Safety in
Automated Surgery
with the da Vinci Robot
INTUITIVE
SURGICAL
PERMANENT
CAUTERY
SPATULA
Master thesis 2015 | Christian Køcks Lykkegaard and Britt Louise Jakobsen
Aalborg University | Faculty of Engineering and Science | School of Information and Communication Technology | Department of Electronic Systems | Section for Control and Automation
AALBORG UNIVERSITY
STUDENT REPORT

School of Information and Communication Technology
Fredrik Bajers Vej 7
9220 Aalborg Øst
Phone +45 9940 8600
Fax +45 9940 9840
http://www.es.aau.dk

Title: Safety in Automated Surgery with the da Vinci Robot
Master Thesis: Control & Automation
Project period: Feb. 2nd – June 3rd 2015
Project group: CA 15gr1032
Participants:

Britt Louise Jakobsen

Christian Køcks Lykkegaard

Supervisor:
Prof. Rafał Wiśniewski

Report: 92 pages
Appendix: 69 pages
Attached: 1 CD

Abstract:

In the course of the last few decades, robotic surgery has become the preferred type of operation within certain types of surgery, allowing the surgeon to perform precision procedures causing the patient a minimal amount of scarring while maintaining an exceptional overview of the operation site for the surgeon.
Advances are made within automation in the control of robotic tools, providing the surgeon with more freedom and higher precision when performing operations. Based on having attended a robotic surgery and recognizing the development within this research community, it is the aim of this thesis to contribute to the advancement through the use of barrier certificates with which safety of the automated control system can be guaranteed.
Control systems are developed for use cases within robotic surgery, including control in 3D Euclidean space and virtual fixture control of a beating heart. System safety is certified through the construction of a barrier enclosing areas termed unsafe thus guaranteeing that the robotic tool will never cross this barrier. Two different approaches are taken: design of safety controllers based on manually constructed control barrier functions, and analytic verification of system safety using a software tool to construct the certificates. The safety verification development constitutes a framework in which any system can be validated in accordance with its safety requirements.
The controllers are implemented in C++ through the ROS framework on a first generation da Vinci surgical robot, and safety is verified for the developed control systems thus demonstrating the applicability of the theory of barrier certificates.

The content of this report is freely available, but publication (with source reference) may only take place by agreement with the authors.

Preface

This report documents the development process of a safe controller for automation of a surgical robot arm with patient safety guaranteed through barrier certificates. The access to robot measurement data and opportunity to implement a controller heavily benefits from the previous work carried out on the da Vinci surgical robot in the Control Laboratory at Aalborg University. The project is rated at 30 ECTS-points, and the work is conducted by the 4$^{\text{th}}$ semester group 1032 within the graduate program in Control and Automation at Aalborg University during the spring of 2015.

Reading Guide

The primary focus of this report is to design a controller and a barrier certificate, the certificate guaranteeing the safe control of a surgical robot, as an approach to draw closer to the possibility of implementing automated control tasks by surgical robots. After an introduction into surgical robotics and the definition of barrier certificates, two approaches to the design problem are described:

- Explicit approach: A barrier certificate is constructed and a safe controller is designed according to the method described in [Wieland and Allgöwer, 2007].
- Analytic approach: A controller is designed, criteria are constructed for a barrier certificate and safety is verified by use of Putinar's Positivstellensatz.

Symbols, acronyms and a glossary are presented in the nomenclature before the main report. A variant of the Harvard referencing is used for citations, with the author and publication year of the source given in square brackets, e.g. [Lasserre 78], and sources listed in the bibliography at the end of the main report. A comprehensive appendix is included after the bibliography, containing introductions to the used software, detailed derivations, measurement logs and source code. A digital copy of this report along with cited references, source code and simulation results can be found on the enclosed CD.

Acknowledgements

The authors wish to thank Assistant Engineer Simon Jensen for a thorough introduction to the custom made AAU da Vinci hardware and design of a dynamic heart phantom platform; Ph.D. Tobias Leth for guidance in reference frame construction for robot kinematics; Post Doc. Karl Damkjær Hansen for an introduction to the AAU da Vinci robot operative system and help in implementing inverse kinematics; and Assistant Professor Christoffer Sloth for help and guidance with the theory behind barrier certificates and the use of SOSTOOLS.

Last but not least, it is desired to thank chief surgeon Johan Poulsen, robot assistant nurse Jane Petersson and surgeon Grazvydas Tuckus for sharing their insights in the use of surgical robotics and allowing the authors to attend a robotic surgery at Aalborg University Hospital.

Contents

Nomenclature V

1 Introduction 1
1.1 Highlights in the Development of Surgical Robotics . . . 1
1.2 State-of-the-Art in Surgical Robotics . . . 2
1.3 Envisioned Future for Robotic Surgery at Aalborg University Hospital . . . 3
1.4 Configuration of da Vinci at Aalborg University . . . 4

2 Safety Guarantee by Barrier Certificates 9
2.1 Constraints for a Barrier Certificate . . . 9
2.2 Approaches to the Problem of Guaranteeing System Safety . . . 12

3 Controller Design from CBFs 13
3.1 The Control Law . . . 13
3.2 Using CBFs in the Control Design for the da Vinci Robot . . . 16

4 Founding Safety with Static Boundaries 17
4.1 Modelling of Slide Movement . . . 18
4.2 Construction of CBF . . . 20
4.3 Control Design . . . 25
4.4 MATLAB Implementation and Results . . . 30
4.5 Implementation on the da Vinci Robot . . . 33
4.6 Conclusion for the Slide Safety Controller . . . 38

5 Safety and Surgery on Beating Hearts 39
5.1 Modelling the Dynamics of the System . . . 39
5.2 Construction of CBF . . . 42
5.3 Control Design . . . 42
5.4 MATLAB Implementation and Results . . . 43
5.5 Implementation on the da Vinci Robot . . . 44
5.6 Conclusion for the Beating Heart Controller . . . 47

6 Safety in the 3D Euclidean Space 48
6.1 Modelling of Robot Hand Movement in 3D . . . 49
6.2 Construction of CBF . . . 50
6.3 Control Design . . . 51
6.4 MATLAB Implementation . . . 53

6.5 Implementation on the da Vinci Robot 55
6.6 Conclusion for Safety in the 3D Euclidean Space 65

7 Interim Conclusion **67**

8 Safety Verification with Barrier Certificates **69**
8.1 Recasting the Barrier Certificate Definition 71

9 Barrier Certificate Search with SOSTOOLS **73**
9.1 Formulation of a Barrier Certificate in SOSTOOLS Syntax 73
9.2 Barrier Certificate Search for First Order Robot Slide System 76
9.3 Barrier Certificate Search for Second Order Robot Slide System 86
9.4 Conclusion on the Use of SOSTOOLS 89

10 Conclusion and Discussion **90**

Literature **93**

Appendix A Interfacing da Vinci with ROS **96**
A.1 General structure of a ROS setup 97
A.2 Setup of Low Level Control and how to Initiate ROS 98
A.3 Specific Structure of this Thesis - The gr1032 Development Branch 99
A.4 Structure of Moveit and Why it is Not Used 101

Appendix B Links and Joints 3D Overview **105**

Appendix C Kinematic Models of the Robot **106**
C.1 Existing Kinematics for the AAU da Vinci Robot 107
C.2 Defining Kinematics According to Denavit-Hartenberg Convention 111
C.3 Defining da Vinci Kinematics for Active Joints 114

Appendix D Dynamic Model of a Beating Heart **117**

Appendix E MATLAB Toolbox SOSTOOLS **118**

Appendix F Measurement Logs **119**
F.1 Step Response of Slide Position 119
F.2 Step Response of the da Vinci Robot in 3D Cartesian Space 121

Appendix G MATLAB Implementation **124**
G.1 Implementation of Safety Controllers 124
G.2 Safety Verification with SOSTOOLS 131

Appendix H Da Vinci Implementation of Controllers **138**

Appendix I Developed Auxiliary Files **160**

Appendix J Attached CD **165**

Nomenclature

Acronyms

API	Application Programmable Interface
CBF	Control Barrier Function
CLF	Control Lyapunov Function
DARPA	Defence Advanced Research Project Administration, research administration under DoD
DH	Denavit-Hartenberg, convention for placement of coordinate frames
DOF	Degrees Of Freedom
FK	Forward Kinematics
FPGA	Field Programmable Gate Array
GUI	Graphical user interface
IK	Inverse Kinematics
KDL	Kinematics and Dynamics Library
MASH	Mobile Advanced Surgical Hospital
MIS	Minimally Invasive Surgery
MPC	Model Predictive Control
NASA	National Aeronautics and Space Administration
REP	ROS Enhancement Proposals
RIO	Reconfigurable Input/Output
ROS	Robotic Operating System
RPY	Roll Pitch Yaw angles, extrinsic rotation about x, y, z axes, respectively
SDP	Semi-Definite Programming
SOS	Sum of Squares
SRDF	Semantic Robot Description Format
TCP/IP	Transmission Control Protocol/Internet Protocol
UDP	User Datagram Protocol
UI	User Interface
URDF	Unified Robot Description Format
XACRO	XML Macros
YAML	Yet Another Markup Language

Symbols

$\mathbf{A}$	Linear system matrix, $\mathbf{A} \in \mathbb{R}^{n\times n}$	$[\cdot]$
$\mathbf{B}$	Linear input matrix, $\mathbf{B} \in \mathbb{R}^{n\times m}$	$[\cdot]$
$\mathcal{C}$	Controllability matrix, $\mathcal{C} \in \mathbb{R}^{n\times n}$	$[\cdot]$
$\mathbf{C}$	Linear output matrix, $\mathbf{C} \in \mathbb{R}^{q\times n}$	$[\cdot]$
Δ	Distance between safe and unsafe regions $\mathcal{X}_0$ and $\mathcal{X}_u$, when defining $B(\S)$ in SOS	$[\cdot]$

$\mathbf{D}$	Linear feedforward matrix, $\mathbf{D} \in \mathbb{R}^{q\times m}$	$[\cdot]$
Γ	Discrete version of the linear system matrix $\mathbf{A}$, $\Gamma \in \mathbb{R}^{n\times n}$	$[\cdot]$
$\bar{\mathbf{K}}$	Augmented feedback matrix, $\bar{\mathbf{K}} \in \mathbb{R}^{m\times n}$	$[\cdot]$
$\mathbf{K}_d$	Feedback matrix calculated from discrete matrices, $\mathbf{K}_d \in \mathbb{R}^{m\times n}$	$[\cdot]$
$\mathbf{K}$	Feedback matrix calculated from continuous matrices, $\mathbf{K} \in \mathbb{R}^{m\times n}$	$[\cdot]$
$\mathbf{L}_d$	Observer gain calculated from discrete matrices, $\mathbf{L}_d \in \mathbb{R}^{n\times m}$	$[\cdot]$
M_0	Semimajor axis of a CBF which is a function of two state variables	$[\cdot]$
$\mathbf{M}$	Gain correction for an observer, $\mathbf{M} \in \mathbb{R}^{n\times m}$	$[\cdot]$
$\bar{\mathbf{N}}$	Gain correction for a system without an observer, $\bar{\mathbf{N}} \in \mathbb{R}^{m}$	$[\cdot]$
$\mathbf{N}$	Gain correction for a system with an observer, $\mathbf{N} \in \mathbb{R}^{m}$	$[\cdot]$
$\mathcal{O}$	Observability matrix, $\mathcal{O} \in \mathbb{R}^{n\times n}$	$[\cdot]$
Φ	Discrete version of the linear input matrix $\mathbf{B}$, $\Phi \in \mathbb{R}^{n\times m}$	$[\cdot]$
$\mathbf{Q}$	Real positive semidefinite symmetric SOS coefficient matrix	$[\cdot]$
$\mathbf{R}$	Rotation matrix describing relative rotation between coordinate frames, $\mathbf{R} \in \mathbb{R}^{3\times 3}$	$[\cdot]$
$\Sigma[\mathbf{x}]$	Denotion of an SOS polynomial in the variable $\mathbf{x}$	$[\cdot]$
$\mathcal{T}$	The set of all (safe) states in a transition area between the safe and unsafe region	$[\cdot]$
T_h	Heart beat period	[s]
$\mathbf{T}$	Transformation matrix describing rotation and translation between two coordinate frames, $\mathbf{T} \in \mathbb{R}^{4\times 4}$	$[\cdot]$
T_s	Sample period	[s]
T	Final value of a time series	[s]
$V(\mathbf{x})$	Lyapunov function	$[\cdot]$
$\mathcal{X}_0$	The set of all safe initial states $\mathcal{X}_0 \subseteq \mathcal{X}$	$[\cdot]$
$\mathcal{X}_u$	The set of all unsafe states, $\mathcal{X}_u \subset \mathcal{X}$	$[\cdot]$
$\mathcal{X}$	The set of all considered states for which safety should be guaranteed, $\mathcal{X} \subseteq \mathbb{R}^n$	$[\cdot]$
$\mathcal{Y}$	The set of all safe states but excluding the set $\mathcal{T}$, i.e. $\mathcal{Y} = \mathcal{X}_0 \setminus \mathcal{T}$	$[\cdot]$
α	Rotation angle about local x axis	[rad]
$B(\mathbf{x})$	Barrier function where $B(\mathbf{x}) \in C^1(\mathcal{X})$	$[\cdot]$
β	Rotation angle about local y axis	[rad]
$\mathbf{c}$	Centre coordinate for ellipsoid	[m]
c_p	Computation time	[s]
δ_{err}	Maximum allowed distance between reference and state	$[\cdot]$
d_{ref}	Distance between a beating heart and robot end effector	[m]
$\mathbf{d}$	Disturbance input, restricted to $\mathbf{d} \in \mathcal{D} \subseteq \mathbb{R}^p$ and convex.	$[\cdot]$
$\bar{\varepsilon}$	Minimum value of $B(\mathbf{x})$ on the unsafe set $\mathcal{X}_u$, when defining $B(\mathbf{x})$ in SOS	$[\cdot]$
ε	Value of $B(\mathbf{x})$ at the border between $\mathcal{T}$ and $\mathcal{Y}$	$[\cdot]$
f_s	Sampling time	[Hz]
f	A function, $f : \mathbb{R}^n \to \mathbb{R}^n$, mostly a system function in a SS system	$[\cdot]$
g	Either: A function, $g : \mathbb{R}^n \to \mathbb{R}^{n\times m}$, mostly an input function in a SS system. Or: In SOS g is a polynomial which is positive on some set	$[\cdot]$
h	A function, $h : \mathbb{R}^n \to \mathbb{R}^{n\times p}$, mostly a disturbance function in a SS system	$[\cdot]$
$k_0(\mathbf{x})$	Safety controller	$[\cdot]$
Λ_i	Physical boundary where $i = h =$ hard boundary and $i = s =$ soft boundary	$[\cdot]$
λ	Eigenvalue	$[\cdot]$
L_gB	Lie derivative of $B(\mathbf{x})$ along the vector field $g(\mathbf{x})$, i.e. $L_gB(\mathbf{x}) = \frac{dB(\mathbf{x})}{d\mathbf{x}} g(\mathbf{x})$	$[\cdot]$

L_fB	Lie derivative of $B(\mathbf{x})$ along the vector field $f(\mathbf{x})$, i.e. $L_fB(\mathbf{x}) = \frac{dB(\mathbf{x})}{d\mathbf{x}} f(\mathbf{x})$	[$\cdot$]
$\mathbf{z}$	Monomial vector in the variable $\mathbf{x}$	[$\cdot$]
m	Number of inputs to a system	[$\cdot$]
n	Number of states in a system	[$\cdot$]
ω_n	Natural frequency of a second order system	[rad/s]
$\mathbf{p}$	Position vector describing translation between coordinate frame origins, $\mathbf{p} \in \mathbb{R}^3$	[m]
q	Number of outputs from a system	[$\cdot$]
$\mathbf{r}$	Radius vector for ellipsoid	[m]
$\sigma(\mathbf{x})$	Parameter that determines the linear combination between the safe and non-safe controller. It is restricted such that $0 \leq \sigma(\mathbf{x}) \leq 1$	[$\cdot$]
s	The Laplace operator	[$\cdot$]
τ	Time constant	[s]
θ	Rotation angle about local z axis	[rad]
$\tilde{u}(\mathbf{x})$	Linear position controller	[$\cdot$]
$\mathbf{u}$	Control input restricted to $\mathbf{u} \in \mathbb{R}^m$ where m is the number of inputs	[$\cdot$]
ω_h	Frequency of a beating heart	[rad/s]
x_1	Position state of the robot in a system	[m]
x_2	Velocity state of the robot in a system	[m/s]
x_{h10}	Initial value of the position state	[m]
x_{h1}	The position state of a beating heart	[m]
x_{h12}	Initial value of the velocity state	[m/s]
x_{h2}	The velocity state of a beating heart	[m/s]
$\mathbf{x}$	The state variable which is restricted to $\mathbf{x} \in \mathbb{R}^n$ where n is the number of states	[$\cdot$]
$\mathbf{y}$	Output vector, $\mathbf{y} \in \mathbb{R}^q$ where q is the number of outputs	[$\cdot$]
ζ	Damping coefficient	[$\cdot$]

General Notation Remarks

Vectors are written i bold upright lowercase letters e.g. $\mathbf{x}$, and vector entries are written as the same italic lowercase letter with a subscript generally denoting its entry e.g. x_1. The composition of the entries will be clear from the context. Matrices are written in bold upright uppercase letters e.g. $\mathbf{A}$, its transpose is denoted by $\mathbf{A}^T$ and its inverse is denoted by $\mathbf{A}^{-1}$.

The n-dimensional real Euclidian space is denoted by $\mathbb{R}^n$ and subsets of the real space are written in calligraphic letters e.g. $\mathcal{X} \subseteq \mathbb{R}^n$. A function is written in italic letters followed by the variable(s) it is a function of e.g. $f(\mathbf{x})$. A differentiable function defined on $\mathbb{R}^n$ is denoted by $f \in C^1(\mathbb{R})$.

The time derivative of a variable is indicated by a dot above the symbol e.g. $\dot{\mathbf{x}} = d\mathbf{x}(t)/dt$. In general the notation (t) denoting a function of time is implicit for the state vector $\mathbf{x}$, and is only included to emphasize the time dependency. For functions of the state e.g. $q(\mathbf{x})$ the notation $(\mathbf{x})$ is left out when the dependency is clear from the context. The derivative notation $dB(\mathbf{x})/d\mathbf{x}$ implies the row vector of partial derivatives of B with respect to $x_1,...,x_n$.

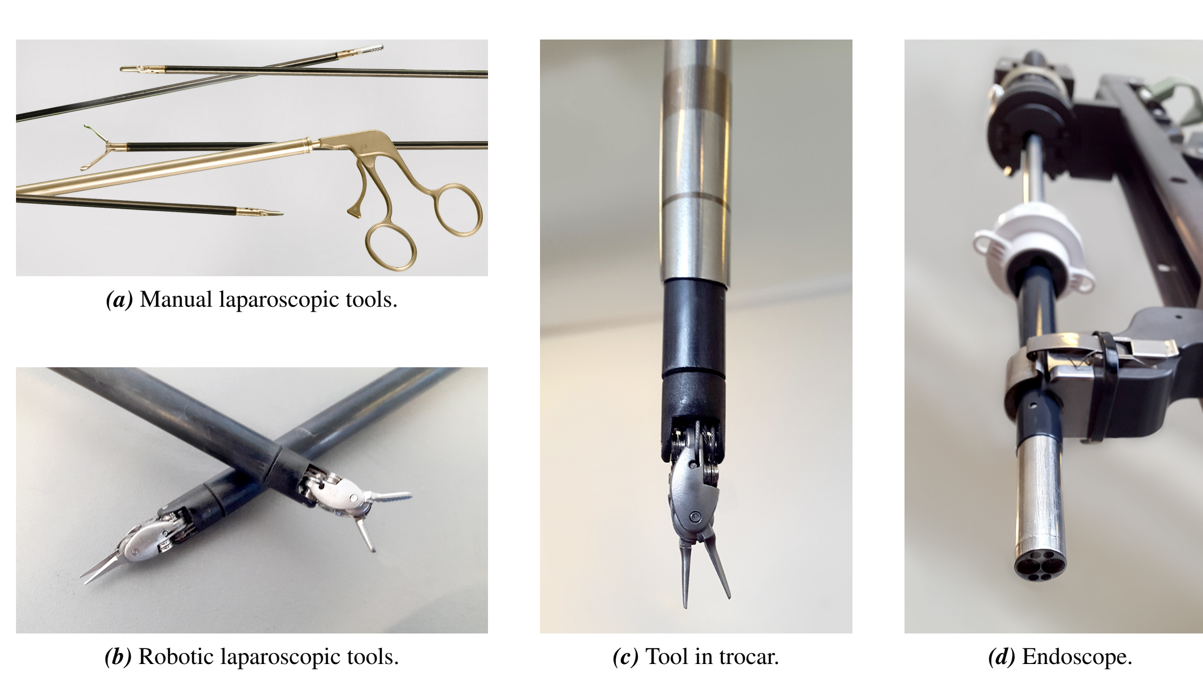
Function names such as $B(\mathbf{x})$ or $k_0(\mathbf{x})$ represent the same functionality throughout, but may attain different configurations and coefficient values for different system models. The relevant configuration and numerical values should be clear from the context. The same applies for parameters such as Δ or $\bar{\varepsilon}$.

Note that the function $B(\mathbf{x})$ should not be confused with the matrix $\mathbf{B}$.

Chapter 1

Introduction

When the recovery from an injury or disease requires a surgical procedure, traditionally this has been done with open surgery, where the patient is cut open for the surgeon to perform the procedure. For many types of operation, however, alternatives to open surgery have emerged, especially over the last half century. In minimally invasive surgery (MIS), as opposed to traditional open surgery, only small incisions are made in the patient's abdomen or pelvis in order to gain access to the area under surgery, hence causing less trauma beyond this confined area. This in general provides the patient with quicker recovery, shorter hospital stay and less scarring. One type of MIS is laparoscopy, invented in the beginning of the 20th century [Hatzinger et al., 2006], where thin metal telescopes (laparoscopes) with specialized surgical tools attached are inserted into the patient through trocars, allowing the surgeon to maneuver the tools in the inflated abdomen guided by visual feedback from a flexible miniature camera (endoscope) inserted alongside the surgical tools [Peters, 2013], see figure 1.1. In the 1980s robotic laparoscopic surgery was introduced as a master-slave system, where the surgeon controls a robot arm holding the surgical tools from a master console, instead of manipulating the instruments manually.



(a) Manual laparoscopic tools.

(b) Robotic laparoscopic tools.

(c) Tool in trocar.

(d) Endoscope.

Figure 1.1: Tools used in laparoscopic surgery.

1.1 Highlights in the Development of Surgical Robotics

While the idea of roboticized telemedicine dates back to 1925 [Novak, 2012], the development of telesurgery was founded by the National Aeronautics and Space Administration (NASA) in the 1970s combining research within virtual reality, robotics and medicine, and the first robotic surgery procedure

was accomplished in 1985, followed by the first laparoscopic robotic surgical procedure in 1987 [Wall and Marescaux, 2013; Galeota-Sprung et al., 2004]. In 1998 the first fully endoscopic robotic surgery were performed and the idea of operating on a beating-heart were initiated [Galeota-Sprung et al., 2004].

The first commercially available surgical robot was introduced in the early 1990s. At the same time major research within telesurgery was funded by Defence Advanced Research Project Administration (DARPA), concurrent with the U.S. Army developing the Mobile Advanced Surgical Hospital (MASH) for loading and teleoperating wounded soldiers in vehicular operating rooms [Wall and Marescaux, 2013; Galeota-Sprung et al., 2004].

Surgical robots teleoperated from more than a few meters away is, however, still incipient. In 1996 the first tests were performed demonstrating the successful use of telementoring and telemanipulation of the endoscope by a surgeon placed several 100 m away from the operating room, and in 2001 the first transatlantic telesurgical procedure, the Lindbergh Operation, was performed by a team of French doctors in New York operating on a patient in Strasbourg [Wall and Marescaux, 2013]. More research into remotely telementored and teleoperated robotic surgery was performed during the 2000s with the NASA Extreme Environment Mission Operations (NEEMO) projects and as part of the DARPA Trauma Pod program launched in 2005 [Satava et al., 2011; Hannaford and Rosen, 2006].

1.2 State-of-the-Art in Surgical Robotics

Most surgical robots used for telesurgery are master-slave systems which can be fully controlled by the surgeon, see figure 1.2. The patient manipulator consists of 2-4 robotic arms, each having 6-7 degrees of freedom (DOF) including the arm, wrist and the end effector (the tip of the laparoscopic tool), one of the arms holding a stereo-vision endoscope [Abbeel et al., 2014]. The end effectors are positioned by high-precision motors and are able to reach spaces a human hand cannot [Hannaford and Rosen, 2006], and furthermore development is progressing within flexible end effector tools [Satava et al., 2011,p 74].

(a) Surgeon master console and slave robotic patient manipulator. ***(b)*** The robotic patient manipulator.

Figure 1.2: Example of a master-slave robotic surgical system: the da Vinci.

The 3D visual feedback from the endoscope is sent to the master console, and the control signals for the surgical instrument are generated with the controller joystick, which scales the surgeon's movements down to micro-movements [Hoffman, 2010] steerable through the (zoomed) 3D visual feedback. It also

filters away tremor, and development is made within haptic feedback to the joystick [Satava et al., 2011,p 89], enhancing the surgeon's feel, enabling greater dexterity, accuracy and stability than a human hand.

In the first generations of surgical robotics the master and slave had to be in the same room, but although the feasibility of conducting surgical interventions remotely has been demonstrated, there has not been drivers strong enough to justify its implementation [Satava et al., 2011,p 38]. Experiments and development are made within minimizing and coping with the delays for long-distance telesurgery and within miniaturization and robustness of the surgical robotic systems for use in harsh environments such as war and space.

Robotic surgical procedures are beginning to show superiority to conventional surgery for some procedures, but is still considerably more expensive [Hannaford and Rosen, 2006]. The excessive price is particularly owed to Intuitive Surgical's many patents securing Intuitive the predominant market share with more than 3000 da Vinci Surgical System units installed worldwide (of which 70 % are in the U.S.) [Hoffman, 2010]. Autonomous procedures are still only implemented for entirely pre-planned motions of an operation, and depending on the type of operation not all subtasks in an operation are suited for autonomy [Abbeel et al., 2014; Hannaford et al., 2013].

1.3 Envisioned Future for Robotic Surgery at Aalborg University Hospital

At Aalborg University Hospital robotisized MIS has been implemented since 2008, and now count two da Vinci surgical robots employed at the urology and gynaecological wards, each performing 230 surgeries a year. The authors were granted access to attend a prostatectomy (removal of the prostate gland) at the hospital, during which robot assistant nurse Jane Petersson explicated the spectacle on the monitors revealing the process of gaining access to the cancerous prostate, cleaving way through tissue involving exposure of veins and nerves that must not be cut during the surgery. A view from the surgery is seen in figure 1.3 showing the da Vinci robot reaching into the patient's abdomen.

Figure 1.3: Prostatectomy at Aalborg University Hospital conducted by surgeon Grazvydas Tuckus with the da Vinci Xi robot. The robot arms are wrapped in sterile bags.

Johan Poulsen, chief surgeon at the urology ward and manager of the Center for Minimally Invasive Surgery at Aalborg Univeristy Hospital, concurs with the stance that robotic laparoscopy provides the surgeon with greater dexterity, stability and precision due to the design of the robot tools, the tremor filtering, micro-movement down-scaling and 3D visual overview of the surgical site. It also allows the surgeon a much better work posture than manual laparoscopic surgery and he argues that it is easier to learn operating the robot than manual laparoscopic tools, and with the generations now entering the job market mastering robotic technologies for surgery will come naturally.

At present the da Vinci robot is routinely used in Denmark in procedures within gastroenterology, gynaecology and urology. Dr. Poulsen, who is one of the nationally leading experts within robotic surgery, argues that the next few years will also see robotics applied in surgery of the alimentary tract as well as in otorhinolaryngological, lung and heart surgery, in pace with the purchase and maintenance price of surgical robots going down.

Improvements and Further Development within Surgical Robotics

As described in the preceding, the da Vinci surgery robot is well established nowadays and an immensely important tool in certain types of operations. According to assistant nurse Jane Petersson and Dr. Johan Poulsen, two specific use cases are of great interest in the further expansion of robotic surgery:

- Safety in robotic surgery. As nurse Jane Petersson explains, during surgery the surgical tools are in close contact with nerves and organs that must not be cut under any circumstances. Unfortunately, it happens occasionally, though very rarely. It is of great interest to be able to guarantee a safe operation.
- The possibility to automate or semi-automate certain operations. The operation on a beating heart is a good example of this. If a virtual fixture can be obtained, the need for bypass can be avoided. It is also clear that once automated robots are taken into use, higher demands will be put forth to ensure safety.

According to Dr. Poulsen one of the greatest challenges in operating on a beating heart is the concluding part of the operation suturing the operation site, as the moving tissue is easily pinched or tugged. In order for the advantages of robotisized heart surgery to compensate for the drawbacks of manual bypass operations, it is paramount to have a very exact model of the heart movement. As the heart movement is not a beat as such but rather an expansion and contraction movement propagating from one end of the heart to the other, even a highly complex model will need correction from position measurements of the surface of the heart. He sees it as a viable possibility to mount sensors of up to two by two centimetres by sowing them to the surface of the heart and having a tracking system such as e.g. the Vicon system (Vicon is an indoor tracking system similar to the outdoor GPS) being part of the range of surgical instruments in order to get extremely exact position data for the motion-compensated surgical robot.

Dr. Johan Poulsen suggests a surface mounted on a cylinder, which can be controlled to periodically move up and down, as a first step in testing a surgical robot in following the movement of a surface.

1.4 Configuration of da Vinci at Aalborg University

The configuration at the Control and Automation laboratory at Aalborg University is based on a first generation da Vinci robot, where the patient manipulator is detached from its surgeon controller console

and modified to be controllable by automated processes. As seen in figure 1.2 the da Vinci patient manipulator constitutes four "arms". In this thesis controllers are developed for one of these arms, an overview of the terminology used for the robot outlined in figure 1.4, depicting the distinction of the

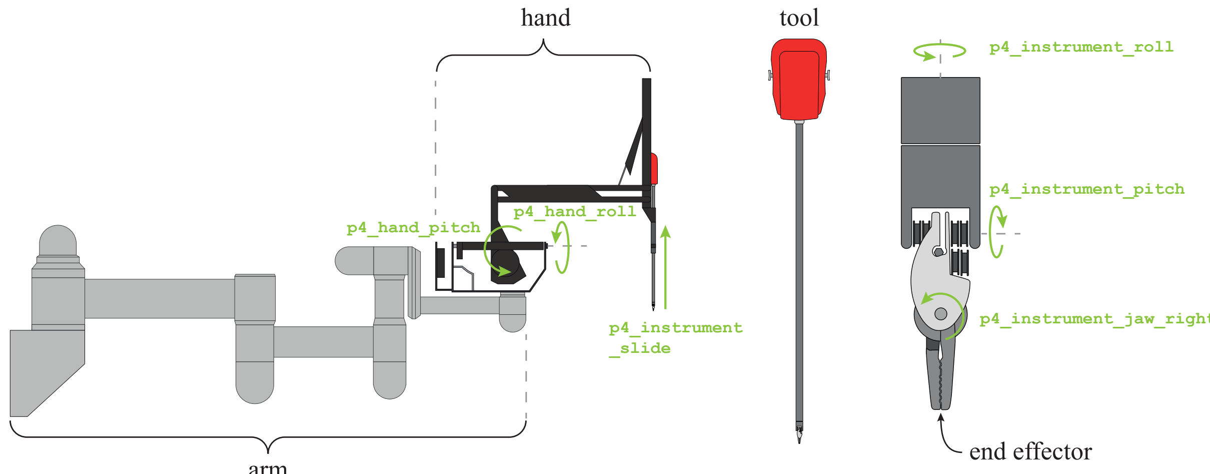


Figure 1.4: Naming convention used for the da Vinci robot.

robotic parts each comprising a number of links connected by joints. The "arm" joints are fixed by electromagnets and are hence uncontrollable, but can be manually released for positioning. The "hand" consists of the controllable dynamic joints (joint naming convention is displayed in green), including the "tool", which is a replaceable instrument worth 10.000 kr only licensed to be used for 10 operations. The outermost point of the instrument is labelled the "end effector" and it is this which should follow a reference point in the developed controllers.

The technical overview presented in figure 1.5 is structured in descending abstraction layers with the highest in the top (i.e. the Robotic Operating System (ROS) - an open source software framework for robots [Quigley et al., 2009], see appendix A for further details), which establishes a wireless TCP/IP communication channel receiving all positions from the robot as feedback and produces position control signals to the NI (National Instruments) single board Reconfigurable Input/Outputs (RIOs) which handle all input/output communication with the user. The NI single board RIOs consist of a primary and a secondary board. The reason for having two RIO boards is solely the lack of input/outputs on one board.

The RIO boards direct the control signals to a cascaded controller taking in a position reference from the user and delivering a current control signal to the ESCON motor driver. The velocity and current controllers are implemented in Field Programmable Gate Array (FPGA) based hardware to ensure sufficient controller speed relative to the system [Wisniewski et al., 2015]. The ESCON motor driver manages advanced processing and essentially delivers an appropriate PWM signal for the actuators (seven Maxon motors) which represent the lowest abstraction layer, located at the bottom of the figure.

The NI single board RIOs concurrently handle most safety precautions and enabling/disabling movement of the arm itself (see appendix B and C for an overview of the arm and its kinematics, respectively) through solenoids. In order to prevent potential violation of physical joint constraints, stricter constraints on each joint position are set in the low-level motor controllers, that disable the controller if exceeded.

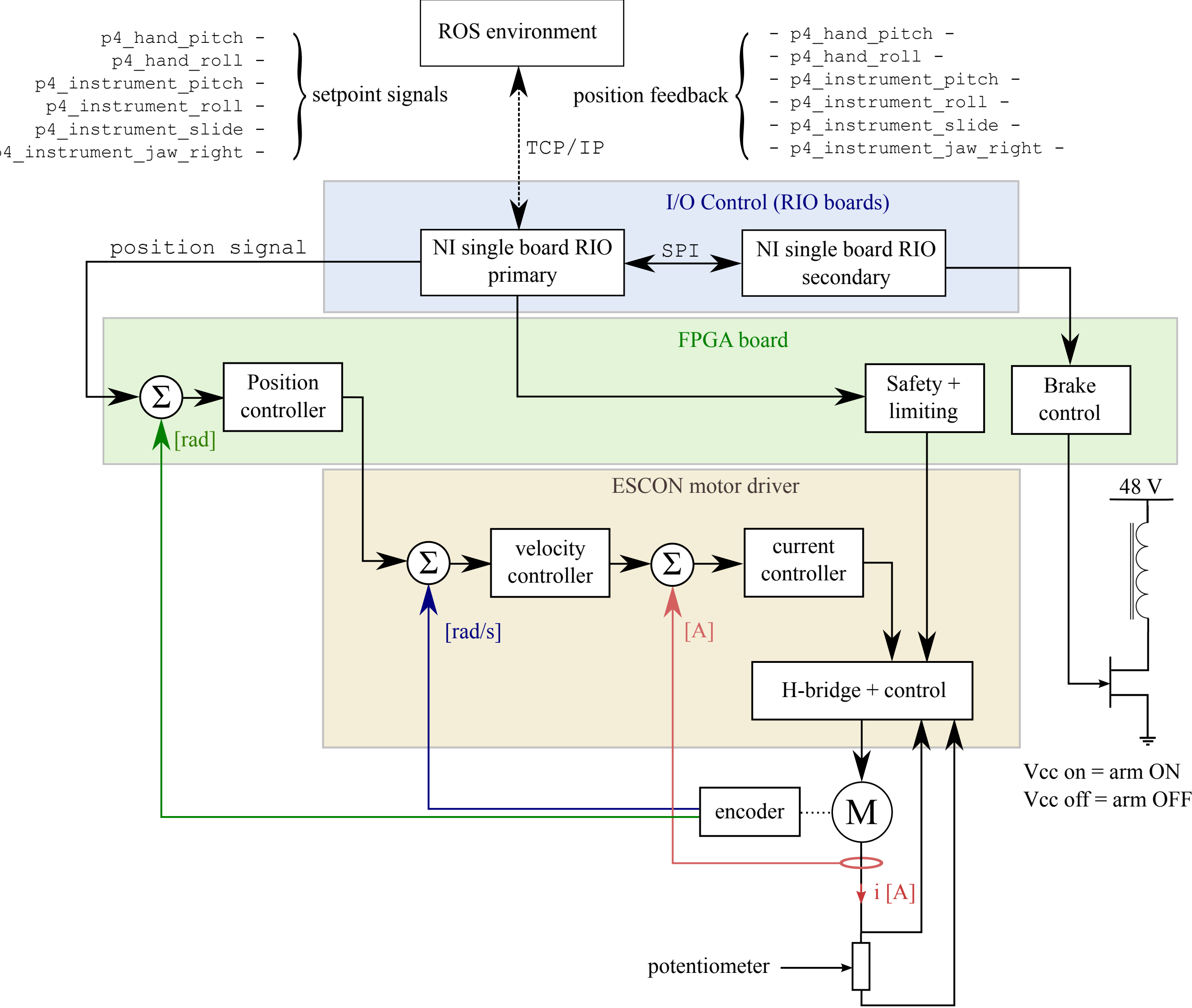


Figure 1.5: Overview of the custom made hardware and controllers for the 1st generation da Vinci surgical robot located at Aalborg University's department of Control and Automation. The setup consist of several layers. It is desired to work solely in the ROS (Robotic Operating System) environment such the underlying layers are left untouched It can be seen that all joint angles can be controlled from the ROS framework.

An introduction to the da Vinci system has at this point be given. Thus the contribution of this project will be outlined in the upcoming subsection.

1.4.1 Overview of Thesis Contribution to the AAU da Vinci System

The focus of this thesis is the highest abstraction layer as seen in the top of figure 1.5, i.e. the ROS environment. The purpose of this layer primarily constitutes the implementation of algorithms that require heavy processing and non real-time processing or tasks with loose timing constraints [Wisniewski et al., 2015]. Given the topics described in the introduction to this project and the desire to automate surgery by means of the da Vinci robot, this thesis will practically and theoretically encompass the tasks outlined in table 1.1.

Problem	Origin
Design a position controller such that it is possible to specify regions where it can be guaranteed that the end effector will never enter.	Provide the surgeon with a feature ensuring that certain regions will never be touched, e.g. veins, organs and similar as explained by assistant nurse Jane Petersson.
Design a position controller taking user input relative to a point on the surface of a beating heart while ensuring that the heart is not penetrated.	Founding automated control on beating hearts where a safe distance to the heart is maintained such that the heart under no circumstances is penetrated thus ensuring a virtual fixture as desired from Dr. Johan Poulsen.
Modelling the system sufficiently without touching the underlying controllers.	The cascaded controllers seen in figure 1.5 are not to be touched. Safety is desired solely from the ROS environment.
Implement the controllers physically on the da Vinci robot. This also implies an understanding of the entire ROS framework	Verify the theory in practice and thereby be a first mover on this topic

Table 1.1: Problems to be solved throughout this thesis and why they are desired to be solved.

The problems given in table 1.1 may be solved in a number of ways and it is not initially obvious which is the more appropriate one. For this reason, it is chosen to bifurcate a solution strategy, thus two approaches are used:

- The design of a safe end effector setpoint controller, utilizing control barrier functions such that the robot is physically prevented from entering unsafe areas, thereby guaranteeing safety in real time [Sloth and Wisniewski, 2014].
- The design of a controller unrestricted from safety considerations. The safety is ensured before the controller is physically implemented by conducting an analysis adjudicating system safety. Thus the controller will be given a *pass* verdict if it does not violate the safety constraints and a *not pass* verdict if it violates predefined safety conditions, thus suggesting an iteration of the controller such that it becomes safe.

The analysis of these two strategies will provide an indication as to which method may be the most appropriate one to use given a specific problem, its complexity taken into account. Consequently, this report presents algorithms, analyses, controllers, software development and formal verification that guarantees safety for automated surgeries. The chapters of this thesis are structured in the following way:

- A certificate to guarantee system safety is established in chapter 2, forming the basis of the analyses in the following chapters.
- In chapter 3 a method to apply the theory from chapter 2 to design a controller that guarantees safety within specified regions is described. This method is used and implemented in the following three chapters, chapter 4 providing an exhaustive example of how to apply the theory to the da Vinci robot. Chapter 5 concerns safety while operating on a beating heart and finally chapter 6 presents safety of the system in three dimensional space. An interim conclusion is given in chapter 7, concluding the need for an easier way to construct the certificates for safety in higher dimensions.

- Accordingly, in chapter 8 the theory behind a rephrasing of the certificate definition from chapter 2 is described, allowing for the application of automated software to search for certificates. This software is described in chapter 9, providing examples of how to apply the software to the use-cases described in chapter 4.

The conclusion to the two approaches is given in the discussion in chapter 10 along with an outlook on future application of the provided theory and implementations.

Chapter 2

Safety Guarantee by Barrier Certificates

A crucial matter when designing a controller for automated operation of robotic surgery tools is the necessity of guaranteed patient safety. The system has to not only be able to prevent the surgery tool from entering certain regions, e.g. penetrating the wall of the heart or cutting an artery, but to guarantee that this cannot happen under any circumstances.

Casting the controller design problem as an optimization problem with constraints, such as Model Predictive Control (MPC), could in principle guarantee that the tool would not enter a predefined area. Indeed, MPC is a method which is very popular at the higher abstraction layers, such as setpoint control [Maciejowski, 2002] which is the case in this specific study. However, most solvers such as the Matlab plugin `cvx` requires convexity in the performance function and its constraints to be able to find a global minimum. This will at best be a lucky special case that unsafe regions can be defined through a convex function. Furthermore, MPC is mostly used in systems with slow dynamics, i.e. dynamics where the time constant is measured in seconds or even minutes [Wang and Boyd, 2010]. This is obviously due to heavy online computations and numerous iterations. Systems containing these time constants are usually thermal systems and not mechanical systems. Additionally, the feasibility of the optimization problem is not very transparent and it is well known that `cvx` is very likely to crash due to infeasibility.

Another very elegant and computationally efficient approach to the safe controller analysis and design problem is the use of barrier certificates, which provide a formal proof of safe operation in infinite time horizon [Prajna et al., 2007; Sloth and Wisniewski, 2014]. This chapter describes the requirements for the construction of barrier certificates along with notation used in relation to these.

2.1 Constraints for a Barrier Certificate

When a barrier certificate can be found for a (closed-loop) dynamical system, the controller is guaranteed to be safe. In the following the notion of safety is defined in order to describe the guarantee extent of a barrier certificate. A general state-space representation of an n-dimensional non-linear system is considered:

$$\dot{\mathbf{x}} = f_{cl}(\mathbf{x}) + h(\mathbf{x})\,\mathbf{d} = f(\mathbf{x}) + g(\mathbf{x})\,\mathbf{u} + h(\mathbf{x})\,\mathbf{d} \tag{2.1}$$

where

- $\mathbf{x}$ is the state, $\mathbf{x}(t) \in \mathbb{R}^n$
- $\mathbf{u}$ is the control input, $\mathbf{u}(t) \in \mathbb{R}^m$
- $\mathbf{d}$ is the disturbance input, $\mathbf{d}(t) \in \mathcal{D} \subseteq \mathbb{R}^p$
- f is a non-linear function, $f : \mathbb{R}^n \to \mathbb{R}^n$
- g is a non-linear function, $g : \mathbb{R}^n \to \mathbb{R}^{n\times m}$
- h is a non-linear function, $h : \mathbb{R}^n \to \mathbb{R}^{n\times p}$

Consider a subspace of the state-space $\mathcal{X} \subseteq \mathbb{R}^n$ defining e.g. the physically feasible states for the system

in equation (2.1). Within this region $\mathcal{X}$, define the two non-intersecting subspaces $\mathcal{X}_u \subset \mathcal{X}$ and $\mathcal{X}_0 \subseteq \mathcal{X}$, defining an unsafe and a safe region, respectively. The unsafe region contains the states which the trajectory of the system must never enter, e.g. for a surgical robot this space could be the collection of veins and organs near the operation site, for which perforation is prohibited. The safe region contains all the states which the trajectory of the system is allowed to and may be required to enter, e.g. the operation site and a region for entering the area in the abdomen. Now safety of a closed-loop control system is given according to [Sloth and Wisniewski, 2014; Prajna et al., 2007] as:

Definition 2.1 (Safety of a System)
Denote a trajectory starting in $x(0) = x_0$ and with bounded disturbance function $\bar{d} : \mathbb{R}_{\geq 0} \to \mathcal{D}$ by $\phi_{x_0}^{\bar{d}}$, defined by

$$\frac{d\phi_{x_0}^{\bar{d}}}{dt} = f_{cl}\left(\phi_{x_0}^{\bar{d}}(t)\right) + h\left(\phi_{x_0}^{\bar{d}}(t)\right)\bar{d}(t) \tag{2.2}$$

The system $\Gamma_{cl} = (f_{cl}, h, \mathcal{X}, \mathcal{X}_0, \mathcal{X}_u, \mathcal{D})$ is unsafe if there exists a $t \in [0,\mathrm{T}]$ such that the trajectory $\phi_{\mathcal{X}_0}^{\bar{d}} : [0,T] \to \mathbb{R}^n$ with initial state $x_0 \in \mathcal{X}_0$ and bounded disturbance function $\bar{d}$ satisfies

$$\left(\phi_{\mathcal{X}_0}^{\bar{d}}([0,t]) \cap \mathcal{X}_u\right) \neq \emptyset \qquad \text{and} \qquad \phi_{\mathcal{X}_0}^{\bar{d}}([0,t]) \subseteq \mathcal{X} \tag{2.3}$$

The system Γ_{cl} is safe if there are no unsafe trajectories.

A graphical interpretation of equation (2.3) is shown in figure 2.1.

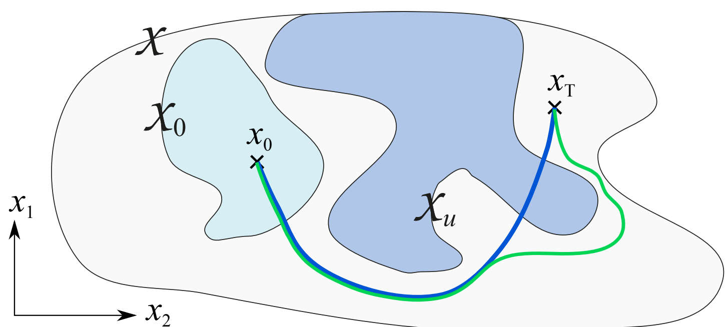


Figure 2.1: Graphical interpretation of equation (2.3) in the state space. The blue trajectory is unsafe because $\left(\phi_{\mathcal{X}_0}^{\bar{d}}([0,t]) \cap \mathcal{X}_u\right) \neq \emptyset$, while the green trajectory is safe.

Disturbances are not considered in the scope of this project, and hence $\mathbf{d} \in \mathcal{D}$ is considered to be zero in the remainder of this thesis.

For the system in equation (2.1) safety can be guaranteed if a barrier certificate for the system exists. A barrier certificate is defined as a function of the system state, satisfying a set of inequalities, entailing that its zero level set in the state space forms a barrier between the safe set of initial states $\mathcal{X}_0$ and the unsafe set $\mathcal{X}_u$, thereby certifying system safety [Prajna et al., 2007].

If a barrier certificate can be defined, safety can be guaranteed for the closed-loop system in the region $\mathcal{X}$, with unsafe region $\mathcal{X}_u$, defined by positive values of the barrier function, and (safe) initial region $\mathcal{X}_0$, defined by non-positive values of the barrier function. In the below the notation $L_{f_{cl}}B(\mathbf{x})$ denotes the

Lie derivative of $B(\mathbf{x})$ along the vector field of the closed-loop system $f_{cl}(\mathbf{x})$, corresponding to the time derivative of the barrier function i.e.

$$L_{f_{cl}}B(\mathbf{x}) = \frac{dB(\mathbf{x})}{d\mathbf{x}} f_{cl}(\mathbf{x}) = \frac{dB(\mathbf{x})}{d\mathbf{x}} \frac{d\mathbf{x}(t)}{dt} = \frac{dB(\mathbf{x}(t))}{dt} \tag{2.4}$$

Requiring that the time derivative of the barrier function must be nonpositive on the entire set $\mathcal{X}$ (see Definition 2.2) corresponds to the value of the barrier function decreasing over time, hence seeking the minimum of the (convex) barrier certificate. Requiring the trajectory of the state to start within the safe set $\mathcal{X}_0$ this means that the trajectory will never cross the zero level set and enter the unsafe set $\mathcal{X}_u$, as this set only contains values of the barrier function larger than the initial value.

Definition 2.2 (Barrier Certificate)
If a barrier certificate can be constructed as a continuous and differentiable function $B(\mathbf{x}) : \mathcal{X} \to \mathbb{R}$ adhering to the following inequalities [Prajna et al., 2007]:

$$B(\mathbf{x}) \leq 0 \qquad \forall\ \mathbf{x} \in \mathcal{X}_0 \tag{2.5)a}$$
$$B(\mathbf{x}) > 0 \qquad \forall\ \mathbf{x} \in \mathcal{X}_u \tag{2.5)b}$$
$$L_{f_{cl}}B(\mathbf{x}) \leq 0 \qquad \forall\ \mathbf{x} \in \mathcal{X} \tag{2.5)c}$$

Then safety of the closed-loop system $f_{cl}(\mathbf{x})$, as defined in Definition 2.1, is guaranteed.

From equation (2.5) it can be seen that the function $B(\mathbf{x})$ must be constructed such that its zero level set delimits and separates the safe and the unsafe regions, while the Lie derivative constraint imposes that the derivative $dB(\mathbf{x})/d\mathbf{x}$ must have the opposite sign of the state derivative $d\mathbf{x}/dt$ for any state within the region $\mathcal{X}$, where $B(\mathbf{x})$ is defined. Note how according to equation (2.5)c the barrier certificate requires mere stability and not asymptotic stability ($L_{f_{cl}}B(\mathbf{x}) < 0$) of the system trajectory. This is rarely enough when dealing with physical systems, however, mathematically it is sufficient.

Furthermore from equation (2.5)c it is deduced that a controller incorporating the barrier certificate in its design will ensure stability if $B(\mathbf{x})$ has a finite minimum value. This entails that $B(\mathbf{x})$ is radially unbounded if $\mathcal{X}$ encompasses the entire state-space:

$$\lim_{\mathbf{x}\to\pm\infty} B(\mathbf{x}) = \infty \qquad \text{if} \qquad \mathcal{X} = \mathbb{R}^n \tag{2.6}$$

Nexus to Lyapunov Functions

As it can be seen from equation (2.5) the definition of a barrier certificate strongly resembles that of a Lyapunov function, and indeed the Lie derivative nonpositivity constraint is identical to the time derivative constraint to a Lyapunov function $V(\mathbf{x})$, a Lyapunov candidate function for a stable system given by

$$V(\mathbf{x}) > 0 \qquad \forall\ \mathbf{x} \in \mathbb{R} \setminus \{0\} \tag{2.7)a}$$
$$\dot{V}(\mathbf{x}) \leq 0 \qquad \forall\ \mathbf{x} \in \mathbb{R} \tag{2.7)b}$$

and for a system with an asymptotically stable equilibrium in $\mathbf{x} = 0$, equation (2.7)b is replaced by

$$\dot{V}(\mathbf{x}) < 0 \qquad \forall\ \mathbf{x} \in \mathbb{R} \setminus \{0\} \tag{2.7)c}$$

As such a barrier certificate can be seen as an offset Lyapunov function with negative values in the safe region. The stable focus may also be offset from $\mathbf{x} = 0$. However, a barrier function may also take other (non-convex) forms.

2.2 Approaches to the Problem of Guaranteeing System Safety

Two approaches to the problem of guaranteeing safety of a system through the construction of barrier certificates are used in the following: design of safe controllers and safety verification of control systems.

In chapter 3 a method of designing guaranteed safe controllers from barrier certificates is described, based on [Wieland and Allgöwer, 2007]. This method is used in chapters 4 through 6, where barrier certificates are constructed by hand, and guaranteed safe controllers are designed and implemented on the da Vinci surgical robot. The safe controller is used in combination with a linear state space controller designed through pole placement for setpoint control in the safe region. In chapter 4 and 5 system models of different orders are considered in 1D Cartesian space, with static and dynamic boundaries (zero level sets) of the barrier function, respectively, while in chapter 6 a system model is considered in 3D Cartesian space.

In chapter 8 a recasting of Definition 2.2 according to [Lasserre, 2009] is presented, allowing for automated construction of barrier certificates with an existing software toolbox for MATLAB. When a barrier certificate can be found using this toolbox, system safety is hereby certified. In chapter 9 this method of safety verification is applied to linear position-control systems corresponding to the linear state space controllers presented in chapter 4.

Chapter 3

Controller Design from CBFs

This chapter lays the basics of all shared control theory applied in the following chapters dealing with the design of a safe controller.

Based on [Artstein, 1983], who founded Control Lyapunov Functions (CLFs) based on Lyapunov functions described in equation (2.7), a Control Barrier Function (CBF) can be created according to [Wieland and Allgöwer, 2007] from where the definition below is stated:

Definition 3.1 (Control Barrier Function)
Given a system $\dot{\mathbf{x}} = f(\mathbf{x}) + g(\mathbf{x})\mathbf{u}$, the function $B : \mathbb{R}^n \to \mathbb{R}$ is a CBF if the below constraints are fulfilled:

$$\mathbf{x} \in \mathcal{X}_u \quad \Rightarrow \quad B(\mathbf{x}) > 0 \tag{3.1)a}$$

$$L_g B(\mathbf{x}) = 0 \quad \Rightarrow \quad L_f B(\mathbf{x}) < 0 \tag{3.1)b}$$

$$\{\mathbf{x} \in \mathcal{X} \mid B(\mathbf{x}) \leq 0\} \neq \emptyset \tag{3.1)c}$$

where

$B(\mathbf{x})$ is a control barrier function

$L_f B(\mathbf{x})$ is the Lie derivative of $B(\mathbf{x})$ along the vector field $f(\mathbf{x})$, i.e. $\frac{dB(\mathbf{x})}{d\mathbf{x}} f(\mathbf{x})$

$L_g B(\mathbf{x})$ is the Lie derivative of $B(\mathbf{x})$ along the vector field $g(\mathbf{x})$, i.e. $\frac{dB(\mathbf{x})}{d\mathbf{x}} g(\mathbf{x})$

The requirement in equation (3.1)b can be hard to fulfill, and can be replaced with a relaxed constraint to obtain a weak CBF:

$$L_g B(\mathbf{x}) = 0 \quad \Rightarrow \quad L_f B(\mathbf{x}) \leq 0 \tag{3.1)d}$$

Note how Definition 3.1 put forth demands for the open loop system as opposed to Definition 2.2 which describes the closed loop system. Equation (3.1)a essentially states the same as equation (2.5)b, i.e. when $B(\mathbf{x}) > 0$ then $\mathbf{x}$ is in the unsafe region. This makes it possible to design both the unsafe and the safe region by shaping $B(\mathbf{x})$. Equation (3.1)b puts forth the requirement that the gradient along the vector field $f(\mathbf{x})$ must point away from the unsafe area, bounded by the zero level set of the barrier function, whenever the state cannot be controlled by the input (except in the critical point as the system is in its equilibrium at this point). Equation (3.1)c simply states that the safe area must contain some states as control otherwise is impossible.

3.1 The Control Law

It is convenient to divide the control law into two controllers. A controller that is used in an area close to the unsafe region and a controller used in the safe region. The reason for this divided control law is

the ability to apply linear control theory in the safe area (the system can be well approximated as a linear second order system, see section 4.1) and thereby make use of the benefits from linear control. Thus the safe set $\mathcal{X}_0$ is divided into two sections: a transition space $\mathcal{T}$ being the subset of the safe region closest to the unsafe set where safety control should be applied, and the remaining safe region $\mathcal{Y}$ where linear control can be applied, thus introducing a divided control law:

$$u(\mathbf{x}) = \begin{cases} \tilde{u}(\mathbf{x}) & \text{if} \quad \mathbf{x} \in \mathcal{Y} \subset \mathcal{X}_0 \\ k_0(\mathbf{x}) & \text{if} \quad \mathbf{x} \in \mathcal{T} = \mathcal{X}_0 \setminus \mathcal{Y} \end{cases} \tag{3.2}$$

where

$\tilde{u}(\mathbf{x})$ is the non-safe controller applied on $\mathcal{Y}$ well within the safe region, $\tilde{u}(\mathbf{x}) \in \mathbb{R}^m$

$k_0(\mathbf{x})$ is a controller guaranteeing system safety, applied in the transition space $\mathcal{T}$ close to the unsafe set, $k_0(\mathbf{x}) \in \mathbb{R}^m$

For a linear system, the non-safe controller can be determined by linear state feedback:

$$\tilde{u}(\mathbf{x}) = \bar{\mathbf{N}}\mathbf{x}_{\text{ref}} - \mathbf{K}\mathbf{x} \tag{3.3}$$

where

$\mathbf{K}$ is a constant feedback matrix for a system with n states and m inputs, $\mathbf{K} \in \mathbb{R}^{m \times n}$

$\bar{\mathbf{N}}$ is a constant to ensure unity gain from reference to output, $\bar{\mathbf{N}} \in \mathbb{R}^{m \times m}$

$\mathbf{x}_{\text{ref}}$ is the position reference, $\mathbf{x}_{\text{ref}} \in \mathbb{R}^m$

$\mathbf{x}$ is the state vector, $\mathbf{x} \in \mathbb{R}^n$

The two controllers can be combined as a linear combination determined by a parameter $\sigma(\mathbf{x}) \in [0,1]$ [Wieland and Allgöwer, 2007]. This ensures that the switch between the two controllers occurs with less fluctuations.

$$u(\mathbf{x}, \tilde{u}) = \sigma(\mathbf{x})k_0(\mathbf{x}) + (1 - \sigma(\mathbf{x}))\tilde{u}(\mathbf{x}) \tag{3.4}$$

Note the two extremities of $\sigma(\mathbf{x})$:

$$\sigma(\mathbf{x}) = \begin{cases} 0 & \Rightarrow \quad \text{Pure control by pole placement, i.e. } u(\mathbf{x}) = \tilde{u}(\mathbf{x}) = \bar{\mathbf{N}}\mathbf{x}_{\text{ref}} - \mathbf{K}\mathbf{x} \\ 1 & \Rightarrow \quad \text{Pure safety control i.e. } u(\mathbf{x}) = k_0(\mathbf{x}) \end{cases}$$

The interval between 0 and 1 can be refined such that the transition between the two control laws is not instantaneous. This smoothing can be performed with a smooth approximation of the unit step (a bump function) of $B(\mathbf{x})$ by introducing a scalar $\varepsilon > 0$ [Wieland and Allgöwer, 2007]:

$$\sigma(\mathbf{x}) = \begin{cases} 0 & \text{if} \quad B(\mathbf{x}) \leq -\varepsilon \\ -2\left(\frac{B(\mathbf{x})}{\varepsilon}\right)^3 - 3\left(\frac{B(\mathbf{x})}{\varepsilon}\right)^2 + 1 & \text{if} \quad B(\mathbf{x}) \in (-\varepsilon, 0) \\ 1 & \text{if} \quad B(\mathbf{x}) \geq 0 \end{cases} \tag{3.5}$$

A block diagram of a linear closed loop system with control input as described in equation (3.4) is depicted in figure 3.1.

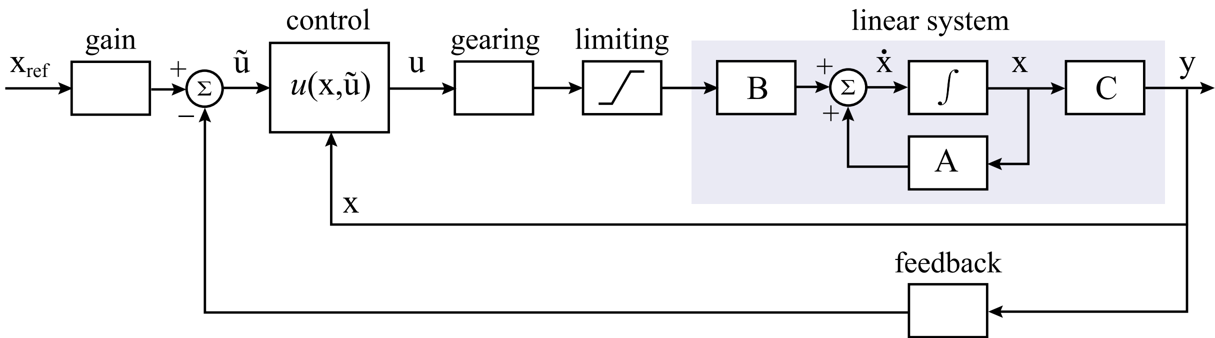


Figure 3.1: Block diagram of the control system. The limiter limits the control signal such that it does not exceed the physical boundaries of the da Vinci robot. The gearing ensures that there is a 1:1 mapping between the control signal and the physical position, i.e. meters for prismatic joints and radians for revolute joints.

3.1.1 Uniform Construction of the Safety Controller $k_0(\mathbf{x})$

The control law ensuring safety can be found as [Wieland and Allgöwer, 2007]:

$$k_0(\mathbf{x}) = \begin{cases} -\dfrac{L_f B(\mathbf{x}) + \sqrt{(L_f B(\mathbf{x}))^2 + \kappa^2 L_g B(\mathbf{x})(L_g B(\mathbf{x}))^T}}{L_g B(\mathbf{x})(L_g B(\mathbf{x}))^T}(L_g B(\mathbf{x}))^T & \text{if} \quad L_g B(\mathbf{x}) \neq 0 \\ 0 & \text{if} \quad L_g B(\mathbf{x}) = 0 \end{cases} \tag{3.6}$$

where κ is a design variable. High values of κ implies increased controller aggressiveness. Equation (3.6) indeed ensures safety for the closed loop system $\dot{\mathbf{x}} = f(\mathbf{x}) + g(\mathbf{x})k_0(\mathbf{x})$. This is easily proven as:

$$L_{f_{cl}} B(\mathbf{x}) = L_f B(\mathbf{x}) + L_g B(x) k_0(\mathbf{x})$$

For $L_g B(\mathbf{x}) \neq 0$:

$$\begin{aligned} L_{f_{cl}} B(\mathbf{x}) &= L_f B(\mathbf{x}) + L_g B(\mathbf{x}) \left(-\frac{L_f B(\mathbf{x}) + \sqrt{(L_f B(\mathbf{x}))^2 + \kappa^2 L_g B(\mathbf{x})(L_g B(\mathbf{x}))^T}}{L_g B(\mathbf{x})(L_g B(\mathbf{x}))^T}(L_g B(\mathbf{x}))^T \right) \\ &= L_f B(\mathbf{x}) - L_g B(\mathbf{x})(L_g B(\mathbf{x}))^T \frac{L_f B(\mathbf{x}) + \sqrt{(L_f B(\mathbf{x}))^2 + \kappa^2 L_g B(\mathbf{x})(L_g B(\mathbf{x}))^T}}{L_g B(\mathbf{x})(L_g B(\mathbf{x}))^T} \\ &= L_f B(\mathbf{x}) - L_f B(\mathbf{x}) - \sqrt{(L_f B(\mathbf{x}))^2 + \kappa^2 L_g B(\mathbf{x})(L_g B(\mathbf{x}))^T} \\ &= -\sqrt{(L_f B(\mathbf{x}))^2 + \kappa^2 L_g B(\mathbf{x})(L_g B(\mathbf{x}))^T} \quad \leq 0 \quad \forall \quad \mathbf{x} \end{aligned}$$

As all terms within the square root are squared, no imaginary numbers occur, and as a result $L_{f_{cl}} B(\mathbf{x})$ will always be nonpositive when $L_g B(\mathbf{x}) \neq 0$. According to equation (3.6), when $L_g B(\mathbf{x}) = 0$:

$$L_{f_{cl}} B(\mathbf{x}) = L_f B(\mathbf{x}) + L_g B(\mathbf{x}) \cdot 0 = L_f B(\mathbf{x})$$

As $B(\mathbf{x})$ is constructed such that whenever $L_g B(\mathbf{x}) = 0$ then it is always true that $L_f B(\mathbf{x}) < 0$, it is thereby verified that $Lf_{cl} B(\mathbf{x}) \leq 0$ for all $\mathbf{x} \in \mathcal{X}$.

3.2 Using CBFs in the Control Design for the da Vinci Robot

The theory presented in this chapter allows a way to construct a controller such that safety is guaranteed, if the constraints on the CBF in Definition 3.1 are obeyed. It may, however, be noted that when the controller is transferred to a digital system, the certificate is built upon an infinite sampling rate as the certificates are described in continuous time. Adopting the certificate to a discrete system is indeed a challenge worth respecting.

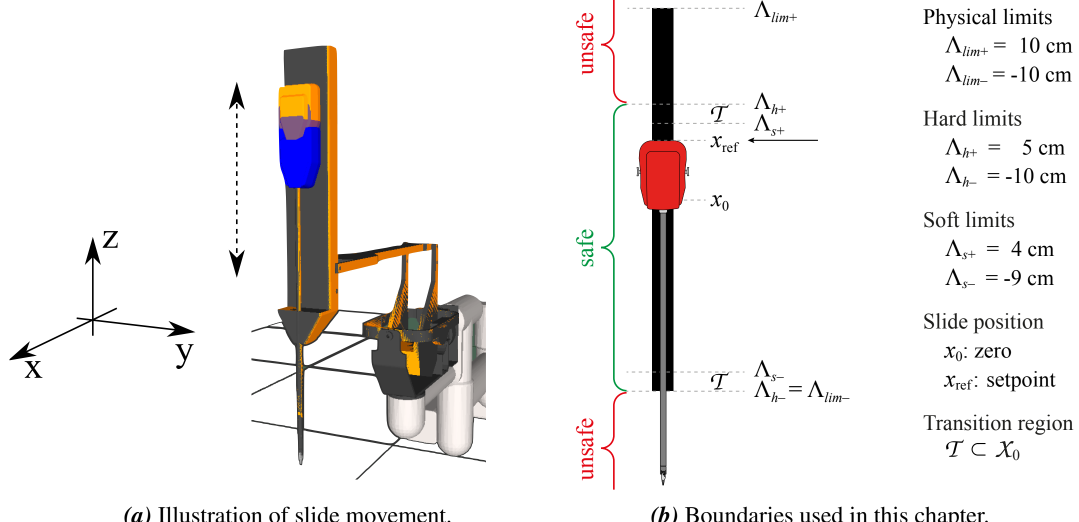
CBFs will be used in the following chapters to design safe controllers for the da Vinci surgical robot. In chapter 4 a controller is designed that ensures safety for the slide movement of the robot (sliding the tool up and down in one dimension, see figure 1.4) thereby illustrating the usefulness of the theory. In chapter 5 the theory is used to establish the basics for surgery on a beating heart and in chapter 6 the system is expanded to 3D Cartesian space including all the controllable joints of the robot. For the ROS implementation of the controllers, the reader is referred to appendix A.3.

Chapter 4

Founding Safety with Static Boundaries

This chapter intends to implement and analyse a controller ensuring safety if the demands from Definition 3.1 are obeyed. This shall first be tested on the slide movement on the da Vinci surgical robot as it comprises a prismatic joint and a 1:1 mapping from slide *joint_angle* to 1D position. Hence any inverse kinematics solver can be bypassed in the early phase of this project which is an important simplification to eliminate initial complications.

The slide movement is visualized in figure 4.1a and an overview of terms used in this section is found in figure 4.1b, which also encompasses the case study considered in this chapter. It puts forth the demands that the upper slide region, i.e. the interval $[\Lambda_{h+}, \Lambda_{\text{lim+}}]$ is an unsafe area and the rest is considered safe. Furthermore, everything outside the slide physical limits, i.e. $[-\infty, \Lambda_{\text{lim-}}]$ and $[\Lambda_{\text{lim+}}, \infty]$ is also considered unsafe. This case study is purely made up with the purpose to demonstrate the use of a safety controller.



(a) Illustration of slide movement. ***(b)*** Boundaries used in this chapter.

Figure 4.1: The slide position of the robotic instrument is visualized for the instrument house. As the remaining robot joints are not considered in this chapter, there is a one-to-one correspondence between instrument house position and instrument tip position. Slide house position in x_0 corresponds to tool tip position in zero in the z-dimension.

The boundaries for the CBF sets in slide position are summed up in table 4.1.

$\mathcal{X}$	$\mathcal{X}_u$	$\mathcal{X}_0$
$\mathcal{X} = \{\mathbf{x} \in [\Lambda_{\text{lim}-}, \Lambda_{s-}] \cup [\Lambda_{s+}, \Lambda_{\text{lim}+}]\}$	$\mathcal{X}_u = \{\mathbf{x} \in [\Lambda_{\text{lim}-}, \Lambda_{h-}] \cup [\Lambda_{h+}, \Lambda_{\text{lim}+}]\}$	$\mathcal{X}_0 = \{\mathbf{x} \in [\Lambda_{h-}, \Lambda_{s-}] \cup [\Lambda_{s+}, \Lambda_{h+}]\}$

Table 4.1: CBF state intervals for the robotslide position, with limits as given in figure 4.1b i.e. Λ_{lim} is the physical slide limit (± 0.1 m), Λ_s is a soft limit denoting a transition line and Λ_h is a hard limit where a trajectory at all cost can not cross.

The interval $\mathcal{Y} = \{\mathbf{x} \in [\Lambda_{s-}, \Lambda_{s+}]\}$ is safe thus $\tilde{u}(\mathbf{x})$ stated in equation (3.3) can be used in this region. As the control law stated in equation (3.4) utilizes Lie derivatives, a system model is required before any controller design may be initiated.

4.1 Modelling of Slide Movement

To obtain a model of the slide movement (along the 3D z-axis), the step response will be measured. This can be done by subscribing to the `joint_state` topic in ROS (topics are ROS syntax for communication lines, see appendix A for an introduction to ROS). The experiment is described in further details in appendix F, and the result is plotted in figure 4.2.

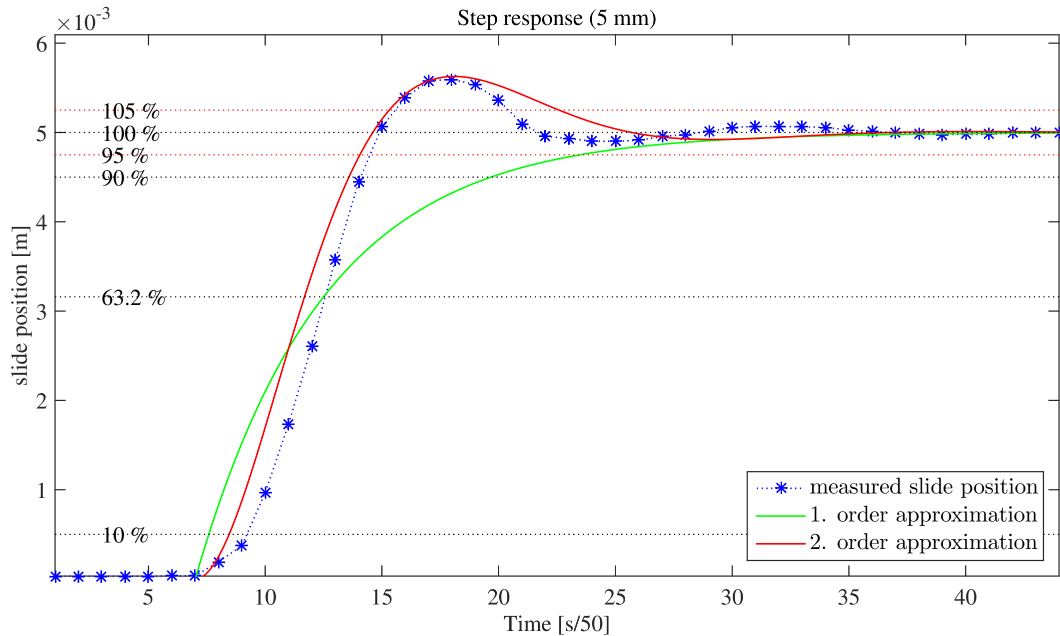


Figure 4.2: Step response from 0 mm to 5 mm. Plot details and measurements can be found in appendix J as `matlab_scripts/slide_step/plot_slide_pos.m`. The experiment is described in appendix F.

The system can clearly be approximated with an underdamped second order model. For initial simplicity, however, a simple first order model of the slide movement is used, followed by the second order system approximation. This introduces a number of other challenges which is the reason for initial simplicity. These models of the robot slide movement will throughout this chapter be referred to as the first and second order models, and are treated in the following way:

- The first order system model based on *position* only is presented in subsection 4.1.1. Its CBF is constructed in subsection 4.2.1, the controller designed in subsection 4.3.1, subsection 4.4.1 shows the MATLAB implementation and finally the implementation on the da Vinci robot is presented in subsection 4.5.1.
- The second order system model based on *position and velocity* is presented in subsection 4.1.2. Its CBF is constructed in subsection 4.2.2, the controller designed in subsection 4.3.2, its MATLAB implementation is presented in subsection 4.4.2 and finally subsection 4.5.2 documents the implementation on the da Vinci robot.

4.1.1 1D First Order Model based on Position

The system can be approximated with a linear first order system with a time constant τ. The time constant is read from figure 4.2 as the time lapse from the step is applied until the state has travelled 63.2 % of the distance to the reference:

$$\tau_s = 110\,\text{ms}$$

A linear system can be outlined as:

$$\begin{aligned} Y(s) &= \frac{1}{\tau_s s + 1} U(s) \\ &= \frac{1/\tau_s}{s + 1/\tau_s} U(s) = (s + 1/\tau_s)^{-1}\, 1/\tau_s\, U(s) \qquad \underset{\text{compare to obtain SS form}}{\overrightarrow{Y(s) = (\mathbf{C}(s\mathbf{I}-\mathbf{A})^{-1}\mathbf{B}+\mathbf{D})U(s)}} \\ \dot{x} &= \underbrace{-\tau_s^{-1} x}_{\mathbf{A}x} + \underbrace{\tau_s^{-1}}_{\mathbf{B}} u \end{aligned} \tag{4.1}$$

The system matrix **A**, input matrix **B**, output matrix **C** and feedthrough matrix **D** can be read from the above equation:

$$\mathbf{A} = -\tau_s^{-1}, \qquad \mathbf{B} = \tau_s^{-1}, \qquad \mathbf{C} = 1 \quad \text{and} \quad \mathbf{D} = 0 \tag{4.2}$$

This completes the first order approximation.

4.1.2 1D Second Order Model based on Position and Velocity

The second order approximation has the form:

$$\frac{Y(s)}{U(s)} = \frac{\omega_n^2}{s^2 + 2\zeta\omega_n s + \omega_n^2} \tag{4.3}$$

where

$Y(s)$	is the output in the Laplace domain	[m]
$U(s)$	is the input in the Laplace domain	[m]
ω_n	is the natural frequency of the system	[rad/s]
ζ	is the damping coefficient	[·]
s	is the Laplace operator	[rad/s]

The model can unambiguously be approximated from the rise time t_r, settling time t_s (5 % settling time) and the overshoot M_p [Franklin et al., 2010, pp. 134-136]. They are measured from figure 4.2 with the purpose to find ω_n and ζ:

$$\begin{aligned} \omega_n &= \frac{1.8}{t_r} = \frac{1.8}{0.106\,\text{s}} = 17\,\text{rad/s} \\ \zeta &= \frac{-1}{\omega_n \cdot t_s} \log(5\,\%) = \frac{-1}{17 \cdot 0.320} \log(0.05) = 0.55 \end{aligned}$$

Equation (4.3) can be transformed into state space form:

$$Y(s)s^2 + 2\zeta\omega_n Y(s)s + \omega_n^2 Y(s) - \omega_n^2 U(s) = 0$$

$$\ddot{y}(t) + 2\zeta\omega_n\dot{y}(t) + \omega_n^2 y(t) - \omega_n^2 u(t) = 0$$

Choose $y(t) = x_1(t)$ to represent the position, and let $\dot{x}_1(t) = x_2(t)$

$$\begin{bmatrix} \dot{x}_1(t) \\ \dot{x}_2(t) \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -\omega_n^2 & -2\zeta\omega_n \end{bmatrix} \begin{bmatrix} x_1(t) \\ x_2(t) \end{bmatrix} + \begin{bmatrix} 0 \\ \omega_n^2 \end{bmatrix} u(t) \tag{4.4)a$$

$$y(t) = \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} x_1(t) \\ x_2(t) \end{bmatrix} \tag{4.4)b$$

where

$x_1(t)$	is the position	[m]
$x_2(t)$	is the velocity	[m/s]
$y(t)$	is the output (slide position)	[m]
$u(t)$	is the control input	[·]

Thus the linear system matrices are:

$$\mathbf{A} = \begin{bmatrix} 0 & 1 \\ -\omega_n^2 & -2\zeta\omega_n \end{bmatrix}, \qquad \mathbf{B} = \begin{bmatrix} 0 \\ \omega_n^2 \end{bmatrix}, \qquad \mathbf{C} = \begin{bmatrix} 1 & 0 \end{bmatrix} \quad \text{and} \quad \mathbf{D} = 0 \tag{4.5}$$

Which completes the second order model.

4.2 Construction of CBF

To illustrate the usefulness of CBFs, a palpable example hereof will be created with direct application to the da Vinci robot. This example does not directly constitute application to a patient but favour the theory in a neat and comprehensible sense and secure a way to visually and physically verify the method.

4.2.1 Construction of CBF Based on the First Order Model

In this subsection, the state vector $x \in \mathbb{R}$ consists of the position only. Thus, a parabola is now introduced as CBF as it allows an easy way to define $\mathcal{X}_u$ and $\mathcal{X}_0$ from table 4.1.

$$B(x) = ax^2 + bx + c \tag{4.6}$$

The parameters a, b and c can be easily chosen to fulfil the requirements in equation (2.5)a and (2.5)b for a barrier function, thereby fulfilling the parallel requirements for the CBF in equation (3.1)a and (3.1)c. From equation (3.1)b it is required that either $L_gB(x) \neq 0 \ \forall x \in \mathcal{X}$, or that $L_fB(x) < 0$ when $L_gB(x) = 0$, as the input in that case will not have any influence on the state. Analysing $L_gB(x) = 0$

$$L_gB(x)\Big|_{g(x)=\mathbf{B}} = (2ax+b)\cdot\tau^{-1} = 0 \qquad \Rightarrow \qquad x = \frac{-b}{2a}$$

it is seen that this is only the case in $x = \frac{-b}{2a}$ which is indeed the critical point for a one dimensional parabola. As is the case for Lyapunov functions (see equation (2.7)c) in the critical point the requirement on the derivative is relaxed to $L_fB(x) \leq 0$.

$$L_fB(x) = \frac{d}{dx}B(x)f(x)\Big|_{f(x)=\mathbf{A}x} = (2ax+b)(-\tau^{-1}x) = -2\tau^{-1}ax^2 - \tau^{-1}bx$$

$$L_fB(x)\Big|_{x=\frac{-b}{2a}} = -2\tau^{-1}a\left(\frac{-b}{2a}\right)^2 - \tau^{-1}b\left(\frac{-b}{2a}\right) = 0$$

Hence the CBF is valid for all choices of a,b. The scalar c must be less than zero to comply with equation (3.1)c. At this point in time, three equations with three unknowns can be outlined to fulfil the initial demand in figure 4.1b. The value of $B(x)$ in the vertex of the parabola is within the safe region, and can thus be chosen as any *negative* real number, here chosen to be -0.025.

$$\left.\begin{aligned} a\Lambda_{h+}^2 + b\Lambda_{h+} + c = 0 \\ a\Lambda_{h-}^2 + b\Lambda_{h-} + c = 0 \\ a\left(\frac{\Lambda_{h-}+\Lambda_{h+}}{2}\right)^2 + b\left(\frac{\Lambda_{h-}+\Lambda_{h+}}{2}\right) + c = -0.025 \end{aligned}\right\} \quad \begin{aligned} a &= 1.7778 \\ b &= 0.0889 \\ c &= -0.0089 \end{aligned}$$

The CBF is plotted in figure 4.3 from which it is seen that the demands from table 4.1 are fulfilled.

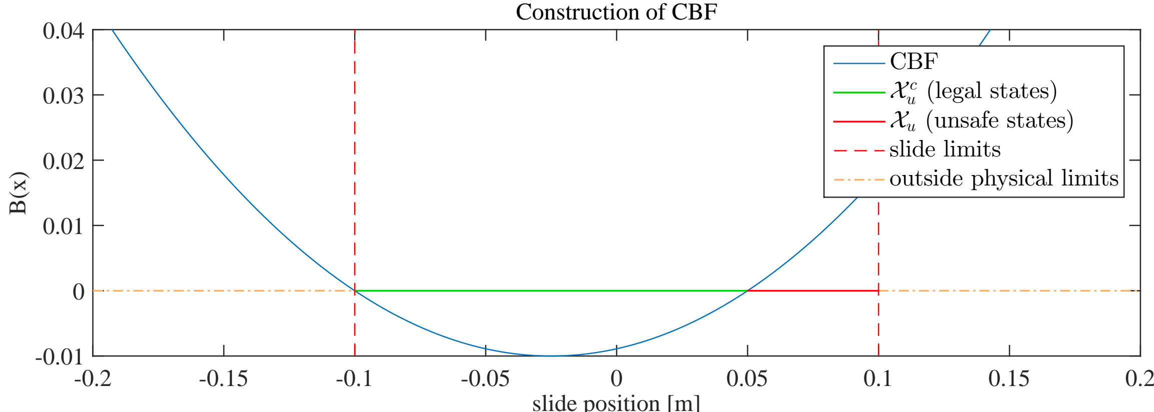


Figure 4.3: Barrier function shown along with the $\mathcal{X}_u$ and $\mathcal{X}_u^c$. Plot details and MATLAB script can be found in appendix J as `matlab_scripts/plot_parabola/plot_parabola.m`

4.2.2 Construction of CBF Based on the Second Order model

Consider now for the second order system the same candidate CBF as given in equation (4.6). Note that $B(\mathbf{x})$ is a function of position only, such that the CBF is:

$$B(\mathbf{x}) = ax_1^2 + bx_1 + c$$

For this system model the Lie derivative $L_gB(\mathbf{x}) = 0\ \forall \mathbf{x}$:

$$L_gB(\mathbf{x}) = \frac{dB(\mathbf{x})}{d\mathbf{x}}g(\mathbf{x})\Big|_{g(\mathbf{x})=\mathbf{B}} = \begin{bmatrix}\frac{\partial B(\mathbf{x})}{\partial x_1} & \frac{\partial B(\mathbf{x})}{\partial x_2}\end{bmatrix}\begin{bmatrix}0 \\ \omega_n^2\end{bmatrix} = 0$$

This puts forth the requirement that $L_fB(\mathbf{x}) < 0\ \forall \mathbf{x}$. Accordingly:

$$\begin{aligned} L_fB(\mathbf{x}) = \frac{dB(\mathbf{x})}{d\mathbf{x}}f(\mathbf{x})\Big|_{f(\mathbf{x})=\mathbf{A}\mathbf{x}} &= \begin{bmatrix}\frac{\partial B(\mathbf{x})}{\partial x_1} & \frac{\partial B(\mathbf{x})}{\partial x_2}\end{bmatrix}\begin{bmatrix}0 & 1 \\ -\omega_n^2 & -2\zeta\omega_n\end{bmatrix}\begin{bmatrix}x_1 \\ x_2\end{bmatrix} \\ &= (2ax_1+b)x_2 - 0\cdot(\omega_n^2x_1 + 2\zeta\omega_n x_2) \\ &= (2ax_1+b)x_2 \end{aligned}$$

As the velocity can be both decreasing and increasing for all positions, this demand is impossible to fulfil with this candidate CBF and it is therefore invalid. A solution is to include the velocity in the barrier function.

Safety constraints on velocity are not of any significant importance as such, but they are necessary for $L_g B(\mathbf{x})$ to obtain values different from zero as opposed to the Lie derivative of the invalid CBF presented above. Consider instead the elliptic paraboloid as CBF:

$$B(\mathbf{x}) = \left(\frac{(x_1 - x_{10})^2}{a_2^2} + \frac{(x_2 - x_{20})^2}{b_2^2} \right) c_1 + c_2 \tag{4.7}$$

where

x_1	is the position	[m]
x_2	is the velocity	[m/s]
x_{10}	is the extremity point for x_1 and thereby equilibrium point for an upward paraboloid	[m]
x_{20}	is the extremity point for x_2 and thereby equilibrium point for an upward paraboloid	[m/s]
a_2	is a constant that dictates the level of curvature in the $x_1 - B(\mathbf{x})$ plane	$[\cdot]$
b_2	is a constant that dictates the level of curvature in the $x_2 - B(\mathbf{x})$ plane	$[\cdot]$
c_1	is a constant that dictates if the paraboloid points upward ($c_1 > 0$) or downward ($c_1 < 0$)	$[\cdot]$
c_2	is a constant that dictates the offset in $B(\mathbf{x})$ axis	$[\cdot]$

The elliptic paraboloid allows constraints on both position and velocity. To ensure that the position demands from table 4.1 are still fulfilled and are so for all possible velocities (constrained by the slide movement's physical limits), the below values are chosen:

$$x_{10} = \frac{\Lambda_{h-} + \Lambda_{h+}}{2} = -0.025$$
$$x_{20} = 0$$
$$M_0 = 4$$

where M_0 denotes the semimajor axis of the zero level set of the CBF. Note that x_{20} ensures velocity equilibrium in 0 m/s and note that the velocity outermost points are determined far bigger than the robot's physical limits to ensure that all position values on the interval $[-0.1 \;\; 0.05]$ are considered safe (almost) independently of the velocity. This is sketched in figure 4.4 along with the chosen values.

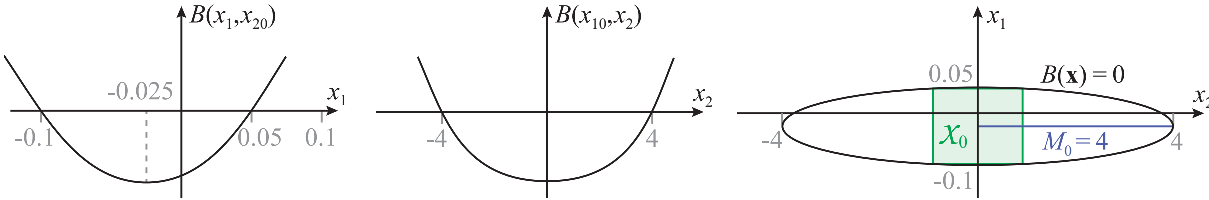


Figure 4.4: Choice of critical point coordinate x_{10} as the middle of the safe position interval. The semi-major axis of the CBF's zero level set is chosen much larger than the physical limits of the slide velocity.

Having chosen the coordinates x_{10} and x_{20} for the critical point, an arbitrary negative value is chosen for $B(x_{10}, x_{20})$. Using the four known coordinates from the zero level set (Λ_{h+}, x_{20}), (Λ_{h-}, x_{20}), $(x_{10}, -M_0)$ and (x_{10}, M_0), five equations with four unknowns can be outlined with the below numerical values.

$$\left.\begin{aligned}
\left(\frac{\left(\frac{\Lambda_{h-}+\Lambda_{h+}}{2}-x_{10}\right)^2}{a_2^2}+\frac{(0-x_{20})^2}{b_2^2}\right)c_1+c_2=-1.000\\
\left(\frac{(\Lambda_{h+}-x_{10})^2}{a_2^2}+\frac{(0-x_{20})^2}{b_2^2}\right)c_1+c_2=0\\
\left(\frac{(\Lambda_{h-}-x_{10})^2}{a_2^2}+\frac{(0-x_{20})^2}{b_2^2}\right)c_1+c_2=0\\
\left(\frac{\left(\frac{\Lambda_{h-}+\Lambda_{h+}}{2}-x_{10}\right)^2}{a_2^2}+\frac{(4-x_{20})^2}{b_2^2}\right)c_1+c_2=0\\
\left(\frac{\left(\frac{\Lambda_{h-}+\Lambda_{h+}}{2}-x_{10}\right)^2}{a_2^2}+\frac{(-4-x_{20})^2}{b_2^2}\right)c_1+c_2=0
\end{aligned}\right\}
\quad
\begin{aligned}
x_{10} &= -0.025\\
x_{20} &= 0.000\\
B(x_{10},x_{20}) &= -1.000\\
\\
a_2 &= 0.075\\
b_2 &= 4.000\\
c_1 &= 1.000\\
c_2 &= -1.000
\end{aligned}$$

The Lie derivatives can now be calculated as:

$$\begin{aligned}
L_gB(\mathbf{x}) &= \frac{dB(\mathbf{x})}{d\mathbf{x}}g(\mathbf{x})\Big|_{g(\mathbf{x})=\mathbf{B}} = \begin{bmatrix}\frac{\partial B(\mathbf{x})}{\partial x_1} & \frac{\partial B(\mathbf{x})}{\partial x_2}\end{bmatrix}\begin{bmatrix}0\\ \omega_n^2\end{bmatrix}\\
&= \begin{bmatrix}\frac{c_1(2x_1-2x_{10})}{a_2^2} & \frac{c_1(2x_2-2x_{20})}{b_2^2}\end{bmatrix}\begin{bmatrix}0\\ \omega_n^2\end{bmatrix} = \frac{c_1\omega_n^2(2x_2-2x_{20})}{b_2^2}\Big|_{x_{20}=0}\\
&= \frac{2c_1\omega_n^2}{b_2^2}x_2
\end{aligned}\tag{4.8}$$

It is seen that $L_gB(\mathbf{x}) \neq 0 \quad \forall\ x_2 \neq 0$, hence $L_fB(\mathbf{x})$ is for that reason analysed and evaluated at $x_2 = 0$:

$$\begin{aligned}
L_fB(\mathbf{x}) &= \frac{\partial B(\mathbf{x})}{\partial \mathbf{x}}f(\mathbf{x})\Big|_{f(\mathbf{x})=\mathbf{Ax}} = \begin{bmatrix}\frac{\partial B(\mathbf{x})}{\partial x_1} & \frac{\partial B(\mathbf{x})}{\partial x_2}\end{bmatrix}\begin{bmatrix}0 & 1\\ -\omega_n^2 & -2\zeta\omega_n\end{bmatrix}\begin{bmatrix}x_1\\ x_2\end{bmatrix}\\
&= \begin{bmatrix}\frac{c_1(2x_1-2x_{10})}{a^2} & \frac{c_1(2x_2-2x_{20})}{b^2}\end{bmatrix}\begin{bmatrix}x_2\\ -\omega_n^2x_1-2\zeta\omega_nx_2\end{bmatrix}\\
&= \frac{c_1x_2(2x_1-2x_{10})}{a_2^2} - \frac{c_1(2x_2-2x_{20})(\omega_n^2x_1+2\zeta\omega_nx_2)}{b_2^2}\Big|_{x_{20}=0}\\
&= 2c_1\left(\frac{x_1-x_{10}}{a_2^2} - \frac{\omega_n^2x_1+2\zeta\omega_nx_2}{b_2^2}\right)x_2\Big|_{x_2=0} = 0
\end{aligned}\tag{4.9}$$

It is noted that $L_fB(\mathbf{x}) = 0$ for all points $(x_1,0)$ (i.e. at zero velocity) which does not fulfil equation (3.1)b. It does, however, fulfil equation (3.1)d thus proving $B(\mathbf{x})$ from equation (4.7) to be a weak CBF. Note that when the velocity is zero, the slide movement is stable, although only marginally stable. As an engineering reflection it is considered that when the state leaves the marginally stable equilibrium, $x_2 \neq 0$, the safety controller will ensure that the state will increase its distance to the unsafe set and move towards its stable equilibrium in $(x_{10},x_{20}) = (-0.025,0)$.

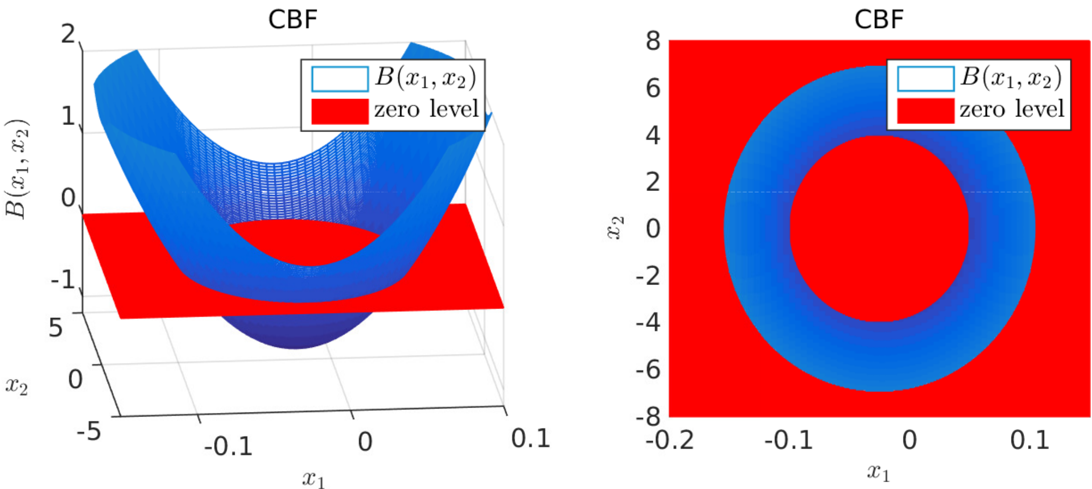


Figure 4.5: CBF for the second order system model. In the right plot the curve is seen from "above" for comparison with figure 4.4. Plot details and MATLAB script can be found in appendix J on the path `matlab_scripts/plot_cbf_2d/plot_cbf_2d.m`

The elliptic paraboloid with its proper boundaries is plotted in figure 4.5, from which it is seen how $B(\mathbf{x}) < 0$ only within the specified region, and positive for all $\mathbf{x} \in [\Lambda_{h-}, \Lambda_{h+}] \times [-M_0, M_0]$. It is also seen that for small velocities (physically $x_{2,\max} \approx 0.5\,\text{m/s}$) the CBF forms a nearly square subset of the safe set, i.e. $\{\mathbf{x} \in [-0.1, 0.05] \times [-0.5, 0.5]\}$, leaving $\mathcal{X}_0$ almost independent of the velocity in this area, as prescribed in figure 4.4.

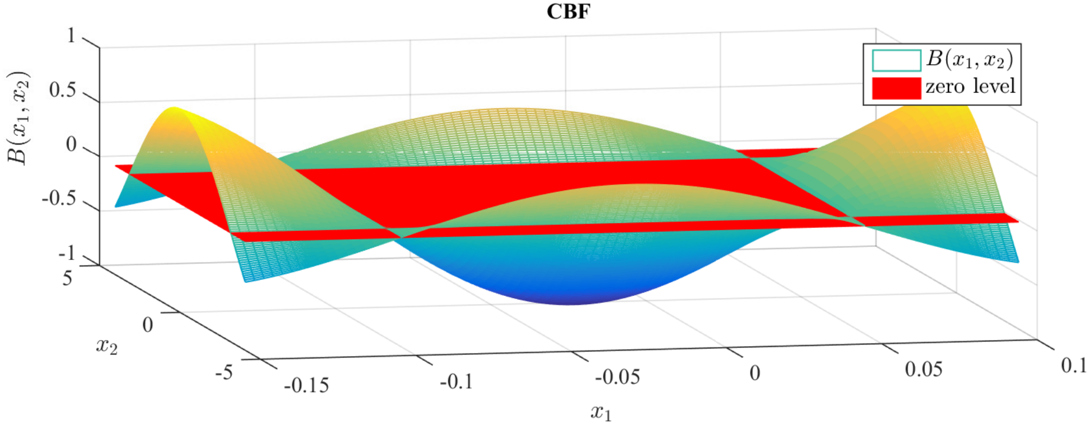


Figure 4.6: CBF. Example to demonstrate how other CBFs suffer when $x_2 = 0$.

Remark: The issue caused by $x_2 = 0$ is not only the case for this specific CBF but for many CBFs. Take for example another CBF that could fulfil the position requirements from table 4.1:

$$B(\mathbf{x}) = \cos(c_3 x_1 + c_4) \cdot \cos(c_5 x_2 + c_6)$$

It turns out that the coefficients $c_3 = 21.00, c_4 = 119.91, c_5 = 0.50, c_6 = 3.15$ induce a CBF with the same properties as the one depicted in figure 4.5. The CBF is outlined graphically in figure 4.6.

Thus $L_g B(\mathbf{x})$ can be found as:

$$L_gB(\mathbf{x}) = -c_5\omega_n^2\cos(c_3x_1 + c_4)\sin(c_5x_2 + c_6)$$

Now note that $L_gB(\mathbf{x}) = 0$ when $c_6 + c_5x_2 = i\pi$, $i \in \mathbb{Z}$, which is true for e.g. $x_2 \approx 0$. This implies the requirement that $L_fB(\mathbf{x}) < 0$ whenever $x_2 \approx 0$. However, taking a look at $L_fB(\mathbf{x})$:

$$L_fB(\mathbf{x}) = c_5\cos(c_3x_1 + c_4)\sin(c_5x_2 + c_6)(\omega_n^2x_1 + 2\zeta\omega_nx_2) - c3x_2\cos(c_5x_2 + c_6)\sin(c_3x_1 + c_4)$$

quickly poses the fact that $L_fB(\mathbf{x})$ is not necessarily negative for $x_2 \approx 0$ due to the sign alternation caused by the term $\sin(c_3x_1 + c_4)$ in the boundaries.

4.3 Control Design

This section constitutes the design of the two controllers. The controller based on the first order approximation is straight forward whereas the controller based on the second order approximation requires an observer because velocity measurements are not available through ROS in the current setup.

4.3.1 Control Design Based on the First Order Model

To be able to find $k_0(x)$ from section 3.1, the constant ε used in equation (3.5) must be found. It can be determined from the CBF from equation (4.6) such that it complies with the requirements from table 4.1:

$$\varepsilon = |B(\Lambda_{s+})| = |B(\Lambda_{s-})| = 0.00249 \tag{4.10}$$

Utilizing $\sigma(x)$ as described in equation (3.5) secures a neat way to incorporate the transition between the position controller and the safety controller $k_0(x)$, the latter which which gradually takes over when the trajectory exceeds Λ_s and has fully taken over when the trajectory reaches Λ_h. Figure 4.7 illustrates how ε and $B(x)$ are connected.

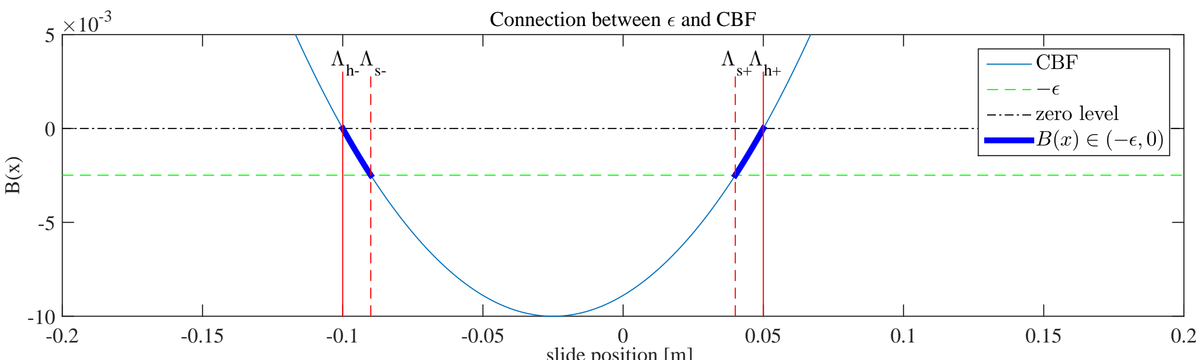


Figure 4.7: Connection between ε and CBF. MATLAB script and plot details can be found in appendix J as `matlab_scripts/plot_epsilon/plot_epsilon_slide_1d.m`

The system is approximated as a linear system on the form $\dot{\mathbf{x}} = \mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{u}$, thus pole placement can be used. No constraints to the constant feedback matrix $\mathbf{K}$ will be outlined except stability. It will therefore be determined from the pole placement method where a closed loop pole that is ten times faster than the open loop pole will be placed. Ackermann's formula can be used [Horowitz, 2014a]:

1. Identify the desired closed loop polynomial as $\mathbf{A}_{cl}(s) = s^n + a_{c(n-1)}s^{n-1} + \cdots + a_{c1}s + a_{c0}$:
$$\mathbf{A}_{cl}(s) = s + 10\tau^{-1}$$
2. Identify the open loop polynomial as $\mathbf{A}_{ol}(s) = s^n + a_{n-1}s^{n-1} + \cdots + a_1 s + a_0$:
$$\mathbf{A}_{ol}(s) = \lambda + \tau^{-1}$$
3. Compute the feedback matrix in controllable canonical form:
$$\bar{\mathbf{K}}^T = \begin{bmatrix} \bar{k}_1 \\ \vdots \\ \bar{k}_n \end{bmatrix} = \begin{bmatrix} a_{c0} - a_0 \\ \vdots \\ a_{c(n-1)} - a_{n-1} \end{bmatrix} \qquad \Rightarrow \qquad \bar{\mathbf{K}}^T = 10\tau^{-1} - \tau^{-1} = 9\tau^{-1}$$
4. Compute the similarity transform Q recursively as:
$$\mathbf{Q} = \begin{bmatrix} \mathbf{q}_1 & \mathbf{q}_2 & \cdots & \mathbf{q}_n \end{bmatrix} \qquad \Rightarrow \qquad \mathbf{Q} = \tau^{-1}$$
where
$$\begin{aligned} \mathbf{q}_n &= \mathbf{B} \\ \mathbf{q}_{j-1} &= \mathbf{A}\,\mathbf{q}_j + a_{j-1}\mathbf{B} \end{aligned}$$
5. Compute the feedback matrix as:
$$\mathbf{K} = \bar{\mathbf{K}}\mathbf{Q}^{-1} = 9\tau^{-1}\frac{1}{\tau^{-1}} = 9 \tag{4.11}$$

The constant feedback matrix $\bar{\mathbf{N}}$, ensuring unity gain between reference and output, can be computed as [Stoustrup, 2014]:
$$\bar{\mathbf{N}} = -\left(\mathbf{C}\mathbf{A}_{cl}^{-1}\mathbf{B}\right)^{-1} = -\left(\mathbf{C}\left(\mathbf{A} - \mathbf{B}\mathbf{K}\right)^{-1}\mathbf{B}\right)^{-1} = 10 \tag{4.12}$$

With the Lie derivatives computed as:
$$L_f B(\mathbf{x}) = -2a\tau^{-1}x^2 - b\tau^{-1}x \qquad \wedge \qquad L_g B(\mathbf{x}) = 2a\tau^{-1}x + b\tau^{-1} \tag{4.13}$$

Recapitulation 4.1 (Control Law for First Order Approximation)
The complete control law can be determined from equation (3.4):
$$u(x) = \sigma(x)k_0(x) + (1 - \sigma(x))(\bar{\mathbf{N}}x_{\text{ref}} - \mathbf{K}x)$$

where

$\sigma(x)$	is computed from equation (3.5) with the ε found in equation (4.10) and the CBF found in equation (4.6)
$k_0(x)$	is computed from equation (3.6) with the Lie derivatives stated in equation (4.13)
$\bar{\mathbf{N}}$	is found in equation (4.12)
$\mathbf{K}$	is found in equation (4.11)

This completes the control design based on a first order system approximation.

4.3.2 Control Design Based on the Second Order Model

A necessary condition for a controller is that the system is controllable:

$$\mathcal{C} = \begin{bmatrix} \mathbf{B} & \mathbf{A}\mathbf{B} \end{bmatrix} = \begin{bmatrix} 0 & \omega_n^2 \\ \omega_n^2 & -2\zeta\omega_n^3 \end{bmatrix} \qquad \text{thus} \quad \text{rank}(\mathcal{C}) = 2 = n \qquad \Rightarrow \quad \text{controllable}$$

To design the smoothing in the transition space $\mathcal{T}$ for the second order approximation, ε is found as the level set value of $B(\mathbf{x})$ at the position soft limit with zero velocity:

$$\varepsilon = |B(\Lambda_{s+}, x_{20})| = |B(\Lambda_{s-}, x_{20})| = 0.2489 \tag{4.14}$$

At this point, two cases will be considered:

- **1.** Construction of $\mathbf{K}$ and $\bar{\mathbf{N}}$ in a similar way as in subsection 4.3.1. This is possible in an ideal simulation because the velocity can be extrapolated by means of the forward Euler approach (ideal design).
- **2.** Development of an observer to estimate the velocity based on the model and position measurements. This is necessary on a real system.

Controller Design for MATLAB Simulation

The design of $\mathbf{K}$ and $\bar{\mathbf{N}}$ will follow the exact same procedure as described for the first order model except now $\mathbf{K} \in \mathbb{R}^{1\times 2}$, while $\bar{\mathbf{N}}$ remains as a scalar. The entire design procedure is therefore not elaborated.

However, it is of interest to slow down the system dynamics slightly compared to the controller based on a first order system. This is to enter the transition region with a lower velocity and thereby allow the safety controller some transition space to navigate the trajectory back to its safe area. The eigenvalues of the second order system is found to:

$$\lambda_{\text{2nd order system}} = \begin{cases} -10.295 - 14.765\,j \\ -10.295 + 14.765\,j \end{cases}$$

The feedback vector can be found with the MATLAB command `acker` based on pole placement slightly faster than the system itself.

$$\mathbf{K} = \texttt{acker(A,B,C,D,[-40 -50])} = \begin{bmatrix} 5.173 & 0.214 \end{bmatrix} \tag{4.15}$$

The DC gain can now be corrected with:

$$\bar{\mathbf{N}} = -\left(\mathbf{C}\,\mathbf{A}_{cl}^{-1}\,\mathbf{B}\right)^{-1} = -\left(\mathbf{C}\,(\mathbf{A} - \mathbf{B}\,\mathbf{K})^{-1}\,\mathbf{B}\right)^{-1} = 6.173 \tag{4.16}$$

These matrices are used in the MATLAB simulation.

Observer Design for Implementation on the da Vinci Robot

As the CBF and hence the safety controller are functions of both position and velocity of the end effector, measurements of both are needed. For the present setup, however, velocity measurements are not available through ROS, and thus an observer is designed to estimate the velocity. A necessary condition for an observer is that the system is observable:

$$\mathcal{O} = \begin{bmatrix} \mathbf{C} \\ \mathbf{CA} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \qquad \text{thus} \quad \text{rank}(\mathcal{O}) = 2 = n \qquad \Rightarrow \quad \text{observable}$$

An observer (discrete version) with proper gain corrections can be designed as [Stoustrup, 2014]:

$$\hat{\mathbf{x}}(k+1) = \Gamma\hat{\mathbf{x}}(k) + \Phi\mathbf{K}_d\hat{\mathbf{x}}(k) + \mathbf{L}_d\underbrace{(\mathbf{C}\hat{\mathbf{x}}(k) - \mathbf{y}(k))}_{\text{error}} + \mathbf{M}x_{\text{ref}} \tag{4.17}$$

with the associated continuous system:

$$\dot{\mathbf{x}} = \mathbf{A}\mathbf{x} + \mathbf{B}(\mathbf{N}x_{\text{ref}} - \mathbf{K}\mathbf{x})$$
$$\mathbf{y} = \mathbf{C}\mathbf{x}$$

where

k	is the current sample
Γ	is the discretized system matrix
Φ	is the discretized input matrix
$\mathbf{K}_d$	is control gain calculated from the discretized matrices
$\mathbf{L}_d$	is observer gain calculated from the discretized matrices
$\mathbf{M}$	is gain correction to ensure unity gain for the observer
$\mathbf{N}$	is gain correction to ensure unity gain for the system

The equations are implemented in simulink as shown in figure 4.8.

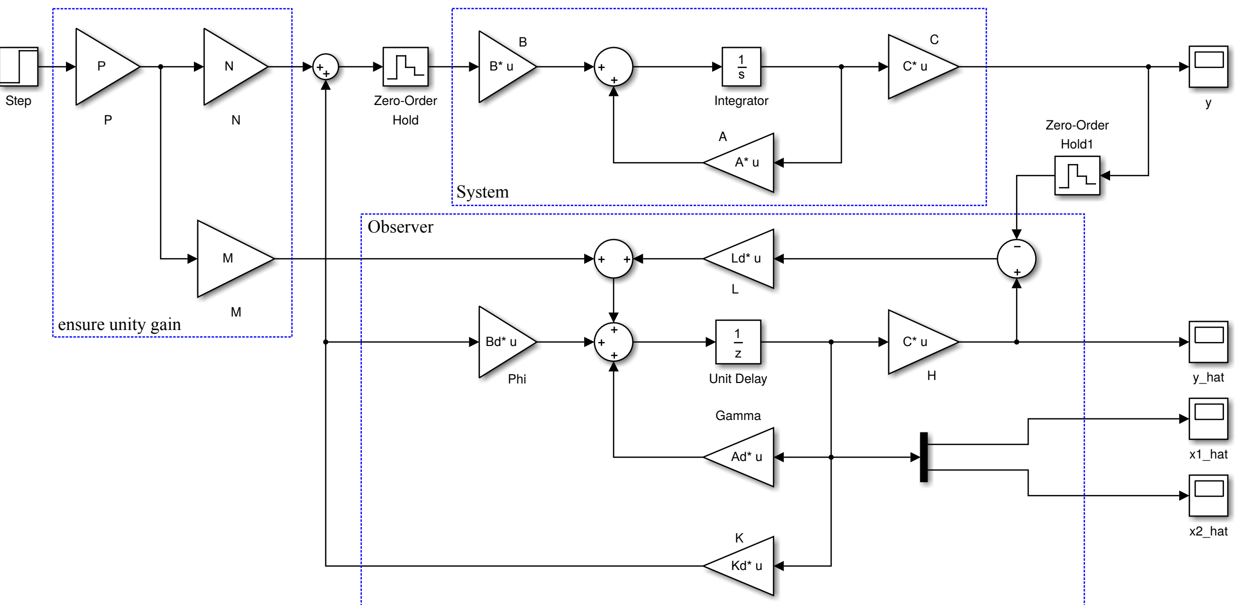


Figure 4.8: Simulink implementation of the discrete observer.

The augmented (discrete) system with $\mathbf{x}_{\text{augmented}} \in \mathbb{R}^4$ is formulated as:

$$\begin{bmatrix}\mathbf{x}(k+1)\\ \hat{\mathbf{x}}(k+1)\end{bmatrix} = \underbrace{\begin{bmatrix}\Gamma & \Phi\mathbf{K}_d\\ -\mathbf{L}_d\mathbf{C} & \Gamma+\Phi\mathbf{K}_d+\mathbf{L}_d\mathbf{C}\end{bmatrix}}_{\Gamma_{cl}}\begin{bmatrix}\mathbf{x}(k)\\ \hat{\mathbf{x}}(k)\end{bmatrix} + \underbrace{\begin{bmatrix}\Phi\mathbf{N}\\ \mathbf{M}\end{bmatrix}}_{\Phi_{cl}}x_{\text{ref}}$$

$$\mathbf{y}(k) = \underbrace{\begin{bmatrix}\mathbf{C} & 0 & 0\end{bmatrix}}_{\mathbf{C}_{cl}}\begin{bmatrix}\mathbf{x}(k)\\ \hat{\mathbf{x}}(k)\end{bmatrix}$$

The discrete matrices Γ and Φ can be found as [Horowitz, 2014b]:

$$\Gamma = \mathrm{e}^{\mathbf{A}T_s} = \begin{bmatrix} 0.986 & 0.009 \\ -2.623 & 0.817 \end{bmatrix} \tag{4.18}$$

$$\Phi = \int_0^{T_s} \mathrm{e}^{\mathbf{A}\mu} d\mu \cdot \mathbf{B} = \begin{bmatrix} 0.014 \\ 2.623 \end{bmatrix} \tag{4.19}$$

where
e is the matrix exponential
T_s is the sampling time, $T_s = 100\,\mathrm{ms}$

The sampling time T_s is according to Assistant Engineer Simon Jensen limited to 100 Hz caused by the TCP/IP communication channel between ROS and the underlying hardware as seen in figure 1.5.

The feedback matrix and the observer gain can now be found. They will again be calculated in MATLAB, as the design procedure follow the exact same as in subsection 4.3.1. The poles (p_i) will be placed from the below considerations:

- No overshoot in the closed loop step response, i.e. $\mathrm{Im}(p_i) = 0$.
- Asymptotic stability, i.e. $|p_i| < 1$
- Slow closed loop step response
- Positive feedback, i.e. the controller and observer gains must be negative
- An observer significantly faster than the closed loop system $\Gamma + \Phi\mathbf{K}_d$

$$\mathbf{K}_d = -\texttt{acker}\left(\Gamma, \Phi, \begin{bmatrix} 0.5 & 0.35 \end{bmatrix}\right) = \begin{bmatrix} -11.360 & -0.305 \end{bmatrix} \tag{4.20}$$

$$\mathbf{L}_d = -\texttt{acker}\left(\Gamma^T, \mathbf{C}^T, \begin{bmatrix} 0.01 & 0.02 \end{bmatrix}\right) = \begin{bmatrix} -1.773 \\ -68.184 \end{bmatrix} \tag{4.21}$$

The matrix $\mathbf{M}$ introduces zeros in the closed loop transfer function $\mathbf{y}(k)/x_{\text{ref}}$, which can be eliminated by designing the zeros close to the cut-off frequency. This means that the characteristic polynomial of the matrix $\Gamma_{za} + \tilde{\mathbf{M}}\mathbf{C}_{za}$ has zeros close to the cut-off frequency, where $\Gamma_{za} = \Gamma + \Phi\mathbf{K}_d + \mathbf{L}_d\mathbf{C}$ and $\mathbf{C}_{za} = -\mathbf{K}_d$ [Stoustrup, 2014]. The MATLAB function `acker` can again be used:

$$\tilde{\mathbf{M}} = -\texttt{acker}\left(\Gamma_{za}^T, \mathbf{C}_{za}^T, \begin{bmatrix} 0.01 & 0.02 \end{bmatrix}\right) = \begin{bmatrix} 0.014 \\ 2.623 \end{bmatrix}$$

To ensure unity gain between reference and system state, the $\mathbf{N}$ matrix can be computed as [Stoustrup, 2014]:

$$\mathbf{N} = -\left(\mathbf{C}_{cl}\Gamma_{cl}^{-1}\tilde{\Phi}_{cl}\right)^{-1} \qquad \text{where} \quad \tilde{\Phi} = \begin{bmatrix} \Phi \\ \tilde{M} \end{bmatrix}$$

$$\mathbf{N} = 13.739 \tag{4.22}$$

The matrix $\mathbf{M}$ ensuring unity gain between reference and observer state, can now be calculated as [Stoustrup, 2014]:

$$\mathbf{M} = \tilde{\mathbf{M}}\mathbf{N} = \begin{bmatrix} 0.186 \\ 36.040 \end{bmatrix} \tag{4.23}$$

Thereby, all unknowns from equation (4.17) are calculated.

Recapitulation 4.2 (Control Law for Second Order Approximation)
The complete controller based on the second order system approximation is now designed as:

$$\begin{aligned} \mathbf{1.} \quad & \hat{\mathbf{x}}(k+1) = \Gamma\hat{\mathbf{x}}(k) + \Phi\mathbf{K}_d\hat{\mathbf{x}}(k) + \mathbf{L}_d(\mathbf{C}\hat{\mathbf{x}}(k) - \mathbf{y}(k)) + \mathbf{M}x_{\text{ref}} \\ \mathbf{2.} \quad & \mathbf{u}(k) = \sigma(\mathbf{x})k_0(\hat{\mathbf{x}}) + (1-\sigma(\mathbf{x}))(\mathbf{N}\cdot x_{\text{ref}} - \mathbf{K}_d\hat{\mathbf{x}}(k)) \end{aligned}$$

where

$\sigma(\mathbf{x})$	is computed from equation (3.5) with ε from in (4.14) and the CBF found in (4.7)
$k_0(\mathbf{x})$	is computed from equation (3.4) with Lie derivatives from (4.8) and (4.9)
Γ	is found in equation (4.18)
Φ	is found in equation (4.19)
$\mathbf{N}$	is found in equation (4.22)
$\mathbf{M}$	is found in equation (4.23)
$\mathbf{L}_d$	is found in equation (4.21)
$\mathbf{K}_d$	is found in equation (4.20)
$\mathbf{C}$	is found in equation (4.5)

This completes the control design. The implementation constitutes both a MATLAB simulation and an actual implementation on the da Vinci robot in ROS. The MATLAB implementation is outlined first.

4.4 MATLAB Implementation and Results

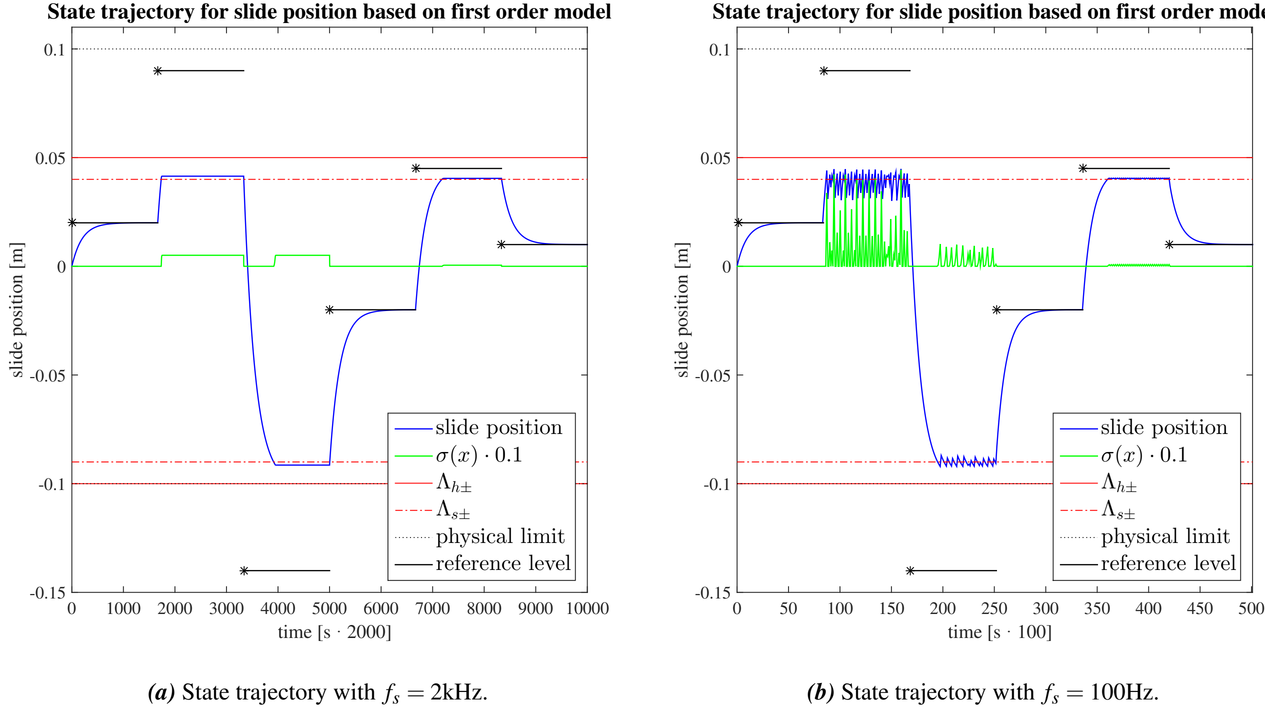
The results shown in this section are based on the implemented controller found in appendix G and in appendix J under the path `matlab_scripts/slide_controller/slide_controller.m`. All plots are made with the following characteristics:

- The sampling rate is tested with both $f_s = 2\,\text{kHz}$ (expected sampling rate in the long run) and $f_s = 100\,\text{Hz}$ (current limitation of the sample rate caused by the TCP/IP communication channel).
- The control signal is limited to ± 0.1 m (actual limit for slide movement).
- The velocity is limited to ± 1 m/s (conservatively estimated limit for slide movement).
- The forward Euler method is used to extrapolate the states.
- The design parameter κ from equation (3.4) is set to $\kappa = 1$ (neutral).
- The simulation time is 5 s. Various setpoints will indicate the behaviour in different regions.

The `model` variable in the first line in appendix G is set to the number 1 for the first order model and 2 for the second order model.

4.4.1 MATLAB Results Based on the First Order Model

The state trajectory composing slide position is plotted in figure 4.9, from which it is seen how the correct position is obtained on the safe interval $\mathcal{Y} = \{x \in [\Lambda_{s-}, \Lambda_{s+}]\}$. When setpoints are given outside the safe area, the safety controller ensures that the hard boundaries Λ_{h+} and Λ_{h-} are not exceeded at any time and that the position finds its equilibrium at a state determined by the combination of the two controllers. It is, however, noted that lowering the sampling frequency from 2 kHz to 100 Hz entails a closed loop



(a) State trajectory with $f_s = 2\text{kHz}$.

(b) State trajectory with $f_s = 100\text{Hz}$.

Figure 4.9: State trajectory for slide position for $\kappa = 1$. The MATLAB implementation can be found in appendix G. The plot is based on forward Euler.

system which is too fast compared to the sampling rate, and for setpoints outside the safe region, the position is oscillating between linear and safety control.

To verify that equation (3.1)b is fulfilled in the simulation, the Lie derivatives are plotted in figure 4.10, from which it can be seen that $L_gB(x) \neq 0$ for all $x \neq \frac{-b}{2a}$, and that $L_fB(x) = 0$ in $x = \frac{-b}{2a}$, which essentially fulfils equation (3.1)b. Indeed, even for $\mathcal{X}$ defined as the entire range $[\Lambda_{h-}, \Lambda_{h+}]$ (compare to table 4.1), the chosen CBF would have been valid because the only place where $L_gB(x) = 0$ is at the critical point. Note also that the Lie derivatives are independent of the scalar κ.

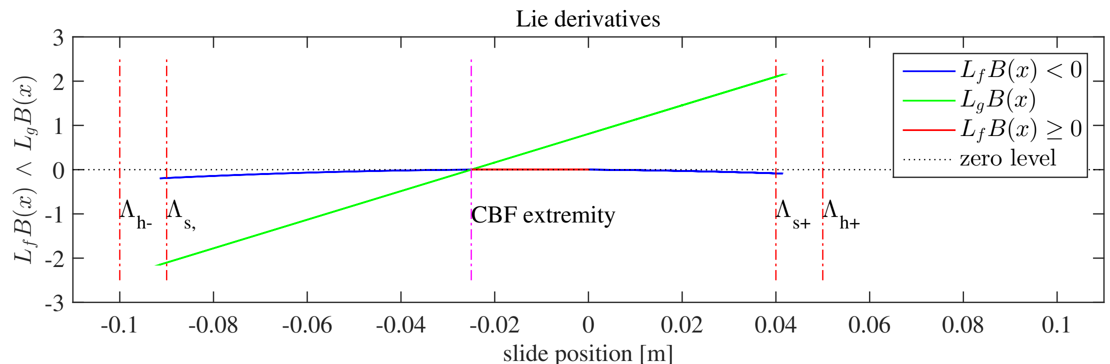
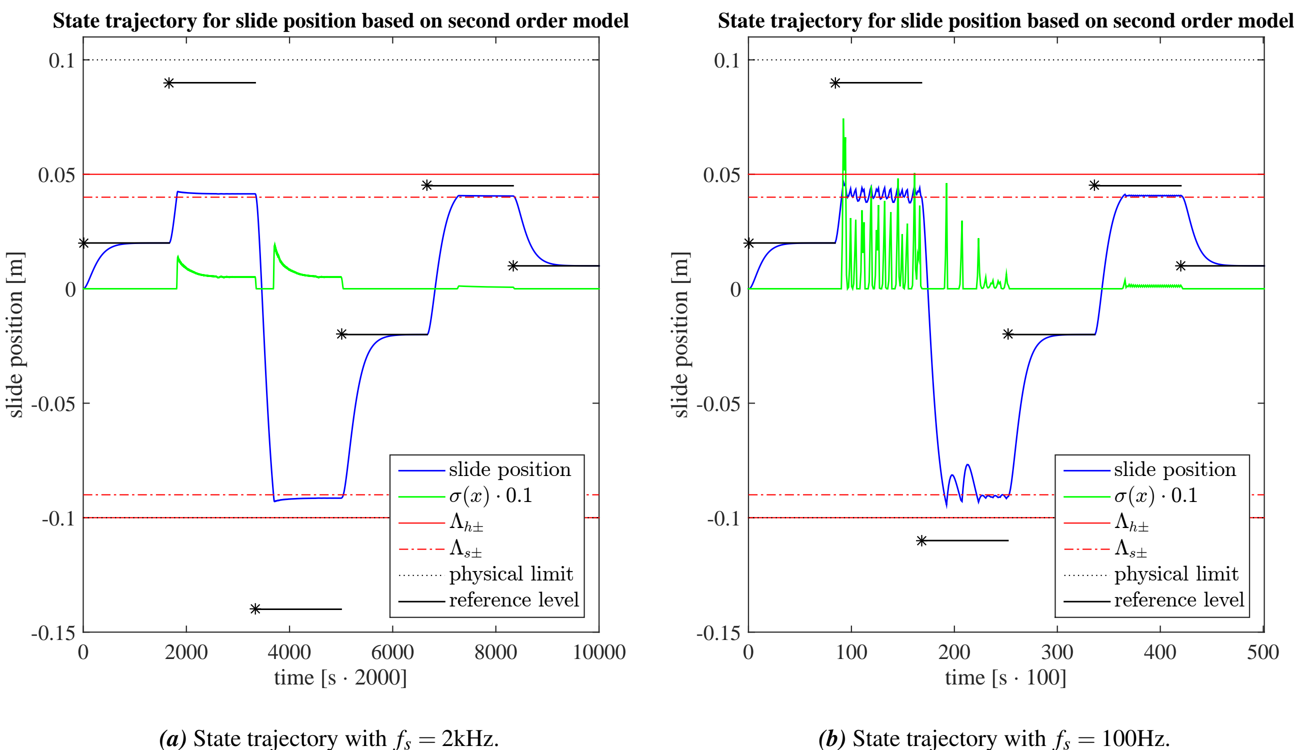


Figure 4.10: Lie derivatives of the CBF along the vector fields $f(\mathbf{x}) = \mathbf{A}\mathbf{x}$ and $g(\mathbf{x}) = \mathbf{B}$.

Varying κ increases the aggressivity of the safety controller and can create fluctuations in the transition area if the sampling rate is too low. Likewise $\sigma(x)$ will fluctuate along with an increased κ. One must be careful when increasing κ when the sampling rate is relatively low.

4.4.2 MATLAB Results Based on the Second Order Model

The state trajectory composing slide position based on the second order model is shown in figure 4.11. It is seen how the boundaries are respected at all times regardless of irresponsible setpoints within the unsafe region, which cause $\sigma(\mathbf{x})$ to increase and thereby letting the control law be the linear combination of the safety controller $u(\mathbf{x}) = k_0(\mathbf{x})$ and the linear controller by pole-placement $\tilde{u}(\mathbf{x})$. The state trajectory shown in figure 4.11 verifies that the slide position does not exceed its limits even when setpoints are given outside the safe region.



(a) State trajectory with $f_s = 2$kHz.

(b) State trajectory with $f_s = 100$Hz.

Figure 4.11: State trajectory for position based on second order system approximation.

The Lie derivatives for the second order model and CBF are plotted in figure 4.12. It is seen how $L_gB(\mathbf{x}) = 0$ and $L_fB(\mathbf{x}) = 0$ at the same time which in general is critical but accepted in this specific case as it is caused by $x_2 = 0$ which implies $u(\mathbf{x}) = 0 \;\Rightarrow\; x_1 \to 0$ which is safe.

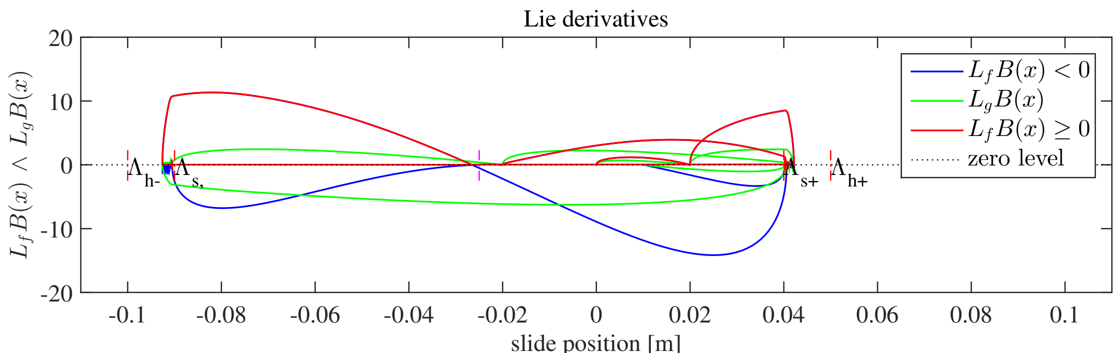


Figure 4.12: Lie derivatives of the CBF for the second order model.

4.4.3 Verification of the Observer Developed for the Robot Implementation

The observer developed in subsection 4.3.2 will be verified with a step input at 1 cm. The estimated position and velocity is plotted with the success criteria that $\hat{x}_1 = x_{\text{ref}}$ for $t \to \infty$ and that no overshoot in the position occurs. The velocity must stay within a reasonable velocity span, i.e. below 1 m/s when the time constant is considered. The error is defined as $\hat{y}$–y and must obviously stay very low in steady state.

The observer is initialized with an estimation error in both position and velocity, and the result is plotted in figure 4.13. It is seen how the position, velocity and error all comply with the expected outcome and fulfil the requirements.

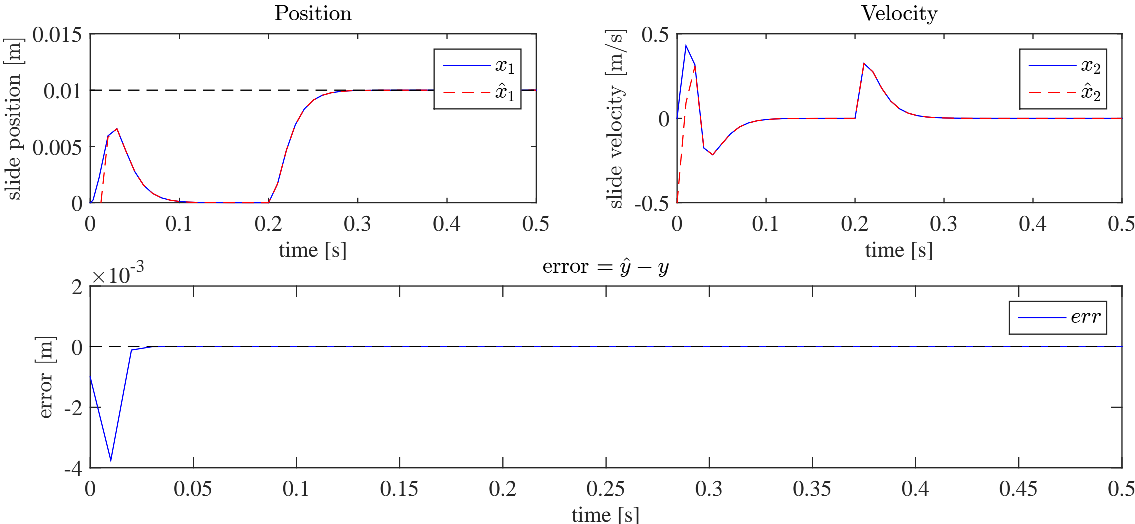


Figure 4.13: Simulation results from the simulink implementation of the observer. Plot details and simulink file can be found in appendix J by running `run_observer.m` under the path `matlab_scripts/observer/`.

The outlined plots throughout this section conclude the MATLAB simulation. The MATLAB implementation shows the expected scenario, i.e. that the state trajectory complies with the outlined safe and unsafe regions. It shall now be seen how to implement the results on the da Vinci robot itself.

4.5 Implementation on the da Vinci Robot

The implementation constitutes the below listed bullet points:

- The controller will be implemented in C++. **Reason:** Along with Python, C++ is *the* ROS compatible standard. The reason to use C++ over Python is to optimize speed performance. Furthermore, it is the general opinion among ROS experts (such as Postdoc Karl Damkjær Hansen and others) that C++ is more useful in robot simulations and development and lastly, the already existing code at the Robotic Surgery Group - Aalborg University, is by far most developed in C++. However, Python as a scripting language may be more user friendly, easier to get started with and in many cases more readable.
- Real-time signal processing to ensure fixed sample rates. **Reason:** The observer matrices are built upon fixed sampling rate and for that reason it is crucial to comply with a fixed sample rate.
- Algorithm development to connect these two bullet points.

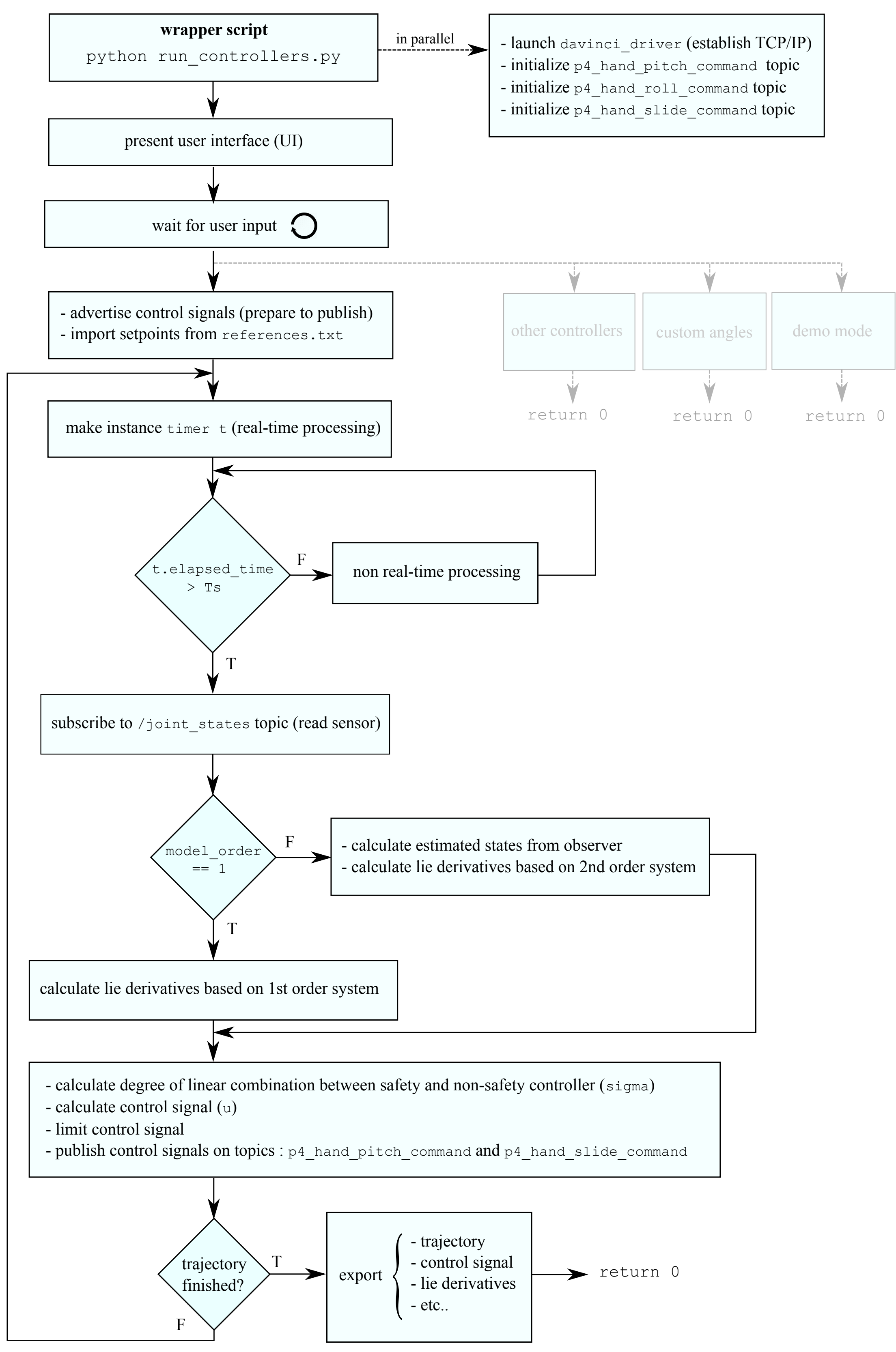


Figure 4.14: Algorithm for slide safety controller. The source code associated with the algorithm can be found in appendix H and in appendix J. It can also be found at github at the Robotic Surgery Group - Aalborg University under the repository gr1032 (*https://github.com/AalborgUniversity-RoboticSurgeryGroup/*).

The controller is implemented at the highest abstraction layer, i.e. the ROS environment depicted in figure 1.5. While ROS runs on most Linux laptops, it is not ROS itself that puts forth limitations for real time signal processing, neither is it the potentiometers that measure the angle. The bottleneck is caused by the TCP/IP communication channel which according to Assistant Engineer Simon Jensen is limited to 100 Hz. For that reason, the maximum allowed execution time $c_{p,\max}$ is:

$$c_{p,\max} = \frac{1}{100\,\text{Hz}} = 10\,\text{ms}$$

The main algorithm is depicted in figure 4.14, and the source code can be found in appendix H and in appendix J. A complete description on how to run the developed framework is found in section A.3.

4.5.1 Implementation on the da Vinci Robot Based on the First Order Model

All plots and measurements in this subsection can be reconstructed by running the MATLAB script `plot_data` found in appendix J in the folder `measurements/slide_safety_controller/1D_1st_order`. The execution time is validated first as it is essential for the controller to complete successfully. Figure 4.15 shows a plot of measured execution time for each iteration, verifying that $c_p < c_{p,\max}$ and thereby the real-time part.

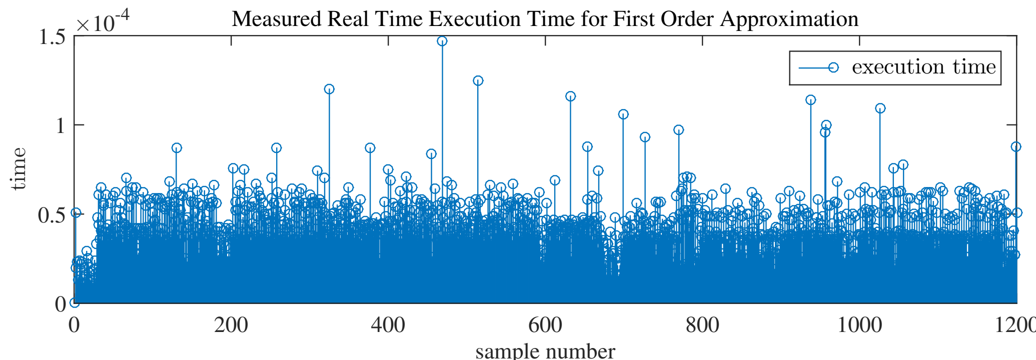


Figure 4.15: Execution time for the first order system approximation. It is seen that the controller never exceeds a computation time of 150 μs.

The measured state trajectory is plotted in figure 4.16. Here it is seen how the controller ensures that $x \in \mathcal{X}_u^c$ for all x. The poor system approximation is also revealed as the trajectory has an overshoot which was not included in the simulation found in figure 4.9. This is however not surprising as the first order approximation was created to simplify the control barrier function and to avoid the observer design as an initial approach. A consequence of the overshoot is seen when setpoints very close to Λ_{s+} are given. Here the slide movement starts to oscillate, caused by $\sigma(x)$. This is obviously not a good thing, but nevertheless it is the intended outcome when $x \in \mathcal{T}$. Additionally, it is seen how $\sigma(x)$ allows $k_0(x)$ to force the position from reaching its setpoint when $x_{\text{ref}} \in \mathcal{X}_u$ but allows x to cross x_{ref} when $x_{\text{ref}} \in \mathcal{T}$. It is finally seen that the safety controller is fully functional for both $\mathbb{R}^-$ and $\mathbb{R}^+$.

The Lie derivatives are calculated based on the measured position. The result is seen in figure 4.17. It is seen that the Lie derivatives from figure 4.17 are very similar to the theoretical Lie derivatives from figure 4.10.

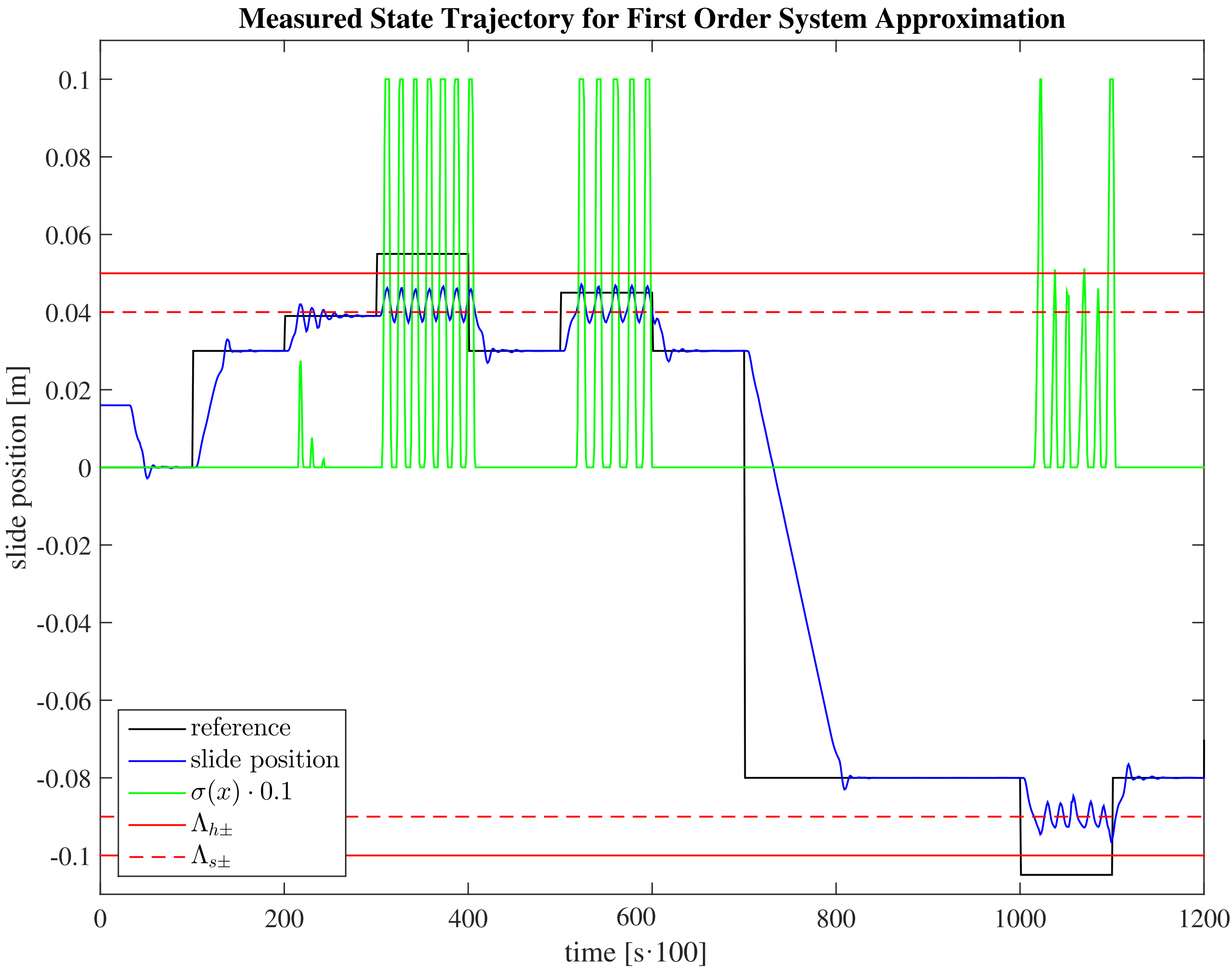


Figure 4.16: Measured position trajectory based on the first order system approximation. Joint velocity limiting to 25 % becomes effective when large steps are given, thus altering the system dynamics. This feature is set in the underlying low-level controllers on the FPGA (see figure 1.5) in the development phase to protect the system from excessive mechanical stress. This is particularly noticeable for the step at sample 700, but is not important in this context as the aim merely is to show setpoint tracking and safety controller effectiveness.

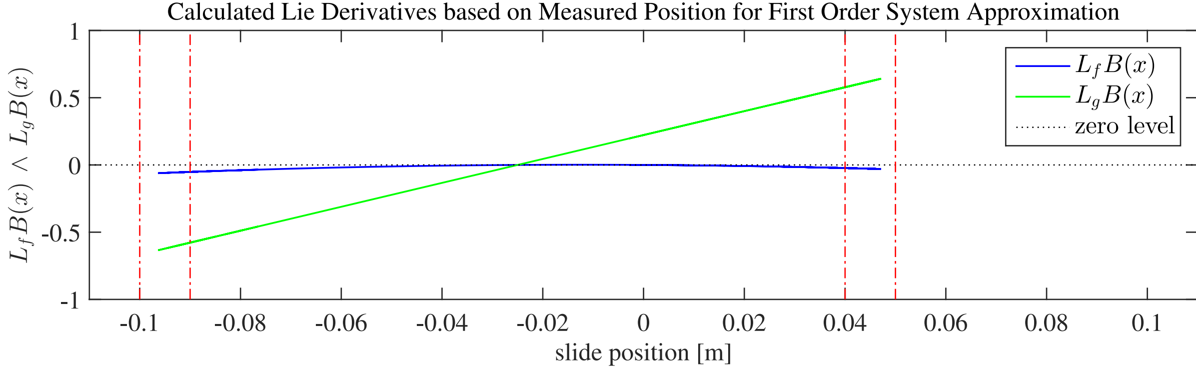


Figure 4.17: Calculated Lie derivatives based on measured position for the first order approximation.

4.5.2 Implementation on the da Vinci Robot Based on the Second Order Model

All plots and measurement in this subsection can be reconstructed by running the MATLAB script `plot_data` found in appendix J in the folder `measurements/slide_safety_controller/2D_2nd_order`. Again, the execution time is validated first. A plot of measured execution time for each iteration is shown in figure 4.18. It is seen that the real time part is completed within the allowed 10 ms (100 Hz sample rate). Note that the execution time is slightly higher than in figure 4.15 because the velocity is estimated with the observer.

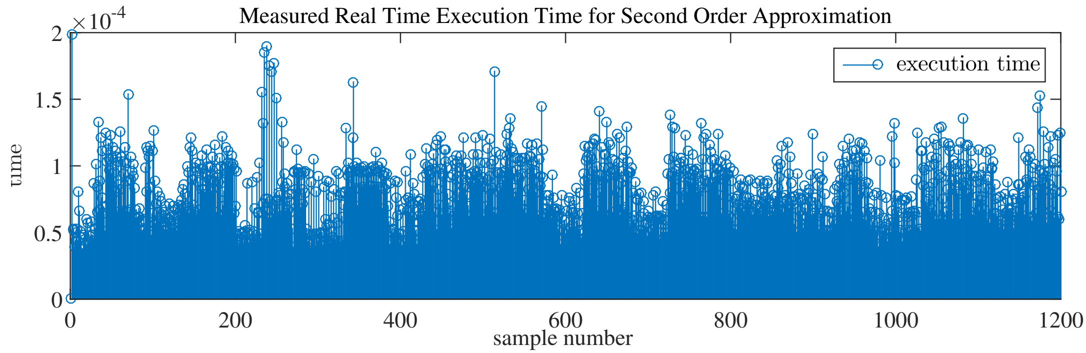


Figure 4.18: Execution time for second order system approximation. It is seen that the controller never exceeds a computation time of 200 μs.

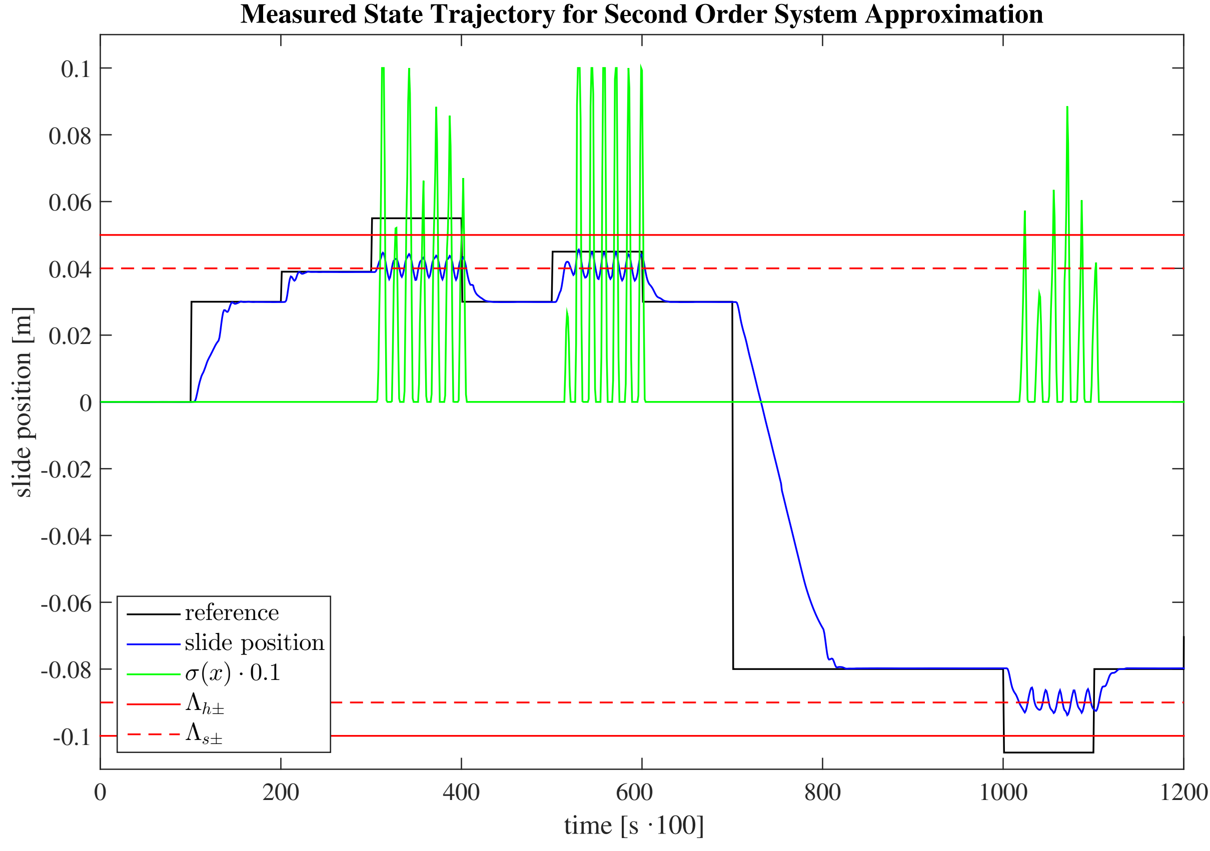


Figure 4.19: Measured position trajectory based on the second order approximation. Similar to figure 4.16 it is seen how joint velocity limiting becomes effective when large steps are given.

The measured state trajectory is plotted in figure 4.19, visualizing how the overshoot is eliminated, which was indeed the main purpose of the development of a controller based on a second order approximation. Also, note how it is possible to give setpoints close to the the set $\mathcal{T}$ and still avoid oscillations caused by $\sigma(\mathbf{x}) \neq 0$. This is a big advantage when a doctor needs to operate close to an unsafe region. Finally, just as in figure 4.16, it is seen how $\sigma(\mathbf{x})$ allows $k_0(\mathbf{x})$ to force the position from reaching its setpoint when $x_{\text{ref}} \in \mathcal{X}_u$ but allows x to cross x_{ref} when $x_{\text{ref}} \in \mathcal{T}$, as is intended with $\sigma(\mathbf{x})$.

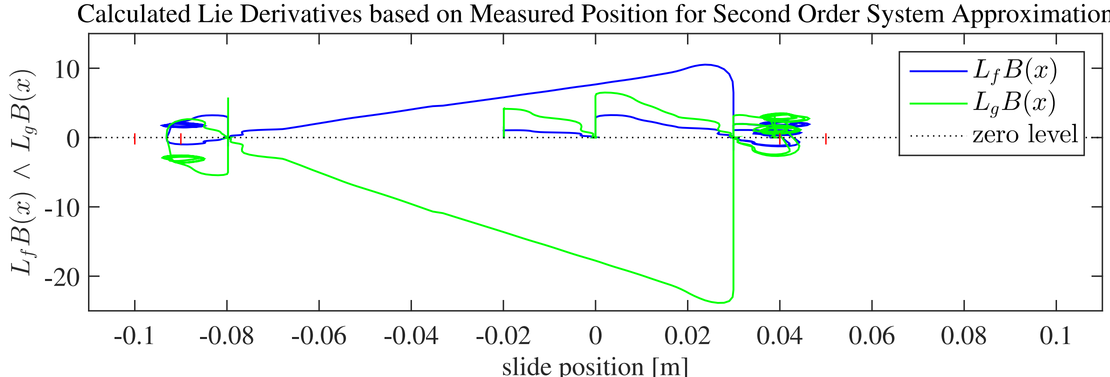


Figure 4.20: Calculated Lie derivatives based on measured position and estimated velocity for the second order system approximation.

The Lie derivatives are calculated based on the measured position and the estimated velocity. The result is seen in figure 4.20 showing that the Lie derivatives are quite different from the simulated derivatives in figure 4.12. This may be due to an imprecise state estimation.

4.6 Conclusion for the Slide Safety Controller

In conclusion, this chapter verifies the usefulness of the theory presented in chapter 3 verifying that it complies with both simulation and real world implementation. It is seen how the trajectory is forced away from the unsafe region whenever setpoints are given in $\mathcal{X}_u$, thus the theory presented in [Wieland and Allgöwer, 2007] does indeed turn out to be very useful in practical implementations. It is worth noting that the construction of barrier certificates can be quite time consuming (especially when the state vector grows in order) and that the velocity causes some challenges when barrier certificates are constructed. For that reason, one may look in other directions when higher order systems are considered.

The results from this chapter suggest some improvements in the current setup:

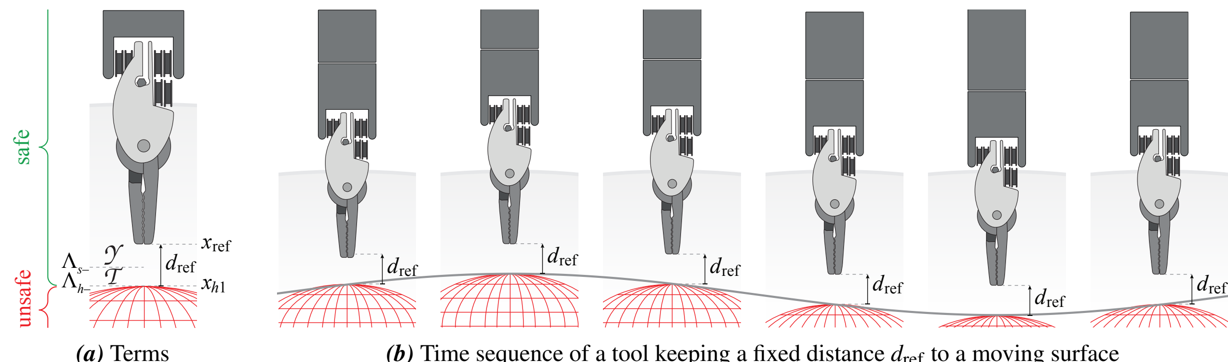
- Replace the TCP/IP communication channel with a User Datagram Protocol (UDP) channel such that the topic `joint_states` updates faster, i.e. preferably with 2 kHz. This will improve the precision and resolution and ultimately help save human lives.
- Improve the transition region $\mathcal{T}$ such that precise setpoints can be given in this region. This could imply a purely non-linear controller in $\mathcal{T}$.
- Improve the bump function $\sigma(\mathbf{x})$ for the second order system such that the braking distance is optimized.

The above considerations conclude this chapter.

Chapter 5

Safety and Surgery on Beating Hearts

This chapter establishes the fundamentals for a system ensuring safety when a surgeon operates on a patient with a beating heart. In addition to the safety, it is the objective to establish a system with virtual fixture of the heart, i.e. such that the surgeon experiences the beating heart as static. As suggested by chief surgeon at Aalborg University Hospital, Johan Poulsen (see section 1.3), a proper initial set-up emulating a beating heart could be a surface mounted on a cylinder moving periodically as a sinusoid. This scheme is outlined in figure 5.1 illustrating how the virtual fixture of the heart should be obtained by dynamically positioning the end effector at a fixed distance from the surface of the heart.



(a) Terms ***(b)*** Time sequence of a tool keeping a fixed distance d_{ref} to a moving surface

Figure 5.1: The objective is to attain a virtual fixture of the moving object by controlling the end effector distance to its surface, such that the surgeon experiences the object as static and can operate as if the object was in fact standing still. An overview of terms applied is found in figure 5.1a.

Thus a model of heart movement is needed along with a model of the robot dynamics, such that a reference relative to the surface of the heart can be given. These models are presented in the following, after which a Control Barrier Function (CBF) can be constructed and from this a controller is designed that ensures safety of the system.

5.1 Modelling the Dynamics of the System

First a model of a beating heart is introduced. This is needed in order to be able to give a dynamic position reference such thath the end effector can be controlled to keep a fixed distance to the heart surface. The the system model from section 4.1 is recapitulated, and the dynamic reference is introduced through an augmented system.

5.1.1 Modelling the Beating Heart as a Sinusoid

The simplified heart movement can be represented as a matrix with eigenvalues on the imaginary axis in the complex frequency domain:

$$\dot{\mathbf{x}}_h = \begin{bmatrix} x_{h1} \\ x_{h2} \end{bmatrix}^{\cdot} = \begin{bmatrix} 0 & \omega_h \\ -\omega_h & 0 \end{bmatrix} \begin{bmatrix} x_{h1} \\ x_{h2} \end{bmatrix} \qquad \text{with} \qquad \mathbf{x}_h(0) = \begin{bmatrix} x_{h10} \\ x_{h20} \end{bmatrix} \tag{5.1}$$

where

x_{h1} is the position of a point on the surface of the heart
x_{h2} is the velocity of the point
ω_h is the heartbeat frequency, $\omega_h = 2\pi/T_h$ rad/s with an average heartbeat period T_h= 1.1 s [Duindam and Sastry, 2007]
x_{h10} is the initial value of x_{h1}
x_{h12} is the initial value of x_{h2}

Note that the magnitude of ω_h determines the frequency and that the initial conditions determine the amplitude of the heart oscillation, e.g. if $\mathbf{x}_h(0) = [0 \quad 2]^T$ this indicates a sine with an amplitude of 2, while $\mathbf{x}_h(0) = [3 \quad 0]^T$ corresponds to a cosine with an amplitude of 3. The heartbeat will with this system have a natural oscillation around zero.

5.1.2 Modelling the Robot Slide Movement

As in chapter 4, the slide movement of the robot is considered the degree of freedom of the system, illustrated in figure 4.1. To model the one dimensional robot movement along the slide axis, the same system model can be used as presented in equation (4.1), i.e.:

$$\dot{x}_1 = -\tau^{-1} x_1 + \tau^{-1} u \tag{5.2}$$

where

x_1	is the position of the end effector	[m]
τ	is the time constant of the first order system, measured in figure 4.2, $\tau = 110\,\text{ms}$	[s]
u	is the input to the system	[m]

The one dimensional model is used to simplify the equations thus precluding complications at this stage. Additionally, the second order model merely proved few advantages in the form of elimination of the overshoot.

Introducing a Dynamic Reference in the Model

In order to control the robot end effector to maintain a fixed distance to the surface of the heart, an additional state can be added the system. Thus, a new set of states is introduced representing the movement of the beating heart, the movement of the robot and the fixed distance d_{ref} between the two. Combining equation (5.1) and equation (5.2) with the relative distance yields the below stated linear state space system:

$$\dot{\mathbf{x}}(t) = \begin{bmatrix} x_1 \\ x_{h1} \\ x_{h2} \\ d_{\text{ref}} \end{bmatrix}^{\cdot} = \underbrace{\underbrace{\begin{bmatrix} -1/\tau & 0 & 0 & 0 \\ 0 & 0 & \omega_h & 0 \\ 0 & -\omega_h & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}}_{\mathbf{A}} \begin{bmatrix} x_1 \\ x_{h1} \\ x_{h2} \\ d_{\text{ref}} \end{bmatrix}}_{f(\mathbf{x})} + \underbrace{\underbrace{\begin{bmatrix} 1/\tau \\ 0 \\ 0 \\ 0 \end{bmatrix}}_{\mathbf{B}}}_{g(\mathbf{x})} \tilde{u}(x) \tag{5.3}$$

$$\mathbf{y}(t) = \underbrace{\begin{bmatrix} 1 & 0 & 0 & 0 \end{bmatrix}}_{\mathbf{C}} \mathbf{x}(t) \tag{5.4}$$

The control law is introduced as a linear position controller on the form from equation (3.3), where the position reference follow the dynamic movement of the heart at a fixed distance d_{ref} from its surface, i.e. $x_{\text{ref}} = x_{h1} + d_{\text{ref}}$ such that:

$$\begin{aligned} \tilde{u}(\mathbf{x}(t)) &= \bar{\mathbf{N}} x_{\text{ref}} - \mathbf{K} x_1 \\ &= \bar{\mathbf{N}}\Big(x_{h1}(t) + d_{\text{ref}}\Big) - \mathbf{K} x_1(t) \\ &= \begin{bmatrix} -\mathbf{K} & \bar{\mathbf{N}} & 0 & \bar{\mathbf{N}} \end{bmatrix} \mathbf{x}(t) \\ &= \bar{\mathbf{K}} \mathbf{x}(t) \end{aligned} \tag{5.5}$$

where

- $\mathbf{K}$ is the linear system controller designed according to equation (4.11), $\mathbf{K} \in \mathbb{R}$
- $\bar{\mathbf{N}}$ is the system gain ensuring unity gain between position and reference, found according to equation (4.12), $\bar{\mathbf{N}} \in \mathbb{R}$
- $\bar{\mathbf{K}}$ is the augmented feedback vector, $\bar{\mathbf{K}} \in \mathbb{R}^{1\times 4}$

Note how the closed loop system can be rewritten in a more intuitive way, such that the system is reduced to three states taking d_{ref} as input:

$$\dot{\begin{bmatrix} x_1 \\ x_{h1} \\ x_{h2} \\ d_{\text{ref}} \end{bmatrix}} = \underbrace{\underbrace{\begin{bmatrix} -1/\tau & 0 & 0 & 0 \\ 0 & 0 & \omega_h & 0 \\ 0 & -\omega_h & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}}_{\mathbf{A}} \begin{bmatrix} x_1 \\ x_{h1} \\ x_{h2} \\ d_{\text{ref}} \end{bmatrix}}_{f(\mathbf{x})} + \underbrace{\underbrace{\begin{bmatrix} 1/\tau \\ 0 \\ 0 \\ 0 \end{bmatrix}}_{\mathbf{B}}}_{g(\mathbf{x})} \underbrace{\underbrace{\begin{bmatrix} -\mathbf{K} & \bar{\mathbf{N}} & 0 & \bar{\mathbf{N}} \end{bmatrix}}_{\bar{\mathbf{K}}} \begin{bmatrix} x_1 \\ x_{h1} \\ x_{h2} \\ d_{\text{ref}} \end{bmatrix}}_{\tilde{u}(x)} \tag{5.6}$$

$$\dot{\begin{bmatrix} x_1 \\ x_{h1} \\ x_{h2} \end{bmatrix}} = \underbrace{\begin{bmatrix} -(\mathbf{I}+\mathbf{K})/\tau & \bar{\mathbf{N}}/\tau & 0 \\ 0 & 0 & \omega_h \\ 0 & -\omega_h & 0 \end{bmatrix}}_{\mathbf{A}_{cl}} \begin{bmatrix} x_1 \\ x_{h1} \\ x_{h2} \end{bmatrix} + \underbrace{\begin{bmatrix} \bar{\mathbf{N}}/\tau \\ 0 \\ 0 \end{bmatrix}}_{\mathbf{B}_{cl}} d_{ref} \tag{5.7}$$

Note the eigenvalues of $\mathbf{A}_{cl}$:

$$\lambda_{\mathbf{A}_{cl}} = \begin{cases} -90 \\ 5.71\,j \\ -5.71\,j \end{cases}$$

The eigenvalues reveal a stable (controllable) subsystem consisting of the robot end effector and an obviously marginally stable (uncontrollable) subsystem consisting of the beating heart.

This concludes the modelling of the augmented system. In the next section a CBF will be constructed.

5.2 Construction of CBF

A barrier certificate for the system presented in equation (5.6), with a zero level set at the surface of the heart, comprises moving boundaries and hence should be a function of both robot and heart position. Definition 2.1 implies that the CBF should be constructed with unsafe region $\mathcal{X}_u$ below the surface of the heart, i.e. such that $B(\mathbf{x})$ is positive if $x_1 < x_{h1}$ and negative otherwise, with x_1 being the position of the robot end effector and x_{h1} being the position of the heart surface. The coherence is clear:

$$B(\mathbf{x}) > 0 \quad \forall\ \mathbf{x} \in \mathcal{X}_u = \{(x_1, x_{h1}) \mid x_{h1} - x_1 > 0\} \tag{5.8}$$

Thus a CBF can be constructed as:

$$B(\mathbf{x}) = \tilde{c}(x_{h1} - x_1) \tag{5.9}$$

with a positive constant $\tilde{c} > 0$, which is chosen to be $\tilde{c} = 1$. Thus according to Definition 3.1 this is a valid CBF if $L_g B(\mathbf{x}) \neq 0$ and $\{\mathbf{x} \in \mathcal{X} \mid B(\mathbf{x}) \leq 0\} \neq \emptyset$. The latter is trivial as the safe states are present whenever $x_1 > x_{h1}$. Thus $L_g B(\mathbf{x})$ is analysed:

$$\begin{aligned} L_g B(\mathbf{x}) = \frac{dB(\mathbf{x})}{d\mathbf{x}} g(\mathbf{x}) \Big|_{g(\mathbf{x})=\mathbf{B}} &= \begin{bmatrix} \frac{\partial B(\mathbf{x})}{\partial x_1} & \frac{\partial B(\mathbf{x})}{\partial x_{h1}} & \frac{\partial B(\mathbf{x})}{\partial x_{h2}} & \frac{\partial B(\mathbf{x})}{\partial d_{\text{ref}}} \end{bmatrix} \mathbf{B} \\ &= \begin{bmatrix} -\tilde{c} & \tilde{c} & 0 & 0 \end{bmatrix} \begin{bmatrix} 1/\tau \\ 0 \\ 0 \\ 0 \end{bmatrix} \Big|_{\tilde{c}=1} = \frac{-1}{\tau} \neq 0 \quad \forall \mathbf{x} \in \mathcal{X} \end{aligned} \tag{5.10}$$

It is seen that $L_g B(\mathbf{x}) \neq 0$ for all $\mathbf{x} \in \mathcal{X}$. The CBF is therefore valid on $\mathcal{X}$.

5.3 Control Design

As introduced in equation (3.2), the safe set $\mathcal{X}_0$ can be divided into two sections: the transitions space $\mathcal{T}$ close to the unsafe set and the remaining part $\mathcal{Y} = \mathcal{X}_0 \setminus \mathcal{T}$. These sections are indicated in figure 5.1a. The control law is, just as in chapter 4, split in two parts. A linear controller where no safety precautions are taken is used in $\mathcal{Y}$ and a controller ensuring safety is used in $\mathcal{T} \bigcup \mathcal{X}_u$. The controller ensuring safety is defined in equation (3.6).

The transition between the two controllers on $\mathcal{T}$ is determined by $\sigma(\mathbf{x})$ which is defined in equation (3.5). Thus a scalar ε is required, determining when the effect of the safety controller $k_0(\mathbf{x})$ is brought into effect, while the weighting of $k_0(\mathbf{x})$ is determined by $\sigma(\mathbf{x})$. To give the safety controller some margin to force the robot end effector away from $\mathcal{X}_u$, it is reasonable to let $k_0(\mathbf{x})$ take effect when the end effector comes within a distance from the unsafe set of 1 cm, thus:

$$\varepsilon = 0.01 \tag{5.11}$$

The safety controller, as defined in equation (3.6), requires the Lie derivatives of the CBF, where $L_g B(\mathbf{x})$ is found in equation (5.10), and $L_f B(\mathbf{x})$ is found as:

$$L_f B(\mathbf{x}) = \frac{dB(\mathbf{x})}{d\mathbf{x}} f(\mathbf{x}) \Big|_{f(\mathbf{x})=\mathbf{A}\mathbf{x}}$$

$$= \begin{bmatrix} \frac{\partial B(\mathbf{x})}{\partial x_1} & \frac{\partial B(\mathbf{x})}{\partial x_{1h}} & \frac{\partial B(\mathbf{x})}{\partial x_{2h}} & \frac{\partial B(\mathbf{x})}{\partial d_{\text{ref}}} \end{bmatrix} \begin{bmatrix} 1/\tau & 0 & 0 & 0 \\ 0 & 0 & \omega_h & 0 \\ 0 & -\omega_h & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_{x1} \\ x_{x2} \\ d_{\text{ref}} \end{bmatrix}$$

$$= \begin{bmatrix} -\tilde{c} & \tilde{c} & 0 & 0 \end{bmatrix} \begin{bmatrix} -x_1/\tau \\ \omega_h x_{h2} \\ -\omega_h x_{h1} \\ 0 \end{bmatrix} = \tilde{c}\left(\omega_h x_{h2} + \frac{x_1}{\tau}\right)\Big|_{\tilde{c}=1} = \omega_h x_{h2} + \frac{x_1}{\tau} \tag{5.12}$$

Recapitulation 5.1 (Control Law for Dynamic System with Relative Reference)
Using equation (3.4), the control law can be summarized as:

$$u(\mathbf{x}) = \sigma(\mathbf{x})k_0(\mathbf{x}) + (1 - \sigma(\mathbf{x}))\tilde{u}(\mathbf{x})$$

where

$\sigma(\mathbf{x})$	is calculated from equation (3.5) with $B(\mathbf{x})$ found in equation (5.9) and ε found in equation (5.11)
$k_0(\mathbf{x})$	is calculated from equation (3.6) with the Lie derivatives from (5.10) and (5.12)
$\tilde{u}(\mathbf{x})$	is calculated from equation (5.5)

This completes the control design.

5.4 MATLAB Implementation and Results

The results presented in this section are implemented with the following characteristics:

- Extrapolation by means of forward Euler.
- A sampling rate $f_s = 2\,\text{kHz}$. The current realistic sampling rate at $f_s = 100\,\text{Hz}$ is not plotted as the forward Euler method proves itself unstable for this system at a 100 Hz sampling rate.
- Simulation time is 5 s.
- The distance d_{ref} is initially set to 3 cm which is safe. At 2 s the reference distance d_{ref} is altered to -1 cm emulating a surgeon who accidentally forced the robot to penetrate the heart. The expected outcome of this is obviously that the safety controller prevents this by ensuring a distance between end effector and the beating heart.
- Initial conditions are set such that the robot end effector is positioned at a distance to the heart greater than the desired distance. The expected outcome is that the system will attempt to track the distance $x_{\text{ref}} - x_{h1} = d_{\text{ref}}$.

The MATLAB implementation itself can be found in appendix G and in appendix J under the path `matlab_scripts/beating_heart/beating_heart_controller.m`. The state trajectory is plotted in figure 5.2.

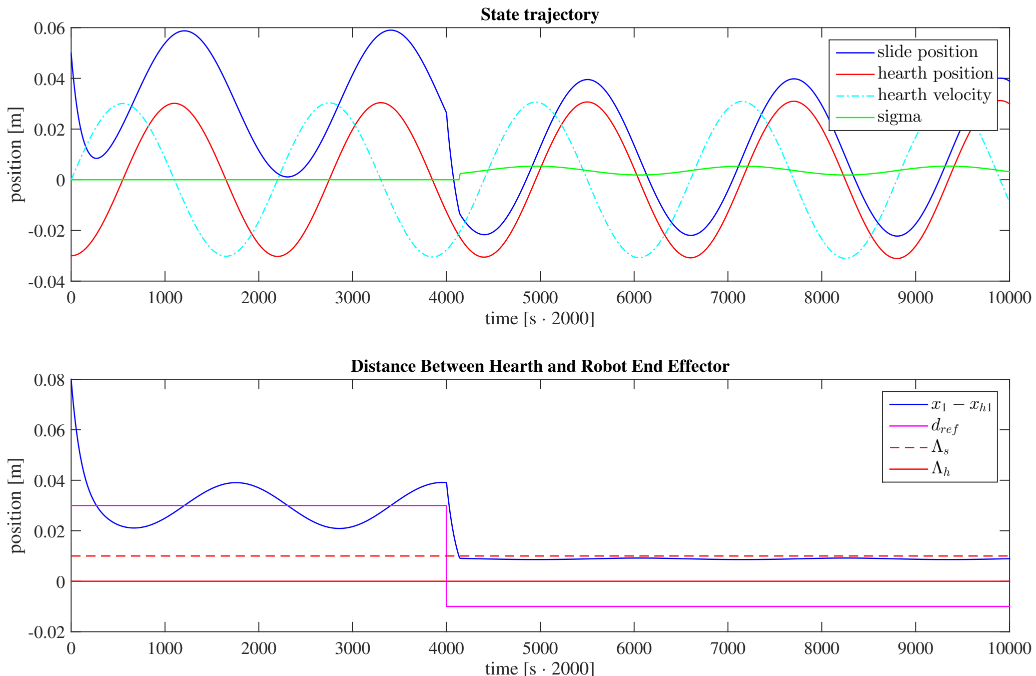


Figure 5.2: Dynamic system with heart position defined as the zero level set of the CBF. References are given as relative distances to the moving surface. When a reference outside the safe region is given, the safety controller prevents the tool from entering the unsafe region.

It is from figure 5.2 seen that the robot end effector settles at a distance at $d_{\text{â€¢}}tref = 3\,\text{cm}$ until the simulation reaches 2 s ($k = 4000$). It is, however, seen that d_{ref} is not constant which is due to the lack of integral action in the system. This could obviously be resolved by including integral action in the controller. At 2 s d_{ref} changes to -1 cm which causes the safety controller to react. The state x_1 settles at a safe distance from the beating heart which is left untouched, and the simulated controller is thereby verified to work as intended.

5.5 Implementation on the da Vinci Robot

A complete description on how to run the developed framework is found in section A.3.

For the implementation of the beating heart controller, the setup depicted in figure 5.3 is used. A tissue phantom representing the heart is mounted on top of a cylinder controlled by a motor to move in a sinusoid. In the motor controller interface the amplitude, frequency and offset of the sinusoid can be set.

The algorithm that implements the beating heart controller is outlined in figure 5.5. The developed software can be found in appendix J and in appendix H. The execution time is not shown here as it is verified in figure 4.15 and figure 4.18 that it is far below the allowed maximum execution time and this controller does not constitute heavier calculations. The measured state trajectory is plotted on figure 5.4.

It can from figure 5.4 be seen how the end effector moves along with the beating heart with a nearly constant distance d_{ref}. It is seen how the safety controller ensures that $x_1 > x_{h1}$ for all t even when the distance d_{ref} is set to the unsafe value -2 cm. It is also seen how d_{ref} can be set to various values. The lack of integral action is again exposed in the plot. It is also seen how the trajectory fluctuates significantly more than the simulated response from figure 5.2 which is due to the far lower sampling rate.

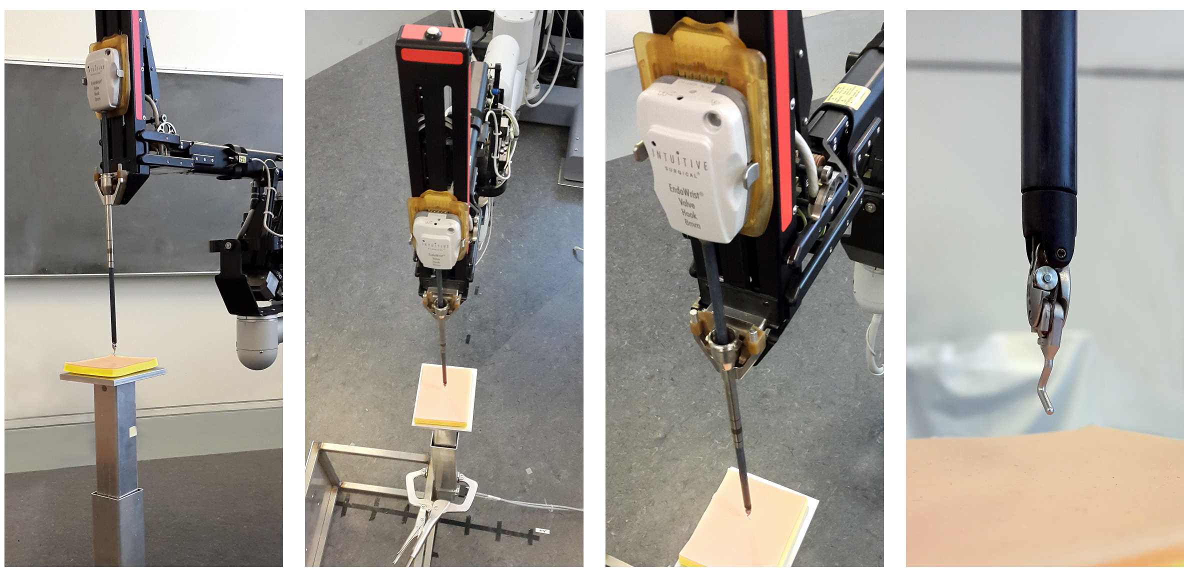


Figure 5.3: Setup of the beating heart implementation. A tissue phantom is mounted on a cylinder controlled by a motor to move as a sinusoid. The robot end effector is following the movement at a distance d_{ref}.

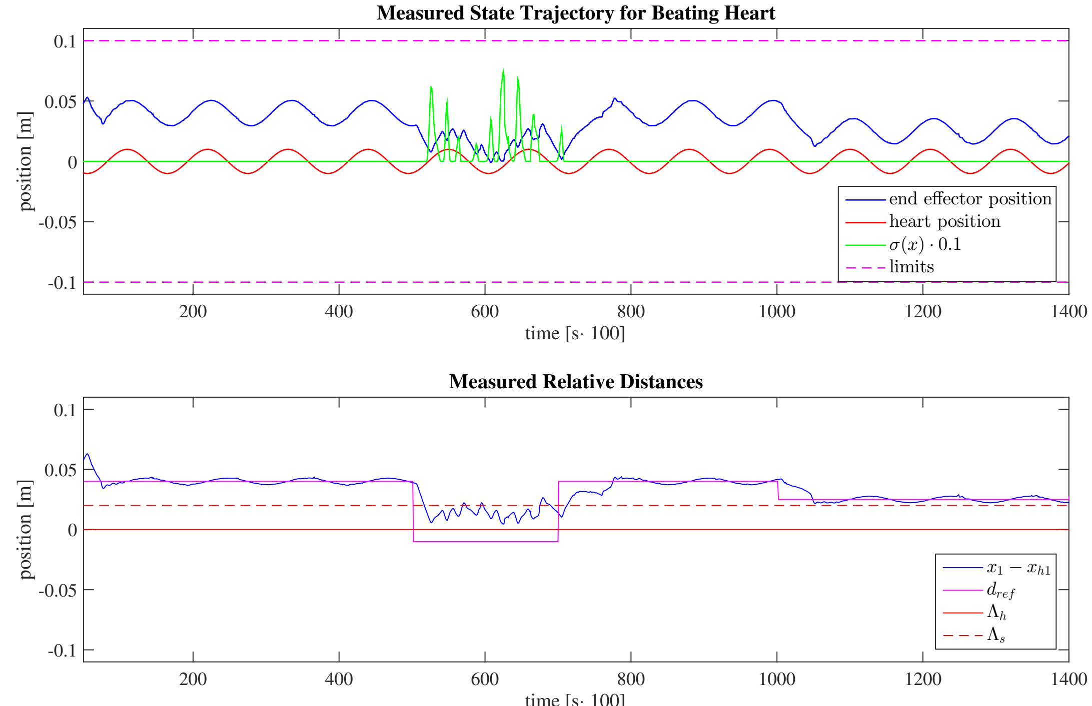


Figure 5.4: It is seen how x_1 moves along with the beating heart with a nearly constant distance d_{ref}. It is seen how the safety controller ensures that $x_1 > x_{h1}$ for all t even when the distance d_{ref} is set to the unsafe value -2 cm. It is also seen how d_{ref} can be set to various values. Measurement files and plotting details can be found in appendix J under the path `measurements/beating_heart`.

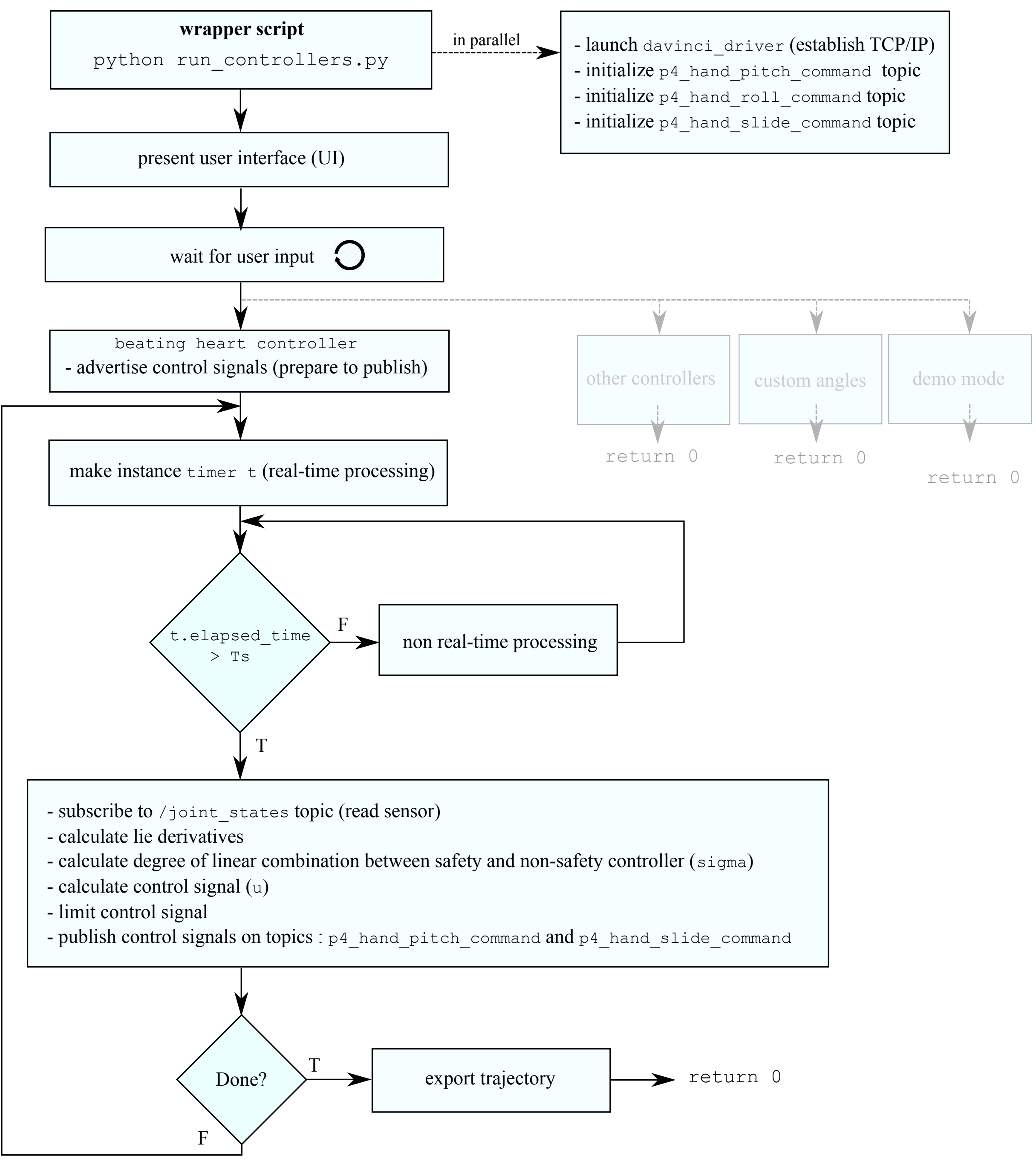


Figure 5.5: Algorithm for dynamic slide safety controller. The source code associated with the algorithm can be found in appendix H and in appendix J. It can also be found at github at the Robotic Surgery Group - Aalborg University under the repository gr1032 (*https://github.com/AalborgUniversity-RoboticSurgeryGroup/*). Note that the control signal is published to the pitch_command topic as well. This is used to change the dynamics in the *x*-direction in case of resonance.

5.6 Conclusion for the Beating Heart Controller

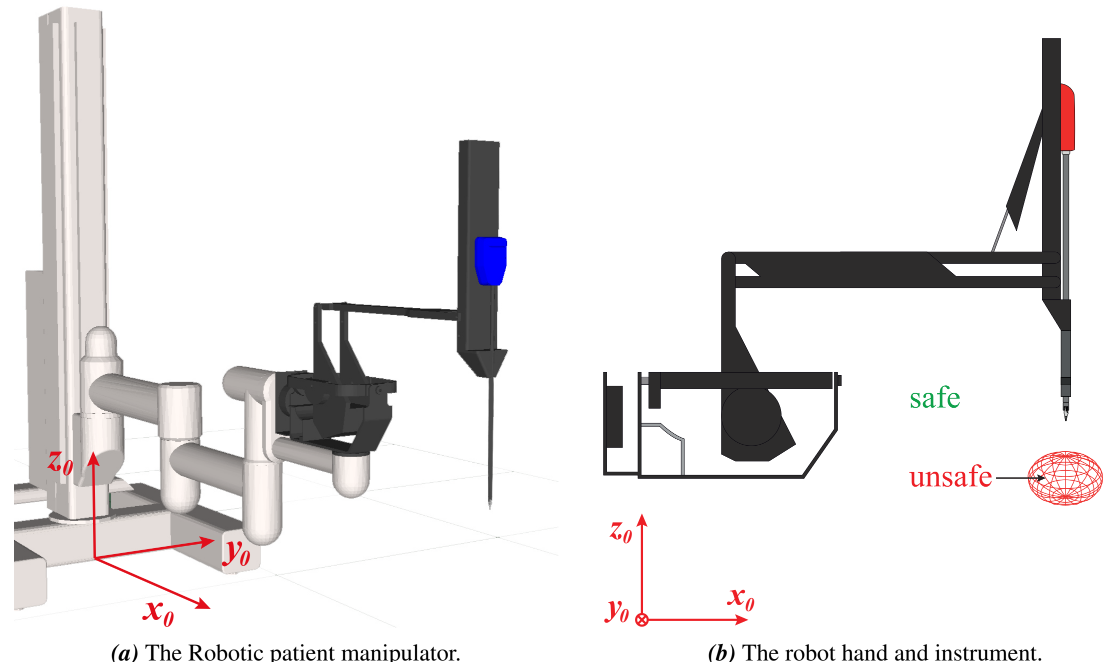
In conclusion, this chapter verifies that a control barrier function can be found which indeed ensures safety for the beating heart following controller, which takes a relative position in the form of the distance to the heart d_{ref} as input. It is seen how the developed controller would benefit from a higher sampling rate. It is also seen that the controller suffers from the lack of integral action, and future work on this topic should include integral action as well.

Finally it is noted that the synchronization of the motor controlled sinusoid movement and the modelled sinusoid implemented on the robot, should be automated for a future test setup through measurements to allow for model adjustment.

Chapter 6

Safety in the 3D Euclidean Space

This chapter presents the design of a safe controller for the robotic patient manipulator in 3D Cartesian space, where the unsafe region is defined as an ellipsoid with static boundaries, i.e. fixed in space with respect to time. The unsafe region is defined such that it is within the reach of the instrument tip. It is desired to construct the unsafe area such that the interior of the ellipsoid is unsafe, thus representing a heart or another vital sensitive organ which must not be cut under any circumstances. An important simplification is made throughout the entire chapter, i.e. the orientation of the hand is not considered.



(a) The Robotic patient manipulator.

(b) The robot hand and instrument.

Figure 6.1: Patient manipulator and a fixed region within the reachable space $\mathcal{X}$ that is unsafe, $\mathcal{X}_u$, marked by a red ellipsoid thus implying an ellipsoid as zero level set of the CBF.

Given the reachability of the robotic end effector which is tested to be roughly a square with the measures [0.2 0.2 0.2] m centered in (0,0,0), the sets considered in this chapter can be outlined in table 6.1.

$\mathcal{X}$	$\mathcal{X}_u$	$\mathcal{X}_0$	$\mathcal{T}$
$\mathcal{X} = \{x \in [-\bar{\Lambda}_{\text{lim}}, \bar{\Lambda}_{\text{lim}}],$ $y \in [-\bar{\Lambda}_{\text{lim}}, \bar{\Lambda}_{\text{lim}}],$ $z \in [-\bar{\Lambda}_{\text{lim}}, \bar{\Lambda}_{\text{lim}}]\}$	$\mathcal{X}_u$ consist of an ellipsoid with semi-axes r_x, r_y, r_z.	$\mathcal{X}_u^c = \mathcal{X} \backslash \mathcal{X}_u$	$\mathcal{T}$ is a layer around the ellipsoid with thickness = 0.01 m.

Table 6.1: Sets considered in this chapter, where $r_x = 0.03$ m, $r_y = 0.06$ m, $r_z = 0.03$ m are lengths of the semi-axes of an ellipsoid and $\bar{\Lambda}_{\text{lim}} = 0.1$m being the extremity of the box encircling the reachable space.

The theory, analysis and implementation aspects presented in this chapter differ from the previous work in this report. In contrast to chapter 4 and chapter 5, which solely comprise the prismatic slide joint, this chapter concerns the five independent revolute joints as well (see figure 1.4). Accordingly, this chapter presents the material required to be able to manoeuvre the da Vinci robot in the 3D Cartesian space and to be able to specify an unsafe set $\mathcal{X}_u$ contained within an ellipsoid. This implies the topics:

- Development of a sufficient model describing movement in three dimensions.
- Construction of a control barrier function fulfilling the demand for an ellipsoid encirclement of $\mathcal{X}_u$.
- Development of the control system.
- Implementation in MATLAB.
- Implementation on the da Vinci robot. This entails a kinematic description of the robot links and joints such that a translation between the 3D Cartesian space and the 6D joint space can be established.

Thus the first topic will be to model the movement sufficiently.

6.1 Modelling of Robot Hand Movement in 3D

The system shall be modelled as a linear state space system on the form:

$$\begin{aligned}\dot{\mathbf{x}} &= \mathbf{A}\,\mathbf{x} + \mathbf{B}\,\mathbf{u} \\ \mathbf{y} &= \mathbf{C}\mathbf{x}\end{aligned}$$

The movement in the three dimensional space may be modelled as three decoupled systems. While there may be coupling between the systems, the decoupling simplifies the modelling phase significantly and it is still a realistic version of the real scenario.

$$\dot{\mathbf{x}} = \dot{\begin{bmatrix} x_x \\ x_y \\ x_z \end{bmatrix}} = \underbrace{\underbrace{\begin{bmatrix} -1/\tau_x & 0 & 0 \\ 0 & -1/\tau_y & 0 \\ 0 & 0 & -1/\tau_z \end{bmatrix}}_{\mathbf{A}} \begin{bmatrix} x_x \\ x_y \\ x_z \end{bmatrix}}_{f(\mathbf{x})} + \underbrace{\begin{bmatrix} 1/\tau_x & 0 & 0 \\ 0 & 1/\tau_y & 0 \\ 0 & 0 & 1/\tau_z \end{bmatrix}}_{g(\mathbf{x})=\mathbf{B}} \mathbf{u} \tag{6.1}$$

where

x_x	is the position in the x-axis
x_y	is the position in the x-axis
x_z	is the position in the x-axis
τ_x	is the time constant associated with a step purely in the x-axis
τ_y	is the time constant associated with a step purely in the y-axis
τ_z	is the time constant associated with a step purely in the z-axis

The time constants are measured and as presented in section F.2 are found to be:

$$\tau_x = 0.070\,\mathrm{s}, \qquad \tau_y = 0.100\,\mathrm{s}, \qquad \tau_z = 0.040\,\mathrm{s}$$

With a sufficient model established the CBF can be considered.

6.2 Construction of CBF

A CBF is proposed that complies with the first two constraints in Definition 2.2, i.e. a function that is positive on the set $\mathcal{X}_u$ and nonpositive on the set $\mathcal{X}_0$. In order to make sure that the robot tool can be prevented from penetrating the heart or another desired three dimensional region, the unsafe set $\mathcal{X}_u$ is defined as an ellipsoid. This ellipsoid enclosing the region $\mathcal{X}_u$ as visualized in figure 6.1b, must be the zero level set of the CBF. The CBF is proposed of the form:

$$B(\mathbf{x}) = -\left(\left(\frac{x_x - c_x}{r_x}\right)^2 + \left(\frac{x_y - c_y}{r_y}\right)^2 + \left(\frac{x_z - c_z}{r_z}\right)^2 - 1\right) \tag{6.2}$$

where

$[c_x\ c_y\ c_z]$ is the coordinate of the center of the ellipsoid, $\mathbf{c} \in \mathbb{R}^3$

$[r_x\ r_y\ r_z]$ is the lengths of the semi-axes of the ellipsoid, $\mathbf{r} \in \mathbb{R}^3_+$

The CBF is visualized in figure 6.2. The zero level set enclosing the unsafe region $\mathcal{X}_u$ shown as a green ellipsoid. Note that the function $-B(\mathbf{x})$ will have the same zero level set, but have positive values outside the ellipsoid, indicating that it is the safe area $\mathcal{X}_0$ that is enclosed by the ellipsoid.

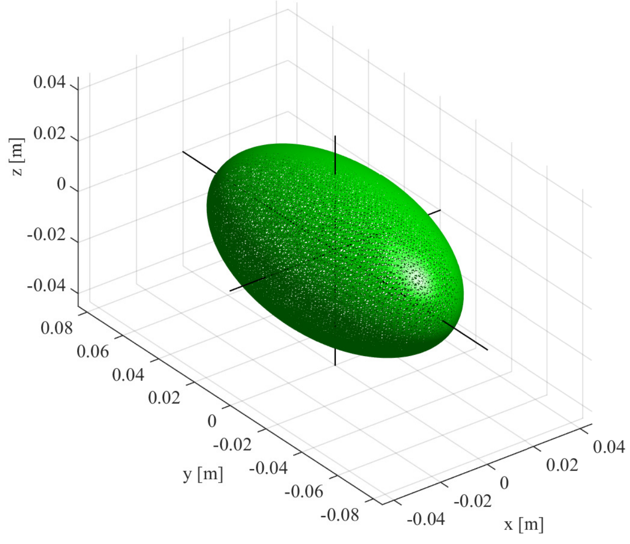


Figure 6.2: CBF on the form described in equation (6.2) with $\mathbf{c} = [0,\ 0,\ 0]$ and $\mathbf{r} = [0.03,\ 0.05,\ 0.03]$.

As suggested in equation (3.1)b, if $L_f B(\mathbf{x}) \neq 0\ \forall \mathbf{x}$ then safety can always be guaranteed. Thus testing $L_f B(\mathbf{x})$:

$$L_g B(\mathbf{x}) = \frac{dB(\mathbf{x})}{d\mathbf{x}} g(\mathbf{x}) \Big|_{g(\mathbf{x})=\mathbf{B}} = \begin{bmatrix} \frac{\partial B(\mathbf{x})}{\partial x_x} & \frac{\partial B(\mathbf{x})}{\partial x_y} & \frac{\partial B(\mathbf{x})}{\partial x_z} \end{bmatrix} \begin{bmatrix} 1/\tau_x & 0 & 0 \\ 0 & 1/\tau_y & 0 \\ 0 & 0 & 1/\tau_z \end{bmatrix}$$

$$= \begin{bmatrix} \frac{2}{r_x^2}(c_x - x_x) & \frac{2}{r_y^2}(c_y - x_y) & \frac{2}{r_z^2}(c_z - x_z) \end{bmatrix} \begin{bmatrix} 1/\tau_x & 0 & 0 \\ 0 & 1/\tau_y & 0 \\ 0 & 0 & 1/\tau_z \end{bmatrix}$$
$$= 2 \begin{bmatrix} \frac{1}{r_x^2 \tau_x}(c_x - x_x) & \frac{1}{r_y^2 \tau_y}(c_y - x_y) & \frac{1}{r_z^2 \tau_z}(c_z - x_z) \end{bmatrix} \tag{6.3}$$
$$\neq 0 \quad \forall \quad x_x \neq c_x, \quad x_y \neq c_y, \quad x_z \neq c_z$$

It can be seen that $L_g B(\mathbf{x}) = 0$ in the centre of the ellipsoid $[c_x, c_y, c_z]$, which is the vertex of the barrier function. While $L_g B(\mathbf{x}) = 0$ implies the demand $L_f B(\mathbf{x}) \leq 0$, it causes no issue in this specific case. Simply subtract a small region around the center of the ellipsoid, call it $\mathcal{S} = \{x_x = [-\delta \;\; \delta],\, x_y = [-\delta \;\; \delta]\; x_z = [-\delta \;\; \delta]\}$ where δ is some small scalar, from $\mathcal{X}$ such that $\bar{\mathcal{X}} = \mathcal{X} \backslash \mathcal{S}$. Thus $B(\mathbf{x})$ is valid on $\bar{\mathcal{X}}$. These considerations are valid as no trajectory under any circumstances will penetrate the surface of the ellipsoid, thus no reason to include the interior in $\mathcal{X}$. With $L_g B(\mathbf{x}) \neq 0 \; \forall \mathbf{x} \in \bar{\mathcal{X}}$, the control design can be initiated.

6.3 Control Design

The control design is built upon the control law from equation (3.4), i.e:

$$u(\mathbf{x}, \tilde{\mathbf{u}}) = \sigma(\mathbf{x}) k_0(\mathbf{x}) + (1 - \sigma(\mathbf{x}))\tilde{u}(\mathbf{x}) \tag{6.4}$$

with:

$$\tilde{u}(\mathbf{x}) = \bar{\mathbf{N}} x_{\text{ref}} - \mathbf{K}\mathbf{x} \tag{6.5}$$

where

$u(\mathbf{x}, \tilde{u})$	is the control input where safety ensured, $u(\mathbf{x}, \tilde{u}) \in \mathbb{R}^3$
$\tilde{u}(\mathbf{x})$	is the input where no safety is considered, $\tilde{u}(\mathbf{x}) \in \mathbb{R}^3$
$k_0(\mathbf{x})$	is the controller ensuring safety, $k_0(\mathbf{x}) \in \mathbb{R}^3$
$\sigma(\mathbf{x})$	founds a linear combination between the two control laws, $\sigma(\mathbf{x}) \in \mathbb{R}$
$\bar{\mathbf{N}}$	ensures unity gain from reference input to output, $\bar{\mathbf{N}} \in \mathbb{R}^{3\times3}$
$\mathbf{K}$	is the gain in the controller where no safety is considered, $\mathbf{K} \in \mathbb{R}^{3\times3}$
$\mathbf{x}$	is the state vector, $\mathbf{x} = [x_x \quad x_y \quad x_z]^T \in \mathbb{R}^3$

The input $\tilde{u}(\mathbf{x})$ is used in the safe area and is found according to regular pole placement where stability is the main design consideration, thus the MATLAB command `acker` is simply used to places the poles faster than the system itself and in the left half plane in the complex frequency domain:

$$\mathbf{K} = \texttt{acker}\Big(\mathbf{A}, \mathbf{B}, \Big[-15 \quad -15 \quad -15\Big]\Big) = \begin{bmatrix} 0.050 & 0 & 0 \\ 0 & 0.500 & 0 \\ 0 & 0 & -0.385 \end{bmatrix} \tag{6.6}$$

The matrix $\bar{\mathbf{N}}$ is found as [Stoustrup, 2014]:

$$\bar{\mathbf{N}} = -\left(\mathbf{C}\mathbf{A}_{cl}^{-1}\mathbf{B}\right)^{-1} = \begin{bmatrix} 1.050 & 0 & 0 \\ 0 & 1.500 & 0 \\ 0 & 0 & 0.615 \end{bmatrix} \tag{6.7}$$

The safety controller $k_0(\mathbf{x})$ requires both $L_g B(\mathbf{x})$ and $L_f B(\mathbf{x})$. With $L_g B(\mathbf{x})$ found in equation (6.3),

$L_f B(\mathbf{x})$ is found to:

$$
\begin{aligned}
L_f B(\mathbf{x}) &= \frac{dB(\mathbf{x})}{d\mathbf{x}} f(\mathbf{x}) \Big|_{f(\mathbf{x})=\mathbf{A}\mathbf{x}} = \begin{bmatrix} \frac{\partial B(\mathbf{x})}{\partial x_x} & \frac{\partial B(\mathbf{x})}{\partial x_y} & \frac{\partial B(\mathbf{x})}{\partial x_z} \end{bmatrix} \mathbf{A}\mathbf{x} \\
&= \begin{bmatrix} \frac{2}{r_x^2}(c_x - x_x) & \frac{2}{r_y^2}(c_y - x_y) & \frac{2}{r_z^2}(c_z - x_z) \end{bmatrix} \begin{bmatrix} -1/\tau_x & 0 & 0 \\ 0 & -1/\tau_y & 0 \\ 0 & 0 & -1/\tau_z \end{bmatrix} \begin{bmatrix} x_x \\ x_y \\ x_z \end{bmatrix} \\
&= -2\left(\frac{1}{r_x^2 \tau_x}(c_x x_x - x_x^2) + \frac{1}{r_y^2 \tau_y}(c_y x_y - x_y^2) + \frac{1}{r_z^2 \tau_z}(c_z x_z - x_z^2) \right)
\end{aligned} \tag{6.8}
$$

The control law presented in equation (6.5) suggests a smooth transition on $\mathcal{T}$, just as derived in chapter 4 and chapter 5. This obeys with the desire to cover the ellipsoid with a 1 cm thick rim (from table 6.1), such that the scalar $\varepsilon > 0$ is introduced according to equation (3.5), i.e.:

$$
\sigma(\mathbf{x}) = \begin{cases} 0 & \text{if } B(\mathbf{x}) \leq -\varepsilon \\ -2\left(\frac{B(\mathbf{x})}{\varepsilon}\right)^3 - 3\left(\frac{B(\mathbf{x})}{\varepsilon}\right)^2 + 1 & \text{if } B(\mathbf{x}) \in (-\varepsilon, 0) \\ 1 & \text{if } B(\mathbf{x}) \geq 0 \end{cases} \tag{6.9}
$$

where ε can be found by considering the CBF:

$$
\varepsilon = B(\mathbf{x})\Big|_{\substack{x_z = 0 \\ x_x = r_x + 0.01 \\ x_y = 0}} = -\left(\left(\frac{r_x + 0.01 - c_x}{r_x} \right)^2 + \left(\frac{c_y}{r_y} \right)^2 + \left(\frac{c_z}{r_z} \right)^2 - 1 \right)\Bigg|_{\substack{r_x = 0.03 \\ r_y = 0.06 \\ r_z = 0.03 \\ c_x = 0 \\ c_y = 0 \\ c_z = 0}} = 0.778 \tag{6.10}
$$

Note that setting $x_x = r_x + 0.01$ and $x_y = x_z = 0$ ensures that a 1 cm thick transition layer is designated around the ellipsoid. Thus, from equation (3.6), the non-linear controller ensuring safety can be found as:

$$
k_0(\mathbf{x}) = \begin{cases} -\frac{L_f B(\mathbf{x}) + \sqrt{(L_f B(\mathbf{x}))^2 + \kappa^2 L_g B(\mathbf{x})(L_g B(\mathbf{x}))^T}}{L_g B(\mathbf{x})(L_g B(\mathbf{x}))^T} (L_g B(\mathbf{x}))^T & \text{if } L_g B(\mathbf{x}) \neq 0 \\ 0 & \text{if } L_g B(\mathbf{x}) = 0 \end{cases} \tag{6.11}
$$

The control law can thereby be summarized.

Recapitulation 6.1 (Control Law for Safety Controller in the Euclidean Space)
Using equation (6.4), the control law can be summarized as:

$$
u(\mathbf{x}, \tilde{u}) = \sigma(\mathbf{x}) k_0(\mathbf{x}) + (1 - \sigma(\mathbf{x}))\tilde{u}(\mathbf{x}) \tag{6.12}
$$

where

$\sigma(\mathbf{x})$ is calculated in equation (6.9) with ε from equation (6.10)

$k_0(\mathbf{x})$ is calculated from equation (6.11) with Lie derivatives from (6.3) and (6.8)

$\tilde{u}(\mathbf{x})$ is calculated from equation (6.5) with $\bar{\mathbf{N}}$ from (6.6) and $\mathbf{K}$ from (6.7)

6.4 MATLAB Implementation

The MATLAB implementation can be found in found in appendix G and in appendix J under the path `matlab_scripts/safe_3d/safety_in_3d.m`. The implementation is built upon these considerations:

- Forward Euler extrapolation.
- Sampling time $f_s = 2\,\text{kHz}$.
- Simulation time at 10 s.
- Various setpoints are given to illustrate how the controller ensures that the trajectory is redirected in alternative paths to ensure that the ellipsoid is not penetrated at any time.

The trajectory (blue) along with the immediate path (red) between the given setpoints (blue stars) and the unsafe set outlined by $B(\mathbf{x}) = 0$ as an ellipsoid (green) are plotted in figure 6.3. The black circle indicates the initial position (outside $\mathcal{X}_u$) of the trajectory and the destination coordinate is indicated as a green circle.

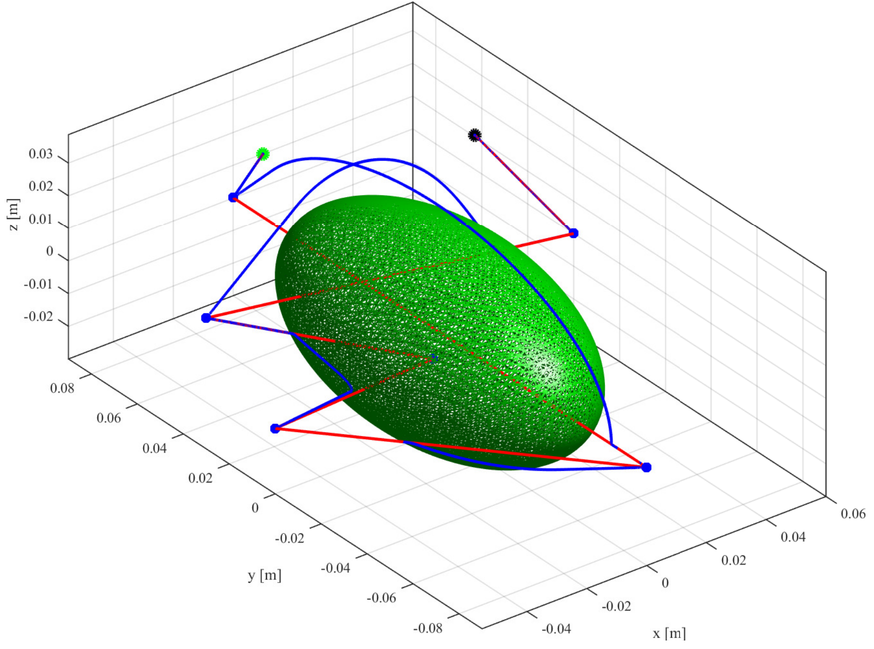


Figure 6.3: Result of the MATLAB implementation of the safe controller which allows manoeuvring in the Euclidean space with an ellipsoid as zero level set of the CBF outlining $\mathcal{X}_u$. The trajectory (blue) along with the immediate trajectory (red) determined from the setpoints given (blue stars). The black circle indicates initial position and the green circle indicates the destination.

It is from figure 6.3 seen how the controller ensures that the state never enters the interior of the ellipsoid which indeed was the purpose of the controller. It is also seen how the controller ensures a smooth and elegant detour to reach the desired setpoint while ensuring safety. If a setpoint is given in the interior of $\mathcal{X}_u$, the state will settle at the shortest safe distance from that setpoint. It is finally seen how setpoints in the safe area are reached without any problems.

The state trajectory is also plotted in figure 6.4 where each coordinate trajectory is plotted individually which makes it slightly easier to see the effect of $\sigma(\mathbf{x})$.

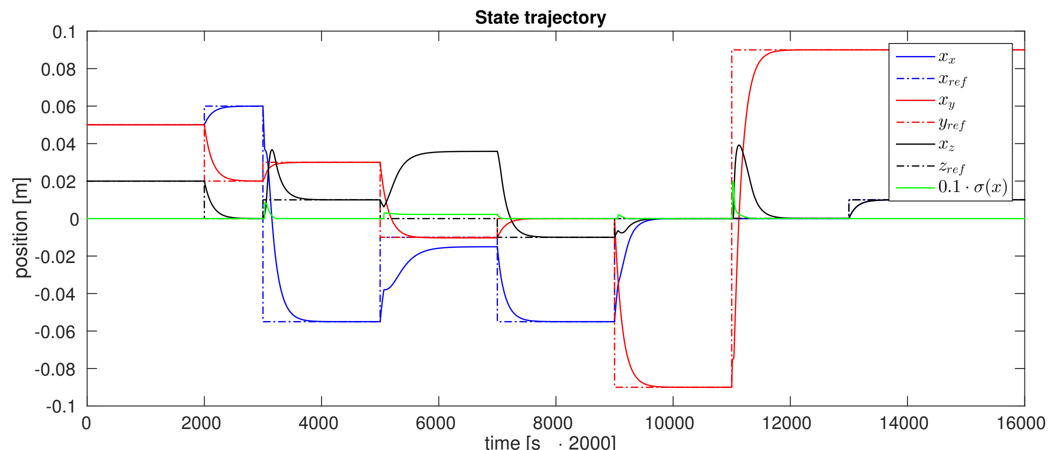


Figure 6.4: The same trajectory as depicted in figure 6.3. It is seen how $\sigma(\mathbf{x})$ increases in value when the trajectory approaches the zero level set of $B(\mathbf{x})$ and the position is adjusted accordingly.

It is from figure 6.4 seen how $\sigma(\mathbf{x})$ increases in value when the trajectory approaches the zero level set of $B(\mathbf{x})$, and the position is adjusted accordingly. Additionally, figure 6.4 is the only plot where the actual dynamics can be seen, which clearly demonstrate the first order approximations in both x, y and z. It is also seen how the dynamics is completely changed when the trajectory enters $\mathcal{T}$, hence causing a highly non-linear system, for which it actually is very difficult to predict the exact behaviour, except that it will escape the unsafe area.

The thickness of 1 cm, outlining $\mathcal{T}$, ensured by ε, is visualized in figure 6.5 along with the simulated trajectory.

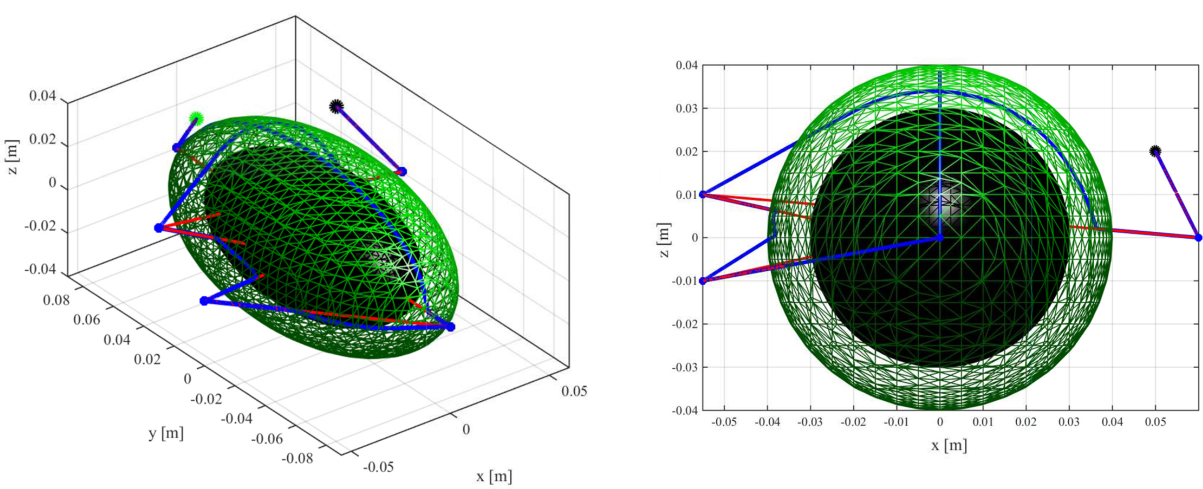


Figure 6.5: The inner (black) ellipsoid marks the surface of $\mathcal{X}_u$ and the rim between the two ellipsoids is $\mathcal{T}$. Thus $\mathcal{X}_0 = \mathcal{X} \backslash \mathcal{X}_u$. The trajectory is initiated from the black circle.

It is from figure 6.5 seen that the trajectory starts to be redirected when it enters $\mathcal{T}$ in a smooth manner. The trajectory will always take the shortest safe path.

6.5 Implementation on the da Vinci Robot

In contrast to the MATLAB implementation, the implementation on the da Vinci robot comprises some additional topics. The code developed as a result of the implementation can be found in appendix H, appendix I and appendix J. It can also be cloned from GitHub with the following git commands:

- `git clone https://github.com/AalborgUniversity-RoboticSurgeryGroup/Gr1032`
- `git clone https://github.com/AalborgUniversity-RoboticSurgeryGroup/davinci_description --branch reduced_robot`
- `git clone https://github.com/AalborgUniversity-RoboticSurgeryGroup/davinci_driver --branch gr1032`

It is also important to setup the ROS framework as described in appendix A. A complete description on how to run the developed framework is found in section A.3. The implementation of the controller derived in this chapter consists of the topics stated below:

- A description of the kinematic chain. This is required to compute the end effector position. The kinematic description is written in the ROS framework, located in the `davinci_description` package under the folder `robots` as a number of `xacro` (XML macros) files. These `xacro` files are at launch time converted to a Unified Robot Description Format (URDF) such that kinematic solvers can access the information. The ROS framework and file system is described more thoroughly in appendix A.
- Integration of an inverse kinematic solver such that a desired (x, y, z) position can be obtained from the six active joint angles (see figure 1.4), i.e. from:
  - `p4_hand_roll`
  - `p4_hand_pitch`
  - `p4_instrument_slide`
  - `p4_instrument_roll`
  - `p4_instrument_pitch`
  - `p4_instrument_jaw_right`
- Integration of a forward kinematic solver such that the position can be read from the six angles.
- An original description of the kinematic chain is present at the Robotic Surgery Group - Aalborg University at GitHub. However, a modification of the kinematic chain is required such that only active joints are a part of the kinematic chain. This is necessary for the kinematic solver as it will not be able to distinct passive joints from active.
- Algorithm development such that the above topics are connected and real-time signal processing is ensured. As discussed in section 4.5, this should be done in C++ in the ROS framework.

Thus before proceeding to the actual implementation, a kinematic description is necessary.

6.5.1 Kinematics of the AAU da Vinci Robot

The upcoming subsections will first describe how kinematics in general are defined, then how the kinematics are desired to be defined for the da Vinci robot followed by the original kinematic description (the structure of the `xacro` files when this project was initiated). It concludes with a necessary modification of the original kinematic description such that a sufficiently fast solver can be applied.

How a Kinematic Description is Defined

A kinematic description of an object requires defining a right-handed coordinate frame fixed in the object and a coordinate frame fixed in inertial space, the latter which the position and orientation of the object can be described relative to. This relative orientation and position of an object (or the frame i fixed in it) with respect to another frame $j = i - 1$ can be described through a transformation matrix $\mathbf{T}$, containing the orthonormal rotation matrix $\mathbf{R}$ and the translation vector $\mathbf{p}$ of the frame origin, as

$$ {}_i^j\mathbf{T} = \begin{bmatrix} \mathbf{R} & \mathbf{p} \\ 0 & 1 \end{bmatrix}, \qquad \text{where} \qquad \mathbf{p} = \begin{bmatrix} a \\ b \\ d \end{bmatrix} \tag{6.13} $$

and the simplest rotation matrices $\mathbf{R}$ are rotations about a single axis, which can be combined to obtain an arbitrary rotation

$$ \mathbf{R}_x(\alpha) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha \\ 0 & \sin\alpha & \cos\alpha \end{bmatrix} \qquad \mathbf{R}_y(\beta) = \begin{bmatrix} \cos\beta & 0 & \sin\beta \\ 0 & 1 & 0 \\ -\sin\beta & 0 & \cos\beta \end{bmatrix} \qquad \mathbf{R}_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{6.14} $$

where
α, β, θ are angles
$\mathbf{R}_x(\alpha)$ is rotation around the x-axis
$\mathbf{R}_y(\beta)$ is rotation around the y-axis
$\mathbf{R}_z(\theta)$ is rotation around the z-axis

For the robotic patient manipulator, which comprises a number of links, coordinate frames are defined for each degree of freedom, i.e. placed in each joint such that one of the frame axes is the axis of free rotation or translation. The kinematic chain is now the sequence of alternate links and joints starting from the link fixed in inertial space and ending at the tip of the robotic tool. The link preceding a joint is its parent link, while the link succeeding it is its child link. The transformation between any two frames is given as the product of the transformation matrices in the kinematic chain between them. An example of a sequence of transformations can be seen in appendix C.

Desired Kinematic Description

For the resolution of frame definitions it is preferred to adapt the robot coordinate frame convention Denavit-Hartenberg (DH) because it is one of the most widespread kinematic descriptions in the robot kinematics community, and because it describes transformations between two successive frames in the

kinematic chain on a succinct and standardized form, i.e.:

$$ {}^{i-1}_{i}\mathbf{T} = \begin{bmatrix} \mathbf{R}_z(\theta_i) & \begin{bmatrix} 0 \\ 0 \\ d_i \end{bmatrix} \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \mathbf{R}_x(\alpha_i) & \begin{bmatrix} a_i \\ 0 \\ 0 \end{bmatrix} \\ 0 & 1 \end{bmatrix} \tag{6.15} $$

In the DH kinematic description each frame is fixed with respect to its parent link, and its z-axis is aligned with the actuation axis of its child link. The free rotation/translation always takes place around/along the local z axis, and as seen from equation (6.15) any fixed or free rotation/translation about/along the z-axis is implemented (intrinsically) before any fixed rotation/translation about/along the (new) x-axis. For more details on the DH convention and robot frames defined according to it, see section C.2.

Implementable Kinematic Description

In the robot description in ROS (the implementation framework), however, the kinematics are described in the `xacro` files on the form

$$ {}^{i-1}_{i}\mathbf{T} = \begin{bmatrix} \mathbf{R}_z(\text{yaw})\mathbf{R}_y(\text{pitch})\mathbf{R}_x(\text{roll}) & \begin{bmatrix} a_i \\ b_i \\ d_i \end{bmatrix} \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \mathbf{R}_z(\theta_i^*)\mathbf{R}_y(\beta_i^*)\mathbf{R}_x(\alpha_i^*) & \begin{bmatrix} a_i^* \\ b_i^* \\ d_i^* \end{bmatrix} \\ 0 & 1 \end{bmatrix} \tag{6.16} $$

Here fixed translations are implemented first, then RPY rotations (extrinsic roll (about x-axis), pitch (about y-axis), yaw (about z-axis) rotation), and finally the free rotation or translation (denoted by * in equation (6.16)) about/along one of the rotated axes. Furthermore, the convention here is that each frame (joint) is fixed in its child link (corresponding to the fixed rotations preceding the free rotation), and not in its parent link as in the DH convention. The transformation ${}^{i-1}_{i}\mathbf{T}$ is implemented in joint i in the `xacro` file as (for joint 8)

```
1 <joint name="p4_hand_pitch" type="revolute">
2 <origin
3 xyz="0 0 0"
4 rpy="1.5708 0 0" />
5 <parent link="rcm_vitual0" />
6 <child link="rcm_vitual1" />
7 <axis xyz="0 0 1" />
8 ...
9 </joint>
```

Taking this small code example for the `p4_hand_pitch` joint, line 3 and 4 constitute the fixed translation and rotation of equation (6.16). Line 7 represents the latter part of equation (6.16), and combined with the information given from the `revolute` parameter, it suggests a free rotation about the z-axis (i.e. θ_i^*) and that $\mathbf{R}_x(\alpha_i^*) = \mathbf{R}_y(\beta_i^*) = \mathbf{I}$ and $a_i^* = b_i^* = d_i^* = 0$.

Modifying the Kinematic Description to fit the Kinematic Solver

The indisputable most important application of the kinematic description, is to use it for forward kinematics (FK) and inverse kinematics (IK). The IK is not a trivial task and can be implemented in different ways. Three candidate technologies are described in table 6.2.

Technology	Advantage	Disadvantage	Conclusion
MoveGroup (a thorough test of MoveGroup is made in section A.4.)	This is an API which is easy and simple to use. It offers a GUI to initialize joints and links and certain poses. It is possible to specify joint limits.	It is slow and requires computational time between every setpoint. Also, it implements its own controller destructing the dynamics.	Due to the slow processing time and the shattered dynamics, the solver offered by the MoveGroup API is rejected.
KDL (offered by The Orocos Project) (it is in fact the underlying solver of MoveGroup)	All additional features offered by MoveGroup can be bypassed and the speed can thereby be increased.	Interfacing with KDL directly complicates the code significantly. No specification of joint limits is possible. Cannot distinct between passive and active joints.	This is a good solution with the only real issue being the lack of joint limit specification. Passive joints can be omitted by redefining the kinematic description.
Custom design	Full control of all aspects of the solver and the possibility to tailor each developed feature.	The development of an IK solver for a six DOF manipulator is not a trivial task.	The extra features offered by a custom IK solver does not match the development costs which it will induce.

Table 6.2: Candidate technologies to implement inverse kinematics and forward kinematics.

Weighting the advantages and disadvantages from table 6.2, a proper solver is chosen as the KDL solver. Thus, the original kinematic description/chain needs to be modified to exclude all passive joints such that it only describes the active joints. This is necessary for the KDL solver as it otherwise will attempt to control passive joints. A simplified overview of the problem and how it is solved is outlined in figure 6.6

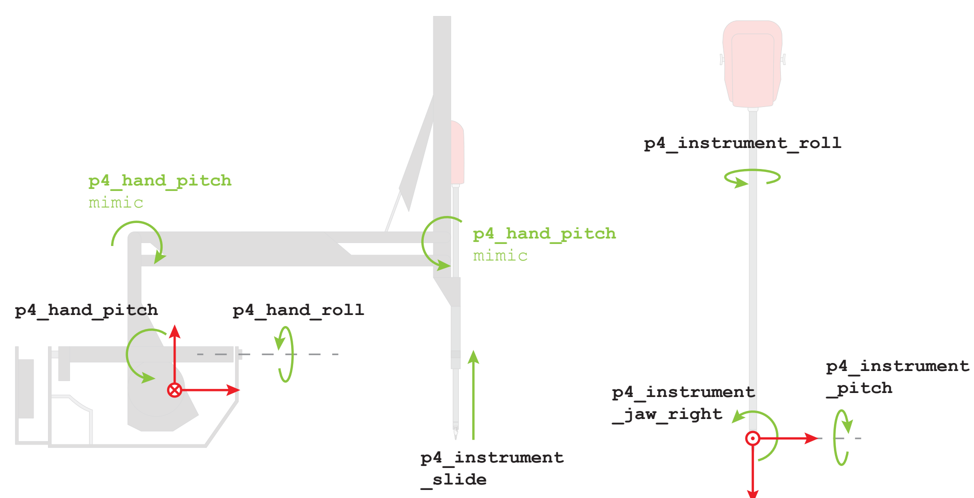


Figure 6.6: Modified kinematic chain. The two mimic joints (green) must be bypassed.

The new modified set of frames is defined for the da Vinci kinematics, adhering to the DH constraint that

all free rotation/translation is about/along the local z-axis. In conclusion the set of frames is modified to comply with the KDL solvers needs, i.e. a kinematic chain consisting of purely active joints.

The new modified kinematic description is depicted more thoroughly in figure 6.7b. The parameters associated with these are listed in table 6.3b. The original kinematic description, containing the two passive joints mimicking the hand pitch movement are depicted in figure 6.7a and the associated parameters are listed in table 6.3a for comparison.

	fixed translation [m]			fixed rotation [rad]			freedom	
frame	a (x)	b (y)	d (z)	roll	pitch	yaw	$\alpha^*, \beta^*, \theta^*$ or d^*	joint name
7	-0.042	0	0.161	0	0	0	α_7^*	`hand_roll`
8	0	0	0	0	-0.288	0	$-\beta_8^*$	`hand_pitch`
9	0.011	0	0.186	π	0.288	0	β_8	mimic `hand_pitch`
10	0.520	0	0	π	0	0	$-\beta_8$	mimic `hand_pitch`
11	0	0	-0.120	0	0	0	d_{11}^*	`instrument_slide`
12	0.052	0	0	π	0	$\pi/2$	θ_{12}^*	`instrument_roll`
13	0	0	0.177	0	0	0	$-\alpha_{13}^*$	`instrument_pitch`
14L	0	0	0.009	$\pi/2$	$\pi/2$	0	$-\theta_{14L}^*$	`instrument_jaw_left`
14R	0	0	0.009	$\pi/2$	$\pi/2$	0	θ_{14R}^*	`instrument_jaw_right`

(a) Original robot hand kinematics including mimicking joints, corresponding to figure 6.7a. The two mimicking joints mimic `hand_pitch` are removed in the modified description.

	fixed translation [m]			fixed rotation [rad]			freedom	
frame	a (x)	b (y)	d (z)	roll	pitch	yaw	θ^* or d^*	joint name
7	0.482	0	0.047	0	$\pi/2$	0	θ_7^*	`hand_roll`
8	0	0	0	$\pi/2$	0	0	θ_8^*	`hand_pitch`
9	0.097	0	0	0	$-\pi/2$	0	d_9^*	`instrument_slide`
10	0	0	0	0	0	0	θ_{10}^*	`instrument_roll`
11	0	0	0	0	$\pi/2$	0	θ_{11}^*	`instrument_pitch`
12L	0.009	0	0	$-\pi/2$	0	0	θ_{12L}^*	`instrument_jaw_left`
12R	0.009	0	0	$-\pi/2$	0	0	θ_{12R}^*	`instrument_jaw_right`

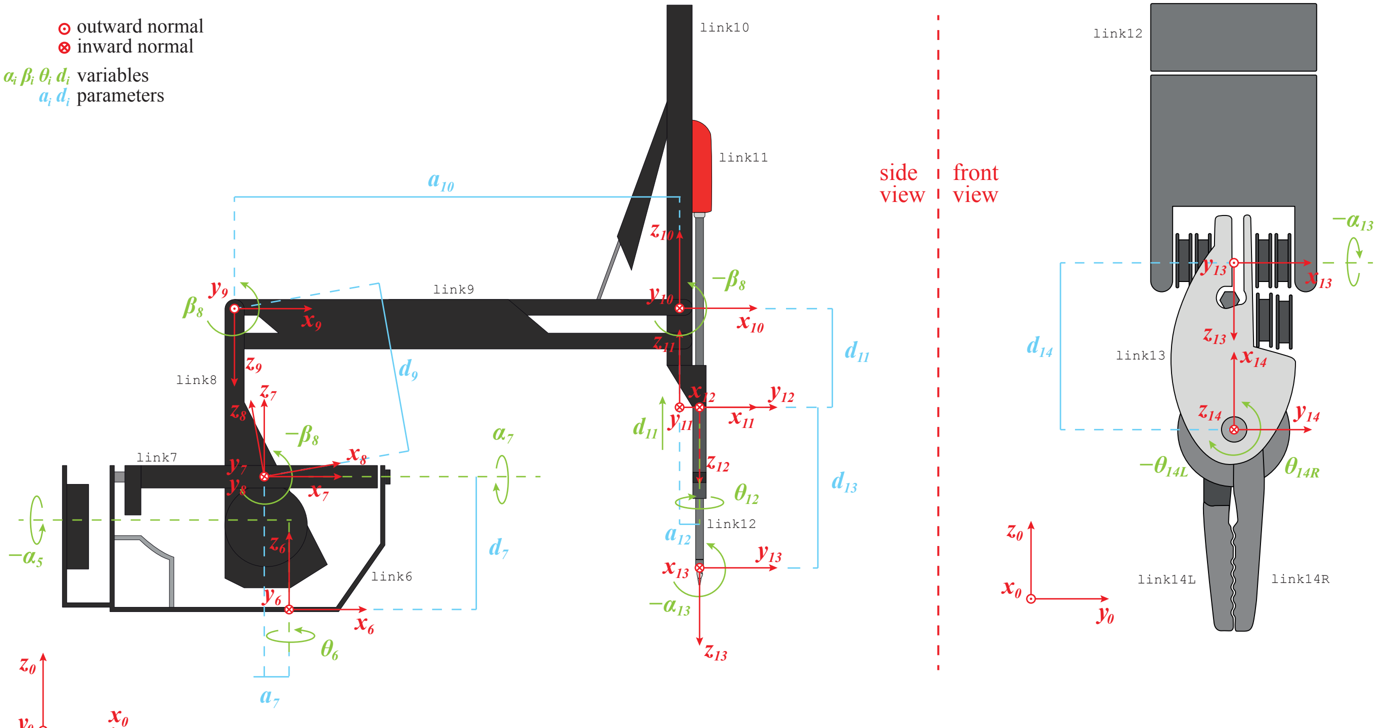
(b) New modified robot hand kinematics excluding mimicking joints, corresponding to figure 6.7b.

Table 6.3: Parameters and variables (marked with *) for the robot kinematic description implemented in ROS and visualized in figure 6.7. Free angles are named with α for rotation about the $+x$-axis, β about the $+y$-axis and θ about the $+z$-axis.

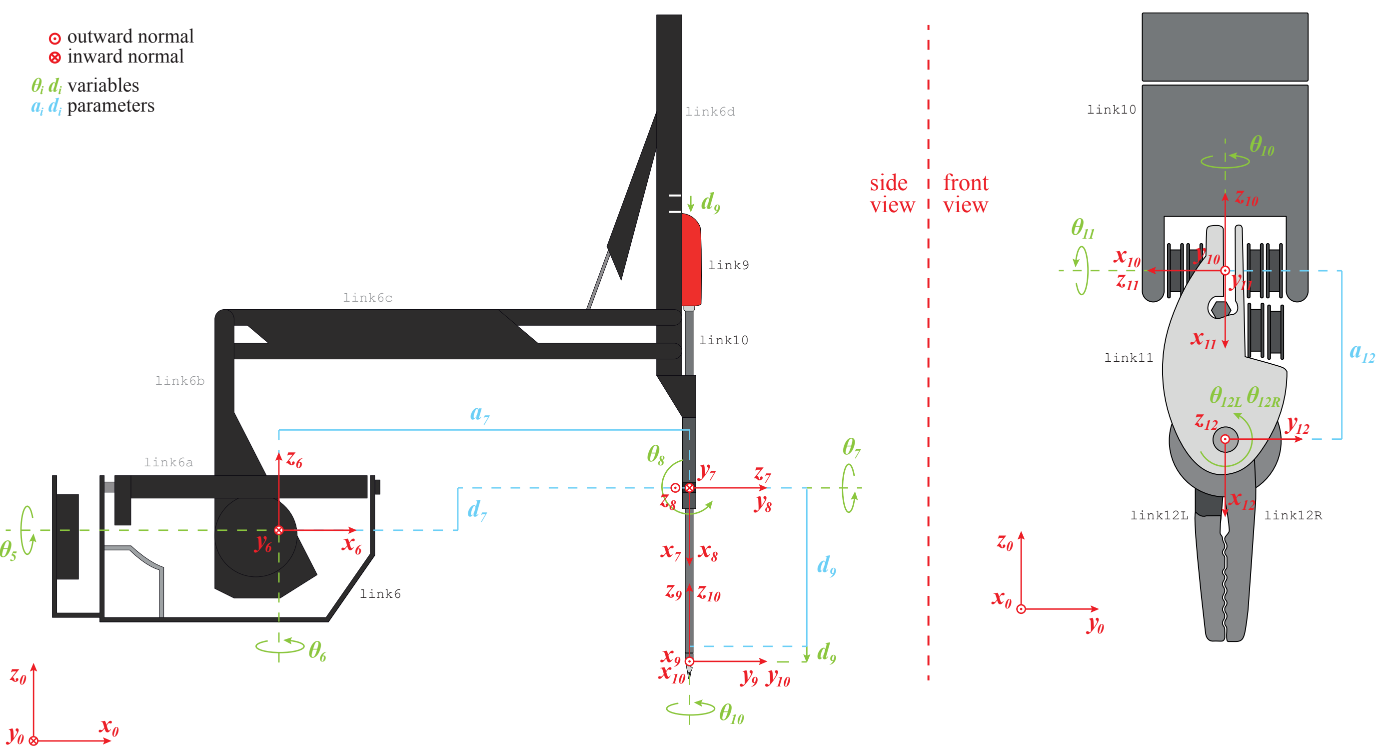
The values are measured on the da Vinci robot in the AAU control and surgery laboratory. Note that the new set of frames does not adhere to the DH standard completely, but they follow it as close as possible. The reason that the DH convention is not implemented fully is the underlying structure in the way the URDF file is read (in equation (6.16) the free rotation/translation is post-multiplied the fixed rotation/translation, while in equation (6.15) the free rotation/translation is pre-multiplied). This is not straightforward to correct. The modified frames are tested and verified in section C.3.

At this point, all variables from equation (6.16) are outlined in table 6.3b and implemented in the corresponding `xacro` files in the `davinci_description` package, i.e.:

- `robot/remote_center_manipulator.xacro` containing all joints associated with `p4_hand`
- `robot/instruments/needle_driver.xacro` containing all joints associated with `p4_instrument`



(a) Original robot hand kinematics including mimick joints, parameters given in table 6.3a.



(b) New robot hand kinematics excluding mimick joints, parameters given in table 6.3b.

Figure 6.7: Orientation and position of coordinate frames 7,8, etc., corresponding to the controllable joints of the robotic patient manipulator. For convenience of placing the hand roll and pitch frames in the pivot point (the stagnant fixed point) in the new kinematic description in figure 6.7b, a virtual link is inserted in the `xacro` file after each of these two joints.

In conclusion a three dimensional position of the tool tip is described in a frame oriented as the inertial frame and offset such that a robot configuration with all free angles and slide set to zero equals a position of the tool tip in [0,0,0].

For more details on the original and new kinematic descriptions implemented in ROS; including all measurements of parameters, gearing ratios, code for testing the kinematic description in MATLAB, and measurements of distances for different configurations; please refer to section C.1 and C.3 in appendix C.

Employing Inverse Kinematics for a Controller in 3D Cartesian Space

The controller relies on the use of inverse kinematics for the (ambiguous) mapping from 3D Cartesian space to 6D joint space. The inverse of the kinematic transformation matrix presented in equation (6.13) is [Murray et al., 1994]:

$$ {}^{j}_{i}\mathbf{T}^{-1} = \begin{bmatrix} {}^{j}_{i}\mathbf{R}^{T} & -{}^{j}_{i}\mathbf{R}^{T}\,{}^{j}_{i}\mathbf{p} \\ 0 & 1 \end{bmatrix} \tag{6.17} $$

With the sequence of transformations from frame k to frame i being represented by the transformation matrix ${}^{k}_{i}\mathbf{T}$, the inverse transformation, from frame i to frame k, is then its matrix inverse ${}^{k}_{i}\mathbf{T}^{-1}$ [Murray et al., 1994]:

$$ {}^{k}_{i}\mathbf{T} = {}^{k}_{j}\mathbf{T}\,{}^{j}_{i}\mathbf{T} = \begin{bmatrix} {}^{k}_{j}\mathbf{R}\,{}^{j}_{i}\mathbf{R} & {}^{k}_{j}\mathbf{p} + {}^{k}_{j}\mathbf{R}\,{}^{j}_{i}\mathbf{p} \\ 0 & 1 \end{bmatrix} \quad \Leftrightarrow \quad {}^{k}_{i}\mathbf{T}^{-1} = \begin{bmatrix} {}^{j}_{i}\mathbf{R}^{T}\,{}^{k}_{j}\mathbf{R}^{T} & -{}^{j}_{i}\mathbf{R}^{T}\,{}^{k}_{j}\mathbf{R}^{T}\,{}^{k}_{j}\mathbf{p} - {}^{j}_{i}\mathbf{R}^{T}\,{}^{j}_{i}\mathbf{p} \\ 0 & 1 \end{bmatrix} \tag{6.18} $$

A mapping from six to three degrees of freedom is a surjective map, i.e. several elements in the 6D domain may map to the same element in the 3D co-domain. However the inverse map from 3D to 6D is neither injective nor surjective, as each element in the 3D domain can map to several elements in the 6D co-domain, and hence the mapping requires a decision of the "best" map.

In practice this mapping from the desired 3D Cartesian space configuration to a prudent 6D joint space position of the da Vinci robotic patient manipulator is handled by the KDL inverse kinematics solver. Then the transformed joint position commands are published on the appropriate ROS topic and thereby passed to the low level controllers (see figure 1.5).

The IK KDL solver employed in ROS utilizes the kinematic chain from the URDF, which is generated from the `xacro` link and joint kinematic description as presented in section C.3. From this chain a FK position solver is created along with an IK velocity solver in order to define the IK position solver.

The KDL IK position solver utilizes the Newton-Raphson (NR) iterative numerical technique through the function `CartToJnt` to determine a prudent joint configuration implementing the desired Cartesian configuration. Note that the NR requires both position and velocity (derivative of position) to find a solution.

The actual implementation of the KDL solver follows the recommended implementation method as described by The Orocos Project main web page [Orocos, 2015]. Thus, the algorithm deployed for the KDL solver follows the steps described below:

Utilize KDL for Forward Kinematics

- Make a KDL tree (several chains) from URDF.
- Fetch all KDL segments, i.e. links
- Create chain from desired links: `p4_rcm_base` to `needle_driver_jawbone_right`.
- Create FK kinematic solver based on that chain
- Get number of joints and create a joint array
- Ask for joint values
- Compute rotation/translation matrix $\mathbf{T}$

Utilize KDL for Inverse Kinematics

- First three points are identical to FK solver
- Instantiate position solver $\mathbf{x}$ for Newton-Raphson (NR) based on that chain
- Instantiate velocity solver $\dot{\mathbf{x}}$ for NR
- Specify NR resolution and no. of iterations
- Ask for desired (x, y, z) destination
- Call solver to compute necessary 6D angles
- Deliver result in a vector `q`

The KDL solver is, however, far from perfect. At present time, the solver has a tendency to choose joint angles requiring multiple revolutions on the unit circle to obtain the same position as, obviously, less than one revolution could have done. This is caused by the lack of joint limit specification. The problem and the solution is outlined in figure 6.8 where `q(i)` is the ith controllable joint, e.g. `p4_instrument_roll`.

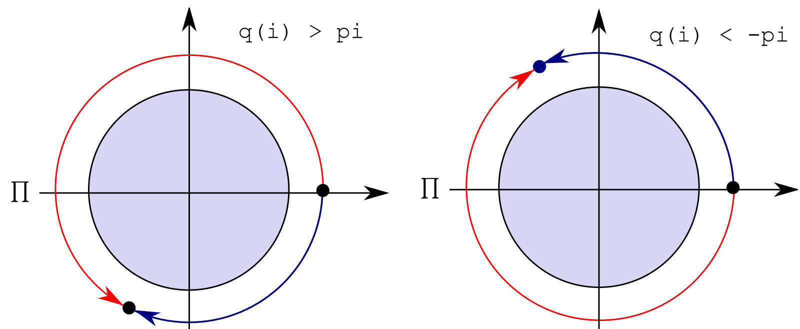


Figure 6.8: Adjusting the KDL sultion for `q(i) > pi` and `q(i) < -pi` from red to blue direction of rotation.

The solution found in figure 6.8 can be implemented as:

$$\text{while}(|q(i)| > \pi): \quad q(i) = \begin{cases} -(\pi - (q(i) - \pi)) & \text{if} \quad q(i) > \pi \\ (\pi + (q(i) + \pi)) & \text{if} \quad q(i) < -\pi \\ q(i) & \text{otherwise} \end{cases}$$

This solution does, however, introduce limitations. The trajectory will inevitably be corrupted with flicker and other unsightly behaviours when the solver wishes to spin a joint multiple times around the unit circle. Additionally, not all joints can reach all the way to $\pm\pi$ which can cause imprecise settling stages.

With the FK and IK solver ready, the algorithm connecting the subtasks can be presented.

6.5.2 Main Algorithm

The main algorithm is outlined in figure 6.9.

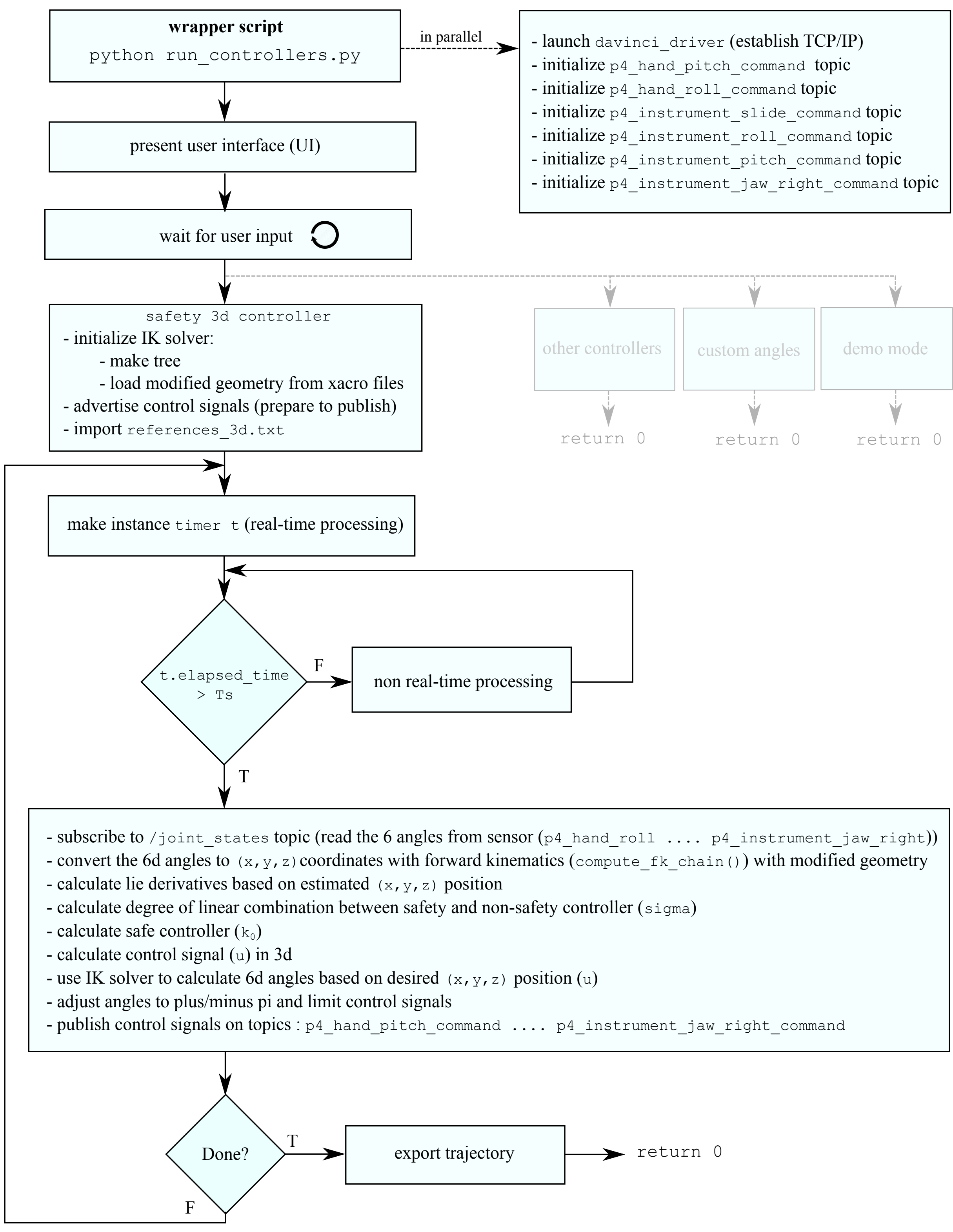


Figure 6.9: Main algorithm for a safe controller in the 3 dimensional euclidean space.

The algorithm presented in figure 6.9 comprises the entire code structure which will not be elaborated here, though as written in the introduction, it can be found in appendix H. The real-time part runs every 0.01 s (100 Hz) which corresponds to the publish rate which also is set to 100 Hz. With all subtasks implemented appropriately, it is time to analyse the results.

6.5.3 Results

The results presented intends to demonstrate the below listed features:

- That the controller can find setpoints on $\mathcal{Y} = \mathcal{X}_0 \setminus \mathcal{T}$.
- That the controller is stable on $\mathcal{Y}$.
- That the controller will redirect the trajectory to a safe area if the direct path to the setpoint passes through $\mathcal{X}_u$ or $\mathcal{T}$.
- That the controller will ensure that the trajectory is redirected to a safe area if a setpoint is given on the unsafe set $\mathcal{X}_u$.
- That $\sigma(\mathbf{x})$ increases its value as the trajectory reaches $\mathcal{X}_u$ starting from $\mathcal{X}_0$ through $\mathcal{T}$ (from 0 to 1).

The execution time of the controller is validated first, and a measurement is plotted in figure 6.10. From this it is seen that the real time part is completed well within the 10 ms (100 Hz sample rate). As expected, the execution time is considerably higher than for the 1D system (compare to figure 4.15) because of the necessity of an inverse kinematics solver.

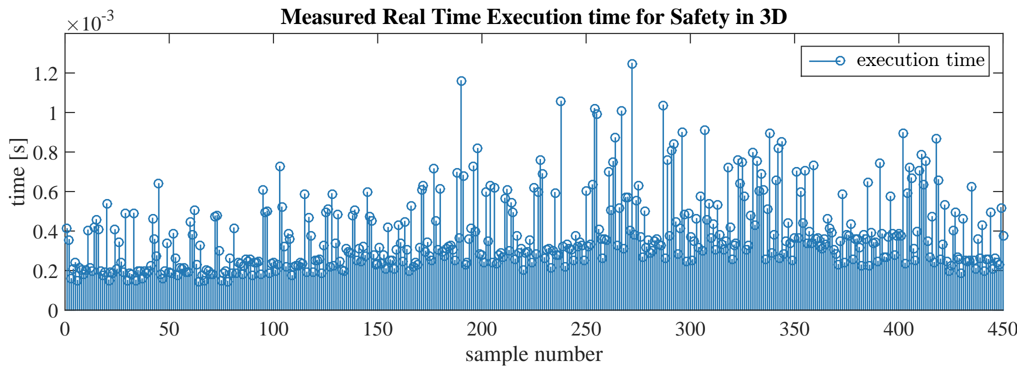


Figure 6.10: Execution time for 3D system. It is seen that the controller never exceeds a computation time of 1.2 ms. This plot can be reconstructed by running the MATLAB script `measurements/safety_3d/execution_time/plot_exe_3d.m` in appendix J.

The features listed above can be verified in the trajectory plot shown in figure 6.11. Plot details and measurements can be reconstructed by running the MATLAB script `plot_3d_meas.m` found in appendix J in the folder `measurements/safety_3d`. The first 2-3 setpoints given in figure 6.11 are within the set $\mathcal{Y}$ and thereby safe. It is seen how the desired setpoints initially are reached (with a slight detour caused by the KDL solver). It is seen that when a setpoint is given which requires a path through $\mathcal{X}_u$, the safety controller gradually takes over and redirects the trajectory around $\mathcal{X}_u$ and finally allows it to find its reference value.

It is also seen that $\sigma(\mathbf{x})$ increases along with a shorter distance to $\mathcal{X}_u$, as indicated with green when $\sigma(\mathbf{x}) \in (0, 0.25]$ and magenta when $\sigma(\mathbf{x}) \in (0.25, 1]$ in figure 6.11. It can be noted that not all setpoints are reached completely, which is due to imprecise kinematics, joint values computed by the IK solver that are outside the implementable range and last but not least, the lack of integral action in the controller. Finally, when a setpoint is given on $\mathcal{X}_u$, it is seen how the safety controller quite aggressively forces the trajectory away from $\mathcal{X}_u$. All these features are indeed the expected outcome of the controller, thus verifying the implementation.

The result depicted in figure 6.11 and its associated analysis concludes the implementation.

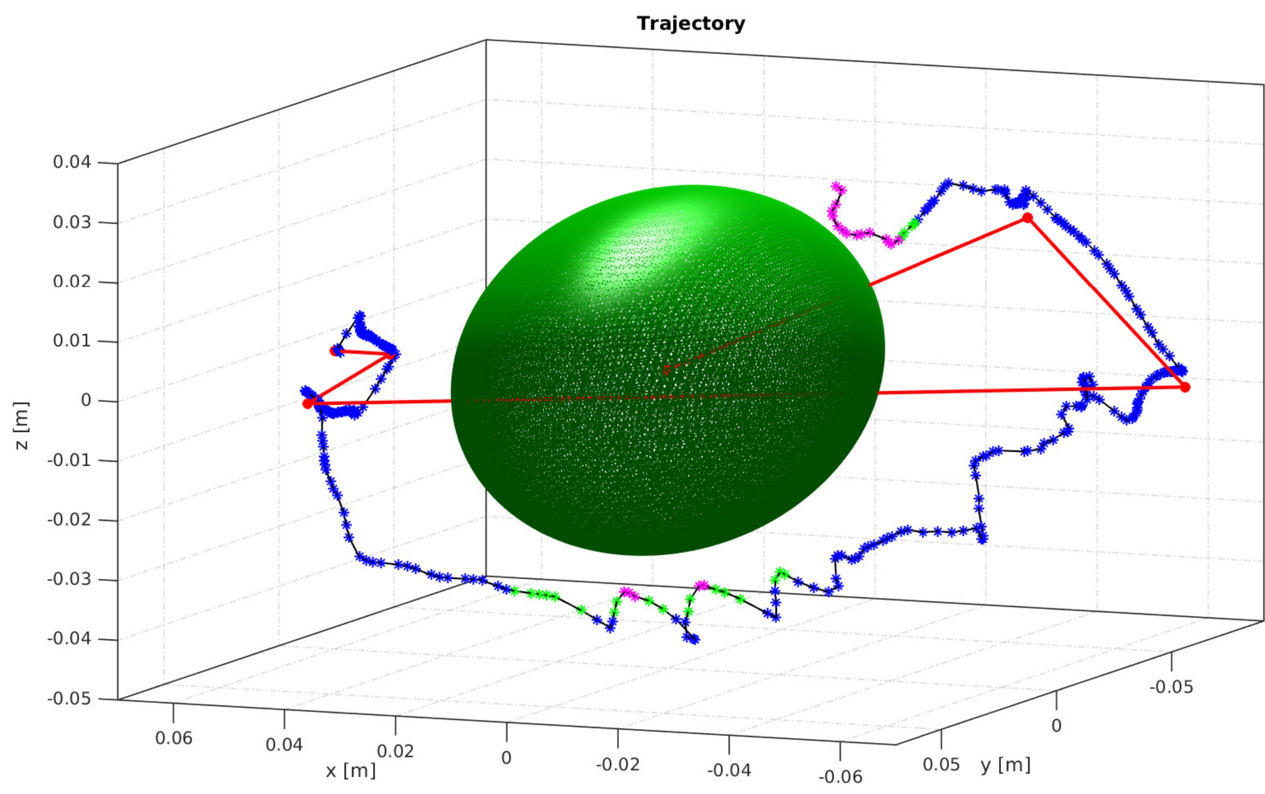


Figure 6.11: Results from safety controller in the 3D Euclidean space. The interior of the green ellipsoid comprises $\mathcal{X}_u$ and around it is a 1 cm thick rim which constitutes $\mathcal{T}$ (not plotted). The trajectory is plotted with a blue color, and it is overlay with a green color if $\sigma(\mathbf{x}) \in (0, 0.25]$ and a magenta color if $\sigma(\mathbf{x}) \in (0.25, 1]$. The red lines demonstrate the direct path between two desired setpoints (red circles). The trajectory is initiated in $\mathbf{x} = (0.06, 0.08, -0.01)$ (safe) and has a desired final destination in $\mathbf{x} = (0, 0, 0)$ (unsafe). This plot can be reconstructed by running the MATLAB script `measurements/safety_3d/plot_3d_meas.m` in appendix J.

6.6 Conclusion for Safety in the 3D Euclidean Space

A big step has been taken towards ensuring safety in the three dimensional Euclidean space, i.e. a three dimensional control barrier function which is defined to fulfil the demands given in Definition 3.1 and a developed controller which proves itself very efficient in the MATLAB simulation environment, i.e. with ideal sampling rate and under no influence of an IK solver and other stochastic uncertainties. The controller fulfils all the initial outlined requirements.

The implemented controller on the da Vinci robot does indeed show the full potential of using barrier certificates in surgical robotics and proves that safety control can be implemented on a robot moving in 3D Cartesian space. It does, however, suffer from a poor IK solver, imprecise kinematics and the lack of integral action. Also, the model derived for the three dimensional space is very simplified, though sufficient to establish a "proof of concept" framework.

An additional huge simplification is the consistent disregard of the orientation of the end effector in the implementation, thus merely guaranteeing safety for the position of the end effector, not the full extent of the physical volume of the robotic tool.

Due to present limitations in the TCP/IP connection to the robot the implementation is made at a 100 Hz sampling rate, which causes erratic behaviour. Based on the results in chapter 4 and chapter 5 it is expected that an increase of the sampling rate to 2 kHz will smooth the behaviour of the trajectory on the verge of the transition region, where the safety controller starts to take effect. It is anticipated that the trajectory will settle at a near constant distance to the unsafe set on its path to setpoints "on the other side" of this region, similar to what was seen in the MATLAB simulation in section 6.4.

These topics are important to investigate and implement in future work.

Chapter 7

Interim Conclusion

An introduction to the concept of automated surgery with robotic manipulators is given in chapter 1. This founds the need for a way to guarantee safety in such operations. Thus, the initial task posed in subsection 1.4.1 concerns two approaches to the problem of ensuring safety for the da Vinci robotic manipulator, i.e.:

1. The design of a safe controller ensuring safety in real-time.
2. The analysis of a controller, posing the question if it is safe.

The first bullet point is at this point investigated. The theory presented in [Wieland and Allgöwer, 2007] is adopted and described in chapter 3 which ensures that the barrier certificate requirements outlined in chapter 2 are obeyed, thereby allowing the development of safe controllers.

The theory is applied to specific use cases. First, a palpable example is conducted in chapter 4 which ought to give experience with control barrier functions (CBFs) and the way the theory is applied. The outcome is not only a fully functional safe controller in one dimension, but also a valuable experience in the application of the theory. As expected, when the system order increases, the difficulty in constructing a valid CBF, is also increased. Though, with a system order $n = 2$, it is indeed still possible. Primarily because the states can be translated into physically meaningful quantities such as position and velocity. However, it is easy to imagine that as n increases and the physical interpretation of the states obscures to abstract states, this approach will be nearly impossible.

The next step is taken in chapter 5 where the problem consists of ensuring safety for a beating heart, such that a virtual fixture can be ensured in a safe manner. The problem here differs from chapter 4 because the CBF is dynamic. Though, again, a successful implementation is performed and a valid CBF can be found. The dimension of the system is kept low which simplifies the task of finding a valid CBF. The lack of integral action is obvious in this chapter and with a more advanced/realistic model of the heart, the search for a valid CBF will be a highly non-trivial task. If not impossible.

The dimension of the considered system is yet again increased in chapter 6. A safe controller in the 3D Euclidean space is developed with an associated valid CBF alongside. It is from here seen that the creativity and complexity increases yet a step. With a simplified model of the robotic manipulator, a successful analysis and implementation is performed. The result is as expected and indeed quite convincing, but it is also clear that to reach the end goal of a realistic model of the heart or other vital organs and of the robotic manipulator, the system order must be increased. Again, this implies serious challenges in the construction of a valid CBF.

Accordingly, it is desirable to find a different approach to defining barrier certificates that is more efficient for higher order systems. Indeed, an efficient and straightforward approach may be difficult to derive, but the success criteria for constructing CBFs for higher order systems is not to find a simple method, but to find a method at all. Thus the success criteria can more appropriately be defined as: *If it is possible, it is better.* This is where the second bullet point becomes active.

The MATLAB toolbox SOSTOOLS can be used to perform a "controller analysis". This toolbox uses Sums of Squares optimization to solve problems, so it is necessary to cast the definition of the barrier certificate as an SOS problem in order to perform a barrier certificate search with the toolbox. Hence the background to the SOS formulation of the problem is presented in the upcoming chapter, after which the SOSTOOLS toolbox is introduced, and used for barrier certificate search.

This approach intends to solve the second bullet point, i.e. to find a way to analyse if a control system is safe, thereby giving it a "safe" or "not safe" verdict.

Chapter 8

Safety Verification with Barrier Certificates

The construction of a valid barrier certificate can be a non-trivial task, as it was seen in the construction of a CBF for the second order system in subsection 4.2.2. And with increasing order of the system for which safety need to be guaranteed the difficulty rapidly increases. Hence it is desired to use a more methodical approach to construct barrier certificates, and for this purpose the MATLAB toolbox SOSTOOLS can be used [Prajna et al., 2007] (for acquisition, see appendix E). This toolbox requires that the problem is cast as a sum of squares (SOS) program, which is why this chapter is dedicated to give an introduction to the concept of SOS and how the barrier certificate definition can be recast as an SOS problem.

Now, instead of manually constructing a CBF for the open-loop system in order to design a safety controller from it, a barrier certificate for a closed-loop system is sought with SOSTOOLS. When a barrier certificate can be found for a closed-loop system $f_{cl}(\mathbf{x})$ according to Definition 2.2 this signifies a verification that the system, and hence the controller, is safe according to the outlined problem. SOSTOOLS can be used to search for a polynomial barrier certificate given that: [Prajna et al., 2007]

- The vector field of the closed-loop system is polynomial.
- The sets $\mathcal{X}$, $\mathcal{X}_0$ and $\mathcal{X}_u$ can be described as intervals in each of the n dimensions, which can be defined through positivity of polynomials on each interval.

Furthermore the recasting of the barrier certificate definition as an SOS problem requires the use of SOS polynomials. An SOS polynomial $p(\mathbf{x})$ is denoted by $p \in \Sigma[\mathbf{x}]$ signifying that p is a polynomial in the variable $\mathbf{x}$ with coefficients in the set of SOS variables: Σ. A polynomial $p(\mathbf{x})$ is SOS if there exist polynomials $f_1, \dots, f_m$ such that [Parrilo, 2003]

$$p(\mathbf{x}) = \sum_{j=1}^{m} f_j^2(\mathbf{x}) \tag{8.1}$$

A short illustrating example is given initially with the intention to provide some understanding about SOS polynomials and the notation applied.

Example 8.1 (Sum of Squares Polynomial)
Consider the second order polynomial $f_1(x) = ax^2 + bx + c$. Comparing to the structure of an SOS polynomial $f_2(x) = (d+ex)^2 = d^2 + 2dex + e^2x^2$, it is seen that if the relationship:

$$a = e^2, \qquad b = 2de, \qquad c = d^2 \qquad \left(\text{or simply } b = 2\sqrt{ac}\right)$$

holds, then a, b, c are SOS variables, denoted by $a, b, c \in \Sigma$, because they are the coefficients in an SOS polynomial. It can additionally be stated that:

$$f_1(x) \in \Sigma[x]$$

where $\Sigma[x]$ denotes a set of polynomials in x with coefficients in Σ, i.e. $f_1(x)$ is SOS.

Declaring a polynomial in SOSTOOLS is done by defining a monomial vector for which the program solves for polynomial coefficients. Introducing the notion of a monomial vector as a vector $\mathbf{z}$ in $\mathbf{x}$ of degree deg; e.g. if $\mathbf{x} \in \mathbb{R}^2$ and $deg = [0:2]$ each entry has the form $x_1^a x_2^b$ with exponents $a+b = deg = 0,...,2$ i.e.

$$\mathbf{z} = [x_1^0x_2^0 \quad x_1^1x_2^0 \quad x_1^0x_2^1 \quad x_1^2x_2^0 \quad x_1^1x_2^1 \quad x_1^0x_2^2]^T = [1 \quad x_1 \quad x_2 \quad x_1^2 \quad x_1x_2 \quad x_2^2]^T \tag{8.2}$$

Now, according to [Parrilo, 2003] an SOS polynomial $p \in \Sigma[\mathbf{x}]$ can be formulated on a quadratic form comprising a coefficient matrix and a monomial vector

$$p = \mathbf{z}^T \mathbf{Q} \mathbf{z}, \qquad p \geq 0 \quad \forall\, \mathbf{x} \in \mathbb{R}^n \tag{8.3}$$

where

$\mathbf{z}$	is a monomial vector in $\mathbf{x} \in \mathbb{R}^n$
$\mathbf{Q}$	is a real positive semidefinite symmetric coefficient matrix

With monomials and SOS polynomials defined, a polynomial barrier certificate can now be constructed. An SOS description of a problem, however, is a global description of nonnegativity (positive or zero), and it is desired to set up local requirements for the barrier polynomial on each of the sets $\mathcal{X}$, $\mathcal{X}_0$ and $\mathcal{X}_u$.

A way to be able to define nonnegativity locally is to use Positivstellensätze. A Positivstellensatz is a structure theorem of a polynomial which is positive on some set, and gives an algebraic certificate that a solution exists for a system of real polynomial inequalities [d'Angelo and Putinar, 2009]. In Putinar's Positivstellensatz, presented in Theorem 8.2, a compact set $\mathbb{K} \subset \mathbb{R}^n$ (a strict subset of the state-space, i.e. not including the entire state-space) is defined by the nonnegativity of some polynomials g_j. Now the positivity of a polynomial h on the set $\mathbb{K}$ can be expressed in terms of a weighted sum of these polynomials g_j with SOS polynomials as coefficients $q_j \in \Sigma[\mathbf{x}]$ [Laurent, 2009,pp 184-186],[Lasserre, 2009,pp 28-29].

Theorem 8.2 (Putinar's Positivstellensatz)
Given the finite family of polynomials $(g_j)_{j=1}^m$ and the subset $Q(g)$ generated by the family $(g_j)_{j=1}^m$ [Lasserre, 2009,p 29]

$$\text{polynomials} \qquad (g_j)_{j=1}^m \in \mathbb{R}[\mathbf{x}] \tag{8.4)a}$$

$$\text{set} \qquad Q(g) = Q(g_1,...,g_m) \equiv \left\{ q_0 + \sum_{j=1}^m q_j g_j \;\middle|\; (q_j)_{j=0}^m \in \Sigma[x] \right\} \tag{8.4)b}$$

Assume that there exists a function $u(\mathbf{x}) \in Q(g)$ such that the level set $\{\mathbf{x} \in \mathbb{R}^n \mid u(x) \geq 0\}$ is compact [Lasserre, 2009,p 29]. Given a polynomial h and the compact basic semialgebraic set $\mathbb{K}$ defined by the nonnegativity of the polynomials $g_1, \ldots, g_m$

$$\text{polynomial} \qquad h \in \mathbb{R}[\mathbf{x}] \tag{8.5)a}$$

$$\text{set} \qquad \mathbb{K} \equiv \left\{ \mathbf{x} \in \mathbb{R}^n \mid (g_j)_{j=1}^m \geq 0 \right\} \tag{8.5)b}$$

If the polynomial h is strictly positive on the set $\mathbb{K}$, then $h \in Q(g)$, which means that h can be formulated as

$$h = q_0 + \sum_{j=1}^m q_j g_j \tag{8.6}$$

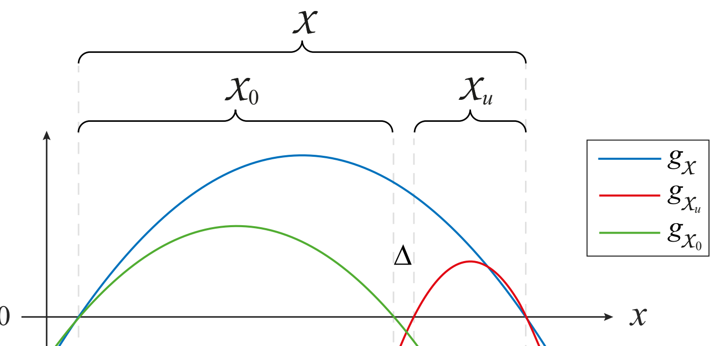


Figure 8.1: Example of g-polynomials defining each of the sets $\mathcal{X}$, $\mathcal{X}_u$ and $\mathcal{X}_0$ by their nonnegativity, with a distance Δ between the safe and unsafe sets.

In equation (8.5)b the region of nonnegativity of the polynomial(s) $(g_j)_{j=1}^m$ define the extent of the set $\mathbb{K}$, the gs being polynomials in the state variables x as defined by equation (8.4)a, e.g. in 1D Cartesian space $g(\mathbf{x})$ may be a parabola which is positive-valued on the interval $\mathbf{x} \in [a,b]$, hence defining the semialgebraic set $\mathbb{K} = \{\mathbf{x} \in [a,b]\}$. In this way appropriate polynomials g_j can be constructed to define each of the sets $\mathcal{X}$, $\mathcal{X}_0$ and $\mathcal{X}_u$ as given by Definition 2.2. An example of a polynomial g_1 for each of the sets is sketched in figure 8.1.

The variables $(q_j)_{j=1}^m$ in equation (8.4)b are SOS and thereby nonnegative per definition. Hence from the definition of the gs and qs it can be seen from equation (8.6) that the polynomial h is positive on the set $\mathbb{K}$. Outside the set $\mathbb{K}$ one or more g_js are negative, and hence the sign of h cannot be determined outside $\mathbb{K}$. Rearranging equation (8.6) to

$$\underbrace{q_0(\mathbf{x})}_{\geq 0\,\forall\,\mathbf{x}} = h(\mathbf{x}) - \sum_{j=1}^{m} \underbrace{q_j(\mathbf{x})}_{\geq 0\,\forall\,\mathbf{x}} \underbrace{g_j(\mathbf{x})}_{\geq 0\,\forall\,\mathbf{x}\in\mathbb{K}} \qquad \in \Sigma[\mathbf{x}] \tag{8.7}$$

however, the right-hand expression will always be nonnegative due to the SOS equality. Using SOS-TOOLS it is possible to solve for the unknown h setting up a number of SOS inequalities corresponding to defining the right-hand side of equation (8.7) as being nonnegative.

8.1 Recasting the Barrier Certificate Definition

Now equation (8.7) can be seen as a template for the reformulation of the requirements for a barrier certificate in Definition 2.2. Setting up each of the requirements for the barrier certificate $B(\mathbf{x})$ on each of the semialgebraic sets $\mathcal{X}$, $\mathcal{X}_u$ and $\mathcal{X}_0$, is a matter of defining one or more polynomials $g_j(\mathbf{x})$ for each set such that the intersection of the nonnegative regions of the polynomials outline the set. E.g. in order to define the region $\mathcal{X}_u \subset \mathcal{X} \subset \mathbb{R}$ construct a polynomial $g(\mathbf{x})$ such that it is positive within the unsafe interval and its zero level set constitutes the desired border of the region. An example of a polynomial $g(\mathbf{x})$ defining the region $\mathcal{X}_u$ is seen in figure 8.1 as the red curve. If several polynomials $g_j(\mathbf{x})$ are used to define $\mathcal{X}_u$, the set is defined by the nonnegative intersection region, i.e. the subset of the state-space where all of the $g_j(\mathbf{x})$s are nonnegative.

When the polynomials $g_j(\mathbf{x})$ have been defined for each of the sets, the polynomial $h(\mathbf{x})$ in equation (8.7) is substituted according to Definition 2.2, such that $h(\mathbf{x}) \geq 0$ on the relevant set. That is, when defining $\mathcal{X}$ according to equation (2.5)c, the polynomial $h(\mathbf{x})$ can be written as $-L_{f_{cl}}B(\mathbf{x})$ and when defining $\mathcal{X}_0$ use $h(\mathbf{x}) = -B(\mathbf{x})$ according to equation (2.5)a. However, when defining $\mathcal{X}_u$ according to equation (2.5)b,

the positivity constraint on $B(\mathbf{x})$ has to be transformed into a nonnegativity constraint. This can be done by introducing a small scalar value $\bar{\varepsilon} > 0$, such that $B(\mathbf{x}) \geq \bar{\varepsilon}$ or identically $B(\mathbf{x}) - \bar{\varepsilon} \geq 0$ which can now be substituted for $h(\mathbf{x})$.

Definition 8.3 (Barrier Certificate Recast to SOS Formulation)
In summary, referring to the requirements for a barrier certificate in Definition 2.2 and the SOS formulation of the polynomial h in equation (8.7) based on Putinar's Positivstellensatz, the inequalities defining the barrier certificate $B(\mathbf{x})$ can be set up as

$$-B(\mathbf{x}) \geq 0 \quad \forall\ \mathbf{x} \in \mathcal{X}_0 \quad \Leftarrow \quad -B(\mathbf{x}) - \sum_{j=1}^{m} q_j g_j \in \Sigma[\mathbf{x}] \tag{8.8)a}$$

$$B(\mathbf{x}) - \bar{\varepsilon} \geq 0 \quad \forall\ \mathbf{x} \in \mathcal{X}_u \quad \Leftarrow \quad B(\mathbf{x}) - \bar{\varepsilon} - \sum_{j=1}^{m} q_j g_j \in \Sigma[\mathbf{x}] \tag{8.8)b}$$

$$-L_{f_{cl}} B(\mathbf{x}) \geq 0 \quad \forall\ \mathbf{x} \in \mathcal{X} \quad \Leftarrow \quad -L_{f_{cl}} B(\mathbf{x}) - \sum_{j=1}^{m} q_j g_j \in \Sigma[\mathbf{x}] \tag{8.8)c}$$

With this formulation the Lie derivative is required to be nonpositive on set $\mathcal{X}$ and the barrier certificate is required to be nonpositive on the safe set $\mathcal{X}_0$ and greater than or equal to the constant $\bar{\varepsilon}$ on the unsafe set $\mathcal{X}_u$. When defining the sets, the safe and unsafe regions must be separated by some distance Δ. This is illustrated in figure 8.2.

The task of searching for a barrier certificate validating system safety is now a matter of the fairly easy construction of the polynomials $g_j(\mathbf{x})$ to define the desired outline of each of the sets $\mathcal{X}$, $\mathcal{X}_u$ and $\mathcal{X}_0$, as in the example in figure 8.1, in SOSTOOLS. However, it is also a matter of deciding the polynomial degree (through monomials) of $B(\mathbf{x})$ and of each of the SOS polynomials $q_j(\mathbf{x})$. As small a degree "as possible" is desired for all monomials, but it may be necessary to increase the degree iteratively. This introduces numerical errors, which may require that the size of $\bar{\varepsilon}$ must be increased. When no solution can be found, it may be necessary to increase the distance Δ separating the safe and unsafe regions, i.e. contract the safe region by altering the polynomials $g_j(\mathbf{x})$ outlining the region.

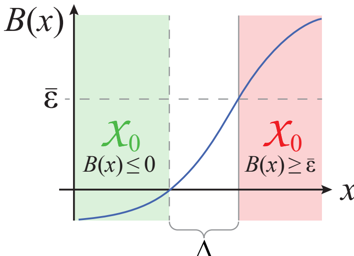


Figure 8.2: The value of $B(x)$ on the unsafe set is at least the small positive value $\bar{\varepsilon}$, while the value on the safe set is nonpositive. This requires that the two sets are separated by a small distance Δ.

In the following chapter the syntax for SOSTOOLS is introduced and barrier certificates are sought to validate safety of systems, based on linear closed-loop systems representing the da Vinci robot from the preceding chapters.

Chapter 9

Barrier Certificate Search with SOSTOOLS

As presented in chapter 8 a polynomial barrier certificate can be constructed using SOS optimization by employing the MATLAB toolbox SOSTOOLS. This toolbox is a convex relaxation framework based on sum of squares decompositions of multivariate polynomials and semidefinite programming solvers [Prajna et al., 2007] (for acquisition, see appendix E).

In this chapter barrier certificates are found with SOSTOOLS by using Putinar's Positivstellensatz, presented in Theorem 8.2, to set the constraints for the barrier certificate on each of the sets $\mathcal{X}$, $\mathcal{X}_u$ and $\mathcal{X}_0$. First, section 9.1 introduces the basic functions in SOSTOOLS needed to formulate the requirements for the barrier certificate, and a step-by-step guide to define the program is presented, concluding with an overview of how to evaluate the validity of a solution. In section 9.2 the one-dimensional system comprising the da Vinci robot slide movement from chapter 4 is analysed, initiated by an exhaustive example of the full formulation of a barrier certificate search for the first order system model, and finally, in section 9.3, the robot slide movement modelled as a second order system is safety validated.

9.1 Formulation of a Barrier Certificate in SOSTOOLS Syntax

An SOS program is the framework in which the barrier certificate is defined by setting up the SOS requirements from Definition 8.3, and searching for the barrier certificate corresponds to solving the SOS program. This section gives a short introduction to the SOSTOOLS formulation of the parameters and variables necessary to set up the requirements for the barrier certificate, based on the SOSTOOLS user guide [Papachristodoulou et al., 2013]. An overview of necessary SOS functions is given in table 9.1.

Syntax	Explanation
`pvar x1;` `prog = sosprogram(x1);`	Initialization of an SOS program `prog` in the state variable `x1`, which is declared as type `pvar` (symbolic variable)
`Z = monomials(x1,deg);` `[prog,q] = sossosvar(prog,Z);`	Parametrize an SOS polynomial `q` in the SOS program `prog`. The degree of the SOS polynomial is defined by the monomial vector `Z` of degree `deg` (i.e. deg(`q`) $=$ 2`deg`)
`Z = monomials(x1,deg);` `[prog,B] = sospolyvar(prog,Z);`	Parametrize a polynomial `B` in the SOS program `prog`. The degree of the polynomial is defined by the monomial vector `Z` of degree `deg` (i.e. deg(`B`) $=$ `deg`)
`prog = sosineq(prog,-B-q·g);`	Declare the inequality constraint, e.g. `-B-q·g` (or more exact: `-B-q·g` $\in \Sigma[x_1]$) in the SOS program `prog`
`prog = sossolve(prog);`	Solve the SOS program `prog` i.e. find coefficients for all polynomials conforming with all constraints
`getB = sosgetsol(prog,B)`	After solving, get the solution (with coefficients) for the polynomial `B`

Table 9.1: SOSTOOLS functions necessary to search for a barrier function as given by Definition 8.3.

An SOS program is initialized with the command `sosprogram`, and polynomials and SOS polynomials can be declared in the program as functions of the variables that are input to the program (see table 9.1) with `sospolyvar` and `sossosvar`, respectively. When the necessary SOS variables and polynomials are defined, the inequalities in Definition 8.3 can be defined with the function `sosineq`, and when all constraints are set up, the program is (attempted to be) solved by calling `sossolve`. This will return an overview of the precision of the solution (if any was found) as a residual error norm, number of iterations and time elapsed for solving the problem. To get the solution (coefficients) found for any of the SOS variables or polynomials, call the function `sosgetsol` (more about evaluating the solution is found in subsection 9.1.2).

9.1.1 Step-by-step Guide to Search for a Polynomial Barrier Certificate in SOSTOOLS

Searching for a polynomial barrier certificate in SOSTOOLS requires the definition of all of the variables and polynomials given by Definition 8.3 as follows:

- **Initialize the Program**
  First declare the state space variables $\mathbf{x} \in \mathbb{R}^n$ as `pvar`, and initialize the SOS program with the system states by the function `sosprogram`.
- **Define the Vector Field**
  The open-loop state space system $f_{ol}(\mathbf{x})$ is defined, and a controller is found according to pole placement or another preferred method. Then write the closed-loop system equation $f_{cl}(\mathbf{x})$ in terms of the symbolic state vector.
- **Set up the Constraints for the Polynomial Barrier Certificate**
  Declare a monomial vector $\mathbf{z}_B$ in $\mathbf{x}$ (or part of $\mathbf{x}$) of sufficiently large degree, and parametrize the polynomial $B(\mathbf{x})$ as a function of $\mathbf{z}_B$ with `sospolyvar`. The problem of finding the coefficients for the barrier certificate is now for each region $\mathcal{X}$, $\mathcal{X}_u$ and $\mathcal{X}_0$ a matter of defining the following:
  - **Define the Polynomials $g_j(\mathbf{x})$**
    Define one or more polynomials g_j that are positive in the region to be defined, e.g. $\mathcal{X}$, and negative outside.
  - **Declare the SOS Variables $q_j(\mathbf{x})$**
    Declare monomial vectors $\mathbf{z}_{q_j}$ in x of appropriate degree (preferably as small as possible), and parametrize the SOS polynomials (multipliers) q_j with `sossosvar`.
  - **Set up the Inequality**
    Cf. the nonnegativity of an SOS polynomial (q_0), each `sosineq` can be formulated as given by Definition 8.3. For equation (8.8)b choose a small positive number $\bar{\varepsilon}$. The inequality pertaining to a set may be defined in terms of several g_js; if the set is defined by
    - $g_1 \bigcap g_2 \bigcap \ldots \bigcap g_m$, then write $h - \sum q_j g_j \geq 0$
    - $g_1 \bigcup g_2 \bigcup \ldots \bigcup g_m$, then write $h - q_1 g_1 \geq 0$, $h - q_2 g_2 \geq 0$ etc.

    Note that each expression in the inequalities in Definition 8.3 must have even degrees in the leading and trailing terms in order for the expressions to be SOS.
- **Solve the SOS Program**
  With all inequalities defined in the program, SOSTOOLS is now ready to solve for the barrier certificate with `sossolve`, if any certificate exists for the given system $f_{cl}(\mathbf{x})$. If no solution is found, adjusting the degree of the SOS variables q_j or the polynomial $B(\mathbf{x})$ may yield a solution.

Also, adjusting Δ and ε may increase the probability of finding a solution. Otherwise it can be concluded that safety cannot be guaranteed of the system under scrutiny and the set $\mathcal{X}_0$ may be forced smaller.

9.1.2 Evaluating the SOSTOOLS Solution

If no valid solution is found, the tested closed-loop system $f_{cl}(\mathbf{x})$ cannot be guaranteed to be safe. It may, however, still be possible to find a barrier certificate and validate safety using a different degree of the polynomial $B(\mathbf{x})$ or SOS polynomials $q_j(\mathbf{x})$. Otherwise the safe region $\mathcal{X}_0$ may have to be smaller.

When the SOS program is solved, the list of information printed out in the MATLAB terminal includes a number of useful parameters for evaluating the validity of the solution, summarized in table 9.2. If the problem is either primal or dual infeasible, indicated by `pinf` or `dinf` being 1, respectively, obviously no solution could be found. The feasibility ratio is an indicator of the feasibility as well, and converges to 1 for feasible solutions and to -1 for strongly infeasible solutions [Aylward et al., 2008]. A value in between is an indicator of numerical problems, which will also be written in the overview. The `residual norm` is the norm of the numerical error in the solution [Papachristodoulou et al., 2013], and when numerical problems cause this error to exceed a tolerance set in the SOSTOOLS solver, this is indicated as a warning of numerical errors with `numerr = 1`, while a numerical error of 2 indicates failure of the solver.

Parameter	**Explanation**
`pinf = 0`	Primal infeasibility of the problem is indicated with `pinf = 1`
`dinf = 0`	Dual infeasibility of the problem is indicated with `dinf = 1`
`feasratio = 1`	Feasibility ratio converges to 1 for feasible solutions and to -1 for strongly infeasible solutions, while values in between indicate numerical problems in the solution
`numerr = 0`	Numerical error warning is indicated with `numerr = 1` if the residual norm exceeds a tolerance value (default to 1e-9, see `sossolve.m` line 61), and complete failure of the solution is indicated with `numerr = 2`
`Residual norm`	Norm of the numerical error in the solution
`[Q,Z,f] = findsos(-B-g*q);`	Test that the solution found complies with the requirement that the inequality is in fact SOS by checking that it can be resolved as $\mathtt{Z}^T\mathtt{QZ}$ or $\sum \mathtt{f}^2$

Table 9.2: SOSTOOLS parameters useful in the evaluation of the validity of a solution barrier function.

A final test that the solution is valid is to check that the SOS inequalities, i.e. the formulation of the constraints according to Definition 8.3, are SOS. This can be done by testing that each of the expressions can be resolved in the form in equation (8.1) or equation (8.3), i.e. as $\sum f^2(\mathbf{x})$ or $\mathbf{z}^T(\mathbf{x})\mathbf{Q}\mathbf{z}(\mathbf{x})$ with the function `findsos` using the polynomial coefficients from the solution (using `sosgetsol` as described in table 9.1). If any of the three expressions cannot be resolved in these forms, it can be concluded that it is not SOS, and hence the solution does not comply with the requirements for a barrier certificate, i.e. the solution is invalid.

9.2 Barrier Certificate Search for First Order Robot Slide System

In order to search for a barrier certificate the robot slide system from chapter 4 is reintroduced, and will be recapitulated along with new notation. As illustrated in figure 9.1 the physical limits of the slide movement, ± 10 cm, defines the set $\mathcal{X}$, and the unsafe region $\mathcal{X}_u$ is the upper 5 cm of this interval. The safe set should be as large as possible, but due to the fact that the barrier certificate must have a minimum value of $\bar{\varepsilon}$ on the unsafe set, the safe set $\mathcal{X}_0$ is separated from $\mathcal{X}_u$ by a small distance Δ, as was illustrated in figure 8.2. This can be summarized (with units in meter) as

- Considered subset of the state space is the interval between the physical limits of the slide movement, i.e. $\mathcal{X} = \{x_1 \in [-0.1, 0.1]\} \subset \mathbb{R}$
- The unsafe set is $\mathcal{X}_u = \{x_1 \in [0.05, 0.1]\} \subset \mathcal{X}$
- The safe set is as much of the remaining part of the considered set as possible, i.e. $\mathcal{X}_0 = \{x_1 \in [-0.1, 0.05 - \Delta]\} \subset \mathcal{X} \setminus \mathcal{X}_u$

It is noted that the design criteria differ from the criteria in chapter 4 given that the barrier certificate now must be valid in the entire safe region.

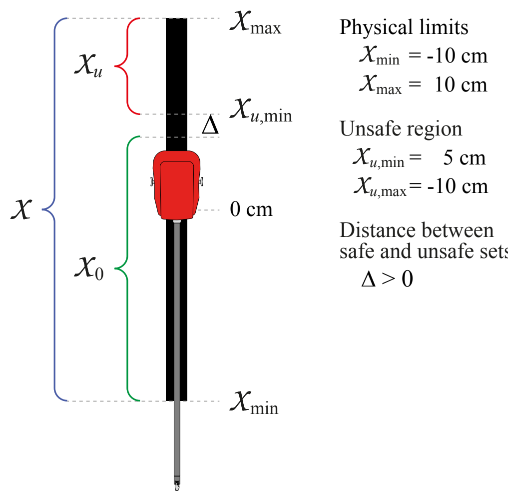


Figure 9.1: The boundaries in slide position of the robotic instrument for each of the sets $\mathcal{X}$, $\mathcal{X}_u$ and $\mathcal{X}_0$, is visualized for the instrument house.

Using the first order linear model of the robot slide system described in subsection 4.1.1, and the linear position controller described in subsection 4.3.1 with proportional gain $\mathbf{K}$ and unity gain between reference and position secured by $\bar{\mathbf{N}} = \mathbf{K} + 1$, the closed-loop system is recapitulated as

$$\begin{aligned}\dot{x}_1 = \mathbf{A}x_1 + \mathbf{B}u &= \mathbf{A}x_1 + \mathbf{B}(\bar{\mathbf{N}}x_{\text{ref}} - \mathbf{K}x_1) \\ &= \underbrace{(\mathbf{A} - \mathbf{BK})}_{-\tau^{-1}(\mathbf{K}+1)} x_1 + \underbrace{\mathbf{B}\bar{\mathbf{N}}}_{\tau^{-1}(\mathbf{K}+1)} x_{\text{ref}} \\ &= \tau^{-1}(\mathbf{K}+1)(x_{\text{ref}} - x_1), \qquad \text{with } \tau = 110\,\text{ms}\end{aligned} \tag{9.1}$$

This system is used in the following subsections in the barrier function search. First with a reference in zero giving detailed explanations of the program formulation, and subsequently for the same system, testing for how wide a range of references safety can be guaranteed. Last, a coordinate shift from the

position-reference-space to the error-space is performed in order to simplify the search for reference intervals yielding valid solutions.

9.2.1 Safety Verification of First Order System with Zero Reference

To give a clear picture of the structure of the SOS program, an initial exhaustive example is given for the one-dimensional first order system with zero as reference position. Commands associated with SOSTOOLS is marked in brown in the following code snips. The full code can be found in subsection G.2.1 and in appendix J on the path `matlab_scripts/sostools/1storder_noRef.m`

Search for a barrier certificate by first defining the open-loop system, and designing a controller (with pole placement) as described in subsection 4.3.1.

```
1 % Time constant from measurement
2 tau = 0.11;
3 % State−space matrices from first order system
4 A = −1/tau;
5 B = 1/tau;
6 K = place(A,B,10*eig(A));
```

Define the desired distance Δ between the safe and unsafe sets along with the minimum value $\bar{\varepsilon}$ of the barrier function on the unsafe set.

```
1 % Distance between defined safe and unsafe regions
2 delta = 1e−3;
3
4 % Minimum value of the barrier certificate on the unsafe set Xu
5 epsilon = 1e−3;
```

Then the symbolic state variables are declared for the SOS program in SOSTOOLS with the command `pvar`. Now the SOS program `prog` can be initialized using the function `sosprogram` which takes the state variable as input.

```
1 % Declare state variables
2 pvar x1
3
4 % Initialize the sum of squares program
5 prog = sosprogram(x1);
```

The vector field or derivative of the state can now be defined in terms on the symbolic state variable. This function is necessary for the SOS program when requiring that the Lie derivative of the barrier certificate must be negative on the set $\mathcal{X}$.

```
1 % Vector field dx/dt = fx (closed loop)
2 fx = (A−B*K)*x1;
```

For ease of defining a (1D) function g that is positive on an interval $[p_1, p_2]$, a parabola function is used.

```
1 function [a,b,c] = parabola(p1,p2,a)
2     if ~exist('a','var')
3         a=−1;
4     end
5     b=a*(p1^2−p2^2)/(p2−p1);
6     c=−a*p1^2−b*p1;
7 end
```

Now declare the polynomial barrier function with the command `sospolyvar`. To do this, a monomial vector must be specified with `monomials` (see the monomial example in equation (8.2)), which takes the

state variable and the monomial degree(s) as input. The monomial degrees for $B(x_1)$ are chosen as low as possible until a solution can be found. In this case a solution can be found for a degree of $B(x_1)$ that is [0:4], i.e. a polynomial in x_1 of degrees zero through four.

```
1 % Declare the polynomial barrier function
2 zB = monomials(x1,0:4);
3 [prog,Bar] = sospolyvar(prog,zB);
```

Now the set $\mathcal{X}$ can be defined as the slide region according to figure 9.1 using the Lie derivative inequality in equation (8.8)c, which is defined with the command `sosineq`. The SOS polynomials q are of the form in equation (8.3), i.e. $q = \mathbf{z}^T \mathbf{Q} \mathbf{z}$ (so the degree of q is twice the degree of the monomial vector $\mathbf{z}$), and are declared with the command `sossosvar`, also taking a monomial vector as input.

```
1 % Define space X in Rn
2 [a,b,c] = parabola(−0.1,0.1); % get coefficients for parabola which is positive for x in [−0.1,0.1] m
3 gX = a*x1^2+b*x1+c;
4
5 zX = monomials(x1,0:4);
6 [prog,qX] = sossosvar(prog,zX);
7
8 prog = sosineq(prog,−diff(Bar,x1)*fx−gX*qX);
```

Similarly, the unsafe region $\mathcal{X}_u$ is defined according to the SOS inequality in equation (8.8)b as the area between slide positions 5-10 cm as given by figure 9.1.

```
1 % Define space Xu in X
2 [a,b,c]=parabola(0.05,0.1);
3 gXu = a*x1^2+b*x1+c;
4
5 zXu = monomials(x1,0:4);
6 [prog,qXu] = sossosvar(prog,zXu);
7
8 prog = sosineq(prog,Bar−epsilon−gXu*qXu);
```

And finally the region $\mathcal{X}_0$ is defined according to the SOS inequality in equation (8.8)a as $\mathcal{X}_0 \subset \mathcal{X} \setminus \mathcal{X}_u$, separated from the unsafe set by the distance Δ.

```
1 % Define space X0 in X
2 [a,b,c]=parabola(−0.1,0.05−delta);
3 gX0 = a*x1^2+b*x1+c;
4
5 zX0 = monomials(x1,0:4);
6 [prog,qX0] = sossosvar(prog,zX0);
7
8 prog = sosineq(prog,−Bar−gX0*qX0);
```

With all three areas defined according to Definition 8.3, the program is ready to be solved by using the command `sossolve`. If a solution is found, an overview of the solution accuracy is printed in the MATLAB terminal as the residual norm, feasibility ratio, number of iteration steps and solving time. To get the polynomial $B(x_1)$ use the function `sosgetsol`.

```
1 % Solve for barrier certificate
2 prog = sossolve(prog);
3 getB = sosgetsol(prog,Bar)
```

From the terminal printout it is verified that the problem is neiter primal or dual infeasible, and that the feasibility ratio for this solution is given as 1.0122, which is fairly close to 1 and hence indicates that the solution is valid. The solution is found in 15 iterations with a residual norm of 7.5521e-10 and thereby no indication of numerical errors.

To additionally verify that the solution is indeed valid, it is tested that the solution complies with the inequalities being SOS by testing if they can be resolved to the form in equation (8.3).

```
% Get coefficients for the remaining polynomials
getdBdx = diff(getB,x1)
getqXu1 = sosgetsol(prog,qXu);
getqX01 = sosgetsol(prog,qX0);
getqX1 = sosgetsol(prog,qX);

% Test if the inequalities are SOS
[Q,~,~] = findsos(getB−epsilon−gXu*getqXu1);
[Q2,~,~] = findsos(−getB−gX0*getqX01);
[Q3,~,~] = findsos(−detdBdx*fx−gX*getqX1);
```

This is indeed the case, and it is thereby verified that a barrier certificate is found for the closed-loop system in equation (9.1), namely

$$B(x_1) = 373.0249 \cdot x_1^4 + 151.3339 \cdot x_1^3 + 16.8843 \cdot x_1^2 - 6.4509e\text{-}6 \cdot x_1 - 0.061301 \tag{9.2}$$

The first three coefficients of the polynomial barrier certificate are of ample size (not in the order 1e-5 or less), while the fourth seem unimportant. Decreasing the degree of the monomial $\mathbf{z}_B$ (and thereby the order of B) to [0,2:4] gives the solution

$$B(x_1) = 290.559 \cdot x_1^4 + 112.4642 \cdot x_1^3 + 12.2165 \cdot x_1^2 - 0.044304 \tag{9.3}$$

which is also a valid barrier certificate, found in 13 iterations with feasibility ratio 0.9797 and a residual norm of 1.0412e-09. The two barrier certificates are depicted in figure 9.2, from which it is seen that both comply with the requirements for a barrier certificate, i.e. they are positive ($B(x_1) \geq \bar{\varepsilon}$) on $\mathcal{X}_u = \{x_1 \in [0.05, 0.1]\}$ and negative on $\mathcal{X}_0 = \{x_1 \in [-0.1, 0.05 - \Delta]\}$, and their Lie derivatives are nonpositive on $\mathcal{X} = \{x_1 \in [-0.1, 0.1]\}$, in accordance with Definition 2.2. It is also seen that the value of each of the barrier certificates on the set $\mathcal{X}$ seem reasonable being neither exceptionally small or large, verifying that the choice of $\bar{\varepsilon}$ seems reasonable.

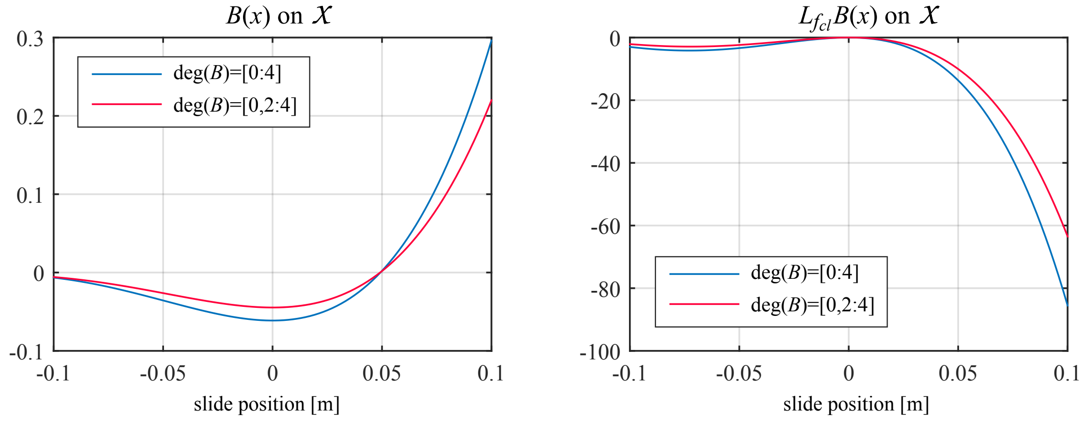


Figure 9.2: Barrier certificates found with SOSTOOLS for the first order system in equation (9.1) with zero reference, that comply with the requirements in Definition 8.3.

It is noted that the safety verification is accomplished with a fourth order polynomial barrier certificate, while it can be argued that it could be possible with a second order polynomial. This is, however, not important for the safety verification of the system, where the conclusion is that a solution can be found, thus the system is safe.

9.2.2 Verifying a Range of Reference Positions for the First Order System

Now a non-zero reference is introduced. As it is desired to verify safety for a whole range of references, the barrier polynomial is now formulated as a function of both position and reference $B(x_1, x_{\text{ref}})$, thus introducing the reference as an independent state implying an augmented state space equation equivalent to equation (9.1):

$$\dot{\mathbf{x}} = \begin{bmatrix} x_1 \\ x_{\text{ref}} \end{bmatrix}^{\cdot} = \begin{bmatrix} \mathbf{A}+\mathbf{BK} & \mathbf{B}\bar{\mathbf{N}} \\ 0 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_{\text{ref}} \end{bmatrix} = \begin{bmatrix} -\tau^{-1}(\mathbf{K}+1) & \tau^{-1}(\mathbf{K}+1) \\ 0 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_{\text{ref}} \end{bmatrix} \tag{9.4}$$

In the SOS program constraints are set up for the new variable `xref` for each of the sets, included as extra terms in the sums in Definition 8.3, e.g. for the set $\mathcal{X}$ a new term with SOS polynomial $q(\mathbf{x})$ and polynomial $g(x_{\text{ref}})$, positive on the reference interval `[rMin, rMax]`, is included in the inequality as:

```
1 % Constraint on the set X being nonpositive for the interval of references
2 [a,b,c] = parabola(rMin,rMax);
3 gX2 = a*xref^2+b*xref+c;
4
5 zX2 = monomials([x1,xref],0:2);
6 [prog,qX2] = sossosvar(prog,zX2);
7
8 prog = sosineq(prog,-[diff(Bar,x1) diff (Bar,xref) ]*[ fx ;0] - gX1*qX1 - gX2*qX2);
```

The goal is to verify safety of this system for a range of references, preferably for all references within the safe position interval, such that the sets would be as depicted in figure 9.3a.

Expected Barrier Certificate Geometry

In order to get a picture of the expected outcome, a brief analysis of the considered state space is made. It can be seen from equation (9.4) that the system vector field will be zero when $x_1 = x_{\text{ref}}$, which in turn means that the Lie derivative of the barrier function will be zero for $x_1 = x_{\text{ref}}$, marked in figure 9.3b with a green line. To the upper left of this line the system will have a positive time derivative, thus requiring that the derivative of the barrier function with respect to x_1 must be negative in this region in order to comply with equation (2.5)c i.e. $L_{f_{cl}}B(\mathbf{x}) \leq 0$. Thus, to the left of this line the barrier polynomial will have decreasing values in the positive x_1 direction. To the lower right of the green line the system will

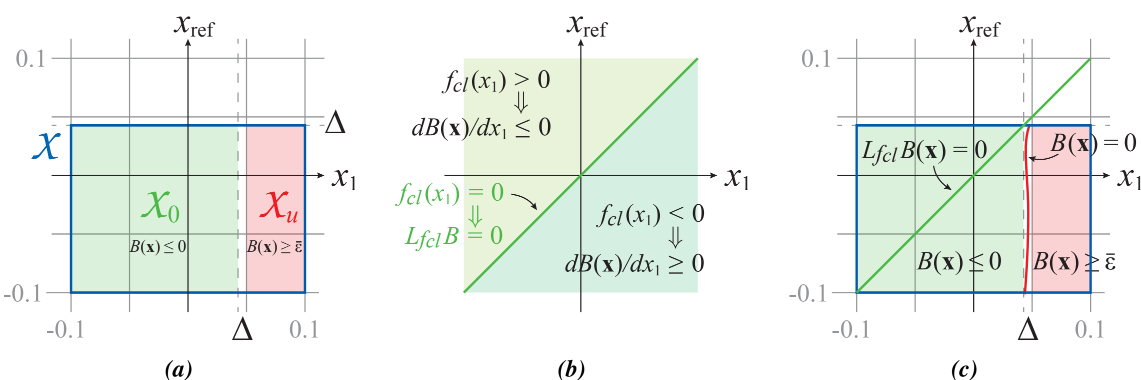


Figure 9.3: Outline of the safe and unsafe sets marking the value requirements for the barrier certificate as a function of the robot position and the position reference. Figure 9.3b sketches that $L_{f_{cl}}B(\mathbf{x}) = 0$ in $x_1 = x_{\text{ref}}$ as inferred from equation (9.1).

have negative time derivative, thus analogously requiring that the derivative $dB(\mathbf{x})/dx_1$ must be positive in this region, meaning that $B(\mathbf{x})$ will have increasing values in the positive x_1 direction here.

In figure 9.3c the zero level set of a barrier function is sketched with a red line between the safe and unsafe regions. As this level set gets closer to the green line indicating the zero-value of the Lie derivative for increasing values of x_{ref}, it is expected that the value of $B(\mathbf{x})$ is increasing along this line in the positive x_{ref} direction.

Results and Conclusions

A number of tests are run with SOSTOOLS varying the values of the parameters $\bar{\varepsilon}$, Δ, and the degree of the SOS polynomials q_j and the polynomial $B(\mathbf{x})$, in order to see for how high a level of the upper end of the allowed reference interval, `rMax`, solutions can be found. The MATLAB implementation can be found in appendix G.2.2 and in appendix J under the path `matlab_scripts/sostools/1storder_withRef.m`. As seen from figure 9.3 $\bar{\varepsilon}$ is the minimum value of the barrier function on the unsafe set, and increasing this may require the zero level set to be pushed further away from $\mathcal{X}_u$. This may in turn require also increasing the distance Δ to the safe set, i.e. contracting the region $\mathcal{X}_0$. The conclusions are summarized in table 9.3.

Parameter	Effect of variation
$\deg(B)$	In general the feasibility ratio is better (closer to one) when testing for higher degrees of $B(\mathbf{x})$ ([0:6] or [0:8] compared to [0:4]), and solutions can be found for larger intervals of the reference when $B(\mathbf{x})$ has higher degree.
$\deg(q_j)$	Increasing the degree of the SOS polynomials (monomial degrees [0:6] compared to [0:2] or [0:4]) generally degrades the feasibility ratio.
$\bar{\varepsilon}$	Increasing $\bar{\varepsilon}$ decreases the allowed reference interval and also shows a trend of slightly increasing the residual norm. Increasing $\bar{\varepsilon}$ iteratively proves that gradually an increase in Δ is also required in order for solutions to be found.
Δ	Generally increasing Δ will also decrease the allowed interval of references, and shows a trend of decreasing the residual norm until some limit.
$\mathbf{K}$	Lowering the gain of the controller increases the allowed reference interval.

Table 9.3: Effect of varying different parameters in the SOS program; see figure 8.2 for a visualization of $\bar{\varepsilon}$ and Δ. Results are only included for solutions where all inequalities were verified to be SOS with `findsos`.

Numerical problems are reported for all solutions found, and the residual norms (size of numerical error in the solution) are in general in the order of -3 and -4, so it is desired to keep the degree of polynomials low and to increase the value of $\bar{\varepsilon}$, compared to subsection 9.2.1 to be sure that the numerical error of the solution does not cause the barrier polynomial to attain nonpositive values on the unsafe set.

The choice of parameter values are explained in table 9.4 and the barrier certificate is plotted in figure 9.4 separately for the safe and unsafe sets to validate the sign of $B(\mathbf{x})$ on each set. Evaluating the curve for the barrier polynomial and its Lie derivative, the geometric considerations are verified in that $B(\mathbf{x})$ indeed is (slightly) decreasing along the x_1 axis to the left of the line $x_1 = x_{\text{ref}}$, and increasing to the right, as seen on the plot of $\mathcal{X}_0$ (and increasing along the x_1 axis on the entire set $\mathcal{X}_u$). It is also verified that the Lie derivative is zero along the line $x_1 = x_{\text{ref}}$. Thereby it is verified that the barrier certificate is valid according to Definition 2.2, guaranteeing safety for the first order system in equation (9.4) with references in the interval $x_{\text{ref}} \in [\texttt{rMin}, \texttt{rMax}] = [-0.1, 0.017]\,\text{m}$.

Choice	Reason
$\deg(B) = [0{:}6]$	Increasing the degree of $B(\mathbf{x})$ from [0:4] to [0:6] elevates the upper limit for allowed references. Increasing the degree (keeping the remaining parameters constant) yields an infeasible solution.
$\deg(q_j) = [0{:}2]$	Keeping the degree low, as increased degree entails poorer feasibility ratio and greater residual norm.
$\bar{\varepsilon} = 0.1$	Smaller values of $\bar{\varepsilon}$ allows solutions to be found for larger values of `rMax`, while the residual norm speaks in favour of increasing $\bar{\varepsilon}$. A compromise is made, as this choice of $\bar{\varepsilon}$ gives a residual norm that is approximately twice as big as with $\bar{\varepsilon} = 0.01$, while it is almost an order of magnitude smaller than for $\bar{\varepsilon} = 1$.
$\Delta = 0.015$	It is preferred to have as small a value of Δ as possible, as this increases the value of `rMax` for which solutions can be found. On the other hand lower values of Δ deteriorate the feasibility ratio. The value is increased slightly along with $\bar{\varepsilon}$ until a solution is found.
$\mathbf{K} = 0.2$	Lower gain allows a larger value of `rMax` (the demand $L_{f_{cl}}B(\mathbf{x}) \leq 0$ is affected by $\mathbf{K}$), and it is known from the implementation described in section 4.5, that at present time the controller gain implemented is approximately this size.

Table 9.4: Chosen value for each of the parameters. The barrier certificate is plotted in figure 9.4.

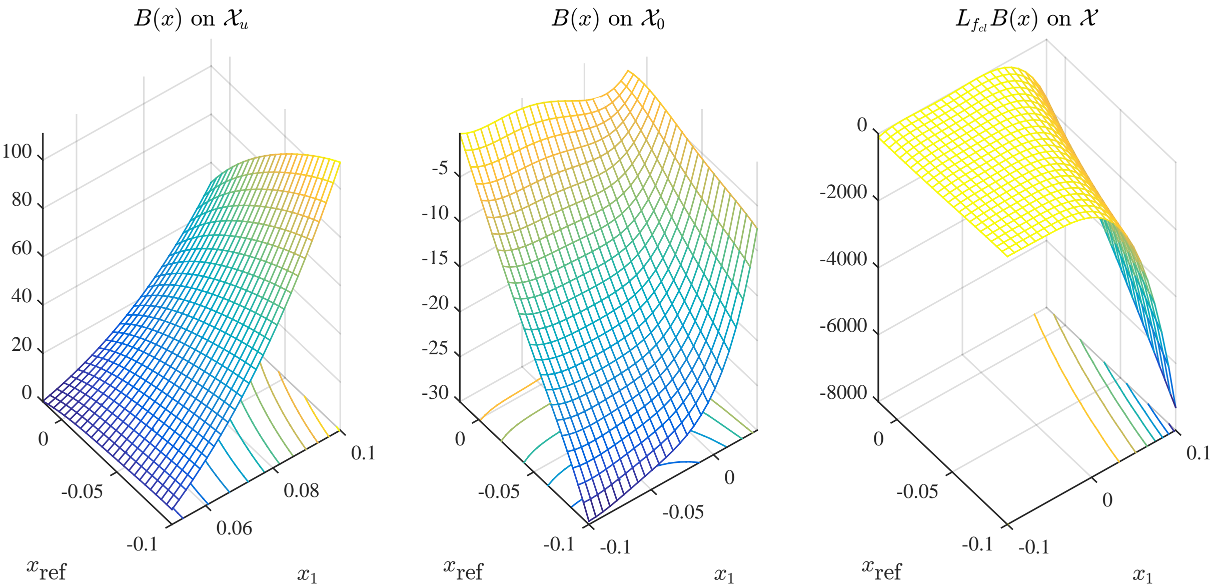


Figure 9.4: Barrier certificate of degree [0:6], all SOS polynomials of (monomial) degree [0:2], $\bar{\varepsilon} = 0.1$, $\Delta = 0.015$ and gain $\mathbf{K} = 0.2$. The solution is found for an interval of references $\mathcal{X} = \{x_{\text{ref}}[-0.1, 0.017]\}$, with `feasratio=0.9888` and `Residual norm=3.5e-4`. The plots can be generated by running the script `1storder_withRef.m` such that a custom 3D rotation is possible (located in appendix G.2.2 under the path `matlab_scripts/sostools/1storder_withRef`).

It can be concluded that this approach to determining for which references system safety can be guaranteed is not ideal. The tests conducted are not exhaustive, as the possibilities of combining parameter values are vast. This also means that it is highly possible that a "better" barrier certificate can be found certifying safety for a wider range of references without compromising on the feasibility ratio. It is in particular desired to increase the value of `rMax` such that it gets closer to $\mathcal{X}_{u,\max}$. In order to decrease the

number of parameters, a different approach is used in the following.

9.2.3 Considering the Error as the Independent Variable

As it is seen from table 9.3 there are many tuning parameters when searching for a barrier certificate in SOSTOOLS, and iteratively finding a combination giving as large an interval of allowed references outside $\mathcal{X}_u$ as possible can be a long process. In the previous section a barrier certificate was found validating the system for references in the interval $x_{\text{ref}} \in [-0.1, 0.017]$ m, i.e. references up to a distance of 3.3 cm from the unsafe set, $\mathcal{X}_u = \{x_1 \in [0.05, 0.1]\}$ (see figure 9.1).

A different approach is now used in order to validate system safety for references closer to the unsafe set. Instead of formulating the barrier certificate in terms of the robot position and the position reference as in equation (9.1), the dimensionality of the problem can be reduced through a coordinate shift to the error state as seen in figure 9.5a, the error being:

$$x_{\text{err}} = x_{\text{ref}} - x_1$$

giving the error dynamics

$$\begin{aligned} \dot{x}_{\text{err}} = -\dot{x} &= -\tau^{-1}(\mathbf{K}+1)(x_{\text{ref}} - x_1) \\ &= -\tau^{-1}(\mathbf{K}+1)x_{\text{err}} \end{aligned} \tag{9.5}$$

Now safety can be tested for relative positions instead of absolute.

Relating the Error Space Sets to the Sets for Absolute Position

As seen from figure 9.1, the unsafe region is the upper part of the interval $\mathcal{X}$, which means that for references given outside the unsafe set, unsafety of the system will only occur if the robot end effector position is above the reference, corresponding to a negative error. This is illustrated in figure 9.5b. Hence it is required to restrict the error in the negative direction to some value -δ_{err}, thus restricting the upper value of allowed (safe) references to a distance δ_{err} from the unsafe region, as sketched in figure 9.5c.

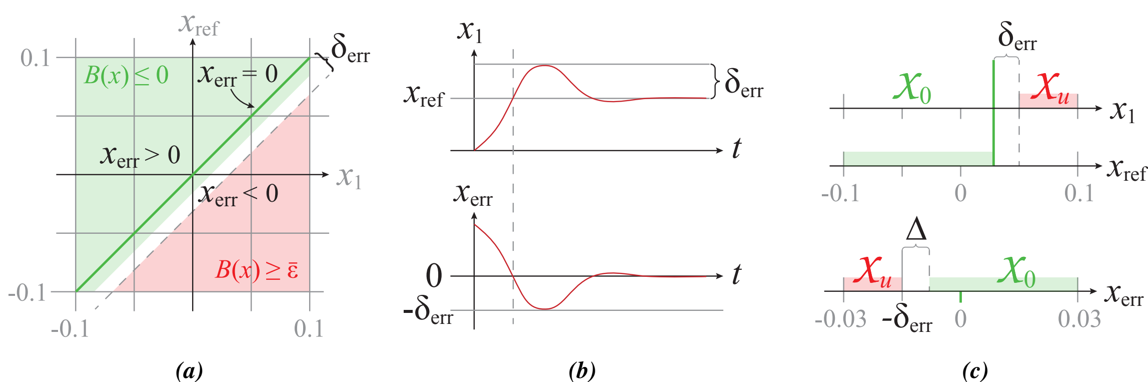


Figure 9.5: If the error is certified to stay above the value -δ_{err}, system safety is guaranteed for all references up to a safety distance of δ_{err} from the unsafe set.

When restricting the error to a lower limit -δ_{err}, this restriction corresponds to not allowing the end effector position more than the distance δ_{err} above the reference anywhere outside the unsafe region. However, if a barrier certificate can be found for the error state system with unsafe set being values

below -δ_{err}, this means that safety of the system in equation (9.1) can be guaranteed for references up to the distance δ_{err} from the unsafe region.

The system equation in equation (9.1) is tested and known to be valid for small steps of approximately 5 mm, thus the set to be considered in the error state space is chosen to ± 3 cm. This gives the set definitions for the error state system:

- The set considered is well over the usual reference step size of approximately 5 mm, $\mathcal{X} = \{x_{\text{err}} \in [-0.03, 0.03]\}$.
- The unsafe set includes values below -δ_{err}, $\mathcal{X}_u = \{x_{\text{err}} \in [-0.03, -\delta_{\text{err}}]\}$.
- The safe set is a distance Δ from the unsafe set, where $\Delta < \delta_{\text{err}}$ such that the safe set will include $x_{\text{err}} = 0$, $\mathcal{X}_0 = \{x_{\text{err}} \in [-\delta_{\text{err}} + \Delta, 0.03]\}$.

If a valid solution can be found, it will certify that steps in positive direction (upwards) of 3 cm is acceptable, and will never yield an error below $-\delta_{\text{err}}$, and that references can safely be given as long as they have a distance of at least δ_{err} to the unsafe positions, hence certifying safety of the system:

- The set considered is the set described in figure 9.1, $\mathcal{X} = \{x_1 \in [-0.1, 0.1]\}$.
- The unsafe set is also seen in figure 9.1, $\mathcal{X}_u = \{x_1 \in [0.05, 0.1]\}$.
- The safe set for the references is $\mathcal{X}_0 = \{x_{\text{ref}} \in [-0.1, 0.05 - \delta_{\text{err}}]\}$ ensuring that $\mathcal{X}_0 \subseteq \mathcal{X} \setminus \mathcal{X}_u$.

Results and Conclusions

The parameters $\bar{\varepsilon}$, Δ, δ_{err}, and the degree of the SOS polynomials q_j and the polynomial $B(x_{\text{err}})$, are tweaked to find the smallest possible value of δ_{err} yielding a valid solution. The MATLAB implementation of this approach can be found in appendix G.2.2 and in appendix J under the path `matlab_scripts/sostools/1storder_error.m`. The findings conform with the conclusions presented in table 9.3, with the additional conclusions listed in table 9.5.

Parameter	Effect of variation
$\deg(B)$	In general the residual norm is lower when testing for higher degrees of $B(x_{\text{err}})$ ([0:6] compared to [0:4]).
$\deg(q_j)$	Increasing the degree of the SOS polynomials (monomial degrees [0:4] compared to [0:2]) generally increases the residual norm. Having different degrees for the different SOS polynomials also generally increases the residual norm.
$\bar{\varepsilon}$	Increasing $\bar{\varepsilon}$ in general increases the residual norm of the solution.
Δ	Decreasing Δ too much will preclude a solution to be found, otherwise Δ does not have much influence on the solution.
$\mathbf{K}$	Lowering the gain of the controller decreases the residual norm of the solution.
δ_{err}	Decreasing δ_{err} increases the residual norm of the solution.

Table 9.5: Effect of varying different parameters in the SOS program. Results are only included for solutions where all inequalities were verified to be SOS.

The choice of parameter values are explained in table 9.6 and the barrier certificate is plotted in figure 9.6. It is seen that the barrier certificate is positive on the unsafe region below -9 mm, i.e. $\mathcal{X}_u = \{x_{\text{err}} \in [-0.03, -\delta_{\text{err}}]\} = \{x_{\text{err}} \in [-0.03, -0.009]\}$, and nonpositive on the safe region $\mathcal{X}_0 = \{x_{\text{err}} \in [-\delta_{\text{err}} + \Delta, 0.03]\} = \{x_{\text{err}} \in [-0.005, 0.03]\}$, and that its Lie derivative is nonpositive on the entire set $\mathcal{X} = \{x_{\text{err}} \in [-0.03, 0.03]\}$, confirming that it is a valid barrier certificate in accordance with Definition 2.2.

Choice	Reason
$\deg(B) = [0{:}6]$	Increasing the degree of the barrier certificate from [0:4] to [0:6] does not change the curve much on the interval $\mathcal{X}$, while it does lower the residual norm an order.
$\deg(q_j) = [0{:}4]$	The degree of the SOS polynomials are chosen as low as possible allowing a solution to be found with the remaining parameters.
$\bar{\varepsilon} = 1e\text{-}2$	Increasing $\bar{\varepsilon}$ by an order of magnitude scales up the barrier certificate approximately ten times in value, but also increases the residual norm approximately an order of magnitude. A compromise is made choosing a scaling of $B(x_{\text{err}})$ yielding neither very small nor very large values of $B(x_{\text{err}})$ on the set $\mathcal{X}$, still giving a solution with relatively low residual norm.
$\Delta = 4e\text{-}3$	The value of Δ is desired as low as possible, and at least lower than δ_{err} (such that the safe set will include $x_{\text{err}} = 0$). The lowest value yielding a solution is chosen.
$\mathbf{K} = 0.2$	A low gain is chosen identical to the choice in subsection 9.2.2, as implementable gains are ascertained to be of this order.
$\delta_{\text{err}} = 9e\text{-}3$	No solutions could be found for $\delta_{\text{err}} < 9\,\text{mm}$.

Table 9.6: Chosen value for each of the parameters. The barrier certificate is plotted in figure 9.6.

Relating this relative position to the absolute positions, this certificate guarantees safety for the system in equation (9.1) with references up until a minimum distance of 9 mm from the unsafe set in $\mathcal{X}_u = \{x_1 \in [0.05, 0.1]\}$ (see figure 9.1), i.e. system safety is guaranteed for references in the interval $\mathcal{X}_0 = \{x_{\text{ref}} \in [-0.1, 0.041]\}$. Thus, this barrier certificate verifies safety for a considerably less restrictive set of references, as it guarantees safety for references at distances as low as 9 mm from the unsafe region, compared to the 3.3 cm found with the approach described in subsection 9.2.2.

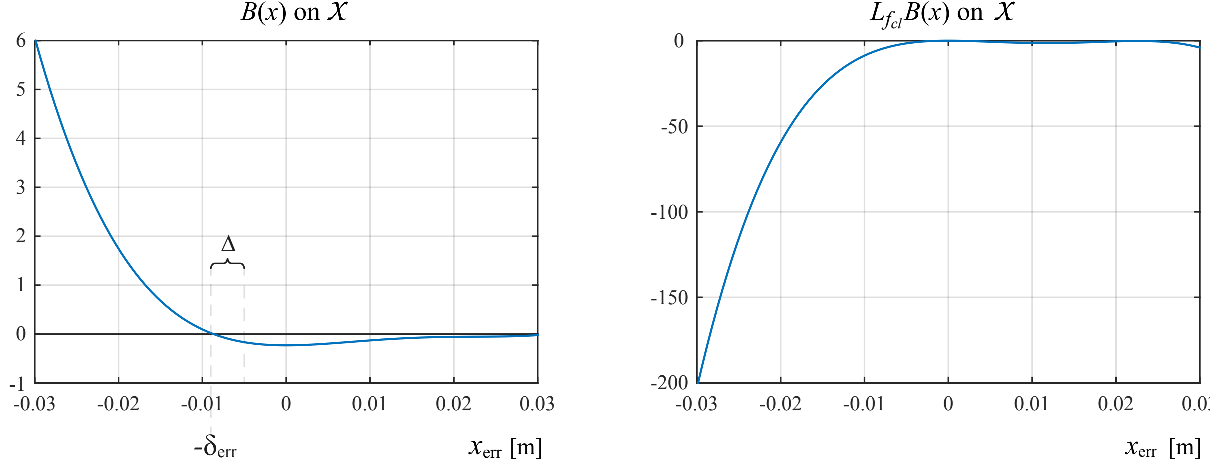


Figure 9.6: Barrier certificate of degree [0:6], all SOS polynomials of degree [0:4], with $\bar{\varepsilon} = 1e\text{-}2$, $\Delta = 4e\text{-}3$, $\delta_{\text{err}} = 9e\text{-}3$ and gain $\mathbf{K} = 0.2$. The solution has a `feasratio=1.0262` and `Residual norm=2.8e-7`.

A corresponding positive value can be found restricting the error in the positive direction hence also restricting how much the end effector position is allowed to be below any reference. Although this is not tested, it is clear that if a barrier certificate for the error can be found ensuring that the error will stay within the interval $\mathcal{X}_0 \subseteq \{x_{\text{err}} \in [-\delta_{\text{err}}, \delta_{\text{err}}]\}$ i.e. with unsafe regions on both sides of this interval, this means that for any reference, the end effector position can be guaranteed never to be more than a distance δ_{err} away from the reference.

9.3 Barrier Certificate Search for Second Order Robot Slide System

To complete the construction of barrier certificates with SOSTOOLS for the robot slide movement parallel to the construction of CBFs in chapter 4, and in order to test SOSTOOLS on a system of a higher state space dimension, finally a barrier certificate is found for the second order model of the robot slide position from subsection 4.1.2.

As for the first order model in section 9.2, safety is validated for the closed loop system using a linear position controller. The system is tested for safety compliance using the same controller gains as presented in subsection 4.3.2 with proportional gain $\mathbf{K} = [K_1 \;\; K_2] = [5.173 \;\; 0.214]$ and unity gain between reference and position secured by $\bar{\mathbf{N}} = K_1 + 1 = 6.173$. The closed-loop second order system with $x_1 =$ position and $x_2 =$ velocity of the end effector, is recapitulated as

$$\dot{\mathbf{x}} = \dot{\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}} = \begin{bmatrix} 0 & 1 \\ -\omega_n^2 & -2\zeta\omega_n \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 0 \\ \omega_n^2 \end{bmatrix} \left(\bar{\mathbf{N}} x_{\text{ref}} - \underbrace{\begin{bmatrix} K_1 & K_2 \end{bmatrix}}_{\mathbf{K}} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \right), \quad \text{with} \quad \begin{matrix} \omega_n = 17\,\text{rad/s} \\ \zeta = 0.55 \end{matrix} \tag{9.6}$$

From the analysis in subsection 9.2.3 it is found that a coordinate shift from absolute to relative positions in the form of the position error proves the more efficient method to validate system safety for a wide range of references. Using the position error $x_{\text{err},1}$ as the free variable and letting $x_{\text{err},2}$ signify the rate of change of the error, the dynamics of the error state can be expressed as

$$\begin{aligned} x_{\text{err},1} &= x_{\text{ref}} - x_1 \\ \dot{x}_{\text{err},1} &= -\dot{x}_1 = -x_2 = x_{\text{err},2} \\ \ddot{x}_{\text{err},1} &= \dot{x}_{\text{err},2} = -\dot{x}_2 = \omega_n^2 x_1 + 2\zeta\omega_n x_2 - \omega_n^2(\bar{\mathbf{N}} x_{\text{ref}} - (K_1 x_1 + K_2 x_2)) \\ &\qquad\qquad\qquad = (2\zeta\omega_n + K_2)x_2 - \omega_n^2(K_1+1)(x_{\text{ref}} - x_1) \\ &\qquad\qquad\qquad = -(2\zeta\omega_n + K_2)x_{\text{err},2} - \omega_n^2(K_1+1)x_{\text{err},1} \end{aligned}$$

which can be compressed to standard state space form as

$$\dot{\mathbf{x}}_{\text{err}} = \dot{\begin{bmatrix} x_{\text{err},1} \\ x_{\text{err},2} \end{bmatrix}} = \begin{bmatrix} 0 & 1 \\ -\omega_n^2(K_1+1) & -(2\zeta\omega_n + K_2) \end{bmatrix} \begin{bmatrix} x_{\text{err},1} \\ x_{\text{err},2} \end{bmatrix} \tag{9.7}$$

Relating the Error State Space Sets to the Sets for Absolute Position

The relation between the state space system of absolute positions in equation (9.6) and relative positions in equation (9.7) is equivalent to the relation presented in subsection 9.2.3. Again, as seen in figure 9.1, the unsafe positions are positions in the upper end of the interval of positions in the set $\mathcal{X}$, which means that for references given outside the unsafe set, the system can only be unsafe if the robot end effector position is above the reference, corresponding to a negative error. This is illustrated in figure 9.7b with an overshoot, although the system presented should not overshoot due to the placement of real poles (in -40 and -50, see equation (4.15)). When the error can be guaranteed to stay above a value -δ_{err}, this is equivalent to guaranteeing that the end effector position will never be more than the distance δ_{err} above the reference which in turn means that the system is safe for all references with a minimum distance δ_{err} to the unsafe region. This is illustrated in figure 9.7c.
Equivalent to the first order system, the model in equation (9.6) is tested and known to be valid for

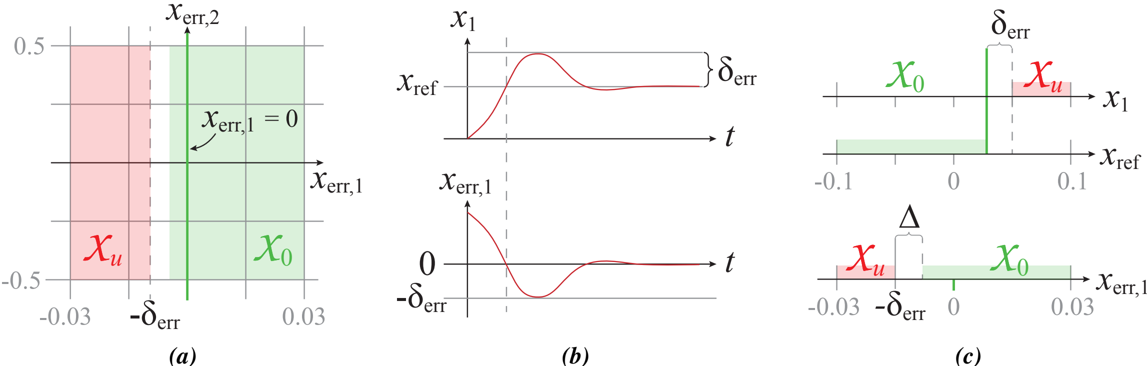


Figure 9.7: With references given outside the unsafe region, system safety can be guaranteed if the error is certified to stay above the value $-\delta_{\text{err}}$. When this is the case, system safety is guaranteed for all references up to a safety distance of δ_{err} from the unsafe set.

small steps of approximately 5 mm, and again the set X to be considered is chosen to ± 3 cm. For the second order system, the sets X, X_u and X_0 must also be specified for the velocity dimension. As the rate of change of the error is related to the end effector velocity by a factor -1, the considered interval of velocities is chosen as the upper limits for the robot slide velocity of approximately ± 0.5 m/s. This is visualized in figure 9.7a, and gives the set definitions for the second order error state system:

- The set considered is well over the usual reference step size of 5 mm, and the the upper bound on the velocity considered is the physical limits of the system, $X = \{x_{\text{err},1} \in [-0.03, 0.03],\ x_{\text{err},2} \in [-0.5, 0.5]\}$.
- The unsafe set includes relative positions below -δ_{err}, $X_u = \{x_{\text{err},1} \in [-0.03, -\delta_{\text{err}}],\ x_{\text{err},2} \in [-0.5, 0.5]\}$.
- The safe set is a distance Δ from the unsafe position, where $\Delta < \delta_{\text{err}}$ such that the safe set will include $x_{\text{err},1} = 0$, $X_0 = \{x_{\text{err},1} \in [-\delta_{\text{err}} + \Delta, 0.03],\ x_{\text{err},2} \in [-0.5, 0.5]\}$.

If a valid solution can be found, it will certify that that steps in positive direction (upwards) of 3 cm and system velocities up to ± 0.5 m/s are acceptable, and will never yield an error below -δ_{err}, which means that references can safely be given as long as they have a distance of at least δ_{err} to the unsafe positions, hence certifying safety of the system:

- The positions considered are as described in figure 9.1, and the velocity is bounded by the physical limits of the system, $X = \{x_1 \in [-0.1, 0.1],\ x_2 \in [-0.5, 0.5]\}$.
- The unsafe positions are also seen in figure 9.1, $X_u = \{x_1 \in [0.05, 0.1],\ x_2 \in [-0.5, 0.5]\}$.
- The safe set for the reference is $X_0 = \{x_{\text{ref}} \in [-0.1, 0.05 - \delta_{\text{err}}]$ ensuring that $X_0 \subseteq X \setminus X_u$.

Results and Conclusions

The parameters $\bar{\varepsilon}$, Δ, δ_{err}, and the degree of the SOS polynomials q_j and the polynomial $B(\mathbf{x}_{\text{err}})$, are tweaked to find the smallest possible value of δ_{err} yielding a valid solution. The MATLAB implementation of this certificate can be found in appendix G.2.4 and in appendix J under the path `matlab_scripts/sostools/2ndorder_error.m`. The findings conform with the conclusions presented in table 9.3 and table 9.5. The choice of the parameter values is presented in table 9.7 and the barrier certificate is plotted in figure 9.8.

Choice	Reason
$\deg(B) = [0{:}4]$	The residual norm of the solution is growing when the polynomial degrees are increased, thus the smallest degree of $B(\mathbf{x}_{\text{err}})$ yielding a solution is chosen.
$\deg(q_j) = [0{:}1]$	Decreasing the degree of the SOS polynomials decreases the residual norm of the solution, hence the degree is chosen as low as possible.
$\bar{\varepsilon} = 5e\text{-}2$	The value of $\bar{\varepsilon}$ is chosen from the solution yielding the best compromise between small residual norm and feasibility ratio close to 1.
$\Delta = 5e\text{-}3$	The smallest possible value of Δ yielding a solution for the chosen δ_{err}.
$\mathbf{K} = [5.173\ \ 0.214]$	Choice of closed loop system with real poles from equation (4.15).
$\delta_{\text{err}} = 8e\text{-}3$	No solutions could be found for $\delta_{\text{err}} < 8$ mm.

Table 9.7: Chosen value for each of the parameters. The barrier certificate is plotted in figure 9.8.

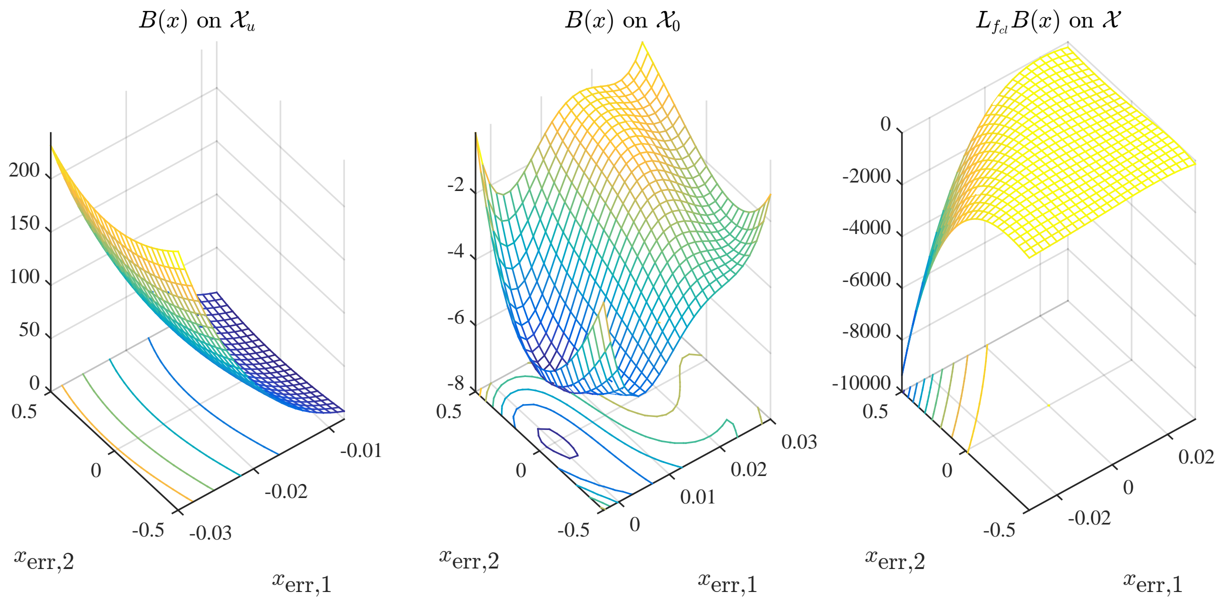


Figure 9.8: Barrier certificate of degree [0:4], all SOS polynomials of degree [0:1], $\bar{\varepsilon}$ = 5e-2, Δ = 5 mm, δ_{err} = 8 mm and gain $\mathbf{K} = [5.173\ \ 0.214]$ gives `feasratio=1.0176` and `Residual norm=3.2e-6`.

It is seen from figure 9.8 that the barrier certificate is positive on the unsafe region for positions below -8 mm, i.e. $\mathcal{X}_u = \{\mathbf{x}_{\text{err}} \in [-0.03, 0.008] \times [-0.5, 0.5]\}$, and nonpositive on the safe region $\mathcal{X}_0 = \{\mathbf{x}_{\text{err}} \in [-0.005, 0.03] \times [-0.5, 0.5]\}$, and that its Lie derivative is nonpositive on the entire set $\mathcal{X} = \{\mathbf{x}_{\text{err}} \in [-0.03, 0.03] \times [-0.5, 0.5]\}$, confirming that it is a valid barrier certificate for the system in equation (9.7) in accordance with Definition 2.2.

Relating the relative position to the absolute positions, this certificate guarantees safety for the system in equation (9.6) for refereces up until a minimum distance of 8 mm from the unsafe positions $\{x_1 \in [0.05, 0.1]\}$ (see figure 9.1), i.e. system safety is guaranteed for references in the interval $\mathcal{X}_0 = \{x_{\text{ref}} \in [-0.1, 0.042]\}$. Comparing this result to the conclusion drawn for the first order system in subsection 9.2.3, it is seen that for the second order system safety is certified for references that are 1 mm closer to the unsafe set. This is attributed coincidence in the combination of tested parameter values, and it is thus expected that safety can be guaranteed for the first order system for references closer to the unsafe set than 9 mm using a different combination of parameter values.

9.4 Conclusion on the Use of SOSTOOLS

To wrap up the approach of using the MATLAB toolbox SOSTOOLS in the construction of barrier certificates presented in this chapter, the following closing considerations are regarded:

- Formulating and setting up the constraints for a barrier certificate has successfully been implemented in SOSTOOLS. An overview has been gained into the evaluation of solutions, and barrier certificates validating safety for first and second order systems have been constructed. Furthermore, a thorough step-by-step guide has been compiled which can be used as a launch pad for future studies at Aalborg University in using the toolbox for system safety validation.
- Safety validation of higher dimension systems can be seen as a natural extension of the tested systems, and it is considered a relatively staightforward task to expand the 1D system to a 3D system, such as the system presented in chapter 6, and test for safety using the same principles for the parameters. It is expected that dynamic barrier certificates such as the one described in chapter 5, can be found with SOSTOOLS, as it has been proven to exist.
- It is deemed an elaborate approach to use SOSTOOLS for constructing barrier certificates for systems of low dimension, such as the ones considered in this chapter, and it is assessed that the manual construction of CBFs presented in chapters 3 through 6 is the most efficient approach for systems of a dimensionality allowing for intuitive visualization of the barrier function. It is, however, also assessed that when physical visualization of the problem is not possible, an approach such as using SOSTOOLS may be the only feasible way of constructing a barrier certificate and validating safety of a system.

This concludes the analysis of barrier certificate search with SOSTOOLS.

Chapter 10

Conclusion and Discussion

This chapter will conclude on the results obtained throughout this thesis and put the solution and entire strategy into perspective in the discussion part.

Conclusion

Safety aspects in robotic surgery and automated robotic surgery are found to be *the* important factor, as analysed in chapter 1. Concurrently, it founds the basic framework in the long term goal of obtaining virtual fixtures. Consequently, a barrier certificate is stated in chapter 2 which modifies and adapts the Lyapunov stability criteria to enable a way to define safe and unsafe regions within the state-space.

A theoretical controller is developed in chapter 3 based on control barrier functions which ensures that the barrier certificate requirements presented in chapter 2 are obeyed at all times. Thus, the control barrier function allows a way to ensure safety in real-time with astounding few calculations.

The control topology presented in chapter 3 is applied to three use cases which intend to commence a solution to the problem of guaranteeing safety in automated surgeries, i.e.:

- A concrete example of the use of control barrier functions is founded in chapter 4. It comprises the instrument slide movement. The system is modelled as both a first and second order system, thereby slowly increasing the complexity of the CBFs, such that necessary experience in the construction of CBFs can be gathered. The result is a successful controller guaranteeing safety by never entering predefined unsafe regions in one dimension for both a first and second order system approximation.
- A safe regulator is designed to ensure system safety in relation to virtual fixtures in chapter 5. A dynamic CBF is constructed in accordance with the desire of virtual fixture and thus founds safety for operation on a beating heart. The result for this use case is that a safe distance between heart and robotic end effector can be set as desired.
- Then safety considerations are extended to the 3D Euclidean space in chapter 6 which implies additional implementation challenges such as a kinematic description (mapped and verified in appendix C), forward kinematics and inverse kinematics. The construction of a CBF is taken to higher dimensions forming a barrier enclosing the interior of an ellipsoid, thus representing a heart or another vital organ fixed in space. The result is a valid CBF ensuring that the robot end effector is kept outside the ellipsoid at all time.

All three use cases are implemented in a simulation environment in MATLAB with convincing results, i.e. system safety is ensured by preventing the system state from entering specified unsafe regions. The controllers are furthermore implemented in C++ in the ROS (Robotic Operating System) framework. The ROS framework is founded in appendix A as a necessary condition to allow any implementation on

the da Vinci robot. All development within ROS is tailored for this project and did not exist at project initiation. The implemented controllers comply with the expected outcome and do indeed behave as desired, i.e. ensuring safety by evading the predefined unsafe regions. Additionally, the implemented controllers are verified to require very little processing power making them ideal as real-time controllers.

The three use cases do, however, consist of simple models where the system order does not exceed 3. An important conclusion is drawn from the use cases, which already could be inferred from the one dimensional safe slide controller (developed in chapter 4) with system order 2. That is, for high order systems where the physical interpretation of the state vector is obscured, the construction of a valid CBF is a highly non-trivial task – if not impossible.

For this reason, the problem is turned upside down in chapter 8, thus no restrictions are put forth in the controller development. Instead, the closed loop system is evaluated and the question is asked whether it complies with the barrier certificate requirement in chapter 2. The verdict is hereafter given as *pass* or *not pass*. For this purpose, chapter 8 presents the global SOS (Sum Of Squares) positivity characteristic and through Putinar's Positivstellensatz recast the barrier certificate formulation as a problem of local positivity, thus allowing sets of unsafe and safe regions to be defined by unrestricted polynomials.

The strategy presented in chapter 8 is applied with the MATLAB toolbox SOSTOOLS in chapter 9 such that barrier certificates can be searched for by automated means. Here, a framework is developed such that the toolbox takes a closed loop system description and a description of the safe and unsafe regions as inputs. The developed framework delivers an unambiguous certificate answering if the system is safe, thus constituting the *pass* and *not pass* verdict. The slide controller developed in chapter 6 is accordingly taken as an example and the framework is verified with this example. Both the first and second order system approximation is analysed in the designed SOSTOOLS framework. It is, as expected, certified to be safe in almost the entire desired range. These examples conclude and verify the use of the developed framework. The framework can easily handle other systems, as the task merely comprises other closed loop system descriptions as input in other dimensions with different safe and unsafe sets. This is a trivial task.

Hence, it can be concluded that the two initially desired strategies comprising the design and analysis of a safe controller are investigated and solved sufficiently to provide a "proof of concept" framework. This applies for both theory, simulation and implementation.

Discussion and Future Work

The developed solution proves itself very efficient in both theory and simulation. However, the implementation aspect suffers from a number of issues which should be investigated in future work. This includes:

- Incorporate integral action in all controllers to eliminate steady state errors.
- Increase the sampling rate from 100 Hz to 2 kHz which indeed is the long term goal. All controllers will draw benefits from this on the transition set $\mathcal{T}$. This may, however, introduce challenges as the allowed execution time (process time between every sample) is lowered to 0.5 ms which is less than the actual execution time in figure 6.10 for the safety controller in the 3D Euclidean space. Therefore, optimization must be performed in the implementation.

- Improvement of the inverse kinematics solver as it occasionally chooses joint angles requiring multiple revolutions around the unit circle to obtain a position which could be reached with an angle less than π.

Additionally, the position controller already implemented on the FPGA (as seen in figure 1.5), is left untouched. It may with removed to draw benefits from a more clear dynamics. This will require another system model, but may well be worth the trouble.

Furthermore, a consistently disregarded topic in this project is the use of trajectory planning. The controllers developed take only small steps as input. However, large step sizes have been given to the controllers in this project to demonstrate certain features, but obviously, it is desired to construct a trajectory planning layer taking the setpoints as input and breaking the path down into a sequence of adjacent points, thus ensuring that small step sizes are given to the controller.

Additionally, at no point the orientation of the robot hand has been considered. Obviously, ensuring safety for the end effector is not sufficient as the heart or other vital organs can be penetrated or crushed by collision with the physical volume of the robotic tool other than the tip of the tool. This is an important topic in future work. Collision avoidance for the robotic parts themselves must also be studied when employing all four of the da Vinci arms in the setup, which is indeed the long term objective.

It is suggested for future use of the framework developed for barrier certificate search with SOSTOOLS to conduct the search in a more methodical manner by running the search like a Monte Carlo simulation, each time varying a parameter while keeping the others fixed. In this way the chance of determining a valid barrier function is maximized, thus indispensably invalidating system safety if no valid certificate can be found.

As explained by assistant nurse Jane Petersson in section 1.3, there are veins, nerves and other organs which must not be cut during a surgery. It has been the aim to construct barrier certificates that can represent these parts of the body. However, it is clear that a realistic barrier certificate representing these parts is far away. Especially because they are time dependent and because, from time to time, the surgeon needs to move these parts to be able to operate in a certain area, thus reshaping these parts. Consequently, a very creative and adaptive barrier function is required and will as a necessary condition require robot vision (a continuous video stream analysis) such that these parts can be tracked. A way to resolve this complex problem of high dimensionality could be a combination of the design approach and analysis approach in the following way:

1. Search for a barrier certificate using the framework developed in chapter 9.
2. Apply this barrier certificate as control barrier function in a similar way as done in chapter 4, chapter 5 and chapter 6.
3. Analyse the situation. Adjust the barrier certificate if necessary and describe the new closed loop system.
4. Take the new closed loop system as input to the framework developed in chapter 9 and start from 1 again.

This iterative approach may along with the preceding listed bullet points get the robotic surgery industry one step closer to the end goal of guaranteed safe robotic surgery with the da Vinci robot, which has been the sole application of the safety controllers derived throughout this project. However, it should take very little imagination to envisage that this way of constructing controllers has the potential to be used in many other industries where safety is critical or simply where regions are desirable to be left untouched.

Bibliography

Abbeel, Goldberg, Kehoe, Kahn, Mahler, Kim, Lee, Lee, Nakagawa, Patil, and Boyd, **2014**. Pieter Abbeel, Ken Goldberg, Ben Kehoe, Gregory Kahn, Jeffrey Mahler, Jonathan Kim, Alex Lee, Anna Lee, Keisuke Nakagawa, Sachin Patil, and W. Douglas Boyd. *Autonomous Multilateral Debridement with the Raven Surgical Robot*. IEEE International Conference on Robotics and Automation, 2014.

Artstein, **1983**. Zvi Artstein. *Stabilization with Relaxed Controls*. Nonlinear Analysis, Methods & Applications, Vol. 7, No. 11, pp. 1163-1167, 1983.

Aylward, Parrilo, and Slotine, **February 2008**. Erin M. Aylward, Pablo A. Parrilo, and Jean-Jacques E. Slotine. *Stability and Robustness Analysis of Nonlinear Systems via Contraction Metrics and SOS Programming*, Massachusetts Institute of Technology, February 2008.

d'Angelo and Putinar, **2009**. John P. d'Angelo and Mihai Putinar. *Polynomial optimization on odd-dimensional spheres*. The IMA Volumes in Mathematics and its Applications, 149, 2009.

Duindam and Sastry, **November 2007**. Vincent Duindam and Shankar Sastry. *Geometric Motion Estimation and Control for Robotic-Assisted Beating-Heart Surgery*. IEEE RSJ International Conference on Intelligent Robots and Systems, pages 871–876, 2007.

Franklin, Powel, and Emami-Naeini, **2010**. Gene F. Franklin, J. David Powel, and Abbas Emami-Naeini. *Feedback Control of Dynamic Systems*. Pearson, 2010. ISBN 978-0-13-500150-9.

Galeota-Sprung, Juárez, Kamlet, nas, Rothchild, and Ziperstein, **2004**. Benjamin Galeota-Sprung, Elisa Juárez, Marti Kamlet, Ana Mascare nas, Alissa Rothchild, and Joshua Ziperstein. *History of Robotic Surgery*. Brown University website, 2004. URL `http://biomed.brown.edu/Courses/BI108/BI108_2004_Groups/Group02/Group%2002%20Website/`.

Hannaford and Rosen, **2006**. Blake Hannaford and Jacob Rosen. *Doc at a distance*. IEEE Spectrum, 2006.

Hannaford, Rosen, Friedman, King, Glozman, Ma, Kosari, and White, **April 2013**. Blake Hannaford, Jacob Rosen, Diana W. Friedman, Hawkeye King, Daniel Glozman, Ji Ma, Sina Nia Kosari, and Lee White. *Raven-II: An Open Platform for Surgical Robotics Research*. IEEE Transactions on Biomedical Engineering, 60(4), 954–959, 2013.

Hatzinger, Kwon, Langbein, Kamp, Häcker, and Alken, **November 2006**. Martin Hatzinger, S.T. Kwon, S. Langbein, S. Kamp, Axel Häcker, and Peter Alken. *Hans Christian Jacobaeus: Inventor of Human Laparoscopy and Thoracoscopy*. Journal of Endourology, 20(11), 848–850, 2006.

Hoffman, 2010. Alan N. Hoffman. *Intuitive Surgical, Inc.: How Long Can Their Monopoly Last?* RSM Case Development Centre, 2010. Reference no. 310-106-1.

Horowitz, 2014a. Roberto Horowitz. *ME 232 Advanced Control Systems I Lecture 17 - State Variable Feedback Control*. University of California, Berkeley, 2014.

Horowitz, 2014b. Roberto Horowitz. *ME 232 Advanced Control Systems I Lecture 8 - Discrete Time Models from Sampling Continuous Time Models*. University of California, Berkeley, 2014.

Lasserre, 2009. Jean-Bernard Lasserre. *Moments, positive polynomials and their applications*. Imperial College Press optimization series. Imperial College Press, 2009.

Laurent, 2009. Monique Laurent. *Sums of squares, moment matrices and optimization over polynomials*. The IMA Volumes in Mathematics and its Applications, 149, 2009.

Maciejowski, 2002. J.M. Maciejowski. *Predictive Control with Constraints*. 2002. ISBN 0-201-39823-0.

Murray, Li, and Sastry, 1994. Richard M. Murray, Zexiang Li, and S. Shankar Sastry. *A Mathematical Introduction to Robotic Manipulation*. CRC Press, 1994. ISBN 9780849379819.

Novak, March 2012. Matt Novak. *Telemedicine predicted in 1925*. Smithsonian, 2012.

Open Source Robotic Foundation, 2015a. Open Source Robotic Foundation. *catkin/CMakeLists.txt*. ROS.org, 2015. URL http://wiki.ros.org/catkin/CMakeLists.txt.

Open Source Robotic Foundation, 2015b. Open Source Robotic Foundation. *catkin/package.xml*. ROS.org, 2015. URL http://wiki.ros.org/catkin/package.xml.

Orocos, 2015. Orocos. *Orocos Kinematics and Dynamics - Smarter in motion!* 2015. URL http://www.orocos.org/kdl.

Antonis Papachristodoulou, James Anderson, Giorgio Valmorbida, Stephen Prajna, Peter Seiler, and Pablo A. Parrilo, October 2013. Antonis Papachristodoulou, James Anderson, Giorgio Valmorbida, Stephen Prajna, Peter Seiler, and Pablo A. Parrilo. *SOSTOOLS - Sum of Squares Optimization Toolbox for MATLAB*, 3 edition, 2013.

Parrilo, April 2003. Pablo Parrilo. *Semidefinite programming relaxations for semialgebraic problems*. Springer-Verlag, 2003.

Peters, 2013. Walter R. Peters. *Minimally-invasive Surgery*. web, 2013. URL http://www.fascrs.org/patients/disease-condition/minimally-invasive-surgery-expanded-version.

Prajna, Jadbabaie, and Pappas, August 2007. S. Prajna, A. Jadbabaie, and G. J. Pappas. *A framework for worstcase and stochastic safety verification using barrier certificates*. IEEE Transactions on Automatic Control, 52(8), pp. 1415–1428, 2007.

Quigley, Gerkey, Conley, Faust, Foote, Leibs, Berger, Wheeler, and Ng, 2009. Morgan Quigley, Brian Gerkey, Ken Conley, Josh Faust, Tully Foote, Jeremy Leibs, Eric Berger, Rob Wheeler, and Andrew Ng. *ROS: an open-source Robot Operating System*. 2009.

Satava, Yoo, Gilbert, Broderick, Garcia, Doarn, and Moses, **2011**. Richard M. Satava, Andrew C. Yoo, Gary R. Gilbert, Timothy J. Broderick, Pablo Garcia, Charles R. Doarn, and Gerald R. Moses. *Surgical Robotics: Systems Applications and Visions*. Springer, 2011.

Sloth and Wisniewski, **2014**. Christoffer Sloth and Rafael Wisniewski. *Towards Safe Robotic Surgical Systems*. Automation and Control, Department of Electronic Systems, Aalborg University, 2014.

Sloth, Wisniewski, Larsen, Leth, and Poulsen, **2012**. Christoffer Sloth, Rafael Wisniewski, Jesper Larsen, John Leth, and Johan Poulsen. *Model of Beating-Heart and Surgical Robot*. 2012.

Stoustrup, **2014**. Jakob Stoustrup. *State Space Methods - Lecture 5: introducing reference signals, anti-windup, optimal control*. Aalborg University, 2014.

Sucan and Chitta, **2013**. Ioan A. Sucan and Sachin Chitta. *PR2/Setup Assistant/Quick Start*. MoveIt, 2013. URL `http://moveit.ros.org/wiki/PR2/Setup_Assistant/Quick_Start#STEP_3:_Add_Virtual_Joints`.

Wall and Marescaux, **2013**. James Wall and Jacques Marescaux. *Telemicrosurgery*. Springer, 2013.

Wang and Boyd, **2010**. Yang Wang and Stephen Boyd. *Fast Model Predictive Control Using Online Optimization*. IIEE Transactions on Control Systems Technology, Vol 18, No 2, March 2010, 2010.

Wieland and Allgöwer, **2007**. Peter Wieland and Frank Allgöwer. *Constructive Safety Using Control Barrier Functions*. The 7th IFAC Symposium on Nonlinear Control Systems 21-24 August, 2007, Pretoria, South Africa, 2007.

Wisniewski, Sloth, Jensen, and Hansen, **2015**. Rafael Wisniewski, Christoffer Sloth, Simon Jensen, and Karl Damkjær Hansen. *Instrumentation of the da Vinci Robotic Surgical System*. 2015.

Appendix A

Interfacing da Vinci with ROS

This appendix contains:

- An installation guide and an introduction immediately after this itemize.
- A description of the general structure of ROS in section A.1.
- A description of how to initiate ROS and how to setup the low level controllers in section A.2.
- An overview of the final developed framework and how to run it in section A.3.
- An explanation of why the Moveit package is *not* used in section A.4.

Accordingly, this appendix ought to give concrete knowledge to utilize the ROS environment wrt. the da Vinci surgery robot at Aalborg University as it comprises an immense load of files, packages and various GUI interfaces. It also intends to provide an overview of the code structure and the underlying thoughts. The ROS environment is currently only developed for Ubuntu. The content of this appendix is accordingly assuming Ubuntu as operating system and assumes additionally basic knowledge in Unix.

To install ROS on a private laptop, it is recommended to follow the below URL:

http://wiki.ros.org/ROS/Installation

Once ROS is installed, it is possible to work on the `surgery-srv.lab.es.aau.dk` computer. It may introduce some advantages to work directly on the server in the lab as it provides additional GUI applications such as rviz, but it is obviously more convenient to work from a private laptop and the GUIs can be set-up without too much trouble locally. Connection to the server can be established through `ssh`:

```
$ ssh <user>@surgery-srv.lab.es.aau.dk
```

To get started with everything, open a terminal and initialize a ROS workspace as:

```
$ mkdir -p daVinci_ws/src
```

Then navigate to the source directory (`src`) and type:

```
$ catkin_init_workspace
```

This creates a number of necessary files and folders. The code located at the "Robotic Surgery Group - Aalborg University" can be copied/cloned to the `src` folder. The original environment (clean configuration) can e.g. be cloned with the following git terminal commands:

```
$ git clone https://github.com/AalborgUniversity-RoboticSurgeryGroup/davinci_description
$ git clone https://github.com/AalborgUniversity-RoboticSurgeryGroup/davinci_driver
```

Each command clones a package. The name and file structure of a package should follow a certain standard, i.e. the ROS Enhancement Proposals (REP) (it is not just the packages that should follow the REP standard, but in fact the entire ROS workspace). This ought to make it easier to share and reuse code. The code developed in this thesis obeys to a large extend the REPs but exceptions may occur.

The two initial packages used are in that sense:

- `davinci_description`
- `davinci_driver`

If the installation takes place on a local laptop, be sure to install all ROS dependencies. The necessary dependencies can be checked by running `rosdep check <package_name>`.

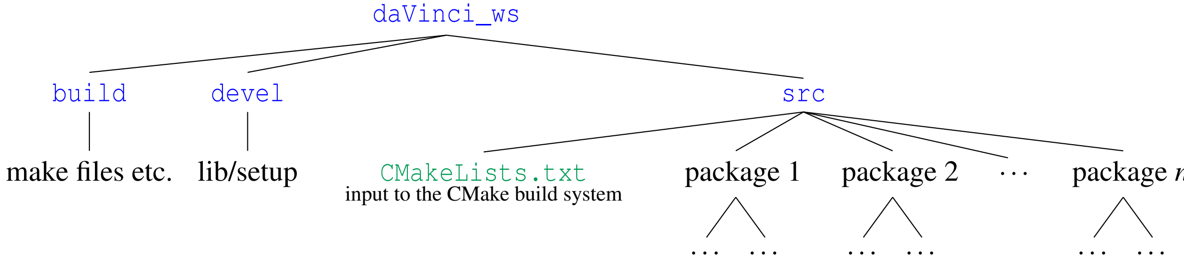
To build the entire environment, open a terminal, navigate to the root of the workspace (`daVinci_ws/`) and type:

```
$ catkin_make
```

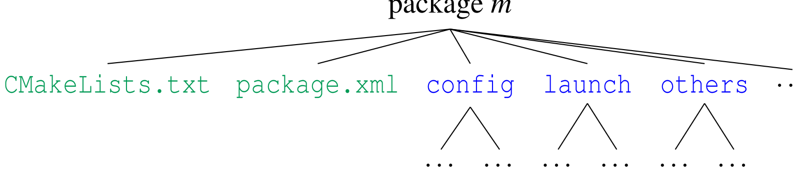
This connects all executables and the environment should hereafter be ready for use.

A.1 General structure of a ROS setup

After the workspace is created (called `daVinci_ws`), the packages are cloned and the environment is build, the overall code structure should look like the tree structure found below:



Each package has a similar structure. While the content of each package may vary, they always have a file called `package.xml` and `CMakeLists.txt`, and often the structure shown below.



Before elaborating on the significance of these folders and files, it is to some extend important to have an overview of the general used terms in the ROS environment. Those terms are briefly mentioned in figure A.1.

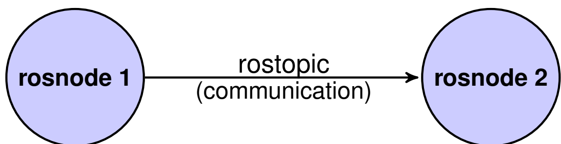


Figure A.1: Coherence between rosnodes and rostopics. A node is simply a process that performs some computation and a topic is the communication channel between two or more ROS nodes. Two often used terms in this context are to publish/subscribe to a topic. To "publish" means to send a message from a topic and one can decode the message by "subscribing" to a topic.

With a basic understanding of ROS nodes and topics, the generic content of the two required files (`CMakeLists.txt` and `package.xml`) and the often used `launch` folder can be elaborated in table A.1. Other folders and files like `src`, `config`, `include` and similar are indeed also often used. They all have the purpose to enhance overview. The name should to some extend be self explaining, e.g. the `config` folder includes configuration files for the da Vinci robot, the `src` folder often includes C++ files used for algorithms designed for specific purposes etc.

CMakeLists.txt	**package.xml**	**launch**
Package/project description, catkin version, specification of required packages (not ROS packages but packages to create CMake environment variables), catkin dependencies and definitions and the specification of catkin build targets (executables and library targets). *	It provides information about the maintainer, version, package name (e.g. `davinci_driver`) and author. It specifies build tool dependencies (for the package to build itself - typically only catkin), build dependencies (required packages at build time), run-time dependencies and test dependencies (not used). **	The content of a launch folder is primary used to start a group of nodes with unique topics and/or parameters. They are executed by the `roslaunch` terminal command followed by package name and lastly the name of the launch file, i.e.: `roslaunch <package name> <name of launch file>`.

Table A.1: Brief explanation of the purpose of the most common used folder names in a package.
* [Open Source Robotic Foundation, 2015a], ** [Open Source Robotic Foundation, 2015b].

With a somewhat superficial, but sufficient, introduction to ROS, the concrete interfacing can be considered.

A.2 Setup of Low Level Control and how to Initiate ROS

Before the communication between ROS and da Vinci may be considered, all low level PID controllers must run correctly and the RIO configuration must be performed.

From the `aau86730` computer, launch the `p4_primary_Control` icon located on the desktop and connect `RT Single Board RIO (172.26.12.32)` by right clicking the icon and press connect. Subsequently, navigate to `p4_prim_control_FPGA_multichannel_7_FLOAT_SPI_5.vi` and open it. This launch a GUI comprising access to the seven low level controllers which are activated from the arrow in the upper left corner. The controller gains, setpoints, maximum step size and various calibration options are easily accessible from this GUI, though it should not be necessary to modify any of those.

Be sure that the gearing factors are specified as follows:

Intrument Jaw Left	**Intrument Jaw Right**	**Intrument Pitch**	**Instrument Roll**	**Instrument Slide**	**Hand Pitch**	**Hand Roll**
12	12	12.4	7.5	1340	200	200

Table A.2: Measured gearing factors. Gearing factors are measured such that $\pi/4$ from ROS corresponds to 45 degrees on the real robot.

To allow the ROS environment access to the full range of setpoints, launch `p4-control_prim-main4.vi` and activate this GUI in a similar manner. This GUI acts merely as interface and offers no user options

as such. All necessary setup before initiating ROS is at this point in time performed.

ROS

It is important to notice that every time a new terminal is commenced it is important to source the bash file from the workspace, i.e.

```
$ source devel/setup.bash
```

The following list of commands must be executed from the root of the workspace. It is first of all important to collect all ROS nodes such that they are able to communicate with each other. Open a terminal and run:

1. `$ roscore` *# Leave this running in the terminal*

Now, to secure the TCP/IP connection between ROS and the RIO board (Rx & Tx of setpoints), launch the driver from a new terminal:

2. `$ roslaunch davinci_driver davinci_driver.launch` *# Leave this running*

With these files processes running, the environment is proper set up. The remainder of this appendix will first elaborate what the moveit Application Programmable Interface (API) consist of, why it could be a good idea to use it and why it essentially is not used.

A.3 Specific Structure of this Thesis - The gr1032 Development Branch

Be sure that ROS is installed according to the beginning of this appendix but no packages should be cloned. Additionally, the low level controllers and ROS must be initiated as described in section A.2. The `gr1032` package can be cloned from github as:

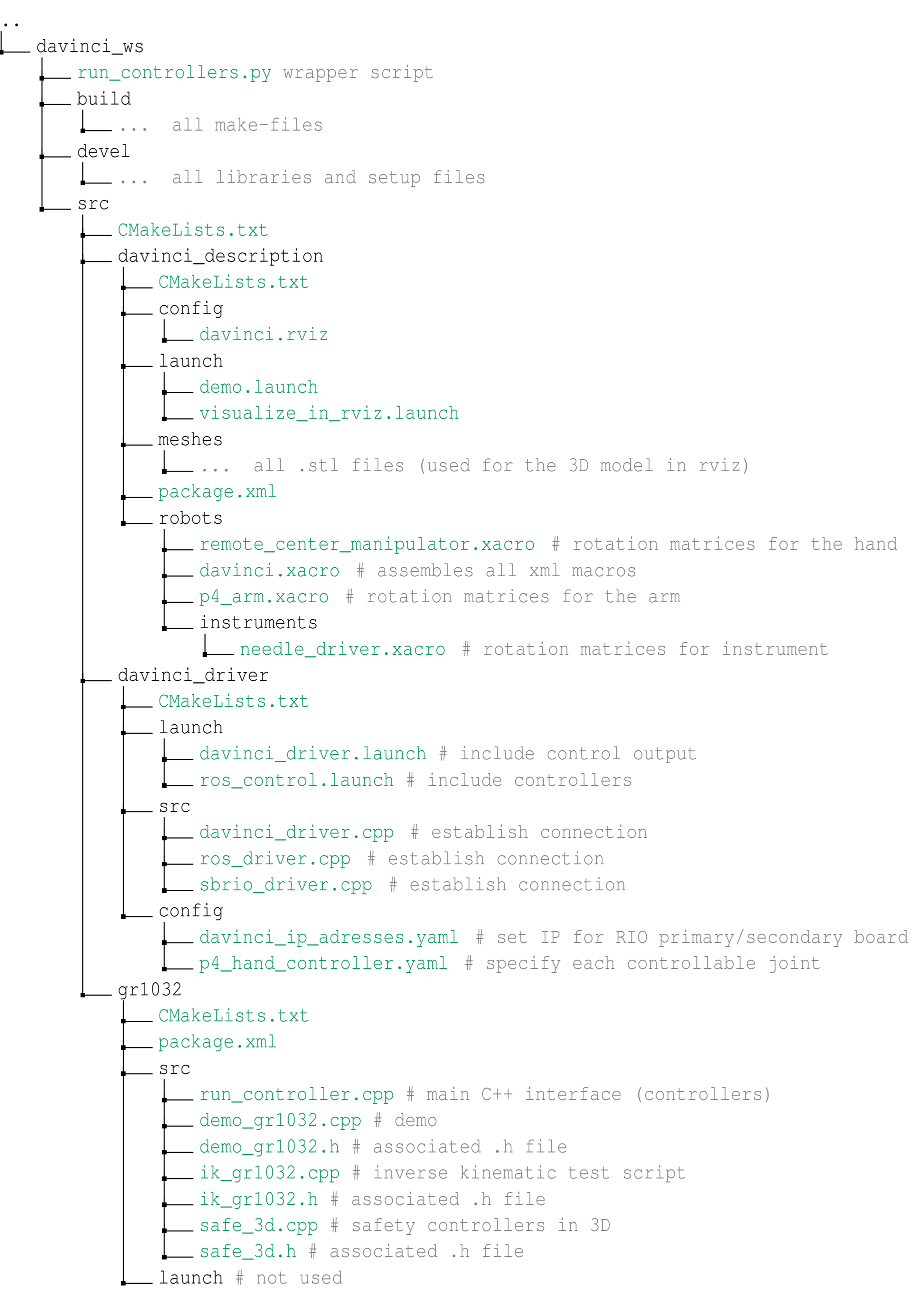
```
git clone https://github.com/AalborgUniversity-RoboticSurgeryGroup/Gr1032
```

If the original `davinci_description` and `davinci_driver` package are cloned to the workspace, delete them. It is important that the `gr1032` branch is cloned from the driver package and the description package as:

```
git clone https://github.com/AalborgUniversity-RoboticSurgeryGroup/davinci_driver
--branch gr1032
```

```
git clone https://github.com/AalborgUniversity-RoboticSurgeryGroup/davinci_description
--branch reduced_robot
```

The file structure in the `daVinci` workspace should be similar to the one depicted below (plus additional files).



The python wrapper script `run_controllers.py` located in the root of the workspace is not located in any package and it must be created apart from the ROS framework. It can be copied from appendix I or from appendix J.

The controllers are executed by running the two commands below in two individual terminals:

- `roslaunch davinci_driver davinci_driver.launch`
- `python run_controllers.py`

The launch file launches all necessary drivers to interface with the da Vinci robot and the wrapper scripts ensures that the control signal is published on the appropriate topics and that the main C++ file `run_controllers.cpp` is executed with proper ROS syntax.

After running these commands, a Graphical user interface (GUI) appears as shown below:

```
*****************************************************
The following options are avaiable:
-----------------------------------------------------
press 'a' to run slide safety controller
press 'b' to specify custom joint angles (FK mode)
press 'c' to run demo
press 'd' to run beating heart controller
press 'e' to specify custom 3D angles (IK mode)
press 'f' to run 3D safety controller
*****************************************************
```

The desired controller is now ready to be executed by entering the inquired letters. The `gr1032` package provides also demo's with the exclusive purpose of demonstrating the capabilities of the da Vinci robot. The file and code structure is not as such described deeper in this appendix. The algorithms associated with each controller is described under each appertaining chapter in the main report.

A.4 Structure of Moveit and Why it is Not Used

Moveit is an interface to standard robots. The use of Moveit ought to ease trajectory planning and to ease interfacing with da Vinci. The moveit package includes the very handy `move_group` node which searches for a Unified Robot Description Format (URDF) which is a description of the robot (containing parameters like joint limits, kinematics etc.) and it searches for a Semantic Robot Description Format (SRDF) which is a parameter generated by the setup assistant (elaborated in subsection A.4.1), thus representing parameters not in the URDF. This could be group state configurations or alike.

In that sense the `move_group` node offers a (when the ROS learning curve is conquered) sorely easy user interface from both python, C++ and a GUI. It is indeed an apparent starting interface to use, and the first successful interface to the robot was indeed established through the `move_group` node and for that reason described in this appendix. As is shall be seen, `move_group` has some disadvantages when the objective is a real-time safety controller, which mostly consist of:

- Speed. The `move_group` node offers many features, including static obstacle avoidance and trajectory planning. All very useful applications, but they slow down the process and proofs itself useless when the controllers developed in chapter 4 and chapter 5 are to be implemented.
- The `move_group` node already includes controllers hence shattering the dynamics modelled. The safety controllers developed are at a lower abstraction layer.

A significant amount of code is developed with the `move_group` node. The low level details will, however, not be elaborated. However, the results of the work undertaken with the Moveit package, can be cloned as the development branch at github:

```
$ git clone https://github.com/AalborgUniversity-RoboticSurgeryGroup/
davinci_moveit_config --branch develop   # Clone driver, description and Moveit package from development branch
```

To give an overview of the code structure when the Moveit package is used, the directory tree on the

following page is provided. It shows merely the "interesting files" seen from a developers point of view. In reality, additionally files are present.

To run the code, be sure that step **1** and **2** in section A.2 is carried through. Thus, it is possible to allow trajectory planning, by linking the OMPL (Open Motion Planning Library) to the system by running:

3.a `$ roslaunch davinci_moveit_config move_group.launch` *# Leave this running*

If a 3D GUI interface is desired, open a new terminal and launch:

3.b `$ roslaunch davinci_bringup visualization.launch` *# This opens rviz*

Press the "add" button in `rviz` and add the "MotionPlanning" option to the panel where start and goal state can be specified. Hereafter, plan and execute the specified goal. This cause the arm of da Vinci to reach out for the specified goal state consisting of five joint angles.

To launch the developed C++ interface, which allows 3D setpoints (by the KDL inverse kinematic solver) and custom joint specification, open a terminal and type:

4 `$ rosrun davinci_moveit_config MoveGroupInterfaceExecute`

This executes a GUI where a IK test and FK test can be executed. Both of them ask the slide position to move to the position 0.005 m and move back to 0.00 cm immediately. Thus no delay between the two queries are desired. The position is concurrently recorded (the recording can be done similar to the setup described in appendix F). Thus the trajectories are plotted in figure A.2.

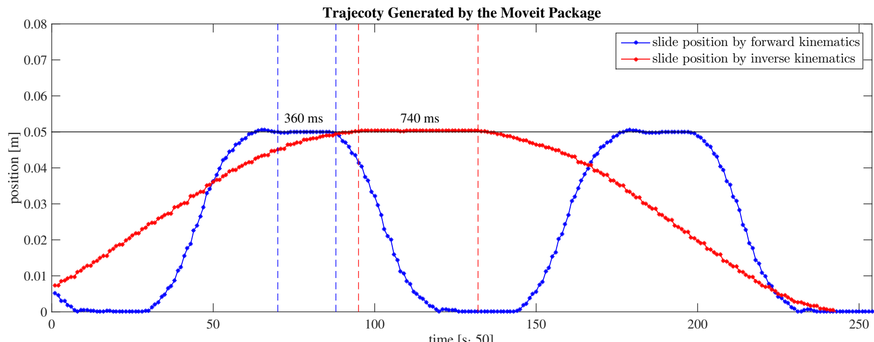


Figure A.2: Trajectories plotted by means of Moveit. It is seen that the processing time is high and nearly useless when the objective differs from trajectory planning. The code used to generate the trajectories is showed in figure A.3. It is also seen that the dynamics are limited. Measurement files and plotting details can be found in appendix J by running `run_moveit_trajectory.m` under the path `measurements/moveit_test`.

As seen from figure A.2, the `move_group` node requires processing time to calculate a trajectory which is nearly useless for real-time controllers. The processing time is due to the highly advanced trajectory generation calculated by the `move_group` node. An example of how to use of the `move_group` node is provided in figure A.3. The code snippet are used to generated the trajectories in figure A.2. It is important to include the proper Moveit libraries and structure of `CMakeLists.txt` and `package.xml` correctly. These are found at GitHub when the development `moveit` development branch is cloned. The Setup Assistant is important for the moveit package, thus mentioned in the coming subsection.

```
1  while(1) {
2      ROS_INFO("set test angles");
3      joints [p[2]] = 0;   // hand roll
4      joints [p[1]] = 0;  // hand pitch
5      joints [p[0]] = 0.0;  // slide
6      joints [p[3]] = 0;  // inst roll
7      joints [p[4]] = 0;  // inst pitch
8      joints [p[5]] = 0;  // jaw right
9      group.setJointValueTarget(joints);
10     group.move();
11
12     ROS_INFO("set test angles");
13     joints [p[2]] = 0;   // hand roll
14     joints [p[1]] = 0;  // hand pitch
15     joints [p[0]] = 0.005;  // slide
16     joints [p[3]] = 0;  // inst roll
17     joints [p[4]] = 0;  // inst pitch
18     joints [p[5]] = 0;  // jaw right
19     group.setJointValueTarget(joints);
20     group.move();
21 }
```

```
1  while(1) {
2      geometry_msgs::Pose target_pose3;
3      target_pose3.position.x = 0.000000 + off_x;
4      target_pose3.position.y = 0.000000 + off_y;
5      target_pose3.position.z = 0.000000 + off_z;
6      group.setPoseTarget(target_pose3);
7      moveit::planning_interface::MoveGroup::Plan my_plan_3;
8      bool success_3 = group.plan(my_plan_3);
9      ROS_INFO("success = %d", success_3);
10     group.move();
11
12     geometry_msgs::Pose target_pose4;
13     target_pose4.position.x = 0.000000 + off_x;
14     target_pose4.position.y = 0.000000 + off_y;
15     target_pose4.position.z = 0.000000 + off_z;
16     group.setPoseTarget(target_pose4);
17     moveit::planning_interface::MoveGroup::Plan my_plan_4;
18     bool success_4 = group.plan(my_plan_4);
19     ROS_INFO("success = %d", success_4);
20     group.move();
21 }
```

Figure A.3: The code snippet to the left shows how to use forward kinematics with the `move_group` node (`p` is a string array containing the six joints). The snippet to the right shows how to use the inverse kinematics solver with the `move_group` node.

A.4.1 Setup Assistant Associated with MoveGroup

To run the setup assistant, open a terminal, navigate to the root of the workspace and type:

- `$ roslaunch davinci_moveit_config setup_assistant.launch` *# GUI is launched*

A GUI offering eight setup options will now be present. Load the current `davinci_moveit_config` package. The content of the eight options will be explained in the below itemize as it is important that all options are configured correctly for the kinematic solver to work correctly.

1. **Start:** It is possible to specify a new configuration package. This should only be necessary to do once. Since the `davinci_moveit_config` package is cloned from the development branch, it is sufficient to edit the existing package by pressing the associated button while the path to `davinci_moveit_config` is specified correctly. This list is auto-generated from the associated xacro files specified in the `davinci_ description` package (from where a URDF file is generated and initially fed to the setup assistant). The default mode of operation disable collisions between adjacent links, links that can not physically collide, links that are always in collision and links that are in collision in the start-up mode. This enhances processing time [Sucan and Chitta, 2013].

2. **Virtual Joints:** It is here the robot is attached to the physical world by use of a virtual frame.

Virtual Joint Name	Child Link	Parent Frame	Type
`virtual_joint`	`base_link`	`world`	`fixed`

3. **Planning Groups:** It is from here possible to describe the joints of the `p4_arm` of da Vinci. The Orocos KDL kinematic solver seems to be dependent of at least six DOF (six active joints). It is possible to describe the arm by means of either joints, links or as a chain. It is chosen to describe the arm as joints. Be sure that a group `"gripper"` is added with the following kinematic specifications:

- Kinematic Solver: `kdl_kinematic_plugin/KDLKinematicPlugin`
- Kin. Search Resolution: 0.005 (default)
- Kin. Search Timeout (sec): 0.005 (default)
- Kin. Solver Attempts: 3 (default)

It is furthermore important that it has the following joints specified:

```
gripper
├── joints
│   ├── p4_instrument_slide - Prismatic
│   ├── p4_instrument_roll - Revolute
│   ├── p4_hand_pitch - Revolute
│   ├── p4_hand_roll - Revolute
│   ├── p4_rcm_instrument_holder_upper_bar_joint - Revolute
│   ├── p4_rcm_upper_bar_base_joint - Revolute
│   └── p4_instrument_jaw_right - Revolute
├── Links       # Leave this empty
├── Chain       # Leave this empty
└── Subgroups  # Leave this empty
```

This ensures that the group `gripper` can operate with six DOF.

4. **Robot Poses:** It is from here possible to specify standard positions for the arm. The code developed during this thesis utilized a pose for an initial positions, hence be sure that a pose named `ready` is present under the group `gripper`. All joint states should be set to zero for this pose.

5. **End Effectors:** The end effector is specified as shown:

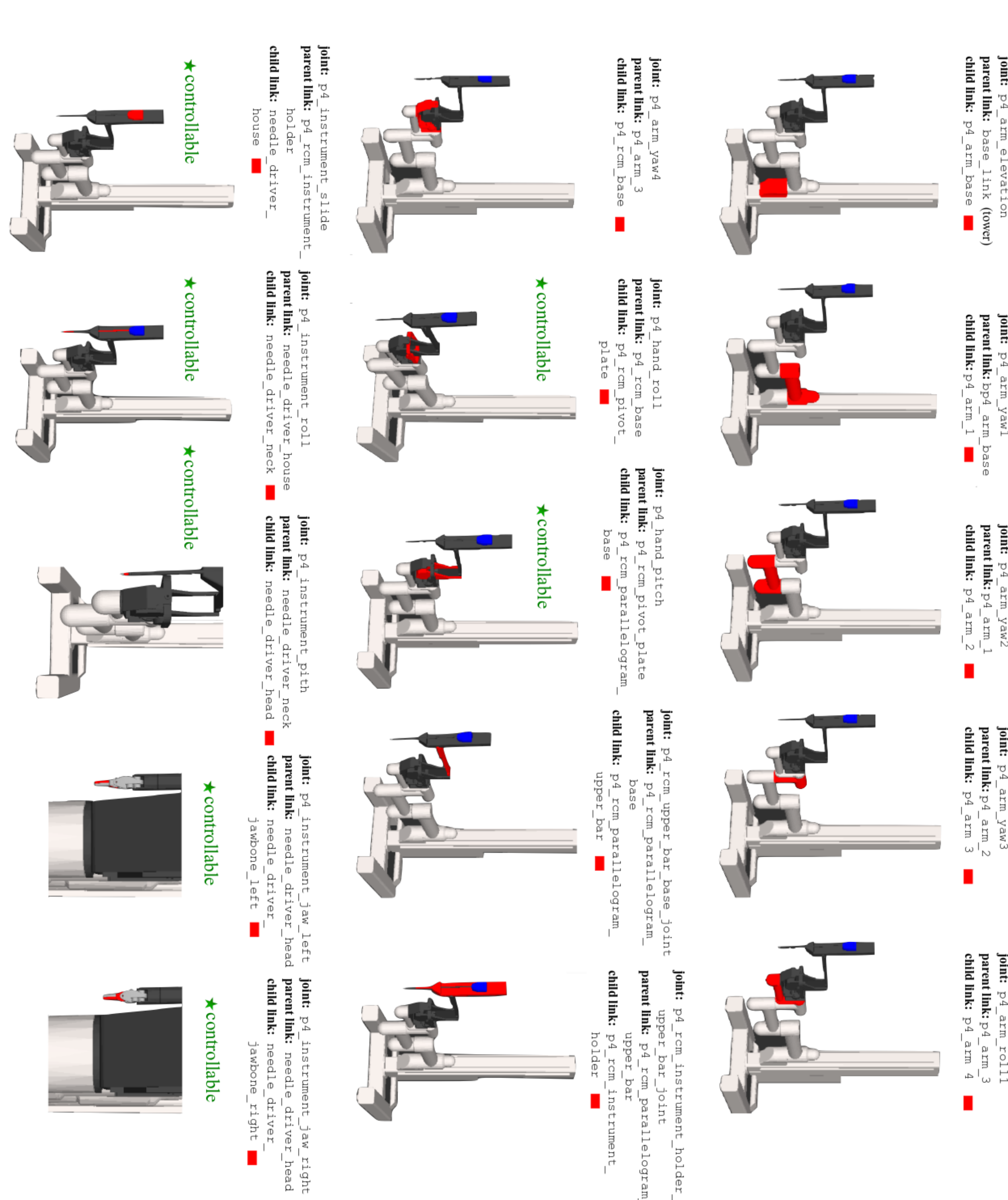
End Effector Name	Group Name	Parent Link	Parent Group
Gripper	gripper	base_link	-leave this empty-

6. **Passive Joints:** A list of all joints will be available. It is important to specify the passive joints such that the `davinci_moveit_config` package know which joints are controllable, i.e. mark all passive joints in the presented table.

7. **Configuration Files:** The package will be generated from here. Be sure that `p4_hand_controller.yaml`, `davinci_moveit_controller_manager.launch`, `controllers.yaml`, `CMakeLists.txt` and `package.xml` are not erased by the setup assistant.

In conclusion, the `moveit` package is too slow, and the applied file structure is accordingly different.

Appendix B
Links and Joints 3D Overview



Appendix C

Kinematic Models of the Robot

A rotation matrix ${}^a_b\mathbf{R}$ is an orthonormal matrix ($\mathbf{R}^{-1} = \mathbf{R}^T$) describing the rotation between two right-handed coordinate frames Ψ_a and Ψ_b such that any vector ${}^b\mathbf{v}$ (including Ψ_b coordinate axes) given in the Ψ_b frame can be "rotated" into Ψ_a coordinates by the operation

$$ {}^a v = {}^a_b\mathbf{R}\, {}^b v \tag{C.1} $$

Note that the matrix ${}^a_b\mathbf{R}$ can also be seen as the rotation required of the frame Ψ_a for it to coincide with Ψ_b. Rotation of the frame Ψ_a with an angle θ counterclockwise about a single axis (equal to clockwise "rotation" of the any vector in Ψ_b) correspond to the rotation matrices

$$ {}^a_b\mathbf{R}_x(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix} \quad {}^a_b\mathbf{R}_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix} \quad {}^a_b\mathbf{R}_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{C.2} $$

A sequence of rotations, transforming the vector ${}^c v$ given in the Ψ_c frame to Ψ_a coordinates, is implemented as

$$ {}^a\mathbf{v} = \underbrace{{}^a_b\mathbf{R}\, {}^b_c\mathbf{R}}_{{}^a_c\mathbf{R}}\, {}^c\mathbf{v} \tag{C.3} $$

The translation of the origin from the coordinate system Ψ_a to Ψ_b can be described by the position vector ${}^a_b\mathbf{p}$, which is a vector given in the Ψ_a coordinate frame. The relative configuration of two coordinate frames is their relative position and orientation, which can be expressed expressed by a homogeneous transformation matrix

$$ {}^a_b\mathbf{T} = \begin{bmatrix} {}^a_b\mathbf{R} & {}^a_b\mathbf{p} \\ 0 & 1 \end{bmatrix} \tag{C.4} $$

The inverse of a configuration matrix is

$$ {}^a_b\mathbf{T}^{-1} = \begin{bmatrix} {}^a_b\mathbf{R}^T & -{}^a_b\mathbf{R}^T\, {}^a_b\mathbf{p} \\ 0 & 1 \end{bmatrix} \tag{C.5} $$

A sequence of configurations is implemented as

$$ {}^a_n\mathbf{T} = {}^a_b\mathbf{T}\, {}^b_c\mathbf{T} \ldots {}^m_n\mathbf{T} = \begin{bmatrix} {}^a_b\mathbf{R}\, {}^b_c\mathbf{R} \ldots {}^l_m\mathbf{R}\, {}^m_n\mathbf{R} & {}^a_b\mathbf{p} + {}^a_b\mathbf{R}\, {}^b_c\mathbf{p} + \ldots + ({}^a_b\mathbf{R}\, {}^b_c\mathbf{R} \ldots {}^l_m\mathbf{R}\, {}^m_n\mathbf{p}) \\ 0 & 1 \end{bmatrix} \tag{C.6} $$

where each matrix $\mathbf{T}$ is a function of a rotation angle θ and a translation distance, which may be functions of time.

C.1 Existing Kinematics for the AAU da Vinci Robot

The position of the end effector (the tip of the instrument) given in an inertial frame can be described as a sequence of joint rotations of the robot and the instrument, and translation from the inertial origin via the fixed-length links and the slide of the instrument.

A coordinate frame is defined for each degree of freedom, with origin on the axis of rotation. A set of coordinate frames and transformation matrices between the frames are given according to the ROS `xacro` files `tower`, `p4_arm`, `remote_center_manipulator` and `needle_driver`.

The position and orientation of the ith coordinate frame is given as a transformation matrix from the $i-1$th frame, where fixed distances and rotations are measured along/about the axes of the $i-1$th frame while free distances and rotations are measured along/about the axes of the ith frame. The parameters and variables shown in figure C.1 are given in table C.1.

frame	a [m]	b [m]	d [m]	fixed rot. α [rad]	free rot. θ [rad]	name
1	0	0	d_1^*	I	I	`elevation`
2	0.186	0	0.554	$\mathbf{R}_z(\pi/2)\mathbf{R}_x(\pi)$	$\mathbf{R}_z(\theta_2^*)$	`arm_yaw1`
3	0	0.583	0	$\mathbf{R}_z(\pi/2)\mathbf{R}_x(-\pi)$	$\mathbf{R}_z(-\theta_3^*)$	`arm_yaw2`
4	0.479	0	-0.001	I	$\mathbf{R}_z(-\theta_4^*)$	`arm_yaw3`
5	0.057	0	0.198	I	$\mathbf{R}_x(-\theta_5^*)$	`arm_roll1`
6	0.352	0	-0.117	I	$\mathbf{R}_z(\theta_6^*)$	`arm_yaw4`
7	-0.042	0	0.161	I	$\mathbf{R}_x(\theta_7^*)$	`hand_roll`
8	0	0	0	$\mathbf{R}_y(-0.288)$	$\mathbf{R}_y(-\theta_8^*)$	`hand_pitch`
9	0.011	0	0.186	$\mathbf{R}_y(0.288)\mathbf{R}_x(\pi)$	$\mathbf{R}_y(-\theta_8)$	`upper_bar`
10	0.520	0	0	$\mathbf{R}_x(\pi)$	$\mathbf{R}_y(-\theta_8)$	`instrument_holder`
11	0	0	-0.120 + d_{11}^*	I	I	`instrument_slide`
12	0.052	0	0	$\mathbf{R}_z(\pi/2)\mathbf{R}_x(\pi)$	$\mathbf{R}_z(\theta_{12}^*)$	`instrument_roll`
13	0	0	0.177	I	$\mathbf{R}_x(-\theta_{13}^*)$	`instrument_pitch`
14L	0	0	0.009	$\mathbf{R}_y(\pi/2)\mathbf{R}_x(\pi/2)$	$\mathbf{R}_z(-\theta_{14L}^*)$	`instrument_jaw_left`
14**R**	0	0	0.009	$\mathbf{R}_y(\pi/2)\mathbf{R}_x(\pi/2)$	$\mathbf{R}_z(\theta_{14\mathbf{R}}^*)$	`instrument_jaw_right`

Table C.1: Variables (marked with *) and parameters for the robot in figure C.1. I is the identity matrix (no rotation). Recent measures indicate that $d_2 = 0.812$, $a_2 = 0.198$, $a_4 = 0.435$, $\alpha_8 = \mathbf{R}_y(-0.07)$, $\alpha_9 = \mathbf{R}_y(0.07)\mathbf{R}_x(\pi)$, $a_9 = 0$ and $d_{11,\text{fixed}} = 0.188$ ($\Rightarrow d_{12} = 0.472$).

I.e. according to table C.1, the transformation between frame 1 and 2 is given as:

$$
{}^1_2\mathbf{T} = \begin{bmatrix} \mathbf{R}_z(\pi/2)\mathbf{R}_x(\pi)\mathbf{R}_z(\theta_2^*) & \mathbf{p}_2 \\ 0 & 1 \end{bmatrix}, \qquad \mathbf{p}_2 = [0.186 \quad 0 \quad 0.554]^T \tag{C.7}
$$

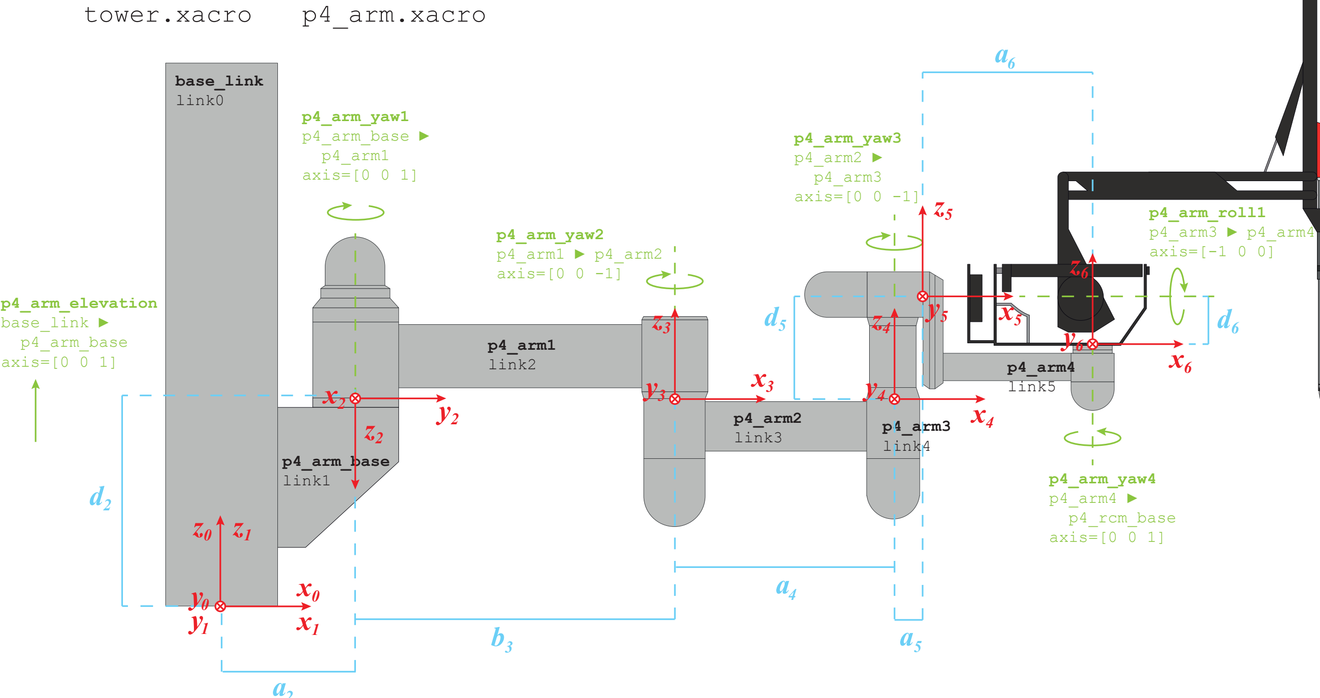
The physical, low level controller and ROS limits for each of the variables are given in table C.2

limits	θ_7^*	θ_8^*	d_{11}^*	θ_{12}^*	θ_{13}^*	θ_{14L}^*	θ_{14R}^*
physical	± 1.670	[-0.951, 0.912]	[0.169, 0.410]	± 4.712	[-1.466, 1.536]	[-1.850, $\theta_{14\mathbf{R}}^*$]	[θ_{14L}^*, 1.702]
FPGA	[-1.333, 1.424]	[-0.812, 0.773]	[0.170, 0.409]	[-4.294, 4.416]	[-0.977, 0.908]	[-0.785, 1.335]	
`xacro`	$\pm\pi/2$	[-0.8, 1]	± 0.12	$\pm 3\pi/2$	± 1.5	± 1.8	± 1.8

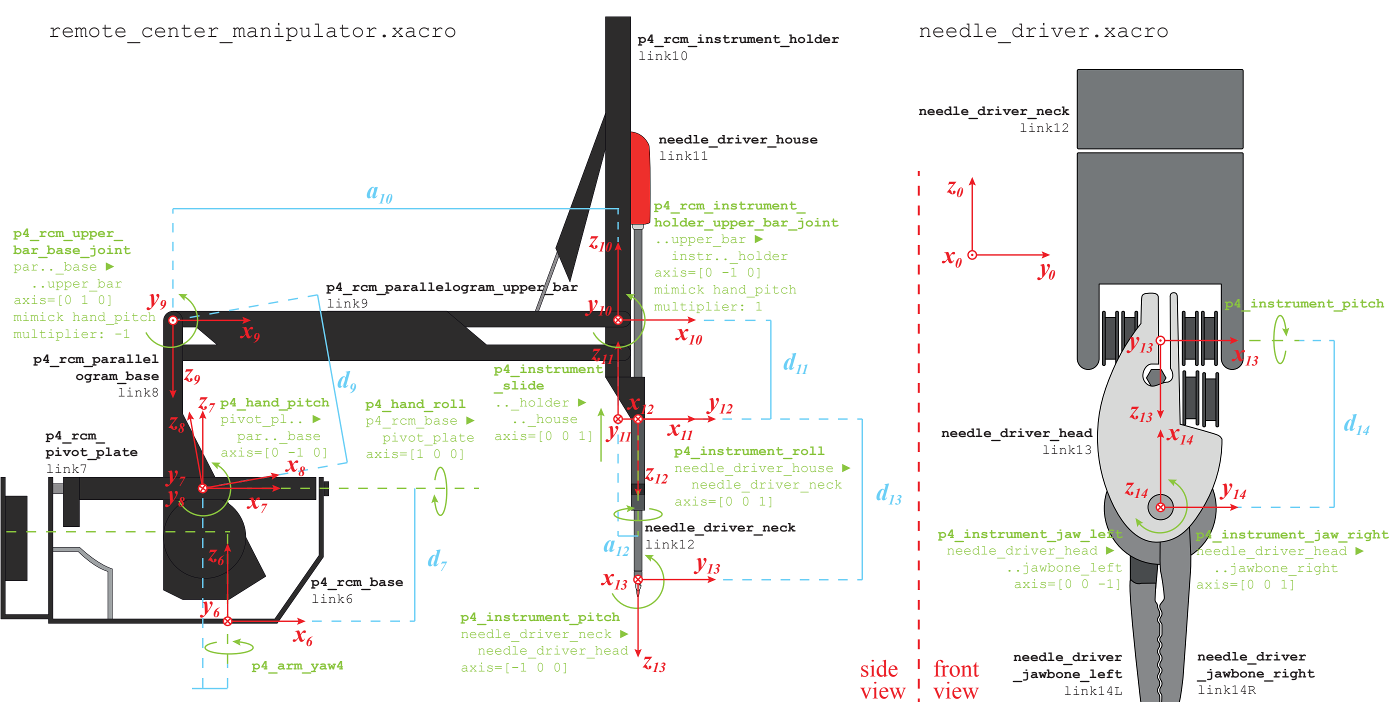
Table C.2: Limits on the (controllable) variables in table C.1 and figure C.1. The low level controller limits in the FPGA are set to avoid the physical limits, by switching off the motors on violation. The physical limits are measured limits.

⊙ outward normal
⊗ inward normal

θ_i d_i variables
a_i d_i parameters



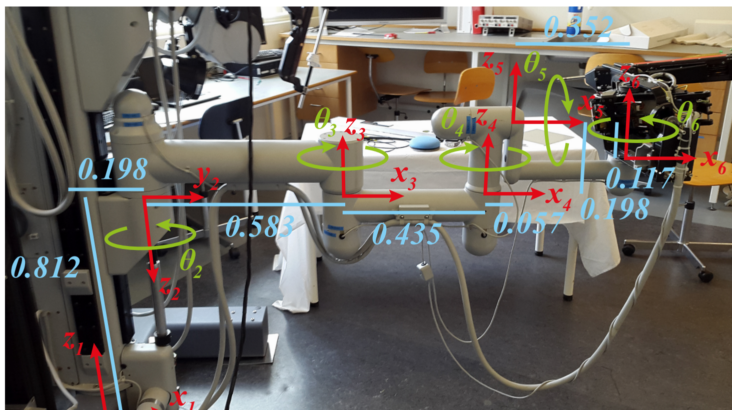
(a) Coordinate frames for the joints on the robot arm.



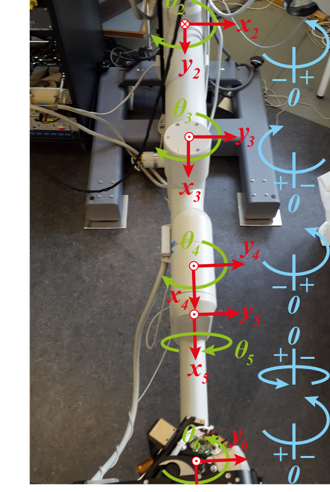
(b) Coordinate frames for the joints on the robot hand and instrument.

Figure C.1: Orientation and position of coordinate frames Ψ_0, Ψ_1, ..., Ψ_{14} according to the ROS `xacro` files.

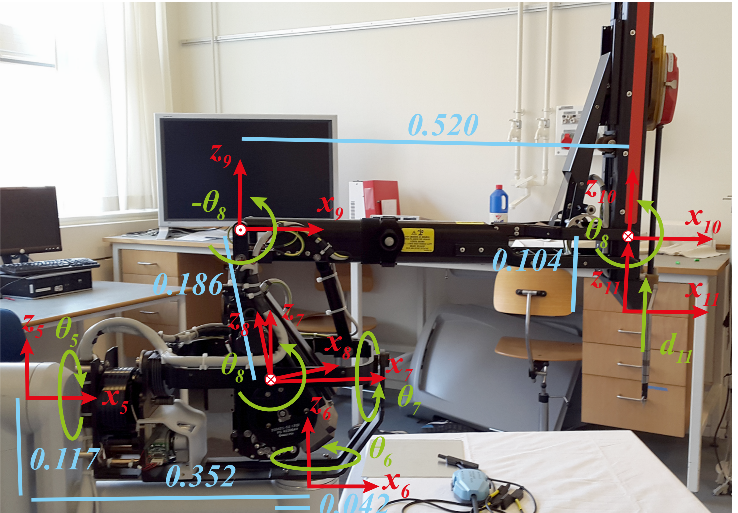
The first 6 degrees of freedom are elevation and rotation of the arm joints, and are manually set preoperatively and fixed, hence only the last 7 variables are controllable for trajectory planning. The frames are superimposed on the robot in figure C.2.



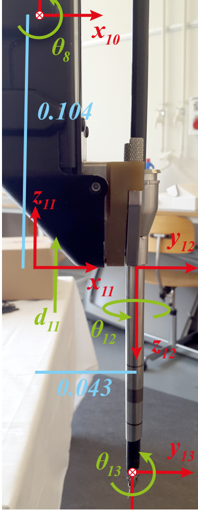
(a) Robot arm.



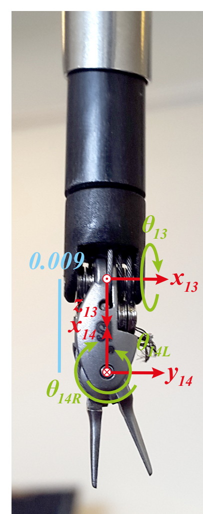
(b) Robot arm.



(c) Robot hand.



(d) Instr.



(e) Instr.

Figure C.2: Coordinate frame placement, distances and positive rotation direction for the robot arm, hand and instrument. In figure C.2b the positive rotation direction is shown for both **R**OS (green) and potentiometers (blue).

The position of the potentiometers measuring the joint variables 1-6 can be read from the interface to the secondary RIO as voltages. The scaling factor from these potentiometer voltages to the joint angle (in radians) are found through measurements and are given in table C.3.

joint rotation [rad]	θ_2, yaw1	θ_3, yaw2	θ_4, yaw3	θ_5, roll1	θ_6, yaw4
scaling factor [rad/V]	-0.225365326	0.302076216	-0.306198114	-0.311665937	0.314159265

Table C.3: Factor from potentiometer voltage measurements to arm joint angles.

C.1.1 Testing Existing Kinematics in MATLAB

MATLAB script and measurement files can be found in appendix J on the path `matlab_scripts/kinematic_models/robot_kinematics.m`. The single-axis rotation matrices are defined according to equation (C.2)

```
function rotation = rot(axis,angle)
    if axis==1
        rotation = [1 0 0; 0 cos(angle) -sin(angle); 0 sin(angle) cos(angle)];
    elseif axis==2
        rotation = [cos(angle) 0 sin(angle); 0 1 0; -sin(angle) 0 cos(angle)];
    elseif axis==3
        rotation = [cos(angle) -sin(angle) 0; sin(angle) cos(angle) 0; 0 0 1];
    end
end
```

The parameters are set according to table C.1 (corrected according to measurements, see table C.3) and the transformation matrices are computed as follows

```
%% Existing reference frames according to xacro files

% parameters: distances [m], a: along x, b: along y, d: along z
a = [0.0 0.198 0.0 0.435 0.057 0.352 -0.052 0.0 0.0 0.430 0.0 0.052 0.0 0.0 0.0];
b = [0 0 0.583 0 0 0 0 0 0 0 0 0 0 0 0];
d = [0 0.812 0 -0.001 0.198 -0.117 0.161 0 0.186 0 -0.104 0.0 0.177 0.009 0.009];

% parameters: rotations [rad]
R = [eye(3) rot(3,pi/2)*rot(1,pi) rot(3,pi/2)*rot(1,-pi) eye(3) eye(3) eye(3) eye(3) rot(2,-0.1745) rot(2,0.1745)*rot(1,pi) rot(1,pi) eye(3)
        rot(3,pi/2)*rot(1,pi) eye(3) rot(2,pi/2)*rot(1,pi/2) rot(2,pi/2)*rot(1,pi/2) ];
for i = 1:length(a)
    Rot(:,:, i) = R(:,(i-1)*3+1:i*3);
end

% ----------------------------------------------------------------------------------------------------
% variables: actuation axes
ax = [3 3 -3 -3 -1 3 1 -2 2 -2 3 3 -1 -3 3];

% first make the variable rotation matrices (assume all variables are angles)
for i = 1:length(a)
    Rot_var(:,:, i) = rot(abs(ax(i)),sign(ax(i))*state(i));
end
% eliminating the two rotations where the variable is a distance
Rot_var(:,:,1) = eye(3);
Rot_var(:,:,11) = eye(3);

% making the variable translation vectors
for i = 1:length(a)
    for j = 1:3
        if i == 1 || i == 11
            if j == abs(ax(i))
                p(j,i) = sign(ax(i))*state(i);
            end
        else
            p(j,i) = 0;
        end
    end
end

% Transformation matrices (forward kinematics)
for i = 1:length(a)
    fixed = [Rot(:,:, i) [a(i) b(i) d(i) ]'; zeros(1,3) 1];
    free = [Rot_var(:,:, i) p(:, i); zeros(1,3) 1];
    Trans(:,:, i) = fixed*free;
end
```

To test the accuracy of the defined kinematics, computed distances are compared to measured distances. The results are shown in table C.4, for different state configurations, with state = [state$_{\text{arm}}$, state$_{\text{hand}}$] $= [\{d_1, \theta_2, \theta_3, \theta_4, \theta_5, \theta_6\}, \{\theta_7, \theta_8, d_{11}, \theta_{12}, \theta_{13}, \theta_{14L}, \theta_{14R}\}]$ (as $\theta_8 = \theta_9 = \theta_{10}$, 9 and 10 are left out).

dist.	calc.	meas.
$\lvert {}^6_7 p \rvert$	16.92	16
$\lvert {}^6_8 p \rvert$	16.92	16
$\lvert {}^6_9 p \rvert$	35.43	34
$\lvert {}^6_{10} p \rvert$	55.52	53
$\lvert {}^6_{11} p \rvert$	49.75	51
$\lvert {}^6_{12} p \rvert$	54.36	52
$\lvert {}^6_{13} p \rvert$	49.18	47
$\lvert {}^6_{14} p \rvert$	49.07	47
$\lvert {}^0_{14} p \rvert$	231.49	229

(a)

dist.	calc.	meas.
$\lvert {}^6_7 p \rvert$	16.92	16
$\lvert {}^6_8 p \rvert$	16.92	16
$\lvert {}^6_9 p \rvert$	29.72	27
$\lvert {}^6_{10} p \rvert$	39.53	36
$\lvert {}^6_{11} p \rvert$	45.42	44
$\lvert {}^6_{12} p \rvert$	49.57	46
$\lvert {}^6_{13} p \rvert$	60.15	65
$\lvert {}^6_{14} p \rvert$	60.77	66
$\lvert {}^0_{14} p \rvert$	243.72	241

(b)

dist.	calc.	meas.
$\lvert {}^6_7 p \rvert$	16.92	16
$\lvert {}^6_8 p \rvert$	16.92	16
$\lvert {}^6_9 p \rvert$	33.81	31
$\lvert {}^6_{10} p \rvert$	44.99	46
$\lvert {}^6_{11} p \rvert$	44.71	49
$\lvert {}^6_{12} p \rvert$	49.82	50
$\lvert {}^6_{13} p \rvert$	53.64	53
$\lvert {}^6_{14} p \rvert$	53.99	54
$\lvert {}^0_{14} p \rvert$	185.16	189

(c)

dist.	calc.	meas.
$\lvert {}^6_7 p \rvert$	16.92	16
$\lvert {}^6_8 p \rvert$	16.92	16
$\lvert {}^6_9 p \rvert$	31.66	32
$\lvert {}^6_{10} p \rvert$	62.96	65
$\lvert {}^6_{11} p \rvert$	53.98	55
$\lvert {}^6_{12} p \rvert$	56.84	56
$\lvert {}^6_{13} p \rvert$	44.05	42
$\lvert {}^6_{14} p \rvert$	43.49	41
$\lvert {}^0_{14} p \rvert$	217.15	207

(d)

Table C.4: Calculated and measured distances [cm] between frame origins. In C.4a all variables are set to zero. In C.4b state$_{\text{hand}}$ =[1.3, 0.7, -0.05, 0, 0, -0.23, 0]. In C.4c state = [{0, 0.2, 0.5, -1.4, 0, 0.8}, {-0.5, 0.5, 0.03, 0, 0, -0.2, 0}]. In C.4d state = [{0, -0.2, -0.6, 0.9, 0, 0.4}, {-0.6, -0.6, 0, 0, 0, -0.2, 0}].

C.2 Defining Kinematics According to Denavit-Hartenberg Convention

In order to simplify calculations, the robot coordinate frame convention DH is adapted, and a new set of coordinate frames and transformation matrices are established. According to the DH convention a frame is placed such that

- frame i is fixed with respect to link i
- the z_i axis is aligned with link $i+1$ actuation axis
- variable/parameter θ_i is the angle from x_{i-1} to x_i about z_{i-1}
- variable/parameter d_i is the distance from origin $i-1$ to x_i measured along z_{i-1}
- parameter a_i is the distance from z_{i-1} to z_i measured along x_i
- parameter α_i is the angle from z_{i-1} to z_i about x_i

Using this convention, all transformations between frames can be written on the form

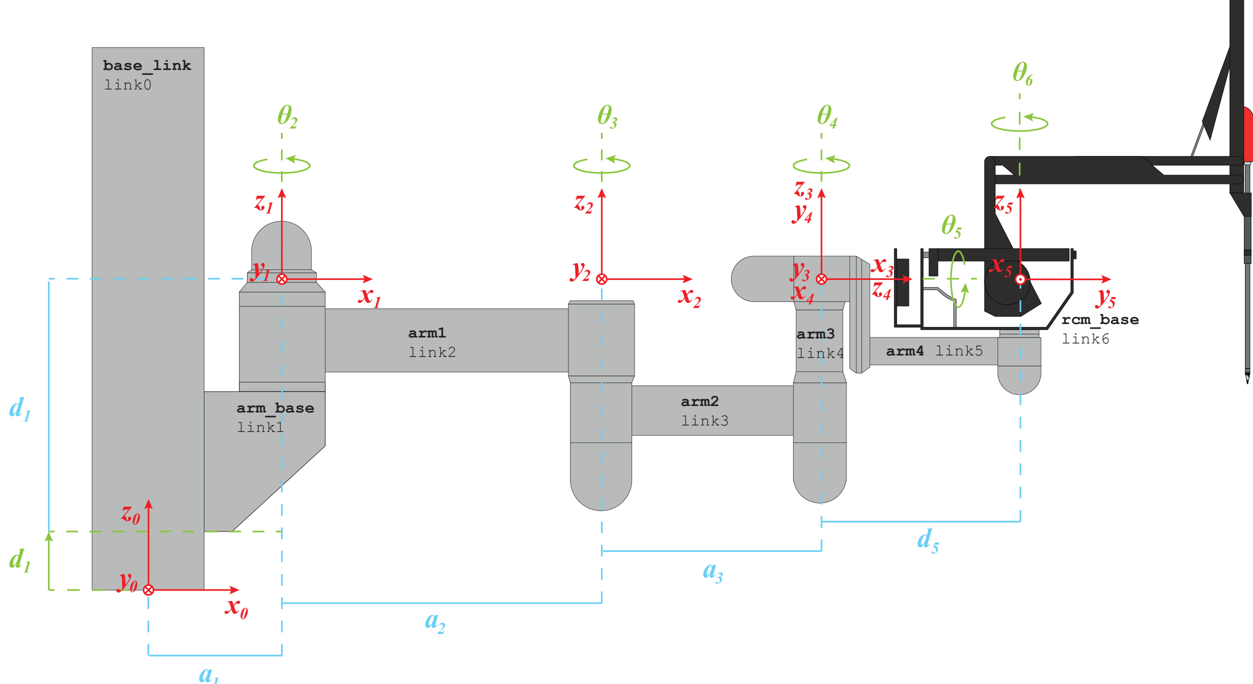
$$ {}^{i-1}_{i}T = \begin{bmatrix} \mathbf{R}_z(\theta_i) & \begin{bmatrix} 0 \\ 0 \\ d_i \end{bmatrix} \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \mathbf{R}_x(\alpha_i) & \begin{bmatrix} a_i \\ 0 \\ 0 \end{bmatrix} \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} \cos(\theta_i) & -\cos(\alpha_i)\sin(\theta_i) & \sin(\alpha_i)\sin(\theta_i) & a_i\cos(\theta_i) \\ \sin(\theta_i) & \cos(\alpha_i)\cos(\theta_i) & -\sin(\alpha_i)\cos(\theta_i) & a_i\sin(\theta_i) \\ 0 & \sin(\alpha_i) & \cos(\alpha_i) & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \text{(C.8)} $$

The placement of coordinate frames according to the DH convention is shown in figure C.3 and the parameters used for this set of frame transformations are given in table C.5.

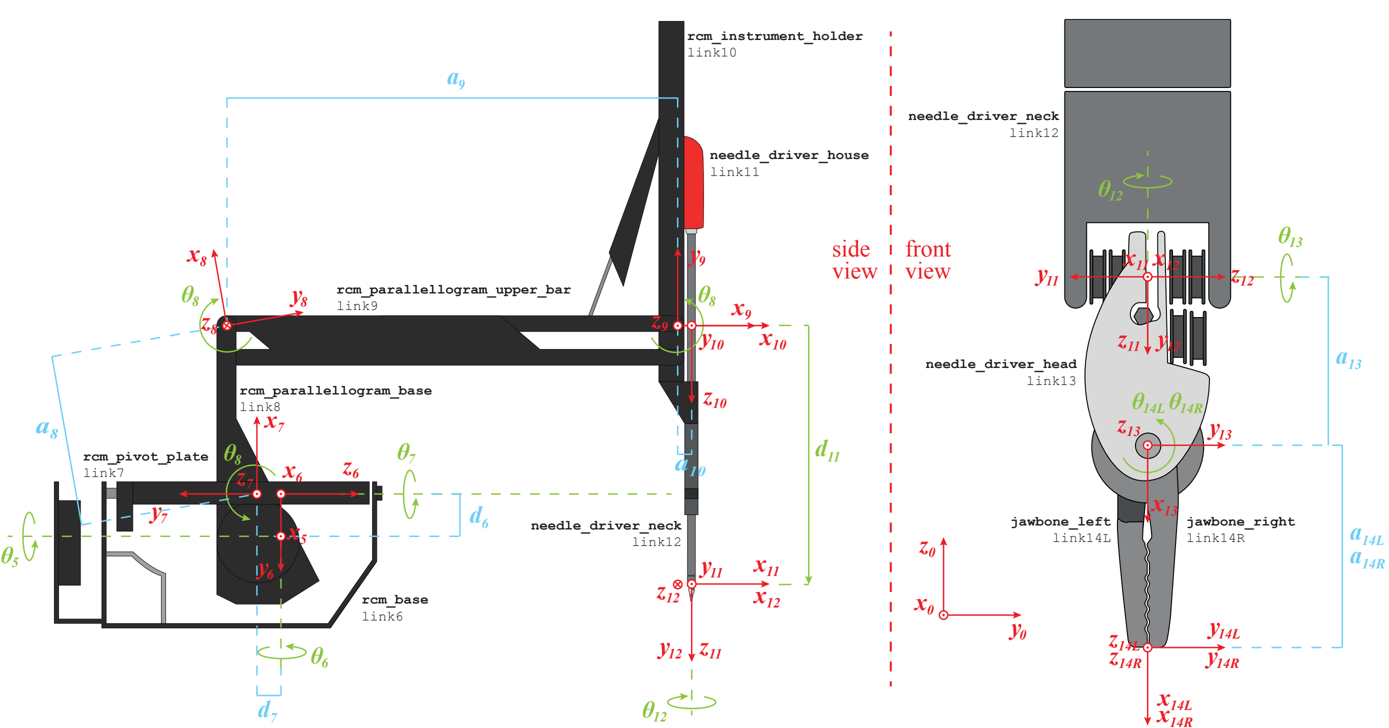
C.2.1 Testing DH Kinematics in MATLAB

The new transformation matrices are computed and tested similarly, to determine the accuracy of the defined robot kinematics, and the results are seen in table C.6. MATLAB script and measurement files can be found in appendix J on the path `matlab_scripts/kinematic_models/robot_kinematics.m`.

⊙ outward normal
⊗ inward normal
θ_i d_i variables
a_i d_i parameters



(a) Coordinate frames for the joints on the robot arm.



(b) Coordinate frames for the joints on the robot hand and instrument.

Figure C.3: Orientation and position of coordinate frames $\Psi_0, \Psi_1, ..., \Psi_{14}$ defined according to the DH convention.

i	θ_i [rad]	d_i [m]	a_i [m]	α_i [rad]
1	0	$0.453+d_1^*$	0.198	0
2	θ_2^*	0	0.582	0
3	θ_3^*	0	0.435	0
4	$\pi/2+\theta_4^*$	0	0	$\pi/2$
5	$\pi+\theta_5^*$	0.412	0	$\pi/2$
6	θ_6^*	0.047	0	$-\pi/2$
7	$-\pi/2+\theta_7^*$	-0.035	0	$-\pi/2$
8	$0.03+\theta_8^*$	0	0.190	π
9	$0.03+\pi/2+\theta_8^*$	0	0.515	π
10	θ_8^*	0	0.040	$\pi/2$
11	0	$0.282+d_{11}^*$	0	0
12	θ_{12}^*	0	0	$\pi/2$
13	$\pi/2+\theta_{13}^*$	0	0.009	$\pi/2$
14L	θ_{14L}^*	0	0.009	0
14R	θ_{14R}^*	0	0.009	0

Table C.5: Variables (marked with *) and parameters for the robot in figure C.3 defined according to the DH convention, where θ_i and d_i are rotation/translation along z_{i-1}, while a_i and α_i are translation/rotation along x_i.

```
%% Coordinate frames defined according to Denavit−Hartenberg convention
% Parameters
a_fix = [0.198 0.5820 0.435 0 0 0 0 0.1900 0.515 0.0400 0 0 0.0095 0.0095 0.0095];
d_fix = [1 0 0 0 0.4122 0.0474 −0.0450 0 0 0 0.282 0 0 0 0];
alpha = [0 0 0 pi/2 pi/2 −pi/2 −pi/2 pi pi pi/2 0 pi/2 pi/2 0 0];
theta_fix = [0 0 0 pi/2 pi 0 −pi/2 10/180*pi 10/180*pi+pi/2 0 0 0 pi/2 0 0];

% Variables (signs are included as long as state comes from old frame convention)
d_free = [d1 0 0 0 0 0 0 0 0 0 −d11 0 0 0 0];
theta_free = [0 −th2 −th3 −th4 −th5 th6 th7 th8 th8 th8 0 th12 −th13 th14L −th14R];

% Transformation matrices
for i = 1:length(a_fix)
   Tz = [rot(3,theta_free(i)+theta_fix(i)) [0;0;d_free(i)+d_fix(i)]; zeros(1,3) 1];
   Tx = [rot(1,alpha(i)) [a_fix(i);0;0]; zeros(1,3) 1];
   T_DH(:,:,i) = Tz*Tx;
end
```

(a)

dist.	calc.	meas.
$\lvert{}^5_6p\rvert$	4.74	5
$\lvert{}^5_7p\rvert$	6.54	7
$\lvert{}^5_8p\rvert$	24.71	24
$\lvert{}^5_9p\rvert$	49.60	49
$\lvert{}^5_{10}p\rvert$	53.15	53
$\lvert{}^5_{11}p\rvert$	47.94	48
$\lvert{}^5_{12}p\rvert$	47.94	48
$\lvert{}^5_{13}p\rvert$	48.04	48
$\lvert{}^5_{14}p\rvert$	48.16	48
$\lvert{}^0_{14}p\rvert$	230.20	229

(b)

dist.	calc.	meas.
$\lvert{}^5_6p\rvert$	4.74	5
$\lvert{}^5_7p\rvert$	6.54	7
$\lvert{}^5_8p\rvert$	23.79	24
$\lvert{}^5_9p\rvert$	35.40	34
$\lvert{}^5_{10}p\rvert$	39.20	39
$\lvert{}^5_{11}p\rvert$	57.82	64
$\lvert{}^5_{12}p\rvert$	57.82	64
$\lvert{}^5_{13}p\rvert$	58.54	67
$\lvert{}^5_{14}p\rvert$	59.23	68
$\lvert{}^0_{14}p\rvert$	243.20	241

(c)

dist.	calc.	meas.
$\lvert{}^5_6p\rvert$	4.74	5
$\lvert{}^5_7p\rvert$	6.54	7
$\lvert{}^5_8p\rvert$	25.14	24
$\lvert{}^5_9p\rvert$	39.04	40
$\lvert{}^5_{10}p\rvert$	43.02	42
$\lvert{}^5_{11}p\rvert$	51.33	51
$\lvert{}^5_{12}p\rvert$	51.33	51
$\lvert{}^5_{13}p\rvert$	51.86	52
$\lvert{}^5_{14}p\rvert$	52.40	52
$\lvert{}^0_{14}p\rvert$	184.99	189

(d)

dist.	calc.	meas.
$\lvert{}^5_6p\rvert$	4.74	5
$\lvert{}^5_7p\rvert$	6.54	7
$\lvert{}^5_8p\rvert$	21.65	22
$\lvert{}^5_9p\rvert$	58.87	58
$\lvert{}^5_{10}p\rvert$	61.22	61
$\lvert{}^5_{11}p\rvert$	42.53	41
$\lvert{}^5_{12}p\rvert$	42.53	41
$\lvert{}^5_{13}p\rvert$	42.08	40
$\lvert{}^5_{14}p\rvert$	41.67	40
$\lvert{}^0_{14}p\rvert$	216.08	207

Table C.6: Calculated and measured distances [cm] between frame origins. The same states are used as given in table C.4.

C.3 Defining da Vinci Kinematics for Active Joints

As the kinematics described via the `xacro` files implement translations first, and then RPY rotations (extrinsic roll (about x-axis), pitch (about y-axis), yaw (about z-axis) rotation), the DH convention cannot be implemented directly in the robot kinematics through the joint description in the `xacro` files. Furthermore, the convention here is that each frame (joint) is fixed in its child link (corresponding to the fixed rotations preceding the free rotation), and not in its parent link as in the DH convention.

A compromise is made, defining a new set of frames for the `xacro` kinematics, adhering to the DH constraint that each free rotation/translation is about/along the local z-axis. Furthermore, for convenience of the inverse kinematics solver, the two passive joints mimicking the hand pitch movement are removed from the kinematic chain, also removing a series of links (marked with grey in figure C.4b). For convenience of placing the hand roll and pitch frames in the pivot point, a virtual link is inserted in the `xacro` file after each of these two joints.

Transformation matrices describing the kinematics of the `xacro` files are written on the form

$$ {}^{i-1}_{i}T = \begin{bmatrix} \mathbf{R}_z(\text{yaw})\mathbf{R}_y(\text{pitch})\mathbf{R}_x(\text{roll}) & \begin{bmatrix} a_i \\ b_i \\ d_i \end{bmatrix} \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \mathbf{R}_z(\theta_i^*) & \begin{bmatrix} 0 \\ 0 \\ d_i^* \end{bmatrix} \\ 0 & 1 \end{bmatrix} \tag{C.9} $$

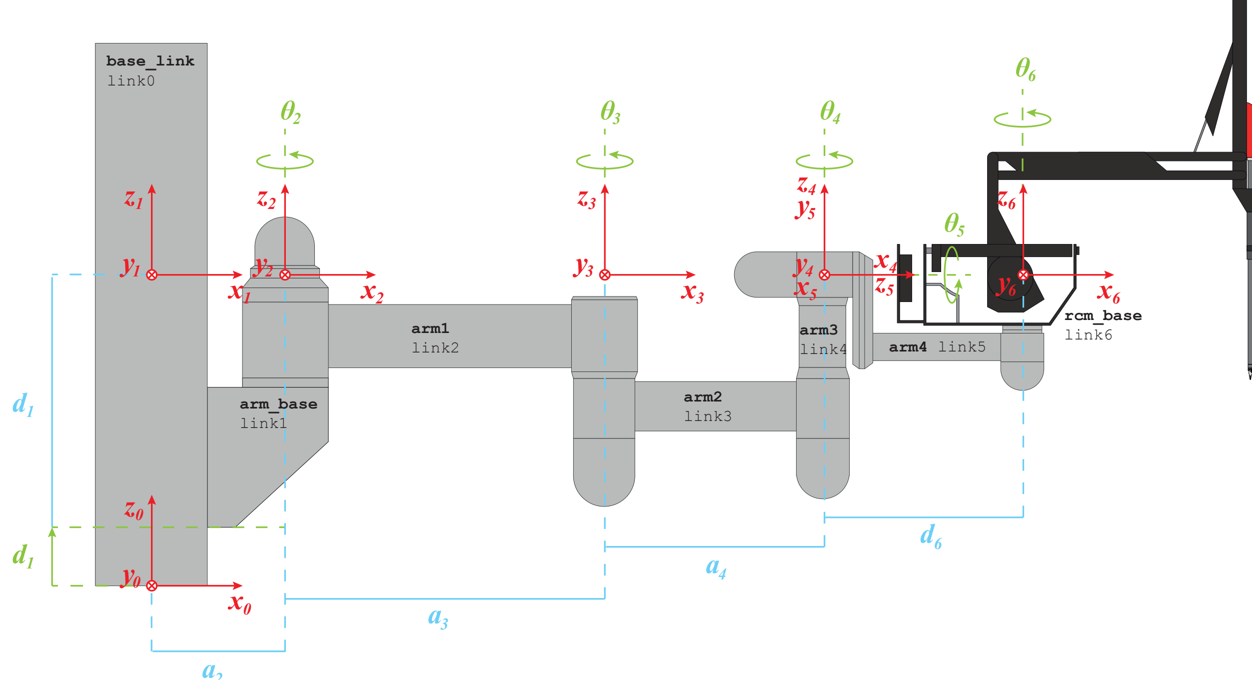
where the transformation described in ${}^{i-1}_{i}T$ is implemented in joint i in the `xacro` file as (for joint 8)

```
<joint name="p4_hand_pitch" type="revolute">
<origin
    xyz="0 0 0"
    rpy="1.5708 0 0" />
<parent link="rcm_vitual0" />
<child link="rcm_vitual1" />
<axis xyz="0 0 1" />
...
</joint>
```

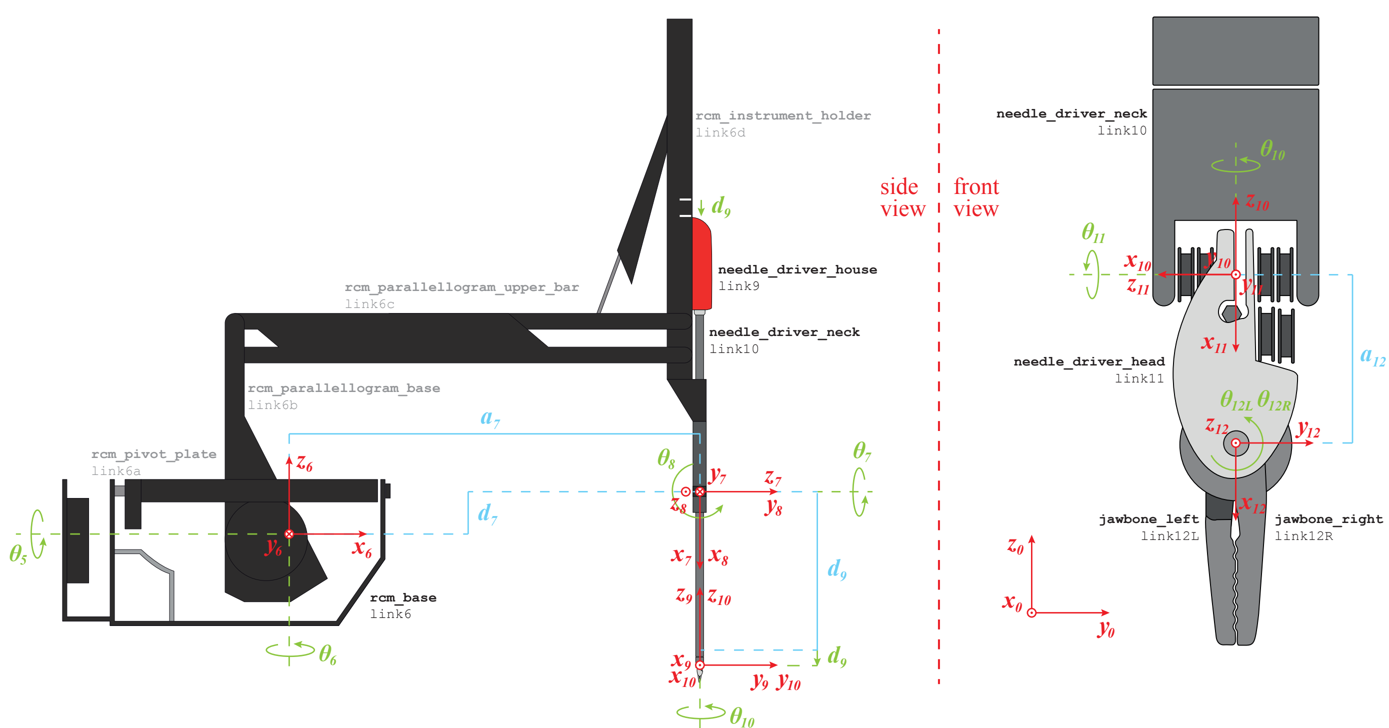
The measures for all translations and rotations are displayed in table C.7.

	fixed translation [m]			fixed rotation [rad]			freedom	
frame	a (x)	b (y)	d (z)	roll	pitch	yaw	θ^* or d^*	joint name
1	0	0	0.998	0	0	0	d_1^*	`elevation`
2	0.198	0	0	0	0	0	θ_2^*	`arm_yaw1`
3	0.582	0	0	0	0	0	θ_3^*	`arm_yaw2`
4	0.435	0	0	0	0	0	θ_4^*	`arm_yaw3`
5	0	0	0	0	$\pi/2$	0	θ_5^*	`arm_roll1`
6	0	0	0.412	0	$-\pi/2$	0	θ_6^*	`arm_yaw4`
7	0.482	0	0.047	0	$\pi/2$	0	θ_7^*	`hand_roll`
8	0	0	0	$\pi/2$	0	0	θ_8^*	`hand_pitch`
9	0.097	0	0	0	$-\pi/2$	0	d_9^*	`instrument_slide`
10	0	0	0	0	0	0	θ_{10}^*	`instrument_roll`
11	0	0	0	0	$\pi/2$	0	θ_{11}^*	`instrument_pitch`
12L	0.009	0	0	$-\pi/2$	0	0	θ_{12L}^*	`instrument_jaw_left`
12R	0.009	0	0	$-\pi/2$	0	0	θ_{12R}^*	`instrument_jaw_right`

Table C.7: Fixed translations and rotations implemented via `xacro` as described in equation (C.9), followed by a free rotation or translation about the (new) z-axis.



(a) Coordinate frames for the joints on the robot arm.



(b) Coordinate frames for the joints on the robot hand and instrument.

Figure C.4: Orientation and position of coordinate frames Ψ_0, Ψ_1, ..., Ψ_{12} defined according to the compromise between the DH convention and the `xacro` convention.

C.3.1 Testing Active Joint Kinematics in MATLABb

As for the previous sets of coordinate frames, the transformations are tested in MATLAB to check the conformity with the two other kinematic chains. MATLAB script and measurement files can be found in appendix J on the path `matlab_scripts/kinematic_models/robot_kinematics.m`.

```
%% Coordinate frames defined as a compromise between DH and the xacro syntax, excluding passive joints
% parameters: distances [m], and rotations [rad]
a = [0 0.198 0.582 0.435 0 0 0.482 0 0.097 0 0 0.009 0.009];
d = [0.998 0 0 0 0 0.412 0.047 0 0 0 0 0 0];
roll = [0 0 0 0 0 0 0 pi/2 0 0 0 -pi/2 -pi/2];
pitch = [0 0 0 0 pi/2 -pi/2 pi/2 0 -pi/2 0 pi/2 0 0];

% Variables (signs are included as long as state comes from old frame convention)
d_free = [d1 0 0 0 0 0 0 0 -d11 0 0 0 0];
theta_free = [0 -th2 -th3 -th4 -th5 th6 th7 th8 0 th12 -th13 th14L -th14R];

% Transformation matrices (forward kinematics)
for i = 1:length(a)
    fixed = [rot(2,pitch(i))*rot(1, roll(i)) [a(i) 0 d(i)]'; zeros(1,3) 1];
    free = [rot(3,theta_free(i)) [0 0 d_free(i)]'; zeros(1,3) 1];
    Trans(:,:, i) = fixed*free;
end
```

The new frame transformations result in a set of calculated distances corresponding relatively well to the measured distances, as seen in table C.8.

state as in	table C.4a	table C.4b	table C.4c	table C.4d
calculated distance	2.31 m	2.38 m	1.85 m	2.17 m
measured distance	2.29 m	2.41 m	1.89 m	2.07 m

Table C.8: Calculated and measured distances between origin of the inertial and the tool tip frames, when using the the active joint kinematics for the calculations.

Appendix D

Dynamic Model of a Beating Heart

The motion of a point on the surface of the heart can be described as a quasi-periodic rigid 3D motion, which is a combination of the two periodic motions of the diaphragm and the heart [Duindam and Sastry, 2007]. The two separate movements can be described as the vector field [Sloth et al., 2012]

$$x(t_0) = \begin{bmatrix} \sin(\omega_d t_0) \\ \cos(\omega_d t_0) \\ \sin(2\omega_d t_0) \\ \cos(2\omega_d t_0) \\ \sin(\frac{3}{2}\sin(\omega_d t_0)) \\ \cos(\frac{3}{2}\sin(\omega_d t_0)) \\ \sin(-\frac{3}{2}\sin(\omega_d t_0)) \\ \cos(-\frac{3}{2}\sin(\omega_d t_0)) \\ \sin(\omega_h t_0) \\ \cos(\omega_h t_0) \\ \sin(2\omega_h t_0) \\ \cos(2\omega_h t_0) \\ \sin(\frac{9}{4}\cos(\omega_h t_0)) \\ \cos(\frac{9}{4}\cos(\omega_h t_0)) \\ \sin(\frac{6}{8}\cos(2\omega_h t_0) - \frac{9}{8}) \\ \cos(\frac{6}{8}\cos(2\omega_h t_0) - \frac{9}{8}) \end{bmatrix}, \qquad \dot{x}(t) = \begin{bmatrix} \omega_d x_2 \\ -\omega_d x_1 \\ 2\omega_d x_4 \\ -2\omega_d x_3 \\ \frac{3}{2}\omega_d x_2 x_6 \\ -\frac{3}{2}\omega_d x_2 x_5 \\ -\frac{3}{2}\omega_d x_2 x_8 \\ \frac{3}{2}\omega_d x_2 x_7 \\ \omega_h x_1 0 \\ -\omega_h x_9 \\ 2\omega_h x_{12} \\ -2\omega_h x_{11} \\ -\frac{9}{4}\omega_h x_9 x_{14} \\ \frac{9}{4}\omega_h x_9 x_{13} \\ -\frac{6}{8}\omega_h x_{11} x_{16} \\ \frac{6}{8}\omega_h x_{11} x_{15} \end{bmatrix} + \begin{bmatrix} x_2 & 0 \\ -x_1 & 0 \\ 2x_4 & 0 \\ -2x_3 & 0 \\ \frac{3}{2}x_2 x_6 & 0 \\ -\frac{3}{2}x_2 x_5 & 0 \\ -\frac{3}{2}x_2 x_8 & 0 \\ \frac{3}{2}x_2 x_7 & 0 \\ 0 & x_{10} \\ 0 & -x_9 \\ 0 & 2x_{12} \\ 0 & -2x_{11} \\ 0 & -\frac{9}{4}x_9 x_{14} \\ 0 & \frac{9}{4}x_9 x_{13} \\ 0 & -\frac{6}{8}x_{11} x_{16} \\ 0 & \frac{6}{8}x_{11} x_{15} \end{bmatrix} \cdot d \tag{D.1}$$

where

t_0	is the start time ($t_0 = 0$)	[s]
ω_d	is the frequency of the diaphragm movement, read off ECG ($\omega_d = \frac{2\pi}{4}$)	[rad/s]
ω_h	is the frequency of the heart movement, read off mechanical ventilator ($\omega_h = \frac{2\pi}{1.1}$)	[rad/s]
x_i	is the ith entry of the state vector x	[·]
d	is a disturbance vector ($d_1 = d_{\text{diaphragm}} \equiv [-0.4, 0.4]$ and $d_2 = d_{\text{heart}} \equiv [-0.11, 0.11]$)	[rad/s]

The two transformation matrices describing the movement of the diaphragm frame relative to the inertial frame, and the heart frame relative to the diaphragm frame can be composed from the state at time t as [Sloth et al., 2012]

$${}^0_dH(t) = \begin{bmatrix} x_6 & x_5 x_8 & x_5 x_7 & 0 \\ -x_5 & x_6 x_8 & x_6 x_7 & \frac{7}{6}x_4 - \frac{7}{6} \\ 0 & -x_7 & x_8 & \frac{5}{3}x_1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad {}^d_hH(t) = \begin{bmatrix} x_{14}x_{16} & x_{13} & -x_{14}x_{15} & \frac{14}{15}x_{10} - 2 \\ -x_{13}x_{16} & x_{14} & x_{13}x_{15} & -\frac{10}{9}x_9 - \frac{10}{9} \\ x_{15} & 0 & x_{16} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{D.2}$$

The desired position of the robot manipulator can be formulated as the desired transformation (rotation and distance) from the heart (surface point) frame.

Appendix E

MATLAB Toolbox SOSTOOLS

This appendix serves as a short description of how to download and install the MATLAB toolbox SOS-TOOLS.

SOSTOOLS is a free third-party Matlab toolbox developed by engineering departments of four major universities. A zip file of the toolbox can be downloaded from

http://www.cds.caltech.edu/sostools

also containing a user guide [Papachristodoulou et al., 2013] to sum of squares (SOS) problems and how to formulate a problem to solve it with the toolbox, along with a set of demos.

SOSTOOLS takes as input the SOS program formulation, recasts it as a Semi-Definite Programming (SDP) problem, calls SDP solvers, and recasts the solution to the SDP problem into the solution to the SOS problem. This means that the toolbox requires that an SDP solver toolbox is installed, e.g. the SeDuMi (Self-Dual-Minimization) solver, which can be downoaded from

http://sedumi.ie.lehigh.edu/downloads

The toolboxes are activated in Matlab by adding the downloaded unzipped folders to the Matlab path.

Appendix F

Measurement Logs

This appendix contains measurement description of most experiments carried out.

F.1 Step Response of Slide Position

The test setup for this test is fairly simple and includes:

- The da Vinci robot
- A laptop (preferable with 10 GB storage available on the AFS drive)

The setup is depicted in figure F.1.

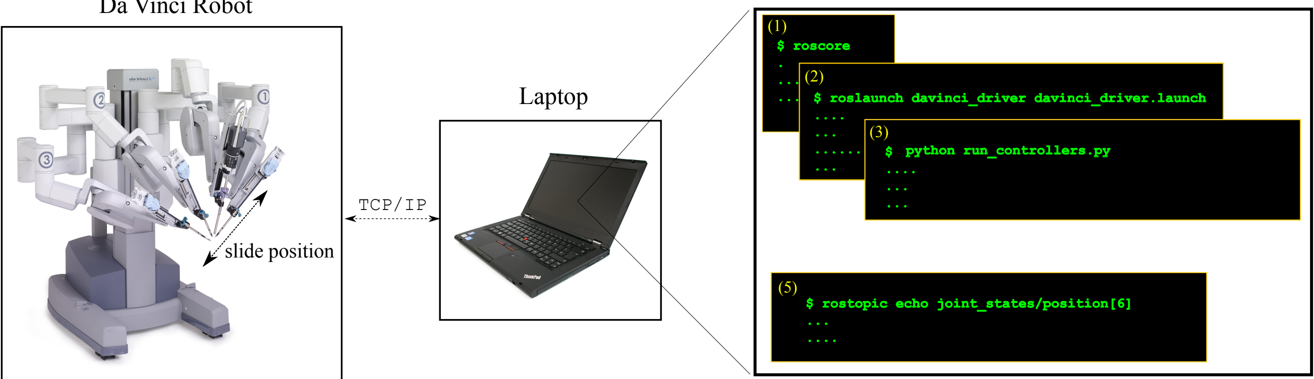


Figure F.1: Test setup to measure slide position.

- Configure the ROS environment as described in appendix A, i.e. make sure all low level controllers are running, that roscore is running and that the davinci_driver (on branch gr1032) is running. Make sure that the gr1032 package is cloned and that the python wrapper script found in appendix I is copied to the root of the workspace.

At this point, three terminals should be running. Now, open two additional terminals and prepare both by typing:

- ssh <user name>surgery-srv.lab.es.aau.dk
- cd <path_to_ root_of_workspace>
- source devel/setup.bash.

First Terminal

Type:

```
python run_controllers.py
```

This launches the User Interface (UI) shown below.

```
1  ****************************************************
2  The following options are avaiable:
3  ----------------------------------------------------
4  press 'a' to run slide safety controller
5  press 'b' to specify custom joint angles (FK mode)
6  press 'c' to run demo
7  press 'd' to run beating heart controller
8  press 'e' to specify custom 3D angles (IK mode)
9  press 'f' to run 3D safety controller
10 ****************************************************
```

Type **b + enter** to enter custom joint angle mode. It is by default at its zero position for all joint angles. Type 0.005 for slide position and zero for the remaining angles.

Second Terminal

By subscribing to the `joint_state` topic (`rostopic echo joint_states`), all information about the current states can be fetched from the sensors, i.e. the potentiometers that measure all joint angles. An example of this is shown below.

```
1  ---
2  header:
3    seq: 4553
4    stamp:
5      secs: 1428950592
6      nsecs: 666452523
7    frame_id: ''
8  name: ['p4_hand_pitch', 'p4_hand_roll', 'p4_instrument_jaw_left', 'p4_instrument_jaw_right', '
       p4_instrument_pitch', 'p4_instrument_roll', 'p4_instrument_slide']
9  position: [-0.021504180505871773, 0.027300411835312843, 0.0006707065622322261, -0.00013414131535682827,
       0.0012072718236595392, -0.0896063968539238, 1.055011398420902e-05]
10 velocity: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
11 effort: [-0.5, -0.5, -0.5, -0.5, -0.5, -0.5, -0.5]
12 ---
```

For this test, it is more appropriate to merely publish the slide position, this can be done by:

```
rostopic echo joint_states/position[6]
```

Which gives an output as shown below.

```
1  ---
2  8.20564400783e-06
3  ---
4  8.20564400783e-06
5  ---
6  8.20564400783e-06
7  ---
```

Instead of leaving the output as a terminal output, the information is mapped to a `.txt` file with a suitable name, e.g:

```
rostopic echo joint_states/position[6] > taus_05cm_1_speedlimit_100.txt
```

Use the MATLAB script and the recorded measurement data found in appendix J under the path `matlab_scripts/slide_step/plot_slide_pos.m`, to plot the recorded slide position along with an estimated first and second order approximation. The step response is seen in figure F.2.

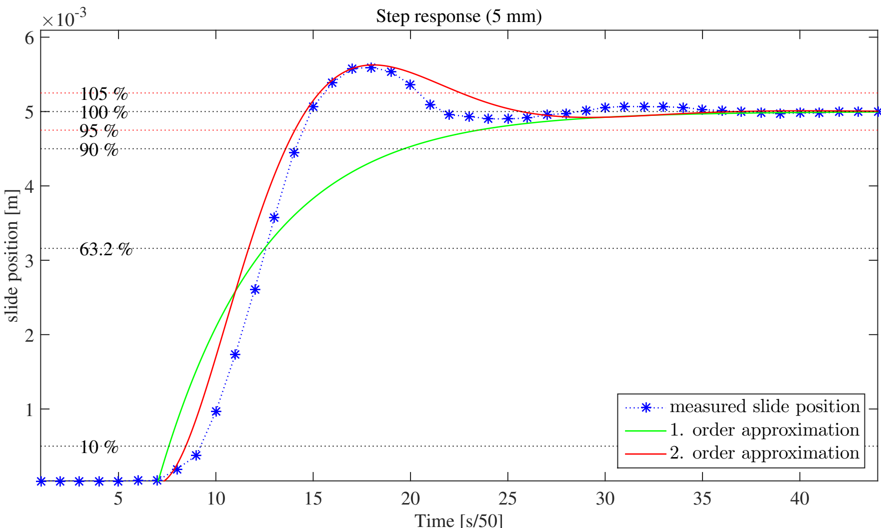


Figure F.2: Step response from 0 mm to 5 mm. Plot details and measurements can be found in appendix J as `matlab_scripts/slide_step/plot_slide_pos.m`

.

This completes this measurement log.

F.2 Step Response of the da Vinci Robot in 3D Cartesian Space

This test is carried out in a similar manner as the test in section F.1, now subscribing to the full `joint_state` topic (`rostopic echo joint_states`) as described in section F.1. This displays the measurements of the joint angles from the potentiometers, that correspond to angles in radians (and for `instrument_slide`: in meters), secured by the calibration of the motor gearings. An example of a measurement can be seen in section F.1.

The measurements are copied directly from the terminal output, and the joint measurements extracted from this string. Use the MATLAB script and the recorded measurement data found in appendix J under the path `matlab_scripts/step_3d/3d_time_const_measurement.m`, to plot the recorded step responses. The step responses for steps in the x, y and z direction are shown in figure F.3.
From the plots it is seen, as expected, that a step in one Cartesian coordinate direction is not completely decoupled from the other two directions, because of the movements being implemented by the six revolute joints of the da Vinci robot. It is, on the other hand, also seen that the movements in the other directions are sub-millimeter and that they settle approximately at their initial value. Thus it is considered that the three directions can be seen as approximately decoupled, and a first order approximation of the step response for each direction is presented in figure F.4.

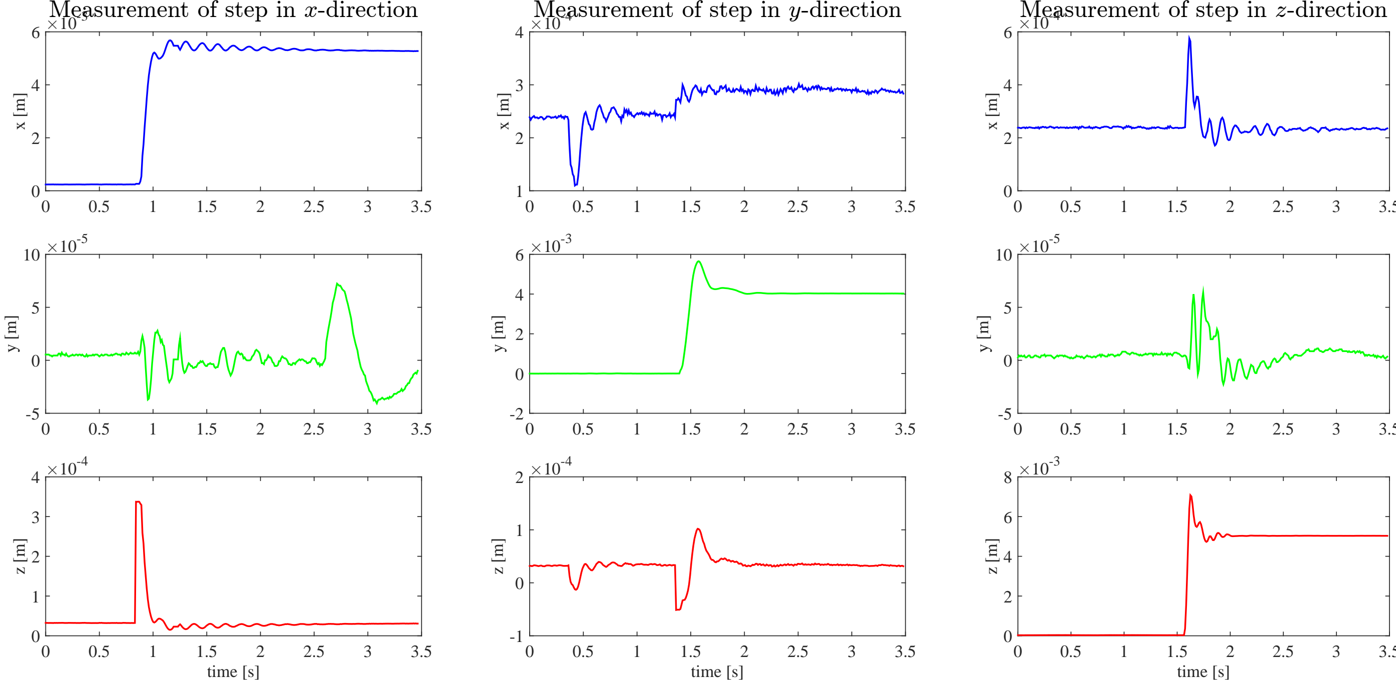


Figure F.3: Step responses for the x, y and z directions from 0 mm to 5 mm. Plot details and measurements can be found in appendix J in `matlab_scripts/step_3d/3d_time_const_measurement.m`.

For the step in the x direction it is seen how the end effector oscillates for a long time after the step movement, which can be attributed to the flexibility of the structure. The step in the x direction is mainly implemented through a `hand_pitch` angle change (see figure 1.4).

It is seen how the step input of 5 mm in the y direction only causes a robot movement of 4 mm in this direction. This is experiences in all tests, and is attributed to uncertainties in the kinematics and in the inverse kinematics solver. The step in the y direction is mainly implemented through a `hand_roll` angle change (see figure 1.4).

The step in the z direction is mainly implemented through a `instrument_slide` angle change (see figure 1.4), and should be comparable to figure F.2. It is, however, seen that the dynamics have changed, which is expected since an inverse kinematics solver is employed (see subsection 6.5.1). This does not explain the considerably lower time constant, which may be explained by a change in the ROS setup from using MoveGroup for publishing control signals to doing it directly (see appendix A for an overview of the two approaches to publishing control signals in ROS), or from changes in the low level controllers and joint motor gearings.

The time constants inferred from these measurements are

$$\tau_x = 70.088 \text{ ms}$$
$$\tau_y = 100.288 \text{ ms}$$
$$\tau_z = 40.114 \text{ ms}$$

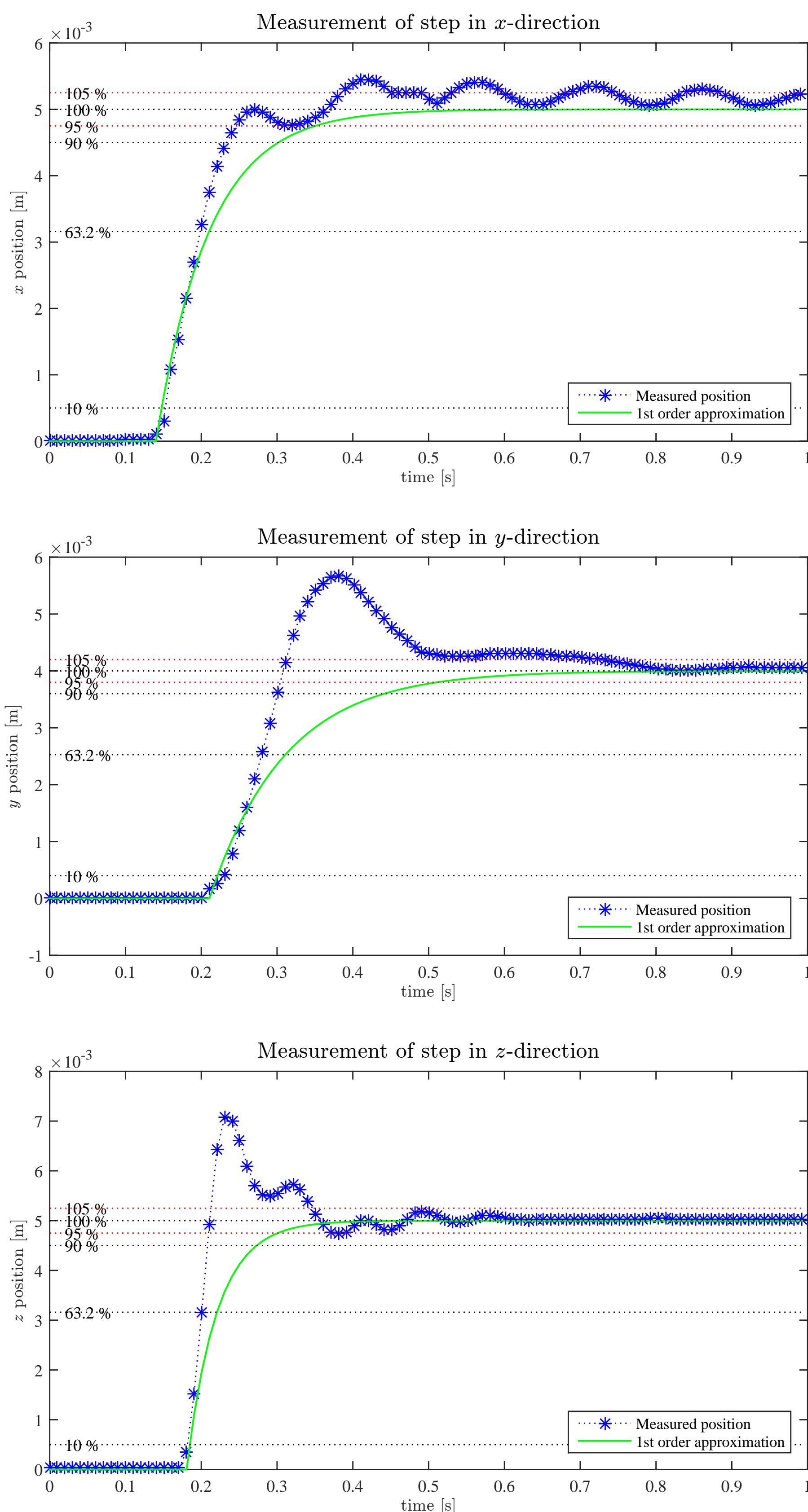


Figure F.4: Step response for each direction from 0 mm to 5 mm along with first order approximations. It is seen how uncertainties in the y direction cause the robot to step to only 4 mm. Plot details and measurements can be found in appendix J in `matlab_scripts/step_3d/3d_time_const_measurement.m`

Appendix G

MATLAB Implementation

G.1 Implementation of Safety Controllers

This appendix contains the MATLAB implementation of the controllers developed throughout the project, i.e.:

- Safety controller for a system with static boundaries for instrument slide in subsection G.1.1.
- Safety controller for a system with dynamic boundaries to simulate a set-up consisting of a beating heart and a safe distance between heart and robot end effector. This is in subsection G.1.2.
- Safety controller for a system in 3D Euclidean space comprising control of all joints in the robotic hand, employing forward an inverse kinematics. This is in found subsection G.1.3.

G.1.1 Implementation of Instrument Slide Safety Controller with Static Boundaries

This section contains the MATLAB implementation of the slide controller developed in chapter 4. No user inputs are required to run the script. The script shown here includes no plotting, but the script found in appendix J under the path `matlab_scripts/slide_controller/slide_controller.m` includes all plotting details.

```
1  model = 2; % 1 = first order model, 2 = second order model
2
3  %--- parabola coeficients for position constraints ---%
4  a = 16/9; b = 4/45; c = -2/225;
5  %--- elliptic paraboloid coeficients for position constraints ---%
6  x10 = 1/40; x20 = 0; a1 = -3/40; b1 = -10; c1 = 1; c2 = -1;
7
8  if model == 2
9      s = tf('s'); % prepare Laplace operator
10     ts = (28-9)*1/50; % 5 percent settling time
11     tr = 0.1; % rise time
12     wn = 1.8/tr; % calculate natural frequency
13     zeta = -1/(wn*ts)*log(0.02); % calculate the damping ratio
14     H = wn^2/(s^2 + 2*zeta*wn*s + wn^2); % calculate transfer function
15     num = wn^2; % Specify numarator
16     den = [1 2*zeta*wn wn^2]; % specify denominator
17     A = [0 1; -wn^2 -2*zeta*wn];
18     B = [0 wn^2]';
19     C = [1 0];
20     D = 0;
21     sys = ss(A,B,C,D)
22     x(1,1) = 0 % initial state position
23     x(2,1) = 0; % initial state velocity
24     K = acker(sys.a,sys.b,[-14 -15]);
25 elseif model == 1
26     tau = 0.110; % time constant
```

```
27      a_sys = −1/tau; %
28      b_sys = 1/tau; % sine wave frequency
29      sys = ss(a_sys,b_sys,1,0);
30      x(1,1) = 0; % initial state;
31      K = acker(a_sys,b_sys,[1.1*eig(sys.a)]); % control gain
32  end
33
34  kappa = 1; % design parameter
35  Nbar = − inv(sys.c*inv(sys.a−sys.b*K)*sys.b); % ensure unity gain
36  scrsz = get(groot,'ScreenSize'); % get screen information
37
38  %−−− Find epsilon −−−%
39  x_epsilon = 0.04; % find epsilon from desired soft limit
40  epsilon = a*x_epsilon^2 + b*x_epsilon + c; % find epsilon
41  syms x0
42  softlims = solve(a*x0^2 + b*x0 + c == epsilon); % find soft limits
43  epsilon = abs(epsilon); % specify ep silon as a positive number
44
45  %−−− make reference vector −−−%
46  XREF = [0.02 0.09 −0.14 −0.02 0.045 0.01]; % simulation setpoints
47  xref = XREF(1); % initial reference
48
49  f = 100; Ts = 1/f; % sampling frequency
50  N = 5; % simulation time in seconds
51  fprintf('Simulation time: %d seconds\n', N)
52
53  i = (0:Ts:N); % make simulation resolution realistic
54  utilde = zeros(round(length(i)),1); % init utilde
55  Rplot(1) = 1; % init reference plot
56
57  for R = 1:length(i)
58    %−−− set various references −−−%
59    REFS = 6;
60    if R == round(length(i)/REFS)*1
61        xref = XREF(2);
62        Rplot(2) = R;
63    elseif R == round(length(i)/REFS)*2
64        xref = XREF(3);
65        Rplot(3) = R;
66    elseif R == round(length(i)/REFS)*3
67        xref = XREF(4);
68        Rplot(4) = R;
69    elseif R == round(length(i)/REFS)*4
70        xref = XREF(5);
71        Rplot(5) = R;
72    elseif R == round(length(i)/REFS)*5
73        xref = XREF(6);
74        Rplot(6) = R;
75    end
76
77    %−−− physical constraints for velocity −−−%
78    if 1
79      if model == 2
80        max_vel = 1;
81        if x(2,R) > max_vel
82            x(2,R) = max_vel;
83        elseif x(2,R) < −max_vel
84            x(2,R) = −max_vel;
85        end
86      end
87    end
88
89    %−−− output −−−%
90    y(:,R) = sys.C*x(:,R);
```

```
91
92  %--- determine sigma ---%
93  if model == 1
94      if (a*(x(1,R))^2 + b*(x(1,R)) + c) <= -epsilon
95          sigma = 0;
96      elseif ((a*(x(1,R)).^2 + b*(x(1,R)) + c) > -epsilon) && ...
97              ((a*(x(1,R)).^2 + b*(x(1,R)) + c) < 0)
98          sigma = -2*((a*(x(1,R)).^2 + b*(x(1,R)) + c)/epsilon).^3 - ...
99                  3.*((a*(x(1,R)).^2 + b*(x(1,R)) + c)/epsilon ).^2 + 1;
100     else
101         sigma = 1;
102     end
103 elseif model == 2
104     cbf = (a.*(x(1,R)).^2 + b.*x(1,R) + c);
105     if cbf <= -epsilon
106         sigma = 0;
107     elseif (cbf > -epsilon) && (cbf < 0)
108         if model == 1
109             sigma = -2*((a*(x(1,R)).^2 + b*(x(1,R)) + c)/epsilon).^3 - ...
110                     3.*((a*(x(1,R)).^2 + b*(x(1,R)) + c)/epsilon ).^2 + 1;
111         elseif model == 2
112             sigma = -2*((a*(x(1,R)).^2 + b*(x(1,R)) + c)/epsilon).^3 - ...
113                     3.*((a*(x(1,R)).^2 + b*(x(1,R)) + c)/epsilon ).^2 + 1;
114         end
115     else
116         sigma = 1;
117     end
118 end
119
120 %--- print every thousand iteration to user ---%
121 if mod(R,1000) == 1
122     if R ~= 1
123         fprintf (' iter  = %d of %d\n', R-1, length(i)-1);
124     else
125         fprintf (' iter  = %d of %d\n', R, length(i)-1);
126     end
127 end
128
129 %--- find lie derivatives ---%
130 if model == 2
131     LgB(1,R) = (c1*wn^2*(2*x(2,R) + 2*x20))/b1^2;
132     LfB(1,R) = (c1*x(2,R)*(2*x(1,R) + 2*x10))/a1^2 - ...
133         (c1*(2*x(2,R) + 2*x20)*(x(1,R)*wn^2 + 2*x(2,R)*zeta*wn))/b1^2;
134 elseif model == 1
135     LgB(1,R) = (2*(a)*(x(:, R)) + (b))*(sys.b);
136     LfB(1,R) = (2*(a)*(x(:, R)) + (b))*((sys.a)*x(:, R));
137 end
138
139 %-- Find controller by pole placement --%
140 utilde(1,R) = xref*Nbar - K*x(:,R);
141
142 %--- Find safe controller ---%
143 threshold = 0.001;
144 if abs(LgB(1,R)) >= threshold
145     k0(1,R) = -( ( LfB(1,R) + sqrt(LfB(1,R)^2 ...
146         + kappa^2*LgB(1,R)*LgB(1,R)' )) / (LgB(1,R)*LgB(1,R)') ) *LgB(1,R);
147     kplot(1,R) =  k0(1,R);
148 else
149     k0(1,R) = 0;
150     kplot(1,R) = k0(1,R);
151 end
152
153 %--- control law ---%
154 u0(1,R) = sigma*k0(1,R)+(1-sigma)*utilde(1,R);
```

```
155
156   %--- physical constraints for control signal  ---%
157   slide_lim  = 0.1;
158   if  u0(1,R) > slide_lim
159       u0(1,R) = slide_lim;
160   elseif  u0(1,R) < -slide_lim
161       u0(1,R) = -slide_lim;
162   end
163
164   %--- save the LfclB ---%
165   LfclB(1,R) = LfB(1,R) + LgB(1,R).*k0(1,R);
166
167   %--- extrapolate with forward euler ---%
168   xdot = sys.a*x(:,R) + u0(1,R)*sys.b;
169   x(:,R+1) = xdot*Ts + x(:,R);
170   sig(1,R) = sigma;
171
172 end
```

G.1.2 Implementation of Instrument Slide Safety Controller with Dynamic Boundaries

This section contains MATLAB implementation of the controller developed in chapter 5. All plotting details are omitted in this section, but the controller can also be found in appendix J under the path `matlab_scripts/beating_heart/beating_heart_controller.m` which includes the plotting section as well.

```
1  %--- setup systems ---%
2  k = 9;
3  Nbar = 10;
4  tau = 0.110;
5  wh = 2*pi/1.1;
6  A = [-1/tau  0    0   0;
7        0       0    wh  0;
8        0      -wh   0   0;
9        0       0    0   0];
10 B = [(1/tau)  0 0  0]';
11 K = [-k Nbar 0 Nbar];
12
13 %--- initial conditions ---%
14 x(1,1)  = 0.05;  % initial  state  position
15 x(2,1)  = -0.03; % initial  heart  position
16 x(3,1)  = 0;  % initial  heart  velocity
17 x(4,1)  = 0.03;  % initial  distance
18
19 kappa = 1;
20 scrsz = get(groot,'ScreenSize'); % get screen information
21
22  f  = 2000; Ts = 1/f;  % sampling frequency
23 N = 5;  % simulation time in  seconds
24  fprintf ('Simulation time: %d seconds\n', N)
25
26  i  = (0:Ts:N); % make simulation resolution  realistic
27  utilde  = zeros(round(length(i)),1);  % init  utilde
28 Rplot(1)  = 1;  % init  reference  plot
29
30 %--- run controller ---%
31 epsilon  = 0.01;
32 for  R = 1:length(i)
33
```

```
34    if mod(R,1000) == 1
35        if R ~= 1
36            fprintf (' iter  = %d of %d\n', R-1, length(i)-1);
37        else
38            fprintf (' iter  = %d of %d\n', R, length(i)-1);
39        end
40    end
41
42    %--- give an unsafe distance ---%
43    if R > f*2
44        x(4,R) = -0.01;
45    end
46
47    cbf = x(2,R) - x(1,R);
48    %--- determine sigma ---%
49    if cbf <= -epsilon
50        sigma = 0;
51    elseif ( (cbf > -epsilon) && (cbf < 0)    )
52        sigma = -2*(cbf/epsilon).^3 - 3.*(cbf/epsilon).^2 + 1;
53    else
54        sigma = 1;
55    end
56    %sigma = 0;
57
58    %--- find lie derivatives ---%
59    LgB(1,R) = -1/tau;
60    LfB(1,R) = wh*x(3,R) + x(1,R)/tau;
61
62    %-- Find controller by pole placement --%
63    utilde (1,R) = K*x(:,R);
64
65    %--- Find safe controller ---%
66    threshold = 0.001;
67    if abs(LgB(1,R)) >= threshold
68        k0(1,R) = -( ( LfB(1,R) + sqrt(LfB(1,R)^2 ...
69            + kappa^2*LgB(1,R)*LgB(1,R)' )) / (LgB(1,R)*LgB(1,R)') ) *LgB(1,R);
70        kplot(1,R) =  k0(1,R);
71    else
72        k0(1,R) = 0;
73        kplot(1,R) = k0(1,R);
74    end
75
76    %--- control law ---%
77    u0(1,R) = sigma*k0(1,R)+(1-sigma)*utilde(1,R);
78
79    %--- save the LfclB ---%
80    LfclB(1,R) = LfB(1,R) + LgB(1,R).*k0(1,R);
81
82    %--- extrapolate with forward euler ---%
83    xdot = A*x(:,R) + u0(1,R)*B;
84    x(:,R+1) = xdot*Ts + x(:,R);
85
86    %--- record sigma ---%
87    sig(1,R) = sigma;
88
89    %--- calculate distance between heart and robot end effector ---%
90    Delta(1,R) = x(1,R) - x(2,R);
91 end
```

G.1.3 Implementation of 3D Euclidean Space Safety Controller

This section contains the MATLAB implementation of the controller of the full robotic hand and instrument in 3D Euclidean space developed in chapter 6. All plotting details are omitted in this section, but the controller can also be found in appendix J under the path `matlab_scripts/safe_3d/safety_in_3d.m`.

```
close all;
clear;
clc;
format long;
hfile = matlab.desktop.editor.getAll;

%--- setup systems ---%
taux = 0.070;
tauy = 0.100
tauz = 0.041

A = [-1/taux    0         0 ;
0         -1/tauy     0 ;
0          0         -1/tauz];
B = [ 1/taux    0         0 ;
0          1/tauy     0 ;
0          0          1/tauz];
K = place(A,B,[-15 -15 -15])
C = eye(3);
x = [0.05 0.05 0.02]'; % initial conditions
xref = [0.05 0.05 0.02]'; % initial reference

Nbar = - inv(C*inv(A-B*K)*B); % ensure unity gain
kappa = 1;
scrsz = get(groot,'ScreenSize'); % get screen information

f = 2000; Ts = 1/f; % sampling frequency
N = 10; % simulation time in seconds
fprintf('Simulation time: %d seconds\n', N)

i = (0:Ts:N); % make simulation resolution realistic
%utilde = zeros(3,round(length(i))); % init utilde
Rplot(1) = 1; % init reference plot

cx = 0;
cy = 0;
cz = 0;
rx = 0.03;
ry = 0.06;
rz = 0.03;

%--- run controller ---%
for R = 1:length(i)
xref_vec(:,R) = xref;
if mod(R,1000) == 1
if R ~= 1
fprintf(' iter = %d of %d\n', R-1, length(i)-1);
else
fprintf(' iter = %d of %d\n', R, length(i)-1);
end
end

if R > f*1
xref = [0.06 0.02 0.0]';
end
if R > f*1.5
xref = [-0.055 0.03 0.01]';
```

```
58  end
59   if  R > f*2.5
60  xref  = [-0.01 -0.01 0]';
61  end
62   if  R > f*3.5
63  xref  = [-0.055 0 -0.01]';
64  end
65   if  R > f*4.5
66  xref  = [0  -0.09 0]';
67  end
68   if  R > f*5.5
69  xref  = [0  0.09  0.0001]';
70  end
71   if  R > f*6.5
72  xref  = [0.01  0.09  0.01]';
73  end
74
75  cbf  = -(((x(1,R)-cx)/rx)^2 + ((x(2,R)-cy)/ry)^2 + ((x(3,R)-cz)/rz)^2 -1);
76
77  %--- determine sigma ---%
78  epsilon  = 0.777777777777778;
79   if  cbf  <= -epsilon
80  sigma = 0;
81   elseif  (  (cbf  > -epsilon) && (cbf <  0)     )
82  sigma = -2*(cbf/epsilon).^3 - 3.*(cbf/epsilon).^2  + 1;
83  else
84  sigma = 1;
85  end
86
87  LgB(R,:) =  [(2/( taux*rx^2)) *(cx-x(1,R))  ...
88   (2/( tauy*ry^2)) *(cy-x(2,R))  ...
89   (2/( tauz*rz^2)) *(cz-x(3,R))];
90  LfB(1,R) = -2*( ((cx*x(1,R)-x(1,R)^2)/(taux*rx^2)) +  ...
91   ((cy*x(2,R)-x(2,R)^2)/(tauy*ry^2)) +  ...
92   ((cz*x(3,R)-x(3,R)^2)/(tauz*rz^2)));
93
94   utilde  (:, R) = -K*x(:,R) + Nbar*xref(:);
95
96  %--- Find safe controller ---%
97  threshold  = 0.000001;
98  %abs(LgB(R,:));
99
100  if  norm(LgB(R,:),2) >= 0.0000001
101 k0 (:, R) = -( (  LfB(1,R)  + sqrt (LfB(1,R)^2  ...
102 + kappa^2*LgB(R,:)*LgB(R,:)' )) /    (LgB(R,:)*LgB(R,:)')    ) *LgB(R,:) ';
103 else
104 k0 (:, R) = [0  0  0]';
105 disp(R);
106 end
107 %sigma = 0;
108 u0 (:, R) = sigma*k0(:,R)+(1-sigma)*utilde(:,R);
109
110 %--- extrapolate with forward euler ---%
111 xdot  = A*x (:, R)  + B*u0(:,R);
112 x (:, R+1) = xdot*Ts + x (:, R);
113
114 sig (1, R) = sigma;
115  xref_plot (1, R) = xref (1) ;
116  xref_plot (2, R) = xref (2) ;
117  xref_plot (3, R) = xref (3) ;
118 end
```

G.2 Safety Verification with SOSTOOLS

This appendix contains the MATLAB/SOSTOOLS safety verification (barrier certificate search) tested throughout the project, i.e.

- Safety verification of a first order system given a zero reference in subsection G.2.1.
- Safety verification of a first order system given a range of references in subsection G.2.2.
- Safety verification of a first order system in terms of the error state in subsection G.2.3.
- Safety verification of a second order system in terms of the error state in subsection G.2.4.

G.2.1 Safety Verification of First Order System with Zero Reference

This section contains the MATLAB/SOSTOOLS implementation of the barrier certificate search described in subsection 9.2.1. All plotting details are omitted in this section, but can be found in appendix J under the path `matlab_scripts/sostools/1storder_noRef.m`.

```
1  % 1D system WITHOUT REFERENCE
2  clear all; clc;
3
4  % Time constant from measurement
5  tau = 0.11;
6  % State-space matrices for first order system
7  A = -1/tau;
8  B = 1/tau;
9  K = place(A,B,[10*eig(A)]);
10
11 % Distance between safe and unsafe regions
12 delta = 1e-3;
13
14 % Minimum value of the barrier certificate on the set Xu
15 epsilon = 1e-3;
16
17 % Set upper and lower limits for the set intervals X, Xu and X0
18 Xmax = 0.1;
19 Xmin = -0.1;
20 Xumax = Xmax;
21 Xumin = 0.05;
22 X0max = Xumin-delta;
23 X0min = Xmin;
24
25 % Set degree of barier certificare and SOS polynomials
26 degB = [0,2:4];
27 degq = [0:4];
28
29 % ============================================
30 % Declare state variable
31 pvar x1
32
33 % Initialize the sum of squares program
34 prog = sosprogram(x1);
35
36 % Vector field dx/dt = fx (closed loop)
37 fx = (A-B*K)*x1;
38
39 % Declare the polynomial barrier function
40 zB = monomials(x1,degB);
```

```
41 [prog,Bar] = sospolyvar(prog,zB);
42
43 % ============================================
44 % Define space X in Rn
45 [a,b,c] = parabola(Xmin,Xmax); % get coefficients for parabola which is positive for x in [-0.1,0.1]
46 gX = a*x1^2+b*x1+c;
47 zX = monomials(x1,degq);
48 [prog,qX] = sossosvar(prog,zX);
49
50 prog = sosineq(prog,-diff(Bar,x1)*fx-gX*qX);
51
52 % Define space Xu in X
53 [a,b,c]=parabola(Xumin,Xumax);
54 gXu = a*x1^2+b*x1+c;
55 zXu = monomials(x1,degq);
56 [prog,qXu] = sossosvar(prog,zXu);
57
58 prog = sosineq(prog,Bar-epsilon-gXu*qXu);
59
60 % Define space X0 in X
61 [a,b,c]=parabola(X0min,X0max);
62 gX0 = a*x1^2+b*x1+c;
63 zX0 = monomials(x1,degq);
64 [prog,qX0] = sossosvar(prog,zX0);
65
66 prog = sosineq(prog,-Bar-gX0*qX0);
67
68 % ============================================
69 % Solve for barrier certificate
70 prog = sossolve(prog);
71 getB = sosgetsol(prog,Bar)
72
73 % Get coefficients for the remaining polynomials
74 getdBdx = diff(getB,x1)
75 getqXu1 = sosgetsol(prog,qXu);
76 getqX01 = sosgetsol(prog,qX0);
77 getqX1 = sosgetsol(prog,qX);
78
79 % Test if the inequalities are SOS
80 [Q,~,~] = findsos(getB-epsilon-gXu*getqXu1);
81 [Q2,~,~] = findsos(-getB-gX0*getqX01);
82 [Q3,~,~] = findsos(-getdBdx*fx-gX*getqX1);
```

G.2.2 Safety Verification of First Order System for Reference Interval

This section contains the MATLAB/SOSTOOLS implementation of the barrier certificate search described in subsection 9.2.2. All plotting details are omitted in this section, but can be found in appendix J under the path `matlab_scripts/sostools/1storder_withRef.m`.

```
1 % 1D system WITH REFERENCE INTERVAL
2 clear all ; clc;
3
4 % Time constant from measurement
5 tau = 0.11;
6 % State-space matrices for first order system
7 A = -1/tau;
8 B = 1/tau;
9 K = 0.2;
10 Nbar = K+1;
11
12 % Distance between safe and unsafe regions
```

```
13 delta = 0.015;
14
15 % Minimum value of the barrier certificate on the set Xu
16 epsilon = 1e-1;
17
18 % Set upper and lower limits for the set intervals X, Xu and X0
19 Xmin = -0.1;
20 Xmax = 0.1;
21 Xumin = 0.05;
22 Xumax = Xmax;
23 X0min = Xmin;
24 X0max = Xumin-delta;
25 rMin = -0.1;
26 rMax = 0.017;
27
28 % Set degree of barier certificare and SOS polynomials
29 degB = [0:6];
30 degq = [0:2];
31
32 % =============================================
33 % Control Barrier Function Search for 1D system
34 pvar x1 xref
35 xtilde = [x1; xref];
36
37 % Initialize the sum of squares program
38 prog = sosprogram(xtilde);
39
40 % Vector field dt/dx = fx (closed loop)
41 fx = [A-B*K B*Nbar; 0 0]*xtilde;
42
43 % Declare the polynomial barrier function
44 zB = monomials(xtilde,degB);
45 [prog,Bar] = sospolyvar(prog,zB);
46
47 % =============================================
48 % Define space X in Rn
49 [a,b,c] = parabola(Xmin,Xmax); % get coefficients for parabola which is positive for x in [-0.1,0.1]
50 gX1 = a*x1^2+b*x1+c;
51 zX1 = monomials(xtilde,degq);
52 [prog,qX1] = sossosvar(prog,zX1);
53
54 [a,b,c] = parabola(rMin,rMax);
55 gX2 = a*xref^2+b*xref+c;
56 zX2 = monomials(xtilde,degq);
57 [prog,qX2] = sossosvar(prog,zX2);
58
59 prog = sosineq(prog,-diff(Bar,x1)*fx-gX1*qX1 - gX2*qX2 );
60
61 % Define space Xu in X
62 [a,b,c]=parabola(Xumin,Xumax);
63 gXu1 = a*x1^2+b*x1+c;
64 zXu1 = monomials(xtilde,degq);
65 [prog,qXu1] = sossosvar(prog,zXu1);
66
67 [a,b,c]=parabola(rMin,rMax);
68 gXu2 = a*xref^2+b*xref+c;
69 zXu2 = monomials(xtilde,degq);
70 [prog,qXu2] = sossosvar(prog,zXu2);
71
72 prog = sosineq(prog,Bar-epsilon-gXu1*qXu1-gXu2*qXu2);
73
74 % Define space X0 in X
75 [a,b,c]=parabola(X0min,X0max);
76 gX01 = a*x1^2+b*x1+c;
```

```
77  zX01 = monomials(xtilde,degq);
78  [prog,qX01] = sossosvar(prog,zX01);
79
80  [a,b,c]=parabola(rMin,rMax);
81  gX02 = a*xref^2+b*xref+c;
82  zX02 = monomials(xtilde,degq);
83  [prog,qX02] = sossosvar(prog,zX02);
84
85  prog = sosineq(prog,-Bar-gX01*qX01-gX02*qX02);
86
87  % ============================================
88  % Solve for barrier certificate
89  prog = sossolve(prog);
90  getB = sosgetsol(prog,Bar)
91
92  % Get coefficients for the remaining polynomials
93  getdBdx = [diff(getB,x1) diff(getB,xref)]
94  getqXu1 = sosgetsol(prog,qXu1);
95  getqX01 = sosgetsol(prog,qX01);
96  getqX1 = sosgetsol(prog,qX1);
97  getqXu2 = sosgetsol(prog,qXu2);
98  getqX02 = sosgetsol(prog,qX02);
99  getqX2 = sosgetsol(prog,qX2);
100
101 % Test if the inequalities are SOS
102 [Q,~,~] = findsos(getB-epsilon-gXu1*getqXu1-gXu2*getqXu2);
103 [Q2,~,~] = findsos(-getB-gX01*getqX01-gX02*getqX02);
104 [Q3,~,~] = findsos(-getdBdx*fx-gX1*getqX1-gX2*getqX2);
```

G.2.3 Safety Verification of First Order Error State System

This section contains the MATLAB/SOSTOOLS implementation of the barrier certificate search described in subsection 9.2.3. All plotting details are omitted in this section, but can be found in appendix J under the path `matlab_scripts/sostools/1storder_error.m`.

```
1   % 1D first order system FOR ERROR STATE
2   clear all; clc;
3
4   % Time constant from measurement
5   tau = 0.11;
6   % State-space matrices for first order system
7   A = -1/tau;
8   B = 1/tau;
9   K = 0.2;
10  Nbar = K+1;
11
12  % Distance between safe and unsafe regions
13  delta = 4e-3;
14
15  % Minimum value of the barrier certificate on the set Xu
16  epsilon = 1e-2;
17
18  % Set upper and lower limits for the set intervals X, Xu and X0
19  Xmin = -0.03;
20  Xmax = 0.03;
21  Xumin = Xmin;
22  Xumax = -0.009;
23  X0min = Xumax+delta;
24  X0max = Xmax;
25
26  % Set degree of barier certificare and SOS polynomials
```

```
27 degB = [0:6];
28 degq = [0:4];
29
30 % ============================================
31 % Control Barrier Function Search for 1D system
32 pvar xerr
33
34 % Initialize the sum of squares program
35 prog = sosprogram(xerr);
36
37 % Vector field dt/dx = fx (closed loop)
38 fx = (A-B*K)*xerr;
39
40 % Declare the polynomial barrier function
41 zB = monomials(xerr,degB);
42 [prog,Bar] = sospolyvar(prog,zB);
43
44 % ============================================
45 % Define space X in Rn
46 [a,b,c] = parabola(Xmin,Xmax);
47 gX1 = a*xerr^2+b*xerr+c;
48 zX1 = monomials(xerr,degq);
49 [prog,qX1] = sossosvar(prog,zX1);
50
51 prog = sosineq(prog,-diff(Bar,xerr)*fx-gX1*qX1);
52
53 % Define space Xu in X
54 [a,b,c]=parabola(Xumin,Xumax);
55 gXu1 = a*xerr^2+b*xerr+c;
56 zXu1 = monomials(xerr,degq);
57 [prog,qXu1] = sossosvar(prog,zXu1);
58
59 prog = sosineq(prog,Bar-epsilon-gXu1*qXu1);
60
61 % Define space X0 in X
62 [a,b,c]=parabola(X0min,X0max);
63 gX01 = a*xerr^2+b*xerr+c;
64 zX01 = monomials(xerr,degq);
65 [prog,qX01] = sossosvar(prog,zX01);
66
67 prog = sosineq(prog,-Bar-gX01*qX01);
68
69 % ============================================
70 % Solve for barrier certificate
71 prog = sossolve(prog);
72 getB = sosgetsol(prog,Bar)
73
74 % Get coefficients for the remaining polynomials
75 getdBdx = diff(getB,xerr)
76 getqXu1 = sosgetsol(prog,qXu1);
77 getqX01 = sosgetsol(prog,qX01);
78 getqX1 = sosgetsol(prog,qX1);
79
80 % Test if the inequalities are SOS
81 [Q,~,~] = findsos(getB-epsilon-gXu1*getqXu1);
82 [Q2,~,~] = findsos(-getB-gX01*getqX01);
83 [Q3,~,~] = findsos(-getdBdx*fx-gX1*getqX1);
```

G.2.4 Safety Verification of Second Order Error State System

This section contains the MATLAB/SOSTOOLS implementation of the barrier certificate search described in section 9.3. All plotting details are omitted in this section, but can be found in appendix J under the path `matlab_scripts/sostools/2ndorder_error.m`.

```
% 1D second order system FOR ERROR STATE
clear all ; clc;

% Settling and rise time from measurement
ts = (28-9)*1/50; % 5 percent settling time
tr = 0.1; % rise time
wn = 1.8/tr; % natural frequency
zeta = -1/(wn*ts)*log(0.02); % damping ratio

% State-space matrices for second order system
A = [0 1; -wn^2 -2*zeta*wn];
B = [0 wn^2]';
K = acker(A,B,[-40 -50]);

% Distance between safe and unsafe regions
delta = 5e-3;

% Minimum value of the barrier certificate on the set Xu
epsilon = 5e-2;

% Set upper and lower limits for the set intervals X, Xu and X0
velMin = -0.5;
velMax = 0.5;
Xmin = -0.03;
Xmax = 0.03;
Xumin = Xmin;
Xumax = -0.008;
X0min = Xumax+delta;
X0max = Xmax;

% Set degree of barier certificare and SOS polynomials
degB = [0:4];
degq = [0:1];

% ============================================
% Control Barrier Function Search for 1D system
pvar xerr1 xerr2
xerr = [xerr1; xerr2];

% Initialize the sum of squares program
prog = sosprogram(xerr);

% Vector field dt/dx = fx (closed loop)
fx = (A-B*K)*xerr;

% Declare the polynomial barrier function
zB = monomials(xerr,degB);
[prog,Bar] = sospolyvar(prog,zB);

% ============================================
% Define space X in Rn
[a,b,c] = parabola(Xmin,Xmax);
gX1 = a*xerr1^2+b*xerr1+c;
zX1 = monomials(xerr,degq);
[prog,qX1] = sossosvar(prog,zX1);

[a,b,c] = parabola(velMin,velMax);
```

```
58  gX2 = a*xerr2^2+b*xerr2+c;
59  zX2 = monomials(xerr,degq);
60  [prog,qX2] = sossosvar(prog,zX2);
61
62  prog = sosineq(prog,-[diff(Bar,xerr1)  diff (Bar,xerr2)]* fx-gX1*qX1-gX2*qX2);
63
64  % Define space Xu in X
65  [a,b,c]=parabola(Xumin,Xumax);
66  gXu1 = a*xerr1^2+b*xerr1+c;
67  zXu1 = monomials(xerr,degq);
68  [prog,qXu1] = sossosvar(prog,zXu1);
69
70  [a,b,c]  = parabola(velMin,velMax);
71  gXu2 = a*xerr2^2+b*xerr2+c;
72  zXu2 = monomials(xerr,degq);
73  [prog,qXu2] = sossosvar(prog,zXu2);
74
75  prog = sosineq(prog,Bar-epsilon-gXu1*qXu1-gXu2*qXu2);
76
77  % Define space X0 in X
78  [a,b,c]=parabola(X0min,X0max);
79  gX01 = a*xerr1^2+b*xerr1+c;
80  zX01 = monomials(xerr,degq);
81  [prog,qX01] = sossosvar(prog,zX01);
82
83  [a,b,c]  = parabola(velMin,velMax);
84  gX02 = a*xerr2^2+b*xerr2+c;
85  zX02 = monomials(xerr,degq);
86  [prog,qX02] = sossosvar(prog,zX02);
87
88  prog = sosineq(prog,-Bar-gX01*qX01-gX02*qX02);
89
90  % ============================================
91  % Solve for  barrier   certificate
92  prog = sossolve(prog);
93  getB = sosgetsol(prog,Bar)
94
95  % Get coefficients  for  the  remaining polynomials
96  getdBdx = [ diff (getB,xerr1)   diff (getB,xerr2) ]
97  getqXu1 = sosgetsol(prog,qXu1);
98  getqX01 = sosgetsol(prog,qX01);
99  getqX1 = sosgetsol(prog,qX1);
100 getqXu2 = sosgetsol(prog,qXu2);
101 getqX02 = sosgetsol(prog,qX02);
102 getqX2 = sosgetsol(prog,qX2);
103
104 % Test  if  the  inequalities  are SOS
105 [Q,~,~] = findsos(getB-epsilon-gXu1*getqXu1-gXu2*getqXu2);
106 [Q2,~,~] = findsos(-getB-gX01*getqX01-gX02*getqX02);
107 [Q3,~,~] = findsos(-getdBdx*fx-gX1*getqX1-gX2*getqX2);
```

Appendix H

Da Vinci Implementation of Controllers

This appendix features the the source code for the files:

- `run_controllers.cpp` (this file includes the `main` C++ function and the safety controllers developed in chapter 4 (slide) and chapter 5 (beating heart)).
- `safe_3d.cpp` (this file implements the safety controller developed in chapter 6 in the 3D euclidean space).

Along with their associated `.h` files, they are both located at the Robotic Surgery Group - Aalborg University at github (*https://github.com/AalborgUniversity-RoboticSurgeryGroup/*) as the repository `gr1031`. Some additional test files are developed, such as `ik_gr1032.cpp` and `ik_gr1032.cpp`. These files are only used for demonstrations and testing and features nothing "new", thus not shown here, but can be found at github as well.

run_controllers.cpp

```
1  #include "ik_gr1032.h"
2  #include "demo_gr1032.h"
3  #include "safe_3d.h"
4
5  /*** include libraries ***/
6  #include <string>
7  #include <vector>
8  #include <iostream>
9  #include <stdio.h>
10 #include <ros/ros.h>
11 #include <std_msgs/Header.h>
12 #include <std_msgs/Float64.h>
13 #include <time.h>
14 #include "ros/ros.h"
15 #include "std_msgs/String.h"
16 #include <sensor_msgs/JointState.h>
17 #include <math.h>
18 #include <fstream>
19 #include <Eigen/Dense>
20
21 /*** define macros ***/
22 #define K 0.1
23 #define Nbar 1.1
24 #define kappa 1
25 #define a 1.7778
26 #define b 0.0889
27 #define c -0.0089
28 #define epsilon 0.002488888888889
29 #define tau 0.1
30 #define N_samples 100
31 #define a2 0.07500
```

```
32 #define b2 −4
33 #define zeta 0.55
34 #define x10 0.0250
35 #define wn 17
36 #define c1 1
37 #define c2 −1
38
39 /*** synopsises ***/
40 int slide_safety_controller ( int model);
41 int write_meas_to_files();
42 int slide_angles() ;
43 int demo();
44 int beating_heart();
45 int write_sine_data_to_file () ;
46 int ik_angles() ;
47
48 /*** global doubles ***/
49 double x1;
50 double x1_hat;
51 double x2_hat;
52 double xref = 0.00;
53 double LgB;
54 double LfB;
55 double k0;
56 double utilde ;
57 double u = 0;
58 double sigma = 0;
59 double cbf;
60 double Ac1 = −1/tau;
61 double Gamma1 = 1.095169439874664;
62 double Bc1 = 1/tau;
63 double Phi1 = 0.095169439874664;
64 double N;
65 double P;
66 double err;
67 double x_inst_slide;
68 double x_inst_roll ;
69 double x_inst_pitch;
70 double x_jaw_right;
71 double x_jaw_left;
72 double x_hand_roll;
73 double x_hand_pitch;
74
75 /*** global vectors ***/
76 std::vector<double> x1_vec;
77 std::vector<double> xref_vec;
78 std::vector<double> sigma_vec;
79 std::vector<double> u_vec;
80 std::vector<double> LgB_vec;
81 std::vector<double> LfB_vec;
82 std::vector<double> err_vec;
83 std::vector<double> x1_hat_vec;
84 std::vector<double> x2_hat_vec;
85 std::vector<double> dur_vec;
86 std::vector<double> x1_beat_vec;
87 std::vector<double> dref_vec;
88
89 /*** global matrices ***/
90 Eigen::MatrixXd Gamma(2,2);
91 Eigen::MatrixXd Phi(2,1);
92 Eigen::MatrixXd C(1,2);
93 Eigen::MatrixXd Kd(1,2);
94 Eigen::MatrixXd Ld(2,1);
95 Eigen::MatrixXd x_hat(2,1);
```

```
96  Eigen::MatrixXd x1_eigen(1,1);
97  Eigen::MatrixXd eigen_temp(1,1);
98  Eigen::MatrixXd M(2,1);
99
100 /*** beating heart matrices ***/
101 Eigen::MatrixXd A_beat(4,4);
102 Eigen::MatrixXd B_beat(4,1);
103 Eigen::MatrixXd K_beat(1,4);
104 Eigen::MatrixXd x_beat(4,1);
105
106 /*** global integers ***/
107 int ref_counter = 0;
108
109 /*** callback function to read position sensor ***/
110 void joint_states_callback(const sensor_msgs::JointState::ConstPtr& msg)
111     {
112         // ROS_INFO("slide postion: %f", msg->position[6]);
113         // ROS_INFO("name: %s", msg->name[6].c_str());
114         x1 = msg->position[6];
115         x_inst_slide = x1;
116         x_inst_roll = msg->position[5];
117         x_inst_pitch = msg->position[4];
118         x_jaw_right = msg->position[3];
119         x_jaw_left = msg->position[2];
120         x_hand_roll = msg->position[1];
121         x_hand_pitch = msg->position[0];
122     }
123
124 /*** timer class ***/
125 class timer {
126     private:
127         long double begTime;
128     public:
129         void start() {
130             begTime = clock();
131         }
132 };
133
134 int main(int argc, char **argv) {
135     ros::init(argc, argv, "run_controllers", ros::init_options::AnonymousName);
136
137     /*** define Gamma for slide safety controller ***/
138     Gamma(0,0) = 0.9864;  Gamma(0,1) = 0.0091;
139     Gamma(1,0) = -2.6232; Gamma(1,1) = 0.8167;
140     /*** define Phi for slide safety controller ***/
141     Phi(0,0) = 0.0136;
142     Phi(1,0) = 2.6232;
143     /*** define the output matrix C for slide safety controller ***/
144     C(0,0) = 1;           C(0,1) = 0;
145     /*** define the feedback Kd for slide safety controller ***/
146     Kd(0,0) = 0.25;    Kd(0,1) = -0.03;
147     /*** define the observer gain Ld for slide safety controller ***/
148     Ld(0,0) = -0.25;
149     Ld(1,0) = -0.02;
150     x1_eigen(0,0) = 0;
151     /*** define gains for slide safety controller ***/
152     N = 58;
153     M(0,0) = 0.78;
154     M(1,0) = 152.31;
155     P = 0.0129;
156
157     /*** initialize x_hat ***/
158     x_hat(0,0) = x1;
159     x_hat(1,0) = 0;
```

```
160
161    /*** welcome screen ***/
162    std::cout << "Starting program..." << std::endl;
163    std::cout << "\n ****************************************************** " << std::endl;
164    std::cout << "The following options are avaiable:" << std::endl;
165    std::cout << "--------------------------------------------------------" << std::endl;
166    std::cout << "press 'a' to run slide safety controller" << std::endl;
167    std::cout << "press 'b' to specify custom joint angles (FK mode)" << std::endl;
168    std::cout << "press 'c' to run demo" << std::endl;
169    std::cout << "press 'd' to run beating heart controller" << std::endl;
170    std::cout << "press 'e' to specify custom 3D angles (IK mode)" << std::endl;
171    std::cout << "press 'f' to run 3D safety controller" << std::endl;
172    std::cout << " ****************************************************** " << std::endl;
173
174    char choice;
175    std::cin >> choice;
176
177    /*** initialize callback function for sensor measurements ***/
178    ros::NodeHandle n;
179    ros::Subscriber sub = n.subscribe("joint_states", 1000, joint_states_callback);
180
181    while (choice != 'q') {
182        if (choice == 'a'){
183            /*** remove old files ***/
184            remove("slide_data.txt");
185            remove("slide_ref.txt");
186            remove("control_signal.txt");
187            remove("sigma.txt");
188            remove("LgB.txt");
189            remove("LfB.txt");
190            remove("err.txt");
191            remove("x1_hat.txt");
192            remove("x2_hat.txt");
193            remove("dur.txt");
194
195            /*** ask for underlying approximation ***/
196            int model;
197            std::cout << "model order approximation? (1 or 2)" << std::endl;
198            std::cin >> model;
199
200            /*** run controller ***/
201            slide_safety_controller(model);
202
203            return 0;
204        }
205        else if (choice == 'b') {
206            slide_angles();
207            return 0;
208        }
209        else if (choice == 'c') {
210            demo();
211            return 0;
212        }
213        else if (choice == 'd') {
214            remove("x1_beat.txt");
215            remove("sigma.txt");
216            beating_heart();
217            return 0;
218        }
219        else if (choice == 'e') {
220            ik_pos();
221            return 0;
222        }
223        else if (choice == 'f') {
```

```
224             remove("x.txt");
225             remove("y.txt");
226             remove("z.txt");
227             remove("x_ref.txt");
228             remove("y_ref.txt");
229             remove("z_ref.txt");
230             remove("sigma_3d.txt");
231             remove("exe_3d.txt");
232             safe_3d();
233             return 0;
234         }
235         else {
236             std::cout << "please press one of the given letters or press 'q' to quit" << std::endl;
237             std::cin >> choice;
238         }
239     }
240     return 0;
241 }
242
243 /*** slide safety controller ***/
244 int slide_safety_controller (int model) {
245     std::cout << "Entered safety controller for slide position!" << std::endl;
246     std::cout << "running safety controller " << std::endl;
247
248     /*** prepare to publish control signal ***/
249     ros::NodeHandle node;
250     ros::Publisher setpoints_pub_pitch = node.advertise<std_msgs::Float64>("pitch_command", 1);
251     ros::Publisher setpoints_pub_slide = node.advertise<std_msgs::Float64>("slide_command", 1);
252
253     /*** read setpoints from file ***/
254     std::vector<double> xrefs (200);
255     int i = 0;
256     int iter = 0;
257     std::fstream myfile("references.txt", std::ios_base::in);
258     while (myfile >> xrefs[i])
259     {
260         printf ("%f ", xrefs[i]);
261         i += 1;
262     }
263     int N_xrefs = i;
264     std::cout << N_xrefs << " reference points provided" << std::endl;
265
266     long double Ts = 0.01;
267     /*** run slide controller ***/
268     while(true) {
269
270         /*** subscribe to topics with 100 Hz ***/
271         ros::Rate r(100);
272
273         /*** prepare real time processing ***/
274         timer t;
275         t.start();
276
277         while(true) {
278             if (t.elapsedTime() >= Ts) {
279
280                 /*** read sensor ***/
281                 ros::spinOnce();
282
283                 /*** determine setpoint ***/
284                 if (iter % N_samples == 0) {
285                     xref = xrefs[ref_counter];
286                     ref_counter += 1;
287                 }
```

```
288
289         std::cout << "x1 = " << x1 << std::endl;
290         std::cout << "xref = " << xref << std::endl;
291
292         /*** start timer to meassure execution time ***/
293         std::clock_t start;
294         double dur;
295         start = std::clock();
296
297
298         /*** barrier function based on first order approximation ***/
299         cbf = a*x1*x1 + b*x1 + c;
300
301         if (model == 1) {
302             /*** calculate linear control output ***/
303             utilde = Nbar*xref - K*x1;
304
305             /*** lie derivatives based on first order approximation ***/
306             LgB = 2*a*x1*(1/tau) + b*(1/tau);
307             LfB = -2*(1/tau)*a*pow(x1,2) - (1/tau)*b*x1;
308         }
309         else if (model == 2) {
310             /*** calculate linear control output ***/
311             eigen_temp = Kd*x_hat;
312             double temp_ = (double) eigen_temp(0,0);
313             utilde = temp_ + N*xref*P;
314             x1_eigen(0,0) = x1;
315
316             /*** calculate error ***/
317             double err = (double) x1_eigen(0,0) - x_hat(0,0);
318             std::cout << "error = " << err << std::endl;
319
320             /*** calculate x_hat(k+1) ***/
321             x_hat = Gamma*x_hat + Phi*Kd*x_hat + Ld*( C*x_hat - x1_eigen ) + M*xref*P;
322
323             /*** calculate lie derivatives ***/
324             x1_hat = x_hat(0,0);
325             x2_hat = x_hat(1,0);
326             LgB = x2_hat * (2*c1*pow(wn,2) / pow(b2,2) );
327             LfB = x2_hat * ( (c1*(2*x1_hat+2*x10)) / (pow(a2,2)) - (2*c1*(2*x1_hat*pow(wn,2) + 2*x2_hat*zeta*wn)) / (pow(b2,2)));
328             std::cout << "x_hat = \n" << x_hat << std::endl;
329         }
330
331         /*** safe control law ***/
332         int delta = 0.00001;
333         if (abs(LgB) > delta) {
334             k0 = -LgB*( LfB + sqrt( LfB*LfB + kappa*kappa + LgB*LgB ) )/( LgB*LgB );
335         }
336         else {
337             k0 = 0;
338             std::cout << "k0 = 0!!! " << std::endl;
339         }
340
341         /*** calculate sigma ***/
342         if ((cbf < -epsilon) || (cbf == -epsilon)) {
343             sigma = 0;
344         }
345         else if ((cbf > -epsilon) && (cbf < 0)) {
346             sigma = 3*(-2*pow(cbf/epsilon,3) - 3*pow(cbf/epsilon,2 ) + 1);
347         }
348         else {
349             sigma = 1;
350         }
351         if (sigma > 1) {
```

```
352                 sigma = 1;
353             }
354 
355             std :: cout << "sigma = " << sigma << std::endl;
356 
357             /*** control law ***/
358             u = sigma*k0 + (1 − sigma)*utilde;
359 
360             /*** slide physical limits ***/
361             if (u > 0.099)
362                 u = 0.099;
363             else if (u < −0.099)
364                 u = −0.099;
365 
366             /*** send control signals to robot ***/
367           std_msgs::Float64 zero_msg;
368           zero_msg.data = 0.7;
369             setpoints_pub_pitch.publish(zero_msg);
370           std_msgs::Float64 u_msg;
371           u_msg.data = u;
372             setpoints_pub_slide.publish(u_msg);
373 
374             /*** provide some user information ***/
375             iter += 1;
376             std :: cout << iter << "  iterations " << std :: endl;
377             std :: cout << "model order = " << model << std::endl;
378 
379             /*** write slide measurements to file after inital transients ***/
380             if ( iter > 0) {
381                 x1_vec.push_back(x1);
382                 xref_vec.push_back(xref);
383                 sigma_vec.push_back(sigma);
384                 u_vec.push_back(u);
385                 LgB_vec.push_back(LgB);
386                 LfB_vec.push_back(LfB);
387                 if (model == 2) {
388                     err_vec.push_back(err);
389                     x1_hat_vec.push_back(x1_hat);
390                     x2_hat_vec.push_back(x2_hat);
391                 }
392                 dur_vec.push_back(dur);
393             }
394 
395             /*** check execution time ***/
396             dur = ( std :: clock () − start ) / (double) CLOCKS_PER_SEC;
397             std :: cout << "execution time = " << dur << std :: endl;
398             std :: cout << "\n";
399 
400             if ( iter > N_xrefs*N_samples) {
401                 write_meas_to_files();
402                 ros :: shutdown();
403                 return 0;
404             }
405 
406             break;
407         }
408       else {
409           /*** do some other stuff if necessary***/
410       }
411   }
412 }
413     return 0;
414 }
415 
```

```
416 int beating_heart() {
417     std::cout << "beating heart controller is entered" << std::endl;
418 
419     /*** define local variables ***/
420     double wh = 5.7119866;
421     double Kbar_beat = 0.1;
422     double Nbar_beat = 1.1;
423 
424     /*** define augmented feedback vector ***/
425     K_beat(0,0) = -Kbar_beat; K_beat(0,1) = Nbar_beat; K_beat(0,2) = 0;  K_beat(0,3) = Nbar_beat;
426 
427     /*** prepare to publish control signal ***/
428     ros::NodeHandle node;
429     ros::Publisher setpoints_pub_pitch = node.advertise<std_msgs::Float64>("pitch_command", 1);
430     ros::Publisher setpoints_pub_slide = node.advertise<std_msgs::Float64>("slide_command", 1);
431 
432     /*** subscribe to topics with 100 Hz ***/
433     ros::Rate r(100);
434 
435    /***  initial  conditions ***/
436     x_beat(0,0) = 0.05;
437     x_beat(1,0) = 0.01;
438     x_beat(2,0) = 0;
439     x_beat(3,0) = 0.03;
440 
441     int iter = 0;
442     double cbf_beat;
443     double epsilon_beat = 0.015;
444     long double Ts = 0.01;
445     while(true) {
446         /*** prepare real time processing ***/
447         timer t;
448         t.start();
449         while(true) {
450             if (t.elapsedTime() >= Ts) {
451                 /*** read sensor ***/
452                 ros::spinOnce();
453 
454                 /*** update state vector ***/
455                 x_beat(0,0) = x1;
456                 x_beat(1,0) = 0.01*cos(wh*Ts*iter);
457                 x_beat(2,0) = 0.01*sin(wh*Ts*iter);
458 
459                 if ((iter > 500) && (iter < 700)) {
460                     /*** give unsafe distance ***/
461                     x_beat(3,0) = -0.02;
462                 }
463                 else if (iter > 1000) {
464                     x_beat(3,0) = 0.025;
465                 }
466                 else {
467                     /*** give safe distance ***/
468                     x_beat(3,0) = 0.04;
469                 }
470 
471                 /*** control barrier function ***/
472                 cbf_beat = x_beat(1,0) - x_beat(0,0);
473 
474                 /*** calculate linear control output ***/
475                 eigen_temp = K_beat*x_beat;
476                 utilde = (double) eigen_temp(0,0);
477 
478                 /*** lie derivatives ***/
479                 LgB = -1/tau;
```

```
480                 LfB = wh*x_beat(2,0) + x_beat(0,0)/tau;
481
482                 /*** safe control law ***/
483                 int delta = 0.00001;
484                 if (abs(LgB) > delta) {
485                     k0 = −LgB*(LfB + sqrt(LfB*LfB + kappa*kappa + LgB*LgB))/(LgB*LgB);
486                 }
487                 else {
488                     k0 = 0;
489                     std::cout << "k0 = 0" << std::endl;
490                 }
491
492                 /*** calculate sigma ***/
493                 if ((cbf_beat < −epsilon_beat) || (cbf_beat == −epsilon_beat)) {
494                     sigma = 0;
495                 }
496                 else if ((cbf_beat > −epsilon_beat) && (cbf_beat < 0)) {
497                     sigma = (−2*pow(cbf_beat/epsilon_beat,3) − 3*pow(cbf_beat/epsilon_beat,2) + 1);
498                 }
499                 else {
500                     sigma = 1;
501                 }
502
503                 /*** control law ***/
504                 u = sigma*k0 + (1 − sigma)*utilde;
505
506                 /*** slide physical limits ***/
507                 if (u > 0.099)
508                      u = 0.099;
509                 else if (u < −0.099)
510                      u = −0.099;
511
512                /*** make sure to start robot in a safe area ***/
513                 if ( iter < 50 ) {
514                     u = 0.08;
515                 }
516
517                 /*** send control signals to robot ***/
518               std_msgs::Float64 zero_msg;
519               zero_msg.data = 0.0;
520                 setpoints_pub_pitch.publish(zero_msg);
521               std_msgs::Float64 u_msg;
522               u_msg.data = u;
523                 setpoints_pub_slide.publish(u_msg);
524
525                 /*** give some user information ***/
526                 std::cout << "xh1 = " << x_beat(1,0) << std::endl;
527                 std::cout << "x1 = " << x1 << std::endl;
528                 std::cout << "sigma = " << sigma << std::endl;
529                 std::cout << "u = " << u << std::endl;
530                 std::cout << " iter = " << iter << std::endl;
531                 iter += 1;
532
533                 /*** save measurements ***/
534                 if ( iter > 0) {
535                     x1_beat_vec.push_back(x1);
536                     sigma_vec.push_back(sigma);
537                     dref_vec.push_back(x_beat(3,0));
538                 }
539
540                 /*** write measurements to file and end section ***/
541                 if ( iter == 1400) {
542                     write_sine_data_to_file () ;
543                     return 0;
```

```
544                 }
545                 break;
546             }
547             else {
548                 /*** do some other stuff if necessary***/
549             }
550         }
551     }
552     return 0;
553 }
554 
555 int write_sine_data_to_file() {
556     /*** print position trajectory to file ***/
557     std::ofstream f1;
558     f1.open("x1_beat.txt", std::ios::out | std::ios::app | std::ios::binary);
559     for (int i = 0; i < x1_beat_vec.size(); i += 1) {
560         f1 << x1_beat_vec[i] << std::endl;
561     }
562     f1.close();
563     /*** print reference distance to file ***/
564     std::ofstream f2;
565     f2.open("dref.txt", std::ios::out | std::ios::app | std::ios::binary);
566     for (int i = 0; i < dref_vec.size(); i += 1) {
567         f2 << dref_vec[i] << std::endl;
568     }
569     f2.close();
570     /*** print sigma values to file ***/
571     std::ofstream f3;
572     f3.open("sigma.txt", std::ios::out | std::ios::app | std::ios::binary);
573     for (int i = 0; i < sigma_vec.size(); i += 1) {
574         f3 << sigma_vec[i] << std::endl;
575     }
576     f3.close();
577 }
578 
579 int write_meas_to_files() {
580     /*** print position trajectory to file ***/
581     std::ofstream f1;
582     f1.open("slide_data.txt", std::ios::out | std::ios::app | std::ios::binary);
583     for (int i = 0; i < x1_vec.size(); i += 1) {
584         f1 << x1_vec[i] << std::endl;
585     }
586     f1.close();
587     /*** print position references to file ***/
588     std::ofstream f2;
589     f2.open("slide_ref.txt", std::ios::out | std::ios::app | std::ios::binary);
590     for (int i = 0; i < xref_vec.size(); i += 1) {
591         f2 << xref_vec[i] << std::endl;
592     }
593     f2.close();
594     /*** print sigma values to file ***/
595     std::ofstream f3;
596     f3.open("sigma.txt", std::ios::out | std::ios::app | std::ios::binary);
597     for (int i = 0; i < sigma_vec.size(); i += 1) {
598         f3 << sigma_vec[i] << std::endl;
599     }
600     f3.close();
601     /*** print control signals to file ***/
602     std::ofstream f4;
603     f4.open("control_signal.txt", std::ios::out | std::ios::app | std::ios::binary);
604     for (int i = 0; i < u_vec.size(); i += 1) {
605         f4 << u_vec[i] << std::endl;
606     }
607     f4.close();
```

```
608      /*** print LgB to file ***/
609      std::ofstream f5;
610      f5.open("LgB.txt", std::ios::out | std::ios::app | std::ios::binary);
611      for (int i = 0; i < LgB_vec.size(); i += 1) {
612          f5 << LgB_vec[i] << std::endl;
613      }
614      f5.close();
615      /*** print LfB to file ***/
616      std::ofstream f6;
617      f6.open("LfB.txt", std::ios::out | std::ios::app | std::ios::binary);
618      for (int i = 0; i < LfB_vec.size(); i += 1) {
619          f6 << LfB_vec[i] << std::endl;
620      }
621      f6.close();
622      /*** print error to file ***/
623      std::ofstream f7;
624      f7.open("err.txt", std::ios::out | std::ios::app | std::ios::binary);
625      for (int i = 0; i < err_vec.size(); i += 1) {
626          f7 << err_vec[i] << std::endl;
627      }
628      f7.close();
629      /*** print estimated position to file ***/
630      std::ofstream f8;
631      f8.open("x1_hat.txt", std::ios::out | std::ios::app | std::ios::binary);
632      for (int i = 0; i < err_vec.size(); i += 1) {
633          f8 << x1_hat_vec[i] << std::endl;
634      }
635      f8.close();
636      /*** print estimated velocity to file ***/
637      std::ofstream f9;
638      f9.open("x2_hat.txt", std::ios::out | std::ios::app | std::ios::binary);
639      for (int i = 0; i < err_vec.size(); i += 1) {
640          f9 << x2_hat_vec[i] << std::endl;
641      }
642      f7.close();
643      /*** print duration to file ***/
644      std::ofstream f10;
645      f10.open("dur.txt", std::ios::out | std::ios::app | std::ios::binary);
646          for (int i = 0; i < dur_vec.size(); i += 1) {
647      f10 << dur_vec[i] << std::endl;
648      }
649      f10.close();
650  }
651
652  int slide_angles() {
653      /*** prepare to publish control signal ***/
654      ros::NodeHandle node;
655      ros::Publisher setpoints_pub_pitch = node.advertise<std_msgs::Float64>("pitch_command", 1);
656      ros::Publisher setpoints_pub_slide = node.advertise<std_msgs::Float64>("slide_command", 1);
657      ros::Publisher setpoints_pub_roll = node.advertise<std_msgs::Float64>("roll_command", 1);
658      ros::Publisher setpoints_pub_inst_roll = node.advertise<std_msgs::Float64>("inst_roll_command", 1);
659      ros::Publisher setpoints_pub_inst_pitch = node.advertise<std_msgs::Float64>("inst_pitch_command", 1);
660      ros::Publisher setpoints_pub_inst_jaw_right = node.advertise<std_msgs::Float64>("inst_jaw_right_command", 1);
661
662      while(1) {
663          std::cout << "Type values... (type 9 to exit)" << std::endl;
664
665          std::vector<std::string> names;
666          names.push_back("p4_hand_pitch");
667          names.push_back("p4_hand_roll");
668          names.push_back("p4_instrument_slide");
669          names.push_back("p4_instrument_roll");
670          names.push_back("p4_instrument_pitch");
671          names.push_back("p4_instrument_jaw_right");
```

```
672            names.push_back("p4_instrument_jaw_left");
673
674            double input;
675            std::vector<double> control_signals(7,0.0);
676            for (int i = 0; i < names.size()-1; i += 1) {
677                std::cout << names.at(i) << ":" << std::endl;
678                std::cin >> input;
679                if (input == 9) {
680                    return 0;
681                }
682                control_signals.at(i) = input;
683            }
684
685            /*** load p4_hand_pith ***/
686            std_msgs::Float64 u_msg_0;
687            u_msg_0.data = control_signals.at(0);
688
689            /*** load p4_hand_roll ***/
690            std_msgs::Float64 u_msg_1;
691            u_msg_1.data = control_signals.at(1);
692
693            /*** load p4_instrument_slide ***/
694            std_msgs::Float64 u_msg_2;
695            u_msg_2.data = control_signals.at(2);
696
697            /*** load p4_instrument_roll ***/
698            std_msgs::Float64 u_msg_3;
699            u_msg_3.data = control_signals.at(3);
700
701            /*** load p4_instrument_pitch ***/
702            std_msgs::Float64 u_msg_4;
703            u_msg_4.data = control_signals.at(4);
704
705            /*** load p4_instrument_jaw_right ***/
706            std_msgs::Float64 u_msg_5;
707            u_msg_5.data = control_signals.at(5);
708
709            std::cout << "hand_pitch = " << u_msg_0.data << std::endl;
710            std::cout << "hand_roll = " << u_msg_1.data << std::endl;
711            std::cout << "inst_slide = " << u_msg_2.data << std::endl;
712            std::cout << " inst_roll  = " << u_msg_3.data << std::endl;
713            std::cout << "inst_pitch = " << u_msg_4.data << std::endl;
714            std::cout << "inst_jaw_right = " << u_msg_5.data << std::endl;
715
716            /*** publish setpoints ***/
717            setpoints_pub_pitch.publish(u_msg_0);
718            setpoints_pub_roll.publish(u_msg_1);
719            setpoints_pub_slide.publish(u_msg_2);
720            setpoints_pub_inst_roll.publish(u_msg_3);
721            setpoints_pub_inst_pitch.publish(u_msg_4);
722            setpoints_pub_inst_jaw_right.publish(u_msg_5);
723
724            std::cout << "Done!" << std::endl;
725        }
726        return 0;
727    }
728
729    int demo() {
730        std::cout << "running demo.." << std::endl;
731        demo_func();
732        return 0;
733    }
```

safe_3d.cpp

```cpp
#include "safe_3d.h"

/*** include libraries ***/
#include <string>
#include <ros/ros.h>
#include <kdl_parser/kdl_parser.hpp>
#include <kdl/chain.hpp>
#include <kdl/chainfksolver.hpp>
#include <kdl/chainfksolverpos_recursive.hpp>
#include <kdl/chainiksolverpos_nr.hpp>
#include <kdl/chainiksolvervel_pinv.hpp>
#include <kdl/frames_io.hpp>
#include <stdio.h>
#include <iostream>
#include <vector>
#include <iostream>
#include <stdio.h>
#include <ros/ros.h>
#include <std_msgs/Header.h>
#include <std_msgs/Float64.h>
#include <time.h>
#include "std_msgs/String.h"
#include <sensor_msgs/JointState.h>
#include <math.h>
#include <fstream>
#include <Eigen/Dense>

#define pi 3.14149
#define epsilon 0.7777778
#define cx 0
#define cy 0
#define cz 0
#define rx 0.03
#define ry 0.06
#define rz 0.03
#define taux 0.110
#define tauy 0.110
#define tauz 0.110
#define kappa 0.05

int compute_fk_chain();
int write_meas_to_files_3d();

int i_ref;
int N_iter = 20;

long int iter_ref = 0;

double x_inst_slide;
double x_inst_roll;
double x_inst_pitch;
double x_jaw_right;
double x_jaw_left;
double x_hand_roll;
double x_hand_pitch;
double x1;
double x_cart_meas = 0.6;
double y_cart_meas = 0.0;
double z_cart_meas = 0.0;

std::vector<double> x_ref_vec;
std::vector<double> y_ref_vec;
```

```
63  std::vector<double> z_ref_vec;
64  std::vector<double> x3d_x_vec;
65  std::vector<double> x3d_y_vec;
66  std::vector<double> x3d_z_vec;
67  std::vector<double> x3d_x_ref_vec;
68  std::vector<double> x3d_y_ref_vec;
69  std::vector<double> x3d_z_ref_vec;
70  std::vector<double> sigma3d_vec;
71  std::vector<double> exe_time_vec;
72
73  Eigen::MatrixXd u_3d(3,1);
74  Eigen::MatrixXd utilde_3d(3,1);
75  Eigen::MatrixXd x(3,1);
76  Eigen::MatrixXd xref_3d(3,1);
77  Eigen::MatrixXd K(3,3);
78  Eigen::MatrixXd Nbar(3,3);
79  Eigen::MatrixXd LgB_3d(1,3);
80  Eigen::MatrixXd LfB_3d(1,1);
81  Eigen::MatrixXd sigma(1,1);
82  Eigen::MatrixXd k0(3,1);
83  Eigen::MatrixXd cbf(1,1);
84  Eigen::MatrixXd temp(1,1);
85
86  class timer {
87      private:
88          long double begTime;
89      public:
90          void start() {
91              begTime = clock();
92          }
93  };
94
95  int safe_3d() {
96      /*** static matrices ***/
97      double gain = 0.02;
98      K(0,0) = gain;    K(0,1) = 0.00;    K(0,2) = 0.00;
99      K(1,0) = 0.00;    K(1,1) = gain;    K(1,2) = 0.00;
100     K(2,0) = 0.00;    K(2,1) = 0.00;    K(2,2) = gain;
101
102     Nbar(0,0) = 1 + gain;   Nbar(0,1) = 0.00;       Nbar(0,2) = 0.00;
103     Nbar(1,0) = 0.00;       Nbar(1,1) = 1 + gain;   Nbar(1,2) = 0.00;
104     Nbar(2,0) = 0.00;       Nbar(2,1) = 0.00;       Nbar(2,2) = 1 + gain;
105
106     /*** initialize variable matrices ***/
107     xref_3d(0,0) = 0.06;
108     xref_3d(1,0) = 0.00;
109     xref_3d(2,0) = 0.00;
110
111     x(0,0) = 0.06;
112     x(1,0) = 0.00;
113     x(2,0) = 0.00;
114
115     u_3d(0,0) = 0.00;
116     u_3d(1,0) = 0.00;
117     u_3d(2,0) = 0.00;
118
119     utilde_3d(0,0) = 0.00;
120     utilde_3d(1,0) = 0.00;
121     utilde_3d(2,0) = 0.00;
122
123     std::cout << "K = \n" << K << std::endl;
124     std::cout << "Nbar = \n" << Nbar << std::endl;
125     std::cout << "x = \n" << x << std::endl;
126     std::cout << "xref_3d = \n" << xref_3d << std::endl;
```

```
127    std::cout << "utilde_3d = \n" << utilde_3d << std::endl;
128    std::cout << "u_3d = \n" << u_3d << std::endl;
129
130    /*** prepare real time processing ***/
131    timer t;
132    t.start();
133
134    std::cout << "Inverse Kinematic Mode!" << std::endl;
135
136    ros::NodeHandle node;
137    KDL::Tree my_tree;
138    std::string robot_desc_string;
139    node.param("robot_description", robot_desc_string, std::string());
140    if (!kdl_parser::treeFromString(robot_desc_string, my_tree))
141    {
142       ROS_ERROR("Failed to construct kdl tree");
143    }
144
145    /*** use modified geometry ***/
146    KDL::Chain my_chain;
147    std::string root_link("p4_rcm_base");
148    std::string tip_link("needle_driver_jawbone_right");
149    if (!my_tree.getChain(root_link, tip_link, my_chain))
150    {
151        ROS_ERROR("Failed to get chain from tree");
152    }
153
154    for (unsigned int i = 0; i < my_chain.getNrOfSegments(); ++i)
155    {
156        std::cout << my_chain.getSegment(i).getName() << "(" << my_chain.getSegment(i).getJoint().getName() << ")" << std::endl;
157    }
158
159    // Create solver based on kinematic chain
160    KDL::ChainFkSolverPos_recursive fksolver(my_chain);
161    KDL::ChainIkSolverVel_pinv iksolverv(my_chain);
162    KDL::ChainIkSolverPos_NR iksolver = KDL::ChainIkSolverPos_NR(my_chain,fksolver,iksolverv,100,1e-6);
163
164    KDL::JntArray q(my_chain.getNrOfJoints());
165    KDL::JntArray q_init(my_chain.getNrOfJoints());
166
167    /*** prepare to publish control signal ***/
168    ros::NodeHandle node_pub;
169    ros::Publisher setpoints_pub_pitch = node_pub.advertise<std_msgs::Float64>("pitch_command", 1);
170    ros::Publisher setpoints_pub_slide = node_pub.advertise<std_msgs::Float64>("slide_command", 1);
171    ros::Publisher setpoints_pub_roll  = node_pub.advertise<std_msgs::Float64>("roll_command", 1);
172    ros::Publisher setpoints_pub_inst_roll = node_pub.advertise<std_msgs::Float64>("inst_roll_command", 1);
173    ros::Publisher setpoints_pub_inst_pitch = node_pub.advertise<std_msgs::Float64>("inst_pitch_command", 1);
174    ros::Publisher setpoints_pub_inst_jaw_right = node_pub.advertise<std_msgs::Float64>("inst_jaw_right_command", 1);
175
176    /*** read 3d references ***/
177    std::vector<double> ref_vector;
178    double str_ref;
179    std::ifstream fin("references_3d_mod.txt");
180    while (fin >> str_ref)
181    {
182        ref_vector.push_back(str_ref);
183    }
184    fin.close();
185
186    /*** read references from file ***/
187    int j = 0;
188    for (int i = 0; i < ref_vector.size(); ++i) {
189        if (j == 0) {
190            x_ref_vec.push_back(ref_vector.at(i));
```

```
191             j += 1;
192         }
193         else if (j == 1) {
194             y_ref_vec.push_back(ref_vector.at(i));
195             j += 1;
196         }
197         else if (j == 2) {
198             z_ref_vec.push_back(ref_vector.at(i));
199             j = 0;
200         }
201     }
202 
203     /*** print references ***/
204     std::cout << "\n";
205     std::cout << "x references: " << std::endl;
206     for (int i = 0; i < x_ref_vec.size(); ++i)
207         std::cout << x_ref_vec.at(i) << std::endl;
208     std::cout << "\n";
209     std::cout << "y references: " << std::endl;
210     for (int i = 0; i < y_ref_vec.size(); ++i)
211         std::cout << y_ref_vec.at(i) << std::endl;
212     std::cout << "\n";
213     std::cout << "z references: " << std::endl;
214     for (int i = 0; i < z_ref_vec.size(); ++i)
215         std::cout << z_ref_vec.at(i) << std::endl;
216     std::cout << "\n";
217 
218     /*** give some time to digest references ***/
219     int i = 1;
220     int wait_time = 2;
221     while(i < wait_time+1) {
222         std::cout << "starting in " << (wait_time-i+1) << " seconds.." << std::endl;
223         sleep(1);
224         i += 1;
225     }
226 
227     /*** start controller ***/
228     int iter = 0;
229     while(true) {
230         /*** subscribe to topics with 100 Hz ***/
231         ros::Rate r(100);
232 
233         /*** prepare real time processing ***/
234         timer t;
235         t.start();
236         double long Ts = 0.01;
237 
238         while(true) {
239             if (t.elapsedTime() >= Ts) {
240                 /*** read sensor ***/
241                 ros::spinOnce();
242 
243                 /*** convert to 3D ***/
244                 compute_fk_chain();
245 
246                 /*** start timer to meassure execution time ***/
247                 std::clock_t start;
248                 double dur;
249                 start = std::clock();
250 
251                 /*** update trajectory ***/
252                 if (iter_ref > N_iter) {
253                     xref_3d(0,0) = x_ref_vec.at(i_ref);
254                     xref_3d(1,0) = y_ref_vec.at(i_ref);
```

```
255                 xref_3d(2,0) = z_ref_vec.at(i_ref);
256                 i_ref += 1;
257                 iter_ref = 0;
258             }
259             iter_ref += 1;
260
261             /*** update state vector ***/
262             x(0,0) = x_cart_meas;
263             x(1,0) = y_cart_meas;
264             x(2,0) = z_cart_meas;
265
266             /*** control barrier function ***/
267             cbf(0,0) = -( pow(((x(0,0)-cx)/rx),2) + pow(((x(1,0)-cy)/ry),2) + pow(((x(2,0)-cz)/rz),2) - 1 );
268
269             /*** linear non-safe controller ***/
270             utilde_3d = Nbar*xref_3d - K*x;
271
272             /*** lie derivatives ***/
273             LgB_3d(0,0) = (2/(pow(rx,2)*taux))*(cx-x(0,0));
274             LgB_3d(0,1) = (2/(pow(ry,2)*tauy))*(cy-x(1,0));
275             LgB_3d(0,2) = (2/(pow(rz,2)*tauz))*(cz-x(2,0));
276             LfB_3d(0,0) = (-2/(pow(rx,2)*taux))*(cx*x(0,0)-pow(x(0,0),2)) + (-2/(pow(ry,2)*tauy))*(cy*x(1,0)-pow(x(1,0),2)) + (-2/(
                    pow(rz,2)*tauz))*(cz*x(2,0)-pow(x(2,0),2));
277
278             temp = LgB_3d*LgB_3d.transpose();
279             /*** safe control law ***/
280             if (LgB_3d.squaredNorm() > 0.000001) {
281                 k0 = -LgB_3d.transpose()*sqrt(pow(LfB_3d(0,0),2) + pow(kappa,2)*temp(0,0) ) /(temp(0,0));
282             }
283             else {
284                 k0(0,0) = 0;
285                 k0(1,0) = 0;
286                 k0(2,0) = 0;
287             }
288
289             /*** calculate sigma ***/
290             if ((cbf(0,0) < -epsilon) || (cbf(0,0) == -epsilon)) {
291                 sigma(0,0) = 0;
292             }
293             else if ((cbf(0,0) > -epsilon) && (cbf(0,0) < 0)) {
294                 sigma(0,0) = (-2*pow(cbf(0,0)/epsilon,3) - 3*pow(cbf(0,0)/epsilon,2) + 1);
295             }
296             else {
297                 sigma(0,0) = 1;
298             }
299
300             /*** give time to start in X0 ***/
301             if (iter < 60) {
302                 sigma(0,0) = 0;
303             }
304
305             /*** control law ***/
306             u_3d = sigma(0,0)*k0 + (1 - sigma(0,0))*utilde_3d;
307
308             /*** translation is displaced ***/
309             u_3d(0,0) = u_3d(0,0) + 0.482 + 0.01;
310             u_3d(1,0) = u_3d(1,0);
311             u_3d(2,0) = u_3d(2,0) - 0.059 + 0.01;
312
313             double x_kdl = u_3d(0,0);
314             double y_kdl = u_3d(1,0);
315             double z_kdl = u_3d(2,0);
316
317             KDL::Vector dest_pos(x_kdl,y_kdl,z_kdl);
```

```
318         KDL::Frame dest_frame(dest_pos);
319 
320         /*** Compute ***/
321         int ret = iksolver.CartToJnt(q_init,dest_frame,q);
322 
323         for (unsigned int i = 0; i < q.rows(); ++i)
324         {
325             std::cout << "Joint #" << i << ": " << q(i) << std::endl;
326         }
327 
328         std::vector<double> control_signals;
329 
330         /*** adjust instrument_roll ***/
331         while (q(3) > pi) {
332             q(3) = -(pi - (q(3) - pi));
333             std::cout << "instrument_roll adjusted to : " << q(3) << std::endl;
334         }
335         while (q(3) < -pi) {
336             q(3) = (pi + (q(3) + pi));
337             std::cout << "instrument_roll adjusted to : " << q(3) << std::endl;
338         }
339 
340         /*** adjust instrument_jaw_right ***/
341         while (q(5) > pi) {
342             q(5) = -(pi - (q(5) - pi));
343             std::cout << "instrument_jaw_right adjusted to : " << q(5) << std::endl;
344         }
345         while (q(5) < -pi) {
346             q(5) = (pi + (q(5) + pi));
347             std::cout << "instrument_jaw_right adjusted to : " << q(5) << std::endl;
348         }
349 
350         /*** adjust instrument_jaw_right ***/
351         if (q(5) > 0.1) {
352             q(5) = 0.1;
353             std::cout << "instrument_jaw_right > 0.1" << std::endl;
354         }
355         if (q(5) < -0.1) {
356             q(5) = -0.1;
357             std::cout << "instrument_jaw_right < -0.1" << std::endl;
358         }
359 
360         /*** physical limits for hand_roll ***/
361         if (q(0) > 1.5) {
362             q(0) = 1.5;
363         }
364         else if (q(0) < -1.5) {
365             q(0) = -1.5;
366         }
367         /*** physical limits for hand_pitch ***/
368         if (q(1) > 0.7) {
369             q(1) = 0.7;
370         }
371         else if (q(1) < -0.7) {
372             q(1) = -0.7;
373         }
374         /*** physical limits for instrument_slide ***/
375         if (q(2) > 0.097) {
376             q(2) = 0.097;
377         }
378         else if (q(2) < -0.097) {
379             q(2) = -0.097;
380         }
381         /*** physical limits for instrument_pitch ***/
```

```
382            if (q(4) > 0.8+1.65551) {
383                q(4) = 0.8+1.65551;
384            }
385            else if (q(4) < −0.8+1.65551) {
386                q(4) = −0.8+1.65551;
387            }
388
389            /*** collect all control signals in one vector ***/
390            control_signals.push_back(q(0)); // hand roll
391            control_signals.push_back(q(1)); // hand pitch
392            control_signals.push_back(q(2)); // instr slide
393            control_signals.push_back(q(3)); // inst roll
394            control_signals.push_back((q(4)−1.65551)); // inst pitch
395            control_signals.push_back(q(5)); // jaw right
396
397            /*** execute p4_hand_pith ***/
398            std_msgs::Float64 u_msg_0;
399            u_msg_0.data = control_signals[1];
400
401            /*** execute p4_hand_roll ***/
402            std_msgs::Float64 u_msg_1;
403            u_msg_1.data = control_signals[0];
404
405            /*** execute p4_hand_slide ***/
406            std_msgs::Float64 u_msg_2;
407            u_msg_2.data = control_signals[2];
408
409            /*** load p4_instrument_roll ***/
410            std_msgs::Float64 u_msg_3;
411            u_msg_3.data = control_signals[3];
412
413            /*** load p4_instrument_pitch ***/
414            std_msgs::Float64 u_msg_4;
415            u_msg_4.data = control_signals[4];
416
417            /*** load p4_instrument_jaw_right ***/
418            std_msgs::Float64 u_msg_5;
419            u_msg_5.data = control_signals[5];
420
421            /*** publish joint angles ***/
422            setpoints_pub_pitch.publish(u_msg_0);
423            setpoints_pub_roll.publish(u_msg_1);
424            setpoints_pub_slide.publish(u_msg_2);
425            setpoints_pub_inst_roll.publish(u_msg_3);
426            setpoints_pub_inst_pitch.publish(u_msg_4);
427            setpoints_pub_inst_jaw_right.publish(u_msg_5);
428
429            /*** check execution time ***/
430            dur = ( std::clock() − start ) / (double) CLOCKS_PER_SEC;
431            std::cout << "execution time = " << dur << std::endl;
432            std::cout << "sigma = " << sigma(0,0) << std::endl;
433            std::cout << "u = " << u_3d << std::endl;
434
435            /*** provide some user information ***/
436            iter += 1;
437            std::cout << iter << " iterations " << std::endl;
438            std::cout << "\n";
439
440            /*** write slide measurements to file after inital transients ***/
441            if ( iter > 20) {
442                /*** export trajectory ***/
443                x3d_x_vec.push_back(x(0,0));
444                x3d_y_vec.push_back(x(1,0));
445                x3d_z_vec.push_back(x(2,0));
```

```
446                             /*** export references ***/
447                             x3d_x_ref_vec.push_back(xref_3d(0,0));
448                             x3d_y_ref_vec.push_back(xref_3d(1,0));
449                             x3d_z_ref_vec.push_back(xref_3d(2,0));
450                             /*** export sigma ***/
451                             sigma3d_vec.push_back(sigma(0,0));
452                             /*** export execution time ***/
453                             exe_time_vec.push_back(dur);
454                         }
455
456                         if (iter > N_iter*x_ref_vec.size() + N_iter) {
457                             std::cout << "writing meas to file ... " << std::endl;
458                             write_meas_to_files_3d();
459                             ros::shutdown();
460                             return 0;
461                         }
462
463                         break;
464                 }
465             else {
466                 /*** do some other stuff if necessary***/
467             }
468         }
469 }
470 }
471
472 int compute_fk_chain() {
473         KDL::Tree my_tree;
474         ros::NodeHandle node_meas;
475         std::string robot_desc_string;
476         node_meas.param("robot_description", robot_desc_string, std::string());
477         if (!kdl_parser::treeFromString(robot_desc_string, my_tree)){
478             ROS_ERROR("Failed to construct kdl tree");
479         }
480
481         /*** get chain from reduced robot ***/
482         KDL::Chain my_chain;
483         std::string root_link("p4_rcm_base");
484         std::string tip_link("needle_driver_jawbone_right");
485         if (!(my_tree.getChain(root_link, tip_link , my_chain)))
486         {
487              ROS_ERROR("Failed to get chain");
488         }
489
490         /*** Create solver based on kinematic chain ***/
491         KDL::ChainFkSolverPos_recursive fksolver = KDL::ChainFkSolverPos_recursive(my_chain);
492
493         /*** Create joint array ***/
494         unsigned int nj = my_chain.getNrOfJoints();
495         KDL::JntArray jointpositions = KDL::JntArray(nj);
496          jointpositions (0) = x_hand_roll;
497          jointpositions (1) = x_hand_pitch;
498          jointpositions (2) = x_inst_slide;
499          jointpositions (3) = x_inst_roll;
500          jointpositions (4) = x_inst_pitch;
501          jointpositions (5) = x_jaw_right;
502
503         /*** Create the frame that will contain the results ***/
504         KDL::Frame cartpos;
505
506         /*** Calculate forward position kinematics ***/
507         bool kinematics_status;
508         kinematics_status = fksolver.JntToCart(jointpositions,cartpos);
509          if (kinematics_status >= 0) {
```

```
510                 printf ("%s \n","FK calculated succesfully") ;
511             }
512             else {
513                 printf ("%s \n","Error: could not calculate FK");
514             }
515 
516             /*** subtract translation ***/
517             x_cart_meas = cartpos(0,3) − 0.482;
518             y_cart_meas = cartpos(1,3) + 0.00;
519             z_cart_meas = cartpos(2,3) + 0.059;
520 
521         return 0;
522 }
523 
524 int write_meas_to_files_3d() {
525     std::cout << "writing measurements to files.." << std::endl << std::endl;
526     /*** print x position trajectory to file ***/
527     std::ofstream f1;
528     f1.open("x.txt", std::ios::out | std::ios::app | std::ios::binary);
529     for (int i = 0; i < x3d_x_vec.size(); i += 1) {
530         f1 << x3d_x_vec[i] << std::endl;
531     }
532     f1.close();
533     /*** print y position trajectory to file ***/
534     std::ofstream f2;
535     f2.open("y.txt", std::ios::out | std::ios::app | std::ios::binary);
536     for (int i = 0; i < x3d_y_vec.size(); i += 1) {
537         f2 << x3d_y_vec[i] << std::endl;
538     }
539     f2.close();
540     /*** print z position trajectory to file ***/
541     std::ofstream f3;
542     f3.open("z.txt", std::ios::out | std::ios::app | std::ios::binary);
543     for (int i = 0; i < x3d_z_vec.size(); i += 1) {
544         f3 << x3d_z_vec[i] << std::endl;
545     }
546     f3.close();
547     /*** print x position reference to file ***/
548     std::ofstream f4;
549     f4.open("x_ref.txt", std::ios::out | std::ios::app | std::ios::binary);
550     for (int i = 0; i < x3d_x_ref_vec.size(); i += 1) {
551         f4 << x3d_x_ref_vec[i] << std::endl;
552     }
553     f4.close();
554     /*** print y position reference to file ***/
555     std::ofstream f5;
556     f5.open("y_ref.txt", std::ios::out | std::ios::app | std::ios::binary);
557     for (int i = 0; i < x3d_y_ref_vec.size(); i += 1) {
558         f5 << x3d_y_ref_vec[i] << std::endl;
559     }
560     f5.close();
561     /*** print z position reference to file ***/
562     std::ofstream f6;
563     f6.open("z_ref.txt", std::ios::out | std::ios::app | std::ios::binary);
564     for (int i = 0; i < x3d_z_ref_vec.size(); i += 1) {
565         f6 << x3d_z_ref_vec[i] << std::endl;
566     }
567     f6.close();
568     /*** print sigma to file ***/
569     std::ofstream f7;
570     f7.open("sigma_3d.txt", std::ios::out | std::ios::app | std::ios::binary);
571     for (int i = 0; i < sigma3d_vec.size(); i += 1) {
572         f7 << sigma3d_vec[i] << std::endl;
573     }
```

```
574     f7.close();
575     /*** print execution time to file ***/
576     std::ofstream f8;
577     f8.open("exe_3d.txt", std::ios::out | std::ios::app | std::ios::binary);
578     for (int i = 0; i < exe_time_vec.size(); i += 1) {
579         f8 << exe_time_vec[i] << std::endl;
580     }
581     f8.close();
582
583 return 0;
584 }
```

Appendix I

Developed Auxiliary Files

This appendix contains auxiliary files developed such that the ROS framework is initialized correctly. All packages mentioned in this appendix can be found at GitHub - Robotic Surgery Group - Aalborg University (*https://github.com/AalborgUniversity-RoboticSurgeryGroup*). It includes development of:

- `CMakeLists.txt` (Founds Cmake build system for package `gr1032`)
- `package.xml` (States dependencies for package `gr1032`)
- `run_controllers.py` (wrapper script to run all developed controllers)
- `p4_hand_controller.yaml` (allows direct propagation of the control signal to the joints)
- `davinci_driver.launch` (spawn controllers such the control signal can be passed to the joints)
- `remote_center_manipulator.xacro` (implementation of the kinematics for the hand and instrument slide. A similar one is present for the remaining parts of the instrument, though not shown here).

CMakeLists.txt

```
1  cmake_minimum_required(VERSION 2.8.3)
2  project(gr1032)
3
4  find_package(catkin REQUIRED COMPONENTS
5    roscpp kdl_parser
6  )
7
8  catkin_package(
9  )
10
11 include_directories(
12   ${catkin_INCLUDE_DIRS}
13 )
14
15 add_library(ik_gr1032
16     src/${gr1032}/ik_gr1032
17 )
18 add_library(demo_gr1032
19     src/${gr1032}/demo_gr1032
20 )
21 add_library(safe_3d
22     src/${gr1032}/safe_3d
23 )
24
25 add_executable(run_controllers src/run_controllers.cpp)
26
27 target_link_libraries (run_controllers
28                        ik_gr1032
```

```
29                            demo_gr1032
30                            safe_3d
31   ${catkin_LIBRARIES}
32 )
```

package.xml

```
1  <?xml version="1.0"?>
2  <package>
3    <name>gr1032</name>
4    <version>0.0.0</version>
5    <description>gr1032 Master Thesis</description>
6    <maintainer email="15gr1032@es.aau.dk">gr1032</maintainer>
7    <license>GPLv3</license>
8    <author email="15gr1032@es.aau.dk">gr1032</author>
9    <buildtool_depend>catkin</buildtool_depend>
10   <build_depend>roscpp</build_depend>
11   <build_depend>kdl_parser</build_depend>
12   <run_depend>roscpp</run_depend>
13   <run_depend>kdl_parser</run_depend>
14   <export>
15   </export>
16 </package>
```

run_controllers.py

```
1   1 import os
2   2 os.system("rosrun gr1032 run_controllers /slide_command:=/davinci/slide_position_controller/c    ommand /pitch_command:=/davinci/
        hand_pitch_position_controller/command /roll_command:=/davinc i/hand_roll_position_controller/command /inst_roll_command:=/
        davinci/instrument_roll_position _controller/command /inst_pitch_command:=/davinci/instrument_pitch_position_controller/comman d /
        inst_jaw_right_command:=/davinci/instrument_jaw_right_position_controller/command")
```

p4_hand_controller.yaml

```
1  davinci:
2      # Publish all joint states -----------------------------------
3      joint_state_controller :
4          type: joint_state_controller / JointStateController
5          publish_rate: 100
6
7   # Position Controllers ---------------------------------------
8      slide_position_controller :
9          type: position_controllers / JointPositionController
10         joint : p4_instrument_slide
11         publish_rate: 100
12     hand_pitch_position_controller:
13         type: position_controllers / JointPositionController
14         joint : p4_hand_pitch
15         publish_rate: 100
16     hand_roll_position_controller :
17         type: position_controllers / JointPositionController
18         joint : p4_hand_roll
19         publish_rate: 100
20     instrument_roll_position_controller :
21         type: position_controllers / JointPositionController
22         joint : p4_instrument_roll
23         publish_rate: 100
24     instrument_pitch_position_controller :
25         type: position_controllers / JointPositionController
26         joint : p4_instrument_pitch
```

```
            publish_rate: 100
        instrument_jaw_right_position_controller:
            type: position_controllers / JointPositionController
            joint : p4_instrument_jaw_right
            publish_rate: 100
```

davinci_driver.launch

```
<!--
Launch file for the driver.

Karl D. Hansen (kdh@es.aau.dk)
modified by: Gr1032 in 2015
-->

<launch>
    <!--
    If not using default ips, remember to load them
    into the param server elsewhere.
    -->
    <arg name="default_ips" default="true" />

    <!--
    Load the IPs of the davinci embedded controllers
    into the parameter server as a list.
    -->
    <rosparam
        if="$(arg default_ips)"
        command="load"
        file="$(find davinci_driver)/config/davinci_ip_adresses.yaml"
    />

    <node name="davinci_driver_node" pkg="davinci_driver" type="davinci_driver_node" output="screen" ns="davinci">
        <remap from="/davinci/joint_states" to="/joint_states" />
    </node>

    <!-- Load the robot description into the param server. -->
    <param name="robot_description"
        command="$(find xacro)/xacro.py '$(find davinci_description)/robots/davinci_description.xacro'" />

    <!-- Combine joint values into the tf tree based on the above model. -->
    <node name="robot_state_publisher" pkg="robot_state_publisher" type="state_publisher" />

    <!-- Load joint controller configurations from YAML file to parameter server -->
    <rosparam
        command="load"
        file="$(find davinci_driver)/config/p4_hand_controller.yaml"
    />
    <!-- load the controllers -->
    <node
        name="controller_spawner_joint_state_controller"
        pkg="controller_manager"
        type="spawner"
        respawn="false"
        output="screen"
        ns="davinci"
        args="joint_state_controller"
    />
    <node
        name="controller_spawner_position_controller_slide"
        pkg="controller_manager"
        type="spawner"
        respawn="false"
```

```
            output="screen"
            ns="davinci"
            args=" slide_position_controller "
        />
        <node
            name="controller_spawner_position_controller_hand_pitch"
            pkg="controller_manager"
            type="spawner"
            respawn="false"
            output="screen"
            ns="davinci"
            args="hand_pitch_position_controller"
        />
        <node
            name="controller_spawner_position_controller_hand_roll"
            pkg="controller_manager"
            type="spawner"
            respawn="false"
            output="screen"
            ns="davinci"
            args="hand_roll_position_controller"
        />
        <node
            name="controller_spawner_position_controller_instrument_roll"
            pkg="controller_manager"
            type="spawner"
            respawn="false"
            output="screen"
            ns="davinci"
            args=" instrument_roll_position_controller "
        />
        <node
            name="controller_spawner_position_controller_instrument_pitch"
            pkg="controller_manager"
            type="spawner"
            respawn="false"
            output="screen"
            ns="davinci"
            args="instrument_pitch_position_controller"
        />
        <node
            name="controller_spawner_position_controller_instrument_jaw_right"
            pkg="controller_manager"
            type="spawner"
            respawn="false"
            output="screen"
            ns="davinci"
            args="instrument_jaw_right_position_controller"
        />
</launch>
```

remote_center_manipulator.xacro

```
<?xml version="1.0"?>
<robot name="davinci" xmlns:xacro="http://www.ros.org/wiki/xacro">

  <xacro:include filename="$(find davinci_description)/robots/davinci_link_macro.xacro" />

  <xacro:davinci_link name="p4_rcm_base" color="slate" />

  <joint name="p4_hand_roll"
    type="revolute">
    <origin
```

```
11         xyz="0.482 0 0.047"
12         rpy="0 1.5708 0" />
13       <parent
14         link="p4_rcm_base" />
15       <child
16         link="rcm_vitual0" />
17       <axis
18         xyz="0 0 1" />
19       <limit
20         lower="-1.5708"
21         upper="1.5708"
22         effort="1"
23         velocity="1" />
24     </joint>
25 
26     <link name="rcm_vitual0" />
27 
28     <joint name="p4_hand_pitch"
29       type="revolute">
30       <origin
31         xyz="0 0 0"
32         rpy="1.5708 0 0" />
33       <parent
34         link="rcm_vitual0" />
35       <child
36         link="rcm_vitual1" />
37       <axis
38         xyz="0 0 1" />
39       <limit
40         lower="-0.8"
41         upper="1.0"
42         effort="1"
43         velocity="1" />
44     </joint>
45 
46     <link name="rcm_vitual1" />
47 
48     <joint name="p4_instrument_slide"
49       type="prismatic">
50       <origin
51         xyz="0.097 0 0"
52         rpy="0 -1.5708 0" />
53       <parent
54         link="rcm_vitual1" />
55       <child
56         link="needle_driver_house" />
57       <axis
58         xyz="0 0 1" />
59       <limit
60         lower="-0.12"
61         upper="0.12"
62         effort="1"
63         velocity="1" />
64     </joint>
65 
66 </robot>
```

Appendix J

Attached CD

- **Digital Copy of this Thesis**
- **Literature**
- **MATLAB scripts**
- **Measurement Files**
- **Source Code**