and (B.9) and the two update equations (B.20) and (B.21) are listed below, where the two update equations are simplified by introducing the Kalman gain matrix K,

$$\hat{x}_{t+1|t} = A\hat{x}_{t|t} + Bu_t \tag{B.22}$$
$$P_{t+1|t} = AP_{t|t}A^T + \Sigma_w \tag{B.23}$$
$$K_{t+1} \triangleq P_{t+1|t}C^T(CP_{t+1|t}C^T + \Sigma_v)^{-1} \tag{B.24}$$
$$\hat{x}_{t+1|t+1} = \hat{x}_{t+1|t} + K_{t+1}(y_{t+1} - C\hat{x}_{t+1|t}) \tag{B.25}$$
$$P_{t+1|t+1} = P_{t+1|t} - K_{t+1}CP_{t+1|t} \tag{B.26}$$

Appendix C

The Rauch-Tung-Striebel (RTS) Smoother

Using the Kalman filter allows to estimate the states based on observations up to and including the time t. This section will concern the issue of obtaining estimates of the states based on all observations up to the time T, which is called a smoother. The Rauch-Tung-Striebel (RTS) smoother works by first forward filtering as described in section 4.1, followed by a backward pass through all the samples which is the smoother. All measurements therefore have to be available to be able to run the smoother. The smoother can therefore not be used in real time applications. Unless other is mentioned the following section is inspired by [Jordan, 2003, Chap. 15]

The joint distribution of x_t and x_{t+1} conditional on $y_{0:t}$

$$\mathbb{E}\left[(x_t - \hat{x}_{t|t})(x_{t+1} - \hat{x}_{t+1|t})^T | y_{0:T}\right] = \mathbb{E}\left[(x_t - \hat{x}_{t|t})(Ax_t - A\hat{x}_{t|t})^T | y_{0:T}\right] = P_{t|t}A^T \quad \text{(C.1)}$$

Thus the following distribution,

$$\begin{pmatrix} x_t \\ x_{t+1} \end{pmatrix} \sim \mathcal{N}\left(\begin{bmatrix} \hat{x}_{t|t} \\ \hat{x}_{t+1|t} \end{bmatrix}, \begin{bmatrix} P_{t|t} & P_{t|t}A^T \\ AP_{t|t} & P_{t+1|t} \end{bmatrix}\right) \quad \text{(C.2)}$$

The backward computation is then introduced by calculation the probability of x_t conditioned on x_{t+1} and P_t conditioned on P_{t+1}, both still conditioned on $y_{0:t}$. By using the condition rule from Equation (B.18) on (C.2) it is possible to obtain,

$$\begin{aligned} \mathbb{E}\left[x_t | x_{t+1}, y_{0:t}\right] &= \hat{x}_{t|t} + P_{t|t}A^T P_{t+1|t}^{-1}(x_{t+1} - \hat{x}_{t+1|t}) \\ &= \hat{x}_{t|t} + L_t(x_{t+1} - \hat{x}_{t+1|t}) \end{aligned} \quad \text{(C.3)}$$

where:
$L_t \triangleq P_{t|t}A^T P_{t+1|t}^{-1}$ is the smoother gain matrix introduced to simplify the notation.

Note that the smoother gain only depends of matrices calculated during the forward pass. Using

the condition rule from Equation (B.19) on (C.2) yields,

$$\begin{aligned} Var\left[x_t|x_{t+1}, y_{0:t}\right] =& P_{t|t} - P_{t|t}A^T P_{t+1|t}^{-1} A P_{t|t}) \\ =& P_{t|t} - L_t P_{t+1|t} L_t^T \end{aligned} \tag{C.4}$$

Since the purpose is to estimate x_t conditioned on x_{t+1} it is independent of the future observations $y_{t+1:T}$. Conditional independence can therefore be used to write,

$$\begin{aligned} \mathbb{E}\left[x_t|x_{t+1}, y_{0:T}\right] =& \mathbb{E}\left[x_t|x_{t+1}, y_{0:t}\right] \\ =& \hat{x}_{t|t} + L_t(x_{t+1} - \hat{x}_{t+1|t}) \end{aligned} \tag{C.5}$$

$$\begin{aligned} Var\left[x_t|x_{t+1}, y_{0:T}\right] =& Var\left[x_t|x_{t+1}, y_{0:t}\right] \\ =& P_{t|t} - L_t P_{t+1|t} L_t^T \end{aligned} \tag{C.6}$$

Equation (C.5) and (C.6) are almost what is wanted. The only thing left is to remove the x_{t+1} from the conditions on the left hand side. This can be archived by using the following properties of conditional expectation [Ross, 1996, p. 33,51],

$$\mathbb{E}\left[X|Y\right] = \mathbb{E}\left[\mathbb{E}\left[X|Y,Z\right]|Z\right] \tag{C.7}$$

$$Var[X|Y] = Var[\mathbb{E}\left[X|Y,Z\right]|Z] + \mathbb{E}\left[Var[X|Y,Z]|Z\right] \tag{C.8}$$

First the conditional expectation in Equation (C.7) are used to rewrite Equation (C.5),

$$\begin{aligned} \hat{x}_{t|T} \triangleq& \mathbb{E}\left[x_t|, y_{0:T}\right] \\ =& \mathbb{E}\left[\mathbb{E}\left[x_t|x_{t+1}, y_{0:T}\right]|y_{0:T}\right] \\ =& \mathbb{E}\left[\hat{x}_{t|t} + L_t(x_{t+1} - \hat{x}_{t+1|t})|y_{0:T}\right] \\ =& \hat{x}_{t|t} + L_t(x_{t+1|T} - \hat{x}_{t+1|t}) \end{aligned} \tag{C.9}$$

Where the last step in Equation (C.9) uses the fact that all quantities other than x_{t+1} are constants when conditioned on $y_{0:T}$. From Equation (C.9) it is evident that the estimate of x_t based on all data can be obtained by correcting with the smoother gain, L_t, multiplied with the difference between the smoothed and filtered state x_{t+1}. Secondly utilizing the conditional expectation in Equation (C.8) on Equation (C.6) yields,

$$\begin{aligned} P_{t|T} \triangleq& Var[x_t|y_{0:T}] \\ =& Var[\mathbb{E}\left[x_t|x_{t+1}, y_{0:T}\right]|y_{0:T}] + \mathbb{E}\left[Var[x_t|x_{t+1}, y_{0:T}]|y_{0:T}\right] \\ =& Var[\hat{x}_{t|t} + L_t(x_{t+1} - \hat{x}_{t+1|t})|y_{0:T}] + \mathbb{E}\left[P_{t|t} - L_t P_{t+1|t} L_t^T|y_{0:T}\right] \\ =& L_t Var[x_{t+1} - \hat{x}_{t+1|t}|y_{0:T}] L_t^T + P_{t|t} - L_t P_{t+1|t} L_t^T \\ =& L_t Var[x_{t+1}|y_{0:T}] L_t^T + P_{t|t} - L_t P_{t+1|t} L_t^T \\ =& L_t P_{t+1|T} L_t^T + P_{t|t} - L_t P_{t+1|t} L_t^T \\ =& P_{t|t} + L_t(P_{t+1|T} - P_{t+1|t}) L_t^T \end{aligned} \tag{C.10}$$

Where the fact that expectations taken with respect to $y_{0:t}$ are constants when conditioning

on the larger conditioning set $y_{0:T}$. To summarize the RTS smoothing algorithm in Equation (C.9) and (C.10) along with the smoother gain are listed below.

$$L_t = P_{t|t} A^T P_{t+1|t}^{-1} \tag{C.11}$$

$$\hat{x}_{t|T} = \hat{x}_{t|t} + L_t(x_{t+1|T} - \hat{x}_{t+1|t}) \tag{C.12}$$

$$P_{t|T} = P_{t|t} + L_t(P_{t+1|T} - P_{t+1|t})L_t^T \tag{C.13}$$

Appendix D

Soft Time Timing of Control Algorithm in C/C++

To control the timing of the control loop the function `soft_real_time_sleep()` has been implemented to ensure that that control algorithm runs with a specified time interval T. The timer is implemented using the function `usleep()`. The function takes in three arguments, start time, end time and the time interval T. In Figure D.1 the principal of the timer is illustrated.

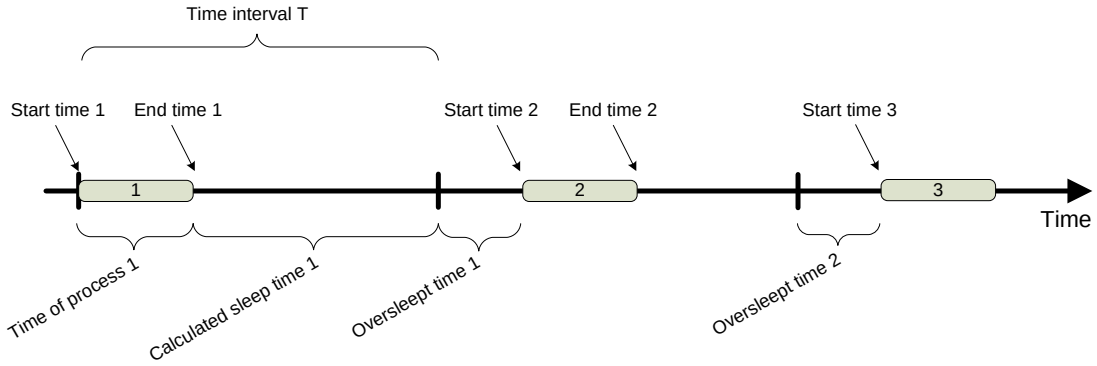


Figure D.1: Illustrated of a time axis slotted with the constant time period T

Before the control process is started, the start time in microseconds is found, using the function `gettimeofday()`, and after the process is done, the end time is found. By taking the difference, the duration of the process can be calculated and hereby how long the timer has to sleep. However, since the function `usleep()` can sleep more than the specified time, this will in time grow to an unacceptable error. Therefore the oversleep time is calculated,

$$\text{oversleep}[n] = \text{start}[n+1] - \text{start}[n] + \text{oversleep}[n-1] - T \tag{D.1}$$

The oversleep time can then be calculated taking into account when calculating the next sleep time,

$$\text{sleep}[t+1] = T - (\text{end}[t+1] - \text{start}[t+1]) - \text{oversleep}[t] \tag{D.2}$$

This means that the error caused by the function `usleep()` over sleeping is corrected. However,

some jitter can occur in the start times since the oversleep times can vary over time.

D.1 Testing

The test is done by running the main control loop run 10,000 times with 20 Hz. The oversleep times is then logged for each loop which are plotted in Figure D.2. From this it can be concluded that the `usleep()` function sleeps an extra 18 micro seconds in average with a maximum of 180 microseconds which is acceptable since the control loop only runs at 20 Hz.

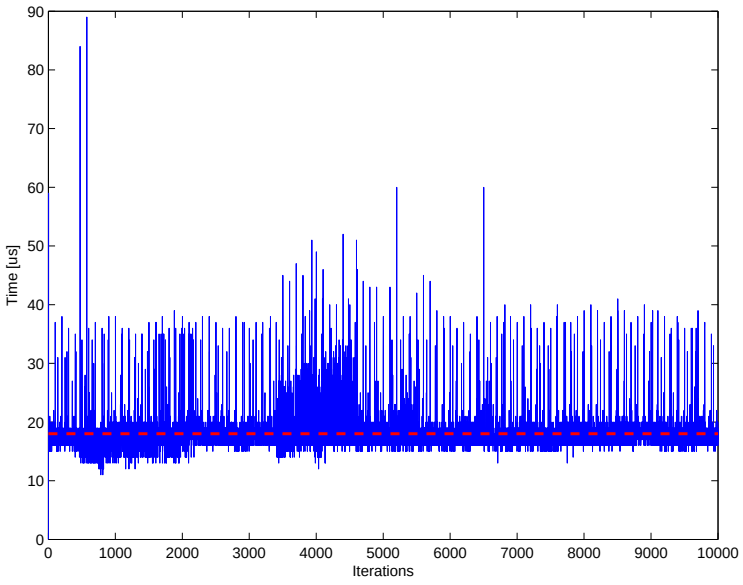


Figure D.2: Time the main loop has overslept at each iteration

The total time spend running the 10,000 loops where 500 seconds and 60 microseconds which is consistent with the expected time since each time interval is supposed to be .05 seconds. The fact that the test took 60 microseconds longer, is only expected since the `usleep()` function, will end up by sleeping over without it being corrected.

It can therefore be concluded that this function makes it possible to run processes at specified time intervals, but with jitter up to 180 microseconds.